Team: Neelesh Gupta, Prasanna Akolkar

EE451 Project Report

Accelerating Approximate Triangle Counting in Sparse Graphs

Introduction:

Triangle counting in a graph is a major concept in graph theory and network analysis. It involves the identification and counting of triangles within a graph. A triangle is defined as a set of three vertices connected to each other, forming a closed loop. There has been significant research in the field of triangle counting algorithms. The applications of counting triangles are diverse, including spam detection by measuring the clustering coefficient, social network analysis, and common topic finding, among others.

In our project, we focus on the approximate counting of triangles in sparse graphs. We define a graph as sparse when the number of edges is significantly less compared to the number of vertices.

This concept can be represented by the following equation:

$$D = \frac{|E|}{\sum_{i=1}^{|V|} i} = \frac{|E|}{\frac{|V|*(|V|-1)}{2}} = \frac{2|E|}{|V| * (|V| - 1)}$$

Fig. 1 Density equation for a graph with |V| vertices and |E| edges

Lower the value of D, sparser will be the graph.

In this project, we have worked on:

- Implementing the serial version of Triangle Counting and Approximate Triangle Counting.
- Implementing the parallel version of Triangle Counting and Approximate Triangle Counting using CUDA.
- Implementing the DOULION algorithm for approximation of the number of triangles.
- Analyzing the accuracy and speedup based on various levels of edge sparsity (or sparsity probability).
- Performing GPU profiling to analyze the L1 and L2 cache utilization in both sparse and dense graphs.

Algorithms

During the project, three different exact triangle counting methods and one approximation method were analyzed and developed. These triangle counting algorithms vary in complexity.

```
1: function AllTriplets(G)
2:     triangles ← 0
3:     for all v ∈ V do
4:         for all u ∈ V where u ≠ v do
5:             for all w ∈ V where w ≠ v and w ≠ u do
6:                 if (v, u) ∈ E and (u, w) ∈ E and (w, v) ∈ E then
7:                     triangles ← triangles + 1
8:                 end if
9:             end for
10:        end for
11:    end for
12:    return triangles/6
13: end function
```

Fig. 2.1 Naive Algorithm for Exact Triangle Counting

In Fig. 2.1, the algorithm for triangle counting iterates over the triplets (u, v, and w) in a nested loop, incrementing the triangle count if there is an edge between each pair of u, v, and w. The major drawbacks of this method are its execution time, which is O(|V|^3), and the issue of duplicate triangle counting, which requires normalization at the end of the function, due to which it is not generally used for counting triangles in a graph.

```
1: function NodeIterator(G)
2:     triangles ← 0
3:     for all v ∈ V do
4:         neighbors_v ← neighbors of v
5:         if |neighbors_v| > 1 then
6:             for all pairs (u, w) in combinations(neighbors_v, 2) do
7:                 if u is neighbor of w then
8:                     triangles ← triangles + 1
9:                 end if
10:            end for
11:        end if
12:    end for
13:    return triangles/3
14: end function
```

Fig. 2.2 Node-Iterator Approach for Exact Triangle Counting

Fig. 2.2 represents the Node-iterator algorithm that outperforms the naive triangle counting approach. Instead of iterating over all neighbors of a vertex v, this algorithm iterates only over the neighbors of those vertices whose degree is greater than or equal to 2. If v has two or more neighbors, the algorithm iterates over all pairs of v's neighbors (u, w). If there is an edge between u and w, the triangle count is incremented. This approach also leads to duplicate counting of triangles, necessitating normalization at the end of the function by dividing the final count by 3. The complexity of this algorithm is O(|V|*d^2), where d is the average degree of all vertices in V.

```
1: function CompactForward(G)
2:     trianges ← 0
3:     vertices ← V sorted by decreasing in-degree ∀v ∈ V
4:     for v in vertices do
5:         neighbors ← sorted neighbors of v
6:         for u in filtered neighbors with lower index than v do
7:             triangles ← triangles + CountCommonNeighbors(u, v)
8:         end for
9:     end for
10:    return triangles
11: end function
```

Fig. 2.3.1 Compact Forward Approach for Exact Triangle Counting

```
function CountCommonNeighbors(u, v)
    count ← 0
    Initialize counters for neighbors of u and v
    while counters not at the end of their lists do
        if current u neighbor equals current v neighbor then
            count ← count + 1
            Move to next neighbor in both lists
        else if current u neighbor has higher index then
            Move to next neighbor in u's list
        else
            Move to next neighbor in v's list
        end if
    end while
    return count
end function
```

