Collaborative (CPU + GPU) Algorithms for Triangle Counting and Truss Decomposition

Update Paper for Static Graph Challenge

Vikram S. Mailthody*, Ketan Date‡, Zaid Qureshi§, Carl Pearson*, Rakesh Nagi‡, Jinjun Xiong†, and Wen-mei Hwu*
*ECE, §CS, ‡ISE, University of Illinois at Urbana-Champaign, Urbana, IL 61801
†Cognitive Computing Systems Research, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 10598
Emails: {vsm2, date2, zaidq2, pearson, nagi}@illinois.edu, jinjun@us.ibm.com, w-hwu@illinois.edu

***Abstract*—In this paper, we present an update to our previous submission from Graph Challenge 2017. This work describes and evaluates new software algorithm optimizations undertaken for our 2018 year submission on Collaborative CPU+GPU Algorithms for Triangle Counting and Truss Decomposition. First, we describe four major optimizations for the triangle counting which improved performance by up to 117x over our prior submission. Additionally, we show that our triangle-counting algorithm is on average 151.7x faster than NVIDIA's NVGraph library (max 476x) for SNAP datasets. Second, we propose a novel parallel k-truss decomposition algorithm that is time-efficient and is up to 13.9x faster than our previous submission. Third, we evaluate the effect of generational hardware improvements between the IBM "Minsky" (POWER8, P100, NVLink 1.0) and "Newell" (POWER9, V100, NVLink 2.0) platforms. Lastly, the software optimizations presented in this work and the hardware improvements in the Newell platform enable analytics and discovery on large graphs with millions of nodes and billions of edges in less than a minute. In sum, the new algorithmic implementations are significantly faster and can handle much larger "big" graphs.**

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

I. Introduction

Triangles and k-trusses are the basic substructures in a graph, which convey information about its cohesiveness. A triangle is defined as a cycle of length three, while a k-truss is defined as a subgraph in which every edge belongs to at least $k-2$ triangles. Triangles and trusses can be enumerated in polynomial time, however sequential algorithms can be unacceptably slow, especially for graphs with billions of nodes and edges. The goal of the Static Graph Challenge [1] is to address this issue by motivating researchers to develop energy- and time-efficient algorithms for the state-of-the-art parallel and high performance systems to handle "big" graphs.

In our prior Graph Challenge submission [2] we presented collaborative CPU+GPU algorithms for triangle counting and truss decomposition using the IBM Minsky architecture. We showcased how a developer can exploit the "zero-copy" and "unified" memory in a simplified memory management scheme, to simultaneously work with both CPU and GPU threads. In the current paper, we build on our previous work and make the following specific contributions:

1) We present optimized versions of triangle counting and k-truss decomposition algorithms, and discuss their performance trade-offs on various memory management schemes. Our optimized versions are upto 117x and 13.9x faster than prior triangle counting and k-truss decomposition algorithms.

2) We show that our optimized single GPU triangle counting algorithm is on average 151.7x (max 476x) faster than Nvidia's NVGraph library [3], for graphs that fit in the GPU memory. Additionally, for large synthetic graphs, our worst case performance matches the NVGraph library.

3) We evaluate the performance of our triangle counting and truss decomposition implementations on the next generation hardware – the high-bandwidth IBM Newell system, which contains two IBM POWER9 CPUs and four Nvidia V100 GPUs, connected by the NVLink 2.0 interconnect. This is in comparison to our previous results on IBM Minsky, which contains two IBM POWER8 CPUs and four Nvidia P100 GPUs, connected by the NVLink 1.0 interconnect.

4) We demonstrate that, on the Newell system, the collaborative algorithm can count triangles for the "Friendster" graph (119M nodes, 1.72G edges) within a minute.

The remainder of the paper is organized as follows. Sections II and III describe the optimized parallel algorithms for triangle counting and truss decomposition. Section IV describes the experimental setup and presents the performance of our optimized algorithms against prior submission on Minsky platform and NVGraph library. Finally, Section V concludes the paper with a summary and future directions.

II. Optimized Triangle Counting Algorithm

Triangle counting algorithm counts the total number of triangles in a graph. This can be achieved by first converting an undirected graph to a directed one, counting triangles for each edge, and then adding them together to get the total triangle count. For an edge, triangle count can be obtained by counting the number of common nodes in the adjacency lists of its head and tail nodes, with the help of a simple two-pointer intersection algorithm. Use of directed edges reduces the triangle counting effort by half. The edges can be ordered lexicographically or by their degrees [4].

