Title
A Comparative Study on Exact Triangle Counting Algorithms on the GPU

Permalink
https://escholarship.org/uc/item/9hf0m6w3

Authors
Wang, Leyuan
Wang, Yangzihao
Yang, Carl
et al.

Publication Date
2016-04-01

DOI
10.1145/2915516.2915521

Peer reviewed
A Comparative Study on Exact Triangle Counting Algorithms on the GPU

Leyuan Wang
University of California, Davis
leywang@ucdavis.edu

Yangzhao Wang
University of California, Davis
yzhwang@ucdavis.edu

Carl Yang
University of California, Davis
cctcyang@ucdavis.edu

John D. Owens
University of California, Davis
jowens@ece.ucdavis.edu

ABSTRACT

We implement exact triangle counting in graphs on the GPU using three different methodologies: subgraph matching to a triangle pattern; programmable graph analytics, with a set-intersection approach; and a matrix formulation based on sparse matrix-matrix multiplies. All three deliver best-of-class performance over CPU implementations and over comparable GPU implementations, with the graph-analytic approach achieving the best performance due to its ability to exploit efficient filtering steps to remove unnecessary work and its high-performance set-intersection core.

CCS Concepts

• Theory of computation → Graph algorithms analysis; Parallel algorithms; • Computer systems organization → Single instruction, multiple data;

1. INTRODUCTION

In recent years, graphs have been used to model interactions between entities in a broad spectrum of applications. Graphs can represent relationships in social media, the World Wide Web, biological and genetic interactions, co-author networks, citations, etc. Understanding the underlying structure of these graphs is becoming increasingly important, and one of the key techniques for understanding is based on finding small subgraph patterns.

The most important such subgraph is the triangle. Many important measures of a graph are triangle-based, such as the clustering coefficient and the transitivity ratio. The clustering coefficient is frequently used in measuring the tendency of nodes to cluster together as well as how much a graph resembles a small-world network [30]. The transitivity ratio is the probability of wedges (three connected nodes) forming a triangle.

A side product obtained from many triangle counting algorithms is triangle enumeration. Instead of solving the

1. We survey triangle counting algorithms that are viable for GPUs and apply a state-of-the-art subgraph matching algorithm to triangle counting based on a filtering-and-joining strategy, achieving a speedup of 9–260× over a sequential CPU implementation.

2. We develop a best-of-class intersection operator and integrate it into our proposed set-intersection-based method that incorporates several optimizations yielding a speedup as high as 630× over the sequential CPU implementation and 5.1× over the previous state-of-the-art GPU exact triangle counting implementation.

3. We formulate a parallel matrix-multiplication-based method on the GPU that achieves 5×–26× speedup over the sequential CPU implementation.

4. We provide detailed experimental evaluations of our proposed algorithms by comparing with theoretical bounds on real-world network data as well as existing best-of-class GPU and CPU implementations.
2. RELATED WORKS

We investigate three approaches—subgraph matching, set intersection, and matrix multiplication methods—to implement exact triangle counting.

Subgraph Matching Algorithms.

Many state-of-the-art subgraph matching algorithms use a backtracking strategy. The first method that leverages this technique is proposed by Ullmann [26]. His basic idea is to incrementally compose partial solutions and discard the completed queries. Later, VF2 [6], QuickSI [21], GraphQL [12], GADDI [31], and SPath [32] have been proposed as improvements to Ullmann’s method. These algorithms exploit different filtering rules, joining orders, and auxiliary information to improve performance. Lee et al. [15] give an in-depth comparison of the above algorithms. For large-scale graphs, Sun et al. [23] have proposed a parallel method to decompose the query graphs into STwigs (two-level trees), and perform graph exploration and join strategies in a distributed memory cloud. More recently, Tran et al. [24] propose a state-of-the-art parallel method using GPUs to both take advantage of the computation power and avoid high communication costs between distributed machines. GPUs with massively parallel processing architectures have been successfully leveraged for fundamental graph operations. Traditional backtracking approaches for subgraph matching, however, cannot efficiently be adapted to GPUs due to two problems. First, GPU operations are based on warps (which are groups of threads to be executed in single-instruction-multiple-data fashion), so different execution paths generated by backtracking algorithms may cause a warp divergence problem. Second, irregular memory access patterns cause memory accesses to become uncoalesced, which degrades GPU performance. Our algorithm is an optimized and simplified version of the method of Tran et al. [24].

