Counting Triangles in Large Graphs on GPU

Adam Polak
Department of Theoretical Computer Science
Faculty of Mathematics and Computer Science
Jagiellonian University in Kraków
polak@tcs.uj.edu.pl

***Abstract*—The clustering coefficient and the transitivity ratio are concepts often used in network analysis, which creates a need for fast practical algorithms for counting triangles in large graphs. Previous research in this area focused on sequential algorithms, MapReduce parallelization, and fast approximations.**

In this paper we propose a parallel triangle counting algorithm for CUDA GPU. We describe the implementation details necessary to achieve high performance and present the experimental evaluation of our approach. The algorithm achieves 15 to 35 times speedup over our CPU implementation, and is capable of finding 8.8 billion triangles in a 180 million edges graph in 12 seconds on the Nvidia GeForce GTX 980 GPU.

***Keywords*-GPU; CUDA; parallel graph algorithms; triangles; clustering coefficient**

I. Introduction

The number of triangles, i.e. cycles of length three, in an undirected graph lays the foundation of the clustering coefficient and the transitivity ratio – concepts which are not only of theoretical interest but are often applied to networks analysis [1], [2]. This creates a need for fast practical algorithms capable of counting triangles in large graphs.

Previous research in this area focused mainly on sequential algorithms [3], [4], parallel algorithms for MapReduce model [5], and various approximation approaches [6], [7].

The emergence of frameworks for general-purpose computing on graphics processor units (GPU), such as Nvidia CUDA, started a new branch of research in parallel computing. General-purpose GPU offers the computing power of a small cluster for much lower price, but it comes at a cost of certain limitations imposed by the architecture of graphic cards. Single Instruction Multiple Data model, high latency global memory, and small cache size are obstacles particularly hard to overcome in case of memory-intensive and highly irregular graph computations.

Nevertheless, a number of studies show that there are methods of dealing with these issues and certain graph problems can be solved effectively on GPU – examples include minimum spanning tree [8], connected components [9], breadth first search [10], [11], and strongly connected components [12].

In this paper we propose a parallel triangle counting algorithm together with its CUDA implementation, and discuss its performance. Comparing to a single-threaded CPU solution, we achieve 8 to 16 times speedup running on the Nvidia Tesla C2050 GPU, and 15 to 35 speedup running on the Nvidia GeForce GTX 980 GPU. We also examine a setup with multiple GPUs. We are able to further speed up the computation up to 2.8 times when running on four Tesla C2050 cards instead of one.

Similar studies have been conducted already by Leist et al. [13], and more recently by Chatterjee [14] and Green et al. [15]. As we argue later, our approach, despite being very simple, significantly outperforms the previous ones.

This paper is structured as follows. Section II outlines the algorithm we use. Section III provides the implementation details and discusses the optimizations we employ. Section IV carefully describes the experiments performed to evaluate our implementation, and presents the results of these experiments. Section V compares our algorithm with other approaches to the problem of counting triangles. Section VI contains our conclusions.

II. Algorithm

A. Known Sequential Algorithms

Schank and Wagner [3] present an extensive list of known sequential algorithms for counting and listing triangles. They analyze their theoretical time complexity, and evaluate them experimentally on both synthetic and real world graphs. Two algorithms, the *edge-iterator* and *forward*, appear to be the winners of this comparison. Their running times are $O(m \deg_{\max})$ and $O(m\sqrt{m})$, respectively, where m denotes the number of edges, and $\deg_{\max}$ denotes the maximum degree of a vertex. They perform similarly for graphs with low deviation from the average degree, but the latter is more robust to skewed degree distributions. Latapy [4] simplifies the *forward* algorithm, reducing its memory needs and running time, and makes its analysis more straightforward. This modified version can be seen as a variant of the *edge-iterator* algorithm with an additional preprocessing phase. This allows a tighter bound on the worst-case complexity and greatly reduces the amount of work to be done in the subsequent counting phase, especially in the case of graphs with a high degree deviation.

We choose *forward* as a starting point for our parallel algorithm. Both the preprocessing and counting phases are not only easily parallelizable but they also rarely access the memory in random fashion, which makes the algorithm a good fit for GPU.