In [2], we presented a parallel triangle counting algorithm whose central idea was to count triangles for each edge in parallel. Here, we present an optimized version of our prior implementation that provides orders of magnitude performance

benefit. Algorithm 1 depicts our improved implementation for the parallel triangle counting algorithm, which differs from our prior implementation in the following aspects:

1) We removed the degree-based preprocessing step of *Algorithm: Forward* [4] that is used for converting undirected graph into a directed one. We observed that the preprocessing overhead actually outweighs its benefits. The new implementation uses lexicographic ordering, which only requires us to read forward edges $e = (u, v)$, with $u < v$, for constructing the CSR structures. This significantly cuts down the preprocessing time, with minimal impact on the triangle counting kernel.
2) In our previous CSR structure, we stored edges as 64-bit integers, such that the lower 32 bits correspond to the head node and upper 32 bits correspond to the tail node. In the new implementation, the head and tail nodes are separated into two 32-bit integer arrays **Zs** and **Zd**, such that for an edge $e = (u, v)$, $\mathbf{Zs}[e] = u$ and $\mathbf{Zd}[e] = v$. Both these arrays are of size m, the number of edges in the graph.
3) In addition to the edge arrays, we also have an array P which is the pointer array. This array is of size $n + 1$, where n is the number of nodes in the graph. The element in the array $P[u]$ holds the index in the array **Zd** of the first edge connected to node u. The size of the adjacency list for the node u can be measured with the expression $P[u + 1] - P[u]$.
4) During execution, each edge is assigned to one thread, which counts the triangles for that edge using an optimized two-pointer intersection algorithm (Algorithm 1). The thread assigned to an edge e only needs to traverse the 32-bit array **Zd** to find common nodes, as opposed to traversing the 64-bit array in our previous implementation. We also eliminated some redundant memory accesses when only one of the pointers was moved forward. Both these intuitive optimizations proved to be extremely effective and provided a significant performance improvement over the previous implementation (see Section IV).

Algorithm 1 Optimized Two-pointer intersection

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

```
1: Δ(e) ← 0
2: u_ptr ← P[u]; v_ptr ← P[v]
3: u_end ← P[u + 1]; v_end ← P[v + 1]
4: w1 ← Zd[u_ptr]; w2 ← Zd[v_ptr]
5: while (u_ptr < u_end) ∧ (v_ptr < v_end) do
6:     if w1 < w2 then w1 ← Zd[++u_ptr];
7:     if w1 > w2 then w2 ← Zd[++v_ptr];
8:     if w1 = w2 then
9:         w1 ← Zd[++u_ptr];
10:        w2 ← Zd[++v_ptr]
11:        Δ(e) ← Δ(e) + 1
12:    end if
13: end while
```

III. Parallel Truss Decomposition

The goal of k-truss decomposition is to remove all edges that are not part of at least $k-2$ triangles. In our prior submission [2], if any of the edges are removed due to insufficient triangles, the algorithm only recounts the triangles for the edges that are affected by the removal before incrementing the k-value. For graphs with a large number of trusses, this is more efficient than recounting triangles for all the edges in each iteration of higher k-values. Here, we describe an improved implementation that provides orders of magnitude in performance over [2].

Algorithm 3 depicts our novel implementation for the efficient truss decomposition algorithm. The new implementation differs from the previous one in the following aspects:

1) The main difference is the way in which edge deletions are handled. Previously, the edges marked for deletion due to insufficient triangles, were removed from the edge array, immediately after the affected edges were identified in every iteration of the inner loop. Since edge removal is done using stream compaction, this caused a significant slowdown. In the new implementation, we use the following two operations for edge deletion:
   a) **Short update:** In this operation, the edge $e = (u, v)$ to be deleted is marked with a sentinel value $u = \mathbf{Zs}[e] = -1$ and $v = \mathbf{Zd}[e] = -1$ in the edge array and everything else, including the row pointer array, is left unmodified. This operation is performed within the while loop (Lines 7–33 in Algorithm 3), after the edges are marked for deletion.
   b) **Long update:** In this operation, we remove the "deleted" edges (with -1 for head and tail vertices), and update the edge array $E = (\mathbf{Zs}, \mathbf{Zd})$ using *stream compaction*. The row pointer array is also updated so that it is consistent with the new edge array. This operation is performed only one time for each k value, after all the triangle counting/edge deletion iterations are finished (Line 34). The updated edge arrays represent the "clean" k-truss graph, which also serves as an input for $(k + 1)^{\text{th}}$ iteration. In other words, long update is a cleanup step after multiple short updates.
2) The triangle-counting function is modified to handle sentinel values in the edge arrays after the "short update" operation. This function still uses the standard two-pointer intersection algorithm, with an additional condition that if any of the pointers encounter a sentinel value (-1) then that pointer is simply incremented. The `if statements` in the modified triangle counting/enumeration algorithm are depicted in Algorithm 2.
3) We create and maintain an array $\mathbf{I}_r$, with length equal to the number of edges. In the location $\mathbf{I}_r[e]$, we use binary search to gather the pointer to the reverse edge

$e' = (v, u)$ (Lines 3–6). The array $\mathbf{I}_r$ is populated once for each k value, to account for possible adjacency list updates in the previous iteration. This array is instrumental in identifying the forward and reverse indices of the newly affected edges (Lines 20–25). In our previous implementation, the indices of the newly affected forward and reverse edges were identified using binary search, which increased communication and caused slowdown. The new array $\mathbf{I}_r$ provides a quick look up of the reverse edges and allows us to dramatically reduce the frequency of binary search in our new implementation.

