Graph algorithms via SuiteSparse:GraphBLAS: triangle counting and K-truss

Timothy A. Davis
Dept. of Computer Science and Engineering
Texas A&M University
College Station, TX, USA
davis@tamu.edu

***Abstract*—SuiteSparse:GraphBLAS is a full implementation of the GraphBLAS standard, which defines a set of sparse matrix operations on an extended algebra of semirings using an almost unlimited variety of operators and types. When applied to sparse adjacency matrices, these algebraic operations are equivalent to computations on graphs. GraphBLAS provides a powerful and expressive framework for creating graph algorithms based on the elegant mathematics of sparse matrix operations on a semiring. To illustrate GraphBLAS, two graph algorithms are constructed in GraphBLAS and compared with efficient implementations without GraphBLAS: triangle counting and constructing the k-truss of a graph.**

***Index Terms*—graph algorithms, sparse matrix computations**

I. Introduction

The GraphBLAS standard [1] defines sparse matrix and vector operations on an extended algebra of semirings. The operations are useful for creating a wide range of graph algorithms. Kepner and Gilbert [2] provide a framework for understanding how graph algorithms can be expressed as matrix computations.

For example, consider the matrix-matrix multiplication, $\mathbf{C} = \mathbf{AB}$. Suppose $\mathbf{A}$ and $\mathbf{B}$ are sparse n-by-n Boolean adjacency matrices of two undirected graphs. If the matrix multiplication is redefined to use logical AND instead of scalar multiply, and if it uses the logical OR instead of add, then the matrix $\mathbf{C}$ is the sparse Boolean adjacency matrix of a graph that has an edge (i, j) if node i in $\mathbf{A}$ and node j in $\mathbf{B}$ share any neighbor in common. The OR-AND pair forms an algebraic semiring, and many graph operations like this can be succinctly represented by matrix operations with different semirings and different numerical types. GraphBLAS provides a wide range of built-in types and operators, and allows the user application to create new types and operators without needing to recompile the GraphBLAS library. Expressing graph algorithms in the language of linear algebra provides:

- a powerful way of expressing graph algorithms with large, bulk operations on adjacency matrices,
- composable graph operations, e.g. $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$,
- simpler graph algorithms in user-code,
- simple objects for complex problems – a sparse matrix with any data type, including user-defined types,
- a well-defined graph object, closed under operations,
- and high performance: serial, parallel, or GPU, allowing the library to optimize bulk graph/matrix operations.

With support from NSF CNS-1514406.

A. Graphs and sparse matrices

Many applications give rise to large graphs, with many nodes and edges. However, typical graphs are very sparse, with n nodes but only $O(n)$ edges.

Any graph $G = (V, E)$ can be considered as a sparse adjacency matrix $\mathbf{A}$. The square case of an n-by-n sparse matrix is useful for representing a directed or undirected graph with $n = |V|$ nodes, where either the matrix entry a_{ij} or a_{ji} represents the edge (i, j). In the rectangular case, an m-by-n sparse matrix can represent a bipartite graph, or a hypergraph, depending on the context. Edges that do not appear in G are not represented in the data structure of the sparse matrix $\mathbf{A}$. A sparse matrix data structure allows huge graphs to be represented, requiring only $O(n+|E|)$ or $O(|E|)$ space, depending on the data structure used.

Values of entries not stored in the sparse data structure have some implicit value. In conventional linear algebra, this implicit value is zero, but it differs with different semirings. Explicit values are called *entries* and they appear in the data structure. The *pattern* of a matrix defines where its explicit entries appear, and can be represented as either a set of indices (i, j), or as a Boolean matrix $\mathbf{S}$ where $s_{ij} = 1$ if a_{ij} is an explicit entry in the sparse matrix $\mathbf{A}$.

The entries in the pattern of $\mathbf{A}$ can take on any value, including the implicit value, whatever it happens to be. It need not be the value zero. For example, in the max-plus tropical algebra, the implicit value is negative infinity, and zero has a different meaning.

II. Overview of GraphBLAS objects, methods, and operations

SuiteSparse:GraphBLAS provides a collection of *methods* to create, query, and free each of its nine different types of objects. Once these objects are created they can be used in mathematical *operations* (not to be confused with the how the term *operator* is used in GraphBLAS). The nine types are described below.

(1) Types: A GraphBLAS type (`GrB_Type`) can be any of 11 built-in types (Boolean, integer and unsigned integers of sizes 8, 16, 32, and 64 bits, and single and double precision

floating point). In addition, user-defined scalar types can be created from nearly any C `typedef`, as long as the entire type fits in a fixed-size contiguous block of memory (of arbitrary size). All of these types can be used to create GraphBLAS sparse matrices or vectors. All built-in types can typecasted as needed; user-defined types cannot.

(2) Unary operators: A unary operator (`GrB_UnaryOp`) is a function $z = f(x)$. SuiteSparse: GraphBLAS comes with 67 built-in unary operators, such as $z = 1/x$ and $z = -x$, with variants for each built-in type. The user application can also create its own user-defined unary operators.

