Collaborative (CPU + GPU) Algorithms for Triangle Counting and Truss Decomposition on the Minsky Architecture

Static Graph Challenge: Subgraph Isomorphism

Ketan Date*, Keven Feng‡, Rakesh Nagi*, Jinjun Xiong†, Nam Sung Kim‡, and Wen-Mei Hwu‡
*Department of Industrial and Enterprise Systems Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801
†Cognitive Computing & University Partnership, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 10598
‡Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801
Emails: {date2, klfeng2, nagi}@illinois.edu, jinjun@us.ibm.com, {nskim, w-hwu}@illinois.edu

***Abstract*—In this paper, we present collaborative CPU + GPU algorithms for triangle counting and truss decomposition, the two fundamental problems in graph analytics. We describe the implementation details and present experimental evaluation on the IBM Minsky platform. The main contribution of this paper is a thorough benchmarking and comparison of the different memory management schemes offered by CUDA 8 and NVLink, which can be harnessed for tackling large problems where the limited GPU memory capacity is the primary bottleneck in traditional computing platform. We find that the collaborative algorithms achieve 28x speedup on average (180x max) for triangle counting, and 165x speedup on average (498x max) for truss decomposition, when compared with the baseline Python implementation provided by the Graph Challenge organizers.**

***Index Terms*—GPU, CUDA, collaborative graph algorithms, triangle counting, truss decomposition**

I. Introduction

Analytics in social network graphs often seeks to identify cohesive subgroups of individuals that might represent a tribe, a company, a gang, and alike. Triangles and trusses represent the most fundamental substructures in a graph or network, that can be used to obtain analytical insights about the systems amenable to graph theoretical models. Triangle counts are required for calculating metrics such as clustering coefficient, transitivity ratio, etc. Trusses can be used to get approximate idea about the community structure in the given network.

As networks grow rapidly with ever increasing rate of information, there is a need to apply parallel computing techniques for big graph analytics. In recent years, there have been significant advancements in Graphics Processing Units (GPUs) which can also provide a cost-effective solution for high performance computing applications. In this paper, we present collaborative CPU + GPU algorithms for triangle counting and truss decomposition, using the IBM Minsky machine, which is an evolutionary high performance computing platform. With its two Power8 Processors and four Tesla P100 GPUs connected with NVLink interconnect, Minsky is extremely amenable for big data analytics.

The rest of the paper is organized as follows. Section II reviews triangle counting and truss decomposition algorithms. Sections III, IV and V describe the sequential algorithms and our parallel algorithms for triangle counting and truss decomposition. Section VI describes the experimental setup and present the numerical comparison of our algorithms against the baseline implementations. Finally, Section VII concludes the paper with a summary and future directions. This paper is prepared as a formal submission for the IEEE/Amazon/DARPA Static Graph Challenge [1].

II. Literature Review

In this section, we review the state-of-the-art literature on triangle counting and truss decomposition, and then we will describe our contributions.

A. Triangle Counting Literature

The sequential algorithms for triangle counting can be classified into two groups: (1) Matrix multiplication based algorithms which operate on the adjacency and incidence matrices of a graph; and (2) Set intersection based methods that iterate over each edge and find common elements from adjacency lists of head and tail nodes. [2] is an experimental study of triangle counting and listing algorithms. They provide a simple enhancement to the well-known *edge-iterator* that makes processing huge networks feasible. [3] focus on space complexity because space is the bottleneck for triangle listing in large graphs. Their algorithms show performance improvement on power law (rather than general sparse) graphs. [2], [3] only deal with the sequential algorithms. To handle very large graphs, [4] propose I/O efficient triangle listing approach when the input graph cannot fit in main memory. It iteratively partitions the graph into subgraphs that fit in memory. There is no focus on parallelization of the triangle listing itself.

In the direction of parallelization, [5] present a MapReduce implementation of the traditional *node-iterator* triangle counting algorithm. They also present a graph partition scheme based on overlapping subsets that provides the correct triangle count and shows significant speedup. [6] present a GPU implementation of triangle counting based on sorted array intersection, which they refer to as *Intersect Path*. Significant speedups are obtained over CPU sequential implementations. [7] focus on a *wedge-sampling* algorithm for estimating the clustering coefficient when exact approaches are prohibitive in time and/or memory. A MapReduce implementation of this approach enables the authors to address graphs with 8.5B edges. [8] propose a parallel sparse matrix multiplication approach to triangle counting and enumeration using a new primitive MASKED SPGEMM. [9] parallelize the sequential *compact-forward* algorithm of [3] for exact and approximate triangle counting using two-way hyper-threading on a multicore machine. [10] present a parallel triangle counting algorithm using CUDA on single and multiple GPUs. Their approach provides 15-35x speedup over a baseline sequential approach. Finally, [11] present a comprehensive comparative study of various exact triangle counting GPU-based algorithms. Of the three approaches, subgraph matching to a triangle pattern, graph set-intersection, and sparse matrix multiplication-based approaches, they conclude that the set-intersection approach provides the best performance. This becomes the motivational direction for our paper.