4) Within the inner loop, we create an array of affected edges E_{aff} using *stream compaction* (Line 8). This array contains the edges that were affected due to potential deletions in the previous iteration, and it serves as the input to the next iteration of the while loop. For the very first iteration, all of the edges are marked as "affected." Additionally, to reduce the triangle counting/enumeration effort by half, we only keep the forward edges (with $u < v$) in the affected edge array, and remove the reverse edges (with $u > v$). Note that E_{aff} is merely a working copy of the edge array. The actual edge array $E = (\mathbf{Zs}, \mathbf{Zd})$ contains both forward and reverse edges, which is a necessity for counting/enumerating triangles for truss decomposition.
5) If an edge $e = (u, v) \in E_{\text{aff}}$ has triangle count $< k - 2$, then its corresponding forward and reverse edges (from array E) need to be marked for deletion (Line 11-27). In our previous implementation, the indices of forward and reverse edges were identified using binary search, which increased communication and resulted in a slowdown. In the new implementation, we create and maintain two additional arrays: Forward edge indices $\mathbf{I}'_f$ and reverse edge indices $\mathbf{I}'_r$, whose lengths are equal to that of E_{aff}. For each edge $e = (u, v) \in E_{\text{aff}}$, in the location $\mathbf{I}'_f[e]$ stores the pointer to the same edge $e \in E$ and $\mathbf{I}'_r[e]$ stores the pointer to the reverse edge $e' = (v, u) \in E$. These arrays can be populated during the creation of E_{aff} (Line 8).

Algorithm 2 Triangle counting/enumeration for k-truss

1: **if** $(w_1 = -1) \vee (w_1 < w_2)$ **then** $w_1 \leftarrow \mathbf{Zd}[++u\_ptr]$;
2: **if** $(w_2 = -1) \vee (w_1 > w_2)$ **then** $w_2 \leftarrow \mathbf{Zd}[++v\_ptr]$;
3: **if** $(w_1 \neq -1) \wedge (w_2 \neq -1) \wedge (w_1 = w_2)$ **then**
4: $\quad \Delta(e) \leftarrow \Delta(e) + 1$; and/or ▷ Triangle counting
5: $\quad W \leftarrow W \cup \{w_1\}$; ▷ Triangle enumeration
6: $\quad w_1 \leftarrow \mathbf{Zd}[++u\_ptr]$; $w_2 \leftarrow \mathbf{Zd}[++v\_ptr]$;
7: **end if**

IV. Computational Experiments

The Graph Challenge [1] organizers provide python reference implementations for triangle-counting and k-truss decomposition. The results presented in this work produce the same outputs as the provided reference implementations. We used preprocessed data sets provided by the organizers in our experiments. Some of the datasets used in this work were chosen to facilitate direct comparison with previous Graph Challenge winning submissions. For brevity, we avoid reporting numbers for similar graphs from same community.

