Parallelize Accelerated Triangle Counting Using Bit-Wise on GPU

Li Lin, Dian Ouyang(✉), Zhipeng He, and Chengqian Li

School of Computer Science and Network Engineering, Guangzhou University, Guangzhou, China
dian.ouyang@gzhu.edu.cn

Abstract. Triangle counting is a graph algorithm that calculates the number of triangles in a graph, the number of triangles is a key metric for a large number of graph algorithms. Traditional triangle counting algorithms are divided into vertex-iterator and edge-iterator when traversing the graph. As the scale of graph data grows, the use of CPU with other architectural platforms for triangle counting has become mainstream. Our accelerating method proposes an algorithm for triangle counting on a single machine GPU, and performs a two-dimensional partition algorithm for large-scale graph data, in order to ensure that large graph data can be correctly loaded into GPU memory and the independence of each partition to obtain the right result. According to the high concurrency of GPU, a bit-wise operation intersection algorithm is proposed. We experiment with our algorithm to verify that our method effectively speeds up triangle counting algorithm on a single-machine GPU.

Keywords: Triangle Counting · GPU · Bit-Wise

1 Introduction

Triangle counting algorithm is the basic algorithm of other graph algorithms, such as K-Truss [1], Clustering Coefficient [2]. Traditional triangle counting algorithm can be divided into vertex-iterator and edge-iterator [3]. Vertex-iterator is to detect the presence of an edge between all neighbors of each vertex. Edge-iterator iterates through each edge in the graph to find the common neighbor vertices of the neighbor lists of two points. Depending on how to intersect lists, there are algorithms including merge-based, binary-search hashing-based and bitmap [3] algorithms, where bitmap is a special form of hashing.

CPU have very powerful and complex computing units and control capabilities [4], many works have proposed multicore CPUs as well as heterogeneous triangle counting algorithms, including multi-core CPUs [5], as well as external memory devices [6], and in-memory devices [7]. Our work focuses on the implementation of triangle counting algorithms on GPU. The existing GPU-implemented algorithm has achieved great results. [8] uses the basic triangle counting algorithm of the GPU. [9] allocates the number of threads according to the workload of each edge. [3] proposes the strategy of using bitmap as a list for fast lookup for processing triangle counting, [10] proposed a triangle counting system tricore, a distributed GPU processing strategy, [11] proposed TRUST, a triangle counting method based on hashing and vertex-centric using distributed GPUs, [12] proposed the most basic parallel strategy using one thread to process one edge. [13] based on map-reduce, [9] used binary search, [12] used the merge-based intersection algorithm, [10] proves that the intersection algorithm based on binary search is superior to the merge-based algorithm on the GPU architecture. [14] uses a similar triangle counting method like [10], besides with a system of a single external memory. [15] instead of proposing a new triangle counting algorithm, but designed a preprocessing strategy before calculating triangles.

Unlike previous lists intersection algorithm to perform triangle counting algorithm based on GPU, we propose a triangle counting algorithm on a GPU-based heterogeneous platform, which uses bit-wise on GPU, and adopts a two-dimensional partition algorithm for large-scale graphs. As well as implementing a preprocessing strategy to optimize the direction of edges between vertices, the purpose is to homogenize the balance of workload between thread blocks.

We experimented with our algorithm and others on a single GPU RTX A6000 based running platform, and for different graphs, our triangle counting algorithm is 1.3× to 21.3× faster than the-state-of-the-art binary-search intersection algorithm implemented on GPU [10] and 1.1× to 1.8× faster than merge-based intersection algorithm implemented on GPU [12].

Generally, the contributions of this paper can be summarized as follows.

1) This project is an algorithm for triangle counting on GPU-based heterogeneous platforms. After partition of a large-scale graph, handling the triangle counting subtask of each independent partition, we propose a new triangle counting algorithm of bit-wise intersection algorithm implemented on GPU.
2) Real-world network graphs are very sparse, their vertex distribution is skewed degree distribution. Therefore, we implement a way to change the direction of each edge during preprocessing to solve the problem of workload imbalance of each thread in warps on GPU, avoid thread diversity on GPU.
3) In order to efficiently perform bit-wise operation, we optimize the distribution of values for each row in the compressed sparse row (CSR). In addition, the data is reasonably stored in GPU memory, which reduces frequent data interaction between CPU main memory and GPU.