Set Intersection Algorithms.

Set intersection of two sorted arrays on the GPU is a well-studied research problem [1, 11]. Previous research mainly focuses on computing only one intersection set for two large sorted arrays. Green et al.’s GPU triangle counting algorithm [11] does set intersection computation for each edge in the undirected graph, and uses a modified intersection path method based on merge path [9], which is considered to have the highest performance for large-sorted-array set intersection on the GPU. Our implementation also includes a merge-path-based strategy when doing set intersection for large neighbor lists, and several other optimizations.

Triangle Counting using Matrix Multiplication.

The triangle counting algorithm using matrix multiplication we implement in this paper is by Azad, Buluç, and Gilbert [2]. It is based on traditional graph theoretic formulations of triangle counting that compute intersections between neighbor lists. These serial algorithms have been parallelized using the MapReduce framework [4]. If matrix multiplication is to be used to perform triangle counting, it is natural to consider some recent GPU advances in sparse matrix multiplication.

Matrix Multiplication Algorithms.

In recent years, some advances have been made in GPU-based SpGEMM algorithms. On GPUs, the standard SpGEMM used for comparison is the NVIDIA cuSPARSE library. Liu and Vinter have designed a fast SpGEMM with a upper bound on nonzero and fast parallel inserts [17]. Dalton et al. have developed an SpGEMM algorithm called expansion, sorting, and compression (ESC) [7]. This algorithm first expands possible nonzero elements into an intermediary matrix, then sorts by columns and rows before finally removing entries in duplicate positions. Since devising a novel SpGEMM is beyond the scope of this paper, we will use the standard SpGEMM algorithm provided by cuSPARSE.

General-Purpose Computing on the GPU.

Modern GPUs are massively parallel processors that support tens of thousands of hardware-scheduled threads running simultaneously. These threads are organized into blocks and the hardware schedules these blocks of threads onto hardware cores. Efficient GPU programs have enough work per kernel to keep all hardware cores busy; ensure that all cores and threads are fully supplied with work (load-balancing); strive to reduce thread divergence (when neighboring threads branch in different directions); and attempt to access memory in large contiguous chunks to maximize achieved memory bandwidth (coalescing); and minimize communication between the CPU and GPU. Because of their high arithmetic and memory throughput, GPUs have been adopted to accelerate large-scale graph processing. Gunrock [29] is a high-performance graph processing framework on the GPU with a data-centric abstraction centered on operations on a vertex or edge frontier, providing a high-level APIs for programmers to quickly develop new graph primitives with small code size and minimal GPU programming knowledge.

3. PARALLEL TRIANGLE COUNTING ALGORITHMS

In this section, we discuss the problem of triangle counting (TC) from the perspective of subgraph matching, set intersection, and matrix multiplication. Our implementations using these approaches all perform exact triangle counting on undirected graphs. Figure 1 will be the sample graph we use to demonstrate our algorithms.

Figure 1: A simple graph example for algorithm illustration.

3.1 TC using Subgraph Matching

Subgraph matching is the task of finding all occurrences of a small query graph in a large data graph. In our formulation, both graphs are undirected labeled graphs. We can use subgraph matching to solve triangle counting problems by
first defining the query graph to be a triangle, then assigning a unified label to both the triangle and the data graph. One advantage of using subgraph matching to enumerate triangles in a graph is that we can get the listings of all the triangles for free. Another advantage is that we can generalize the problem to find the embeddings of triangles with certain label patterns.

