15-618 Final Project Report

Manish Jain (manishj)   Vashishtha Adtani(vadtani)

Title: Parallel Triangle Counting in a graph using CUDA

Summary:

The main aim of our project is to evaluate the practicability of triangle counting in very large graphs with various degree distributions. Triangle counting helps to analyse big graphs/networks. Since, the size of these networks grow rapidly, we will need an algorithm that can cope up with this growth. In this project, we implemented triangle counting algorithm on CUDA. Our algorithm is a slightly optimized version of algorithm presented in the reference paper[1]. We are able to gain on an average 5x speedup than the algorithm presented by authors in the paper.

Background

Graphs can be 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. Therefore, 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. The transitivity ratio is the probability of wedges (three connected nodes) forming a triangle.

So, our code will take a graph G as input and output a single integer which would represent the number of distinct triangles present in the graph. E.g: Our code ran on figure 1 will return 3 number of triangles. With the increase in graph size the amount of search for triangles in the graph increases exponentially. This is where the scope of parallelism comes in, because we are only reading the data and such a large amount of data can be read in parallel to perform the counting more efficiently. Previously, many sequential algorithms have been implemented to solve this problem. Each node's computation in independent with respect to each other and allel to find where all We are planning to implement a parallel algorithm which will run on GPUs to solve this problem.

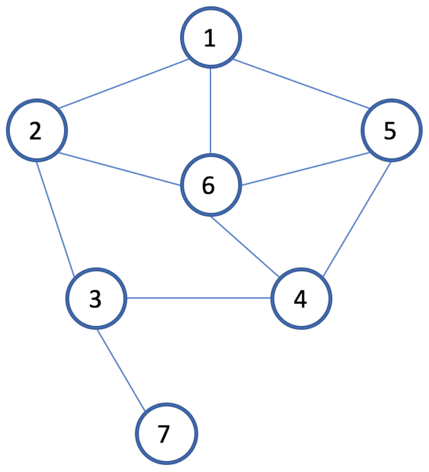


Figure 1: A simple graph example for algorithm illustration

Test Graphs

We are performing the analysis on the graphs mentioned in the reference paper [1]. The data sets were downloaded from DIMACS10 Graph Challenge and the Stanford Network Analysis Project (SNAP). The graph information viz., number of nodes, edges and triangles were all picked from the SNAP and DIMACS.

Dataset Names	#nodes	#edges	#triangles
coAuthorsCiteseer	227,320	1,628,268	2,713,298
coPapersDBLP	540,486	30,491,458	444,095,058
road central	14,081,816	33,866,826	228,918
com-Orkut	3,072,441	234,370,166	627,584,181

Table 1: Test Graphs

Approach (VM-edge_intersection)

We tried solving this problem with a number of implementations. As we learnt from our assignments, it is better to start with a sequential implementation and then go ahead to find the scope of parallelism.

We first parallelized our implementation by iterating over the vertex and □ *(v1, v2)* in the neighbour list of vertex "*u",* we check whether *v2* is in the neighbor list of *v1*. This problem had a complexity of O(n^2) even in parallel solution and would result in large number of cache misses because every check of *v2* would be a cache miss. Moreover, it would do repetitive calculation, 3 times the actual work required to be precise (3 times because we have 3 vertices and we are finding the same set of 3 vertices for every vertex) thus increasing the total time of counting and even increase cache misses count in the repetitive work. Most of the time required in this approach was due to the memory reads. We called this approach ***vm-vertex_intersection***. This approach had a really poor performance even compared to the serial CPU baseline (refer table 2 in result analysis).

