A Comparative Study on Exact Triangle Counting Algorithms on the GPU

Leyuan Wang, Yangzihao Wang, Carl Yang,
John D. Owens

University of California, Davis, CA, USA

31^{st} May 2016

Outline

Why triangle counting?

Graphs have been used to model interactions between entities in many applications.

- Finding small subgraph patterns is useful for understanding the underlying structure of these graphs;
- The most important such graph is the triangle.
- Many important measures of a graph are triangle-based, e.g. clustering coefficient and the transitivity ratio.

Subgraph Matching-based TC

Problem definition: finding all occurrences of a small query graph in a large data graph. 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.

Backtracking strategy (Ullmann [3]):
Incrementally compose partial solutions or abandoning them when it determines they cannot be completed.

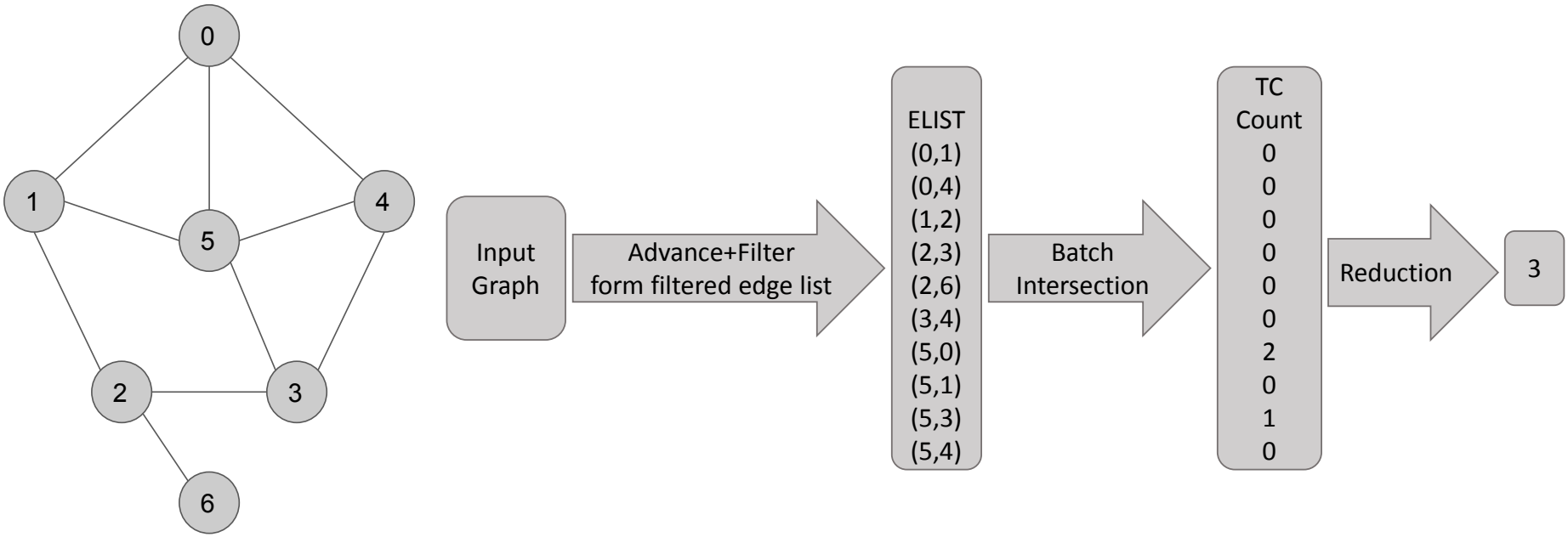
Improvements: filtering rules, joining orders, and auxiliary information to prune out false-positive candidates as early as possible, thereby increasing performance.

Advantages:

- Get the listings of all the triangles for free
- Can be generated to find the embedding of triangles with certain label patterns.

Intersection-based TC

For each edge, count the number of common nodes (intersection) between the neighbor lists of the two end vertices.



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 $(u, v), (w_i, u), (w_i, v), i \in [1, N]$.

