# Indexed Binary operations in the GraphBLAS

Timothy G. Mattson*, Manaswinee Bezbaruah†, Matthias Maier†, Scott McMillan‡
Michel Peletier¶, Erik Welch§, Timothy A. Davis†

*Human Learning Group †Texas A&M University ¶OneSparse §Nvidia
‡Software Engineering Institute, Carnegie Mellon University

***Abstract*—GraphBLAS is a sparse matrix library for graph algorithms. We could improve the GraphBLAS and support a wider range of applications if we had an *indexed binary operator*; i.e., a binary operator that depends on elements from two matrices and their positions (indices). We propose a design for these operators and illustrate their impact on performance and expressiveness for several algorithms including breadth-first-search, argmin/argmax, vector search, and finite-element assembly.**

***Index Terms*—Graph Processing, Graph Algorithms, Graph Analytics, Linear Algebra, GraphBLAS**

## I. Introduction

A graph can be represented by an array. For example, an $N \times N$ adjacency matrix, $\mathbf{A}$, has one row and one column for each of the $N$ vertices in a graph. Element $\mathbf{A}_{ij}$ of the matrix represents an edge between vertex $i$ and vertex $j$. With graphs as matrices, a wide range of graph algorithms can be expressed with linear algebra [12]. With graph algorithms as linear algebra over arrays (almost always sparse), it is natural to define a set of Basic Linear Algebra Subprograms (BLAS) to support such algorithms. We call these the GraphBLAS.

To make this work, we need a mathematically rigorous way to change the operators used inside the linear algebra operations. For example, in many graph algorithms, as you work across the out-edges from a vertex (i.e., a row in the adjacency matrix), you want to keep the edge with the "greatest" weight as opposed to summing across weights. This means reducing the results of operations on the elements in a row with a *max* operation instead of *sum*. We modify operators inside our linear algebra functions through *algebraic semirings*.

The detailed mathematics of semirings is well beyond the scope of this paper. For our purposes, think of a semiring as a domain (the types of the elements in the arrays), a *monoid* (a commutative and associative binary operator with an identity, denoted $\oplus$) and a second binary operator (denoted $\otimes$). The prototypical semiring is defined over real numbers with the $\oplus$ monoid being addition (with zero as the identity) and the $\otimes$ operator being multiplication. Since these operators can be swapped with others in different semirings, we use a "circle op" notation so matrix multiplication would be written as:

$$\mathbf{C} = \mathbf{A} \oplus.\otimes \mathbf{B}.$$

This notation impacts how we talk about semirings. When using different semirings, the $\oplus$ monoid is typically referred to as the *plus* operator and the $\otimes$ operator as the *times* operator, even when addition and multiplication over real numbers has been replaced by other operators. For example, in the famous *tropical semiring* the $\oplus$ is the $min$ operator and the $\otimes$ operator is addition ($+$) resulting in the $min.plus$ semiring.

The GraphBLAS project began in 2013 [14], the official mathematics specification was released in 2016 [11], and the first formal language binding (to C) was released in 2017 [6]. The GraphBLAS is a "living" specification under continuous development by the GraphBLAS forum [1] (with the latest C specification being version 2.1 [5]).

Recently, we have been exploring the use of operators that have access to the indices of the array elements. These are referred to as *indexed operators*. In GraphBLAS version 2.1 we added *indexed unary operators*; i.e., an operator that acts on a single element of a graphBLAS array and has access to the array indices of the single operand. In this paper, we extend this concept to binary operators, that is, indexed *binary* operators.

This paper makes three contributions. First, we describe the indexed binary operator and how it might be incorporated into the GraphBLAS. This is important for the GraphBLAS algorithm community to understand so they can provide feedback and guide development of this capability. Next, for API designers more generally, our design of indexed binary operators is a good example of the value of opacity in an API. In particular, since GraphBLAS objects (including operators) are opaque, indexed binary operators can be added to GraphBLAS with minimal disruption to the other functions in the API. This is a lesson in the value of opacity that we urge other API designers to consider. Finally, we describe use-cases that establish the need for indexed binary operators. It is well known that the GraphBLAS can be used for graph algorithms, graph databases [7], neural networks [9], [16], cyber security [10] and other problems that "look like graphs". As we will show in this paper, with the indexed binary operator, we can address new classes of problems including vector databases and finite element methods.