Final parallel approach (vm-edge_intersection) and optimizations:
We call our final approach ***vm-edge_intersection***. In this approach we iterate over the list of edges *(u, v)* and find all neighbors common to *u* and *v*. The count of common neighbors is this iteration is the number of triangles this *(u, v)* edge is part of of. A detailed description of the algorithm, the custom data structures created and the optimizations done in this final approach is as follow:

Input:
Undirected graph where each node is connected to other nodes by 2 edges (1 incoming and 1 outgoing)

Output:
Counts number of triangles in the given graph

Data structures:

NodeList	- Contains list of neighbours for each node. All list are stored one after the other in 1-d array
ListLen	- Contains number of neighbours for each node

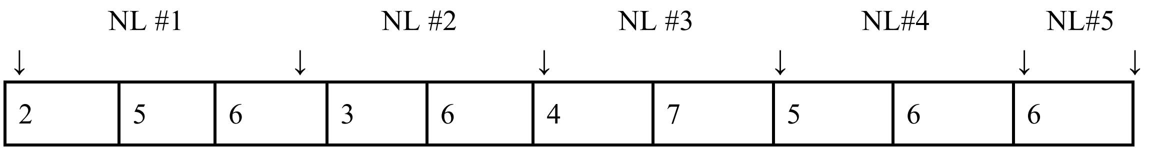


Figure 2: Contents of nodeList structure for graph given in figure1

NL #n = Neighbour list of node number "n"

Pseudo Code:

Step1: Create an array which stores the starting index of neighbour list of each node. This is done parallely by running inclusive sum over listLen array. In above example,
For Node #1 starting index is 0 and for node #2 starting index is 3

0	3	5	7	9	10	10	10

Figure 3: Contents of start_addr structure for graph given in figure1

Step2: Create list of all distinct edges from the nodeList. It is an undirected graph so, for any two connected nodes we have two edges (u, v) and (v, u). We will select only one edge out of the two where (u < v). This step can also be done parallely. To perform this step in parallel, we need to know which edge will go exactly where in the edgeList, so that there are no conflicts. To achieve this, we use the degree of each vertices to calculate a unique edge id for each edge.

(1, 2)	(1, 5)	(1, 6)	(2, 3)	(2, 6)	(3, 4)	(3, 7)	(4, 5)	(4, 6)	(5, 6)

Figure 4: Contents of edgeList structure for graph given in figure1

Step3: Apply intersection rule over edges to find if triangle is present. For every edge (u, v), we fetch neighbour list of both vertices "u" and "v". If there is any common vertex "w" their neighbour list then we count it as a triangle with edges {(u,v) (v,w) (u,w)}. This step is done in parallel for every edge and count of triangles formed by each edge is stored in an CountArray.

Let take an example of edge (1, 2):

Neighbour list of node 1 -> {2, 5, 6}
Neighbour list of node 2 -> {3, 6}

There is only common node in the neighbour list i.e. "6". There only one triangle can be formed through the edge (1, 2)

1	1	0	0	0	0	0	1	0	0

Figure 5: Contents of countArray structure for graph given in figure1

Step4: Reduction is applied on array obtained from last step. And it will give the total count of triangles present in the given graph. For the running example, count will come out to be 3.

Optimizations:

1) This problem is bandwidth bound. As we have to read all the edges and neighbour list for each node and there is not much computation to perform. Our main aim was to reduce the number of memory reads. So, we removed the duplicate edges from the data graph. The list

will now contain only those edges (u, v) where u < v. This will reduce the size of data structure by a factor of 2.

2) In neighbour list, we applied the same optimization. We are not storing the neighbouring nodes in the list where neighbour's node ID is less than current node ID. So, for any vertex "u", all the nodes present in its neighbour list have ID's greater than "u". We can do this optimization because for edge *(u,v)* □ *v < u* the edge is handled in the edglist of vertex "*v*". This also reduces the size of node list by a factor of 2.

3) Another optimization is done while designing the structure of nodeList. The most quick way to declare nodeList is to create an array of pointers of size "numNodes", where each pointer is pointing to its corresponding neighbour list. But this design of data structure is not efficient while moving nodeList data from host to device memory. Therefore, we decided to store nodeList data in 1-d array, so that we can directly move the whole data in 1 cudaMalloc and cudaMemcpy command thus saving cudaMalloc for every node. This approach requires extra calculation (calculating starting index of neighbour list for each node) but that is negligible as compared to doing multiple cudaMalloc and cudaMemcpy command.