Our method follows a filtering-and-joining procedure and is implemented using the Gunrock [29] programming model. Algorithm 1 outlines our subgraph matching-based triangle counting implementation. In the filtering phase, we focus on pruning away candidate vertices that cannot contribute to the final solutions; we note that nodes with degree less than two cannot be matched any query vertex, since every node in a triangle has a degree of two. We use one step of Gunrock’s Advance operator to fill a boolean candidate set (denoted as \( c \_ set \)) for the data graph. If node \( i \) in the data graph is a candidate to node \( j \) in the query graph, then \( c \_ set[i][j] \) has a value of one. Otherwise, \( c \_ set[i][j] = 0 \). We then use our \( c \_ set \) to label candidate edges corresponding to each query edge using another Advance and collect them using a Filter operator. Then we can get an edge list of the new data graph composed of only candidate edges. After collecting candidate edges for each edge in the triangle, we test if each combination of the edges satisfies the intersection rule defined by the input triangle. If so, we count them into the output edge list. Figure 2 shows a running example of the graph sample from Figure 1.

![Figure 2: A simple graph example for SM-based TC illustration.](image)

Since every query node has the same degree of two and the degree of a node in a large connected graph is usually larger than two, the filtering phase cannot prune away that many nodes simply based on node degrees and candidate neighbor lists. The most important, and also the most time-consuming, step is joining. Tran et al. [24] use a two-step scheme to collect candidate edges into a hash table. First, they count the number of candidate edges for each query edge. Then, they compute the address of each candidate edge and assign them to the computed address in the hash table. In our implementation, we first use a massively parallel advance operation to label each candidate edge with its corresponding query edge id. Then we use the select primitive from Merrill’s CUB library\(^1\) to collect candidate edges for each query edge.

\(^1\)http://nvlabs.github.io/cub/

### Algorithm 1 TC using subgraph matching.

**Input:** Query Graph \((\text{T})\), Data Graph \(G\).

**Output:** Number of triangles \(n\) and listings of all matches.

1. **procedure**\( \text{initialize\_candidate\_set}(T, G) \)
2. \( \text{Advance}(G) \triangleright\) Fill \( c \_ set \) based on node label and degree
3. **end procedure**

4. **procedure**\( \text{Collect\_candidate\_edges}(G, T, c \_ set) \)
5. \( \text{Advance}(G) \triangleright\) Label candidate edges with query_edge_id
6. \( \text{filter}(G) \triangleright\) Collect candidate edges
7. **return** \( ELIST \)
8. **end procedure**

9. **procedure** join\_candidate\_edges\((ELIST, \text{intersection\_rule})\)
10. **parallel** for each candidate edge combination \( \{e_i, e_j, e_k\} \)
11. if \( \{e_i, e_j, e_k\} \) satisfy \( \text{intersection\_rules} \)
12. \( \{e_i, e_j, e_k\} \) to output list
13. Add 1 to count
14. **end if**
15. **return count, output list**
16. **end procedure**

### 3.2 TC using Set Intersection

The extensive survey by Schank and Wagner [20] shows several sequential algorithms for counting and listing triangles in undirected graphs. Two of the best performing algorithms, \text{edge-iterate} and \text{forward}, both use edge-based set intersection primitives. The optimal theoretical bound of this operation coupled with its high potential for parallel implementation make this method a good candidate for GPU implementation.

![Figure 3: The workflow of intersection-based GPU TC algorithm.](image)