(3) Binary operators: Likewise, a binary operator (`GrB_BinaryOp`) is a function $z = f(x, y)$, such as $z = x + y$ or $z = xy$. SuiteSparse:GraphBLAS provides 256 built-in binary operators, with variants for each built-in type. User-defined binary operators can also be created.

(4) Select operators: The `GxB_SelectOp` operator is a SuiteSparse extension to the GraphBLAS API. It is used in the `GxB_select` operation to select a subset of entries from a matrix, like `L=tril(A)` in MATLAB.

(5) Monoids: The scalar addition of conventional matrix multiplication is replaced with a *monoid*. A monoid (`GrB_Monoid`) is an associative and commutative binary operator $z = f(x, y)$ where all three domains are the same (the types of x, y, and z) and where the operator has an identity value o such that $f(x, o) = f(o, x) = x$. Performing matrix multiplication with a semiring uses a monoid in place of the "add" operator, scalar addition being just one of many possible monoids. The identity value of addition is zero, since $x + 0 = 0 + x = x$. User-created monoids can be defined with any associative and commutative operator with an identity value. A monoid can also be used in a reduction operation, like `s=sum(A)` in MATLAB.

(6) Semirings: A *semiring* (`GrB_Semiring`) consists of a monoid and a "multiply" operator. Together, these operations define the matrix "multiplication" $\mathbf{C} = \mathbf{AB}$, where the monoid is used as the additive operator and the semiring's "multiply" operator is used in place of the conventional scalar multiplication in standard matrix multiplication via the plus-times semiring. A user application can define its own monoids and semirings.

(7) Descriptors: A *descriptor* object, `GrB_Descriptor`, provides parameter settings that modify the behavior of GraphBLAS operations, such as transposing an input matrix or complementing a mask matrix.

(8) Vectors: A sparse vector, `GrB_Vector`.

(9) Matrices: A sparse matrix, `GrB_Matrix`.

A. Non-blocking mode

GraphBLAS includes a *non-blocking* mode where operations can be left pending, and saved for later. This is very useful for submatrix assignment (like `C(I,J)=A` in MATLAB where `I` and `J` are integer vectors), or or scalar assignment (`C(i,j)=x` where `i` and `j` are scalar integers). Because of how MATLAB stores its matrices, adding and deleting individual entries is very costly. By contrast, SuiteSparse:GraphBLAS exploits the non-blocking mode to allow for fast incremental update of a matrix.

TABLE I
SUITESPARSE:GRAPHBLAS OPERATIONS

function name	description	GraphBLAS notation
`GrB_mxm`	matrix-matrix mult.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{AB}$
`GrB_vxm`	vector-matrix mult.	$\mathbf{w}'\langle\mathbf{m}'\rangle = \mathbf{w}' \odot \mathbf{u}'\mathbf{A}$
`GrB_mxv`	matrix-vector mult.	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{Au}$
`GrB_eWiseMult`	element-wise,	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \otimes \mathbf{B})$
	set-union	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$
`GrB_eWiseAdd`	element-wise,	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \oplus \mathbf{B})$
	set-intersection	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \oplus \mathbf{v})$
`GrB_extract`	extract submatrix	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}(\mathbf{i}, \mathbf{j})$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{u}(\mathbf{i})$
`GrB_assign`	assign submatrix	$\mathbf{C}\langle\mathbf{M}\rangle(\mathbf{i}, \mathbf{j}) = \mathbf{C}(\mathbf{i}, \mathbf{j}) \odot \mathbf{A}$
		$\mathbf{w}\langle\mathbf{m}\rangle(\mathbf{i}) = \mathbf{w}(\mathbf{i}) \odot \mathbf{u}$
`GxB_subassign`	assign submatrix	$\mathbf{C}(\mathbf{i}, \mathbf{j})\langle\mathbf{M}\rangle = \mathbf{C}(\mathbf{i}, \mathbf{j}) \odot \mathbf{A}$
		$\mathbf{w}(\mathbf{i})\langle\mathbf{m}\rangle = \mathbf{w}(\mathbf{i}) \odot \mathbf{u}$
`GrB_apply`	apply unary op.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A})$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u})$
`GxB_select`	apply select op.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A}, \mathbf{k})$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u}, \mathbf{k})$
`GrB_reduce`	reduce to vector	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot [\oplus_j \mathbf{A}(:, j)]$
	reduce to scalar	$s = s \odot [\oplus_{ij} \mathbf{A}(i, j)]$
`GrB_transpose`	transpose	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}'$

B. The accumulator and the mask

An optional accumulator operator ($\odot$) and mask matrix ($\mathbf{M}$) can be specified, written as $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$ where $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$ denotes the application of the accumulator operator, and $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$ denotes the mask operator via the Boolean matrix $\mathbf{M}$. The expression $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$ is computed as follows:

if no accumulator, $\mathbf{Z} = \mathbf{T}$; otherwise $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$
if requested via descriptor, all entries cleared from $\mathbf{C}$
if no mask $\mathbf{M}$
    $\mathbf{C} = \mathbf{Z}$ if the mask is not complemented;
    otherwise $\mathbf{C}$ is not modified
else
    $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$ if the mask is not complemented
    (where c_{ij} is modified if and only if m_{ij} is nonzero);
    otherwise $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{Z}$

The accumulator operator acts like a sparse matrix addition, except that any operator can be used. The pattern of $\mathbf{C} \odot \mathbf{T}$ is the set-union of the patterns of $\mathbf{C}$ and $\mathbf{T}$, and the operator is applied only on the set-intersection of $\mathbf{C}$ and $\mathbf{T}$.