Fig. 2.3.2 Count Common Neighbours Function for Compact Forward Approach

Fig. 2.3.1 illustrates the Compact Forward Approach, an efficient method for counting triangles by utilizing sorted vertices, thereby eliminating redundant checks and avoiding duplicate triangle counting. In this approach, we iterate over the vertices v in descending order of their values. For each vertex v, we examine its sorted neighbors (referred to as u) and consider only those u where the index is lower than v. This is because u with higher indices would have already been accounted for in the earlier iterations of higher-indexed vertices. In the Count Common Neighbours function, we check for a common edge between v and u and increment the triangle count accordingly. The complexity of this algorithm is O(|V|log|V| + |V|*d^2), where d is the average degree of all vertices in V.

All the algorithms discussed above are exact triangle counting algorithms. These exact approaches can be computationally intensive, which is why approximation methods are sometimes used for counting. One such approach explored during the project is Doulion's method for Approximate Triangle Counting (ATC).

```
function Doulion(G, p)
    G' ← SparsifyGraph(G, p)
    triangles_est ← ExactTriangleCount(G')
    return triangles_est / p^3
end function
```

Fig. 2.4.1 Doulion Approach for ATC

```
1: function SparsifyGraph(G, p)
2:     G' ← new empty graph
3:     for all e ∈ E do
4:         if RAND(0, 1) ≤ p then
5:             add edge e to G'
6:         end if
7:     end for
8:     return G'
9: end function
```

Fig. 2.4.2 Sparsify Graph for Doulion Approach

Fig 2.4.1 depicts the Doulion Approach for Approximate Triangle Counting (ATC), in which a graph is sparsified based on the probability p, representing the fraction of edges to be retained. In the 'SparsifyGraph' function, a random value between 0 and 1 is generated for each edge in the original graph. An edge is preserved in the sparsified graph if its corresponding random value is less than or equal to p; otherwise, the edge is discarded. The resulting sparsified graph, obtained at the end of the function, will then be used as input for the Exact Triangle Counting Function. Since the graph is missing edges, the final count of triangles will have to be normalized by dividing the value by p^3 (since the probability of selecting each edge of a triangle is p).

Given density D and average degree d of G and percent edges sparsified p, sparsifying G to G' gives density D' = p*D and average degree d' = p*d. The complexity of the approach is O(|E| + runtime of ExactTriangleCount).

In the context of triangle counting, sparsifying the graph can significantly reduce the number of edges to consider, which can lead to faster computations. However, since this is a probabilistic method, the resulting triangle count after sparsification is an approximation of the true number of triangles in the original graph. The Doulion approach is typically used when an exact count has a priority lower than the speed of the computation.

Hypothesis

Our hypothesis contends that for a graph G (V,E), the approximate triangle count (ATC) approach's performance is dependent on the percent edges sparsified p. As p tends towards the

value of 0, we anticipate a decline in the accuracy of the triangle count, albeit with an improvement in execution speed. Conversely, as p approaches 1, we expect an increase in the accuracy of triangle counting, but this comes at the expense of slower execution speed. Hence, edge sparsification presents a trade-off between accuracy and execution time.

Furthermore, we anticipate that inducing sparsity in a graph will result in lower cache utilization compared to that of a denser graph, potentially affecting the efficiency of the computation.

Experimental Setup

Datasets:

The experimental setup involves datasets from the Suite Sparse Matrix Collection.

Datasets	Vertices	Edges	Density (%)
ak2010	45,292	217,098	0.021167%
asia_osm	11,950,757	25,423,206	0.000036%
belgium_osm	1,441,295	3,099,940	0.000298%
coAuthorsDBLP	299,067	1,955,352	0.004372%
delaunay_n13	8,192	49,094	0.146329%
delaunay_n21	2,097,152	12,582,816	0.000572%
delaunay_n24	16,777,216	100,663,202	0.000072%
hollywood-2009	113,891,327	1,139,905	0.000000%
kron_g500-logn21	1,048,576	89,239,674	0.016233%
road_central	14,081,816	33,866,826	0.000034%
road_usa	23,947,347	57,708,624	0.000020%

