Introducing Streaming into Linear Algebra-based Sparse Graph Algorithms

Peter M. Kogge, Neil A. Butcher, Brian A. Page
Dept. of Computer Science and Engr.
Univ. of Notre Dame
Notre Dame, IN USA
{kogge,nbutcher,bpage1}@nd.edu

***Abstract*—GraphBLAS is a new package designed to provide a standard set of building blocks for graph algorithms based formally in the language of linear algebra. This paper suggests some extensions of the underlying math that would enhance GraphBLAS' ability to stream updates into a computation without a bulk recomputation, and at greatly reduced computational complexity. The process is applied to several examples.**

***Index Terms*—Big Data and Data Analytics, Programming Languages, Causality and Time, Compiler Design and Optimization**

I. Introduction

The **Basic Linear Algebra Subprograms** (BLAS)[1] package provides a standard set of building blocks for basic vector and matrix operations. LINPACK[2] and LAPACK[3] builds on BLAS to provide highly scalable, parallel implementations of linear algebra using conventional arithmetic operations on dense matrices. They form the benchmark used for the Top500 rankings of supercomputers[4]. In contrast, **GraphBLAS**[5] provides a standard set of building blocks for graph algorithms using the language of linear algebra. A graph can be defined by an **adjacency matrix** of size NxN where N is the number of vertices V in the graph. The $[i, j]$ entry of such a matrix is non-zero if there is an edge from $V[i]$ to $V[j]$. Non-zeros other than "1" can represent weights on edges.

GraphBLAS uses this representation at its heart. The innovations are two-fold. First, the programmer may specify operations different from the traditional floating-point add and multiply, but still compatible with the rules of linear algebra, By choosing these functions carefully, virtually any graph kernel operation can be re-written as a series of linear algebra operations (cf. [17]). A popular implementation is SuiteSPARSE [10]. Some very detailed examples are bipartite matching [3], triangle counting [21], and k-truss [1].

The second innovation is on handling **extreme sparsity** where nearly all of the elements of matrices are the equivalent of "zeros." This is critically important for real graph applications where there may be literally billions of vertices, but perhaps at best a few dozen edges from any one vertex.

Finally, GraphBLAS hides the storage of such matrices in an *opaque space* where a wide variety of novel execution models and platforms, especially parallel and/or lazy, may be employed, with their implementations hidden from the users. In its current form this is an excellent match for graph algorithms that are to be executed in "batch mode" where computation is applied to entire matrices at a time.

This paper looks at enhancing GraphBLAS to avoid recomputing an entire matrix operation when a single edge is to be changed. This is particularly important when applications are "streaming," as in real-time applications. The emphasis is on defining how the properties of the functions used in the linear algebra operations affect the complexity of doing computations incrementally.

In organization Section II briefly reviews some key prior work in streaming graphs. Section III focuses on the math framework behind GraphBLAS. Section IV suggests some extensions to reduce the computational complexity of streaming within linear algebra. Section V describes some examples. Section VI addresses concurrency issues.

II. Prior Streaming Work

Graph streaming algorithms do have a some history. A prior definition of streaming (c.f. [5]) focused on cases where the edge set is too big to be held close to the processor. The edges thus must be accessed in some "stream" order, and where that order may not be in the order needed by the algorithm. Multiple passes may thus be needed to access the necessary edges. Such work relates compute complexity to storage needs, expressed as a function of how many passes over the input stream are required. Such work also explores existence proofs for streaming algorithms by reducing problems to others with known solutions. In particular, the *semi-streaming* model [18] develops tradeoffs between storage used to save intermediate edges referenced by the sequence order and needed computation. [12] includes examples of bipartite matching.

At the implementation level, the low level Stinger data structure [4] was designed from the beginning to hold graphs in memory and provide an API to handle streaming updates.A sample benchmark code [20] used Stinger to perform edge

[1]http://www.netlib.org/blas/
[2]http://www.netlib.org/linpack/
[3]http://www.netlib.org/lapack/
[4]https://www.top500.org/
[5]http://graphblas.org/index.php?title=Graph_BLAS_Forum

insertion and deletion operations into a Stinger graph at very high rates when the updates can be batched. The Stinger implementation focuses on synchronization issues to guarantee correct operation. A parallel streaming algorithm for updating cluster coefficients [11] also uses Stinger, with an OpenMP implementation running on both a modern multi-core and a Cray XMT multi-threaded platform.

This paper focuses on *how to derive algorithms* where the initial graph is "in memory," and a stream of updates to that graph is to be handled. The results are intended on demonstrating how a potentially compile-time process may construct a streaming algorithm from a linear algebra-based algorithm that performs a non-streaming batch computation.

III. GraphBLAS Fundamentals

A. Graphs as Matrices

A *graph* G is a pair (V, E) where V is a set of $vertices$ and E is a set of *edges*. Each edge is itself a pair (u, v) where u and v are both vertices from V. G is an *undirected graph* if some (u, v) is in E, then so is (v, u). A *directed graph* does not have this duality.

GraphBLAS performs computation over graphs that are expressed as 2D matrices in one of two forms. An *adjacency matrix* is a $|V|x|V|$ matrix A where $A[u, v]$ is a "1" if u and v are vertices, and (u, v) is in E. If (u, v) is not an edge in G, then $A[u, v]$ is a "0".

The other form is an $|V|x|E|$ *incidence matrix* B where $B[u, e]$ is a "1" if the vertex u is one of the endpoints of edge e. For directed graphs there may be two matrices; $B_{out}[u, e] = 1$ if (u, v) is in E for some v, and $B_{in}[v, e] = 1$ if there is an edge (u, v) for some u.

The "1s" above need not always be the number 1, but may come from some other domain. For example, in an adjacency matrix where edges are *weighted* (as in a road map where edges are distances between cities), the non-zero value for an edge may be the weight. Likewise, for directed graphs, $B[u, e]$ may be "+1" if $e = (u, v)$ and "-1" if $e = (v, u)$.

B. Functional Properties

A GraphBLAS program consists of the application of a series of *operations* to the matrices and vectors that make up a graph. These operations may in turn use *functions* that may either come from a predefined library or be user-defined. GraphBLAS assumes that such functions, both individually and collectively, have certain properties. This section discusses those properties formally, with notation from APL [13].

A *binary function* "$\circ$" is a function whose two arguments a and b come from two sets D_{left} and D_{right} respectively (called the *domain*), and produce a result c from a third set D_{out} called the *range*. When $D_{left} = D_{right} = D$, the function $\circ$ may have special properties as follows:

- $\circ$ is *commutative* if for all a and b, $\circ(a, b) = \circ(b, a)$.
- $\circ$ has an *identity element* $I_\circ$ (from D) if for all a in D, $\circ(a, I_\circ) = \circ(I_\circ, a) = a$.
- b° is a *left element inverse* of b if $\circ(b^\circ, b) = I_\circ$
- a° is a *right element inverse* of a if $\circ(a, a^\circ) = I_\circ$
- If for an element a there is both a left and right element inverse, and they are the same, then a is *invertible*.

