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*—The pattern of computations of graph algorithms makes them difficult to parallelize. They suffer from erratic data access patterns. We propose a set of algorithmic patterns that enable users to take advantage of fine grained parallelism provided by modern CPU and GPU architectures. This allows accesses to be regularized which improves hierarchical cache access. We also propose a parallel stable binning algorithm that can be used for computing set intersection. This is illustrated through its application to triangle counting in large graphs.**

***Index Terms*—graph algorithms, CUDA dynamic parallelism**

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. Challenges of parallelizing graph algorithms on modern architectures

Graph algorithms are hard to parallelize and even harder to get good performance on modern architectures, both on many-core CPUs as well as on GPUs. The key challenge is that the nature of computations depends on the structure of the graph, i.e., the sparsity and specifically the sparsity pattern of the adjacency matrix of the graph. This means that the performance of the algorithm is directly dependent on the input graph's structure. This makes it very challenging to implement graph algorithms on parallel architectures where exclusive write access would be needed to obtain good parallel performance. At a more fundamental level, we also have to deal with having indirect memory access, due to the nature of adjacency lists, the most popular and efficient way to store graphs. Indirect memory access is bad for performance

as it is difficult to utilize the cache effectively, leading to even higher costs for data access. One of the reasons for the popularity of linear algebra formulations for graph algorithms is the maturity of sparse linear algebra codes, especially on multicore CPU and GPUs. While these do indeed improve the performance to a certain extent, they are still sub-optimal. For one, unlike numerical algorithms, most graph algorithms, including triangle counting, are discrete in nature. Secondly, linear algebra formulations, especially those involving matrix-matrix products can require extensive data-movement and potentially increase both storage and compute requirements to $\mathcal{O}(n^2)$. For this reason, we wish to develop efficient graph algorithms that expose fine-grained parallelism and avoid indirect memory access. Such a formulation will enable efficient implementations on manycore processors and on GPUs. We now illustrate this idea using an example of set intersection, as that is one of the most expensive parts of triangle counting.

E. Data-access concerns for set intersection

Set intersection among two lists is an expensive operation if the lists are large. Consider the problem of finding the number of edges in a list E', of length m, that are also present in list E of length n. A standard approach for solving this problem could be to sort E with cost $\mathcal{O}(n \log n)$ and then perform m binary searches for a total cost of $\mathcal{O}((m+n)\log n)$. In many cases this is reasonable. However, from a performance perspective, if n is large, although the complexity of $\log n$ appears reasonable, it is very expensive on modern architectures, as this is random access into a very large array, each access potentially being retrieved from main-memory. An alternate approach is to sort both E and E', and performing a linear scan to determine matches. The complexity of this approach is similar at $\mathcal{O}(n \log n + m \log m)$, but this can be much faster, especially if using efficient sorting algorithms. This is in principle similar to using efficient linear algebra routines to obtain efficient implementations. However, it is possible to do even better if we design from first principles. We describe one such approach now.

If we use a most significant bit radix sort to sort the lists E and E', we can improve the performance significantly, especially if we sort the lists simultaneously and use early termination. Note that a radix sort is efficient becuase data is accessed directly and in a streaming fashion, making it extremely amenable to deep cache hierarchies. Each element of the list is an ordered integer tuple (u, v). For a value (u, v) in E to be equal to a value (u', v') in E', each bit in the binary representation of u must match u' and each bit in v must match v'. We use this idea to bucket the entire list one bit at a time. Figure 1 gives an example of how a recursive bucket traversal can be used to perform intersection.

