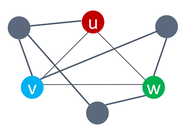
Parallel Clustering Coefficient Computation using GPUs

Tahsin Reza, Matei Ripeanu
Electrical and Computer Engineering
University of British Columbia, Vancouver

Tanuj Kr Aasawat
Department of Information Technology
Jadavpur University, Kolkata, India

Problem and Contributions

Clustering coefficient is the measure of how tightly vertices are bounded in a network [1]. The Triangle Counting problem is at the core of clustering coefficient computation and performance depends on how efficiently triangles are counted. If two vertices $\{u, v\} \in V$ are neighbours, and there exists a third vertex $w \in V$ where $w \in F(u)$ and $w \in F(v)$, then $\{u, v, w\}$ forms a triangle. Clustering coefficient of v, is computed using the following equation:



$$cc_v = \frac{2T_v}{d_v(d_v - 1)}$$

$G = (V, E)$ – an undirected graph
V – set of vertices, E – set of edges
$F(v)$ – set of neighbours of any vertex $v \in V$
d_v – degree of v
d_{max} – maximum vertex degree in G
T_v – number of triangles incident on v
cc_v – clustering coefficient of v

Contribution highlights

- Scalable implementation of parallel clustering coefficient algorithm on GPUs.
- Report performance numbers for large graphs not seen in the literature for single-node in-memory systems before.
- GPU implementation offers 7x speedup over the best reported running time for the same graph.
- We can compute clustering coefficient of each vertex in power-law [2] graphs with up to 32M vertices and 512M edges using a single GPU.
- For the parallel CPU implementation, we present results for graphs with up to 4B edges.

Parallel Clustering Coefficient Algorithm

- *NodeIterator* [4] is the baseline algorithm. It counts the same triangle six times. Complexity - $O(d_{max}^2.|V|)$
- *NodeIterator++* [4] counts each triangle only once. Neighbour list is sorted with respect to vertex degree. Only the lowest degree vertex in each triangle is responsible for counting the triangle.
- Using *NodeIterator++* it is not possible to explicitly count the number of triangles incident on each vertex, which is essential for counting clustering coefficient of each vertex.

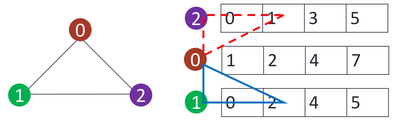


Figure (left): NodeIterator would count the triangle incident on 0 twice, once for 1 and once for 2, while scanning through their respective neighbour list. We can avoid this by not counting the triangle shown in red dotted lines.

- Our technique targets CSR graphs [3] and requires the neighbour list of each vertex to be sorted in ascending order with respect to vertex ID.
- Sorted lists allows us to skip many expensive memory accesses while computing the intersection of two neighbour lists, the key building block of triangle counting.
- The neighbour with the smaller vertex ID is responsible for counting a triangle. We count each triangle incident on a vertex only one time.

We present two implementations

- *Vertex-centric* – a single thread counts triangles incident on a vertex.
- *Edge-centric* – each edge is processed by a single thread. The number of threads that are used to count triangles incident on a vertex is equal to its vertex degree.

Vertex-centric algorithm

```
C[|V|] = 0 // clustering coefficient

for each v ∈ V in parallel
  for each e ∈ F(v)
    count_trainagles(v, e, T)
  C[v] = 2 * T / ( d_v * (d_v - 1))
```

Pseudocode (right): Array U has the same length as E. ($U[x]$, $E[x]$) represents an edge, where x is the array index. For storage efficiency, we recycle U to hold intermediate triangle counts.