## II. GraphBLAS: Math, Notation, and API

Mathematics uses matrices, vectors and operations. An API defines objects (with storage formats) and specifies the signatures of functions that act on them. These details are defined in the GraphBLAS API and explained in the paper that introduced the GraphBLAS binding to C [6]. An API is designed around its use to construct key algorithms.

GraphBLAS algorithms are the focus of the LAGraph [13], [2] projecct which includes production-ready algorithms for application programmers and experimental algorithms that explore what can be done with the GraphBLAS. A core set of algorithms and a notation for expressing them is defined here [15]. In this section, we describe the GraphBLAS at a high level and the notation used to express the algorithms.

In describing GraphBLAS algorithms the matrices and vectors are sometimes defined in terms of vectors of row indices or column indices or as *tuples* which contain the set of indices and the values of a matrix or vector. Matrices are represented as uppercase, bold letters ($\mathbf{A}$), and vectors as lowercase bold letters ($\mathbf{u}$). GraphBLAS objects are *opaque*, meaning their implementation is not defined in the GraphBLAS specification. This gives implementors the freedom to choose storage formats and other details best suited to the target system. This opacity is a key feature of the GraphBLAS design.

The GraphBLAS operations act on objects that hold matrices, vectors, or scalars. The fundamental GraphBLAS *operations* include:

- matrix multiplication (mxm), matrix vector multiplication (mxv) and vector matrix multiplication (vxm),
- kronecker product of two matrices using the $\otimes$ operator,
- element-wise operations using the $\oplus$ (eWiseAdd) or the $\otimes$ (eWiseMult) from the semiring $\oplus.\otimes$,
- assign and extract submatrices, subvectors or individual rows and columns of a matrix,
- apply a function to the elements of an object,
- select elements of an object using a function (predicate),
- matrix transpose.

These operations also support optional *mask* and *accumulation* operations. The notation used for matrix multiplication including these options is:

$$\mathbf{C}\langle\mathbf{M}\rangle \odot= \mathbf{A} \oplus.\otimes \mathbf{B}$$

The result of this matrix product will be written into the output object $\mathbf{C}$ subject to accumulation and masking.

Accumulation applies a binary operator (denoted as $\odot$) in an elementWise addition between the result of the matrix product and the values already in the destination matrix $\mathbf{C}$. The mask ($\mathbf{M}$) is selects elements of the destination matrix, $\mathbf{C}$, that will be overwritten with the results from the GraphBLAS operation. It also defines how elements *not* selected by the mask are impacted by the operation. Although it is logically a "write" mask, a high-quality implementation will use the mask to drive which right-hand side computations are carried out.

The mask has the same number of rows and columns as the output object ($\mathbf{C}$ in our example). By default, the elements of the mask that exist and are non-zero correspond to output elements that are overwritten (written as $\langle\mathbf{M}\rangle$). There are two additional options for defining the elements in the mask.

1) The mask is defined by the elements of $\langle\mathbf{M}\rangle$ that exist *regardless* of their values. This is called a *structural* mask and in our notation it is written as $\langle s(\mathbf{M})\rangle$.
2) The elements of the mask are the complement of the mask elements defined by the default or the structural mask rules. In other words, the mask consists of the elements that are *not* indicated by the mask matrix. This is written as either $\langle\neg\mathbf{M}\rangle$ or $\langle\neg s(\mathbf{M})\rangle$.

Once the mask is defined and we know which elements of the output matrix will be set with the results of the operation (with or without accumulation), we must define what happens to the *other* elements of the output matrix. GraphBLAS offers two options:

1) The elements *not* selected by the mask are left unchanged. The elements from the GraphBLAS operation are therefore *merged* with the existing elements of the destination matrix, $\mathbf{C}$. This is the default behavior of the mask.
2) The elements *not* selected by the mask are deleted. We call this *replace* mode which we denote as $\langle\mathbf{M}, \mathrm{r}\rangle$.