C. GraphBLAS methods and operations

The matrix (`GrB_Matrix`) and vector (`GrB_Vector`) objects include additional methods for setting a single entry, extracting a single entry, making a copy, and constructing an entire matrix or vector from a list of *tuples*. The tuples are held as three arrays `I`, `J`, and `X`, which work the same as `A=sparse(I,J,X)` in MATLAB, except that any type matrix or vector can be constructed.

Table I lists all GraphBLAS operations in the GraphBLAS notation where $\mathbf{AB}$ denotes the multiplication of two matrices over a semiring. Upper case letters denote a matrix, and lower

case letters are vectors. Each operation takes an optional `GrB_Descriptor` argument that modifies the operation. The notation $\mathbf{A} \oplus \mathbf{B}$ denotes the element-wise operator that produces a set-union pattern (like `A+B` in MATLAB). The notation $\mathbf{A} \otimes \mathbf{B}$ denotes the element-wise operator that produces a set-intersection (like `A.*B` in MATLAB). Reduction of a matrix $\mathbf{A}$ to a vector reduces the ith row of $\mathbf{A}$ to a scalar w_i, like `w=sum(A')` in MATLAB.

III. Triangle counting in GraphBLAS

A *triangle* in a graph is a clique of size 3. There are many matrix formulations of the method, but the simplest and fastest one is by Wolf et al. [3], a variant of Cohen's method [4]. See also [5]. If `A` is a symmetric adjacency matrix of a graph, then the MATLAB `tricount.m` function below computes the number of triangles in the graph.

```
function ntri = tricount (A)
L = tril (A) ;
ntri = sum (sum ((L*L).*L)) ;
```

Consider the edge (i, j), so that `A(i,j)=1`, and suppose $i > j$. Then `C(i,j)` is `L(i,:)*L(:,j)`. Since `L` is strictly lower triangular, `L(i,:)` is nonzero only in `L(i,1:i-1)`, and `L(:,j)` is nonzero only in `L(j+1:n,j)`. Combining these two sets gives the nonzero intersection, `C(i,j) = L(i,j+1:i-1)*L(j+1:i-1,j)`. Suppose there is a node `k` in this range `j+1:i-1`, inclusive, that has an edge to both `i` and `j`. That is, `L(i,k)=L(k,j)=1`. This is a triangle (i, j, k), where $j < k < i$. Since these three indices are strictly ordered, the triangle is counted only once, and `C(i,j)` is the number of such triangles for all `k`. If `L(i,j)=0` then the result is not needed since (i, j, k) cannot be a triangle, and thus the final `().*L` step.

GraphBLAS can compute this with a masked matrix multiply, $\mathbf{C}\langle\mathbf{L}\rangle = \mathbf{L}^2$, as shown below. This is much faster and uses much less memory since `L*L` in MATLAB contains many more nonzeros than `L`. In GraphBLAS, only $O(|L|)$ space is needed. However, the C code using GraphBLAS is almost as simple as the MATLAB `tricount.m` function above.

```
int64_t tricount (const GrB_Matrix L)   // L=tril(A), input graph
{
    int64_t ntri ;
    GrB_Index n ;
    GrB_Matrix C ;
    // n = # of rows of L = # nodes in the graph A
    GrB_Matrix_nrows (&n, L) ;
    // C<L> = L*L
    GrB_Matrix_new (&C, GrB_UINT32, n, n) ;
    GrB_mxm (C, L, NULL, GxB_PLUS_TIMES_UINT32, L, L, NULL) ;
    // ntri = sum (C), reduce to a scalar
    GrB_reduce (&ntri, NULL, GxB_PLUS_INT64_MONOID, C, NULL) ;
    return (ntri) ;
}
```

Graph algorithms such as triangle counting are simple and expressive in GraphBLAS. However, the same author writing the algorithm in pure C (without GraphBLAS) may be able to obtain higher performance, because the resulting methods need not conform to the GraphBLAS API. This section compares the performance of the GraphBLAS implementation, above, with four methods:

1) `tri_simple`: a simple sequential method,
2) `tri_mark`: similar to the implementation of `GrB_mxm` in SuiteSparse:GraphBLAS, using $\mathbf{C}\langle\mathbf{L}\rangle = \mathbf{L}^2$. Each thread uses a workspace of n bytes. Computing each column $\mathbf{C}_{*\mathbf{j}}$ can be done in parallel using a sparse saxpy-based method [6].
3) `tri_logm`: similar to `tri_mark`, except that a binary search is used. When computing the matrix product $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}$, a binary search is used if the jth column of the mask $\mathbf{M}$ is very sparse compared with the kth column of $\mathbf{A}$.
4) `tri_bit`: same as `tri_mark`, except that the workspace is reduced to n bits.
5) `tri_dot`: using dot products, $\mathbf{C}\langle\mathbf{L}\rangle = \mathbf{U}^T\mathbf{L}$ where $\mathbf{U}$ is the upper triangular part of $\mathbf{A}$.