Matrix Multiplication-based TC

Method (based on the algorithm of Azad et al. [1]):

1. Order rows in the adjacency matrix A with increasing vertex degree.
2. Break the rearranged matrix into a lower triangular piece L and an upper triangular piece U such that $A = L + U$.
3. Count all the wedges of (i, j, k) where (i, k) is an edge in L with $i \geq k$ and (k, j) is an edge in U with $k \leq j$ by $B = L \cdot U$.
4. Find if the wedges close by element-wise multiplication (Hadamard product) $C = A. * B$
5. Then we can count the number of triangles by summing all the elements in C and divide the sum by 2.

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\mathbf{L} = \begin{pmatrix} 0 & & & & & & \\ 1 & 0 & & & & & \\ 0 & 1 & 0 & & & 0 & \\ 0 & 0 & 1 & 0 & & & \\ 1 & 0 & 0 & 1 & 0 & & \\ 1 & 1 & 0 & 1 & 1 & 0 & \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\mathbf{U} = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ & 0 & 1 & 0 & 0 & 1 & 0 \\ & & 0 & 1 & 0 & 0 & 1 \\ & & & 0 & 1 & 1 & 0 \\ & & & & 0 & 1 & 0 \\ & 0 & & & & 0 & 0 \\ & & & & & & 0 \end{pmatrix}$$

$$\mathbf{B} = \mathbf{LU}$$

$$= \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 2 & 2 & 0 \\ 0 & 1 & 1 & 0 & 2 & 4 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{C} = \mathbf{A} \circ \mathbf{B}$$

$$= \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Our contributions

- We do a comparative study on three specialized parallel triangle counting algorithms with highly scalable implementations on NVIDIA GPUs.
- We 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.
- We develop a best-of-class **intersection** operator and integrate it into our second triangle counting method, yielding a speedup as high as 630× over the sequential CPU implementation.
- We formulate a parallel **matrix-multiplication-based** method that achieves 5–26× speedup over the sequential CPU implementation.

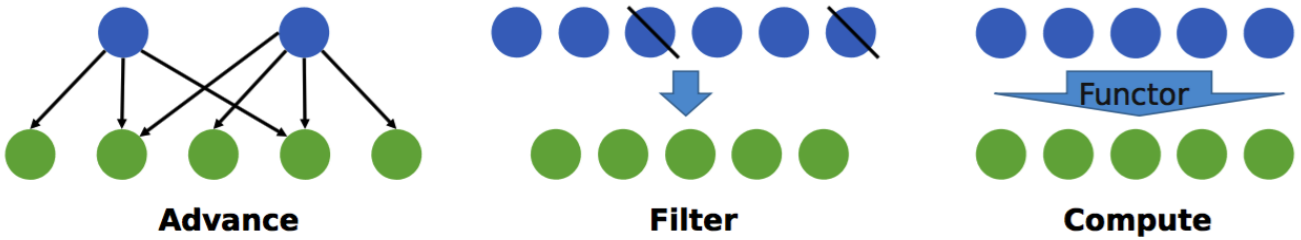
Gunrock Programming Model [4]

Graph represented as Compressed Sparse Row (CSR): sparse matrix ⇒ row offsets, column indices and values.

Bulk-Synchronous Programming: series of parallel operations separated by global barriers.

Data-Centric Abstraction: operations are defined on one or more *frontiers* of active vertices/edges.

- **Advance:** generates a new frontier via visiting the neighbor vertices/edges in the current frontier (work distribution/load balancing)
- **Filter:** removes elements from a frontier via validation test
- **Compute:** user-defined vertex/edge-centric computations that run in parallel, can be combined with advance or filter



TC using Subgraph Matching

Challenges:

- 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.
- Irregular memory access patterns cause memory accesses to become uncoalesced, which degrades GPU performance.

Solutions:

- Follow a *filtering-and-joining* procedure using Gunrock's programming model.
- *Filtering*: prune away candidate vertices that cannot contribute to the final solutions.
- *Joining*: collect candidate edges for each query edge and then combine the edges that satisfy the predefined intersection rules.