We can view the TC problem as a set intersection problem by the following observation: An edge \( e = (u, v) \), where \( u, v \) are its two end nodes, can form triangles with edges connected to both \( u \) and \( v \). Let the intersections between the neighbor lists of \( u \) and \( v \) be \((w_1, w_2, \ldots, w_N)\), where \( N \) is the number of intersections. Then the number of triangles formed with \( e \) is \( N \), where the three edges of each triangle are \((u, v), (w_i, u), (w_i, v)\), \( i \in [1, N] \). In practice, computing intersections for every edge in an undirected graph is redundant. We visit all the neighbor lists using Gunrock’s Advance operator. Then we filter out half of the edges by degree order. Thus, in general, set intersection–based TC algorithms have two stages: (1) forming edge lists; (2) computing set intersections for two neighbor lists of an edge. Different optimizations can be applied to either stage. Our GPU implementation (shown in Algorithm 2) follows the \text{forward} algorithm and uses several operators of the high-performance graph processing library Gunrock [29]. Figure 3 is the flow chart that shows how this algorithm works on the running example (Figure 1).
Algorithm 2 TC using edge-based set intersection.

1: procedure FORM_FILTERED_EDGE_LIST(G)
2:    ADVANCE(G)
3:    FILTER(G)
4:    return ELIST
5: end procedure

6: procedure COMPUTE_INTERSECTION(G, ELIST)
7:    {SmallList,LargeList} = PARTITION(G, ELIST)
8:    LARGE_NEIGHBOR_LIST_INTERSECTION(LargeList)
9:    SMALL_NEIGHBOR_LIST_INTERSECTION(SmallList)
10: end procedure

3.3 TC using Matrix Multiplication

The matrix multiplication formulation comes from Azad, Bulu¸c, and Gilbert’s algorithm for counting and enumerating triangles [2]. Nodes in the adjacency matrix $A$ are arranged in some order. The lower triangular matrix $L$ represents all edges from row $i$ to column $k$ such that $k \leq i$. The upper triangular matrix $U$ represents all edges from row $k$ to column $j$ such that $j \leq k$. The dot product of row $i$ of $L$ with column $j$ of $U$ is a count of all wedges $i - k - j$ where $k \leq i$ and $k \leq j$. By performing a Hadamard product (element-wise multiplication) with $A_{ij}$, we obtain an exact count of the number of triangles $i - k - j$ for $k \leq i$ and $k \leq j$.

Algorithm 3 shows the pseudocode of our GPU implementation and Figure 4. We illustrate this algorithm by giving a running example of the sample graph (Figure 1) in Figure 4.

Algorithm 3 Counts how many triangles are in an undirected graph using matrix multiplication formulation.

Input: Adjacency matrix $A$.
Output: Number of triangles $n$.

1: procedure TRIANGLE_COUNT_MATRIX
2:    Permuted rows $A$ so that it is ordered by an increasing number of nonzeros.
3:    Break matrix into a lower triangular piece $L$ and an upper triangular piece $U$ such that $A = L + U$.
4:    Compute $B = LU$.
5:    Compute $C = A \circ B$ where $\circ$ is the Hadamard product.
6:    Compute $n = \frac{1}{2} \sum_i \sum_j A_{ij}$.
7: end procedure

4. IMPLEMENTATIONS

In this section, we discuss optimizations required for triangle counting algorithms to run efficiently on the GPU.

4.1 The Implementation of Subgraph-Matching-Based TC Algorithm

We make several optimizations to the algorithm proposed by Tran et al. [24] in both filtering and joining phases.

4.1.1 Optimizations for Candidate Node Filtering

The initial filtering phase takes nodes as processing units and uses the massively parallel advance operation in Gunrock to mark nodes with same labels and larger degrees as candidates in a candidate set. We then use another advance to label candidate edges based on the candidate set and use a filter operation in Gunrock to prune out all non-candidate edges and reconstruct the graph. Note that by reconstructing the data graph, we also update node degree and neighbor list information. So we can run the above two steps for a few iterations in order to prune out more edges.

4.2 The Implementation of Set-Intersection-Based TC Algorithm

The reason behind the high performance of our set-intersection-based implementation lies in the specific optimizations we propose in the two stages of the algorithm.