These options compose with the previous options. For example, operations can use *replace semantics* and *structural masks* at the same time which is denoted as $\langle s(\mathbf{M}), \mathrm{r}\rangle$

To complete our overview of the GraphBLAS API, we need to return to the topic of semirings and ways this mathematical concept is relaxed in the GraphBLAS API. Mathematically, the identity of the $\oplus$ operator in a semiring is the annihilator of the $\otimes$ operator. In traditional sparse linear algebra, this identity is the assumed value for elements of an array that are not explicitly defined. GraphBLAS follows the approach from the database community and instead treats array elements that are not defined as nonexistent. Rather than sums over all elements in arrays, the GraphBLAS operations are defined over sets of defined array elements. Hence, undefined elements are never accessed and we avoid the need to associate the $\oplus$ identity with undefined elements. This means the formal mathematical rules concerning the $\oplus$ and $\otimes$ operators can be relaxed in the GraphBLAS. A programmer is free to pair operators, even user-defined operators, as needed. For example, there are graph algorithms where all you want from a binary operator is to return the second operand (an operator denoted as $second$). This operator does not have an identity or an annihilator, but it can still be used in a GraphBLAS semiring. For example, we've seen this operator combined with the $min$ operator leading to the $min.second$ semiring.

## III. Indexed Binary Operators

A typical binary operator in GraphBLAS has the form $z = f(x, y)$. When used as the multiplicative operator in a semiring, $(A \oplus.\otimes B)_{ij} = \oplus_k f(a_{ik}, b_{kj})$, where $\oplus_k$ means "summation" over $k$ using $\oplus$ as the operator. The index binary operator will provide additional parameters,

$$z = f(a_{ia,ja}, ia, ja, b_{ib,jb}, ib, jb, \theta)$$

where $i*$ and $j*$ are row and colunm indices of the two inputs from the $\mathbf{A}$ and $\mathbf{B}$ matrices, and $\theta$ is a scalar. All uses of the operator in a single matrix-matrix multiply would be given the same value of $\theta$.

This indexed binary operator will replace the current binary operator when used as:

- The multiplicative operator of a semiring (the $\otimes$ of $\mathbf{C} = \mathbf{A} \oplus . \otimes \mathbf{B}$) for `GrB_mxm`, `GrB_mxv`, and `GrB_vxm`. The operator would be used as $c_{ij} = \oplus_k f(a_{ik}, i, k, b_{kj}, k, j, \theta)$.
- The primary binary operator of element-wise operations $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$, for the element-wise add, multiply, and Kronecker product. (`GrB_eWiseAdd`, `GrB_eWiseMult`, and `GrB_kronecker`). For the `Add` and `Mult` methods, the operator would be used as $c_{ij} = f(a_{ij}, i, j, b_{ij}, i, j, \theta)$.
- Methods that create, query, revise, or destroy the object.

There are other places where a binary operator is used in GraphBLAS, but we do not anticipate allowing an indexed binary operator in these cases:

- The monoid of a semiring, where the indices of the operands are not well defined.
- The binary operator for reducing a matrix to a scalar. This requires a monoid, and the indices of the operands are not well defined.
- The accumulator operator (the $\odot$ of $\mathbf{C} \odot = \mathbf{T}$ where $\mathbf{T}$ is the result of any GraphBLAS operation).
- The operator for combining duplicates in `GrB_build`, which converts an unordered list of tuples, $(a_{ij}, i, j)$, into a sparse matrix $\mathbf{A}$.
- The apply operation, which allows for a binary operator with one input bound to a scalar. An indexed unary operator is already available to use instead.

The indexed binary operator will have the following signature:

```
void fidxop (void *z,   // output z
    const void *x,      // an entry in matrix A
    GrB_Index ia,       // row index of the entry in A
    GrB_Index ja,       // column index of the entry in A
    const void *y,      // an entry in matrix B
    GrB_Index ib,       // row index of the entry in B
    GrB_Index jb,       // column index of the entry in B
    const void *theta)  // a scalar
```

A new object, the `GrB_IndexBinaryOp`, is created to contain this function pointer. The scalar `theta` is passed to all uses of the operator when used in a single call to GraphBLAS. Its value is not part of the new operator.

