15-618 Project Final Report : Parallel Triangle Enumeration using Linear Algebra

Group 16 : Rini Patel (rinip), Tushar Goyal (tgoyal1)

Summary

We implemented triangle enumeration by extending the linear algebra based triangle counting algorithm [1] and parallelized it for multicore CPUs and NVIDIA GTX 1080 GPU. We present the parallel triangle enumeration results with OpenMP and CUDA implementation using Graph Challenge dataset [2].

Background

Triangle counting is key idea in understanding the clustering of the graph and clustering coefficient [3]. Once triangles are enumerated, the output can be used to calculate local (per-vertex) clustering coefficients. The clustering coefficient helps in determining how close a community is or how dense host internet network at particular area is. [4]. A closely knit community is most likely to have unusually high degrees of “trust”, since lots of your other friends are likely to hear about it. The latter is quite useful in designing the routers and making effective decisions regarding hardware.

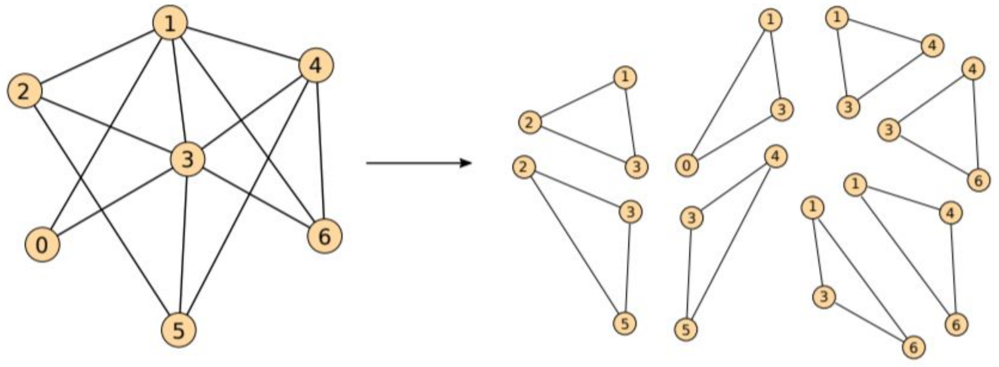


Figure 1. Example triangle counting and enumeration [5]

The time complexity of the simple triangle counting or listing algorithm is $O(n^3)$ where ‘n’ is number of vertices in the graph since it involves three matrix multiplications [6], [7]. Many efforts have been made to reduce this. We extend the algorithm in [1] to do triangle enumeration. The original algorithm does the triangle counting by exploiting loop invariants in problem to avoid matrix multiplication and thus reduce the time complexity to $O(n^2)$ in worst case.

Baseline Algorithm

The algorithm takes in CSR (Compressed Sparse Row) format of graph as an input and iteratively moves one vertex at a time from bottom right region (V_{BR}) of the graph to top left (V_{TL}) as shown in Figure 2. It counts the number of triangles in undirected graph G by starting with all vertices in V_{BR}. When one vertex v is moved from vertex set V_{BR} to V_{TL}, we add the number of triangles now having two vertice in V_{TL} and one vertex in V_{BR} to triangle count Δ. When all vertices in V_{BR} are moved to V_{TL}, Δ would hold the number of triangles in G.

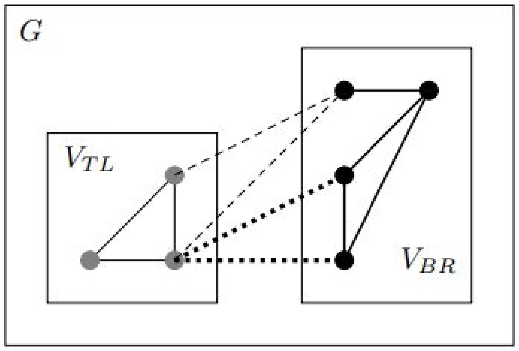


Figure 2. Vertices are split into two disjoint subsets V_{TL} and V_{BR}, where gray vertices belong to V_{TL} and black vertices belong to V_{BR}. Dashed lines form part of a triangle with two vertices in V_{TL} and one vertex in V_{BR}, while dotted lines form part of a triangle with two vertices in V_{BR} and one vertex in V_{TL}.