To perform bucketing, we first take the integer u and mask the highest bit. The resultant value is concatenated with the masked highest bit of v. The resultant value is a number between 0 and 3. This lets us put each value in E and E' into the buckets r_0 through r_3 as shown in the last figure in figure 1. From this point we will recursively compute intersections

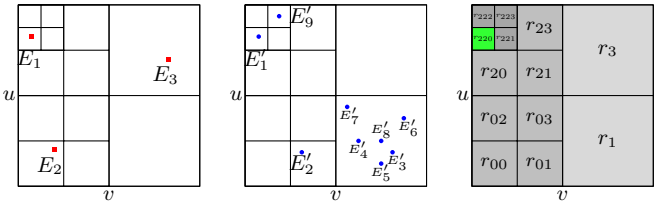

Fig. 1. From left to right: Elements of edge list E with the bucket traversal overlay; elements of edge list E' with the bucket traversal overlay; bucket traversal with region labels.

only if both E and E' have edges in the bucket. As shown in the first and second of figure 1 we can see that bucket r_1 in E does not have any edges while bucket $r3$ in E' does not have any edges. Hence, we do not need to recurse in these buckets at all. Now r_1 has edges E'_3 through E'_8 which do not need to be tested. We have eliminated the need to test 6 out of 9 edges in E'. This is a significant reduction in work. Also, r_3 is eliminated as a potential bucket for recursion in E removing the need to test edge E_3. In the next level of recursion we eliminate buckets r_{00} through r_{03} as well as r_{20}, r_{21}, r_{23}. Recursing on r_{22} further we see that r_{220} is the only bucket which has edges both in lists E and E'. E'_9 in bucket r_{223} does not have corresponding edges in E. Since r_{220} has very few elements in both E and E' we can compare each element in E' to each element in E for r_{220} to find that E'_1 has a matching edge E_1 and is the only edge also present in list E. Although this is a synthetic example, it demonstrates that sparse distributions on E and E' will result in elimination of a large set of candidates for intersection quickly. Each level of recursion is $O(|E| + |E'|)$ i.e. it is $O(n)$ in the size of the lists.

II. Methods

A. Linear algebra approach to triangle counting

The problem of finding the count of triangles can be interpreted as 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 between 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 the number of triangles that the 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 1:

$$n_T = \sum_{ij}(C)/6 \tag{1}$$

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. Two step algorithm

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 focused on maintaining exclusive access to the input and output buffers as well as coalescing our reads and writes. This ensures that the memory accesseses which are sequential will benefit 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 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 lines 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 line 8 and stored in E'_{size}.

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. Although our implementation is on a shared memory SIMD device, we have designed the algorithm to be extended onto distributed memory SPMD architectures. The motivation for our algorithm is the MSD bucketing algorithm which is at the core of the fastest sorting algorithms [14]. It is easy to split up the work among threads with a low communication overhead between recursions and small amount of inter-thread communication.

Note that this algorithm is recursive. It fits very well with the NVIDIA CUDA Dynamic Parallelism extension [21]. The algorithm is shown in 2.