Figure 4: The number of triangles is yielded by summing the elements of $C$, then dividing by two. In this example, the number of triangles is $\frac{6}{2} = 3$. 
4.2.1 Optimizations for Edge List Generation Phase

Previous GPU implementations of intersection-based triangle counting compute intersections for all edges in the edge list. We believe they make this decision because of a lack of efficient edge-list-generating and -filtering primitives; instead, in our implementation, we leverage Gunrock’s operations to implement these primitives and increase our performance. To implement the forward algorithm, we use Gunrock’s advance V2E operator to generate the edge list that contains all edges, and then use Gunrock’s filter operator to get rid of \( e(u, v) \) where \( d(u) < d(v) \). For edges with the same \( d(u) \) and \( d(v) \), we keep the edge if the vertex ID of \( u \) is smaller than \( v \). This efficiently removes half of the workload. We then use segmented reduction in Merrill’s CUB library to generate a smaller induced subgraph contains only the edges that have not been filtered to further reduce two thirds of the workload.

4.2.2 Optimizations for Batch Set Intersection

High-performance batch set intersection requires a similar focus as high-performance graph traversal: effective load-balancing and GPU utilization. In our implementation, we use the same dynamic grouping strategy proposed in Merrill’s BFS work [18]. We divide the edge lists into two groups: (1) small neighbor lists; and (2) large neighbor lists. We implement two kernels (TwoSmall and TwoLarge) that cooperatively compute intersections. Our TwoSmall kernel uses one thread to compute the intersection of a node pair. Our TwoLarge kernel partitions the edge list into chunks and assigns each chunk to a separate thread block. Then each block uses the balanced path primitive from the Modern GPU library\(^2\) to cooperatively compute intersections. By using this 2-kernel strategy and carefully choosing a threshold value to divide the edge list into two groups, we can process intersections with same level of workload together to gain load balancing and higher GPU resource utilization.

4.3 The Implementation of Matrix-Multiplication-Based TC Algorithm

Our matrix-multiplication-based TC algorithm tests how well a highly optimized matrix multiplication function from a standard GPU library (cubgemm from cuSPARSE) performs compared to one implemented using a graph processing library (subgraph matching and set intersection). The cuSPARSE SpGEMM function allocates one CSR row in the left input matrix for each thread, which performs the dot product in linear time. The \( n \) rows need to be multiplied with \( n \) columns as mentioned in Section 3.3. This leads to a \( O(\sum_{e \in E} d(v)^2) \) complexity, which is the same as the set-intersection-based TC algorithm. For a detailed proof that shows this equivalence, see Schank and Wagner [20].

We compute the Hadamard product by assigning a thread to each row, which then performs a multiplication by the corresponding value in adjacency matrix \( A \). We incorporate two optimizations: (1) compute only the upper triangular values, because the output of the Hadamard product is known to be symmetric; (2) use a counter to keep track of the number of triangles found by each thread, allowing the sum of the number of triangles in each column with a single parallel reduction. Step 2 combines lines 5 and 6 in Algorithm 3 so that the reduction can be performed without writing the nonzeros into global memory.

5. EXPERIMENTS AND DISCUSSION

We compare the performance of our algorithms to three different exact triangle counting methods: Green et al.’s state-of-the-art GPU implementation [11] that runs on an NVIDIA K40c GPU, Green et al.’s multicore CPU implementation [10], and Shun et al.’s multicore CPU implementation [22]. Both of the state-of-the-art CPU implementations are tested on a 40-core shared memory system with two-way hyper-threading; their results are from their publications [10, 22]. Currently, our algorithms target NVIDIA GPUs only. However, we hope the discoveries made in this paper can help with triangle counting implementations on other highly parallel processors.

Our CPU baseline is an implementation based on the forward algorithm by Schank and Wagner [20].

