Towards Triangle Counting on GPU using Stable Radix binning

Nishith Tirpankar
School of Computing
University of Utah
Salt Lake City, USA
tirpankar.n@utah.edu

Hari Sundar
School of Computing
University of Utah
Salt Lake City, USA
hari@cs.utah.edu

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

A. Graph algorithms overview

A number of data and network analytics questions on relational data can be posed as graph problems. For example, the transitivity or clustering coefficient tells us how clustered the nodes in a graph are. Clustered or small world networks having large value of clustering co-efficient have enhanced signal-propagation speed, synchronizability and computational power [1]. Nodes in sub-graphs with this property can be targetted for quick or low energy information disbursement. Another example involves finding the count and presence of certain structures in a graph. Identifying clusters of these patterns [2] or sub-graphs can indicate classes of predators in a food-web or interactions between sensors and effectors in a neural network [3]. A commonly occuring use case is of recommendations to connect with friends of friends in large social networks. It can be computed using the length of the path between two users [4].

B. Triangle counting as an application

A k-truss is a maximal subgraph of a given graph such that each edge in it is contained in at least k-2 triangles. The k-truss in a graph is a sub-graph all the applications mentioned here can use. Triangles are the simplest subgraph in a graph. Counting the number of triangles in a graph is building block that can be used in finding the k-truss [5]. This makes triangle counting such a lucritive problem for the sub-graph isomorphism challenge [6].

C. Different approaches - set intersection/Linear algebra/map reduce/approximation methods for triangle counting

Among the prominent methods for computing the count of triangles are ones using set intersection, linear algebra, map reduce and approximation methods. Set intersection algorithms [7] involve computing a set of all the possible edges that could generate triangles and counting the number of intersections with the original adjacency list. Innovations in the ordering of the members of the sets as well as ease of distribution of the work among multiple processes makes this class of algorithms highly performant [8] and ideal for implementation on shared memory systems. This class of algorithms is what inspired our work. Linear algebra approaches involve variantions of $\sum_{i,j} A^2 \circ A$ where A is the adjacency matrix [6]. One variantion involves splitting up the adjacency matrix into lower and upper triangular matrices not including the diagonal $A = L + U$. The product $B = L * U$ counts the number of paths of length 2 in the graph. Finding if the wedges close by performing a Hadamard product $C = A \circ B$ gives us the triangle count $\sum_{ij}(C)/2$ [9], [10]. Map reduce approaches use frameworks such as Hadoop and distribute the adjacency lists among nodes arbitrarily. For a more detailed overview refer to [11], [12]. An interesting class of algorithms rely on wedge sampling to get an approximate count of the number of triangles. The work done in [13] shows an excellent use case that uses the birthday paradox to sample a set of wedges from the set of vertices and finding the approximate number of triangles by finding the number of closed wedges in this set.

D. Argument for improving performance by parallelizing graph algorithms for shared memory use with GPU and OpenMP

???NOT SURE what to write here???

E. Discrete algorithm RADIX BUCKETING for set intersection

The MSD radix bucketing algorithm is used in the fastest sorting algorithms [14]. This is due to the ease of splitting up the work among multiple workers with a low cost of communication between recursions. The stable MSD radix bucketing involves 3 main steps which are performed at each level of the recursion.

The first step - the count involves distributing the input list equally among the workers. Each worker finds the number of elements in the input data that belong to the power of two bucket for that level of recursion. The input list of size n will be split up among p workers such that each worker gets a chunk of size n/p to process. This operation has a constant

work complexity $O(n)$ since it involves masking the d bits where $d = log(R)$ and R is the power of two number of buckets for each recursion. Thus, at depth dep of the recursion, the bucket to which an item q in the input list will belong is given by:

$$(q \gg (2^{dep_{max}-dep} - (d-1))) \wedge (R-1) \tag{1}$$

The equation 1 will result in counts for the R buckets at each worker. The step has a time complexity of $O(n/p)$ assuming p workers can perform the count simultaneously. The pR counts need to be aggregated in order to facilitate the last step where p is the number of workers which can be threads or processes. This is done in the second step of radix bucketing - prefix sum scan. The count is a $p \times R$ matrix where each row is the local count at each worker. The value in collumn j corresponds to the count for the j^{th} bucket. If the $p \times R$ matrix is converted into a single vector of size $1 \times pR$ by concatenating all the rows and performing a prefix sum scan on this vector we get a vector of monotonically increasing index locations to the input array. Reverting the operation by converting the $1 \times pR$ vector to a $p \times R$ matrix will give the resultant index locations each worker can write to. Let us call this matrix $M_{p \times R}$ This operation has a work complexity $O(pRlog(pR))$ with a time complexity defined by the Kogge-Stone or Hillis-Steele scan algorithms [14]. This enables us to perform a EREW(exclusive read, exclusive write) operations in the final step which is ideal for a shared memory GPU model.