TC using Set Intersection

Method:

1. Form edge list.
2. Compute set intersection for two neighbor lists of each edge.

Optimizations:

- We visit all the neighbor lists using Gunrock's Advance operator. Then we filter out half of the edges by node degree/id order to avoid redundant counting.
- Use dynamic grouping strategy to divide the edge lists into two groups (1) small neighbor lists; and (2) large neighbor lists and use different work distribution methods for the two groups in batch set intersection to achieve load balancing and to maximize resource utilization.

TC using Matrix Multiplication

Challenges:

- An efficient sparse matrix multiplication implementation.
- Since sparse matrix multiplication needs to generate an intermediate array, redundant work is done within the matrix formulation.

Solutions:

- Use highly optimized GPU matrix multiplication function *csrgemm* from *cuSPARSE* to compute B.
- Only compute Hadamard product for L/U instead of A since C is symmetric.
- Assign a thread to each row and use a counter to keep track of the number of triangles found by each thread, allowing the final sum with a single parallel reduction.

Dataset

Dataset	Vertices	Edges	Max Degree	Type
coAuthorsCiteseer	227,320	1,628,268	1372	rs
coPapersDBLP	540,486	30,491,458	3299	rs
road_central	14,081,816	33,866,826	8	rm
soc-LJ	4,847,571	137,987,546	20,292	rs
cit-Patents	3,774,768	33,037,896	770	rs
com-Orkut	3,072,441	234,370,166	33,007	rs

Table: 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: regular, s: scale-free, and m: mesh-like.

[1]The datasets are from DIMACS10 Graph Challenge and the Stanford Network Analysis Project (SNAP).