Algorithm 3 Efficient truss decomposition

Input: $G = (V, E)$
Output: k-truss for $3 \leq k \leq k_{max}$

1: $k \leftarrow 3$
2: Mark all $e \in E$ as "affected"
3: **for each** $e = (u, v) \in E$ **do** ▷ Parallel for
4: $\quad$ Binary search $e' = (v, u) \in E$
5: $\quad \mathbf{I}_r[e] \leftarrow e'$
6: **end for**
7: **while** $true$ **do**
8: $\quad (E_{\text{aff}}, \mathbf{I}'_f, \mathbf{I}'_r) \leftarrow StmCmp(E,$ "affected and $u < v$")
9: $\quad$ **if** $E_{\text{aff}} = \emptyset$ **then goto** 34;
10: $\quad$ Mark all $e \in E$ as "not affected"
11: $\quad$ **for each** $e = (u, v) \in E_{\text{aff}}$ **do** ▷ Parallel for
12: $\quad\quad \Delta(e) \leftarrow |adj(u) \cap adj(v)|$ ▷ Algorithm 2
13: $\quad\quad$ **if** $\Delta(e) < k - 2$ **then**
14: $\quad\quad\quad$ **Stage 1:**
15: $\quad\quad\quad$ Lookup $e' = (u, v) \leftarrow \mathbf{I}'_f[e]$
16: $\quad\quad\quad$ Lookup $e'' = (v, u) \leftarrow \mathbf{I}'_r[e]$
17: $\quad\quad\quad$ Mark e' and e'' in E as "delete"
18: $\quad\quad\quad$ **Stage 2:**
19: $\quad\quad\quad$ $W \leftarrow adj(u) \cap adj(v)$ ▷ Algorithm 2
20: $\quad\quad\quad$ **if** $W \neq \emptyset$ **then**
21: $\quad\quad\quad\quad$ $e_1 \leftarrow (u, w)$; $e_2 \leftarrow (v, w)$; s.t. $w \in W$
22: $\quad\quad\quad\quad$ Lookup $e'_1 = (w, u) \leftarrow \mathbf{I}_r[e_1]$
23: $\quad\quad\quad\quad$ Lookup $e'_2 = (w, v) \leftarrow \mathbf{I}_r[e_2]$
24: $\quad\quad\quad\quad$ Mark e_1, e_2, e'_1, e'_2 as "affected"
25: $\quad\quad\quad$ **end if**
26: $\quad\quad$ **end if**
27: $\quad$ **end for**
28: $\quad$ **for each** $e = (u, v) \in E$ **do** ▷ Parallel for
29: $\quad\quad$ **if** e labeled "delete" **then**
30: $\quad\quad\quad$ $u \leftarrow -1$; $v \leftarrow -1$ ▷ Short update
31: $\quad\quad$ **end if**
32: $\quad$ **end for**
33: **end while**
34: $E \leftarrow StmCmp(E,$ "not deleted") ▷ Long update
35: Output E as k-truss edge list
36: **if** $E \neq \emptyset$ **then** $k \leftarrow k + 1$ and **goto** 2

The performance of triangle-counting and k-truss decomposition is evaluated on two hardware platforms, the IBM Minsky and Newell using our collaborative scheme proposed in [2]. First, we shall provide details about our evaluation hardware and then discuss various graph performance results. And then, we briefly introduce terminologies of collaborative schemes.

A. Minsky and Newell Evaluation Hardware

The IBM "Minsky"[5] and "Newell"[6] machines share a common architectural topology. Table I summarizes the components and interconnects that make up the systems. The

TABLE I
IBM MINSKY AND NEWELL ARCHITECTURE COMPARISON.

	Minsky	Newell
CPU	POWER8	POWER9
System RAM	512 GB (230GB/s)	512 GB (240 FB/s)
GPU	NVidia P100 [7]	NVidia V100 [8]
CPU-CPU Interconnect	38.4 GB/s X-bus	64 GB/s X-bus
NVLink Triad Interconnect	V1.0 x2 (80 GB/s)	V2.0 x3 (150 GB/s)

systems comprise two triads, CPU0-GPU0-GPU1 and CPU1-GPU2-GPU3. Components within a triads are connected by the listed "triad interconnect," and the CPUs in each triad are connected by the "CPU-CPU interconnect." The Newell system features substantially improved interconnect bandwidth in addition to generational improvements in CPU and GPU performance. Minsky's triad interconnect is NVLink 1.0 x2, which couples each component with two 20GB/s unidirectional lanes, for a total bi-directional bandwidth of 80GB/s. In Newell, this interconnect is improved to NVLink 2.0 x3, which features three 25GB/s lanes for an aggregate bidirectional bandwidth of 150 GB/s.

B. Collaborative Algorithm

We enable collaborative (CPU+GPU) memory management schemes for both k-truss decomposition and the triangle counting algorithms. Similar to our prior submission, we support three memory management schemes, namely, `Single GPU Memory`, `ZeroCopy Memory` and `Unified Memory`, where all the adjacency matrix resides in the GPU memory, in the host pageable memory and in the system coherent domain respectively. Collaborative execution on graphs are performed with `ZeroCopy memory` and `Unified memory` where we use all 160 CPU threads and 4 GPUs such that 90% of work was done by GPUs and 10% by CPU.

C. Evaluation Methodology

For consistency with Graph Challenge evaluation metrics, we measured total number of edges, nodes, execution time and edges (bidirectional) per second as metric in our evaluation. We measured the execution time after the edges are read into the host memory, therefore it includes any preprocessing time and not the I/O reading time. Our code is compiled with `nvcc` 9.2.88 and `gcc` 4.8.5, and the GPU uses CUDA driver 396.26.