B. Truss Decomposition Literature

According to [12], "A k-truss is a non-trivial, one-component subgraph such that each edge is reinforced by at least $k-2$ pairs of edges making a triangle with that edge." In [12], an approach to find the maximal k-truss of a graph is also discussed. [13] discusses decomposing and distributing graph algorithms into a series of MapReduce tasks with the motivation of benefiting from cloud computing. Examples from degree counting, triangle enumeration, among others, are considered. Various stages of map and reduce functions for enumerating triangles and finding trusses are outlined.

[14] consider truss decomposition in massive graphs and first provide an improved in-memory algorithm for computing k-trusses (for modest-sized graphs). For larger networks an I/O-efficient algorithm is provided and compared with approaches of [12] and MapReduce of [13]. These are based on bottom-up (from $k = 2$ and going up) and top-down (starting from the largest k) approaches. [15] improves the existing distributed k-truss decomposition MapReduce algorithm in execution time and disk usage. Along with theoretical results an algorithm based on graph-parallel abstractions is designed. [16] recognizes the communication overhead of repeated triangle counting in most iterative k-truss algorithms and constructs a *triangle complete* or TC subgraph. Local trusses are found for each TC subgraph in parallel, the union of which provides the global k-truss solution. The algorithm reduces the number of iterations by 3-20x by avoiding repeated triangle counting and finding local maximal k-trusses. Similar reductions are found on the communication side.

[17] generalizes the iterative *h-index* computation, previously proposed in [18] for node degrees. Local algorithms for exact and approximate nucleus decomposition are presented, which are effectively parallelized using OpenMP. Significant speedup for k-truss and nucleus decomposition are reported.

The lack of GPU-based algorithms in the truss decomposition literature (to the best of our knowledge) becomes another motivational direction of our paper.

C. Our Contribution

The main contribution of this paper is collaborative CPU + GPU algorithms made possible due to the powerful Minsky architecture. In the parallel algorithms that we have implemented, we leverage the "zero-copy" memory and the "unified" memory, to store the adjacency list. This centrally stored adjacency list can be directly accessed by both CPU and GPU threads. This greatly simplifies the memory management and as a result, it increases the programmers' productivity. Another advantage of this approach is that the algorithms can handle extremely large problems (using the total memory pool of CPU + GPU), that may be otherwise bounded by the GPU memory. Although this approach puts noticeable strain on the interconnect bandwidth, and is usually slower than GPU-only approach, we will show that with NVIDIA's NVLink interconnect, the collaborative algorithms can also achieve significant performance gains.

III. TRIANGLE COUNTING

In this section, we describe the state-of-the-art sequential triangle listing algorithms, and then describe our contributions.

A. Sequential Algorithms

The most promising sequential algorithm for triangle listing is the *Algorithm: Forward* of [3], which is a set-intersection based algorithm. This algorithm first orders the edges based on node degrees, i.e., $u \prec v$ if $d(u) < d(v)$. The edges (u, v) which follow this ordering are kept and remaining are deleted. In case of a tie, i.e., if $d(u) = d(v)$, the edge with $u < v$ is kept. This is essential step in *Algorithm: Forward*, which converts the undirected edges into directed edges. Then the algorithm iterates over all the remaining edges (u, v) and, for each edge, calculates intersection of adjacency lists of its end nodes, i.e, $adj(u) \cap adj(v)$. The expression $\sum_{(u,v)} |adj(u) \cap adj(v)|$ gives the total number of triangles in the graph. Since the edges are ordered, each triangle is counted exactly once. The time complexity of this algorithm is $O(m\sqrt{m})$, where m is the number of edges in the graph.

B. Parallel Algorithm

Our parallel algorithm is outlined below, which follows a pattern similar to [10] and [11].