Initial triangle count is zero in top left and as iteration advances, net change in triangle count is calculated as vector inner product with the selected rows of the partitioned matrix. The modified algorithm outputs the number of triangles along with the vertex ID for each triangle enumerated.

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c} A_{00} & (a_{01}|A_{02}) \\ \hline \left(\begin{array}{c} a_{10}^T \\ \hline A_{20} \end{array}\right) & \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) \end{array}\right) \rightarrow \left(\begin{array}{c||c} \left(\begin{array}{c|c} A_{00} & a_{01} \\ \hline a_{10}^T & \alpha_{11} \end{array}\right) & \left(\begin{array}{c} A_{02} \\ \hline a_{12}^T \end{array}\right) \\ \hline (A_{20}|a_{21}) & A_{22} \end{array}\right) \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$$

The formula for computing Δ on each iteration is given from above loop induction method as:

$$\Delta = \Delta + a^T{}_{12}A_{20}a_{01}$$

Workload and Computation Analysis

The computational overhead comes from the nature of computation i.e. vector-matrix inner product part ($a^T{}_{12}A_{20}a_{01}$), and the amount of work that needs be done for real graphs having 10M-15M edges and 3M nodes. The locality of computation comes by selecting non-zero rows

A_{20} and computing the inner product with column vector a_{01}. Thus, symmetric nature of adjacency matrix gives sequential accesses of A_{20} and a_{01} in CSR format.

The scope of parallelism comes from the independent computation done for each vertex (i.e. computing the inner product A_{20} with column vector a_{01}). However, as we discovered over the course of our project, the connectivity at different vertices is very sparse and highly irregular thus hindering naive parallelization on basis of vertex and causing workload imbalance. Moreover, we saw that the due to different computation requirement at different vertices, algorithm exhibits high divergence.

Our Approach

Sequential Implementation

We took off by implementing the baseline sequential version presented in paper (which took us some time to understand), and improving it by refactoring the code to get on average 10% speedup on selected benchmarks. We also modified it for storing the enumerated triangle vertices in global output buffer, which can be optionally dumped to disk file (Checkpoint I). See the Graph 1 in results section, the red line presents the speedup of improved sequential over the original implementation presented in paper.

OpenMP Implementation

Algorithm 1 : OpenMP pseudo-code

```
Input : Graph $G(V, E)$, with $N$ number of vertices, $p$ threads
Output : number of triangles, enumerated list of triangles
for $i \leftarrow 1$ to $N - 2$ do in parallel
|    nnz_x = getNonzeroRowsA_{20}(i); # Find the bounds for current vertex
|    nnz_y = getNonzeroColumnsA_{20}(i);
|    for $j \leftarrow 0$ to nnz_y do
|    |    for $k \leftarrow 0$ to nnz_x do
|    |    |    $\Delta$ += getVertexIntersection(j, k);
|    |_   |_   updateLocalTriangleList(thr_id);
|_   localTriangleCount[thr_id] = $\Delta$
globalIdx[] = prefixSum(localTriangleCount);
for $i \leftarrow$ to $p$ do in parallel
|_   globalList[ globalIdx[thr_id] ] = localTriangleList[thr_id];
```

Parallel triangle counting implementation discussed in [1] using OpenMP was trivial because of *reduction(+:num_triangle)*. However, to enumerate the triangles in parallel manner, using atomic directives resulted in high synchronization overhead along with high amounts of false sharing and we got very little speedup.

We used the knowledge derived from assignment 2 and assignment 3 to solve this. We used private thread local arrays for each thread to store vertices list and total triangles per thread, and one global array where each thread would accumulate their respective total number of triangles according to prefix sum index. Thus, now we have less contention for accumulating global triangle counts and to list the vertices of triangles enumerated by threads.

Experimental Evaluation

Parallel Prefix-sum Pitfall: We realised that use of parallel OpenMP implementation for prefix scan of 16 elements is causing us a lot of overhead. We then switched to naive sequential prefix sum implementation and that gave us a significant improvement (+0.83x) in terms of speedup with respect to sequential baseline.

