Fast Linear Algebra-Based Triangle Counting with KokkosKernels

Michael M. Wolf, Mehmet Deveci, Jonathan W. Berry, Simon D. Hammond, Sivasankaran Rajamanickam
Center for Computing Research, Sandia National Laboratories
Albuquerque, NM 87185
{mmwolf,mndevec,jberry,sdhammo,srajama}@sandia.gov

***Abstract*—Triangle counting serves as a key building block for a set of important graph algorithms in network science. In this paper, we address the IEEE HPEC Static Graph Challenge problem of triangle counting, focusing on obtaining the best parallel performance on a single multicore node. Our implementation uses a linear algebra-based approach to triangle counting that has grown out of work related to our miniTri data analytics miniapplication [1] and our efforts to pose graph algorithms in the language of linear algebra. We leverage KokkosKernels to implement this approach efficiently on multicore architectures. Our performance results are competitive with the fastest known graph traversal-based approaches and are significantly faster than the Graph Challenge reference implementations, up to 670,000 times faster than the C++ reference and 10,000 times faster than the Python reference on a single Intel Haswell node.**

I. Background

A. Triangle Counting

The problem of *triangle counting* in an undirected graph G is to find a single integer: the number of three-cycles (triangles) in G. The number of triangles in a graph is an important metric that is used in many network analysis applications, including social network analysis [2], spam detection [3], link recommendation [4], and dense neighborhood graph discovery [5]. Furthermore, triangle counting serves as a building block or starting point for additional important graph algorithms such as triangle enumeration or listing, k-truss computation [6], and subgraph isomorphism. This motivates triangle counting as one of the Graph Challenge problems [7].

In the case of dense graphs, triangle counting is not practical since there are $O(n^3)$ triangles and hence cubic work. However, many real graphs (social networks and others) have the welcome property that the number of triangles is $O(n)$. This is shown in Berry et al. [8], along with an argument that a simple algorithm called MinBucket [9] does this optimal amount of work. There is extensive literature on triangle counting, including approximate methods [10], which bases its counts on a subset of the spectrum of G, sampling methods [11], which samples two-paths or wedges and generates provably good bounds on triangle counts, and big data methods [12], [13], which leverage MapReduce or similar paradigms to obtain triangle counts for graphs that do not fit on one machine.

In this paper, we focus on a linear algebra-based approach to triangle counting that has grown out of work related to our miniTri triangle-based miniapplication [1] and our efforts to pose graph algorithms in the language of linear algebra. We focus on triangle counting on a single compute node, leveraging KokkosKernels [14] to implement this approach efficiently. We obtain results that are competitive with the fastest known graph traversal-based approaches.

B. Linear Algebra Primitives for Graph Algorithms

The Graph BLAS [15], [16] community has been working to standardize a set of building blocks to solve graph problems in the language of sparse linear algebra. Many graph computations can be efficiently written in terms of linear algebra [17], including breadth-first search, betweenness centrality, and triangle counting/enumeration [1], [18] (discussed further in the next subsection). To complement this algorithmic work, there are several implementations related to the Graph BLAS effort: CombBLAS [19], Graphulo [20], D4M [21], GPI [22], GraphPad [23], and the GraphBLAS Template Library [24].

The promise of high performance is one of the appealing aspects of this linear algebra approach to graph algorithms (another being the synergy with explicitly algebraic graph algorithms such as spectral partitioning). If hardware vendors could optimize this small set of sparse linear algebra operations, then a large number of graph kernels could be computed in a performant manner. However, it is still an open question whether this performance will be realized. There have been some promising results (e.g., [25]). However, linear algebra-based graph kernels have traditionally underperformed in our comparisons with graph traversal library counterparts. In this paper, we show positive results for a new linear algebra-based triangle counting implementation. We believe that this is a potentially disruptive event that could affect the design of kernels within generic graph libraries such as the MTGL [26].

C. Linear Algebra-Based Triangle Counting Algorithms

In this subsection, we discuss two linear algebra-based triangle counting algorithms found in the literature.

1) Adjacency and Incidence Matrix Based Method: In previous work, we developed a Graph BLAS like approach for triangle enumeration in terms of an overloaded sparse matrix-matrix multiplication operation $C = A \cdot H$, where A and H are the the adjacency and incidence matrices of the graph, respectively [1]. A simplified version of this algorithm can be used for triangle counting and is used in the Graph Challenge

C++, Python, and Matlab reference implementations. Although this approach is useful for more complex graph calculations such as triangle enumeration and k-truss computation, it is typically less performant than the adjacency matrix-based triangle counting algorithm that is presented below.

2) Adjacency Matrix Based Method: In previous work, Azad, et al. described a triangle counting algorithm in terms of sparse matrix-matrix multiplication followed by an element-wise matrix multiplication: $D = (L \cdot U). * A$, where A is the adjacency matrix for the graph, L is the lower triangular part of A, and U is the upper triangular part of A [18]. This algorithm was the basis for the Graph Challenge Julia reference implementation. For this algorithm, each entry $C(v_1, v_2)$ in the resulting matrix $C = L \cdot U$ contains a count of all wedges (paths of length 2) in the graph with endpoints v_1 and v_2 and different midpoints u_i. The subsequent element-wise matrix multiplication operation $D = C. * A$ filters out (or masks) all the wedges that have end points (v_1, v_2) that are not connected by edges in the graph and thus are not triangles.