If also $D_{out} = D_{left} = D_{right} = D$, then:

- $\circ$ is *associative* if for all a, b, and c, $\circ(a, \circ(b, c)) = \circ(\circ(a, b), c)$.
- If it exists, an *annihilator* (or *zero*) for $\circ$ is some element $0_\circ$ such that $\circ(0_\circ, a) = \circ(a, 0_\circ) = 0_\circ$ for any element a.
- $\circ$ is *closed* under D if for any (a, b), $\circ(a, b)$ is also in D.

If a function $\circ$ is associative and has an identity $I_\circ$, it is called a *monoid*. If in addition $\circ$ is closed under D and all elements of D have inverses, then $\circ$ and D form a *group*.

A pair of functions $\oplus$ and $\otimes$ form a *semi-ring* [16] if:

- $\oplus$ is a commutative monoid with identity $I_\oplus$,
- $\otimes$ is also a monoid with identity element $I_\otimes$,
- The range of $\otimes$ is the same as $D_\oplus$,
- $I_\oplus$ is an annihilator of $\otimes$, that is $I_\oplus = 0_\otimes$,
- $\otimes$ *distributes* over $\oplus$, that is $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$.

C. Notation

We use the following notation for expressions:

- If a and b are scalars, either $a \circ b$ or $\circ(a, b)$ represents the result of the single application of function $\circ$ to a and b.
- If x, y, and z are vectors, then in $z = x \circ y$, $z[i] = x[i] \circ y[i]$.
- If $\circ$ is a monoid, and x is a vector from $D_\circ$, then $\circ/x$ is called a *reduction* of x, and returns the scalar $(...(x[1] \circ x[2]) \circ x[3])... \circ x[n])$, e.g. $+/x$ is the sum of all elements.
- If $\oplus$ and $\otimes$ form a semi-ring, then the *inner product* of x and y, denoted $x \oplus . \otimes y$ is the same as $\oplus/(x \otimes y)$.
- Likewise, if A is an NxM matrix then $A \oplus . \otimes x$ (where x is of length M) is the vector z where $z[i]$ is the inner product of $A[i;]$ and x.
- Likewise, if A is an NxM matrix and B is an MxR matrix, then the *matrix product* $C = AB$ may also be written as $C = A \oplus . \otimes B$, and represents the NxR matrix C where $C[i, j] = A[i;] \oplus . \otimes B[; j]$.
- $v. \circ w$ where v and w are vectors, is the matrix C where $C[i, j] = v[i] \circ w[j]$.

D. Structural Zeros

Section III-A used "0" as a placeholder for the value of a graph matrix entry where there is no corresponding edge. Since most graphs are very *sparse*, most entries in a graph matrix are such "0"s, and the GraphBLAS spec encourages implementations to avoid explicitly "storing" such zeros.

Further, the same graph matrix may be used with several different $(\otimes, \oplus)$ function pairs in different steps of some algorithm. In addition, vectors may also be "sparse" with lots of "zeros" in them. In all such cases the "0"s may mean different values at different times, depending on the functions being applied. GraphBLAS treats all such "0"s as *structural zeros*, having no intrinsic value. In general, all such structural zeros are assumed to be the same as the identify element $I_\oplus$ of the current $\oplus$ (and for inner products thus the same as the annihilator element $0_\otimes$ of the current $\otimes$).

For this paper, we define the function $nz(x)$ as returning the number of structural zeros in x, and $nnz(x)$ as returning

TABLE I
SOME TYPICAL "INNER PRODUCT" FUNCTION PAIRS.

Summation $\oplus$	Product $\otimes$	Relevent Kernels	$\oplus$ a Monoid?	$\otimes$ a Monoid?	$\oplus, \otimes$ Semi-ring?	Domain $\oplus$	$\oplus$ Associative	$\oplus$ Commutative	$I_{\oplus}$	$\oplus$ inverse	Domain $\otimes$ D_{in1}	Domain $\otimes$ D_{in2}	$\otimes$ Associative	$\otimes$ Commutative	$\otimes$ Dist. over $\oplus$	$I_{\otimes}$	$0_{\otimes}$
$+$	$*$	Lin. Alg. [16] Betweenness	Y	Y	Y	Z, R	Y	Y	0	Y	Z, R	Z, R	Y	Y	Y	1	0
$+$	$*$	$\triangle$ Finding [9]	Y	Y	Y	Z^+	Y	Y	0	N	Z^+	Z^+	Y	Y	Y	1	0
max	min		Y	Y	Y	Z, R	Y	Y	$-\infty$	N	Z, R	Z, R	Y	Y	Y	∞	$-\infty$
xor	and	BFS [16]	Y	Y	N	bool	Y	Y	F	Y	bool	bool	Y	Y	N	T	F
$\cup$	$\cap$	[16]	Y	Y	N	U	Y	Y	$\emptyset$	N	U	U	Y	Y	N	U	$\emptyset$
max	$+$	max flow [16]	Y	Y	N	Z, R	Y	Y	$-\infty$	N	Z, R	Z, R	Y	Y	N	0	$-\infty$
max	$select2$	Max ind. set [9]	Y	N	N	Z, R	Y	Y	$-\infty$	N	U	Z, R	Y	N	N		
U is the universal set; $\emptyset$ is the empty set. $select2(x, y) = y$																	

the number of "non-zeros" in x, where x is either a vector or a matrix. $nnz(A[i;])$ is the number of non-zeros in row i of matrix A; $nnz(A[;j])$ is the number of non-zeros in column j of matrix A; while $nnz_{row}(A)$ is the average number of non-zeros in any row of matrix A.

E. Some Function Examples

Table I lists examples of useful pairs $(\oplus, \otimes)$ of functions taken from the GraphBLAS references [9], [16], [17]. What is interesting is that not all pairs have all the properties mentioned above, especially identities, inverses, or distributively, and thus strictly speaking are not semi-rings.

F. Other Graph Compute Patterns

TABLE II
GRAPHBLAS OPERATION FORMATS.