Choice of OpenMP Scheduling: The initial scheduling that we used was static scheduling with default chunk size (loop_count/number_of_threads). This initial approach gave us average speedup of 3.05x (Max speedup = 6.40x, Min speedup = 1.06x). We tried out several strategies for tweaking the scheduling to better suit the application workload. We compared static scheduling with 2 scheduling approaches:

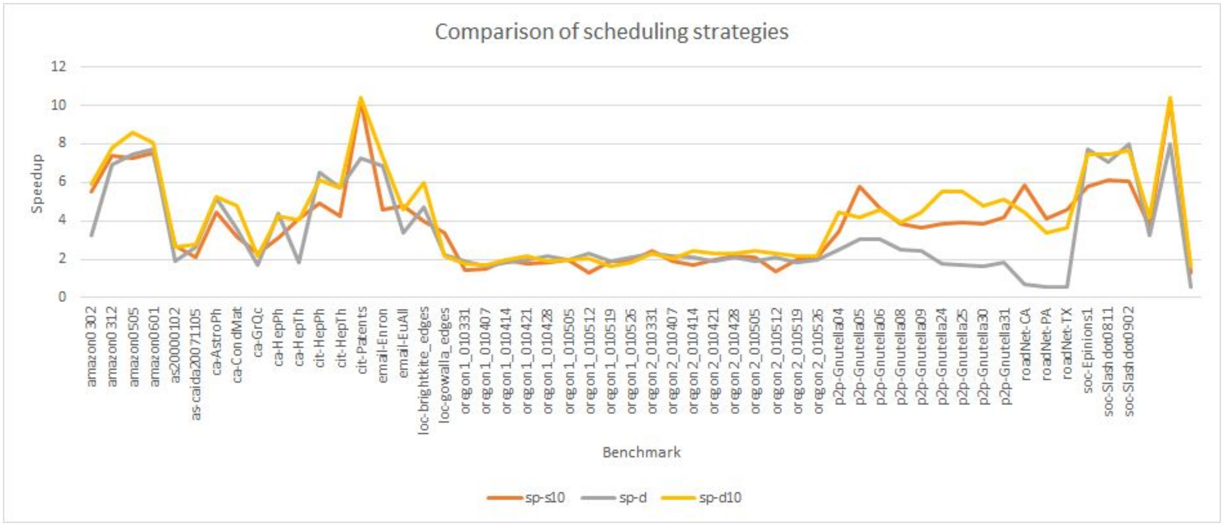


Figure 3. Comparison of speedup with different scheduling strategies with OpenMP

However, from analysis of the benchmarks, it is seen that due to the sparsity of graph and variation in degree of connectivity across vertices, few benchmarks were performing consistently well with dynamic scheduling and few of the benchmark had the degradation in performance due to dynamic scheduling overhead.

Scheduling Type	Chunk Size	Avg. Speedup	Max Speedup	Min Speedup
Static	10	3.69x	10.25x	1.31x
Dynamic	1 (default)	3.22x	7.96x	0.59x

This led us to mix and match the two approaches and try out dynamic scheduling with chunk size of 10, which allows the dynamic scheduling switch to happen less frequently and reduce overhead caused by dynamic workload distribution. The results were very interesting and we saw that speedup achieved was 4.57x (+1.5x than before). This way of scheduling makes sure that workload for each thread is equal and maximum throughput is achieved.

CUDA Implementation

In our naive CUDA implementation in Checkpoint II, we were spawning total threads equal to number of vertices such that each thread was responsible for computing triangles for one vertex of the graph. So basically the outer parallel for loop in Algorithm 1 becomes the kernel. This way of dividing work resulted in very poor workload partition and for some benchmarks we got ~17,000 triangles computed by one thread and 0 by others. This explains the enormous slowdown we got on computation part of CUDA. Also, the overhead of transferring IA and JA values of Compressed Sparse Matrix from device to host is significant, and we believe is the bottleneck of overall performance.