The "simple" method (`tri_simple`) is already a non-trivial method. It requires expert knowledge of how Gustavson's method can be implemented efficiently, including a reduction of the result to a single scalar. The full method is shown below. The `tri_mark` method is much more complex than this. Full details and code are available at http://suitesparse.com.

```
int64_t tri_simple               // # of triangles
(
    const int64_t *restrict Ap,  // column pointers, size n+1
    const int64_t *restrict Ai,  // row indices
    const int64_t n              // A is n-by-n
)
{
    bool *restrict Mark = (bool *) calloc (n, sizeof (bool)) ;
    if (Mark == NULL) return (-1) ;
    int64_t ntri = 0 ;
    for (int64_t j = 0 ; j < n ; j++)
    {
        // scatter A(:,j) into Mark
        for (int64_t p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            Mark [Ai [p]] = 1 ;
        }
        // sum(C(:,j)) where C(:,j) = (A * A(:,j)) .* Mark
        for (int64_t p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            const int64_t k = Ai [p] ;
            // C(:,j) += (A(:,k) * A(k,j)) .* Mark
            for (int64_t pa = Ap [k] ; pa < Ap [k+1] ; pa++)
            {
                // C(i,j) += (A(i,k) * A(k,j)) .* Mark
                ntri += Mark [Ai [pa]] ;
            }
        }
        for (int64_t p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            Mark [Ai [p]] = 0 ;
        }
    }
    free (Mark) ;
    return (ntri) ;
}
```

Multicore parallelism via OpenMP is used for each of the methods except `tri_simple`. SuiteSparse:GraphBLAS, by contrast, is currently only single-threaded. The `tri_*` functions can exploit the fact that the matrix `C` need not be explicitly stored, saving time and space. This cannot be done in GraphBLAS, even if non-blocking mode were to be exploited, because `GrB_reduce` forces all pending operations to be completed on its input matrix.

Experiments were performed on an IBM Minsky system, with 1TB of RAM and 160 hardware threads (IBM Power8

8335-GTB, 4GHz, 20 hardware cores with 8-way threading on each core, `xlc` v31.1.5 compiler). Table II reports the rate of each method on all 63 graphs in the GraphChallenge test set with 200,000 or more edges. The results for the MATLAB `tricount.m` does not appear in Table II since it is prohibitively slow and takes far too much memory.

The rate is $10^{-6}e/t$ where e is the number edges in the graph and t is the run time in seconds. The parallel codes were tested with 1, 2, 4, 8, 16, 32, 64, 128, and 160 OpenMP threads; fastest results for any of these runs are shown in the table. Highest rates are shown in bold. All results include the time to construct `L=tril(A)`, and also `U=triu(A)` for the dot product methods.

IV. K-TRUSS IN GRAPHBLAS

The k-truss $\mathbf{C}$ of a graph $\mathbf{A}$ is a subgraph with the same number of nodes, but where each edge in the k-truss appears in at least $k-2$ triangles in $\mathbf{A}$. The term c_{ij} is the number of triangles containing the edge (i,j). The k-truss can be computed in MATLAB, based on Burkhardt's method [7]:

```
function C = ktruss (A,k)
last_cnz = nnz (A) ;
C = A ;
while (1)
    C = (C*C) .* C ;
    C = C .* (C >= k-2) ;
    cnz = nnz (C) ;
    if (cnz == last_cnz) return ; end
    C = spones (C) ;
    last_cnz = cnz ;
end
```

Below is the equivalent computation in GraphBLAS, almost as elegant and much faster since $\mathbf{C}\langle\mathbf{C}\rangle = \mathbf{C}^2$ does not need to form all of $\mathbf{C}^2$, but only those entries in the pattern of $\mathbf{C}$.

```
// support function
bool sfunc (const GrB_Index i, const GrB_Index j,
    const GrB_Index nrows, const GrB_Index ncols,
    const int64_t *x, const int64_t *support)
{
    return ((*x) >= (*support)) ;
}

GrB_Matrix ktruss_graphblas (GrB_Matrix A, int64_t k)
{
    GxB_SelectOp supportop ;
    GrB_Index n ;
    GrB_Matrix C ;
    GrB_Matrix_nrows (&n, A) ;
    GrB_Matrix_new (&C, GrB_INT64, n, n) ;
    // create the select operator
    int64_t s = (k-2) ;
    GxB_SelectOp_new (&supportop, sfunc, GrB_INT64) ;
    // last_cnz = nnz (A)
    GrB_Index cnz, last_cnz ;
    GrB_Matrix_nvals (&last_cnz, A) ;

    for (int64_t nsteps = 1 ; ; nsteps++)
    {
        // first step: C<A>=A*A ; subsequent steps: C<C>=C*C
        GrB_Matrix T = (nsteps == 1) ? A : C ;
        // note the PLUS-(Logical AND) semiring
        GrB_mxm (C, T, NULL, GxB_PLUS_LAND_UINT64, T, T, NULL);
        // drop entries in C less than k-2
        GxB_select (C, NULL, NULL, supportop, C, &s, NULL) ;
        // cnz = nnz (C)
        GrB_Matrix_nvals (&cnz, C) ;
        if (cnz == last_cnz) return (C) ;
        last_cnz = cnz ;
    }
}
```