Name	Structure	Operation
Inner-Product Operations		
Vec-Vec	$s \leftarrow V_1 \oplus . \otimes V_2$	$s = V_1 \oplus . \otimes V_2$
Mat-Vec	$V_2 \leftarrow MxV_1$	$V_2[i] = M[i;] \oplus . \otimes V_1$
Mat-Mat	$M_3 \leftarrow M_2xM_1$	$M_3[i;j] = M_1[i;] \oplus . \otimes M_2[;j]$
Element by Element Operations		
Scalar-Vec	$V_2 \leftarrow s \circ V_1$	$V_2[i] = s \circ V_1[i]$
Vec-Vec	$V_3 \leftarrow V_2 \circ V_1$	$V_3[i] = V_2[i] \circ V_1[i]$
Mat-Mat	$M_3 = M_2 \circ M_1$	$M_3[i;j] = M_1[i;j] \circ M_2[i;j]$

The top half of Table II three major variants of inner products, based on whether the arguments are vectors or matrices. In addition, GraphBLAS also has a suite of element-by-element operations, as summarized in the bottom of Table II. The functions ("$\circ$") involved in these operations do not have all the constraints as inner products. If $\circ$ is a $\otimes$-like function where a structural zero is an annihilator, then in $y \leftarrow w \circ x$ if some $w[i]$ (or $x[i]$) is a structural zero, then $y[i]$ is also a structural zero for the last two cases, and s is a structural zero for the first case. If $\circ$ is a $\oplus$-like function where a structural zero is an identity element, then if some $w[i]$ (or $y[i]$) is a structural zero, $y[i]$ takes on the other argument value.

GraphBLAS also allows the application of *masks* to the storing of expression results into a target. A mask is an array of booleans of the same dimensions as the target object receiving the results. For example, for a matrix mask, if for the result of a matrix computation for element $[i, j]$ the mask's $[i, j]$ entry is true, then the store goes through. If not, the $[i, j]$ element in the result is unchanged, regardless of the value computed on the right-hand side. An option on a mask application may specify that all non-masked elements in the masked object are to be zeroed first. The GraphBLAS spec uses the notation $v[q] =$, where q is a mask of the same dimensions as v, to express this option.

Also one may optionally "accumulate" a computed value into the designated target object, rather than just overwrite the prior value. The optional binary accumulation function $\odot$ is also programmer-specifiable. Thus for example $y \odot = A \oplus . \otimes x$ represents updating y as $y[i] \leftarrow y[i] \odot (A[i;] \oplus . \otimes x)$. In such cases, it is important to recognize to what a structural zero generated from the right-hand side expression corresponds. If an $I_{\odot}$, then there is no change to the left-hand side element, and any consideration of the element can be skipped. If anything else, then a computation and a store is required.

G. Implementation Considerations

The net effect of Section III-D is that in any reduction involving some $\oplus$, a structural zero in the vector being summed may be treated as equivalent to $I_{\oplus}$, and may be "skipped" in terms of performing the computation. Likewise, in any inner product $x \oplus . \otimes y$, any product $x[i] \otimes y[i]$, where either $x[i]$ or $y[i]$ is a structural zero, is itself equivalent to a $0_{\otimes}$ which if the functions form a semi-ring is equivalent to $I_{\oplus}$, and thus the $\oplus$ sum can again skip all such terms.

Next, since many graphs are notoriously very sparse, the result of operations that produce vectors or matrices are themselves liable to be sparse. Thus, an important consideration is how to generate an object which may be quite sparse without somehow performing computations proportional to the dimensions of the object (i.e. in generating a vector result of length n, we should avoid "initializing" the zero elements). Thus it is likely that some sort of hash table represent the

element, where if any index is not found in the hash table, then the matching element is assumed to be a structural zero.

Also, if (as expected for most GraphBLAS implementations) operations on vectors or matrices can skip the structural zeros, then we can assume operations on vectors x are of time complexity $O(nnz(x))$ rather than $O(|x|)$. In addition, if $\oplus$ is associative and commutative, then we expect that many GraphBLAS implementations may perform the $\oplus$ summations in an "unpredictable" order, and still return "valid" results. This would be the case if, for example, the summation is performed in parallel in some fashion.

Applying an accumulating function $\odot$ to an existing sparse object as in $v \odot = e$ also requires some special consideration. Assuming that a structural zero from the right-hand expression e is $0_{\odot}$, then nothing need be done. However, for non-zero right-hand side elements, then the index for that element need be used as a probe into v. If the index is there, then $v[i] \leftarrow v[i] \odot e[i]$. If not, then a new entry for $v[i]$ must be created, and the value set to $e[i]$.

Applying a mask m to an assignment $v[m] \leftarrow e$ has similar special cases. If $m[i]$ is false, no change to $v[i]$ is necessary, regardless of $v[i]$'s value. If $m[i]$ is true, and both $v[i]$ and $e[i]$ are structural zeros, then again no change is necessary. If both have non-zero values, then the value for $v[i]$ is updated. If $v[i]$ does not have a value, but $e[i]$ does, then a value entry for $v[i]$ must be created. If the other way around, then the value entry for $v[i]$ must be erased and replaced by a structural zero.

If the option to clear all masked elements of v is specified, then it is preferable to build a new empty data structure for v and apply the above only for non-zero $e[i]$s. This avoids a time complexity proportional to the dimensions of v.

H. Complexity

TABLE III
NOTIONAL OPERATION COMPLEXITY

Operation	Dense	Sparse
$A \circ B$	$O(NM)$	$O(nnz(A) + nnz(B)))$
$\oplus/A$	$O(NM)$	$O(nnz(A))$
$A \oplus . \otimes B$	$O(NMP)$	$O(NP(nnz_{row}(A) + nnz_{col}(B))$
$C \odot = E$	$O(NP)$	$O(nnz(E))$
$C[X] \leftarrow E$	$O(NP)$	$O(nnz(E) + nnz(X))$
A is an NxM matrix; B is MxP; C, X is NxP. E is some arbitrary expression.		

Table III compares the approximate complexity of a variety of operations found in GraphBLAS for both a traditional "dense" implementation and one designed to handle possibly large numbers of structural zeros. These complexities assume that the expression is computed in "batch" fashion. The key take-away is that for very sparse objects, an implementation that considers structural non-zeros can typically remove close to one whole dimension from the complexity.

IV. STREAMING

GraphBLAS performs all computations on data stored in an *opaque* memory space separate from the main program. An application uses GraphBLAS to perform such operations by making a series of calls to individual GraphBLAS routines. The time sequence of these calls is called *program order*. When GraphBLAS has been placed in *blocking mode*, all computations associated with one of these calls must be completed before the next in program order may be started. However, in *non-blocking mode*, from the caller's perspective the only steps of a single call that are guaranteed to be performed before control is returned are error-checking of the arguments to the call and any required transfers of data between the caller's memory space and the GraphBLAS opaque space. It is perfectly permissible to construct a directed program flow graph of operations to be executed later "as if in program order" only when a call is made to transfer data "out" of opaque memory. Only at that time are any computations specified in the flow graph that are needed to deliver the desired data guaranteed to be performed.

This non-blocking mode allows all kinds of enhanced concurrency to be performed, but still operates "batch-like," with overall operation counts as in Table III. Of growing value would be a mode akin to non-blocking where, after some program flow graph has been defined (usually with some initial data sets provided and an initial execution performed), incremental updates to that data may be input over time, and the changes echoed through the flow graph to update data structures in opaque space.