First, we outline the performance gains of optimized algorithm in Newell machine over our 2017 submission [2]. Second, we compare our triangle counting algorithm with NVIDIA's NVGraph library [3] (Ref. Table II and III). Third, we show Newell machine is about 2-3 times faster than Minsky for most of the graph dataset with our optimized algorithms with different memory management schemes (Ref. Figure 1). Finally, we run very large graphs using all the 4 GPUs and 160 CPU threads to showcase how the optimized triangle counting algorithm can crunch billions of edges in a less than a minute (Ref. Table IV and V).

NVIDIA NVGraph Implementation Details: We based our NVGraph library implementation on the sample example provided at [3]. First, we load lower adjacency matrix to host memory in CSR format. We explicitly copy the graph to GPU memory to mitigate the copy overhead. Then we call `nvgraphTriangleCount` with `NVGRAPH_CSR_32` flag. We use `Single GPU` memory management scheme of our triangle counting algorithm for fair comparison with NVGraph. For both implementations, total execution time includes GPU kernel execution time and copy time to the GPU.

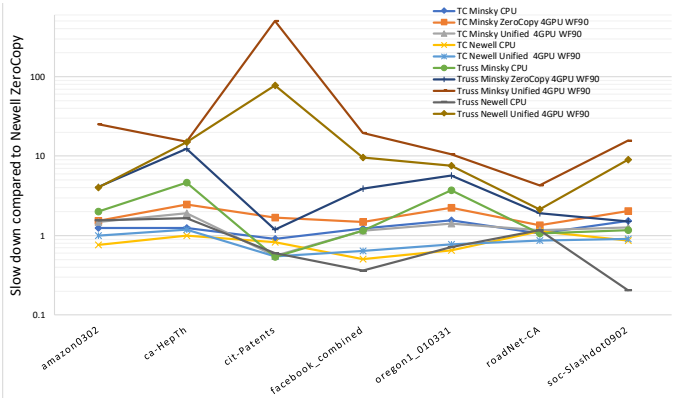

Figure 1. Slowdown of different memory management schemes for triangle counting (TC) and Truss decomposition (Truss) algorithm relative to the ZeroCopy code running on Newell. Although unified memory performance significantly improved in Newell, it is still slow for truss decomposition (Both algorithms are compared against its Newell's ZeroCopy implementation).

D. Results

The results for the various computational experiments are compiled in Tables II-V and Figure 1. From these results, the following observations can be made:

1) Table II: For triangle counting, our proposed optimizations result in an average of 47.6x speedup on SNAP dataset, and an average of 4.84x speedup on synthetic dataset over our prior submission [2]. Our optimized algorithm can process up to 334M edges/second for problems where the CSR can reside in the GPU memory.
2) Table II: The `Single GPU` version of our optimized triangle counting implementation is on average 151.7x (max 476x) faster on SNAP data set, and on average 3.7x (max 6.25x) faster on synthetic data set, compared to Nvidia's NVGraph library. For large synthetic graphs, the speedup factor reduces to slightly lower than 1x.
3) Table III: The `single GPU` version of our optimized truss decomposition implementation is on average, 5.7x faster (max 13.52x) on SNAP data set, and on average 10.91x faster (max 13.9x) on synthetic data set, compared to our prior submission. The optimized algorithm can process up to 115.7M edges/second for problems where the CSR can reside in the GPU memory.
4) In majority of triangle counting tests, we observed that the Newell machine outperforms the Minsky machine by 1.77x in CPU-OpenMP implementation, 1.8x in `ZeroCopy` implementation with 4 GPus, and 2.4x in `Unified` memory implementation with 4 GPUs. The speedup can be attributed to faster interconnect bandwidth, and newer generation of CPUs and GPUs.