Results

Our goal is to count the number of triangles as fast as possible and beat the results of the reference paper [1]. We ran our algorithms on the graphs mentioned in section 4, and have 2 baselines to compare our results from:

1. GPU baseline (reference paper [1]):

- ➢ Results mentioned in the reference paper were generated by running their Algorithm on **NVIDIA K40C**:
  - 6 GHz, 12 GB max memory, 2880 cores with 4.29 TFLOPS peak single precision
  - used CUDA

2. CPU baseline (reference paper [2]):

- ➢ Results mentioned in the reference paper were generated by running their Algorithm on **AMD Opteron Processors**:
  - clocked at 2.20GHz, max 6GB memory
  - gnu g++ compiler version 3.4 with Options "-g -O2"

Our implementation (vm-edge_intersection and vm-vertex_intersection):

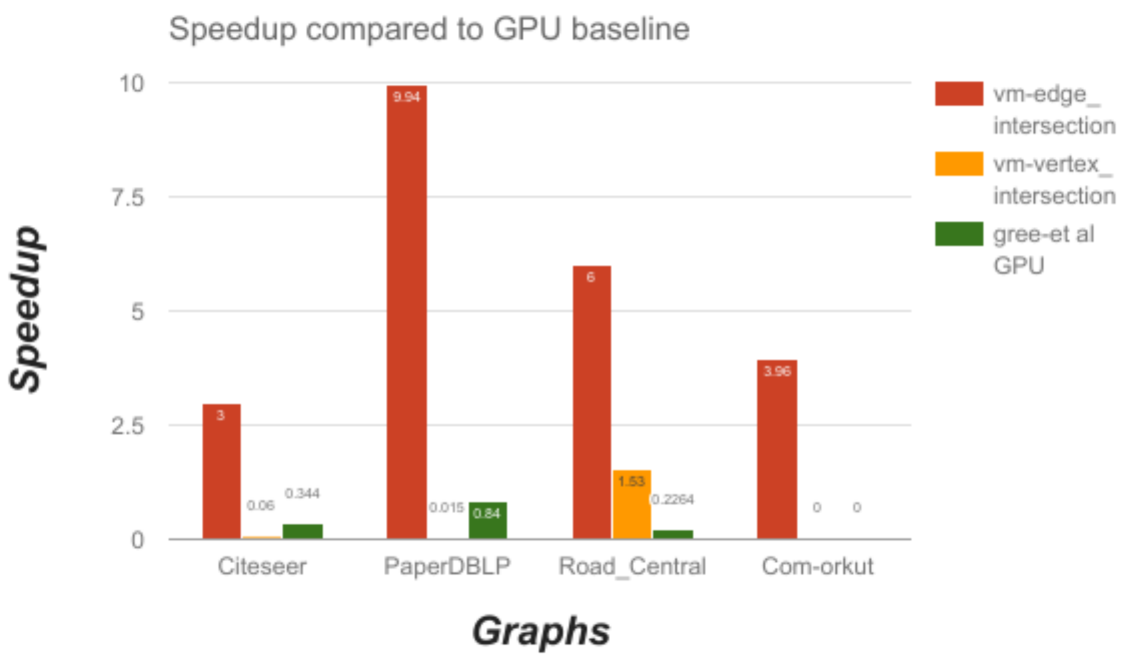
- ➢ Results reported using our implementation were generated by running our version of the algorithm on ghc46.ghc.andrew.cmu.edu with **NVIDIA GTX 1080**:

The results can be summarized as:

Algorithm	Citeseer	PaperDBLP	Road_Central	com-Orkut
CPU baseline	275 ms	100.654 s	3.254 s	1950.837 s
Tc-Intersection (GPU baseline)	4.5 ms	845 ms	60 ms	21.43 s
Green-et al GPU	13 ms	1.0 s	271 ms	-
Vm-vertex_ intersection	640 ms	52.74 s	40 ms	-
Vm-edge_ intersection	1.5 ms	85 ms	10 ms	5.4 s
Speedup compared to CPU baseline	**183x**	**1184.16x**	**325.4x**	**361.12x**
Speedup compared to GPU baseline	**3x**	**9.94x**	**6x**	**3.96x**