If all non-empty k-trusses are desired, the kth truss can be computed more quickly by starting with the $(k-1)$st truss. This can be done in GraphBLAS with little change to the above code, resulting in an All-k-truss method. The methods were implemented both in GraphBLAS and in pure C, without using GraphBLAS. The pure-C methods exploit OpenMP parallelism, and can exploit the fact that both $\mathbf{C}\langle\mathbf{C}\rangle = \mathbf{C}^2$ and the select step can be done in-place. Table III reports the performance on the same system and the first 50 matrices from Table II. In the table, `k` is the smallest k for which the k-truss is empty. Since All-k-truss finds $k-2$ trusses, the rate is $10^{-6}(k-2)e/t$, to compare with the 3-truss results.

The Graph500 and MAWI sets are excluded from Table III. The Graph500 matrices have many cliques and thus require many passes of the all-k-truss methods. For the MAWI set, the saxpy formulation is prohibitively slow as compared to the dot-product methods. For this paper, only the saxpy-based formulations of k-truss and all-k-truss have been implemented.

V. CONCLUSIONS

Triangle counting in SuiteSparse:GraphBLAS is competitive with a highly optimized single-threaded method (`tri_mark`), even faster for some larger graphs. The `tri_simple` method is slower than GraphBLAS, yet the user-level algorithm in GraphBLAS is much simpler code. K-truss in GraphBLAS is also simple and the performance is competitive with a highly optimized and complex algorithm in pure C; the GraphBLAS 3-truss and All-k-truss methods rarely take more than twice the time as the sequential versions in pure C, and are sometimes faster. These results demonstrate that GraphBLAS can be an efficient library that allows end users to write simple yet fast code. All codes used in this paper are at http://suitesparse.com.

A parallel SuiteSparse:GraphBLAS is in progress, and it should be able to match the pure-C parallel versions of these algorithms, or exceed them if a scalable heap-based parallel sparse matrix-matrix multiply were adopted [8].

REFERENCES

[1] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "The GraphBLAS C API specification," Tech. Rep., 2017, http://graphblas.org/.

[2] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. Philadelphia, PA: SIAM, 2011.

[3] 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.

[4] J. Cohen, "Graph twiddling in a map-reduce world," *Computing in Science and Eng.*, vol. 11, no. 4, pp. 29–41, July 2009.

[5] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *2015 IEEE Intl. Parallel and Distributed Proc. Symp. Workshop*, May 2015, pp. 804–811.

[6] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Trans. Math. Softw.*, vol. 4, no. 3, pp. 250–269, 1978.

[7] P. Burkhardt, "Graphing trillions of triangles," *Information Visualization*, vol. 16, no. 3, Sept 2017.

[8] A. Buluç and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *IPDPS08: the IEEE International Symposium on Parallel & Distributed Processing*. IEEE Computer Society, 2008, pp. 1–11.