B. *Sequential* Forward *Algorithm*

Now we briefly describe the sequential *forward* algorithm. First, the algorithm sets an arbitrary linear order $\prec$ on vertices which is consistent with their degrees, i.e. $\deg(u) < \deg(v)$ implies $u \prec v$. Then, for every vertex u, it filters its adjacency list removing vertices v such that $v \prec u$. Since $\prec$ is antisymmetric, u remains unremoved in the adjacency list of v. This turns each undirected edge into a directed one, by choosing the orientation from the vertex with the lower degree to the vertex with the higher degree. In the final step of the preprocessing, the algorithm sorts adjacency lists according to some arbitrary, previously fixed, linear order, e.g. the natural order on vertices' numbers.

After this preprocessing is done, the algorithm iterates over all edges and, for each edge, calculates intersection of adjacency lists of both its ends. The total size of these intersections is the number of triangles in the graph. Since adjacency lists are sorted, each such intersection can be computed by a two pointers merge algorithm (as in *merge-sort*) in time linear in the sum of lengths of both lists. Note that only edges that go forward according to $\prec$ are left. Thus, every triangle is counted exactly once, and it can be shown [3], [4] that no adjacency list is longer than $\sqrt{m}$, which makes the whole algorithm run in $O(m\sqrt{m})$ time.

C. *Parallel* Forward *Algorithm*

The preprocessing phase is easily parallelized with a few *prefix sum* and *sort* routines. We describe it in detail in the next section.

The counting phase is parallelized by assigning a single thread to each edge. Each thread calculates adjacency lists intersection in sequential manner. This gives us an $O(\sqrt{m})$ time algorithm running on $O(m)$ processors, thus having the optimal work.

From a theoretical point of view, our algorithm does not have a polylogarithmic time complexity, and thus does not put the problem into the class NC. However, in practice, the speedup is always bounded by the number of available processors (or cores in the case of GPU). Our algorithm has the optimal work and $O(m)$ speedup, thus it is optimal for graphs with the number of edges greater than the number of processors the algorithm is executed on. This is generally the case since modern graphic cards usually have only a few hundred up to few thousand cores.

III. Implementation

The implementation described in this section and all the tools used to run the experiments described in the next section are available on GitHub[1].

A. *Input Format*

Before going to implementation details it is important to note what the input format of our algorithm is. It is often tempting to use a data representation that is particularly suitable for a specific algorithm. However, in practice it is rarely the case that the data is available in the selected format, and converting it may take a significant amount of time.

In most works on graph computation on GPU the input format is either an *adjacency list* [10]–[12] or an *edge array* [8], [9]. We decided to use the latter. An *edge array* is an array of structures, each structure composed of two values – identifiers of the two vertices a given edge connects. We assume there are no self-loops nor multiple edges, and each (undirected) edge appears exactly twice, once in each direction. We do not assume any particular order of the edges.

The rationale for of our choice is similar to the one presented in [9]. An *adjacency list* can be converted to an *edge array* with a fast and simple single-pass algorithm. The conversion in the opposite direction requires sorting, which makes it much more computationally expensive. Thus, using the *edge array* representation makes the algorithm more versatile in the sense that it can be used in various contexts without any significant overhead for a format conversion.

The above intuitions can be further supported by the following numbers. On the LiveJournal graph, which we use in the experiments (for more details, see Section IV), our CPU solution optimized for an *adjacency list* input runs about 12 seconds, while the solution optimized for an *edge array* input is only 2 seconds slower. On the other hand, converting this graph from the *edge array* representation to the *adjacency list* representation takes about 7 seconds.

B. *Preprocessing Phase*

The preprocessing phase consists of eight steps. They make a heavy use of the Thrust library [16].