Use of the triangular matrices L and U restricts the triangles found such that $v_1 > u_i, v_2 > u_i$ for a triangle formed from wedge $v_1 - u_i - v_2$. However, this method counts each triangle twice since it counts both (v_1, u_i, v_2) and (v_2, u_i, v_1) as separate triangles. Furthermore, implementing this triangle counting algorithm with the sparse matrix-matrix multiplication followed by the element-wise multiply is problematic since the number of wedges in a graph can be prohibitively large (much larger than the number of triangles). However, Azad, et al. explain how these operations can be fused with the element-wise multiply being incorporated as a mask [18].

For these linear algebra-based algorithms to be impactful, we need implementations of these algorithms in software frameworks that provide *performance-portability* - the ability to write kernels at an abstract level, yet exploit advanced architectures effectively. In this work, we leverage such a framework, Kokkos [27], and KokkosKernels, a library of linear algebraic kernels that use Kokkos, to achieve performance-portability for our triangle counting method.

D. KokkosKernels and Optimized SpGEMM

In a recent work, we introduced a Kokkos-based [27] SpGEMM method, KKMEM [28]. KKMEM is designed for portability, hence it runs and performs well on various architectures such as traditional CPUs, NVIDIA GPUs and Intel Knights Landing (KNL) architectures. The proposed methods in this paper use and extend the KKMEM algorithm.

KKMEM is a hierarchical 1D/2D row-wise algorithm. First, it assigns the multiplication of each row to threads. Then, different multiplications within the row are assigned to different vector lanes. KKMEM is a two-phase algorithm: the number of nonzeros of each row of the resulting matrix is calculated in the first (symbolic) phase, then the actual matrix values are computed in the second (numeric) phase. In graph problems, the numeric phase is not necessary when only the structure (and not the values) of the resulting matrix is desired.

KKMEM uses a compression technique to encode multiple columns of right hand side matrix with fewer integers. This reduces the number of operations and the memory requirements in the symbolic phase. This also allows using bitwise operations from union/intersection of different rows. KKMEM has the option to use sparse hashmap based accumulators or dense accumulators. A uniform memory pool data structure is used in the sparse hashmap for better memory scalability. However, the sparse accumulators require more operations than a dense accumulator. We choose dense accumulators for smaller problems and sparse accumulators otherwise. These data structures are used with minimal changes in this paper.

II. Triangle Counting Algorithm

In this section, we describe a new linear algebra-based triangle counting algorithm that we have designed and implemented for the Graph Challenge. We describe algorithm enhancements that can have significant impact on performance and discuss details of our KokkosKernels implementation.

A. Triangle Counting Algorithm: $(L \times L). * L$

We have implemented a new variant of the adjacency matrix-based triangle counting method described above and originally outlined in [18]. Using the lower triangle portion of the adjacency matrix (instead of the full adjacency matrix) on the right hand side of the element-wise multiply ($D = (L \times U). * L$) counts each triangle exactly once instead of twice as in ($D = (L \times U). * A$). However, each triangle is still "counted" twice, only one of which is counted after the element-wise multiplication. (i.e., for triangle (v_1, u_i, v_2), both (v_1, u_i, v_2) and (v_2, u_i, v_1) will be initially counted by SpGEMM).

A further improvement can be made by replacing U by L in the sparse matrix matrix multiplication: $D = (L \times L). * L$. This enhancement adds a constraint such that $C(v_1, v_2)$ (resulting from $L \times L$) is a nonzero if and only if $v_1 > v_2$, which reduces the wedges stored in C. Typically, we also see a reduction in the number of operations and runtime as a result of this constraint. Thus, for this challenge, we have chosen to implement $(L \times L). * L$ instead of $(L \times U). * A$.

B. KokkosKernels-Based Triangle Counting

Here, we describe the KokkosKernels-based implementation of our $L \times L$ based triangle counting algorithm and discuss related implementation details. Algorithm 1 outlines the basic steps that takes place in the overloaded KokkosKernels KKMEM operation that is used for triangle counting. The matrix multiplication traverses each row v of L (each vertex in the graph). It creates a local hashmap $\mathcal{H}$, and inserts the neighboring vertices of v, $L(v)$ (corresponding to nonzero columns in row $L(v)$). For each nonzero column u of $L(v)$, the nonzero columns y of $L(u)$ are traversed. If a column (vertex) y is in the previously created hashmap $\mathcal{H}$, this corresponds to a triangle.

The overall complexity of Line 2 is $O(E)$. For a vertex u, the amount of the work of Line 4 is $d^L(u)$ (the number

Algorithm 1 $(L \times L)$.*L

```
Require: Matrix $L$
1: for each row (vertex) $v \in L$ do
2:   Create a hashmap $\mathcal{H}$, and insert columns $L(v)$ into $\mathcal{H}$.
3:   for each nonzero column (vertex) $u \in L(v)$ do
4:     for each nonzero column (vertex) $y \in L(u)$ do
5:       Query $y$ in $\mathcal{H}$
```

of nonzeros in row u) and this line is executed for u, as the number of nonzeros in column u of L or $d^{L'}(u)$. As a result, each row u results in $d^{L'}(u) \times d^{L}(u)$ amount of work.