Fig. 2.5 Datasets for ATC

	triangles per edge	triangles per node
ak2010 runtime	0.8071193654	3.868762695
asia_osm runtime	0.0009368212648	0.001992928147
belgium_osm runtime	0.002341980813	0.005037136742
coAuthorsDBLP runtime	3.219393235	21.04895224
delaunay_n13 runtime	1.004725628	6.021240234
delaunay_n21 runtime	1.005749269	6.034449577
delaunay_n24 runtime	1.005730078	6.034374833
hollywood-2009 runtime	12938.90602	129.5017281
kron_g500-logn21 runtim	296.358647	25221.77605
road_usa	0.02281135658	0.05497109972
road_central	0.02027807389	0.04876885197

Fig. 2.6 Triangle Density for Datasets

Serial Implementation:

For the serial implementation of the Approximate Triangle Counting (ATC) using the Node Iterator Approach, we utilized a system equipped with an AMD Ryzen 7 3700X CPU, which features 8 cores and 16 threads, complemented by 16 GB of DDR4 RAM.

Parallel Implementation:

For the parallel implementation of the Approximate Triangle Counting (ATC) using the Compact Forward Approach, the system with an Nvidia GeForce RTX 2070 SUPER graphics card, which boasts 8 GB of VRAM and the GPU memory profiling was conducted using CUPTI and NVBench.