**Datasets.** We test our implementations using a variety of real-world graph datasets from the DIMACS10 Graph Challenge [3] and the Stanford Network Analysis Project (SNAP) [16]. Table 1 describes the datasets. The topology of these datasets spans from regular to scale-free.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Vertices</th>
<th>Edges</th>
<th>Max Degree</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>coAuthorsCite</td>
<td>227,320</td>
<td>1,629,268</td>
<td>1372</td>
<td>rs</td>
</tr>
<tr>
<td>coPapersDBLP</td>
<td>540,486</td>
<td>30,491,458</td>
<td>3299</td>
<td>rs</td>
</tr>
<tr>
<td>road_central</td>
<td>14,081,816</td>
<td>33,866,826</td>
<td>8</td>
<td>rm</td>
</tr>
<tr>
<td>soc-LJ</td>
<td>4,847,571</td>
<td>137,987,546</td>
<td>20,292</td>
<td>rs</td>
</tr>
<tr>
<td>cit-Patents</td>
<td>3,774,768</td>
<td>33,037,896</td>
<td>770</td>
<td>rs</td>
</tr>
<tr>
<td>com-Orkut</td>
<td>3,072,441</td>
<td>234,370,166</td>
<td>33,007</td>
<td>rs</td>
</tr>
</tbody>
</table>

Table 1: Dataset Description Table. The edge number shown is the number of directed edges when the graphs are treated as undirected graphs (if two nodes are connected, then there are two directed edges between them) and de-duplicate the redundant edges. Graph types are: r: real-world, s: scale-free, and m: mesh-like.

**Environment.** We ran experiments on this paper on a Linux workstation with 2x3.50 GHz Intel 4-core, hyperthreaded E5-2637 v2 Xeon CPUs, 528 GB of main memory, and an NVIDIA K40c GPU with 12 GB on-board memory. Our GPU programs were compiled with NVIDIA’s nvcc compiler (version 7.5.17) with the -O3 flag.

**Perf. Overview.** Figure 5 compares our three TC implementations and several state-of-the-art shared-memory and GPU TC implementations with a baseline CPU implementation. In general, our intersection-based method shows better performance than the state-of-the-art GPU implementations because of our key optimizations on workload reduction and GPU resource utilization. It achieves comparable performance to the fastest shared-memory CPU TC implementation. Our matrix-based method illustrates some inherent problems with using matrix multiplication to compute TC. Our subgraph-matching-based method shows some performance wins on mesh-like datasets since we prune out a lot of leaf nodes in our filtering base and can thus reduce around one-third of work for mesh-like graphs during the joining phase. Figure 6 confirms our expectation from Section 4.3 that both our intersection-based and matrix-based implementations

\(^2\)Code is available at http://nvlabs.github.io/moderngpu
have runtime proportional to the sum of square degrees of their input graphs.

**Perf. Summary for Subgraph Matching-based TC.**

Our subgraph matching-based triangle counting enumerates all the triangles for free and thus needs a lot of memory in the joining phase. The optimizations of the joining phase give us a performance win compared with previous state-of-the-art subgraph matching algorithms. As shown in Figure 5, most of the time, subgraph matching is slower than intersection-based methods. Subgraph matching, however, does well on some mesh-like datasets that contain a comparatively large number of leaf nodes, which we can filter out in our first stage. For the joining phase, the more candidate edges we get, the more edge combinations we see. When the number of combinations exceeds the number of parallel processors we have, each processor needs to deal with multiple combinations sequentially, which hurts performance.