Edge-centric algorithm

```
C[|V|] = 0 // clustering coefficient
U[|E|] = 0

prpcedure initialize
for each v ∈ V in parallel
  for each e ∈ F(v) U[e] = v

procedure triangle_counting
for each u ∈ U and e ∈ E in parallel
  count_trainagles(u, e, t)
  U[u] = t

procedure cc_computation
for each v ∈ V in parallel
  for each e ∈ F(v)
    T = T + U[e]
  C[v] = 2 * T / ( d_v * (d_v - 1))
```

```
procedure count_triangles(v, e, T)
if (d_v > d_u) L = v, S = e
  else L = e, S = v,
  i = 0, j = 0
  while (j < F(S).length)
    a = F(S)[j], b = F(L)[i]
    if (S == e)
      if (a < e) j++
        if (a < b) i++
        continue
    if (L == e)
      if (b < e) i++
        if (a > b) j++
        continue
    if (a == b) i++, j++, T++
    else if (a > b)  j++
    else if (a < b)  i++
    if (i == (F(L).length - 1)) break
```

Performance

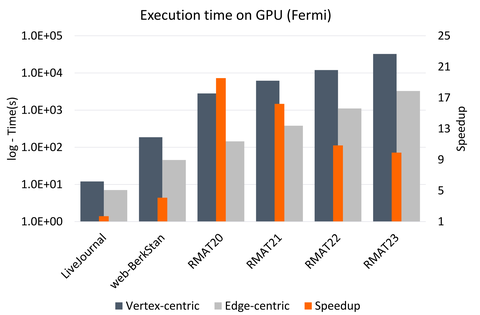
- GPU implementation achieves 7x speedup over the best known work [5] for the same graph (LiveJournal) while CPU implementation offers 3.3x speedup.
- Edge-centric implementation, up to graphs with 512M edges (RMAT25) on a Kepler GPU with 12 GB device memory.
- Edge-centric is constantly better than vertex-centric as it leverages an order of magnitude more parallelism.
- Kepler GPUs offer speedup over Fermi GPUs but not as much as compared to moving from vertex-centric to edge-centric.
- CPU implementation, up to graphs with 4B edges (RMAT28).

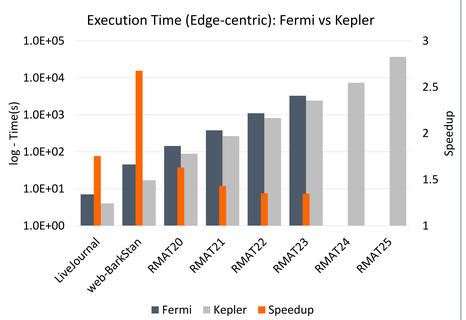
Graph	Vertices	Edges
LiveJournal *	4.8M	69M
web-BerkStan*	0.69M	13M
RMAT20	1M	16M
RMAT21	2M	32M
RMAT22	4M	64M
RMAT23	8M	128M
RMAT24	16M	256M
RMAT25	32M	512M
RMAT26	64M	1B
RMAT27	128M	2B
RMAT28	256M	4B

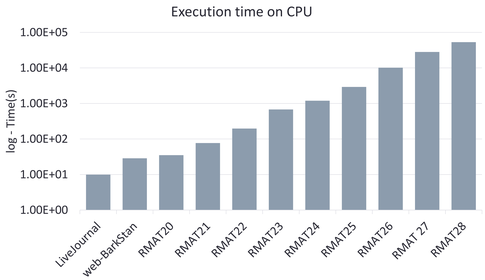
Table (above): Dataset used. *Real-world graphs were obtained from [7]. RMAT (Recursive MATrix) [6] graphs were generated using parameters (A,B,C) = (0.57,0.19,0.19) and an avg. vertex degree of 16. All the graphs we used for evaluation are undirected.

Graph	CPU	GPU (Kepler)	[2]	[4]	[5]	[9]	[10]
LiveJournal	9.94s	4.0s	32.98s	1.7min	29.43s	3.63min	4.70min
web-BerkStan	28.70s	17.0s	-	5.33min	-	1.31min	2.03min

Table (right): Comparison of running time with [2], [4], [5], [9] and [10]. Column headings CPU and GPU indicate our work.