1) We assume that the input graph is a flat file containing lines of integer triplets. Each line represents an edge in which the first two integers represent the tail node and the head node, respectively, and the third integer represents edge weight or label. We assume that there are no self loops, no parallel edges, and each edge appears in forward and reverse directions.
2) Each edge is read from the file and encoded as a 64-bit integer, such that the first 32 bits correspond to the head node and last 32 bits correspond to the tail node. These

64-bit edges are stored into a C++ Standard Template Library (STL) vector. Note that we keep the edges in 64-bit integer format. The edges can be encoded/decoded using simple mask and shift operations. We assume that the edges are ordered by the head node index and then by the tail node index as in the Graph Challenge datasets.

3) The next step is to create the adjacency list, so that the graph can be traversed easily. For this purpose, we use two arrays: edge array $\mathbf{Z}$ and node pointer array P. Array $\mathbf{Z}$ is of size m, equal to the number of edges, and it is used to store their 64-bit identifiers. Array P is of size $n+1$, where n is the number of nodes in the graph. The element $P[i]$ points in the array $\mathbf{Z}$, the first edge connected to node u. The sub-array $\mathbf{Z}[P[u]]$ represents the adjacency list of node u, containing all the relevant edges connected to that node. Its size can be obtained by simply evaluating the expression $P[u+1]-P[u]$, which also represents the degree of node u. The element $P[n]$ indicates the total number of edges. The vector containing the sorted edges can be directly copied into the array $\mathbf{Z}$. The node pointer array P can be created using a simple parallel algorithm, in which each edge is assigned to one thread. The thread i compares the head nodes u_i and u_{i-1} of the edges $\mathbf{Z}[i]$ and $\mathbf{Z}[i-1]$, respectively; and if $u_i \neq u_{i-1}$, it writes i in $P[u_i]$.
4) Next, the preprocessing step is executed in which the edges are pruned based on the degrees of head and tail nodes, which is the crux of *Algorithm: Forward*. The preprocessing step is implemented using *parallel stream compaction*, which includes *parallel predicate construction*, *parallel prefix-sum*, and *scatter* operations. We used *thrust::prefix_sum* function for *stream compaction*.
5) Finally, triangles are counted for each of the remaining edges using the two-pointer intersection algorithm. Each edge is assigned to one thread, which sequentially counts the number of common nodes in the adjacency lists of the head and tail nodes of that edge. The two-pointer intersection pseudocode is depicted in Algorithm 1. The triangle counts of individual edges are summed together to get the total triangle count.

IV. Truss Decomposition

In this section, we describe the sequential truss decomposition algorithms, and our parallel versions.

A. Sequential Algorithm

The sequential truss decomposition algorithm can be described as follows. Initially, the algorithm calculates the number of triangles for each edge. Then for each edge with triangle count less than $k-2$, the algorithm removes the edge, computes the affected edges, and decrements their triangle counts by 1. This process is repeated until none of the edges can be deleted for the current k, which constitute a k-truss. At this point, k is incremented and the steps are repeated until the graph becomes empty. The time complexity of this algorithm is $O(md_{max})$, where d_{max} is the maximum degree of nodes

Algorithm 1 Two-pointer intersection

Input: $\mathbf{Z}, P, e=(u,v) \in \mathbf{Z}$
Output: $\Delta(e)$

```
1:  $\Delta(e) \leftarrow 0$
2:  $u\_ptr \leftarrow P[u]$; $v\_ptr \leftarrow P[v]$
3:  $u\_end \leftarrow P[u+1]$; $v\_end \leftarrow P[v+1]$
4:  while $(u\_ptr < u\_end) \wedge (v\_ptr < v\_end)$ do
5:      $e_1 = (u, w_1) \leftarrow \mathbf{Z}[u\_ptr]$
6:      $e_2 = (u, w_2) \leftarrow \mathbf{Z}[v\_ptr]$
7:      if $w_1 < w_2$ then
8:          $u\_ptr \leftarrow u\_ptr + 1$
9:      else if $w_1 > w_2$ then
10:         $v\_ptr \leftarrow v\_ptr + 1$
11:     else
12:         $u\_ptr \leftarrow u\_ptr + 1$; $v\_ptr \leftarrow v\_ptr + 1$
13:         $\Delta(e) \leftarrow \Delta(e) + 1$
14:     end if
15: end while
```

in G. The space complexity of this algorithm is $O(m+n)$ for storing the graph and triangle counts. [14] showed that by processing the edges based on ascending order of triangle counts and by using a hashtable for storing the edges, the time complexity can be reduced to $O(m\sqrt{m})$, at the expense of additional $O(m)$ space to store the hashmap and $O(m)$ space to store the sorted edges.