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_buf[b] ← MOVE(E, E_cnt_scan[b])
18:     for all b do
19:         E'_buf[b] ← MOVE(E', E'_cnt_scan[b])
20:     for all b do
21:         INTERSECT_COUNT(E_buf[b], E'_buf[b], t_cnt, depth + 1)
```

The triangle count t_cnt is initialized to 0 and it will be updated atomically by the threads as they find valid candidate edges in E'. We can specify the number of radix bits and hence the number of buckets to use for the algorithm up front. The recursive function takes as parameters the edge list E, the candidate edge list E', the pointer to the triangle count t_cnt and the current depth of the recursion $depth$. Early recursion termination happens on lines 4, 6 and 9. The termination conditions on lines 4, 6 are self-explanatory. A hybrid approach towards the tail end of the recursion is known to perform better [16]. When the input is small i.e. $|E|$ and $|E'|$ are smaller than a threshold thr we perform a simple sequential intersection count and add it to t_cnt. Although we have not implemented it, its worth mentioning that the OrderedMerge demonstrated in [17], [18] are alternatives that can yield substantial improvements at the tail end of the recursion.

The first step of the recursion is the parallel count. The parallel count in lines 12 and 13 distributes the input list among p threads. Each thread will process $\frac{n}{p}$ edges. A simple masking operation for the vertices of each edge as shown in 2 tells us the bucket to which q will belong

$$(q \gg (2^{depth_{max}-depth} - (d-1))) \wedge (b-1) \qquad (2)$$

The b counts E_{cnt}, E'_{cnt} shown in line 12 will be populated with a time complexity $O(\frac{|E|}{p})$ and $O(\frac{|E'|}{p})$ assuming the p threads work simultaneously. This count is computed in global shared memory and is a matrix of size $p \times b$ since the local count computed by each thread is not summed up yet. For the move to be performed in parallel by p threads in lines 17 and 19, we need to know the index location in the output buffers E_{buf} and E'_{buf} where the data will be moved.

The steps in lines 14, 15 perform the task of computing the index locations of the move. The operator vec used in line 14 is the vectorization operator [20] that will convert the $p \times b$ matrix into a column vector of size $bp \times 1$. An inclusive scan of this vector gives us the index location boundaries where each thread will perform a move operation for each bucket. The inclusive sum scan has a work complexity of $O(pb \cdot log(pb))$ and a time complexity defined by the Kogge-Stone or Hillis-Steele scan algorithms [14]. This enables us to perform the move as an EREW(exclusive read, exclusive write) operation in the final step. The vec^{-1} operator simply converts the $bp \times 1$ vector into a $p \times b$ matrix. Note that the vec and vec^{-1} and transpose operations do not have to be performed explicitly if the E_{cnt} is stored in column major order.

The last step before recursion is the data movement. In line 17 the edges in E are moved based on the bucket they belong to the offsets pointed by E_{cnt_scan} in the output buffer E_{buf}. The move is done for E' in line 19. This stable operation requires just two buffers of size $2 \times (|E| + |E'|)$ since the input buffer at $depth$ can be used as the output buffer at recursion $depth + 1$ and vice versa.

Intersect_count will be called recursively for each bucket with the corresponding data as shown in line 21. The pointer to a global memory t_cnt is also passed on. The recursive call is made to all the buckets simultaneously - the loop over all buckets has only been shown for clarity. Similarly, the loops in lines 17, 19 also run simultaneously.

III. Results and conclusion

A. Effect of changing the size of sets $|E|$ and $|E'|$ on the running time