C. *"Reordering" adjacency matrices*

Since only the lower portion of the matrix is used in the computation, the performance of $(L \times L). * L$ depends on the order of the rows in the adjacency matrix and can be greatly improved with a good ordering (e.g., one that reduces the number of nonzeros in L). As explained in the previous section, the cost a row (v) in L is proportional to $d^{L}(v)$, the number of nonzeros in that row. However, the row is accessed as many times as it appears in the columns of L, which is $d^{L'}(v)$. Thus, the required number of multiplications for the row is $d^{L'}(v) \times d^{L}(v)$, and the overall number of multiplications is $\sum_{v \in V} d^{L'}(v) \times d^{L}(v)$.

Reordering to reduce the operation count for $L \times L$ is complicated (unlike $L \times U$) since reducing $d^{L}(v)$ for a given v tends to increase $d^{L'}(v)$, and vice versa. For this challenge, we chose to sort the rows by decreasing vertex degree (largest number of nonzeros at the top), which tends to decrease $d^{L}(v)$ and increase $d^{L'}(v)$ for high degree vertices.

It is also important to note that we do not explicitly reorder the matrix rows when forming the matrix L since this would be needlessly expensive. Instead, we keep the row ordering the same and choose the maximum column number in which a row can have a nonzero based on the row number that the row would obtain if we had performed this reordering. Thus, the resulting matrix L is not a lower triangular matrix but a permutation of a lower triangular matrix.

D. *Masking non-triangle wedges*

As previously described by Azad et al. [18], the element-wise multiplication can be combined with the sparse matrix-matrix multiplication operation, with $D = (L \times U). * A$ being replaced by a single function $D = \text{MaskedSpGEMM}(L, U, A)$. This operation avoids the creation of all wedges, greatly reduces the peak memory usage, and can reduce the number of operations. We apply a similar masking technique to our new algorithm and implement $D = \text{MaskedSpGEMM}(L, L, L)$ with KokkosKernels. In order to avoid storing all the wedges, when a row i in L is multiplied with L, the resulting nonzeros are masked to include only the nonzeros that exist in row i of L and thus could be a triangle.

III. Results

A. *Reference Implementations*

It was challenging working with the serial reference implementations of this Graph Challenge. The Python, Julia, and Matlab codes explicitly store counts for all wedges in the graph. This is problematic since typically there are many more wedges in a graph than there are triangles. We saw this as we ran out of memory when running these reference implementation for the larger graphs. The C++ reference implementation that we previously developed [1] does not suffer from this (since it only stores the triangles) but is significantly slower than the Python reference. Our compromise was to run both the Python and C++ reference implementations, so that we could run all the graphs with C++ and get a more reasonable baseline with the Python reference when memory was not an issue. For the largest graphs, the predicted runtimes for the C++ reference implementation would require months of runtime, so we provide a rough execution time estimate for these (average time per row for a week long run multiplied by the number of rows).

In addition to the two Graph Challenge reference implementations, we also compare against the parallel merge-based triangle counting method (TCM) proposed in [29], which we consider state-of-the-art and has shown some of the best results in the open literature. No previous linear algebra-based method has achieved runtimes similar to a specialized graph method such as TCM.

B. *Datasets*

Table I lists the 25 graphs used in our numerical experiments, along with the number of vertices ($|V|$), edges ($|E|$), and triangles ($|T|$) in the graphs. We chose the 20 most time consuming graphs (based on the reference implementations) and five additional large graphs (highlighted in the table) that are commonly used in the literature from [30], in order to have some large non synthetic graphs in this data set. We used the Matrix-Market formatted files provided by the Graph Challenge for all graph challenge problems except for Friendster, which we converted from the TSV format (due to an initial error with the Matrix-Market file). We used publicly available Matrix-Market files from the SuiteSparse Matrix Collection for the five additional problems, following the Graph Challenge procedure of symmetrizing the matrices. To run TCM, we converted these matrix-market format to the Ligra format that is used by TCM [29].

C. *Performance Results*

The bulk of our numerical experiments we ran on a compute node with dual Intel Xeon Haswell processors (E5-2698 v3 @ 2.30GHz, 32 cores total, with 2 hyperthreads per core) and 512 GB of memory. Each method was compiled with the Intel icc 17.1 compiler, and 32 bit integers are used for vertex indices and edge indices (unsigned for the edges) for all methods. For each method, we timed everything after the adjacency matrix, incidence matrix, and/or graph was read into the methods native data structure. For our KokkosKernels implementation

TABLE I
GRAPHS SORTED BY THE NUMBER OF EDGES. COLUMNS LIST THE NUMBER OF VERTICES, EDGES, AND TRIANGLES. HIGHLIGHTED GRAPHS ARE GRAPHS NOT PROVIDED BY GRAPH CHALLENGE.

Graph	$\|V\|$	$\|E\|$	$\|T\|$
cit-HepTh	27,770	352,285	1,478,735
email-EuAll	265,214	364,481	267,313
soc-Epinions1	75,879	405,740	1,624,481
cit-HepPh	34,546	420,877	1,276,868
soc-Slashdot0811	77,360	469,180	551,724
soc-Slashdot0902	82,168	504,230	602,592
flickrEdges	105,938	2,316,948	107,987,357
amazon0312	400,727	2,349,869	3,686,467
amazon0505	410,236	2,439,437	3,951,063
amazon0601	403,394	2,443,408	3,986,507
graph500-scale18	174,147	3,800,348	82,287,285
graph500-scale19	335,318	7,729,675	186,288,972
graph500-scale20	645,820	15,680,861	419,349,784
cit-Patents	3,774,768	16,518,947	7,515,023
graph500-scale21	1,243,072	31,731,650	935,100,883
soc-LiveJournal1	4,847,571	42,851,237	285,730,264
wb-edu	9,845,725	46,236,105	254,718,147
graph500-scale22	2,393,285	64,097,004	2,067,392,370
graph500-scale23	4,606,314	129,250,705	4,549,133,002
graph500-scale24	8,860,450	260,261,843	9,936,161,560
graph500-scale25	17,043,780	523,467,448	21,575,375,802
uk-2005	39,459,925	783,027,125	21,779,366,056
it-2005	41,291,594	1,027,474,947	48,374,551,054
twitter-2010	41,652,230	1,202,513,046	34,824,916,864
Friendster	65,608,366	1,806,067,135	4,173,724,142