B. Parallel Algorithm

The sequential algorithm iterates over each of the edges one at a time. However, each edge can be processed independently and thus if correctly implemented, the algorithm can be efficiently parallelized. The main challenge in a parallel algorithm is that several edges (with less than $k-2$ triangles) may be deleted simultaneously during a particular iteration, which may cause inconsistency in the triangle counts of the surviving edges. To address this issue, the simplest idea is to re-count the triangles each time a subset of edges are deleted. Algorithm 2 depicts this naïve parallel truss decomposition procedure. Time complexity of this algorithm is $O(k_{max}md_{max})$, which is quite large. Space complexity is $O(m+n)$ for the graph and $O(m)$ for *stream compaction*.

Algorithm 3 depicts an efficient version of truss decomposition. The main feature of this algorithm is that, if any of the edges are deleted, it only re-counts the triangles for the edges that are affected due to the deletions. This results in significant savings in the execution time, especially for problems in which maximal trussness is quite large. The time complexity of this algorithm is still $O(k_{max}md_{max})$, but in practice it is much more efficient. The space complexity is $O(m+n)$ for storing the graph plus $O(m)$ space each for storing the triangle counts, "deleted" edge-list, and "working" edge-list at each iteration. It also requires an additional $O(m)$ space for *stream compaction*.

The implementation details for this algorithm are presented below. Most of the steps are similar to the triangle counting

Algorithm 2 Naïve truss decomposition

```
Input: $G = \{V, E\}$
Output: $k$-truss for $3 \le k \le k_{max}$

 1: $k \leftarrow 3$
 2: $flag \leftarrow true$
 3: while $flag = true$ do
 4:     $flag \leftarrow false$
 5:     $E_{\text{work}} \leftarrow StreamCompact(E,$ “degree ordering”$)$
 6:     for each $e = (u, v) \in E_{\text{work}}$ do                ▷ Parallel for
 7:         $\Delta(e) \leftarrow |adj(u) \cap adj(v)|$          ▷ Count triangles
 8:         if $\Delta(e) < k - 2$ then
 9:             Binary search $e' = (v, u) \in E$
10:             Mark $e$ and $e'$ in $E$ as “delete”
11:             $flag \leftarrow true$
12:         end if
13:     end for
14:     $E \leftarrow StreamCompact(E,$ “not delete”$)$
15: end while
16: Output $G$ as $k$-truss
17: if $G$ not empty then
18:     $k \leftarrow k + 1$
19:     goto 2
20: end if
```

Algorithm 3 Efficient truss decomposition

```
Input: $G = \{V, E\}$
Output: $k$-truss for $3 \le k \le k_{max}$

 1: $k \leftarrow 3$
 2: $E_{\text{del}} \leftarrow \emptyset$
 3: $E_{\text{work}} \leftarrow StreamCompact(E,$ “degree”$)$
 4: for each $e = (u, v) \in E_{\text{work}}$ do                    ▷ Parallel for
 5:     $\Delta(e) \leftarrow |adj(u) \cap adj(v)|$              ▷ Count triangles
 6:     Binary search $e' = (v, u) \in E$
 7:     $\Delta(e') \leftarrow \Delta(e)$                ▷ Propagate triangle counts
 8: end for
 9: $flag \leftarrow true$
10: while $flag = true$ do
11:     $flag \leftarrow false$
12:     for each $e = (u, v) \in E_{\text{work}}$ do                ▷ Parallel for
13:         if $\Delta(e) < k - 2$ then
14:             Mark $e$ in $E_{\text{work}}$ as “delete”
15:             Binary search $e' = (v, u) \in E$
16:             Mark $e$ and $e'$ in $E$ as “delete”
17:             $flag \leftarrow true$
18:         end if
19:     end for
20:     $E_{\text{del}} \leftarrow StreamCompact(E_{\text{work}},$ “delete”$)$
21:     for each $e = (u, v) \in E_{del}$ do                   ▷ Parallel for
22:         $W \leftarrow adj(u) \cap adj(v)$           ▷ Enumerate triangles
23:         $e_1 \leftarrow (u, w)$ and $e_2 \leftarrow (v, w)$ s.t. $w \in W$
24:         Binary search $e'_1 = (w, u) \in E$, $e'_2 = (w, v) \in E$
25:         Mark $e_1, e_2, e'_1, e'_2$ in $E$ as “affected”
26:     end for
27:     $E \leftarrow StreamCompact(E,$ “not delete”$)$
28:     $E_{\text{work}} \leftarrow StreamCompact(E,$ “affected and degree”$)$
29:     for each $e = (u, v) \in E_{\text{work}}$ do                ▷ Parallel for
30:         $\Delta(e) \leftarrow |adj(u) \cap adj(v)|$        ▷ Recount triangles
31:         Binary search $e' = (v, u) \in E$
32:         $\Delta(e') \leftarrow \Delta(e)$            ▷ Propagate triangle counts
33:     end for
34: end while
35: Output $G$ as $k$-truss
36: if $G$ not empty then
37:     $k \leftarrow k + 1$
38:     goto 9
39: end if
```

algorithm, since it serves as the main subroutine.