TABLE II

TRIANGLE COUNTING PERFORMANCE (HIGHER IS BETTER), BEST SINGLE-THREADED AND MULTI-THREADED RESULTS IN BOLD FONT

Matrix			single-threaded performance							multi-threaded performance			
				saxpy method				dot product		saxpy method			dot
name	nodes (n) $\times 10^6$	edges (e) $\times 10^6$	Graph BLAS	tri_ simple	tri_ mark	tri_ bit	tri_ logm	Graph BLAS	tri_ dot	tri_ mark	tri_ bit	tri_ logm	tri_ dot
Kronecker products (synthetic)													
Theory-3-4-5-9-16-B1k	0.02	0.22	6.2	2.3	**24.1**	0.7	2.0	5.9	2.2	**24.1**	4.2	8.3	10.1
Theory-3-4-5-9-16-B2k	0.02	0.22	25.0	31.5	**79.7**	14.1	57.4	10.3	58.0	84.7	17.2	**91.4**	58.0
Theory-256-625-Bk	0.16	0.32	41.8	24.4	**132.4**	24.8	17.7	29.8	23.2	**158.8**	148.8	114.3	128.7
Theory-256-625-B1k	0.16	0.32	39.9	22.6	**124.1**	14.5	15.0	28.1	17.5	**144.4**	18.9	85.4	141.4
Theory-256-625-B2k	0.16	0.32	19.8	20.2	**131.6**	25.2	18.5	29.6	24.7	**155.6**	35.4	31.7	136.2
Theory-4-5-9-16-25-Bk	0.13	1.15	64.7	26.6	**134.4**	28.8	20.7	33.9	25.1	**283.8**	98.6	96.3	157.9
Theory-4-5-9-16-25-B1k	0.13	1.58	1.9	0.9	**4.1**	0.2	0.9	3.8	2.5	9.6	2.5	4.9	**18.1**
Theory-4-5-9-16-25-B2k	0.13	1.58	10.3	0.9	**20.9**	1.1	3.7	7.8	12.3	42.4	7.0	24.5	**141.8**
Theory-25-81-256-Bk	0.55	2.07	52.1	24.4	**78.5**	27.9	20.6	31.4	21.7	137.6	95.5	**151.8**	89.1
Theory-25-81-256-B1k	0.55	2.13	8.6	4.0	**10.2**	1.4	3.7	6.2	5.9	21.4	3.9	7.8	**58.7**
Theory-25-81-256-B2k	0.55	2.13	55.6	1.2	**64.1**	14.4	18.1	16.4	20.0	**139.2**	41.2	70.6	81.7
Theory-9-16-25-81-Bk	0.36	2.33	60.0	28.2	**83.2**	28.1	22.7	33.0	23.2	**186.9**	123.6	179.9	127.2
Theory-9-16-25-81-B1k	0.36	2.61	2.3	1.1	**8.1**	0.4	1.1	6.9	3.6	8.1	0.8	2.2	**33.4**
Theory-9-16-25-81-B2k	0.36	2.61	**18.7**	0.7	18.1	2.6	6.9	6.0	17.3	38.2	6.7	20.5	**109.4**
Theory-3-4-5-9-16-25-Bk	0.53	6.91	**66.9**	30.1	35.9	30.8	24.5	33.1	17.4	**288.8**	188.4	236.6	83.6
Theory-3-4-5-9-16-25-B1k	0.53	11.08	0.5	0.2	0.2	0.1	0.2	**1.6**	1.4	3.2	0.7	1.5	**10.2**
Theory-3-4-5-9-16-25-B2k	0.53	11.08	2.9	0.4	1.3	0.4	1.4	3.6	**8.7**	24.6	9.7	25.4	**76.5**
Theory-5-9-16-25-81-Bk	2.17	23.33	**61.7**	34.0	37.2	37.0	28.1	30.7	16.8	**139.2**	110.2	106.9	52.3
Theory-5-9-16-25-81-B1k	2.17	28.67	0.3	0.1	0.1	0.0	0.1	1.6	**1.6**	1.0	0.5	1.1	**13.6**
Theory-5-9-16-25-81-B2k	2.17	28.67	1.7	0.1	0.8	0.3	0.8	3.6	**11.9**	20.9	8.9	20.1	**68.3**
SNAP (various real graphs)													
loc-brightkite_edges	0.06	0.21	13.2	6.2	**33.8**	2.7	5.3	5.5	1.0	**138.1**	20.5	71.5	5.7
cit-HepTh	0.03	0.35	7.9	3.7	**26.5**	1.3	3.1	3.6	0.6	**128.0**	18.3	37.0	5.8
soc-Epinions1	0.08	0.41	6.5	3.3	**20.0**	1.0	3.2	2.3	0.3	**124.2**	17.0	47.2	2.8
email-EuAll	0.27	0.36	11.4	1.4	**26.0**	1.9	4.3	5.5	0.9	**132.8**	71.9	96.5	32.8
cit-HepPh	0.03	0.42	10.3	5.1	**30.9**	2.1	4.3	4.5	0.7	**177.8**	37.8	66.6	9.4
soc-Slashdot0811	0.08	0.47	8.5	3.6	**21.5**	1.2	3.4	3.0	0.4	80.2	22.4	**81.1**	3.4
soc-Slashdot0902	0.08	0.50	8.4	3.6	**21.1**	1.1	3.3	2.9	0.4	**101.4**	25.4	96.9	4.5
loc-gowalla_edges	0.20	0.95	5.0	2.4	**14.8**	0.7	2.3	3.9	0.7	**99.6**	19.6	69.3	18.5
amazon0302	0.26	0.90	16.2	11.0	**36.0**	6.0	9.0	10.2	5.1	**273.5**	242.2	218.5	167.2