To test scaling we demonstrate the effect of changing two major properties of the graph. The number of edges $|E|$ in the graph and the density of the graph which changes as $|E'|$ changes. In the first set of values shown in table I we can see that the increasing the number of edges as well as the density i.e. correspondingly the size of E' does not have a significant impact on the time taken to COMPUTE_E'. The INTERSECTION_COUNT also takes roughly the same amount of time although it is a bit more unpredictable. The change in the time takes by INTERSECTION_COUNT can be attributed to the maximum recursion depth at which the recursion terminates. In set 2 the outliers from set 1 have been showcased. This rapid increase in the runtime is caused due to incorrect tuning of kernels. We are currently working on tuning the kernels to give a uniform performance irrespective of input size and density. It should be noted that the number of triangles in the actual graph does not really have an effect on the running time. The running time for dataset Theory-25-81-Bk.tsv at 59.5 s is close to the running time of Theory-25-81-B1k.tsv at 66.16 s. The difference in the number of triangles 0 against 2025 does not reduce the running time.

Dataset	$\#edges\|E\|$	size of $\|E'\|$	$\#triangles$	Run time in seconds		
				COMPUTE_E'	INTERSECT_COUNT	Total
Theory-3-4-Bk.tsv	48	96	0	0.6	0.4	0.66
Theory-3-4-5-Bk.tsv	480	3360	0	0.61	0.08	0.71
Theory-16-25-Bk.tsv	1600	87600	0	0.63	0.11	0.76
Theory-3-4-5-9-Bk.tsv	8640	319680	0	0.63	0.36	1.04
Theory-3-4-B1k.tsv	62	225	36	0.6	0.03	0.67
Theory-3-4-5-B1k.tsv	692	10830	287	0.6	0.09	0.71
Theory-16-25-B1k.tsv	1682	105620	400	0.62	0.2	0.84
Theory-3-4-B2k.tsv	62	138	1	0.61	0.4	0.67
Theory-3-4-5-B2k.tsv	692	5339	7	0.61	0.08	0.71
Theory-16-25-B2k.tsv	1682	88943	1	0.62	0.13	0.78
Theory-3-4-5-9-B2k.tsv	13166	522804	35	0.64	0.79	1.48
Theory-3-4-5-9-B1k.tsv	13166	1223427	9107	0.73	11.99	12.76
Theory-25-81-Bk.tsv	8100	2154600	0	0.87	58.59	59.5
Theory-25-81-B1k.tsv	8312	2378865	2025	0.9	65.220001	66.160004
Theory-25-81-B2k.tsv	8312	2165433	1	0.9	58.91	59.860001
p2p-Gnutella04_adj.tsv	79988	518694	934	0.55	2.02	2.75
p2p-Gnutella05_adj.tsv	63678	439066	1112	0.59	1.88	2.62
p2p-Gnutella06_adj.tsv	63050	422567	1142	0.55	1.56	2.26
p2p-Gnutella08_adj.tsv	41554	346033	2383	0.56	1.8	2.47
p2p-Gnutella09_adj.tsv	52026	411347	2354	0.55	1.7	2.38
p2p-Gnutella25_adj.tsv	109410	533121	806	0.55	1.98	2.76
p2p-Gnutella30_adj.tsv	176656	923756	1590	0.54	3.47	4.33
p2p-Gnutella31_adj.tsv	295784	1568174	2024	0.54	5.66	6.95

TABLE I

FOUR SETS OF RESULTS. THE FIRST SET CONSISTS OF GRAPHS WITH ZERO, MANY AND SOME TRIANGLES IN THAT ORDER. THE SECOND CONSISTS OF THE OUTLIERS FROM THE FIRST SET. THE THIRD COLLECTION REPRESENTS GRAPHS WHERE THE NUMBER OF EDGES VARIES AS WELL AS THE SIZE OF THE SET E' VARIES BUT THE RUNNING TIME IS ROUGHLY CONSTANT. THE LAST SET CONTAINS OUTLIERS; THE NUMBER OF EDGES AS WELL AS THE SIZE OF E' GENERATED DOUBLES CAUSING RUN TIME TO DOUBLE AND TRIPLE.

B. Effect of changing the structure of the graph on the running time

The p2p-Gnutella dataset has roughly the same density and connectivity since $|E'|$ is about 450000 while the number of edges range from 40000 to about 100000. This can be seen in the third set of the results. As seen in the total running time the structure and connectivity characteristics do not have a significant effect on the running time which is constant at about 2.5 seconds as seen in table I. The set 4 shows some of the outliers. But we can see that the COMPUTE_E' function still takes the same time. The minor differences in the INTERSECT_COUNT can be attributed to the recursion depth and early termination.

C. Conclusion

The strategy to generate the offsets where we can read from and write processed results can be effectively applied for various parallel tasks. It is scalable as can be seen by the run timings for similar graphs in table I. The recursive radix sort is not performing adequately due to a bug in tuning the kernel. We intend on fixing the issue by batching the stage of generation of $|E'|$ and performing the intersection count. Also, the initial implementation has highlighted several areas of improvement. The CUDA dynamic parallelism extension suffers from the overhead of launching kernels. The number of threads that can be launched is also limited by the compute capability of device. It is ideal for traversing fat and short trees. The isolation of data buffers between different workers implies that we can not only use this technique on a shared memory architecture but on a distributed one as well. Although we have not demonstrated that capability of the technique, we can extend the algorithm to work on a hybrid shared and distributed memory architecture easily. This will enable us to process larger graphs on a cluster of GPU nodes.

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.
[21] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-dynamic-parallelism