1) Copy the *edge array* to the GPU memory.
2) Calculate number of vertices using *thrust::reduce* routine with *thrust::maximum* operator, which computes the largest vertex identifier across both ends of all edges.
3) Sort edges, according to the first vertex, in case of a tie according to the second vertex, using *thrust::sort* routine, which performs radix sort. This way the *edge array* becomes a concatenated adjacency list of subsequent vertices, each list sorted.
4) Calculate the *node array*: i-th element of this array points to the first edge in the *edge array* whose first vertex is i. This is done by running $m-1$ threads and

[1]`https://github.com/adampolak/triangles`

letting k-th thread examine edges k and $k+1$. If first vertices of these edges differ, the thread writes $k+1$ to an adequate cell of the *node array*. It may happen that the thread stores this value in more than one cell when there is a vertex with an empty adjacency list.

5) Mark edges going "backwards", i.e. from a vertex with the higher degree to a vertex with the lower degree. In case of a tie, compare identifiers of vertices. The degree of a vertex can be calculated quickly by subtracting two subsequent cells of the *node array*.
6) Remove "backward" edges using *thrust::remove_if* routine, which removes marked elements and compacts the *edge array* preserving the order of elements that are not removed.
7) Transform the *edge array* from an array of structures to a structure of arrays. We call this step *unzipping*.
8) Recalculate the *node array*.

C. Counting Phase

Triangles are counted with the following kernel performing the two pointers merge algorithm. The i-th thread deals with edges whose index in the *edge array* modulo the number of threads equals i. For each such edge the thread sequentially computes the size of the intersection of the neighborhoods of the both ends of this edge.

```
__global__ void CountTriangles(
    int m,
    const int* __restrict__ edge,
    const int* __restrict__ node,
    uint64_t* result) {
  int start = blockDim.x * blockIdx.x + threadIdx.x;
  int step = gridDim.x * blockDim.x;
  uint64_t count = 0;

  // for each assigned edge (u, v)
  for (int i = start; i < m; i += step) {
    int u = edge[i], v = edge[m + i];
    int u_it = node[u];
    int u_end = node[u + 1];
    int v_it = node[v];
    int v_end = node[v + 1];

    // run two pointers merge algorithm
    int a = edge[u_it], b = edge[v_it];
    while (u_it < u_end && v_it < v_end) {
      int d = a - b;
      if (d <= 0) a = edge[++u_it];
      if (d >= 0) b = edge[++v_it];
      if (d == 0) ++count;
    }
  }

  result[start] = count;
}
```

After the kernel is done, elements of *result* array are summed, using *thrust::reduce* routine, to obtain the total triangle count, and the algorithm terminates.

As usual, a careful choice of the number of threads and blocks to run is crucial to achieve high performance. We tuned these parameters experimentally, using a grid search approach, with the number of threads per block going through powers of two from 32 to 1024 and the number of blocks per multiprocessor varying from 1 to 16 in steps of 1. We concluded that it is optimal to run 64 threads per block and 8 blocks per each multiprocessor. These numbers are optimal, or nearly optimal, for all the graphs we used in our experiments, as well as for all the three devices we had for our tests, i.e. NVS 5200M, Tesla C2050, and GTX 980. It is worth noting that on GTX 980 a similar performance can be achieved with other combinations giving 512 threads per multiprocessor, e.g. 256 threads per block and 2 blocks per multiprocessor. However, this is not the case for the two older devices.

D. Optimizations

1) Unzipping Edges.: *CountTriangles* kernel runs 13% to 32% faster when the *edge array* is given as a structure of arrays. Conversion from an array of structures to a structure of arrays is very fast, i.e. it takes less than 30 milliseconds for all the graphs we used in our experiments, largest of them having more than 200 million edges.

2) Sorting Edges as 64-bit Integers.: Sorting edges with *thrust::sort* is approximately 5 times faster when the *edge array* is passed to it as an array of 64-bit integers instead of an array of pairs of 32-bit integers. When using this optimization it is important to remember that, because of the endianness, it produces a slightly different ordering – edges are ordered by the second vertex then, in case of a tie, by the first.