roadNet-PA	1.09	1.54	12.6	10.1	**32.5**	9.1	9.0	18.7	10.5	**252.1**	233.4	224.2	198.9
roadNet-TX	1.38	1.92	12.7	12.9	**25.4**	11.2	11.9	18.8	12.2	**275.7**	269.8	193.1	209.6
flickrEdges	0.11	2.32	2.3	1.7	**4.0**	0.5	1.6	1.0	0.1	**40.6**	20.0	26.6	3.2
amazon0312	0.40	2.35	11.2	8.5	**12.8**	4.1	6.9	6.4	2.7	231.7	**239.7**	222.9	125.4
amazon0505	0.41	2.44	11.3	9.2	**13.5**	4.8	7.9	6.4	2.7	256.2	**286.4**	233.6	133.1
amazon0601	0.40	2.44	11.3	8.5	**14.1**	3.9	6.5	6.3	2.2	**237.6**	161.3	234.6	95.6
roadNet-CA	1.97	2.77	11.8	12.6	**18.1**	9.7	9.6	17.2	9.9	**280.2**	221.1	201.2	153.8
cit-Patents	3.77	16.52	**6.0**	4.9	4.5	3.9	4.4	3.9	2.1	104.3	**155.0**	107.1	67.1
friendster	119.43	1800.00	**5.7**	2.1	2.6	2.6	2.4	3.7	1.2	56.3	61.3	55.7	**61.8**
GenBank (protein k-mers)													
GenBank/V2a	55.04	58.61	9.7	9.6	9.8	9.6	9.2	**13.0**	7.0	114.4	**144.0**	113.8	59.1
GenBank/U1a	67.72	69.39	10.5	9.7	10.3	10.6	9.4	**15.7**	7.3	116.7	**146.9**	119.4	60.8
GenBank/P1a	139.35	148.91	10.1	9.5	10.3	10.5	9.3	**14.5**	7.3	111.5	**133.9**	111.8	58.1
GenBank/A2a	170.73	180.29	9.6	9.0	9.7	10.0	9.0	**14.9**	7.0	126.8	**155.4**	127.1	63.0
GenBank/V1r	214.01	232.71	13.9	13.2	12.9	12.8	11.8	**19.0**	9.2	143.5	**174.9**	142.6	72.7
image-grids													
g-260610-65536	0.07	0.26	26.0	15.5	**123.6**	7.1	10.1	17.4	7.9	**354.1**	195.2	203.5	241.4
g-1045506-262144	0.26	1.05	25.6	14.9	**117.1**	7.8	10.2	17.6	7.8	**418.1**	233.8	306.9	324.6
g-4188162-1048576	1.05	4.19	**24.8**	16.8	16.5	7.6	11.2	17.3	6.8	**395.6**	282.6	330.0	198.2
g-16764930-4194304	4.19	16.76	**24.5**	18.9	9.0	7.4	11.2	17.2	5.3	282.1	265.8	**303.1**	113.5
g-67084290-16777216	16.78	67.08	**24.4**	21.2	16.6	11.6	15.0	17.2	9.2	**298.8**	277.3	277.2	145.8
g-268386306-67108864	67.11	268.39	**24.3**	20.2	15.7	10.5	13.7	17.1	8.3	271.3	**294.1**	266.4	151.3
g-1073643522-268435456	268.44	1073.64	**23.9**	19.9	15.7	12.2	14.8	17.0	8.6	300.5	**315.9**	294.2	168.2
Graph500 (synthetic)													
graph500-scale18-ef16	0.17	3.80	1.2	0.6	**2.5**	0.3	0.6	0.5	0.1	**25.5**	5.3	9.4	0.7
graph500-scale19-ef16	0.34	7.73	**0.9**	0.5	0.7	0.2	0.4	0.4	0.1	**7.1**	3.7	6.8	0.7
graph500-scale20-ef16	0.65	15.68	**0.6**	0.3	0.3	0.1	0.3	0.3	0.0	4.9	2.9	**5.0**	1.1
graph500-scale21-ef16	1.24	31.73	**0.4**	0.2	0.2	0.1	0.2	0.2	0.0	**6.8**	4.5	6.3	1.1
graph500-scale22-ef16	2.39	64.10	**0.3**	0.1	0.1	0.1	0.1	0.2	0.0	4.6	4.8	**5.0**	0.5
graph500-scale23-ef16	4.61	129.25	**0.2**	0.1	0.1	0.1	0.1	0.1	0.0	1.9	**3.7**	1.9	1.0
graph500-scale24-ef16	8.86	260.26	**0.1**	0.1	0.1	0.0	0.1	0.1	0.0	1.0	**3.4**	1.1	0.4
graph500-scale25-ef16	17.04	523.47	0.1	0.0	0.0	0.0	0.0	**0.1**	0.0	0.6	**1.9**	0.7	0.4
MAWI internet traffic													
mawi_201512012345	18.57	19.02	16.3	-	6.9	3.4	6.0	**22.3**	11.5	**40.1**	32.6	35.1	31.1
mawi_201512020000	35.99	37.24	8.8	-	3.8	1.7	3.5	**19.0**	11.7	**37.3**	34.6	32.7	21.8
mawi_201512020030	68.86	71.71	5.0	-	2.1	0.9	1.9	**21.6**	11.2	19.8	16.2	20.4	**27.5**
mawi_201512020130	128.57	135.12	2.7	-	1.2	0.5	1.1	**18.5**	10.6	16.3	13.1	16.9	**25.1**
mawi_201512020330	226.20	240.02	1.3	-	0.5	0.2	0.5	**18.7**	9.6	14.7	10.7	14.7	**25.4**