Results and Analysis

Evaluation of Speedup vs Percent Edges Sparsified p

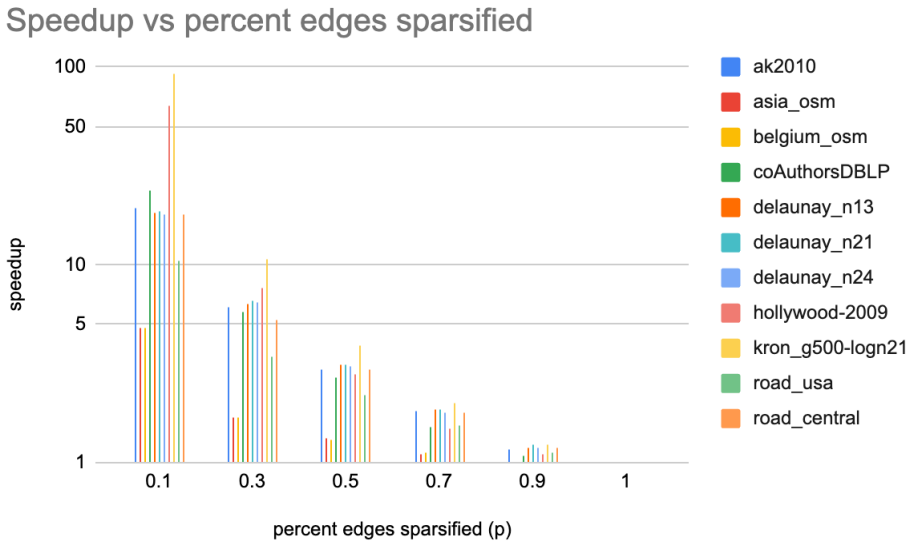

Fig. 3.1 Plot for speedup vs p on CPU

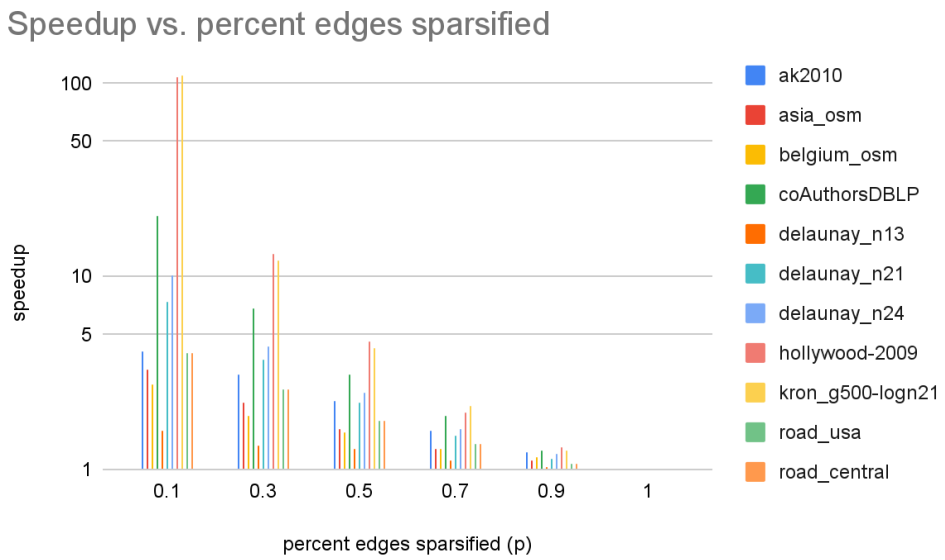

Fig. 3.2 Plot for speedup vs p on GPU (using CUDA)

The speedup is obtained by: $\frac{Time\ Taken\ for\ Exact\ Triangle\ Count\ on\ CPU/GPU}{Time\ taken\ for\ ATC\ on\ CPU/GPU}$

Evaluation of Accuracy Error vs Percent Edges Sparsified p

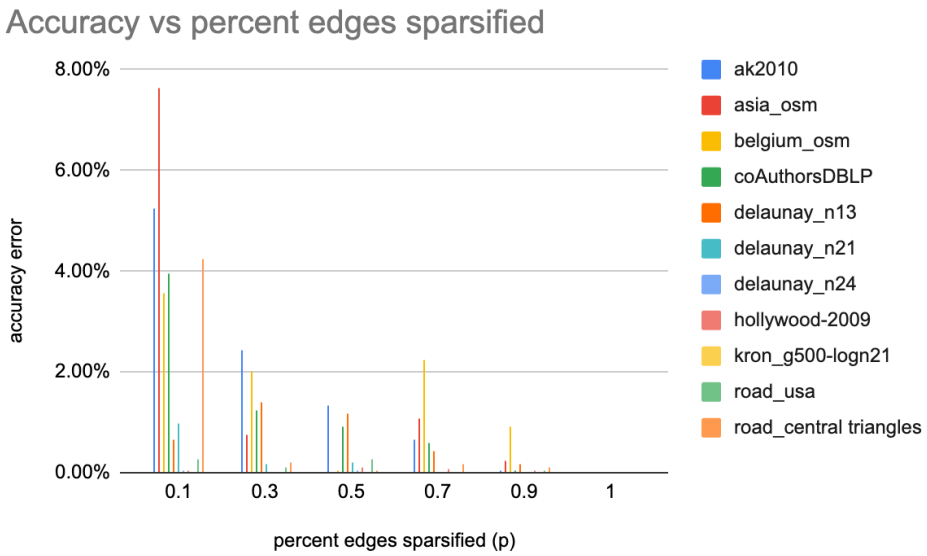

Fig. 3.3 Plot for accuracy error vs p on CPU

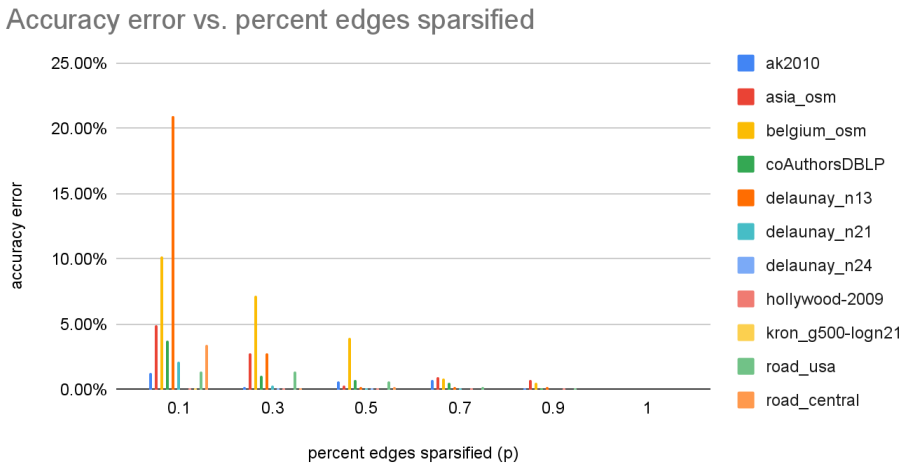

Fig. 3.4 Plot for accuracy error vs p on GPU (using CUDA)

The accuracy can be obtained by: $$\frac{|Exact\ Triangle\ Count - Approximate\ Triangle\ Count| * 100}{Exact\ Triangle\ Count}$$