Experimental Evaluation

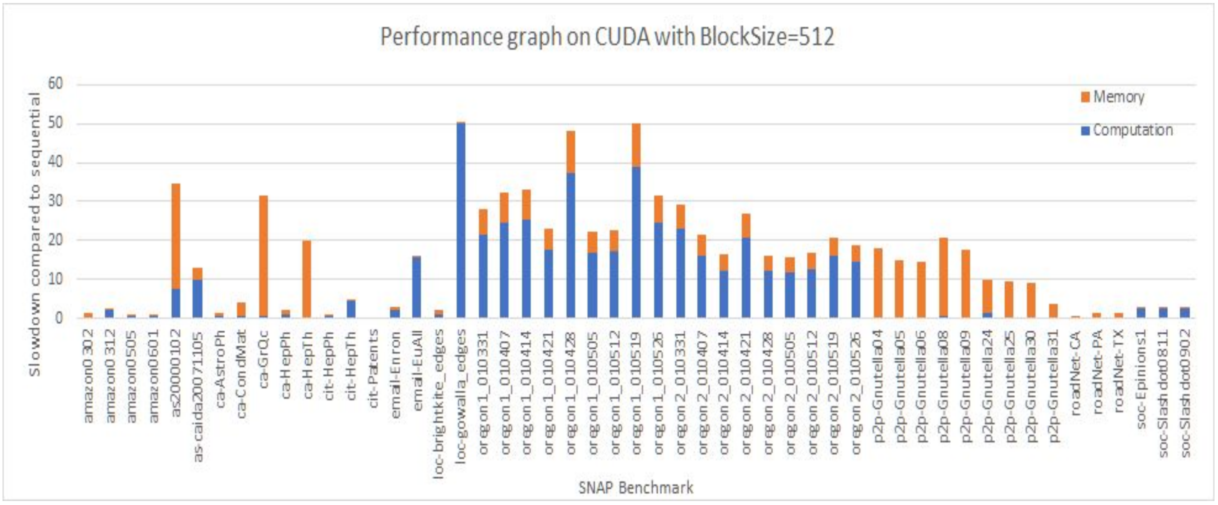


Figure 4. Performance degradation with naive CUDA implementation compared to sequential version

Naive Method Slowdown: Graph represents the slowdown as respect to sequential code. As seen from the performance graph, Blue part indicates the computation part slowdown with respect to sequential and Orange part is the memory overhead. The total bar indicates the overall slowdown. This trend concurs our theory about poor workload distribution and memory being the bottleneck. With this way of partitioning work, we did not get any speedup on CUDA. So, after Checkpoint II, our main focus was to work on the CUDA performance.

To better divide the work amongst CUDA threads, we first tried sorting the vertices based on the degree of connectivity. Heuristic was that if we perform sorting, and then launch the kernels, each warp scheduled by GPU will have around equal work to do. But this strategy surprisingly did not result in any better performance than previous one.

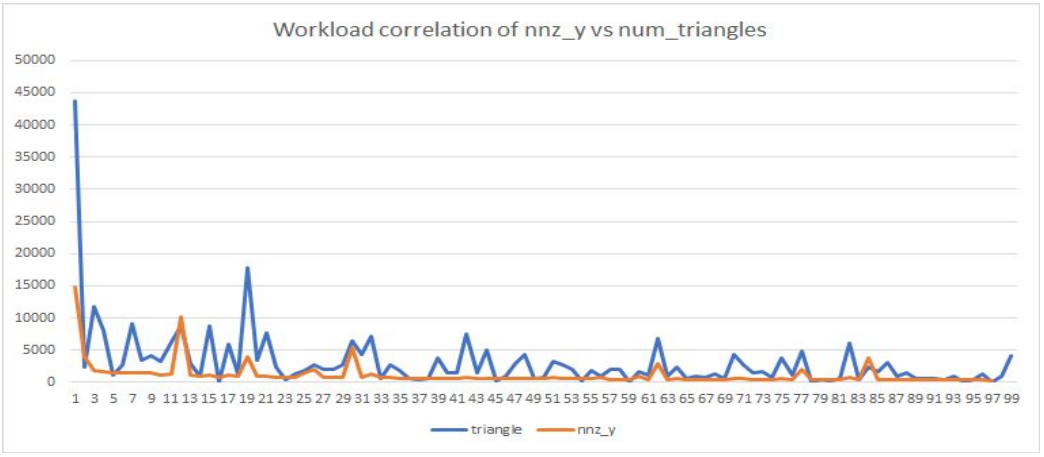


Figure 5. Workload correlation for triangles enumerated by a vertice and its nnz_y value

Better Work Division Approach: After that, we resorted to completely different scheduling approach shown in Figure 6(a). Say you have 5 vertices in a graph, so instead of launching 1 CUDA threads for 1 vertex as shown on the left side, we initially launch 5 threads. Now all threads are responsible for spawning more threads on the fly when they encounter workload higher than some threshold, and each child thread will spawn another children if required following the same heuristic.