TABLE II
BEST AVERAGE TRIANGLE COUNTING TIMES (IN SECONDS) UP TO 32 THREADS ON INTEL HASWELL. * INDICATES APPROXIMATE RUNTIMES. RED HIGHLIGHT = BEST TIME FOR A GRAPH.

	Reference times(s)			TCKK	
Graphs	C++	Python	TCM	Time (s)	Rate
cit-HepPh	2.03E+1	4.85E+0	1.04E-2	4.41E-3	7.99E+7
cit-HepTh	3.14E+1	7.41E+0	9.79E-3	5.03E-3	7.25E+7
email-EuAll	1.05E+2	2.46E+1	6.64E-3	5.80E-3	7.00E+7
soc-Epinions1	6.56E+1	1.56E+1	1.71E-2	3.90E-3	1.08E+8
soc-Slashdot0811	4.80E+1	1.41E+1	1.86E-2	6.11E-3	7.68E+7
soc-Slashdot0902	5.26E+1	1.51E+1	1.89E-2	6.30E-3	8.01E+7
amazon0312	3.43E+1	9.89E+0	2.27E-2	7.54E-2	3.07E+7
amazon0505	3.72E+1	1.22E+1	2.20E-2	1.77E-2	1.33E+8
amazon0601	3.65E+1	1.19E+1	2.08E-2	1.84E-2	1.32E+8
flickrEdges	1.06E+3	1.82E+2	1.87E-1	1.86E-2	1.32E+8
graph500-scale18	7.94E+3	1.05E+3	2.67E-1	1.33E-1	2.85E+7
graph500-scale19	2.38E+4	MEM	5.98E-1	2.73E-1	2.84E+7
cit-Patents	1.80E+2	5.68E+1	1.42E-1	4.97E-1	3.15E+7
graph500-scale20	7.00E+4	MEM	1.01E+0	1.07E-1	1.54E+8
graph500-scale21	2.22E+5	MEM	2.38E+0	1.07E+0	2.96E+7
soc-LiveJournal1	9.46E+3	MEM	1.06E+0	7.33E-1	5.85E+7
wb-edu	9.24E+3	MEM	5.57E-1	2.32E-1	1.99E+8
graph500-scale22	8.84E+5	MEM	5.57E+0	3.07E+0	2.09E+7
graph500-scale23	4.57E+6*	MEM	1.39E+1	8.84E+0	1.46E+7
graph500-scale24	8.39E+6*	MEM	3.39E+1	2.43E+1	1.07E+7
graph500-scale25	2.08E+7*	MEM	8.31E+1	6.89E+1	7.60E+6
uk-2005	5.48E+5*	MEM	6.85E+0	2.03E+0	3.86E+8
it-2004	2.14E+6*	MEM	1.76E+1	4.59E+0	2.24E+8
twitter-2010	1.30E+6*	MEM	9.78E+1	5.10E+1	2.36E+7
Friendster	1.12E+6*	MEM	6.69E+1	7.73E+1	2.34E+7

(TCKK), this included the times to order the rows, to construct matrix L, to compress of matrix L, and to compute the masked sparse matrix-matrix multiplication operation. For the parallel methods TCM and KokkosKernels, we ran each graph for 1, 2, 4, 8, 16, and 32 threads. We also ran each graph on 64 threads (2 threads per core).

Table II shows the average runtimes for the C++ serial reference implementation (estimated runtimes are designated with * for prohibitively slow problems), the Python serial reference implementation (MEM denotes graphs requiring too much memory to complete), the TCM method, and our KokkosKernels implementation (TCKK). For TCM and TCKK, we report the best time across the different numbers of threads up to 32 threads. The cells in the table that are highlighted in red represent the best execution times (or execution times that are within 1% of the best time). From this table, we see that TCKK is much faster than the C++ and Python reference implementations and is as fast or faster than TCM for 24 of 25 graphs, the only outlier being Friendster. We also report the Graph Challenge *rate*, which is defined to be the number of edges in the graph divided by the triangle counting time, of TCKK for each graph. We see a best rate of 386 million edges per second for all graphs (uk-2005) and 154 million edges per second for the Graph Challenge graphs (graph500-scale20).

Additional speedups were achieved by running on 64 threads (2 threads per core) as shown in Table III. For the parallel methods, we report the best time up to 64 threads. Again, we see that TCKK is much faster than the C++ and Python reference implementations. Across the set of graphs, TCM scales better than TCKK from 32 to 64 threads. However, TCKK is still as fast or faster than TCM for 22 of the 25 graphs. The graphs where TCM is as good as TCKK are three of the R-MAT graphs (scale 23-25) and Friendster. Due to the randomness/irregularity of the edge connectivity of these graphs (at least in the current ordering), the compression used by TCKK is not effective, compressing the graph by at most 3%. For TCKK, we see a best rate of 637 million edges per second for all graphs (uk-2005) and 198 million edges per second for the Graph Challenge graphs (amazon0312).