In the third step the individual workers will move the data from the input buffer to the output buffer. Each of the workers will access its chunk of n/p elements in the input buffer. The workers will move each input element to the output buffer address pointer to by the $M[i, j]$ where i is the worker number and j is the bucket the element maps to as given by 1. After moving the element, the pointer value $M[i, j]$ has to be decremented. After the first worker has processed its n/p moves, the first collumn of $M_{p \times R}$ will contain the pointers to the output buffer that contains R bucketed lists. In order to bucket these lists the bucketing steps has to be performed with $dep = dep-1$ on each of the R lists recursively. The recursion stops at $dep = dep_{max}$. Also, at each level the input and output buffers can be swapped to restrict the space requirement of the buffers to $2n$.

This algorithm can be easily extended to perform a set intersection on two lists by recursing through them simultaneously. Early termination of the recursion will occur when any of the lists has 0 elements. A hybrid approach which uses an algorithm like insertion sort when the size of the input list n_{dep} is small i.e. $n_{dep} < 100$ for the $n = 10000$ has been shown to perform better [16]. Although we have not implemented this, an OrderedMerge as demonstrated in [17], [18] yields substantial improvements at the tail end of the recursion.

II. Background and Related work

A. Parallel Graph algorithms

B. GPU

C. Linear algebra

D. Graph

E. Triangle counting

III. Methods

A. Simple Linear algebra approach as a motivation

The problem of finding the count of triangles can be reduced to counting all paths(walks) of length 2 between two vertices if there is a closure i.e. an edge between the two vertices. An element $A(i, j)$ in the square adjacency matrix defines the number of paths or the weight of an edge beween nodes i and j. For an undirected unweighted graph the adjacency matrix elements are 0 or 1. We know that the n^{th} power of an adjacency matrix A^n gives us the number of walks $A^n(i, j)$ of length n that exist between nodes i and j. For $n = 2$ the walks are paths if the diagonal of the adjacency matrix is 0. Thus, each element of A^2 defines the number of paths of length 2. Let C denote the Hadamard product or elementwise product of this with the adjacency matrix $C = A^2 \circ A$. An element of matrix $C(i, j)$ gives us he number of triangles edge (i, j) is a part of. Hence, the sum of all the elements of this matrix gives us the total number of triangles in the graph. Each triangle is counted three times in the matrix C - once for each edge. Also, in an undirected graph each edge is counted twice as $A(i, j)$ and $A(j, i)$ both indicating the same edge. This implies that $\sum_{ij}(C)$ counts each triangle 6 times. The total number of triangles is given by equation 2:

$$n_T = \sum_{ij}(C)/6 \tag{2}$$

Triangle counting using this method has a high work complexity and is viable only for graphs with dense adjacency matrices. Most graphs tend to have sparser adjacency matrices. We can reduce the work complexity by techniques such as using masked multiplication after LU decomposition [10]. The decomposition as well as the masked multiplication is computationally expensive since it uses a significant amount of communication between workers. We can reduce the amount of communication by using a set intersection approach.

B. Short summary of standard algorithms

Our algorithm consists of two main steps. The first step involves finding out all the possible combinations of wedges that exist in our graph. This is computed using the adjacency list representation of the graph. The result of this step is a list of edges closing the wedges hence forming a triangle. The second step is the radix bucket based set intersection. In this step we find out if the list of candidate edges computed before is actually present in the graph. The following sections describe these steps in detail.

C. Finding candidate closure edges E'

While generating the set of candidate edges we have focussed on maintaining exclusive access to the input and output buffers as well as coalescing our reads and writes. This ensures that the memory accessess which are sequential will benifit from hierarchical memory caches in GPU architectures. Ensuring exclusive read and exclusive writes for large E and E' requires us two know two things. The size of the E' output buffer to allocate in the global memory. The read boundaries of E in which each thread will operate to generate the wedges along with the write boundaries of E' where it will write the candidate edges closing these wedges. This is performed by the algorithm 1.