We ran some experiments to come up with the good threshold for workload that each thread should have by weighing the bounds of inner loops, and finally decided to go with nnz_y values as a good estimation of workload as they correlate well with number of triangles calculated for each vertex as shown in Figure 5. So we do the pre-estimation of workload at the beginning of each kernel and flood the GPU by launching nested kernels as and when workload for that kernel exceeds the predefined threshold value (currently using nnz_y > 64). This results in work division for each thread somewhat like in Figure 6(b), and we got very good speedup as compared to out naive implementation earlier. (Please see Results section)

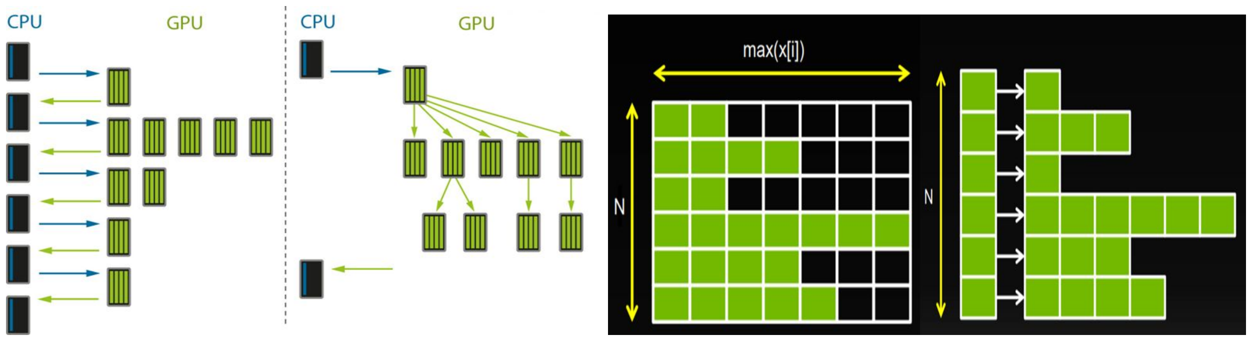


Figure 6. (a) CUDA thread launching strategy naive vs new. (b) Workload distribution with new strategy, each thread will manage to get equal amount of work.

Algorithm 2 : CUDA pseudo-code

```
Input : Graph $G(V, E)$, with $N$ number of vertices
Output : number of triangles
Launch parent kernel with $i \leftarrow 1$ to $N - 2$ threads
|    nnz_x = getNonzeroRowsA$_{20}$($i$); # Find the bounds for current vertex
|    nnz_y = getNonzeroColumnsA$_{20}$($i$);
|    if nnz_y > 64
|    |    Launch child kernel with $i' \leftarrow 0$ to nnz_y threads
|    |    |    for $k \leftarrow 0$ to nnz_x do
|    |    |_   |_   childTriangleCount[child_thr_id] += getVertexIntersection($i'$, $k$);
|    |    cudaDeviceSynchronize();
|    |_   localTriangleCount[thr_id]  = thrust::reduce(childTriangleCount);
|    else
|    |    for $j \leftarrow 0$ to nnz_y do
|    |    |    for $k \leftarrow 0$ to nnz_x do
|    |    |_   |_   $\Delta$ += getVertexIntersection($j$, $k$);
|_   |_   localTriangleCount[thr_id] = $\Delta$;
cudaThreadSynchronize();
total_triangles = thrust::reduce(localTriangleCount);
```

We use shared memory to avoid contention on global buffer for each thread within the block to store their individual triangle counts and use Thrust library reduce operation to accumulate per-block triangle count into global memory. These block triangle counts are also partial, and accumulated after kernel returns into one single variable.