Figure 1 shows the speedups of our KokkosKernels triangle counting implementations relative to the serial reference implementations on Intel Haswell. These speedups are significant, with a best speedup of almost $670,000$ when compared to the C++ reference and a best speedup of over $11,000$ when compared to the Python reference (for those problems that could run). Figure 2 shows the speedups of our KokkosKernels triangle counting implementations relative to TCM. For the synthetic R-MAT graphs, we see a speedup of up to 1.93 times. However, for the largest R-MAT graph, TCKK is 23% slower than TCM. For the other graphs, we see a speedup for all but one of the graphs (Friendster), with a best speedup of 4.5. TCKK is 37% slower than TCM for Friendster.

D. Scalability

Figure 3 shows the strong scaling speedup of TCKK on the graph500-scale23 graph. TCKK scales well to the point that the number of threads equals the number of cores, but does not benefit much (from additional hyperthreads) after that. TCM, however, is able to scale well with two hyperthreads per core. This could be partially attributed to the different runtimes and programming model used (Kokkos/OpenMP vs Cilk).

TABLE III

BEST AVERAGE TRIANGLE COUNTING TIMES (IN SECONDS) UP TO 64 THREADS ON INTEL HASWELL. * INDICATES APPROXIMATE RUNTIMES. RED HIGHLIGHT = BEST TIME (OR WITHIN 1% OF BEST TIME) FOR A GRAPH, PINK HIGHLIGHT = WITHIN 5% OF BEST TIME.

	Reference times(s)			TCKK	
Graphs	**C++**	**Python**	**TCM**	**Time (s)**	**Rate**
cit-HepTh	3.14E+1	7.41E+0	9.79E-3	3.78E-3	9.33E+7
email-EuAll	1.05E+2	2.46E+1	6.64E-3	4.80E-3	7.59E+7
soc-Epinions1	6.56E+1	1.56E+1	1.71E-2	4.82E-3	8.42E+7
cit-HepPh	2.03E+1	4.85E+0	9.56E-3	2.97E-3	1.42E+8
soc-Slashdot0811	4.80E+1	1.41E+1	1.56E-2	4.89E-3	9.60E+7
soc-Slashdot0902	5.26E+1	1.51E+1	1.71E-2	5.14E-3	9.82E+7
flickrEdges	1.06E+3	1.82E+2	1.15E-1	5.04E-2	4.59E+7
amazon0312	3.43E+1	9.89E+0	1.28E-2	1.19E-2	1.98E+8
amazon0505	3.72E+1	1.22E+1	1.37E-2	1.31E-2	1.87E+8
amazon0601	3.65E+1	1.19E+1	1.30E-2	1.25E-2	1.96E+8
graph500-scale18	7.94E+3	1.05E+3	1.82E-1	9.44E-2	4.03E+7
graph500-scale19	2.38E+4	MEM	3.17E-1	2.24E-1	3.46E+7
graph500-scale20	7.00E+4	MEM	6.23E-1	4.15E-1	3.78E+7
cit-Patents	1.80E+2	5.68E+1	9.65E-2	8.96E-2	1.84E+8
graph500-scale21	2.22E+5	MEM	1.29E+0	8.81E-1	3.60E+7
soc-LiveJournal1	9.46E+3	MEM	5.73E-1	3.63E-1	1.18E+8
wb-edu	9.24E+3	MEM	2.52E-1	1.61E-1	2.88E+8
graph500-scale22	8.84E+5	MEM	3.12E+0	2.42E+0	2.65E+7
graph500-scale23	4.57E+6*	MEM	7.67E+0	7.66E+0	1.69E+7
graph500-scale24	8.39E+6*	MEM	1.90E+1	2.14E+1	1.21E+7
graph500-scale25	2.08E+7*	MEM	4.68E+1	6.05E+1	8.66E+6
uk-2005	5.48E+5*	MEM	5.52E+0	1.23E+0	6.37E+8
it-2004	2.14E+6*	MEM	1.11E+1	3.20E+0	3.21E+8
twitter-2010	1.30E+6*	MEM	5.67E+1	4.32E+1	2.78E+7
Friendster	1.12E+6*	MEM	3.55E+1	5.64E+1	3.20E+7

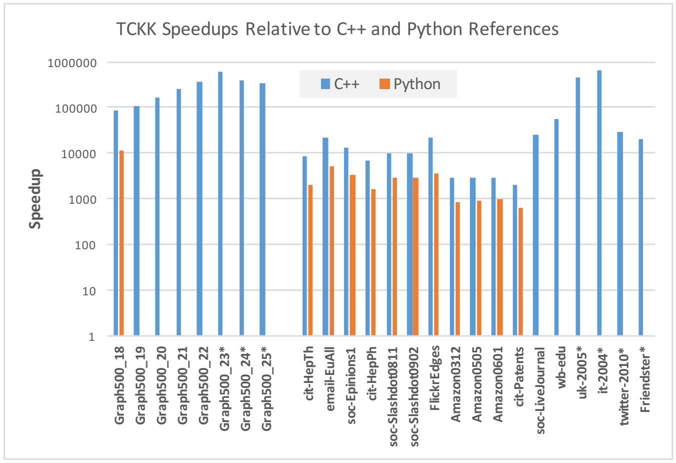

Fig. 1. Speedups relative to serial reference implementations (64 threads). * indicates approximate runtimes.