TABLE II
TRIANGLE COUNTING BENCHMARKS ON DIFFERENT GRAPHS USING SINGLE GPU CONFIGURATION

Graph [1]	n	m	TC	Edges/s			Speedup	
				Minsky (2017 Submission)	Newell NVGraph	Newell (2018 Submission)	2017 vs. 2018	NVGraph vs 2018
as20000102	6,474	12,572	6,584	87,624	27,092	9,179,324	104.76	338.82
ca-GrQc	5,242	14,484	48,260	97,425	34,146	11,389,918	116.91	333.57
oregon1_010331	10,670	22,002	17,144	152,942	51,516	12,561,745	82.13	243.84
ca-HepTh	9,877	25,973	28,339	170,932	60,985	16,373,199	95.79	268.48
p2p-Gnutella04	10,876	39,994	934	278,434	92,201	19,227,384	69.06	208.54
as-caida20071105	26,475	53,381	36,365	362,818	46,531	22,148,726	61.05	476.00
facebook_combined	4,039	88,234	1,612,010	590,514	208,333	60,860,189	103.06	292.13
ca-CondMat	23,133	93,439	173,361	606,459	216,614	43,642,242	71.96	201.47
ca-HepPh	12,008	118,489	3,358,499	806,558	273,141	54,188,075	67.18	198.39
email-Enron	36,692	183,831	727,044	1,260,861	423,159	68,954,329	54.69	162.95
ca-AstroPh	18,772	198,050	1,351,441	1,339,316	467,160	65,039,688	48.56	139.22
loc-brightkite_edges	58,228	214,078	494,728	1,461,942	473,018	40,508,748	27.71	85.64
cit-HepTh	27,770	352,285	1,478,735	2,364,583	811,563	77,766,095	32.89	95.82
email-EuAll	265,214	364,481	267,313	2,410,955	320,800	61,209,886	25.39	190.80
soc-Epinions1	75,879	405,740	1,624,481	-	928,475	105,567,437	-	113.70
cit-HepPh	34,546	420,877	1,276,868	2,743,282	963,862	127,196,634	46.37	131.97
soc-Slashdot0811	77,360	469,180	551,724	3,026,851	1,068,382	110,349,986	36.46	103.29
soc-Slashdot0902	82,168	504,230	602,592	-	1,150,503	114,767,818	-	99.75
amazon0302	262,111	899,792	717,719	5,744,221	1,837,720	163,615,502	28.48	89.03
roadNet-PA	1,088,092	1,541,898	67,150	9,156,436	3,442,643	127,660,219	13.94	37.08
roadNet-TX	1,379,917	1,921,660	82,869	11,096,636	4,382,919	135,405,980	12.20	30.89
flickrEdges	105,938	2,316,948	107,987,357	12,362,594	5,181,185	215,880,798	17.46	41.67
amazon0312	400,727	2,349,869	3,686,467	14,494,180	4,496,023	288,205,484	19.88	64.10
amazon0505	410,236	2,439,437	3,951,063	15,085,724	4,725,439	273,002,037	18.10	57.77
amazon0601	403,394	2,443,408	3,986,507	15,037,375	4,709,385	271,908,517	18.08	57.74
roadNet-CA	1,965,206	2,766,607	120,676	15,357,241	6,141,124	139,467,006	9.08	22.71
cit-Patents	3,774,768	16,518,947	7,515,023	60,333,708	29,686,790	334,014,352	5.54	11.25
graph500-scale18-ef16	174,147	7,600,696	82,287,285	30,958,048	15,408,326	203,201,787	6.56	13.19
graph500-scale19-ef16	335,318	15,459,350	186,288,972	31,373,108	24,857,258	155,426,618	4.95	6.25
graph500-scale20-ef16	645,820	31,361,722	419,349,784	25,717,713	38,418,934	113,259,001	4.40	2.95
graph500-scale21-ef16	1,243,072	63,463,300	935,100,883	20,010,822	42,544,963	90,784,271	4.54	2.13
graph500-scale22-ef16	2,393,285	128,194,008	2,067,392,370	15,549,149	45,919,052	70,330,803	4.52	1.53
graph500-scale23-ef16	4,606,314	258,501,410	4,549,133,002	12,269,656	38,612,314	49,346,912	4.02	1.28
graph500-scale24-ef16	8,860,450	520,523,686	9,936,161,560	-	34,753,053	35,045,368	-	1.01
graph500-scale25-ef16	17,043,780	1,046,934,896	21,575,375,802	-	28,003,095	27,339,066	-	0.98