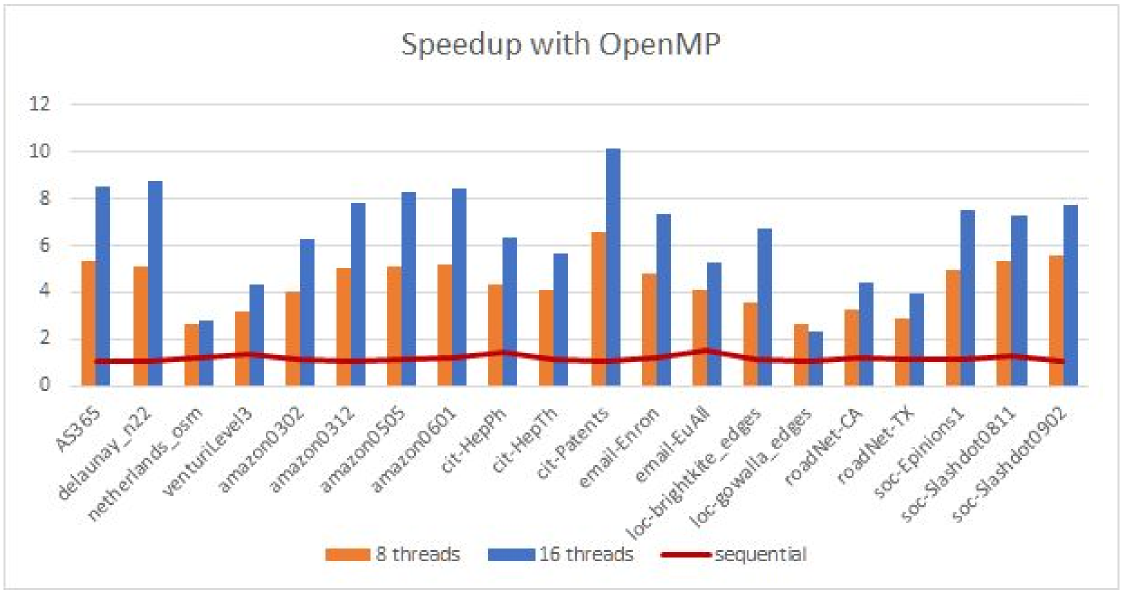
Results

Machine
GHC cluster (Intel(R) Xeon(R) CPU E5-1660 v4 @ 3.20GHz) 8-cores, 2-way hyperthreading for the OpenMP implementation, and NVIDIA GTX 1080 for CUDA.

Benchmark
All the graphs used for experimental evaluations are taken from SNAP dataset for Large Networks and DIMACS10 dataset. They are preprocessed using Python script to be converted into CSR format for the input to our algorithm.

Performance metric
Our performance measurement metric for speedup was timing. We used CycleTimer from assignment 2 to get the timing info for each part of the code. For the code running on GPU, we also used NVIDIA profiler to help us better understand where most of the cycles are going.

OpenMP Implementation
The graph here shows the speedup of OpenMP parallel version of triangle enumeration with 8 threads and 16 threads with respect to the optimized sequential implement. (Also, the red line is the speedup of optimized sequential implementation with respect to original algorithm in paper). As discussed in previous section, the scheduling type we used is OpenMP dynamic scheduling with chunk size of 10 so as to reduce the frequency of dynamic workload balancing overhead.



Graph 1. Speedup results of OpenMP implementation

We found our implementation to be working well with achieving a maximum 10.16x speedup with an average speedup of 4.57x across all the SNAP benchmarks. The lower speedups of 1.4x-2x are seen on relatively smaller chunk of graphs in dataset with approximately 10k nodes and 50000 edges.
Comparing the performance on 8 threads vs 16 threads, it can be seen that having 16 threads gives us more speedup. Since the CPU is having 8 physical cores, it tells us that computational part of algorithm is still not optimized enough to saturate the functional units of 8 cores. This brings forth two possible avenues why we are not getting the peak throughput for the CPU:

1. The time to dynamically schedule some of the workloads and divergent nature of computation causes physical cores to switch to hyperthread counterparts.
2. Due to large nature of graphs, the data never fits in on-chip SRAM cache and hence, processor needs to switch between hyperthreads to hide the memory latency.