**Perf. Summary for Intersection-based TC.**

Our intersection-based TC with only the simple per-thread batch set intersection kernel achieves a $2.51 \times$ speedup (geometric-mean) speedup compared to Green et al.’s CPU implementation [10] and a $4.03 \times$ geomean speedup compared to Green et al.’s GPU implementation [11]. We believe our speedup is the result of two aspects of our implementation: using filtering in edge list generation and reforming the induced subgraph with only the edges not filtered which reduces five-sixths of the workload, and dynamic grouping helps maintain a high GPU resource utilization. In practice we observe that for scale-free graphs, the last step of optimization to reform the induced subgraph show a constant speedup over our intersection method without this step. However, for road networks and some small scale-free graphs, the overhead of using segmented reduction will cause a performance drop. We expect further performance gains from future tuning of launch settings for the large-neighbor-list intersection kernel, which we do not believe is quite optimal yet. Furthermore, we believe that a third kernel to do batch set intersection between one large and one small neighbor list will continue to improve performance. This third kernel would scan each node in the smaller list and search the larger list. Highly skewed scale-free graphs where most intersections are between a very small neighbor list and the same large neighbor list which covers more than half of the nodes in the graph would see a performance improvement from a kernel like this. With little modification, we expect our method could also be used to compute clustering coefficient and transitivity metrics.
Previous work in triangle counting on massively parallel processors like GPUs has concentrated on single implementations using a particular methodology. With this work we analyzed three different GPU implementations based on subgraph matching, set intersection, and matrix multiplication. Our intersection-based implementation achieves state-of-the-art performance and shows potential to gain even better performance. Our matrix-based implementation shows that SpGEMM is the performance bottleneck on GPU due to its unavoidable redundant work, although future work on making customized modifications to reduce redundant work could lead to better speedups. Our subgraph matching method, despite its memory bound, shows good performance on mesh-like graphs that contain a large part of leaf nodes. Its generality also allows programmers to easily extend this method to match more complicated subgraph patterns.

We also note that some optimizations such as filtering edge lists and improving set intersection can easily address other problems on the GPU, while others such as joining order selection and reducing intermediate results are considerably more challenging. We hope that the most challenging ones will serve as interesting future work for the authors and, more importantly, interesting future work for the communities developing frameworks for those particular methodologies.

Acknowledgements

Thanks to Seshadhri Comandur for providing the CPU baseline code for comparisons. We appreciate the funding support of DARPA XDATA under grants US Army award W911QX-12-C-0059, DARPA STTR awards D14PC00023 and D15PC00010 as well as the National Science Foundation under grants CCF-1017399 and OCI-1032859, and a Lawrence Berkeley Laboratory internship. Thanks also to NVIDIA for equipment donations and server time.

6. CONCLUSION

Figure 6: Schank and Wagner [20] show that the runtime of triangle counting on a particular graph should be proportional to the sum of square degrees (SSD) of that graph: \( O(\sum_{v \in V} d(v)^2) \). For our intersection-based (left) and matrix-based (right) implementations, this is true; the line of best fit for both runtime vs. SSD plots has a slope of one. The flat runtime baseline in the right figure indicates kernel launch overhead in the matrix multiplication operation. The y-intercept in a log-log plot is equivalent to the constant term in Big-O notation. The leading constant term (3.059 \times 10^{-7} vs. 6.144 \times 10^{-6}) indicates our implementation of intersection-based triangle counting is faster than our matrix-based implementation.

**Perf. Summary for Matrix Multiplication-based TC.**

Figure 5 shows how the matrix multiplication-based TC compares with other algorithms. It has similar performance to the reference CPU algorithm in two out of three datasets. For the largest dataset of the three “coPapersDBLP”, it does perform 26\times better, indicating better scaling.

Figure 6 shows how matrix-based TC scales. The linear relationship beyond the high slope near the start indicates that the algorithm is bounded by the same complexity as the intersection-based algorithm. The low slope at the start is indicative of the large overhead from the SpGEMM algorithm, which is mitigated once the graph grows to a certain size.

Testing with profiler tools indicate SpGEMM is the bottleneck of our implementation. Since sparse matrix multiplication needs to generate an intermediate array, our results suggest that even an optimized, out-of-the-box SpGEMM cannot compete with a graph processing library, because SpGEMM is doing unnecessary work and within the matrix formulation. There do not seem to be some TC-specific optimizations. For the latter, these include: (1) only calculate the upper triangular of the matrix multiplication; (2) avoid multiplications where \( A \) is known to be zero a priori; and (3) avoid writing the output of the matrix multiplication into global memory. Techniques such as masking the input adjacency matrix \( A \) have been shown to yield speed-ups on distributed systems [2]. Implementing this on a GPU is a good direction for future research.

7. REFERENCES