TABLE III
TRUSS DECOMPOSITION BENCHMARKS ON DIFFERENT GRAPHS USING SINGLE GPU CONFIGURATION

Graph [1]	n	m	k_{max}	Edges/s		Speedup
				Minsky (2017 Submission)	Newell (2018 Submission)	vs. 2017 submission
as20000102	6,474	25,144	10	121,422	463,223	3.81
ca-GrQc	5,242	28,968	44	120,183	202,471	1.68
oregon1_010331	10,670	44,004	16	160,122	453,213	2.83
ca-HepTh	9,877	51,946	32	237,225	407,249	1.72
p2p-Gnutella04	10,876	79,988	4	528,298	7,140,476	13.52
as-caida20071105	26,475	106,762	16	281,055	670,506	2.39
facebook_combined	4,039	176,468	97	26,766	159,387	5.95
ca-CondMat	23,133	186,878	26	474,945	1,416,010	2.98
ca-HepPh	12,008	236,978	239	130,993	263,878	2.01
email-Enron	36,692	367,662	22	123,583	963,079	7.79
ca-AstroPh	18,772	396,100	57	191,690	862,452	4.50
loc-brightkite_edges	58,228	428,156	43	239,730	889,012	3.71
cit-HepTh	27,770	704,570	30	220,285	1,159,954	5.27
email-EuAll	265,214	728,962	20	271,220	2,623,830	9.67
soc-Epinions1	75,879	811,480	33	-	1,181,446	-
cit-HepPh	34,546	841,754	25	471,684	1,905,473	4.04
soc-Slashdot0811	77,360	938,360	35	561,612	1,907,923	3.40
soc-Slashdot0902	82,168	1,008,460	36	-	1,979,478	-
amazon0302	262,111	1,799,584	7	4,923,191	33,140,027	6.73
roadNet-PA	1,088,092	3,083,796	4	14,738,996	93,106,578	6.32
roadNet-TX	1,379,917	3,843,320	4	16,993,209	98,403,250	5.79
flickrEdges	105,938	4,633,896	574	138,952	641,408	4.62
amazon0312	400,727	4,699,738	11	4,081,947	26,760,310	6.56
amazon0505	410,236	4,878,874	11	2,840,998	25,936,061	9.13
amazon0601	403,394	4,886,816	11	2,392,998	28,369,638	11.86
roadNet-CA	1,965,206	5,533,214	4	22,254,544	115,721,301	5.20
cit-Patents	3,774,768	33,037,894	36	3,711,498	41,798,278	11.26
graph500-scale18-ef16	174,147	7,600,696	159	79,320	1,102,217	13.90
graph500-scale19-ef16	335,318	15,459,350	213	83,976	969,474	11.54
graph500-scale20-ef16	645,820	31,361,722	284	74,387	751,343	10.10
graph500-scale21-ef16	1,243,072	63,463,300	373	58,739	476,772	8.12
graph500-scale22-ef16	2,393,285	128,194,008	485	-	296,406	-
graph500-scale23-ef16	4,606,314	258,501,410	625	-	187,628	-

TABLE IV
ZERO-COPY AND UNIFIED MEMORY TRIANGLE COUNTING BENCHMARKS ON LARGE GRAPHS

Graph [1]	n	m	TC	Single GPU		Zero-copy (4 GPUs)		Unified (4 GPUs)	
				Time (s)	Edges/s	Time (s)	Edges/s	Time (s)	Edges/s
flickrEdges	105,938	2,316,948	107,987,357	0.011	215,880,798	0.213	10,867,996	0.026	88,016,563
cit-Patents	3,774,768	16,518,947	7,515,023	0.049	334,014,352	0.196	84,071,878	0.140	118,021,984
Kmer - Graph5	55,042,369	58,608,800	1,443	0.092	638,440,087	0.490	119,559,044	0.603	97,132,052
Network - Graph5	226,196,185	240,023,945	26	5.925	40,511,923	9.139	26,262,472	7.091	33,848,266
graph500-scale18-ef16	174,147	7,600,696	82,287,285	0.037	203,201,787	1.287	5,906,233	1.356	5,606,713
graph500-scale19-ef16	335,318	15,459,350	186,288,972	0.099	155,426,618	2.109	7,329,200	2.399	6,445,287
graph500-scale20-ef16	645,820	31,361,722	419,349,784	0.277	113,259,001	4.323	7,255,159	2,841.665	11,036
graph500-scale21-ef16	1,243,072	63,463,300	935,100,883	0.699	90,784,271	19.481	3,257,717	7,018.016	9,043
graph500-scale22-ef16	2,393,285	128,194,008	2,067,392,370	1.823	70,330,803	77.586	1,652,290	17,019.189	7,532
graph500-scale23-ef16	4,606,314	258,501,410	4,549,133,002	5.238	49,346,912	85.741	3,014,908	>18,000	-
graph500-scale24-ef16	8,860,450	520,523,686	9,936,161,560	14.853	35,045,368	367.430	1,416,659	>18,000	-
graph500-scale25-ef16	17,043,780	1,046,934,896	21,575,375,802	38.294	27,339,066	1,266.137	826,873	>18,000	-
Friendster	119,432,957	1,799,999,986	191,716	-	-	57.036	31,558,975		

TABLE V
ZERO-COPY TRUSS DECOMPOSITION BENCHMARKS ON LARGE GRAPHS

Graph[1]	n	m	$k_{\max}$	Single GPU		Zero-copy (4 GPUs)	
				Time (s)	Edges/s	Time (s)	Edges/s
flickrEdges	105,938	2,316,948	574	7.225	320,704	79.607	29,105
cit-Patents	3,774,768	16,518,947	36	0.790	20,899,139	4.917	3,359,763
Kmer - Graph5	55,042,369	58,608,800	3	0.497	117,814,922	2.757	21,261,171
graph500-scale18-ef16	174,147	7,600,696	159	6.896	1,102,217	61.353	123,885
graph500-scale19-ef16	335,318	15,459,350	213	15.946	969,474	192.019	80,509
graph500-scale20-ef16	645,820	31,361,722	284	41.741	751,343	683.304	45,897
graph500-scale21-ef16	1,243,072	63,463,300	373	133.110	476,772	2,868.196	22,127
graph500-scale22-ef16	2,393,285	128,194,008	485	432.494	296,406	9,709.680	13,203
graph500-scale23-ef16	4,606,314	258,501,410	625	1,377.733	187,628	>10,000	-

TABLE VI
SINGLE GPU TRIANGLE COUNTING COMPARISON WITH 2017 CHAMPIONS.