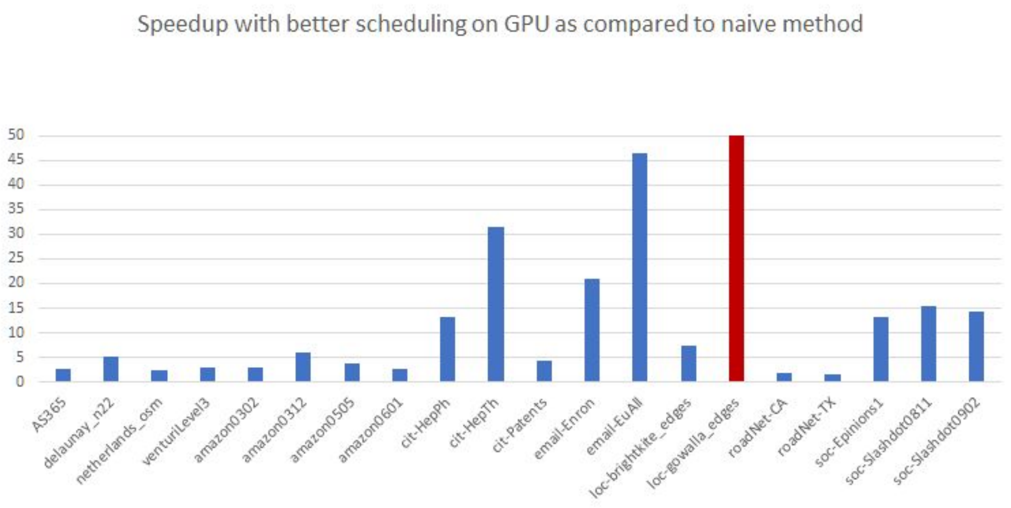
We believe that workload estimation technique similar to the one employed in GPU might achieve better schedulability and extra benefit for the multicore speedups. One possible approach would be to first sort the vertices on basis of decreasing workload estimate and experiment with OpenMP static scheduling to get more speedup. We have left this approach as future work to our implementation.

CUDA Implementation
Combining the workload estimation technique described earlier with the child kernel launch helped us gaining more improvement over naive GPU implementation. Workload estimation helps us in reducing the divergence caused by few highly divergent vertices and chunking them into relatively smaller parts to achieve better utilization of GPU resources.

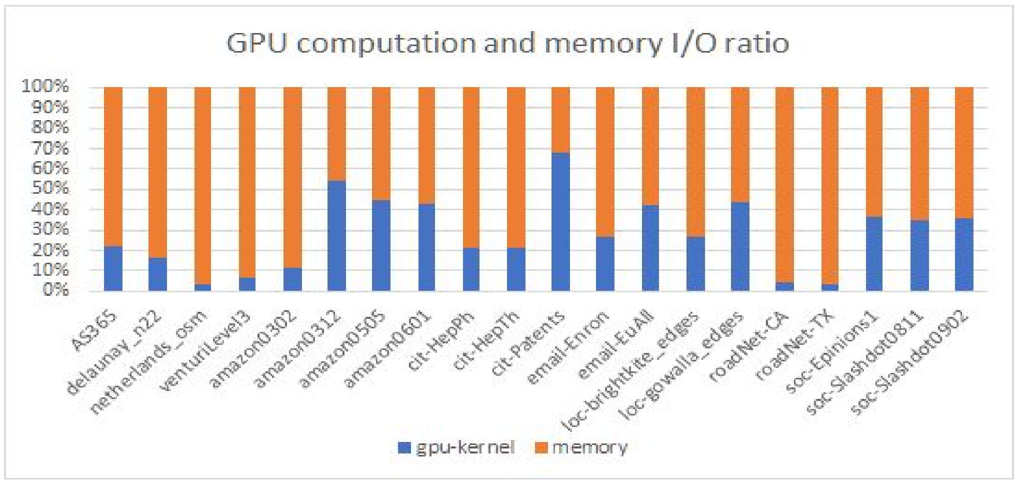
For an example, before any optimization, *loc-gowalla_edges* benchmark had the computation time for 8.28s, whereas sequential CPU implementation for the same took 187ms. The better scheduling approach helped us achieve 44ms of runtime on GPU.