2 Background

In this section, we analyze the processing task on GPU, and the graph format commonly used and previous work on the triangle counting algorithm.

2.1 GPU Architecture

The threads on GPU are executed according to the SIMD (Single Instruction Multiple Data) style. Each thread on GPU processes an edge, and the adjacency table length of each vertex determines the workload of the threads. The running time of a warp is the longest running time of the thread, if the distribution of adjacency tables of the vertices in the graph is extremely unbalanced, it will lead to an imbalance in thread workload.

2.2 Graph Format

CSR is a mainstream graph data compression format we adopt for data compression, suppose each element in adjacency list is represented as $A[i]$ and each element in begin position array is represented as $B[j]$. The adjacency list stores all edges for each vertex, and the begin position array stores the location of the first element of each vertex adjacency list. That is, the neighbor vertices of vertex v starts at $A[B[v]]$ and ends at $A[B[v+1]]-1$.

2.3 Existing Solution

Our method is BFS-based edge-parallelism-iterator, and it's intersection algorithm is related to the intersection algorithm implemented by matrix-multiplication method, assuming that matrix A represents the adjacency matrix of the undirected graph $G(V, E)$, the value of $A^2[i][j]$ represents the vertex i, j has how many paths of length 2-hop. When $A[i][j] = 1$ and $A^2[i][j] = 1$, it means i, j have both 1-hop path and 2-hop paths to form a triangle. Our method uses bit-wise to optimize matrix-multiplication method, it avoids excessive spatial overhead of the latter and simplifies computational complexity.

3 Method

In this section, our goal is to use bit-wise to execute the triangle counting algorithm on the GPU. Figure 1 shows overview of our method.

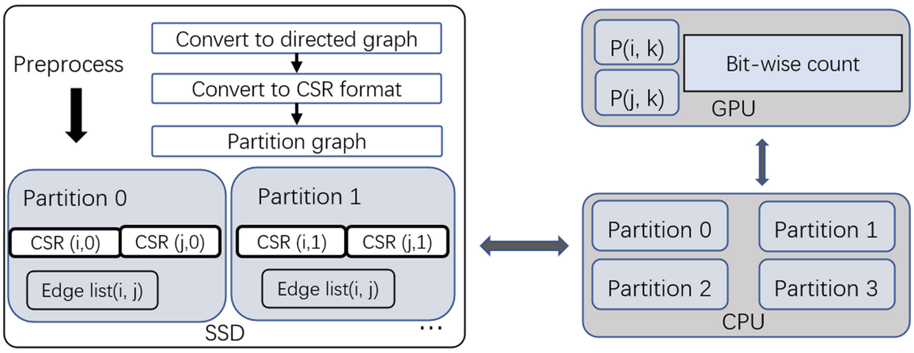

Fig. 1. Bit-wise triangle counting overview.

3.1 Preprocessing

Duplicate edges and looped edges will be removed, and the edge data structure of the original undirected graph is converted into the edge list representation of the directed graph. We use the CSR data format to save all vertex information, and the amount of space used occupies $|V| + |E|$, where $|V|$ is the number of vertices in the graph, $|E|$ is the number of edges in the graph. In the process of converting to a directed graph, it is common practice to use the rank-by-degree strategy to set the direction of the edges, but this usually brings about the imbalance of thread workload.

Therefore, we implemented a preprocessing strategy to balance out-degree of vertices. We first collect the degree $d(u)$ of each vertex u and calculate the average degree $\overline{d} = (|E|)/(|V|)$, when $d(u) \geq \overline{d}$, we treat vertex u as a center vertex, denoted by v_c, otherwise a non-center vertex, denoted by v_n. When two vertices of an edge are v_c, connected to v_n, we set the direction of the edge is $v_n \rightarrow v_c$, when both two vertices of an edge are v_c, or both are v_n, the pointing of the edge can be set arbitrarily. In this way, the degree of v_c is reduced and the degree of v_n is increased. When the two connected vertices are v_c, or both are v_n, the direction of the edge can be set arbitrarily, because changing either of the two will result in the same. Detailed proof is in [15].