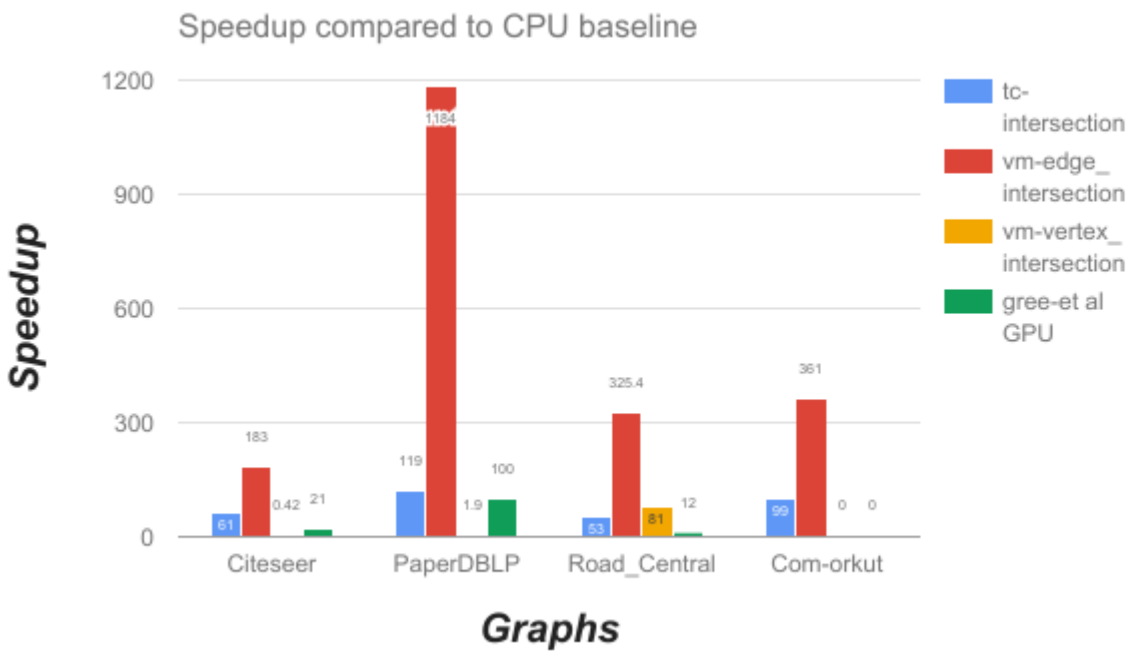
Table 2: Counting Triangle time and speedup summary

The speedup compared to the GPU baseline can be summarized as:



Our implementation performs best with an equal balance of #edges and #triangles and have managed to achieve a max speedup of 9.9x for coPapersDBLP.

The speedup compared to the CPU baseline can be summarized as:



*tc-Intersection mentioned in this graph is the GPU baseline.

Our GPU approach as expected achieves great speedup over CPU, with a max speedup of 1194 on coPapersDBLP

Comparison with Ligra:

As suggested by course staff, we are including our results with ligra as a reference point. The ligra standalone triangle counter was ran on the same ghc46.ghc.andrew.cmu.edu (8 core Intel Xeon CPU E5-1660 v4 @ 3.20GHz with NVIDIA GeForce GTX 1080). Ligra is minimal, parallel, and lightweight graph processing framework for shared memory; and our implementation of the algorithm for the GPU is better than ligra.

**Note: we were able to run the analysis only on com-orkut because that was the only graph we were successful in converting from SNAP to Ligra adjacency graph format.

Algorithm	com-Orkut
Ligra	63 s
Vm-edge_ intersection	5.4 s
Speedup compared to Ligra	**11.6x**

Table 3: Counting Triangle time and speedup compared to ligra implementation.