Graph [1]	**Speedup over [9]**	**Speedup over [10]**
amazon0312	1.55	1.24
cit-HepTh	1.28	0.71
cit-Patents	1.04	1.45
email-EuAll	0.93	0.62
soc-Slashdot0902	1.43	1
graph500-scale18	1.59	2.06
graph500-scale23	0.97	1.46
graph500-scale24	1.02	1.44
graph500-scale25	1.47	1.49

5) The performance improvement is magnified in truss decomposition. We observe that, on average we achieve 3.04x, 3.66x and 3.0x speedup factors over the Minsky machine, for CPU-OpenMP, 4 GPU `ZeroCopy` memory, and 4 GPU `Unified` memory implementations (resp.), suggesting that majority of the speedup was obtained from algorithmic optimizations.
6) Similar to [2], we observed that the `Unified memory` implementation performs worse than the `ZeroCopy` implementation, which in turn performs worse than the `Single GPU` one. Most of the performance loss in the `Unified memory` and `ZeroCopy` implementations can be attributed to the limited NVLink bandwidth. Although it is improved in the Newell system, the NVLink bandwidth is still much lower than the GPU memory bandwidth. A promising research direction would be to improve data locality by partitioning the graph into pieces that can be contained within the GPU memory.
7) Tables IV shows that our optimized triangle counting algorithms with `ZeroCopy memory (4GPU)` was able to count triangles for the billion-edge Friendster graph within a minute, which is quite remarkable.

E. Comparison with 2017 Triangle Counting Champions

In Table VI, we compare our triangle counting performance with the performance of two champions from last year [9], [10]. Our `Single GPU` triangle counting configuration in Newell system outperforms [9] by up to 1.59x and [10] by up to 2.06x for most graphs. For a few smaller graphs we exhibit slightly lower performance compared to 2017 champions.

V. CONCLUSION

To summarize, we proposed and discussed various algorithmic optimizations for triangle counting and truss decomposition over our prior submission. With these optimizations, we achieved 47.6x and 5.7x on average (max 117x and 13.52x) on triangle counting and truss decomposition over our prior submission on SNAP datasets and up to 6.5x and 13.9x on synthetic datasets. We also showed that our efficient `ZeroCopy` implementation of the triangle counting algorithm can process billions of edges from the Friendster dataset, in less than a minute. We discussed the performance benefits offered by the new hardware and established that most of the improvements are due to algorithmic optimizations. This makes us believe that there are still many improvements in the algorithmic space that are yet to be explored. Partitioning the graph into pieces that reside in the native memory of the processing element (RAM for CPU and device memory for GPU), is one such example. This, however, is a non-trivial task, and it is left as a future research direction.

ACKNOWLEDGMENTS

The authors acknowledge Sitao Huang, Mert Hidayetoglu, Raimi S Shah, Karm Nagi and Volodymyr Kindratenko for their help. This work is supported by IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM AI Horizons Network. This work utilizes resources supported by the National Science Foundations Major Research Instrumentation program, grant #1725729, as well as the University of Illinois at Urbana-Champaign.

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] K. Date, K. Feng, R. Nagi, J. Xiong, N. S. Kim, and W. M. Hwu, "Collaborative (cpu + gpu) algorithms for triangle counting and truss decomposition on the minsky architecture: Static graph challenge: Subgraph isomorphism," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2017, pp. 1–7.

[3] "NVIDIA NVGraph api reference documentation," Nvidia, 2018. [Online]. Available: https://docs.nvidia.com/cuda/nvgraph/index.html#nvgraph-api-reference

[4] 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

[5] A. B. Caldeira, V. Haug, and S. Vetter, "IBM power system 822LC for High Performance Computing introduction and technical overview," *IBM Redbooks*, 2016.

[6] A. B. Caldeira, "IBM power system AC922 introduction and technical overview," *IBM Redbooks*, 2018.

[7] (2016) NVIDIA Tesla P100. Nvidia. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[8] (2017) NVIDIA Tesla V100 GPU architecture. Nvidia. [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[9] M. Bisson and M. Fatica, "Static graph challenge on gpu," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2017, pp. 1–8.

[10] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, "Fast linear algebra-based triangle counting with kokkoskernels," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2017, pp. 1–7.