As defined here, a *streaming mode* would allow such small sets of updates to flow through the program flow graph while performing only the minimal amount of computation to compute what has changed, without recomputing from scratch using the whole data set. There are three kinds of "updates":

- *Add an edge*: change an entry in an adjacency or incidence matrix from a structural zero to a non-zero.
- *Delete an edge*: change an entry in an adjacency or incidence matrix from a non-zero to a structural zero.
- *Modify an edge weight* from a non-zero to another.

Similar changes to vectors are also possible, such as when a change to a matrix that is involved in a matrix-vector product changes an element in a result vector:

- Add an element from a structural zero to a non-zero.
- Delete an element from a non-zero to a structural zero.
- Modify an element from a non-zero to another non-zero.

The following subsections investigate the complexity of such updates for different classes of functions.

A. Caveats

Not included here is the cost of checking if some specific entry in an object is a structural zero. For some sparse formats such as CSR this may require a partial linear search through part of a list. For other formats such as using a hash table, this may require multiple table probes. Also not included is modifying a data structure so that some structural zero becomes a non-zero, or vice versa. In addition, not included is the possible cost of addition or deletion of a vertex to a graph. Such modifications change the size of a data structure,

but until edges are added, virtually none of the GraphBLAS operations would be affected.

B. Element-by-Element Operations

The complexity of an update to an element in a matrix or vector that is then involved in an element-by-element operation (such as $w \circ x$ where both are vectors or matrices) is obvious: if either the modified element or the matching element that it is to be combined with is a structural zero, then the new value is also a structural zero. In any case the new value is a single computation of $O(1)$.

C. Reduction Operations

TABLE IV
COMPLEXITY OF UPDATES IN A REDUCTION

.

Type	A & C?	Inverse ?	Selector?	Complexity	Computation
Add	Y			$O(1)$	$s \leftarrow s \circ u$
Add	N			O_{nnz}	$s \leftarrow \circ/w$
Delete	Y	Y		$O(1)$	$s \leftarrow s \oplus w[i]^{\oplus}$
Delete	N			O_{nnz}	$s \leftarrow \circ/w$
Modify	Y	Y		$O(1)$	$s \leftarrow s \oplus w[i]^{\oplus} \oplus u$
Modify	N			O_{nnz}	$s \leftarrow \circ/w$
Delete	Y	N	Y	b!=s: $O(1)$ else O_{nnz}	No change $s \leftarrow \circ/u$
Modify	Y	N	Y	$\oplus(s,c) = c$ $\oplus(s,c) = s$ and $b \neq s$ else O_{nnz}	$s \leftarrow c$ No change $s \leftarrow \circ/w$
"A&C" stands for "is $\oplus$ associative & commutative" s is the result of the reduction. O_{nnz} is the same as $O(nnz(w))$ b is value of element being deleted or modified. For modify, c is value after the element is updated.					

The next level of operation in a GraphBLAS program is a reduction of the form $s \leftarrow \oplus/w$ where w is either a vector or a matrix[6], and s is a scalar. Here changing one element, say $w[i]$, cannot be considered in isolation, as the operation involves all the other elements.

If $\oplus$ is associative and commutative, then the calculation can be re-arranged so that $w[i]$ is the "last" element in the reduction. Thus if the update is going from a structural zero to a non-zero u, then the update need only perform $s \leftarrow s \oplus u$, with complexity $O(1)$. If $\oplus$ has inverses, then deleting a value (making it a zero) can be accounted for by $s \leftarrow s \oplus w[i]^{\oplus}$. Modifying the non-zero $w[i]$ to another non-zero u is then similar: $s \leftarrow s \oplus w[i]^{\oplus} \oplus u$. Both cases are also are $O(1)$.

If $\oplus$ is not both associative and commutative, then the combination cannot be re-arranged, and re-computing the entire expression may be the only option, with u replacing $w[i]$. This is at least $O(nnz(w))$. Table IV summarizes these cases. The first column indicates the kind of update and the next three the properties of $\oplus$. Next is the complexity of the update. The last is the computation sequence.

[6]By convention, if w is an NxM matrix, $+/w$ sums over the second dimension, yielding an Nx1 element column vector.

If $\oplus$ is not associative and commutative, then we cannot rearrange the computation, and the whole operation must be repeated to guarantee an in-order evaluation.

D. Reduction Special Case using Selectors

In a reduction, for the deletion or modification of a term when there is not an inverse, we cannot do an $O(1)$ update, and notionally have to recompute the whole expression. However, there are sometimes some simple tests that can avoid a complete recompute. Consider for example $\oplus = min$ as in $min/(5,3,8)$. If the update is the deletion of the argument 3, there is no way to recover the 5 without starting over. If, however, it is the 5 being deleted, comparing the 5 to the current minimum 3 indicates that it would not be the answer even if we recompute, and thus no re-evaluation is needed. Likewise if the 5 is being modified to a 4, then again this would have no effect on the answer and no re-evaluation needed. However, if the 5 is modified to a 2, with no other changes to the other terms in the reduction, then since $min(3,2) = 2$ we can change the result to 2, again without a recompute.

Other functions like max have the same property. While it is unclear what is the most general class of such functions, we can define a class of functions, called here *selectors*, that have the following properties:

- $\oplus(a,b) =$ either a or b for all as and bs.
- $\oplus$ has a $I_{\oplus}$ such that $\oplus(a, I_{\oplus}) = \oplus(I_{\oplus}, a) = a$

The first property guarantees that the result of a reduction is always one of the elements being reduced. If $\oplus$ is also associative and commutative, then the elements could be shuffled in any order without affecting the answer. Thus, if the update is a deletion of one of the elements (say b), then as long as $b \neq s$, we are guaranteed that its presence or absence in the reduction had no effect on the result s, and thus no additional computation is needed. If $b = s$, then it is possible that the b element was in fact the term that drove the computation, and thus the reduction must be redone.

If the element with value b is modified to a value c, similar reasoning is possible. If $\oplus(s,c) = c$ then we know that the new value c takes precedence and is the new s. Likewise if $\oplus(s,c) = s$ and $b \neq s$, then we know the term providing the original reduction is still there and is still the answer. Only if neither condition holds must we recompute.

Table IV includes these cases.