Figure 2 also shows the scaling of TCKK as the graph sizes grow (left to right) for the synthetic R-Mat graphs (Graph500* – left grouping) and the non synthetic graphs (right grouping). While TCKK achieves significant speedup when compared to TCM on several problems, it also demonstrates an area of improvement for TCKK. For graphs where compression does not help much (e.g., R-Mat graphs and Friendster), TCM is able to catchup to TCKK performance and even do better on two large instances. The contributing factor here relates to a data structure choice. The KokkosKernels SPGEMM symbolic phase allocates a thread local hash table based on an estimate (see [28]). Typically compression helps KokkosKernels to

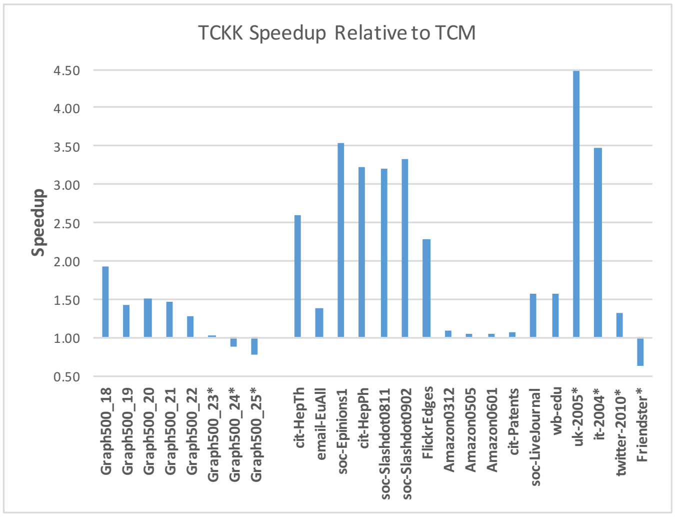

Fig. 2. Speedups relative to TCM reference implementation (64 threads).

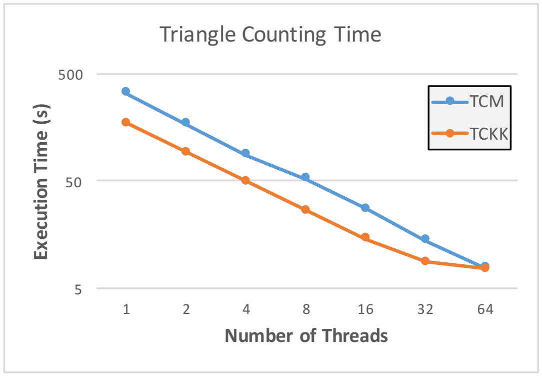

Fig. 3. Strong scaling of TCM and TCKK on graph500-scale23.

arrive at better estimates. However, compression is not very effective on the graph500 graphs resulting in large thread local memory allocation affecting performance as graph sizes increase. This could be alleviated by non-uniform thread local memory allocation. We leave this for future work.

E. Other Architectures

To demonstrate the performance of the KKMEM on a broader spectrum of high-performance, highly-threaded compute nodes, we have additionally benchmarked the code on nodes using Intel's Knights Landing (KNL) self-hosted many-core 7250 Xeon Phi processor (which has 68 4-way-SMT cores running at 1.4GHz) and IBM's OpenPOWER POWER8 variant which offers dual-socket nodes of 8-core sockets. For POWER8, each core provides 8-way SMT threading. The experiments using the KNL in this paper are run in quadrant-cache mode which uses the high-bandwidth MC-DRAM of the processor as a cache for the slower, DDR4 system-capacity memory. In each case, we configure our benchmarking runs to utilize OpenMP's close-policy and select either cores or threads as appropriate. The best runtimes are presented below due to space considerations and are selected by performing

TABLE IV
BEST AVERAGE TCKK TRIANGLE COUNTING TIMES (IN SECONDS) AND RATES (EDGES PER SECOND) FOR DIFFERENT ARCHITECTURES. RED HIGHLIGHTED CELLS = BEST TIME FOR A GRAPH. BLUE HIGHLIGHTED CELLS DESIGNATE HIGHEST RATE FOR ALL GRAPHS AND 20 GRAPH CHALLENGE GRAPHS.