3.2 Graph Partition

To avoid running out of memory, the graph data must be partitioned. First, we use a 2-dimensional graph partition algorithm to divide the task of triangle counting into multiple subtasks, each subtask needs to have two corresponding CSR partitions data, and an edge list partition provides data support. Figure 2 shows the process of creating a 2-dimension CSR and 2-dimension edge list.

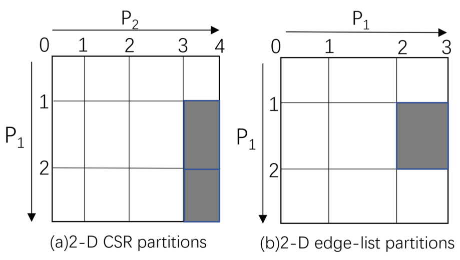

Fig. 2. Graph partition.

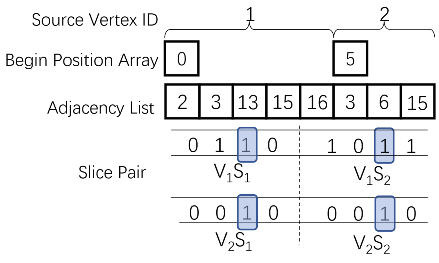

Fig. 3. Bit-wise intersection.

The specific step is to first make a 1-dimensional vertical partition to get P_2, so that each P_2 partition has the same size as possible. Then take the same strategy, make a horizontal partition, take the smallest horizontal partition size as the size of the horizontal vertex range partition, and get partition P_1. Figure 2(a) shows the size of the CSR after partitioning as $P_1 \times P_2$, Fig. 2(b) is the corresponding edge list partition as $P_1 \times P_1$. For example, edge list partition is (i, j), we need CSR partitions (i, k) and (j, k), the value of k is 0 to 3, i = 1, j = 2, that means we need to take the CSR partitions data 4 times to obtain the triangles in edge list partition (1, 2).

3.3 Bit-Wise Triangle Counting Algorithm

Adjacency matrix $A[i][j]$ refers to whether vertex i, j is connected by an edge, and $A^2[i][j]$ indicates that vertex i, j has a connection of 2-hop path. And the number of triangles in graph G is the number of non-zero elements in matrix A AND matrix A^2. As the values of $A[i][j]$ can only be 1 or 0, the operation of matrix multiplication can be converted to bit-wise operation of AND.

To get the bit slices, for edge (u, v), we will get their corresponding neighbor lists *u_list*, *v_list*, respectively, then each time get m(slice fixed size) values $t \in [k, k+m-1]$ of *u_list* and *v_list*. If *u_list*$[t] = 1$, we set the corresponding bit of a slice to 1. After the data is read, the algorithm of intersecting the neighboring vertices of two vertices is calculated by parallel bit-wise operation to calculate the number of triangles. For the retrieved CSR partition data, we convert the data into bit slices data with a fixed slice size. The pseudocode is shown in the Algorithm 1, the neighbor nodes of vertex u are from the *col* array and *row* array of CSR, and using bit-wise to intersect the k-th slice of vertex u and v. The INTERSECT phase is shown in the Fig. 3, assuming that m = 4, and the *jth* slice of each vertex i is represented as V_iS_j.

Algorithm 1. Triangle counting with bit-wise

```
1: Input : Graph CSR row[], col[], edgeId
2: Output : The number of triangles in Graph
3: TCnum=0
4: upos=col[edgeId]
5: u=row[upos]
6: adjList[u]=col[u+1]-col[u]
7: foreach v ∈ adjList[u] do
8:     foreach slice pair ($\mathbf{V_uS_k}$, $\mathbf{V_vS_k}$) do
9:         TCnum+=INTERSECT($\mathbf{V_uS_k}$, $\mathbf{V_vS_k}$)
10:    end for
11: end for
12: return TCnum
```

Vertex 1 is connected to vertex 2, requiring the number of triangles composed by edge (1, 2), starting from the index position $idx = 0$, corresponding vertex is 2, starting to fill $V_1S_1 = 0110$ with slice size 4, and construct the corresponding slice $V_2S_1 = 0010$ of vertex 2, and perform AND bit operation on the two slices, and the result is 1. Consecutive values of 0 at vertex 1 are skipped. At the next time, $idx = 3$, corresponding vertex is 13, fill the slice $V_1S_2 = 1011$ of vertex 1, and the slice $V_2S_2 = 0010$ of vertex 2, and the AND bit operation is also performed, and the result is 1, the total number of triangle is 2.