TABLE III
3-TRUSS AND ALL-k-TRUSS PERFORMANCE (HIGHER IS BETTER)

Matrix			3-truss			All-k-truss			
	nodes	edges	Graph	no GraphBLAS		k	Graph	no GraphBLAS	
name	$\times 10^6$	$\times 10^6$	BLAS	seq.	para.		BLAS	seq.	para.
Kronecker products (synthetic)									
Theory-3-4-5-9-16-B1k	0.02	0.22	0.1	0.2	1.4	20	1.5	2.5	14.1
Theory-3-4-5-9-16-B2k	0.02	0.22	0.9	0.8	5.9	7	4.4	3.9	20.3
Theory-256-625-Bk	0.16	0.32	31.6	0.0	0.3	3	31.5	0.0	0.3
Theory-256-625-B1k	0.16	0.32	0.4	0.0	0.3	4	0.8	0.0	0.5
Theory-256-625-B2k	0.16	0.32	1.7	0.0	0.3	4	3.3	0.0	0.5
Theory-4-5-9-16-25-Bk	0.13	1.15	67.7	0.1	2.3	3	67.4	0.1	2.2
Theory-4-5-9-16-25-B1k	0.13	1.58	0.0	0.0	0.7	29	0.6	0.9	15.2
Theory-4-5-9-16-25-B2k	0.13	1.58	0.2	0.1	1.9	7	1.0	0.7	11.0
Theory-25-81-256-Bk	0.55	2.07	41.7	0.0	0.2	3	42.0	0.0	0.2
Theory-25-81-256-B1k	0.55	2.13	0.0	0.0	0.2	29	0.6	0.2	5.0
Theory-25-81-256-B2k	0.55	2.13	0.1	0.0	0.2	5	0.3	0.0	0.7
Theory-9-16-25-81-Bk	0.36	2.33	58.4	0.0	0.6	3	58.5	0.0	0.6
Theory-9-16-25-81-B1k	0.36	2.61	0.0	0.0	0.3	29	0.5	0.4	8.7
Theory-9-16-25-81-B2k	0.36	2.61	0.1	0.0	0.8	6	0.4	0.1	2.6
Theory-3-4-5-9-16-25-Bk	0.53	6.91	65.5	0.1	1.8	3	65.4	0.1	1.7
Theory-3-4-5-9-16-25-B1k	0.53	11.08	0.0	0.0	0.3	63	0.4	0.6	11.5
Theory-3-4-5-9-16-25-B2k	0.53	11.08	0.1	0.1	1.5	8	0.4	0.4	9.3
Theory-5-9-16-25-81-Bk	2.17	23.33	48.3	0.0	0.2	3	48.5	0.0	0.2
Theory-5-9-16-25-81-B1k	2.17	28.67	0.0	0.0	0.1	85	0.2	0.2	5.7
Theory-5-9-16-25-81-B2k	2.17	28.67	0.0	0.0	0.2	7	0.1	0.0	1.2
SNAP (various real graphs)									
loc-brightkite_edges	0.06	0.21	1.4	2.3	10.5	44	1.9	3.4	9.0
cit-HepTh	0.03	0.35	0.8	1.4	9.1	31	0.4	0.8	5.9
soc-Epinions1	0.08	0.41	0.5	1.1	6.6	34	0.4	0.8	6.0
email-EuAll	0.27	0.36	0.4	0.7	7.4	21	1.8	3.5	24.2
cit-HepPh	0.03	0.42	1.0	1.9	14.4	26	0.6	1.2	9.8
soc-Slashdot0811	0.08	0.47	0.9	1.7	9.5	36	2.1	3.9	21.4
soc-Slashdot0902	0.08	0.50	0.8	1.6	8.6	37	2.1	3.9	23.0
loc-gowalla_edges	0.20	0.95	0.4	0.8	9.2	30	0.7	1.3	12.9
amazon0302	0.26	0.90	2.2	3.9	29.7	8	2.9	5.5	35.0
roadNet-PA	1.09	1.54	11.1	14.9	58.4	5	18.0	31.6	84.9
roadNet-TX	1.38	1.92	10.8	15.1	56.6	5	17.9	32.5	113.3
flickrEdges	0.11	2.32	0.2	0.4	3.5	575	0.2	0.3	4.3
amazon0312	0.40	2.35	1.3	2.3	21.6	12	1.0	1.7	12.8
amazon0505	0.41	2.44	1.2	2.3	20.3	12	0.9	1.7	13.7
amazon0601	0.40	2.44	1.2	2.3	21.2	12	1.1	2.0	16.5
roadNet-CA	1.97	2.77	10.5	15.0	57.7	5	17.3	32.5	113.5
cit-Patents	3.77	16.52	0.9	1.4	11.5	37	5.3	9.7	64.1
friendster	119.43	1800.00	0.1	0.2	1.0	6	0.5	0.7	3.9
GenBank (protein k-mers)									
GenBank/V2a	55.04	58.61	6.3	9.6	30.9	4	9.5	16.6	59.3
GenBank/U1a	67.72	69.39	6.8	9.5	34.6	4	9.7	14.4	57.0
GenBank/P1a	139.35	148.91	6.6	8.4	31.3	4	9.8	15.0	56.7
GenBank/A2a	170.73	180.29	6.2	8.5	35.1	4	10.0	15.1	55.4
GenBank/V1r	214.01	232.71	11.5	18.5	60.2	4	14.8	29.3	87.1
image-grids									
g-260610-65536	0.07	0.26	11.4	16.8	128.5	5	18.5	35.6	142.5
g-1045506-262144	0.26	1.05	11.4	29.5	149.4	5	18.2	50.3	144.3
g-4188162-1048576	1.05	4.19	11.0	36.6	168.6	5	17.7	56.8	156.7
g-16764930-4194304	4.19	16.76	10.5	36.9	166.0	5	17.5	58.8	162.0
g-67084290-16777216	16.78	67.08	11.0	39.5	199.7	5	16.8	58.7	153.6
g-268386306-67108864	67.11	268.39	10.2	38.9	190.7	5	17.5	57.3	148.4
g-1073643522-268435456	268.44	1073.64	10.1	38.6	199.9	5	17.0	55.5	149.5