Performance analysis with workload balancing:

There is a problem of workload imbalance in our approach "**Vm-edge_intersection**". In our approach, every thread in a warp is traversing neighbour list of the vertices (end point of the given edge). Probability of workload imbalance increases with the increase in degree of the vertices. When the degree of graph is large then there is possibility that some of the neighbour lists are small and some are very large. In this case, threads in a warp which get small list will finish up earlier and remain idle till other threads are finished.

To solve this problem, we divided the edge lists in two groups:

1) Small neighbour list
2) Large neighbour list

To divide the workload between these two categories, we need to choose a threshold value of neighbour list length. If list length is above threshold value, it will go in large list category. We implemented two kernels to compute intersection between edges in the above two categories. By using this **2-kernel strategy** and carefully choosing the threshold value, we can process the edges with equal workload in same thread block. And this in turn will reduce the idle time of threads.

To prove our theory we did following experiments:

1) Workload balancing on a graph where graph degree is very less and workload imbalance is not an issue. By observing the results obtained from the experiment, we can see that with different threshold values both kernel are getting different number of edges. However, there is no change in computation time as the high workload imbalance is not there.

	Threshold value				
	1	2	4	6	8
# small edges	4226987	10959922	16920991	16933367	16933413
# large edges	12706426	5973491	12422	50	0
Vm-edge_ intersection	25 ms	25 ms	25 ms	25 ms	25 ms

Table: Time taken to count triangles in graph "road_central" with different threshold values

2) Workload balancing on a graph where graph degree is large and workload imbalance is an issue. By observing the results obtained from the experiment, we can see that with different threshold values both kernel are getting different number of edges. And there is a significant change in computation time with changes in threshold value.

	Threshold value				
	50	100	200	2000	5000
# small edges	5362484	9427119	14039668	15242430	15245729
# large edges	9883245	5818610	1206061	3299	0
Vm-edge_ intersection	120 ms	85 ms	120 ms	110 ms	100 ms

Table : Time taken to count triangles in graph “coPaperDBLP” with different threshold values

There are two important things to observe here:

1) With threshold value of 100, we are getting best result of 85ms. And as expected, when we increase or decrease the threshold value, workload imbalance increases and thus increase the computation time.
2) But there is a contrasting behavior, when we increase the threshold from 200 to 5000, the computation time decreases even with the increase in work imbalance. This is because of the cache utilization. When we compute all the edges in the same kernel, our cache utilization increases because our edges are stored in sorted order. And thus, neighbour list is fetched once and for other times we get cache hits which helps in reducing the computation time.

From above experiments, we learned that we cannot achieve both workload balancing and cache utilization at the same time. So, depending upon the degree of graph we should decide which approach to choose. Graphs where difference in length of neighbour lists is not much, then we should not opt for workload balancing because it will cause more overhead with no benefit. Balancing should be used only when there is a huge problem of workload imbalance and when the benefits from workload balance can overcome the decrease in performance due to increase in cache misses.

Work Distribution

Equal work was performed by both team members

References

[1] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D. Owens, A Comparative Study on Exact Triangle Counting Algorithms on the GPU. In *Proceedings of the ACM Workshop on High Performance Graph Processing Pages 1-8*, 2016

[2] O. Green, P. Yalamanchili, and L.-M. Mungua. Fast triangle counting on the GPU. In Proceedings of the Fourth Workshop on Irregular Applications: Architectures and Algorithms, IA3 '14, pages 1-8, 2014.

[3] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an

experimental study. In Proceedings of the 4th International Conference on Experimental and Ecient Algorithms, WEA'05, pages 606-609, 2005

[4] Ligra: https://github.com/jshun/ligra

[5] Julian Shun and Guy Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 135-146, 2013.

[6] Julian Shun, Farbod Roosta-Khorasani, Kimon Fountoulakis and Michael Mahoney. Parallel Local Graph Clustering. Proceedings of the International Conference on Very Large Data Bases (VLDB), 2016.