	Intel Haswell		**Intel KNL**		**IBM Power 8**	
Graphs	**Time (s)**	**Rate**	**Time(s)**	**Rate**	**Time(s)**	**Rate**
cit-HepTh	3.78E-3	9.33E+7	5.98E-3	5.90E+7	3.22E-3	1.09E+8
email-EuAll	4.80E-3	7.59E+7	6.00E-3	6.07E+7	3.71E-3	9.83E+7
soc-Epinions1	4.82E-3	8.42E+7	9.61E-3	4.22E+7	4.82E-3	8.41E+7
cit-HepPh	2.97E-3	1.42E+8	3.99E-3	1.05E+8	2.13E-3	1.98E+8
soc-Slashdot0811	4.89E-3	9.60E+7	8.39E-3	5.59E+7	4.18E-3	1.12E+8
soc-Slashdot0902	5.14E-3	9.82E+7	8.80E-3	5.73E+7	4.38E-3	1.15E+8
flickrEdges	5.04E-2	4.59E+7	1.09E-1	2.12E+7	4.03E-2	5.75E+7
amazon0312	1.19E-2	1.98E+8	2.23E-2	1.05E+8	1.01E-2	2.30E+8
amazon0505	1.31E-2	1.87E+8	2.18E-2	1.12E+8	9.71E-3	2.51E+8
amazon0601	1.25E-2	1.96E+8	2.10E-2	1.17E+8	9.67E-3	2.53E+8
graph500-scale18	9.44E-2	4.03E+7	2.25E-1	1.69E+7	7.70E-2	4.94E+7
graph500-scale19	2.24E-1	3.46E+7	4.05E-1	1.91E+7	2.12E-1	3.64E+7
graph500-scale20	4.15E-1	3.78E+7	7.77E-1	2.02E+7	5.09E-1	3.08E+7
cit-Patents	8.96E-2	1.84E+8	8.20E-2	2.01E+8	7.18E-2	2.30E+8
graph500-scale21	8.81E-1	3.60E+7	1.91E+0	1.65E+7	1.37E+0	2.32E+7
soc-LiveJournal1	3.63E-1	1.18E+8	5.61E-1	7.64E+7	3.39E-1	1.26E+8
wb-edu	1.61E-1	2.88E+8	3.59E-1	1.29E+8	1.32E-1	3.50E+8
graph500-scale22	2.42E+0	2.65E+7	5.15E+0	1.25E+7	3.72E+0	1.72E+7
graph500-scale23	7.66E+0	1.69E+7	1.18E+1	1.10E+7	9.69E+0	1.33E+7
graph500-scale24	2.14E+1	1.21E+7	3.09E+1	8.43E+6	2.58E+1	1.01E+7
graph500-scale25	6.05E+1	8.66E+6	7.43E+1	7.05E+6	5.71E+1	9.16E+6
uk-2005	1.23E+0	6.37E+8	2.85E+0	2.75E+8	1.65E+0	4.75E+8
it-2004	3.20E+0	3.21E+8	1.03E+01	9.98E+7	5.92E+0	1.74E+8

runs of either 1 thread per core (*i.e.*, ignoring SMT capabilities) or all the threads per core (*i.e.*, using all the available SMT capabilities). Typically, smaller graphs require SMT-1-based executions to reduce the overhead associated with the synchronization of OpenMP parallel-constructs, whereas, larger runs benefit from using all the available threading to increase execution rates.

The benchmarked results are shown in Table IV (largest graphs are not presented since they did not fit into memory on all systems). We note that the POWER8 is a strong performer, exceeding the performance rate of the Haswell in a number of graphs at both smaller and large scales. Use of highly randomized graphs (the Graph-500 collection) on the Knights Landing, tends to show a significant reduction in performance which correlates with our anecdotal experience that the weaker cores provided on the Xeon Phi do not tolerate randomized memory accesses as well as faster-clocked, functional-unit rich multi-core systems.

IV. SUMMARY/CONCLUSIONS

Linear algebra-based approaches to graph algorithms show great promise for impacting high performance data analytics applications. In this paper, we presented such a linear algebra-based approach to triangle counting as part of the IEEE HPEC Graph Challenge. We introduced a new triangle counting algorithm and leveraged Kokkos and KokkosKernels to implement this algorithm in a very efficient manner. With minor modifications to the KokkosKernels sparse matrix-matrix multiplication algorithm KKMEM, we were able to obtain a performance-portable triangle counting implementation TCKK that runs efficiently on many multicore architectures.

TCKK compares favorably to the serial Graph Challenge reference implementations, executing $670,000$ times faster than the C++ reference and up to $11,000$ times faster the Python reference. We also compared TCKK with the parallel merge-based triangle counting method (TCM) proposed in [29], which has some of the best attributed results in the open literature. TCKK compared favorably to TCM, performing as well or better for 24 out of 25 graphs when executing on 32 threads. One of TCKK's main advantages over TCM was the use of compression, which was beneficial on some of the graphs but was not particularly helpful on the random synthetic R-MAT graphs. Moving to 64 threads, TCM is able to scale well with two hyperthreads per core while TCKK does not benefit as much from the additional hyperthreads. However, TCKK still performs as well or better than TCM for 22 out of 25 of graphs when executing on 64 threads. The results of this linear algebra-based method are compelling enough that we believe that this (and other linear-algebra based kernels) may soon impact the design of kernels within generic graph libraries such as the MTGL [26].

This effort has also indicated several promising directions for future work. The "reordering" step greatly impacts the performance of the triangle counting. We presented a simple reordering scheme (decreasing vertex degree), but have observed that alternative reordering methods can lead to better performance for some graphs (e.g., reducing the runtime for Friendster by 40%), due to operation reduction or better load-balancing. In the future, we plan to investigate the various sorting options to obtain a method or heuristic that will work for most graphs. We also plan to improve the scalability of our implementation for high thread counts by moving from a large uniform thread local memory allocation scheme to a non-uniform thread local memory allocation. This will potentially help performance for the large, irregular problems (e.g., large R-MAT graphs) when executed on large numbers of threads. In the future, we also plan to explore different architectures, including GPUs. Since KKMEM has shown good performance on GPUs [28] and our implementation is built on performance-portable Kokkos, we expect fairly good performance with only minor changes to our implementation. Finally, we plan on extending this work to multiple nodes, leveraging decades of experience related to distributed-memory sparse linear algebra computations. Building on this experience and our on-node TCKK implementation, our goal will be to find all triangles in extreme-scale graphs (e.g., trillions of edges) in seconds.