Performance

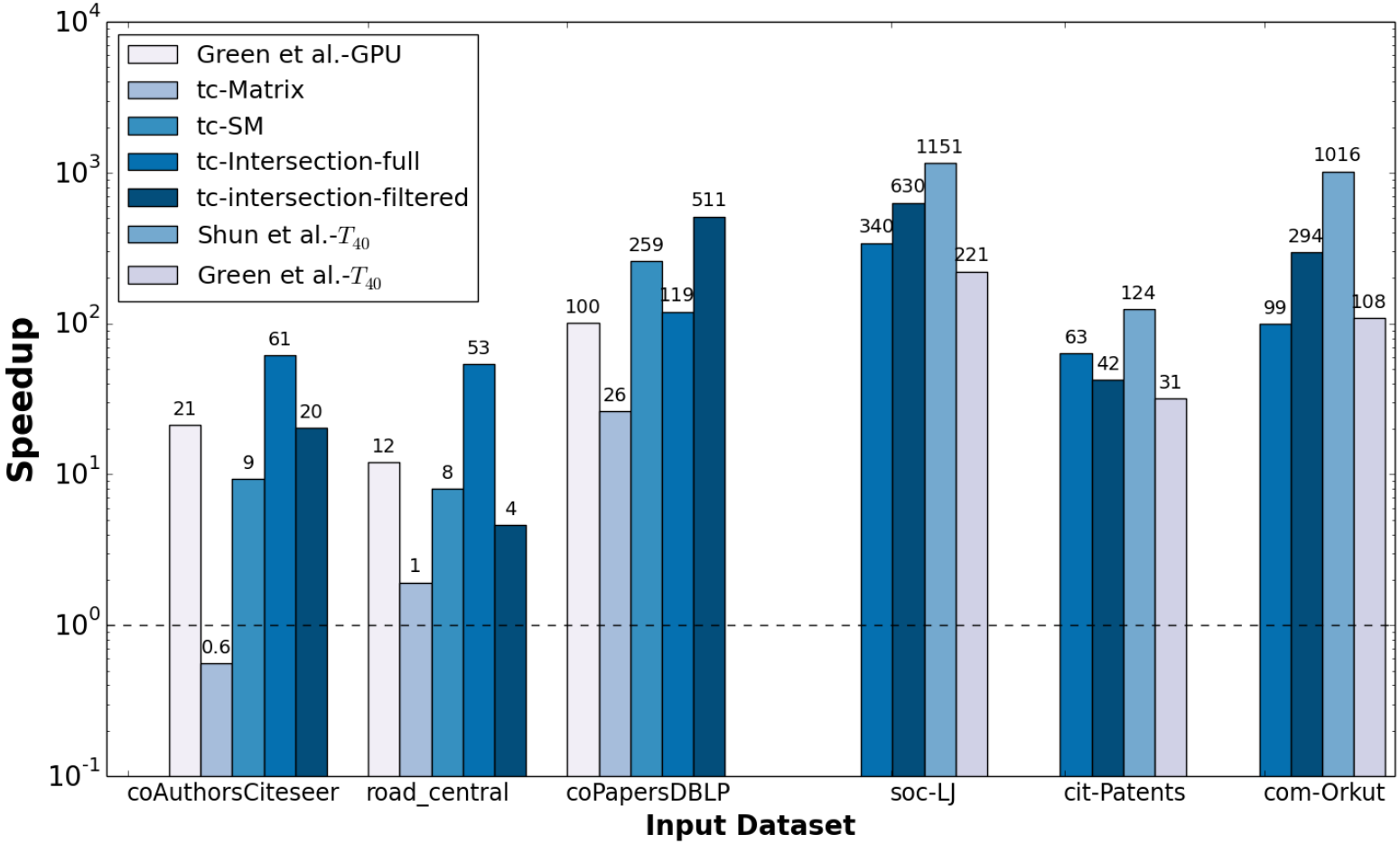
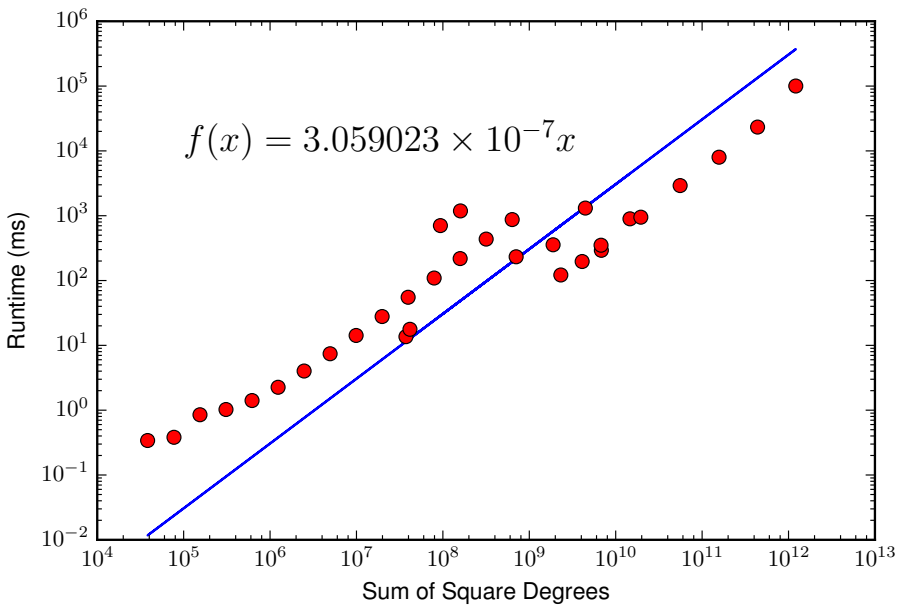


Figure: Execution-time speedup for our four GPU implementations, Green et al.'s GPU implementation, Shun et al.'s 40-core CPU implementation and Green et al.'s 40-core CPU implementation. All are normalized to a baseline CPU implementation
`https://bitbucket.org/seshadhri/escape`