3) Avoiding Unnecessary Reads.: Compare our preliminary version of a while loop in *CountTriangles* kernel:

```
while (u_it < u_end && v_it < v_end) {
  int d = edge[u_it] - edge[v_it];
  if (d <= 0) ++u_it;
  if (d >= 0) ++v_it;
  if (d == 0) ++count;
}
```

with our final version:

```
int a = edge[u_it], b = edge[v_it];
while (u_it < u_end && v_it < v_end) {
  int d = a - b;
  if (d <= 0) a = edge[++u_it];
  if (d >= 0) b = edge[++v_it];
  if (d == 0) ++count;
}
```

The preliminary version reads two values form the *edge array* in every loop execution, while the final version reads only one value when no triangle is found. This difference seems unimportant because these unnecessarily read values are often cached, and the preliminary version performs less work in divergent branches. Nevertheless, the final version runs 36% to 48% faster.

4) Read-Only Data Cache.: Starting with the Nvidia Kepler architecture the L1 cache is disabled for global memory. However, read-only data can be cached in the texture cache. Since our algorithm relies heavily on the *edge array* caching, it is crucial that the *edge array* is marked with the `const __restrict__` qualifiers, which allow compiler to use the texture cache. This change results in 17% to 66% faster kernel execution on the Kepler and Maxwell architectures.

5) Reducing Warp Size.: We can simulate reducing the warp size by doubling the number of threads and making half of the threads within a warp idle. Although our final implementation does not benefit from this method, we find it worth noting since it allowed 30% faster kernel execution at earlier stages of the kernel's development. We believe it is due to the fact that, when a read misses the cache and a thread has to wait for the global memory, all other threads in the warp have to wait as well. Reducing the warp size makes less threads affected by a single cache miss. This effect is especially significant for our algorithm because cache misses happen at different moments for different threads.

6) CPU Preprocessing for Very Large Graphs.: Sorting in the step 3 of the preprocessing phase requires the largest amount of the global memory of the GPU. If the input graph is too large to fit into the memory in this step, we use a slightly modified version of the preprocessing. First, we use the CPU to calculate vertex degrees and remove backward edges. It runs slower than on the GPU but halves the input size. Then, we can move to GPU to sort and *unzip* edges and calculate the *node array*. This optimization allows to process graphs twice larger than without it.

7) Unsuccessful Optimization Attempts.: We tried the virtual warp-centric method [10], collaborative reading to shared memory, and assembler level prefetching. None of these optimizations increased the performance of our implementation, probably due to a high overhead compared to possible gains.

E. Multi-GPU Setup

We propose a simple extension of our algorithm to a multi-GPU setup. The preprocessing phase is run just on a single GPU, then the *edge array* and *node array* are copied to the remaining devices, and each device iterates over its allotted subset of edges.

The speedup of this approach is limited by the Amdahl's law. The preprocessing time is roughly proportional to the number of edges, while the counting phase time appears to be proportional to the number of triangles. The fraction of the execution time spent on the preprocessing varies depending on the graph – for graphs that we use in our experiments this fraction ranges from 0.08 to 0.76, which gives the maximum speedup for 4 GPUs between 3.23 and 1.22. This is roughly consistent with our experimental results. The biggest speedups are obtained for Kronecker graphs, which have large triangles to edges ratios.

A less trivial approach to the multi-GPU parallelization would probably require splitting the graph into (not necessarily disjoint) subgraphs, which then can be processed independently [5], [17]. However, it is not clear if the obtained speedup would compensate the overhead caused by the splitting phase. We do not cover this issue in this paper, but we find it a viable direction for future research.

IV. Experiments

To evaluate the performance of our implementation we ran it on a number of graphs and compared its running time with a baseline single-threaded CPU implementation. The baseline implementation is our own implementation of the *forward* algorithm, and it is slightly faster than the one provided in [4].

The graphs we used are: Internet topology graph, LiveJournal online social network, and Orkut online social network from Stanford Large Network Dataset Collection [18]; Citeseer and DBLP co-paper networks and Kronecker R-MAT graphs from 10th DIMACS Implementation Challenge [19]; Barabási-Albert network [20]; Watts-Strogatz network [1]. Table I summarizes basic properties of these graphs.