Algorithm 1 Compute candidate edges for closure test.

```
1:  function COMPUTE_E'(E)
2:      E ← RADIX_SORT(E)                          ▷ E(u,v) by u first then v
3:      cnt[p] ← PARALLEL_COUNT(E)                 ▷ use transitions
4:      E_len ← PARALLEL_REDUCE(cnt[p])
5:      ALLOCATE(E_index, E_degree, E'_size)       ▷ size E_len
6:      E_index ← PARALLEL_OFFSETS(E, p)
7:      E_degree ← PARALLEL_DEGREE_CALC(E_index, p)
8:      E'_size ← PARALLEL_SIZE_CALC(E_degree, p)
9:      E'_size_scan ← INCLUSIVE_SUM_SCAN(E'_size)
10:     E' ← PARALLEL_GEN(E, E_index, E'_size_scan)
11:     return E'
```

The input to algorithm 1 is the edge list E. Each element in the list E is an ordered tuple (u, v) representing a pair of vertices. If the number of vertices in the graph fit within the bounds of an integer container, a single element of E can be interpreted as a long integer. Accounting for endianness might require reordering (u, v) to (v, u). The radix sort in line 2 is performed on the edge list E using the fast double buffered tunable radix sort from [19]. The resultant list is the adjacency list since it is sorted by u first and v next. We refer to the adjacency list as E from here onwards. Since radix sort has a work complexity of $O(bn)$ where b is the number of bits in the container which is fixed, the asymptotic work complexity of radix sort is $O(n)$. The parallel time complexity of this shared memory implementation is $O(n/p)$ where p is the number of threads executing simultaneously. In steps 3 and 4, the total number of vertices which is the length of the first dimension of the adjacency list E_{len} is found. The next 4 steps are used to calculate the indexes in the input buffer E_{index} and output buffer E'_{size_scan}.

Each step from 6 through 8 is performed in the same kernel although they have been explicitly separated. In line 6 the p processors compute the index locations of the beginning of the adjacency list of each vertex and write the result to E_{index}. The difference between consecutive index values in E_{index} is used in line 7 to compute the degree of each vertex. The number of wedges centered on a vertex in the adjacency list is dependent on the degree of the vertex. For each vertex in the adjacency list, a wedge can be formed by selecting one other edge in the adjacency list. We can remove duplicate wedges by only considering the $\binom{deg}{2} = \frac{deg(deg-1)}{2}$ combinations instead of the deg^2 permutations. The number of candidate edges is the number of closing edges which is 1 for each wedge. This value is computed in 8 and stored in E'_{size}.

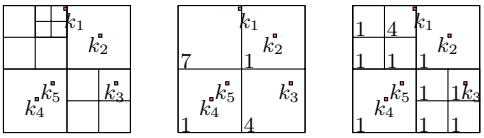

Fig. 1. For a given ordered octree τ and a set of keys.

The inclusive scan sum of E'_{size} gives us the index locations in E' where the combinations corresponding to each vertex in the adjacency list will be written. The inclusive sum is computed using [19]. In the final step in line 10, each thread takes ownership of one or more vertices in the adjacency list E. For each vertex u having degree deg in the adjacency list, the $\frac{deg(deg-1)}{2}$ combinations of edges corresponding to wedges will be written out to the index locations in E' pointed out by E'_{size_scan}.

D. Set intersection using radix bucketing

Counting of the triangles is done by finding out if each edge in E' is actually present in E. Performing a lookup for each member from E' in E can be a $O(|E'|log(|E|))$ operation if E is sorted using a naive binary tree. But E itself can be very large and may not fit in memory even if we batch the lookups of members of E'. In this case we need an algorithm which can scale well and work on shared as well as distributed memory architectures.