ACKNOWLEDGMENT

The research presented in this paper was funded through the Laboratory Directed Research and Development (LDRD) program at Sandia National Laboratories, in the context of the Multi-Level Memory Algorithmics for Large, Sparse Problems Project. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

References

[1] M. M. Wolf, J. W. Berry, and D. T. Stark, "A task-based linear algebra building blocks approach for scalable graph analytics," in *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*. IEEE, 2015, pp. 1–6.

[2] A. Pavan, K. Tangwongsan, S. Tirthapura, and K.-L. Wu, "Counting and sampling triangles from a graph stream," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1870–1881, 2013.

[3] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient semi-streaming algorithms for local triangle counting in massive graphs," in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2008, pp. 16–24.

[4] C. E. Tsourakakis, P. Drineas, E. Michelakis, I. Koutis, and C. Faloutsos, "Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation," *Social Network Analysis and Mining*, vol. 1, no. 2, pp. 75–81, 2011.

[5] N. Wang, J. Zhang, K.-L. Tan, and A. K. Tung, "On triangulation-based dense neighborhood graph discovery," *Proceedings of the VLDB Endowment*, vol. 4, no. 2, pp. 58–68, 2010.

[6] V. Gadepally, J. Bolewski, D. Hook, D. Hutchison, B. Miller, and J. Kepner, "Graphulo: Linear algebra graph kernels for nosql databases," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, May 2015, pp. 822–830.

[7] 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," 2017, unpublished.

[8] J. W. Berry, L. A. Fostvedt, D. J. Nordman, C. A. Phillips, C. Seshadhri, and A. G. Wilson, "Why do simple algorithms for triangle enumeration work in the real world?" *Internet Mathematics*, vol. 11, no. 6, pp. 555–571, 2015.

[9] J. Cohen, "Graph twiddling in a mapreduce world," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29–41, 2009.

[10] C. E. Tsourakakis, "Fast counting of triangles in large real networks without counting: Algorithms and laws," in *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*. IEEE, 2008, pp. 608–617.

[11] C. Seshadhri, A. Pinar, and T. G. Kolda, "Triadic measures on graphs: The power of wedge sampling," in *Proceedings of the 2013 SIAM International Conference on Data Mining*. SIAM, 2013, pp. 10–18.

[12] H.-M. Park, F. Silvestri, U. Kang, and R. Pagh, "Mapreduce triangle enumeration with guarantees," in *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*. ACM, 2014, pp. 1739–1748.

[13] 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.

[14] KokkosKernels. [Online]. Available: https://github/kokkos/kokkoskernels

[15] Tim Mattson et al., "Standards for graph algorithm primitives," in *Proc. IEEE High Performance Extreme Comp. Conf.*, 2013.

[16] A. Buluc, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "Design of the graphblas api for c," in *Parallel and Distributed Processing Symposium Workshops, 2017 IEEE International*. IEEE, 2017.

[17] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011, vol. 22.

[18] A. Azad, A. Buluç, and J. R. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *Proceedings of the IPDPSW, Workshop on Graph Algorithm Building Blocks (GABB)*, 2015. [Online]. Available: http://gauss.cs.ucsb.edu/ aydin/triangles-gabb.pdf

[19] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496 – 509, 2011. [Online]. Available: http://gauss.cs.ucsb.edu/ aydin/combblas-r2.pdf

[20] V. Gadepally, J. Bolewski, D. Hook, D. Hutchison, B. Miller, and J. Kepner, "Graphulo: Linear algebra graph kernels for nosql databases," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*. IEEE, 2015, pp. 822–830.

[21] J. Kepner, W. Arcand, W. Bergeron, N. T. Bliss, R. Bond, C. Byun, G. Condon, K. Gregson, M. Hubbell, J. Kurz, A. McCabe, P. Michaleas, A. Prout, A. Reuther, A. Rosa, and C. Yee, "Dynamic distributed dimensional data model (d4m) database and computation system." in *ICASSP*. IEEE, 2012, pp. 5349–5352.

[22] K. Ekanadham, B. Horn, J. Jann, M. Kumar, J. Moreira, P. Pattnaik, M. Serrano, G. Tanase, and H. Yu, "Graph programming interface: Rationale and specification," IBM Research Report, RC25508 (WAT1411-052) November 19, Tech. Rep., 2014.

[23] M. J. Anderson, N. Sundaram, N. Satish, M. M. A. Patwary, T. L. Willke, and P. Dubey, "Graphpad: Optimized graph primitives for parallel and distributed platforms," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 313–322.

[24] P. Zhang, M. Zalewski, A. Lumsdaine, S. Misurda, and S. McMillan, "Gbtl-cuda: Graph algorithms and primitives for gpus," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 912–920.

[25] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[26] Jonathan W. Berry et al., "Software and algorithms for graph queries on multithreaded architectures," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, 2007, pp. 1–14.

[27] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J Parallel Distrib Comp*, vol. 74, no. 12, pp. 3202–3216, 2014.

[28] M. Deveci, C. Trott, and S. Rajamanickam, "Performance-portable sparse matrix-matrix multiplication for many-core architectures," in *Parallel and Distributed Processing Symposium Workshops, 2017 IEEE International*. IEEE, 2017.

[29] 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.

[30] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, 2011.