Experiments were run on the Nvidia Tesla C2050 GPU, Nvidia GeForce GTX 980 GPU, and Intel Xeon X5650 CPU. We compiled the binaries using the NVCC 7.0 and G++ 4.8 compilers with the `-O3` optimization level.

We measured wall clock time. We started each measurement just before the *edge array* is copied from CPU to GPU, and finished it right after the final result was copied from GPU to CPU and the GPU memory was freed. Before the measurement we called *cudaFree(NULL)* to preinitialize CUDA context, because otherwise the first call to *cudaMalloc* takes additional 100 milliseconds.

Each experiment was run five times. In the paper we report the mean values. The standard deviation never exceeded 0.05 of the mean.

The results are presented in Table I. All execution times are given in milliseconds. The first and the third speedup columns show the GPU over CPU speedup, while the second speedup column shows the 4 GPU over 1 GPU speedup. Measurements marked with † denote graphs too large to fit into the GPU memory, which required part of the preprocessing phase to be run on the CPU (as described in Section III-D6). It accounts for lower performance in case of these graphs. The results for Kronecker graphs are also presented visually in Figure 1.

In order to asses the efficiency of our implementation of the *CountTriangles* kernel we used profiler to measure the cache hit rate and memory bandwidth during kernel execution on the GTX 980 card. These measurements are presented in Table II. We consider the cache hit in the 75% to 80% range being a good result. The GTX 980 card offers 224 GB/s of peak memory bandwidth. Our implementation achieves about half of this value, which is also satisfactory. In conclusion, the profiling analysis shows that there is a room for improvement, but it is not large.

Table I
EXPERIMENTAL RESULTS.

Graph	Nodes	Edges	Triangles	CPU	Tesla C2050		4 × Tesla C2050		GTX 980	
				Time [ms]	Time [ms]	Speedup	Time [ms]	Speedup	Time [ms]	Speedup
Real world graphs										
Internet topology	1.7M	22M	29M	3459	277	12.49	306	0.91	186	18.60
LifeJournal	4M	69M	178M	13829	951	14.54	947	1.00	540	25.61
Orkut	3.1M	234M	628M	82558	†9690	†8.52	†7580	†1.28	2815	29.33
Citeseer	0.4M	32M	872M	4990	578	8.63	456	1.27	329	15.17
DBLP	0.5M	30M	442M	4712	446	10.57	410	1.09	239	19.72
Synthetic graphs										
Kronecker 16	2^{16}	5M	119M	2810	179	15.70	97	1.85	82	34.27
Kronecker 17	2^{17}	10M	288M	6957	476	14.62	219	2.17	219	31.77
Kronecker 18	2^{18}	21M	688M	17808	1274	13.98	499	2.55	558	31.91
Kronecker 19	2^{19}	44M	1626M	45947	3434	13.38	1304	2.63	1443	31.84
Kronecker 20	2^{20}	89M	3804M	116811	9308	12.55	3296	2.82	3942	29.63
Kronecker 21	2^{21}	182M	8816M	297426	†33150	†8.97	†13624	†2.43	12009	24.77
Barabási-Albert	0.2M	20M	3M	5508	327	16.84	263	1.24	155	35.54
Watts-Strogatz	1M	50M	219M	9627	589	16.34	576	1.02	324	29.71

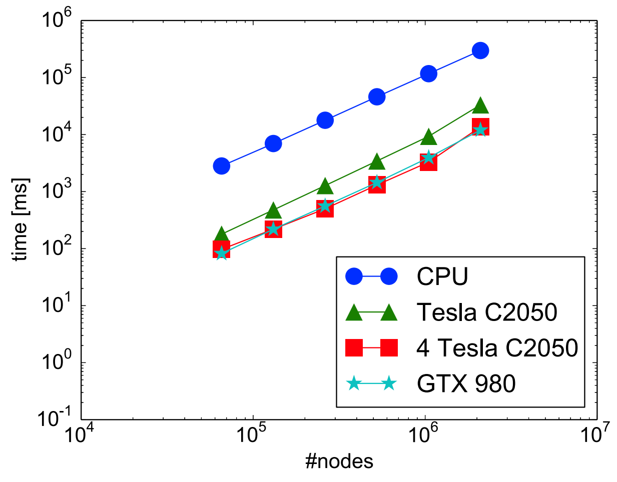

Figure 1. Experimental results for synthetic Kronecker R-MAT graphs.