Complexity Analysis



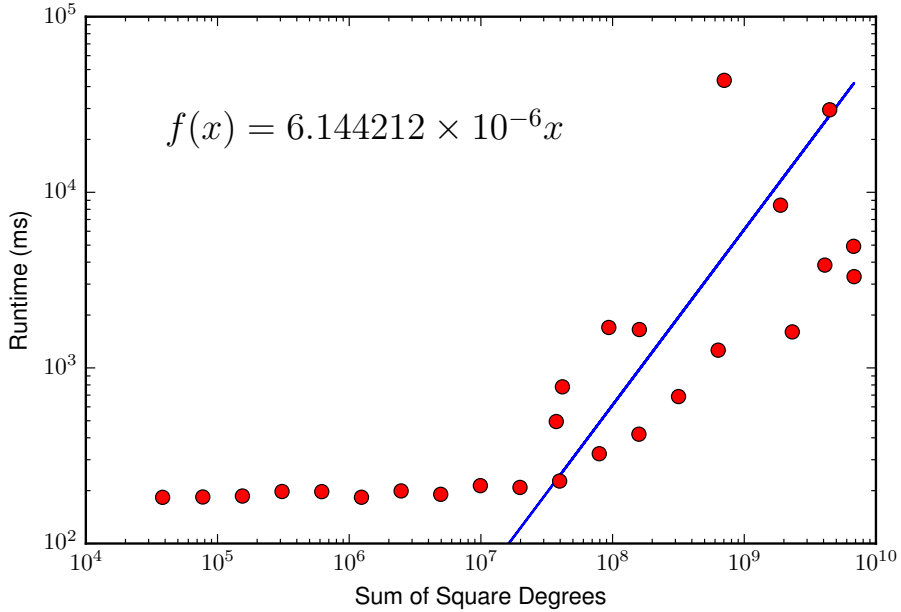


Figure: The runtime of triangle counting on a particular graph should be proportional to the sum of square degrees (SSD) of that graph: $O(\sum_{v \in V} d(v)^2)$. Left, intersection-based method; right, matrix-based method.