Complexity

Vertex-centric $O(d_{max}^2.|V|)$
Edge-centric
`initialize` $O(|V|.d_{max})$
`triangle_counting` $O(|E|.d_{max})$
`cc_computation` $O(|V|.d_{max})$

$2*O(|V|.d_{max}) + O(|E|.d_{max}) < O(d_{max}^2.|V|)$

Discussions and Future Work

- CPU implementation performs better than GPU in the case of RMAT graphs but not for real-world graphs –*Why*?
- Power-law graphs are known to be highly irregular, i.e., vertex degree heavily varies across vertices. While the real-world graphs we experimented with have power-law degree distribution, they are less skewed than the RMAT graphs.
- Irregularity leads to work imbalance among GPU threads and causes scattered memory access pattern, which in turn results in uncoalesced global memory access on the GPUs and hurts performance [8].

Graph	Type	d_{max}	Standard Deviation of vertex degree distribution	CPU	GPU (Kepler)
LiveJournal	Real-world	20293	65.08	9.94s	4.0s
RMAT20	RMAT	146092	355.75	35.17s	88.5s

Graph	Achieved Occupancy	Warp Execution Efficiency	Stall Execution Dependency	Global Memory Load Efficiency	Global Memory Store Efficiency
LiveJournal	0.5	34.80%	32.77%	14.28%	71.28%
RMAT20	0.1	7.03%	88.19%	13.30%	35.51%

Achieved Occupancy (0.0 – 1.0)
Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor. (*Higher is better*)

Warp Execution Efficiency
Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor. (*Higher is better*)

Stall Execution Dependency
Percentage of stalls occurring because an input required by the instruction is not yet available. (*Lower is better*)

Global Memory Load/Store Efficiency
Ratio of requested global memory load/store transactions to actual global memory load/store transactions. (*Higher is better*)

- The RMAT graph suffers from low occupancy as a result of high stall execution dependency. Low number of active warps per processor cycle means more cycles is required to complete a task.
- The RMAT graph incurs low memory efficiency as a result of highly scattered memory access pattern which increases number of memory transactions and wastes memory bandwidth.

Future work

- Load-balancing mechanism on the GPU to improve performance of highly irregular graphs, e.g., virtual-warp technique for handling work imbalance in irregular graphs [8].
- CPU-GPU or multi-GPU implementation to process graphs larger than what a single GPU can accommodated.

References

[1] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of "small-world" networks. Nature, 393:440–442, 1998.

[2] Latapy, M. Main-Memory triangle computations for very large (sparse (power-law)) graphs. Theor. Comp. Sci., 407, 1–3, 458–473, 2008.

[3] Bell, N. and Garland, M. Efficient sparse matrix-vector multiplication on CUDA. In Proc. Conf. Supercomputing, 2009.

[4] Suri, S. and Vassilvitskii, S. Counting triangles and the curse of the last reducer. In WWW. ACM, 2011.

[5] Chu, S. and Cheng, J. Triangle listing in massive networks. ACM Trans. Knowl. Discov. Data 6, 4, Article 17, 32 pages, 2012.

[6] Chakrabarti, D., Zhan, Y. and Faloutsos, C. R-MAT : A Recursive Model for Graph Mining. In SDM, 2004.

[7] SNAP: http://snap.stanford.edu/

[8] Hong, S., Kim, S. K., Oguntebi , T. and Olukotun, T. Accelerating CUDA graph algorithms at maximum warp. In Proc. of ACM symposium on Principles and practice of parallel programming, 2011.

[9] Park, H-M. and Chung, C-W. An efficient MapReduce algorithm for counting triangles in a very large graph. In Proc. of Conference on information & knowledge management. ACM, New York, NY, USA, 539-548, 2013.

[10] Cohen, J. Graph twiddling in a mapreduce world. Computing in Science & Engineering, 11(4):29-41,2009.