E. Inner Products

The inner product $s \leftarrow w \oplus . \otimes x$, where w and x are vectors, is the workhorse of GraphBLAS. If a structural zero in w (or x) is an $\otimes$ annihilator $0_{\otimes}$, then $0_{\otimes} \otimes a = 0_{\otimes} = I_{\oplus}$ for any a, and the $\otimes$ product may be skipped. If $\oplus$ is both associative and commutative, then a change in $w[i]$ from a structural zero to a non-zero u can be computed simply by $s \leftarrow s \oplus (u \otimes x[i])$, an $O(1)$ computation. If in addition, the elements of $D_{\oplus}$ all have inverses under $\oplus$, then changing $w[i] \otimes x[i]$ to a structural zero can be computed as $s \leftarrow s \oplus (w[i] \otimes x[i])^{\oplus})$, and modifying it to another non-zero u by $s \leftarrow s \oplus (w[i] \otimes x[i])^{\oplus}) \oplus (u \otimes x[i])$, Again these are all

$O(1)$ in complexity. If not all of these conditions are true, the special case mentioned in Section IV-D may be applicable. If not, the entire inner product needs to be recomputed from scratch, $O(max(nnz(w), nnz(x)))$. The complexity follows Table IV, except that O_{nnz} is $O(max(nnz(w), nnz(x)))$ with computation $w[i] \otimes x[i]$.

F. Handling Options

Most of the options to GraphBLAS calls also need to be considered when computing complexity. If an implementation akin to Section III-G is in place, then the time complexity for an assignment is at worst proportional to the number of non-zeros being generated by the expression, and the complexities discussed above are still valid.

G. Number of Generated Changes

TABLE V
MAXIMUM CHANGE COUNT GENERATED IN $C = A \oplus . \otimes B$.

Graph	Updates	# of Changes	Changes
D	$A[i,j]$	R	$C[i;]$
D	$B[i,j]$	M	$C[;j]$
U	$A[i,j], A[j.i]$	$2R$	$C[i;], C[j;]$
U	$B[i,j], B[j,i]$	$2M$	$C[;j], C[;i]$
D;$B = A$	$A[i,j], B[i,j]$	$2N$	$C[i;], C[;j]$
D;$B = A^T$	$A[i,j], B[j,i]$	$2N$	$C[i;], C[;i]$
U;$B = A$	$A[i,j], A[j,i]$ $B[i,j], B[j,i]$	$4N$	$C[i;], C[j;]$ $C[;j], C[;i]$
U;$B = A^T$	$A[i,j], A[j,i]$ $B[j,i], B[i,j]$	$4N$	$C[i;], C[j;]$ $C[;j], C[;i]$
"U" = Undirected graph; "D" = Directed graph. A is MxN; B is NxR; C is MxR			

A single update to an object in an expression involving only an element-by-element operation generates at most one outgoing change that must be used for an update to a later operation in program order. This is also true for a reduction (there is only the scalar sum value as output).

However for inner products there may be more than one change generated by a single update applied to an argument. Table V summarizes the possible changes that might be generated by $A \oplus . \otimes B$. The first four cases are when the updated edge is in only one of the two matrices A or B. In these cases distinction is made between directed graphs and undirected where the edge affects two symmetric elements $[i,j]$ and $[j,i]$. For the latter cases both a row and column of C are affected.

The next four cases cover cases where B is either A or A^T, and the edge update thus modified both matrices. For the directed case, there is still a row and column affected, but for the undirected cases, two rows and columns are affected.

Regardless of whether the complexity of each update is $O(1)$ or more, the complexity of handling all these changes, even in the most complex cases, is still considerably less than Table III, making streaming a more efficient computation than recomputing in batch mode.

H. Avoiding Redundant Computations

Table V indicates that in many cases, an inner product expression like $A + .xB$, can create multiple changes from the update of a single edge. In most cases these can be computed independently (and thus concurrently), but consider when $B = A$; a single update can affect both a row and a column of C. If the computation of the row and column elements are done blindly, then the update of the common element $C[i,j]$ at the intersection is done twice. For most of the cases of $\oplus$ of Table V this involves modifying the $C[i,j]$ element with the same value twice. In other cases such as updating an edge in an matrix used in an expression like $A \oplus . \otimes A$ there are typically four such common intersections. Cases where $i = j$ may also result in updating a whole row or column twice. Clearly this must be avoided either by explicit programming for each case or by keeping dynamic track of elements that have been updated and checking before each new update is attempted. For GraphBLAS, at least some of the latter must be done anyway to handle cases where the destination of the inner product includes a mask.

I. Multiple Updates

Most of the early discussion here assumed a single update to some element in an object. Clearly a real application may want to batch several updates at once, as was assumed for the Stinger benchmarks [20]. Even if that is not the case, undirected edges may introduce two updates to an adjacency matrix, and in many case as discussed above, one update may generate multiple changes to a computed object, which if used in a subsequent computation, results in the equivalent of multiple concurrent updates.

Handling such multiple concurrent updates requires special considerations. First, the object(s) receiving the changes must all be updated before attempting any of the next round of computations; otherwise some of the computations that reference two elements that are both updated may be invalid. Second, a record must be kept as each change is computed to prevent duplicate computations as discussed in Section IV-H.

V. EXAMPLES

The following subsections give examples of graph algorithms, what might need to be done to them to adapt to streaming mode, and what is the difference in time complexity between a batch and streaming mode. We assume an initial "batch" operation creates the initial data set that is then modified by the updates. In most cases, the same sequence of GraphBLAS calls can specify both, albeit with a different compilation for the update versus batch.

In all examples, the pseudocode suppresses much detail of a full GraphBLAS code without hopefully missing anything major. Explicit "export" and "import" lines reference what data must cross between the main space and GraphBLAS' opaque space. Calls that result in computations in opaque space are grouped between "Begin" and "End GraphBLAS" lines. The notation for individual method calls uses the notation

from Section III-C. For brevity, only "add edge" codes are discussed; delete and modify typically look similar.

No claim to optimality is made for any of these algorithms, only that they appear correct and tell us something about the ability of the above techniques to relatively mechanically produce a correct streaming code.

A. Triangle Counting

A *triangle* in an undirected graph is a set of three vertices (u, w, v) with edges between each of them. For uniqueness, we also assume all such triangles have $u > w > v$, with no self-loops. Counting such triangles in a graph is well-studied, with low level algorithms published for streaming [5].

Algorithm 1 is abstracted from Example B.6 in [9] (see also [7], [19], [21]) for the counting of the number of unique triangles in a graph. The code assumes the adjacency matrix A has been reduced to its lower diagonal form L where for all $w \geq u,\ L[u, w] = 0$, and for all $w < u,\ L[u, w] = A[u, w]$. Since A is undirected, no information is lost since $A[u, w] = A[w, u]$, and we are assumimg no self-loops.

Algorithm 1 Triangle Counting from GraphBLAS spec
A is nxn adjacency matrix for graph
L is is the lower diagonal form of A

1: Export L into opaque space
2: Begin GraphBLAS calls; Execute in opaque space
3: $\quad C[L] \leftarrow L + . * L^T$;
4: $\quad count \leftarrow +/ + /C$
5: End GraphBLAS calls
6: Import count from opaque state

Line 3 computes an intermediate matrix $L + . * L^T$ where entry $[u, w]$ is the number of unique *wedges* (u, w, v) where there is an edge between u and some v, and another edge between w and the same v, where $u > v$ and $w > v$. "$C[L] \leftarrow$" stores this count into $C[u, w]$ only if $L[u, w]$ is true, i.e. there is also an edge between u and w where $u > w$, completing the triangles where $u > w > v$. Line 4 then sums all these counts into a single number. Assuming n vertices and $|E|$ edges, Line 3's batch complexity is at worst $O(n^2|E|/|V|) = O(n|E|)$. Line 4 is $O(nnz(L + . * L^T))$ or at worst $O(n^2)$. If $|E|$ is typically some multiple of n, the two are approximately equal in complexity.