1) We begin by reading the graph, and creating the initial edge array **Z** and the node pointer array P.
2) Next, a “working” edge array is created using *stream compaction*, in which the edges are pruned based on the degrees of head and tail nodes, similar to *Algorithm: Forward*. This ordered edge array also serves as the initial list of “affected” edges.
3) Next, triangles are counted for each edge in the “working” edge array using the two-pointer intersection algorithm. Each edge is assigned to one thread, which sequentially counts the number of common nodes in the adjacency lists of the head and tail nodes of that edge. Triangle counts for the “reverse” edges are propagated using binary search. This is necessary because in the subsequent iterations, either the forward or the reverse edge might survive depending on the node degrees.
4) In the main loop of the algorithm, first the edges with insufficient number of triangles ($< k - 2$) are marked for deletion. Using binary search, we find the indices of the reverse edges and also mark them for deletion.
5) Next, we create an array of “deleted” edges using *stream compaction*, and enumerate the triangles associated with each edge, to obtain a list of newly affected edges. Using binary search, we search for the indices of the corresponding reverse edges and mark both the forward edge and the reverse edge as “affected.”
6) Next, the edge array **Z** and the node pointer array P are updated using *stream compaction*, to get the new adjacency list for the (possible) next iteration.
7) Finally, a new “working” edge array is created using *stream compaction*, which includes the affected edges that satisfy the degree ordering. Triangles are re-counted for these edges and their “reverse” counterparts.
8) Steps 10 to 34 are repeated until none of the edges get marked for deletion. This means that a k-truss has been found. At this stage, k is incremented and the algorithm returns to Step 9. The algorithm terminates when all the edges are deleted for some k.

V. Collaborative Algorithm

In this section, we describe our CPU + GPU collaborative algorithm under the various memory management schemes.

A. Memory Management

To store the CSR arrays, we experimented with two different memory management schemes:

1) In the first scheme, the required memory is allocated by calling `cudaHostAlloc`, and using the flag `cudaHostAllocMapped`. This is also known as the zero-copy memory allocation, which allows the GPU threads to seamlessly access the data.
2) In the second scheme, we allocated the array by calling `cudaMallocManaged`, which makes use of the "unified memory" introduced in CUDA 6. Managed memory is accessible to both the CPU and GPU using a single pointer. Variables in the managed memory can reside in the CPU physical memory, the GPU physical memory, or even both. The CUDA runtime software and hardware implement data migration and coherence support.

B. Collaboration Scheme

In the collaborative scheme, we used both CPU and GPU threads to perform the various steps of the parallel algorithms for triangle counting and truss decomposition. For CPU multi-threading, we used OpenMP `#pragma omp` directives, whereas GPU code is executed in kernels. The data to be processed is divided into five chunks. Four of those chunks are assigned to the four GPUs and the fifth chunk is assigned to the CPU. Since the GPU kernel calls are asynchronous, both CPU and GPUs can simultaneously process the data chunk assigned to them, and the resources are synchronized by calling `cudaDeviceSynchronize`, which serves as the barrier. Owing to the memory management schemes discussed earlier, no explicit data transfer is required between the CPU and the GPU, and the GPU kernels directly access the required data from the RAM, for both read and write operations.

VI. Computational Experiments

We evaluated the performance of our implementations on various graphs and compared the running time with that of the baseline Python implementation provided by Graph Challenge organizers. Experiments were performed on the IBM Minsky machine, which contains two Power8 CPUs, with 80 cores each, 4.02GHz clock speed, and 256GB memory (total 512GB). It also contains four NVIDIA Tesla P100 GPUs (2 per CPU socket), connected by 80GB/s NVLink interconnect.

We used all four GPUs in our experiments and all 160 CPU threads. The workload was split such that 90% of the data was processed by the GPUs and 10% was processed by the CPU(s). We measured the execution time after the edges are read into the main memory, therefore it includes any preprocessing times and not the reading times. The results are presented in Tables I, II, III, and IV. The maximum energy consumption of a single P100 GPU is 300W. We monitored the energy consumption using `nvidia-smi` and found that the energy consumption did not exceed 45-50W.