Although bit operations are performed on a row and column of the CSR, the CSR is stored in the GPU memory, frequent data interaction between the CPU main memory and the GPU is reduced, and the use of storage space in the GPU is not particularly high, the cores of the GPU can be fully utilized.

4 Evaluation

In this section, we will introduce the implemented experimental environment and the graph datasets and then evaluate the results of the experiment.

4.1 Experimental Environment

We implemented the method on GPU RTX A6000, Intel(R) Xeon(R) Silver 4216 CPU, CUDA toolkit is 7.0, including nvcc, with gcc version 9.3.0 and the operating system was ubuntu20.04, we set the compilation flag to -$O2$.

The datasets we use are from Stanford Network Analysis Project, they are all real-world social network graphs data. All of them will first be transformed from the original data format in order to fit our method.

Table 1. Running Time of Different Triangle Counting Methods on GPU (in seconds).

Dataset	Size	Nodes	Edges	Triangles	Tricore	Adam	Tcbw (CPU)	Tcbw (GPU)
roadNet-PA	44M	1.04M	1.47M	65.57K	0.23	**0.18**	1.64	**0.18**
roadNet-CA	84M	1.87M	2.64M	0.12M	**0.19**	0.27	5	0.25
com-Amazon	13M	0.32M	0.88M	0.64M	0.27	0.24	0.33	**0.14**
facebook-cb	836K	4K	0.08M	1.54M	0.26	0.21	**0.05**	0.12
com-DBLP	14M	0.3M	1M	2.12M	0.27	0.18	0.39	**0.14**
com-Youtube	37M	1.08M	2.85M	2.91M	0.41	0.25	1.38	**0.18**
cit-Patents	268M	3.6M	15.75M	7.17M	0.97	**0.47**	35.5	0.65
as-Skitter	143M	1.62M	10.58M	27.44M	0.74	0.41	7.59	**0.37**
com-lj	479M	3.81M	33.07M	169.6M	0.91	**0.83**	63.6	1.07
twitter-cb	43M	79K	1.69M	12.48M	3.4	0.2	0.68	**0.16**
com-orkut	3.2G	2.93M	0.11G	0.58G	11.65	**2.47**	316	5.56
com-Friendster	31G	62.57M	1.68G	3.88G	**46.84**	*	-	262.07

** means that can not run successfully on the current device.*
- means that result is more than 30 min.

4.2 Evaluation of Experimental Results

We focus on the comparison of triangle counting algorithms on GPU, we compare the following algorithms implemented on GPU:

Tricore [10] is the state-of-the-art triangle counting using binary search based GPU.

Adam et al. [12] is the state-of-the-art triangle counting using merge-based on GPU.

From Table 1, the result shows the running time of different GPU triangle counting algorithms, we can know Adam et al. uses vertex sorting by the size of vertex degree. However, let the small degree vertex point to the larger degree vertex is not suitable for the graph of skewed degree distribution. The preprocessing time is proportional to the number of sides, and the time to calculate the number of triangles is proportional to the number of triangles. Adam et al. does not have the ability to handle large-scale graph and does not design partitioning algorithms. Tricore and our method apply partitioning algorithms that doubles the size of the graph, but has the ability to handle large-scale graphs. Tricore doesn't designed vertex average degree algorithms, which will lead to a greater impact on the speed of graph running for vertex degree distributions that vary greatly, such as com-Amazon, facebook-combined, twitter-combined, our algorithm is nearly 1.3× to 21.3× faster than Tricore. On large graph, Tricore is better than our method, because the slices for bit-wise is regenerated each time, and that's what we're working on in future, Adam et al. performs best at some small graphs, because it uses a simple graph format and does not implement the graph partition algorithm, but our method still has efficiency gains on some graphs, 1.1× to 1.8× faster than Adam et al. And we also implement our method on CPU, using Intel(R) Xeon(R) Silver 4216 CPU, which shows in the column of Tcbw (CPU). Our GPU method is almost 4.3× to 59.4× faster than the CPU method, the latter does not have the consumption of data transfer between the GPU and the CPU, it has an advantage in graphs with high cache hit ratios like facebook-combined, but the degree of parallelism is not as high as the former. The default running time is measured in seconds.