Algorithm 2 Streaming Triangle Counting
Update is edge (u, w), assume $u > w$

1: **Add Edge**(u, w):
2: $L[u, w] \leftarrow True$
3: **for** all v where $L[v, w]$ is not zero **do**
4: $\quad C[u, v] += (x = L[u, w] * L[v, w])$;
5: $\quad count += x$;
6: **for** all v where $L[v, u]$ is not zero, and $v! = u$ **do**
7: $\quad C[v, u] += (x = L[v, w] * L[u, w])$;
8: $\quad count += x$;

Algorithm 2 gives the pseudocode for a derivation of a streaming code using the above techniques. For this application, L is lower triangular and thus is a directed graph, and $+$ and $*$ are both associative and commutative. Only the "Add edge" case is listed here for brevity. Line 3 performs the update into L. The right-hand side of Line 3 of Algorithm 1 corresponds to the sixth case of Table V, and results in at worst $O(n)$ - the update of the u'th row (For loop at line 4) and u'th column (For loop at line 9) of the product, while avoiding the second computation of the $[u, u]$'th[7] element. The "where" part of each loop performs the mask test.

Again, Algorithm 2 represents the code that may be derived in a semi-automatic fashion when we are asked to create a streaming edge addition version of Algorithm 1.

B. Breadth First Search

Breadth First Search (BFS) is the first graph kernel reported in the Graph500[8] website. Starting at some root vertex s, the algorithm follows all outgoing edges and labels all reachable vertices with their distance from s as an edge count. This repeats until there are no more unvisited vertices[9].

Example B.1 of [9] provides a GraphBLAS batch implementation that is functionally close to the reference implementation given on the Graph500 website. This code is a loop that updates two vectors: $v[i]$ is the "level" of vertex i from vertex "s," $q[i]$ is a boolean that indicates that vertex i is a previously untouched vertex that is now reachable from one of the most recent additions to v (i.e. the "frontier" of the search). The code is fairly complex, with focus of computation only on the "frontier." This code (especially with Beamer's additions [6]) is fine for a batch computation, with a complexity of $O(|E|)$. However, it does not provide a mechanism to "reset" a previously assigned level when a new added edge provides a shorter path to some previously reachable vertex, and thus is not a good algorithm to use for streaming.

Algorithm 3 Alternative BFS for streaming updates
Assume v_{init} is a staring point for BFS over A from some starting point s.
$\otimes(a, b)$ = 0 if a or b is 0, and $a + b$ otherwise.
$\oplus(a, b)$ is $max(a, b)$ if either a or b is 0, $min(a, b)$ otherwise

1: Export v_{init} and A into opaque space
2: **repeat**
3: $\quad$ Export updates to A in opaque space
4: $\quad$ Begin GraphBLAS calls
5: $\quad$ **repeat** $v \leftarrow v \oplus . \otimes A$
6: $\quad$ **until** No more changes in v
7: $\quad$ End GraphBLAS calls
8: **until** No more updates

An alternative starts with the assumption that a solution row vector v has the property $v[i]$ is 0 if vertex i is unreachable,

[7]Not possible give the nature of L but compiled here to match the rules.
[8]https://graph500.org/
[9]A more advanced algorithm [6] used in most current implementations also goes backwards from the unvisited vertices at certain points.

and the level of vertex i otherwise[10]. We observe such a vector satisfies the equation $v = v \oplus . \otimes A$ where $\oplus$ and $\otimes$ are defined in Algorithm 3[11] , When there is an edge between vertex i and j, $v[i] \otimes A[i,j]$ (as part of computing $v[j]$) returns 0 if if $v[i]$ or $A[i,j]$ is 0, (implying either vertex i is unreachable, or there is no path from i to j), and $v[i] + A[i,j]$ if both $v[i]$ and $A[i,j]$ are not 0 (i.e. j is one edge removed from vertex i, and thus vertex j can be no further from root than $v[i] + 1$). The reduction operator $\oplus$ then returns for $v[j]$ the minimum level to reach $v[j]$ from any $v[i]$, if any of them are non-zero, and 0 otherwise. $\oplus$ here is both associative and commutative, with $0 = 0_{\oplus}$.

The initial v for Algorithm 3 is notionally either a vector where only $v[s]$ is non-zero or a full solution computed by an optimal batch algorithm. In either case the loop at line 5 will advance the frontier one level at a time until a solution is reached by changing zeros in v to non-zeros.

Not described in Algorithm 3 is how the detection of no more changes in v is done. We could do a swap between two v vectors, and compare old vs new, but that is expensive when the results from a single update are liable to be small in number. Perhaps a better approach is to redefine $\leftarrow$ for GraphBLAS to include an option to have as a side-effect a count of the number of changes between left and right sides.

When a streaming code is built from this, addition of a single edge to A converts two entries (if A is undirected) from structural zeros to non-zero. From the 4th entry in Table V, each of these affect at most one element of v. Since $\oplus$ is associative and commutative, the complexity for at least the first iteration is $O(1)$. Each of these changes may change up to $|V|$ entries (by making some vertices "closer"). The worst case if all vertices become closer is $O(|V|)$.

However values in the $\oplus$ monoid do not have an inverse, so for deleting and modifying an edge in A, the special case as described in Section IV-D is applicable, with a resulting complexity of either $O(1)$ or $O(max(nnz(w), nnz(x)))$ for the first pass.

C. Jaccard

The *Jaccard Coefficient* for any pair of vertices (u, v) in a graph is defined in terms of the similarity of the neighborhoods of u and v. The neighborhood of a vertex u is $N(u) = \{v|(u,v) \text{ in } E\}$, i.e. all vertices reachable via one edge. We formally define a Jaccard coefficient as $Jaccard(u,v) = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|}$. This is the ratio of all neighbors in common to all neighbors.

Algorithm 4 defines a batch algorithm to compute all the Jaccard coefficients of a graph. Line 3 computes an intermediate matrix C, where $C[u,w]$ is the size of the intersection of the corresponding neighborhoods of vertices u and w. Line 4 computes a vector N that contains the size of the neighborhoods for each vertex. Next, line 5 computes the size of the union of each pair of vertices. The outer product $N.+N$ computes a matrix where the $[i,j]$th entry is $N[i] + N[j]$. The last portion of this step subtracts C (which contains the size of the intersection). This removes one copy of the shared vertices of the neighborhoods and results in the size of the union. Line 6 combines the intersection and union values to give the Jaccard coefficients for each pair of vertices.