Graph 2 below shows the comparison of naive GPU implementation over the final approach with better scheduling we employed. The workload balance tends to improve with the new approach. We see most of the graphs achieve better runtime results with comparison to naive approach. The red line for *loc-gowalla_edges* shows the very high improvement (~180x) with respect to naive method.

In Graph 3, we compare the time spent in actual compute kernel compared to time spent to transfer the data from host to device and vice versa. We see that most of the workloads have a significant amount of overhead in transferring data from/to device. Approximately 71% of total end to end time for CUDA implementation is spent in transfer of data with some of the workloads as high as 91%. We think that it can be reduced by employing some form of compression technique such as Base Delta Immediate compression[8], which can reduce the memory storage requirement for the edge list of a graph.



Graph 2. Speedup with new scheduling approach over the naive GPU implementation



Graph 2. Blue part shows the kernel computation time and Orange part is the time to transfer data from host to device and vice versa

Is GPU a better choice over CPU?

We are definitely getting good scalability in terms of speedup with increasing number of threads on CPU, but we believe that due to the highly divergent nature of the algorithm and the work we were able to get done in the interest of time, CPU is more suitable platform for this than GPU. But there is still a wide scope of improvement on GPU side (discussed in Future Work), and a chance of getting really good speedup on GPU.

Conclusion & Future Work

Triangle enumeration is a hard problem due to sparse nature of the large real world graphs. It can be seen as mainly a memory bound problem, but computation part on parallel machine is also hindered because of workload imbalance as apparent from CUDA results. With OpenMP, we are able to get good speedup because the scheduling approach is able to divide work amongst 16 threads equally over the long run. But GPU counterpart needs more work on load balancing, which can be the future work for this project.

Advanced blocking techniques can be employed on GPU as mentioned in Static Graph Challenge paper [5] to dynamically balance out the workload. Also, to reduce the memory bandwidth going from CPU to GPU and back, we can compress the edge list of graph being sent to GPU with some pivot values, and trade some computational overhead with the data transfer overhead. One more thing on the memory front, instead of all threads bringing in the vertices, we can deploy shared memory technique inside a block to read the vertices from global buffer once at the beginning, and later use them for triangle computation collectively by all threads within a block, but this might require us to change the algorithm significantly.

One idea that came to us at the very end of the project was to split up the nested loop so that the divergence cause by nested loop can be avoided. This would require to compute the intermediate result of $a^T_{12}A_{20}$ and then compute the inner product with a_{01}. This would help in coalescing the memory requests to bring data from device (GPU) memory for one vertex at a time to multiple vertex at once. We think that this can help gaining performance and getting energy improvements as well (not that we are measuring it, but it is correlated).

Division of Work

Equal work was performed by both project members.

References

[1] T. M. Low, V. N. Rao, M. Lee, D. Popovici, F. Franchetti, and S. McMillan, “First look: Linear algebra-based triangle counting without matrix multiplication,” in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017.

[2] “Data Sets | GraphChallenge.” [Online]. Available: https://graphchallenge.mit.edu/data-sets.

[3] M. Franceschet, “Local Clustering.” [Online]. Available: https://www.sci.unich.it/~francesc/teaching/network/clustering.html.

[4] T. Roughgarden, “CS167: Reading in Algorithms Counting Triangles.” [Online]. Available: https://www-cs.stanford.edu/~rishig/courses/ref/l1.pdf.

[5] M. Bisson and M. Fatica, “Static graph challenge on GPU,” in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017.

[6] A. Azad, A. Buluc, and J. Gilbert, “Parallel Triangle Counting and Enumeration Using Matrix Algebra,” in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015.

[7] M. Rahman and M. Al Hasan, “Approximate triangle counting algorithms on multi-cores,” in *2013 IEEE International Conference on Big Data*, 2013.

[8] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, “Base-delta-immediate compression,” in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques - PACT ’12*, 2012.