From these results, the following observations can be made.

1) For triangle counting, the GPU-based algorithms show good speedup on a large subset of graphs (28x average and 180x max with zero-copy, and 6x average and 61x max with unified memory). For a few graphs the speedup is less than 1. This is because preprocessing and kernel invocation overheads are dominant in smaller and less dense graphs. In larger and denser graphs, we found that GPU-based algorithms were substantially faster than the python baseline. We also saw that the python baseline code failed to solve the synthetic Graph500 graphs, as opposed to our implementation.
2) This difference can be better seen in the truss decomposition results, which requires repeated invocation of triangle counting algorithm. We can see that the speedup obtained is 165x average and 498x max for zero-copy memory management scheme; and 43x average and 297x max for unified memory management scheme. This speedup is obtained mostly because we have used an efficient algorithm, as opposed to the naïve algorithm implemented in the python baseline code.
3) In majority of the tests, we saw that the unified memory management scheme is on average 3-4x slower than the zero-copy memory management. This corroborates the fact that unified memory management scheme is built for programming productivity and not for performance.
4) In the zero-copy memory management scheme, most of the performance loss is due to the high latency of zero-copy memory access, which achieves the bandwidth of about 6-10GB/s. If the data were to reside in the native memory of the processing element (RAM for CPU and device memory for GPU), that would have been the most efficient implementation. However, this requires non-trivial partitioning of the graph, which we propose as a future direction of research.
5) We also tested two other variants, to provide alternate baselines. In the first variant, 100% of the work was given to the 160 CPU threads (OMP-160). In the second variant, a single GPU was used, with all the arrays stored in the device memory. Due to data locality, both these variants outperform the zero-copy and unified memory implementations (GPU version being the fastest).

VII. Conclusion

To summarize, we proposed collaborative algorithms for triangle counting and truss decomposition for the IBM Minsky architecture. We provided details of the efficient implementations of those algorithms using two different memory management schemes: zero-copy and unified memory. The collaborative algorithms achieve 28x speedup on average (180x max) for triangle counting, and 165x speedup on average (498x max) for truss decomposition, when compared to the baseline python implementation provided by the Graph Challenge organizers. We also concluded that the zero-copy

TABLE I
TRIANGLE COUNTING BENCHMARKS ON REAL-WORLD GRAPHS

Graph	n	m	TC	Python Baseline	OMP-160		Single GPU		Zero-copy		Unified	
				Time (s)	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time(s)	Speedup
cit-Patents	3,774,768	33,037,894	7,515,023	141.298	0.654	216.01	0.274	515.50	1.519	93.03	763.590	0.19
roadNet-CA	1,965,206	5,533,214	120,676	3.543	0.290	12.21	0.180	19.64	0.574	6.17	0.681	5.20
amazon0601	403,394	4,886,816	3,986,507	23.036	0.264	87.40	0.163	141.52	0.657	35.06	41.528	0.55
amazon0505	410,236	4,878,874	3,951,063	23.203	0.264	87.76	0.162	143.24	0.639	36.34	43.449	0.53
amazon0312	400,727	4,699,738	3,686,467	22.058	0.295	74.78	0.162	135.81	0.590	37.36	39.146	0.56
flickrEdges	105,938	4,633,896	107,987,357	306.638	0.403	760.41	0.188	1633.75	1.705	179.82	874.411	0.35
roadNet-TX	1,379,917	3,843,320	82,869	2.386	0.255	9.37	0.173	13.76	0.574	4.16	0.784	3.05
roadNet-PA	1,088,092	3,083,796	67,150	1.946	0.254	7.64	0.169	11.53	0.463	4.20	0.655	2.97
amazon0302	262,111	1,799,584	717,719	3.181	0.228	13.97	0.157	20.27	0.551	5.78	3.454	0.92
soc-Slashdot0811	77,360	938,360	551,724	24.388	0.228	106.94	0.155	157.06	0.548	44.46	8.943	2.73
cit-HepPh	34,546	841,754	1,276,868	8.578	0.264	32.44	0.154	55.80	0.535	16.03	4.932	1.74
email-EuAll	265,214	728,962	267,313	49.110	0.259	189.28	0.151	324.53	0.518	94.74	0.805	61.01
cit-HepTh	27,770	704,570	1,478,735	12.960	0.244	53.09	0.149	86.83	0.507	25.55	3.260	3.98
loc-brightkite_edges	58,228	428,156	494,728	4.146	0.210	19.70	0.147	28.26	0.455	9.11	0.687	6.04
ca-AstroPh	18,772	396,100	1,351,441	3.897	0.219	17.83	0.148	26.30	0.538	7.24	0.495	7.88
email-Enron	36,692	367,662	727,044	8.153	0.272	29.93	0.146	55.82	0.546	14.93	1.101	7.41
ca-HepPh	12,008	236,978	3,358,499	4.260	0.245	17.35	0.147	28.94	0.495	8.61	0.507	8.40
ca-CondMat	23,133	186,878	173,361	0.570	0.218	2.62	0.154	3.69	0.501	1.14	0.525	1.08
facebook_combined	4,039	176,468	1,612,010	2.629	0.232	11.34	0.150	17.57	0.534	4.93	0.517	5.09
as-caida20071105	26,475	106,762	36,365	4.286	0.264	16.24	0.147	29.08	0.477	8.98	0.497	8.62
p2p-Gnutella04	10,876	79,988	934	0.161	0.234	0.69	0.144	1.12	0.526	0.31	0.473	0.34
oregon1_010331	10,670	44,004	17,144	1.518	0.242	6.27	0.144	10.54	0.464	3.27	0.466	3.26
as20000102	6,474	25,144	6,584	0.538	0.234	2.31	0.144	3.75	0.495	1.09	0.548	0.98