Algorithm 4 Batch Jaccard Algorithm
A is nxn adjacency matrix for graph

1: Export A into opaque space
2: Begin GraphBLAS calls; Execute in opaque space
3: $\quad C \leftarrow A + . * A^T$
4: $\quad N \leftarrow +/C$
5: $\quad Q \leftarrow (N. + N) - C$
6: $\quad Jaccard \leftarrow C/Q$
7: End GraphBLAS calls
8: Import count from opaque state

Note that this algorithm computes all $Jaccard[u,w]$ values, even though $Jaccard[u,w] = Jaccard[w,u]$. A real-world algorithm would attempt to avoid these duplicate computations, but that would not affect the "$O()$" time.

For dense graphs the complexity of Algorithm 4 is dominated by the $A + . * A^T$, which is $O(n^3)$. The other steps are $O(n^2)$.

For sparse graphs, line 3 is just a straightforward sparse matrix multiply $O(n^2(nnz_{row}(A)))$. Then to compute the N vector the complexity is $O(nnz(A))$. Computing $N. + N$ is $O(n^2)$, but to subtract the C matrix we perform $O(nnz(C)$. Finally to compute C/Q we have a complexity of $O(nnz(C))$. Again the complexity of the matrix multiply dominates.

Algorithm 5 uses the above techniques to outline an algorithm that streams in new edges. Assuming an undirected graph, according to Table V adding one edge affects 2 rows and 2 columns of C. Further, these sub-vectors are themselves sparse, so the number of updates for each is $O(nnz_{row}(A))$. Further, because $\otimes$ and $\oplus$ in this case are well-behaved, each update is $O(1)$. Thus the complexity of the matrix multiply update drops to $O(nnz_{row}(A))$ per new edge. The loops at lines 4 and 9 include these updates.

The updates to Q that involve the subtraction of C can be done at the same time as the updates to C, so their updates are fused into the above two loops. Again each update is $O(1)$.

The update to the vector N that contains the size of the neighborhoods is $O(1)$, because we are updating exactly two values in N, the size of the neighborhoods of u and w (line 14). Then when recomputing Q we perform $O(n)$ by updating all the elements in each row and column for each vertex. Then to finally update the resulting $Jaccard$ matrix we have to iterate through the updated values in Q and C, which is $O(nnz_{row}(C))$. The overall complexity of a single update is only $O(n)$, a huge savings over a complete recompute.

D. Bipartite Matching

A *bipartite graph* $G = ((V_L, V_R), E)$ consists of two disjoint sets of vertices V_L and V_R, as well an edge set

[10]We assume the starting root vertex is at level 1.

[11]Thus v is akin to an eigenvector for A but with strange functions.

Algorithm 5 Streaming Jaccard
Update is edge (u, w), assume $u > w$

1: **Add Edge**(u, w):
2: $A[u, w] \leftarrow True$
3: $A[w, u] \leftarrow True$
4: **for** all v where $A[v, w]$ is not zero **do**
5: $C[w, v] += (A[u, w] * A[v, w])$;
6: $Q[w, v] -= (A[u, w] * A[v, w])$;
7: $C[v, w] += (A[w, u] * A[w, v])$;
8: $Q[v, w] -= (A[w, u] * A[w, v])$;
9: **for** all v where $A[v, u]$ is not zero, and $v! = u$ **do**
10: $C[v, u] += (A[u, w] * A[v, u])$;
11: $Q[v, u] -= (A[u, w] * A[v, u])$;
12: $C[u, v] += (A[w, u] * A[u, v])$;
13: $Q[u, v] -= (A[w, u] * A[u, v])$;
14: N[u]++; N[w]++
15: **for** v=1 to $|V|$ **do**
16: $Q[v, u] += 1$;
17: $Q[u, v] += 1$;
18: $Q[v, w] += 1$;
19: $Q[w, v] += 1$;
20: **for** all v where $C[v, u]$ is not zero, and $v! = u$ **do**
21: $Jaccard[v, u] += C[v, u]/Q[v, u]$
22: $Jaccard[u, v] += C[u, v]/Q[u, v]$
23: **for** all v where $C[v, w]$ is not zero, and $v! = u$ **do**
24: $Jaccard[v, w] += C[v, w]/Q[v, w]$
25: $Jaccard[w, v] += C[w, v]/Q[w, v]$

$E = \{(u, v)\}$ in which all u vertices are in V_L and all v vertices are in V_R. A valid *bipartite matching* M is a subset of E such that no vertex is incident to more than one edge in M. A matching for an arbitrary graph is, in general, not unique as many permutations of edge sets may exist. A **maximum cardinality matching** is one where there is no matching that covers more vertices by count.

Algorithm 6 gives the pseudocode for a greedy batch bipartite matching. Line 6 performs a matrix-vector operation that returns for each vertex w in V_R the smallest unmatched $U_R[u] = u$ where there is an edge (u, w). Logically ANDing this with U_R zeros out any vertex w in V_R where w is already matched. This implementation constitutes a *similar* heterogeneous algebra to that of $\oplus.\otimes$ [8].

Line 7 prunes F to remove duplicate matches based on some criteria, such as minimum degree, as in the Karp-Sipser heuristic [15]. After pruning, remaining entries indicate new matchings that are inserted into M on line 8. Matched vertices are removed from U and the loop continues until all valid matchings are found.

The matrix-vector operation in Line 6 constitutes the greatest computational work per loop iteration. If it were implemented using a sparse vector, initially dense and becoming increasingly sparse with every iteration, time complexity for each sparse matrix sparse vector multiplication approaches $\Omega(rc)$ where r = rows in iteration and $c = nnz_{col}$ such that

Algorithm 6 Greedy Bipartite Matching
A is $|V_L|$ x V_R adjacency matrix for the graph.
M is list of all edges (u, w) in the matching.
U_L is "unmatched left vertex" sparse vector where $U_L[u] = u$ if the u'th vertex in V_L is unmatched, and 0 if matched.
U_R is "unmatched right vertex" sparse vector where $U_R[w] =$ 1 if vertex w in V_R is unmatched, and 0 if matched.
F is unmatched vertex frontier sparse vector, where $F[w]$, w in V_R is smallest $U[k]$ where $A[k, w] = 1$ and $U[k] \neq 0$.
$\otimes(a, b)$ is integer multiply
$\oplus(a, b) = min(a, b)$

1: Export A and M into opaque space
2: Initialize U_L and U_R vector so all vertices unmatched.
3: Initialize M to empty set.
4: Begin GraphBLAS calls;
5: **repeat**
6: $F \leftarrow U_R \wedge (A^T \oplus . \otimes U_L)$
7: $F' \leftarrow$ *prune* F
8: $M \leftarrow M \cup \{(i, F'[u])\ |F'[u] \neq 0)\}$
9: update U_L, U_R
10: **until** no more valid matchings
11: End GraphBLAS calls
12: Import matching M from opaque space

$r << |V_L|$ and $c << |V_R|$ as iteration count increases [2].