Algorithm 2 Count triangles by counting $|E \cap E'|$.

```
1:  t_cnt = 0
2:  b = 2^d                                  ▷ d = radix bits, b = radix buckets
3:  function INTERSECT_COUNT(E, E', t_cnt, depth)
4:      if |E|0 ∨ |E'|0 then
5:          return
6:      if depth ≥ depth_max then
7:          t_cnt = t_cnt + |E'|
8:          return
9:      if |E| ≤ thr ∨ |E'| ≤ thr then
10:         t_cnt = t_cnt+SEQ_INTERSECT_CNT(E, E')
11:         return
12:     E_cnt[b] ← PARALLEL_COUNT(E, depth)
13:     E'_cnt[b] ← PARALLEL_COUNT(E', depth)
14:     E_cnt_scan ← vec^-1(INC_SUM_SCAN(vec(E_cnt^T)))
15:     E'_cnt_scan ← vec^-1(INC_SUM_SCAN(vec(E'_cnt^T)))
16:     for all b do
17:         E[b] ← MOVE(E, E_cnt_scan[b])
18:     for all b do
19:         E'[b] ← MOVE(E', E'_cnt_scan[b])
20:     for all b do
21:         INTERSECT_COUNT(E[b], E'[b], t_cnt, depth + 1)
```

REFERENCES

[1] D. Watts and S. Strogatz, Collective dynamics of small-world networks, Nature, no. 393, 1998.

[2] Adam Polak, Counting Triangles in Large Graphs on GPU.

[3] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, Uri Alon, Network motifs: Simple building blocks of complex networks, Science 298 (2002) 824827

[4]] Thomas Schank, Dorothea Wagner, Finding, counting and listing all triangles in large graphs, Technical report, Universitt Karlsruhe, Fakultt fr Informatik, 2005.

[5] M. Bisson and M. Fatica, "Static graph challenge on GPU," 2017 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, 2017, pp. 1-8.

[6] Static Graph Challenge: Subgraph Isomorphism, S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, J. Kepner, IEEE High Performance Extreme Computing Conference (HPEC), 2017

[7] O. Green, P. Yalamanchili, and L.-M. Mungua, Fast triangle counting on the GPU, in Proc. 4th Workshop Irregular Appl.: Archit. Algorithms, 2014, pp. 18. [Online]. Available: http://dx.doi.org/ 10.1109/IA3.2014.7

[8] Static Graph Challenge on GPU, Mauro Bisson, Massimiliano Fatica

[9] L. Wang, Y. Wang, C. Yang, and J. D. Owens, A comparative study on exact triangle counting algorithms on the GPU, in Proc. 1st High Performance Graph Process. Workshop, May 2016, pp. 18.

[10] Azad, A. Buluc, and J. Gilbert, Parallel triangle counting and enumeration using matrix algebra, in Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshop, 2015, pp. 804811. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2015.75

[11] T. G. Kolda, A. Pinar, T. Plantenga, C. Seshadhri, and C. Task, Counting triangles in massive graphs with mapReduce, CoRR, 2013. [Online]. Available: http://arxiv.org/abs/1301.5887

[12] C. Seshadhri, A. Pinar, and T. G. Kolda, Wedge sampling for computing clustering coefficients and triangle counts on large graphs, Statistical Anal. Data Mining, vol. 7, no. 4, pp. 294307, 2014. [Online]. Available: http://dx.doi.org/10.1002/sam.11224

[13] M. Jha, C. Seshadhri, and A. Pinar, A space-efficient streaming algorithm for estimating transitivity and triangle counts using the birthday paradox, ACM Trans. Knowl. Discov. Data, vol. 9, no. 3, pp. 15:115:21, Feb. 2015. [Online]. Available: http://doi.acm. org/10.1145/2700395

[14] V. J. Duvanenko, "Parallel In-Place Radix Sort Simplified", Dr. Dobb's Journal, January 2011

[15] Single-pass Parallel Prefix Scan with Decoupled Look-back, Duane Merrill, Michael Garland

[16] V. J. Duvanenko, "Stable Hybrid N-bit-Radix Sort", Dr. Dobb's Journal, January 2010

[17] J. Shun and K. Tangwongsan, Multicore triangle computations without tuning, in Data Engineering (ICDE), 2015 IEEE 31st International Conference on. IEEE, 2015, pp. 149160.

[18] A. S. Tom et al., "Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms," 2017 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, 2017, pp. 1-7.

[19] https://nvlabs.github.io/cub/structcub_1_1_device_radix_sort.html

[20] H.D. Macedo, J.N. Oliveira, Typing linear algebra: A biproduct-oriented approach, Science of Computer Programming, Volume 78, Issue 11, 2013, Pages 2160-2191.