TABLE II
TRUSS DECOMPOSITION BENCHMARKS ON REAL-WORLD GRAPHS

Graph	n	m	k_{max}	Python Baseline	OMP-160		Single GPU		Zero-copy		Unified	
				Time (s)	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
cit-Patents	3,774,768	33,037,894	36	$>$ 4hrs	14.588	-	8.902	-	26.762	-	5,450.760	-
roadNet-CA	1,965,206	5,533,214	4	526.181	0.680	773.33	0.249	2,115.06	1.057	497.72	1.770	297.35
amazon0601	403,394	4,886,816	11	1,443.571	4.756	303.55	2.042	706.85	6.990	206.51	655.323	2.20
amazon0505	410,236	4,878,874	11	2,666.413	5.395	494.25	1.717	1,552.54	8.349	319.36	702.343	3.80
amazon0312	400,727	4,699,738	11	2,213.735	5.557	398.39	1.151	1,922.50	10.351	213.88	662.999	3.34
flickrEdges	105,938	4,633,896	574	$>$ 4hrs	136.301	-	33.349	-	217.902	-	$>$ 4hrs	-
roadNet-TX	1,379,917	3,843,320	4	368.975	0.597	618.08	0.226	1,630.41	0.974	378.93	1.738	212.28
roadNet-PA	1,088,092	3,083,796	4	295.109	0.579	509.82	0.209	1,409.53	0.815	361.93	1.919	153.80
amazon0302	262,111	1,799,584	7	306.633	1.562	196.36	0.366	838.55	1.895	161.78	64.619	4.75
soc-Slashdot0811	77,360	938,360	35	2,863.684	12.392	231.08	1.671	1,713.78	13.957	205.18	200.375	14.29
cit-HepPh	34,546	841,754	25	1,888.288	12.265	153.96	1.785	1,058.04	16.144	116.97	131.961	14.31
email-EuAll	265,214	728,962	20	2,508.102	7.063	355.09	2.688	933.12	7.896	317.63	181.436	13.82
cit-HepTh	27,770	704,570	30	2,387.755	14.477	164.94	3.199	746.50	17.331	137.77	171.494	13.92
loc-brightkite_edges	58,228	428,156	43	1,498.010	11.840	126.52	1.786	838.65	11.863	126.28	44.578	33.60
ca-AstroPh	18,772	396,100	57	854.938	7.072	120.89	2.066	413.71	13.222	64.66	89.842	9.52
email-Enron	36,692	367,662	22	1,053.504	9.681	108.83	2.975	354.10	12.185	86.46	88.356	11.92
ca-HepPh	12,008	236,978	239	1,080.121	8.376	128.95	1.809	597.01	9.057	119.25	50.773	21.27
ca-CondMat	23,133	186,878	26	109.940	2.643	41.59	0.394	279.31	3.132	35.10	7.854	14.00
facebook_combined	4,039	176,468	97	1,235.489	26.478	46.66	6.593	187.39	28.586	43.22	65.646	18.82
as-caida20071105	26,475	106,762	16	143.366	3.405	42.11	0.380	377.27	3.903	36.73	5.938	24.14
p2p-Gnutella04	10,876	79,988	4	3.820	0.380	10.06	0.152	25.21	0.709	5.38	0.661	5.78
oregon1_010331	10,670	44,004	16	39.433	2.426	16.26	0.275	143.42	2.488	15.85	2.851	13.83
as20000102	6,474	25,144	10	12.128	1.472	8.24	0.207	58.53	1.954	6.21	1.711	7.09