GPU Memory Profiling

dataset	nodes	edges	density	triangles	triangles per edge	triangles per node
belgium_osm	1,441,295	3,099,940	0.000002984542894	7260	0.002341980813	0.005037136742
delaunay_n13	8,192	49,094	0.00146329384	49326	1.004725628	6.021240234

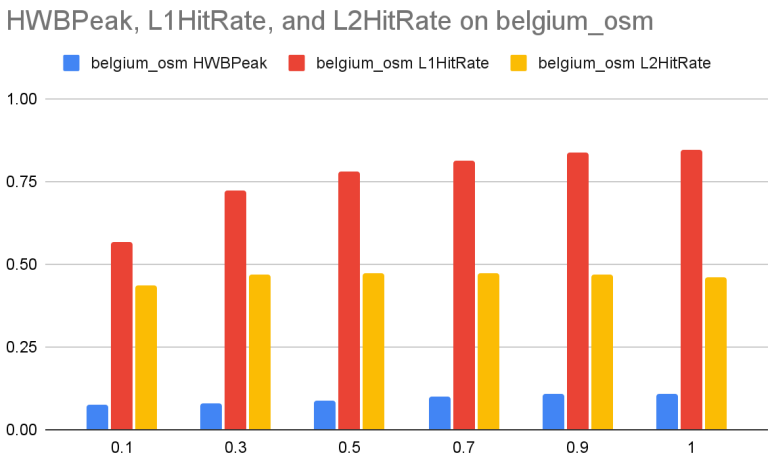

Fig. 3.5 HWBPeak, L1 cache Hit Rate and L2 cache Hit Rate on a sparse graph (belgium osm)

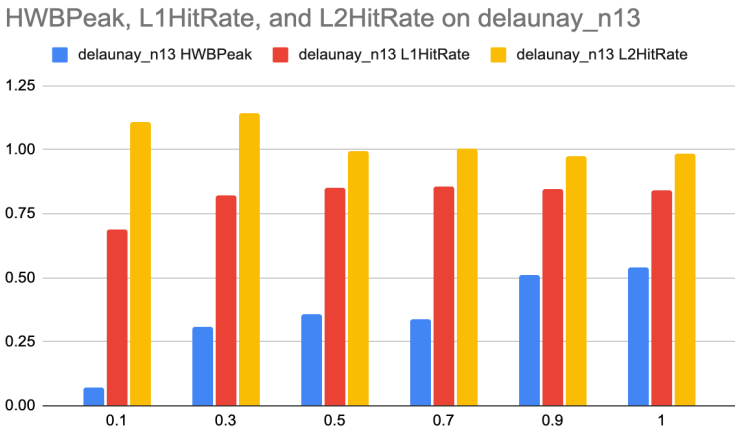

Fig. 3.6 HWBPeak, L1 cache Hit Rate and L2 cache Hit Rate on a denser graph (delaunay_n13)

Conclusion

In this project focused on Approximate Triangle Counting (ATC) in Sparse Graphs, we have noted that there is a clear trade-off between the speed of computation and the accuracy of the triangle count. This balance largely hinges on the value of the percent edges sparsified, denoted by p. Furthermore, we have observed that graphs with a higher density of triangles are more amenable to ATC, especially when employing a larger value of p, yielding faster and more precise results in comparison to graphs with a sparser triangle distribution. Utilizing GPU profiling, we have determined that sparser graphs result in lower cache utilization relative to denser graphs. This finding implies that denser graphs are better suited for approximate triangle counting.

References

[1] Wang, Y., Pan, Y., Davidson, A., Wu, Y., Yang, C., Wang, L., Osama, M., Yuan, C., Liu, W., Riffel, A. T., & Owens, J. D. (2017). Gunrock: GPU Graph Analytics. ACM Transactions on Parallel Computing, 4(1), 3:1-3:49. doi: 10.1145/3108140

[2] Tsourakakis, C.E., Kang, U, Miller, G.L., Faloutsos, C.: Doulion: Counting Triangles in Massive Graphs with a Coin. Proceedings of ACM KDD, 2009

[3] Tsourakakis, Charalampos E. "Fast Counting of Triangles in Large Real Networks: Algorithms and Laws." Machine Learning Department. Carnegie Mellon University.