Table II
PROFILING RESULTS ON GTX 980.

Graph	Cache hit rate	Bandwidth [GB/s]
Real world graphs		
Internet topology	80.78%	95.90
LifeJournal	79.73%	100.28
Orkut	82.71%	98.55
Citeseer	76.68%	117.92
DBLP	78.14%	112.96
Synthetic graphs		
Kronecker 16	80.95%	143.99
Kronecker 17	79.75%	134.33
Kronecker 18	78.35%	128.33
Kronecker 19	77.59%	122.60
Kronecker 20	76.78%	113.37
Kronecker 21	75.81%	93.65
Barabási-Albert	64.45%	137.56
Watts-Strogatz	74.55%	116.82

V. COMPARISON TO RELATED WORK

In the past there were various attempts to count triangles faster than with sequential algorithms. MapReduce approach to the problem [5] has significant overhead, and even for moderately sized graphs the execution time is in the order of minutes. It is beneficial to use it for extremely large graphs, with the number of edges in the order of one billion. Another approach is to use an heuristic approximation algorithm [6], [7]. Such algorithms provide good speedups and usually need little memory, but it comes at the cost of getting only an approximate triangle count, which can differ from the actual count usually by a few percent.

Our CUDA implementation of the parallel triangle counting algorithm, presented in this paper, achieves 8 to 16 times speedup over our optimized single-threaded CPU solution, when run on the Nvidia Tesla C2050 GPU, and 15 to 35 speedup running on the Nvidia GeForce GTX 980 GPU. By using four graphic cards instead of one we can further speedup computation up to 2.8 times, especially when the number of triangles in the input graph is much larger than the number of edges.

Taking into account the speedups achieved by using GPU to solve other graph problems – e.g. 50 times for minimum spanning tree [8], 10 times for connected components [9], 4 to 30 times for breadth first search [11], and 5 to 40 times for strongly connected components [12] – our results seem satisfactory.

The first work we can try to directly compare to is [13]. Unfortunately, such comparison is not easy for a number of reasons. First, the source code for their approach is not available, so we can compare only to the numbers provided in the paper. Second, the paper solves a slightly different problem, which is computing the clustering coefficient. It requires computing the number of triangles but also the number of two-edge paths in the input graph. Fortunately, the latter part is not harder than the former, so we can assume this gives our algorithm at most two times advantage. Third, in the cited paper, experiments were run on the Nvidia GeForce GTX 480 graphic card, which is very similar to the Nvidia Tesla C2050. Nevertheless, they are different devices. Lastly, the execution times measured during these experiments are presented only on plots, so we can obtain only approximate numerical values. Having said that, our algorithm is 45 times faster than the previous one on the Barabási-Albert network, and 7 times faster on the Watts-Strogatz network (the numbers change to 20 and 3, respectively, when comparing runtimes in the four cards setups). We believe this difference is largely due to our choice of the *forward* algorithm.

Another GPU algorithm, proposed in [14], is evaluated on the Nvidia Tesla C1060 which is a slower device than those we used. However, the author reports running times in the order of 20 seconds for graphs with 2000 nodes, which means our approach is orders of magnitude faster.

The most recent work on the topic [15] proposes much more elaborate algorithm, in which also the adjacency list intersection step is parallelized. The algorithm was evaluated on a number of real world graphs, two of which (Citeseer and DBLP) also appear in our experiments. The evaluation was performed on the Nvidia Tesla K40, which is not less powerful than the Nvidia Tesla C2050, which we used. Despite this, our algorithm achieves roughly two times lower execution times for these two graphs.

Yet another approach to count triangles faster is to use a multi-core CPU. According to [13], a parallel counting algorithm running on a 6-core CPU with 12 virtual hyper-threading cores can achieve 7 times speedup over a single-threaded solution. Assuming this result scales well with increasing number of CPUs, it should be possible to achieve performance similar to what we present, on a multiprocessor machine. However, both price and energy consumption of such a setup are likely to be higher than in our case.