TABLE III
ZERO-COPY TRIANGLE COUNTING ON GRAPH500 GRAPHS

Graph	n	m	TC	Time (s)
scale18	174,147	7,600,696	82,287,285	2.741
scale19	335,318	15,459,350	186,288,972	7.767
scale20	645,820	31,361,722	419,349,784	25.030
scale21	1,243,072	63,463,300	935,100,883	76.644
scale22	2,393,285	128,194,008	2,067,392,370	202.187
scale23	4,606,314	258,501,410	4,549,133,002	656.041

TABLE IV
ZERO-COPY TRUSS DECOMPOSITION ON GRAPH500 GRAPHS

Graph	n	m	k_{max}	Time (s)
scale18	174,147	7,600,696	159	428.453
scale19	335,318	15,459,350	213	1,391.646
scale20	645,820	31,361,722	284	4,908.068
scale21	1,243,072	63,463,300	373	18,842.398

memory management is on average 3-4x faster than the unified memory management. Although, our tests are currently limited to small/medium sized problems, with further improvements, the Minsky architecture and our proposed algorithms can be used effectively to analyze large graphs containing hundreds of billions of edges, making Minsky the future of high performance data analytics.

The most promising direction of future research, is to extend the current algorithms to a grid computing environment using hybrid MPI + OpenMP + CUDA architecture. This will allow us to store and solve larger problems more efficiently. However, these "distributed" algorithms would be better served if the input graphs are cleverly partitioned into possibly non-overlapping subgraphs. For this purpose, the partitioning schemes of [4], [5], [11], [16] may be used, which will unleash the true potential of the Minsky machine.

ACKNOWLEDGMENTS

The authors acknowledge Carl Pearson, Juan Gomez-Luna, Omer Anjum, Mohamed Aly, and Karm Nagi for their help.

REFERENCES

[1] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner, "Static graph challenge: Subgraph isomorphism," in *IEEE HPEC*, 2017.

[2] T. Schank and D. Wagner, "Finding, counting and listing all triangles in large graphs, an experimental study," in *International Workshop on Experimental and Efficient Algorithms*. Springer, 2005, pp. 606–609.

[3] M. Latapy, "Main-memory triangle computations for very large (sparse (power-law)) graphs," *Theoretical Computer Science*, vol. 407, no. 1, pp. 458 – 473, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397508005392

[4] 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*. ACM, 2011, pp. 672–680.

[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*. ACM, 2011, pp. 607–614.

[6] O. Green, P. Yalamanchili, and L.-M. Munguía, "Fast triangle counting on the GPU," in *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*. IEEE Press, 2014, pp. 1–8.

[7] T. G. Kolda, A. Pinar, T. Plantenga, C. Seshadhri, and C. Task, "Counting triangles in massive graphs with MapReduce," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. S48–S77, 2014.

[8] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*. IEEE, 2015, pp. 804–811.

[9] J. Shun and K. Tangwongsan, "Multicore triangle computations without tuning," in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 2015, pp. 149–160.

[10] A. Polak, "Counting triangles in large graphs on GPU," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 740–746.

[11] L. Wang, Y. Wang, C. Yang, and J. D. Owens, "A comparative study on exact triangle counting algorithms on the GPU," in *Proceedings of the ACM Workshop on High Performance Graph Processing*. ACM, 2016, pp. 1–8.

[12] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National Security Agency Technical Report*, p. 16, 2008.

[13] ——, "Graph twiddling in a MapReduce world," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29–41, 2009.

[14] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 812–823, 2012.

[15] P.-L. Chen, C.-K. Chou, and M.-S. Chen, "Distributed algorithms for k-truss decomposition," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 471–480.

[16] Y. Shao, L. Chen, and B. Cui, "Efficient cohesive subgraphs detection in parallel," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, 2014, pp. 613–624.

[17] A. E. Sariyüce, C. Seshadhri, and A. Pinar, "Parallel local algorithms for core, truss, and nucleus decompositions," *arXiv preprint arXiv:1704.00386*, 2017.

[18] L. Lu, T. Zhou, Q.-M. Zhang, and H. E. Stanley, "The h-index of a network node and its relation to degree and coreness," *Nature Communications*, vol. Jan 12, p. 7:10168, 2016.