When we modify this to accept a single edge addition, A has exactly one new entry. This causes exactly one change to the output of the product. Given that $\oplus$ is min, the complexity of this update is either $O(1)$ or $O(nnz_{col}(A))$ (see Section IV-D). The logical AND is thus itself $O(1)$. The prune on line 7 depends on the algorithm used, but is likely to be at worst $O(nnz(F'))$. The other steps are liable to be $O(1)$.

The above discussion is for the first iteration of the loop. If changes were made, another iteration is performed, now with U_L having at least one less 0. The worst case is $O(V_L)$ but in practice this is probably an extremely rare case, as a maximal matching has now been found. In any case, this is far better than a full batch recompute.

VI. Concurrency

Introducing parallelism into the above computations would be the next level of enhancement into the streaming process. Starting with a single update, Table V provides guidelines for how to increase concurrency by processing individual elements of rows or columns independently, as long as the double update issue of Section IV-H is handled. One difficulty is in saving the multiple changes into the target data structures. The Stinger work [4] discusses some lock-based algorithms to do such concurrent inserts at high efficiency into a Stinger-supported adjacency matrix. Another issue deals with GraphBLAS codes that have multiple steps in their program graphs and/or a series of fused operations in a single step. A change resulting from one operation looks like an update to an input object for some other later operation. If the code is written in a declarative, single-assignment style where there is no feedback, then the

chain of operations can be "pipelined" so that once one operation has processed all changes presented to it from one update, the resulting changes can be passed on, and then another round of computation resulting from a next update could be started.

More complex is when multiple updates are presented at once. If these come from changes generated by a prior step, then these updates are independent of each other, and the big issue is deconflicting changes that come through the steps (similar to that discussed on Section V).

The most complex case is when there are multiple concurrent streams of incoming updates. Here it is important to ensure a proper ordering to updates that affect the same element. A "delete, add, update" can give far different results that a " update, add, delete" to the same element. Assuming that incoming updates have time tags, then algorithms such as defined in [5] or [12] may be relevant, as might variations of the *time-warp* algorithm [14]. The latter algorithm includes mechanisms to "back-track" when an update turned out to be done out of order. The existence of inverses for the functions used in GraphBLAS operations would be beneficial for implementing such backups.

VII. Conclusions

This paper has taken the extended linear algebra of GraphBLAS and suggested one way to adapt it to develop code in a possibly automated fashion that efficiently processes incremental updates, rather than simply recompute the entire problem from scratch. Special attention has been given to the treatment of "structural non-zeros," and in differentiating between three different kinds of updates: converting a previous zero to non-zero, converting a previous non-zero to a zero, and modifying one non-zero to a different non-zero. Streaming versions of several graph algorithms demonstrate the approach.

While the chosen domain is that of graphs, any problem domain that can be cast in linear algebra is also relevant. Depending on the properties of the functions involved in a linear algebra expression, in many cases updating single elements can be performed in $O(1)$ time.

In addition, several important topics will be explored in future work. First is some prototype implementations to explore real-world performance. Next is how best to augment GraphBLAS' syntax, semantics, and API to allow extensions that allow explicit streaming codes to be written. Additional work would go one step further, and in some sense automate, the compilation of a streaming code using the extended syntax from an initial batch code. Finally, a formalization of the rules for performing multiple updates in parallel is needed.

Acknowledgments

This material is based upon work supported in part by the Univ. of Notre Dame, and in part by the National Science Foundation under Grants CCF-1642280 and CCF-1822939. Also, an exchange with Prof. Tim Davis of Texas A&M (developer of SuiteSparse GraphBLAS[12]) provided tremendous food for thought for the next step of this work.

References

[1] T. A. Davis. Graph algorithms via suitesparse: Graphblas: triangle counting and k-truss. pages 1–6, 09 2018.

[2] A. Azad and A. Buluç. A work-efficient parallel sparse matrix-sparse vector multiplication algorithm. *CoRR*, abs/1610.07902, 2016.

[3] A. Azad and A. Bulu. Distributed-memory algorithms for maximum cardinality matching in bipartite graphs. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 32–42, May 2016.

[4] D. A. Bader, J. Berry, A. Amos-Binks, D. Chavarra-Miranda, C. Hastings, K. Madduri, and S. C. Poulos. Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation. *Georgia Institute of Technology, Tech. Rep*, 2009.

[5] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 623–632, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[6] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[7] 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*, KDD '08, pages 16–24, New York, NY, USA, 2008. ACM.

[8] G. Birkhoff and J. D. Lipson. Heterogeneous algebras. *Journal of Combinatorial Theory*, 8(1):115 – 133, 1970.

[9] A. Buluc, T. Mattson, S. McMillan, J. e Moreira, and C. Yangce. The graphblas c api specification, version 1.2. https://people.eecs.berkeley.edu/ aydin/GraphBLAS_API_C.pdf.

[10] T. A. Davis. Algorithm 9xx: Suitesparse:graphblas: graph algorithms in the language of sparse linear algebra. url-http://faculty.cse.tamu.edu/davis/GraphBLAS_files/toms_graphblas.pdf.

[11] D. Ediger, K. Jiang, J. Riedy, and D. Bader. Massive streaming data analytics: A case study with clustering coefficients. pages 1 – 8, 05 2010.

[12] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2):207–216, Dec. 2005.

[13] K. E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., New York, NY, USA, 1962.

[14] D. Jefferson and H. A. Sowizral. Fast concurrent simulation using the time warp mechanism part 1: Local control. https://www.rand.org/pubs/notes/N1906.html.

[15] R. M. Karp and M. Sipser. Maximum matching in sparse random graphs. In *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)*, pages 364–375, Oct 1981.

[16] J. Kepner, P. Aaltonen, D. A. Bader, A. Buluç, F. Franchetti, J. R. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, J. E. Moreira, J. D. Owens, C. Yang, M. Zalewski, and T. G. Mattson. Mathematical foundations of the graphblas. *CoRR*, abs/1606.05790, 2016.

[17] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, 2011.

[18] A. McGregor. Graph stream algorithms: A survey. *SIGMOD Rec.*, 43(1):9–20, May 2014.

[19] R. Pearce. Triangle counting for scale-free graphs at scale in distributed memory. In *IEEE High Performance extreme Computing Conf.*, pages 1–4, 09 2017.

[20] J. Riedy and D. Bader. Stinger: Multi-threaded graph streaming. 05 2014.

[21] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam. Fast linear algebra-based triangle counting with kokkos kernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, Sept 2017.

[12] http://faculty.cse.tamu.edu/davis/GraphBLAS.html