VI. Conclusions and Future Work

In this paper we proposed a parallel triangle counting algorithm for CUDA. We proved it can be implemented efficiently, and described details necessary to do so. The algorithm achieves 15 to 35 times speedup over our baseline single-threaded solution, and is capable of finding 8.8 billion triangles in a 180 million edges graph in 12 seconds on the Nvidia GeForce GTX 980 GPU. Despite being very simple, our algorithm significantly outperforms, to our best knowledge, all triangle counting algorithms for GPU up to date.

We plan on extending our research in two directions, which we now briefly discuss.

First, it would be interesting to check if methods from [5], [17] can be applied, without a too big overhead, to split the graph into subgraphs which can be processed independently. This could give a better multi-GPU solution, and what is more important, this would allow to count triangles in graphs which do not fit into the GPU memory – which is one of the biggest limitations of our current algorithm.

Second, it might be beneficial to use a different counting algorithm for a small subset of vertices with largest degrees. A natural candidate for such algorithm is matrix multiplication [21].

Acknowledgment

This research was supported by the Polish Ministry of Science and Higher Education program "Diamentowy Grant".

References

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

[2] S. Eubank, V. S. A. Kumar, M. V. Marathe, A. Srinivasan, and N. Wang, "Structural and algorithmic aspects of massive social networks," in *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '04. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2004.

[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 Efficient Algorithms*, ser. WEA'05. Berlin, Heidelberg: Springer-Verlag, 2005.

[4] M. Latapy, "Main-memory triangle computations for very large (sparse (power-law)) graphs," *Theor. Comput. Sci.*, vol. 407, no. 1-3, Nov. 2008.

[5] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW '11. New York, NY, USA: ACM, 2011.

[6] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos, "Doulion: Counting triangles in massive graphs with a coin," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '09. New York, NY, USA: ACM, 2009.

[7] M. Jha, C. Seshadhri, and A. Pinar, "A space efficient streaming algorithm for triangle counting using the birthday paradox," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '13. New York, NY, USA: ACM, 2013.

[8] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan, "Fast minimum spanning tree for large graphs on the gpu," in *Proceedings of the Conference on High Performance Graphics 2009*, ser. HPG '09. New York, NY, USA: ACM, 2009.

[9] J. Soman, K. Kishore, and P. Narayanan, "A fast gpu algorithm for graph connectivity," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, April 2010, pp. 1–8.

[10] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011.

[11] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12, 2012.

[12] J. Barnat, P. Bauch, L. Brim, and M. Ceska, "Computing strongly connected components in parallel on cuda," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011.

[13] A. Leist, K. Hawick, and D. Playne, "Gpgpu and multi-core architectures for computing clustering coefficients of irregular graphs," in *Proceedings of the International Conference on Scientific Computing*, ser. CSC '11, 2011.

[14] A. Chatterjee, "Parallel algorithms for counting problems on graphs using graphics processing units," Ph.D. dissertation, University of Oklahoma, 2014.

[15] O. Green, P. Yalamanchili, and L.-M. Munguía, "Fast triangle counting on the gpu," in *Proceedings of the Fourth Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3 '14, 2014.

[16] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2010, version 1.7.0. [Online]. Available: http://thrust.github.io/

[17] S. Chu and J. Cheng, "Triangle listing in massive networks and its applications," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '11. New York, NY, USA: ACM, 2011, pp. 672–680. [Online]. Available: http://doi.acm.org/10.1145/2020408.2020513

[18] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, 2014.

[19] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, "Graph partitioning and graph clustering. 10th dimacs implementation challenge workshop," 2012, http://www.cc.gatech.edu/dimacs10/.

[20] A.-L. Barabasi and R. Albert, "Emergence of scaling in random networks," *Science*, no. 286, 1999.

[21] N. Alon, R. Yuster, and U. Zwick, "Finding and counting given length cycles," *Algorithmica*, vol. 17, pp. 354–364, 1997.