Subgraph Matching Generality Tests

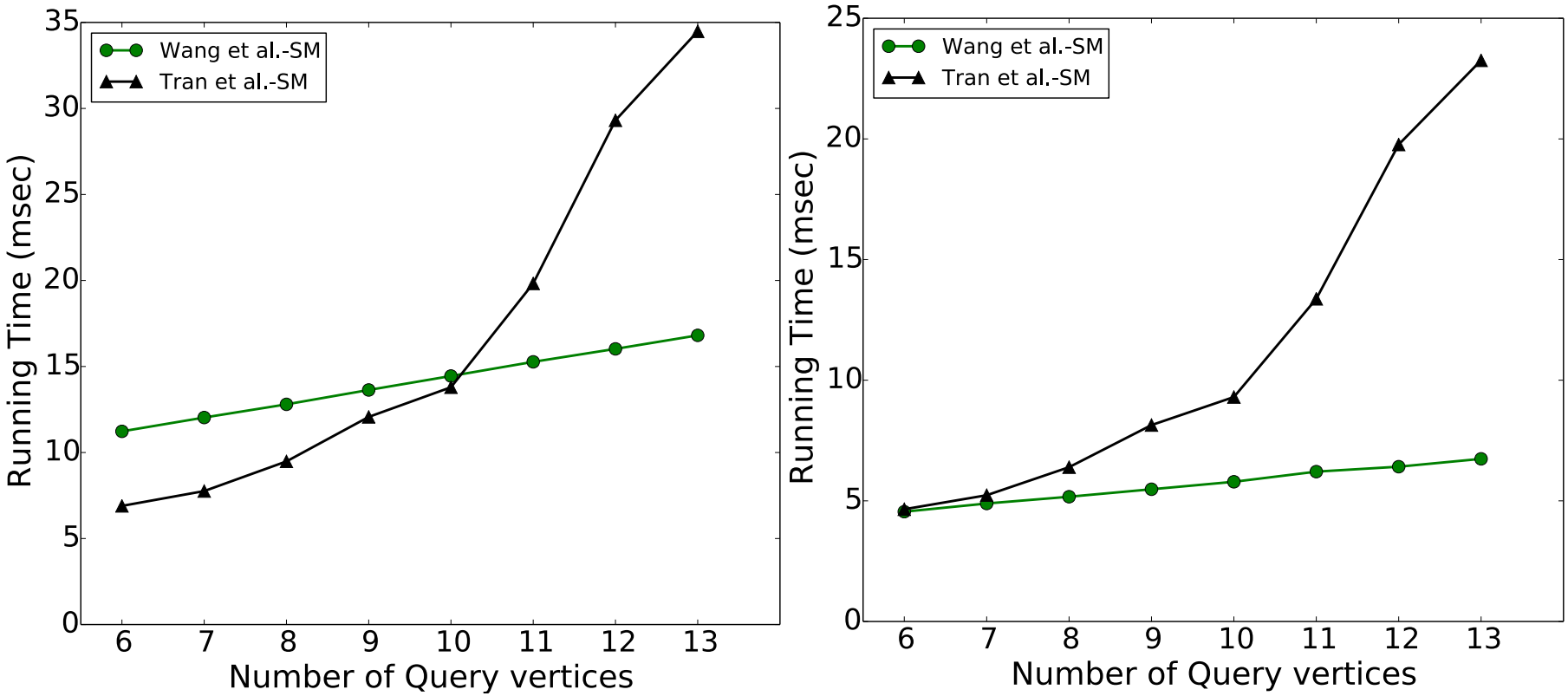


Figure: Speedups on Gowalla (left) and Enron (right) datasets of our PSM and a previous state-of-the-art GPU implementation by Tran et al. [2].

Discussion

- Intersection-based TC shows best performance among our three GPU implementations.
- Subgraph-matching-based TC shows some performance wins on mesh-like datasets since we prune out a lot of leaf nodes in our filtering step.
- Our intersection-based TC achieves better results than previous intersection-based implementations because (1) we filter edge list and reform the induced subgraph to reduce workload (2) we use dynamic work scheduling to maintain a high GPU resource utilization. For scale-free graphs, (1) gives us constant speedups over our implementation without this step. But for road networks and some small scale-free graph, the overhead of (1) cause a performance drop.
- Our matrix multiplication-based TC is bounded by the same complexity as our intersection-based TC, the low slope at the start indicates large overhead from the *SpGEMM* algorithm which is bottleneck of our implementation.

Future Work

- For subgraph matching-based TC, finding a good matching order for the query graph can reduce a lot of intermediate results.
- For intersection-based TC, we expect further performance gains from tuning of launch settings for the large-neighbor-list intersection kernel. We also believe that a third kernel to do batch set intersection between one large and one small neighbor list using a binary search operation will continue to improve performance.
- For matrix multiplication-based TC, we believe that avoiding multiplications where the input adjacency matrix A is known to be zero and avoiding writing the multiplication output to global memory.

Conclusions

- We implemented and compared three triangle counting methods on the GPU based on subgraph matching, set intersection and matrix multiplication.
- Our intersection-based implementation achieved state-of-the-art performance and potential to gain even better performance.
- Our matrix-multiplication-based TC implementation shows that SpGEMM is the performance bottleneck due to its unavoidable redundant work.
- Our subgraph matching implementation is memory bound, but shows good performance on mesh like graphs and it also allows programmers to extend this method to match more complicated subgraph patterns.

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.

[2]Contact Info: leywang@ucdavis.edu

References

A. Azad, A. Buluç, and J. Gilbert.

Parallel triangle counting and enumeration using matrix algebra.

In *IEEE International Parallel and Distributed Processing Symposium Workshop*, IPDPSW 2015, pages 804–811, 2015.

H.-N. Tran, J.-j. Kim, and B. He.

Fast subgraph matching on large graphs using graphics processors.

In M. Renz, C. Shahabi, X. Zhou, and A. M. Cheema, editors, *Proceedings of the 20th International Conference on Database Systems for Advanced Applications*, DASFAA 2015, pages 299–315. Springer International Publishing, Cham, Apr. 2015.

J. R. Ullmann.

An algorithm for subgraph isomorphism.

Journal of the ACM, 23(1):31–42, Jan. 1976.

Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens.

Gunrock: A high-performance graph processing library on the GPU.

In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and*