5 Conclusion

From the above experimental results and analysis, we propose a method that can calculate the number of triangles correctly, and accelerate the algorithm computational efficiency. We implement a preprocessing algorithm that optimizes threads balancing on GPU and partitioning for large-scale graph data to avoid running out of GPU memory. The results show that our proposed method consumes fewer running time than existing work on skewed degree distribution graphs, and that optimized the efficiency of triangle counting algorithm.

Acknowledgments. The authors are very grateful to the anonymous reviewers for their valuable comments.

References

1. Chen, P.-L., Chou, C.-K., Chen, M.-S.: Distributed algorithms for k-truss decomposition. In: 2014 IEEE International Conference on Big Data (Big Data), pp. 471–480. IEEE (2014)

2. Li, X., Chang, L., Zheng, K., Huang, Z., Zhou, X.: Ranking weighted clustering coefficient in large dynamic graphs. World Wide Web **20**, 855–883 (2017). https://doi.org/10.1007/s11280-016-0420-2
3. Bisson, M., Fatica, M.: High performance exact triangle counting on GPUs. IEEE Trans. Parallel Distrib. Syst. **28**(12), 3501–3510 (2017)
4. Qi, X., Wang, M., Wen, Y., Zhang, H., Yuan, X.: Weighted cost model for optimized query processing. In: Zhao, X., Yang, S., Wang, X., Li, J. (eds.) WISA 2022. LNCS, vol. 13579, pp. 473–484. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-20309-1_42
5. Shun, J., Tangwongsan, K.: Multicore triangle computations without tuning. In: 2015 IEEE 31st International Conference on Data Engineering, pp. 149–160. IEEE (2015)
6. Giechaskiel, I., Panagopoulos, G., Yoneki, E.: PDTL: parallel and distributed triangle listing for massive graphs. In: 2015 44th International Conference on Parallel Processing, pp. 370–379. IEEE (2015)
7. Wang, X., et al.: Triangle counting accelerations: from algorithm to in-memory computing architecture. IEEE Trans. Comput. **71**(10), 2462–2472 (2021)
8. Wang, L., Wang, Y., Yang, C., Owens, J.D.: A comparative study on exact triangle counting algorithms on the GPU. In: Proceedings of the ACM Workshop on High Performance Graph Processing, pp. 1–8 (2016)
9. Green, O., et al.: Logarithmic radix binning and vectorized triangle counting. In: 2018 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7. IEEE (2018)
10. Hu, Y., Liu, H., Huang, H.H.: TriCore: parallel triangle counting on GPUs. In: SC 2018: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 171–182. IEEE (2018)
11. Pandey, S., et al.: TRUST: triangle counting reloaded on GPUs. IEEE Trans. Parallel Distrib. Syst. **32**(11), 2646–2660 (2021)
12. Polak, A.: Counting triangles in large graphs on GPU. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 740–746. IEEE (2016)
13. Kolda, T.G., Pinar, A., Plantenga, T., Seshadhri, C., Task, C.: Counting triangles in massive graphs with MapReduce. SIAM J. Sci. Comput. **36**(5), S48–S77 (2014)
14. Huang, J., Wang, H., Fei, X., Wang, X., Chen, W.: TC-Stream: large-scale graph triangle counting on a single machine using GPUs. IEEE Trans. Parallel Distrib. Syst. **33**(11), 3067–3078 (2022)
15. Hu, L., Zou, L., Liu, Y.: Accelerating triangle counting on GPU. In: Li, G., Li, Z., Idreos, S., Srivastava, D. (eds.) SIGMOD 2021: International Conference on Management of Data, Virtual Event, China, 20–25 June 2021, pp. 736–748. ACM (2021)