We need at least 5 new methods for handling this new object:

1) `GrB_IndexBinaryOp_new` to create the object. Only the type of `theta` will be given, not its value.
2) `GrB_IndexBinaryOp_free` to destroy the object.
3) `GrB_IndexBinaryOp_get` to query the object (such as its name, and input/output types).
4) `GrB_IndexBinaryOp_set` to change the object (such as its name).
5) `GrB_IndexBinaryOp_wait` to finalize the object.

The question then becomes, where is the `theta` scalar passed to each call to GraphBLAS? We propose the following: the new `GrB_IndexBinaryOp` will not be passed directly to GraphBLAS operations such as `GrB_mxm`. Instead, we will enable a new kind of `GrB_BinaryOp`, created as a pair containing a `GrB_IndexBinaryOp` and a scalar with the value of `theta`. This binary operator can then be directly passed to an element-wise operation (such as `GrB_eWiseAdd`), or it can be incorporated into a new semiring as the multiplicative operator and the resulting `GrB_Semiring` can then be passed to `GrB_mxm`. This will require the creation of a `GrB_BinaryOp_IndexOp_new` method to create a new `GrB_BinaryOp` that is based on an underlying `GrB_IndexBinaryOp` and the specific value of a `GrB_Scalar theta`. Additional behavior would be added to `GrB_get` to query the value of `theta` in the binary op. The current API for operations that use the corresponding binary operator or semiring based on an indexed binary operator (`GrB_mxm`, `GrB_eWiseAdd`, etc) would not change.

## IV. Algorithms

### A. Breadth-First Search (BFS)

Breadth-first-search (BFS) is one of the most fundamental graph algorithms. It can exploit the `GrB_IndexBinaryOp`, which shows how important this operator is. BFS builds on the observation that vector-matrix multiplication $\mathbf{f}^\mathsf{T}\mathbf{A}$ expresses navigation from the nodes selected by vector $\mathbf{f}$ in the graph represented by $\mathbf{A}$.

When computing the BFS tree, by constructing a parent vector $\mathbf{p}$, each new node in the latest frontier $\mathbf{q}$ needs to determine its parent in the tree, where $\mathbf{p}(j) = k$ if the parent of node $j$ is node $k$. This can be done in the semiring if the multiplicative operator can return the index of its operands.

SuiteSparse:GraphBLAS has a preliminary implementation of a small set of index binary operators. In particular, the $\mathsf{secondi}(a_{ia,ja}, b_{ib,jb})$ operator returns the row index ($ib$) of its second input parameter (thus the name, secondi. This operator can be used in the min.secondi semiring to compute a single step of the BFS, $\mathbf{q}^\mathsf{T}\langle\neg s(\mathbf{p}^\mathsf{T}), \mathrm{r}\rangle = \mathbf{q}^\mathsf{T}\mathbf{A}$, where $\mathbf{q}$ is the current frontier, $\mathbf{p}$ is the parent vector, and $\mathbf{A}$ is the adjacency matrix. The mask is a *complimented structural mask* which means the mask corresponds to the empty elements of the mask vector. Replace semantics are indicated (due to the *r* in the mask expression) so any elements of the vector other than those selected by the mask are deleted.

In the matrix-vector multiply, $z = \mathsf{secondi}(q_k, a_{kj})$ returns the index $k$. If node $k$ is in the current $\mathbf{q}$, then $q_k$ is present and the multiplicative operator will return $k$ if there is an edge $(k, j)$ in the graph. Then $q_j$ will be a reduction (min, via the monoid) of the indices of all candidate parents of node $j$.

Algorithm 1 illustrates the entire push-only BFS that computes the parent vector $\mathbf{p}$. Since line 5 computes the parents of all nodes in the next frontier, the assignment to the parent vector on line 6 updates the $\mathbf{p}$ vector with the parents of the newly visited nodes.

Without this operator, an additional step is required. The vector-matrix multiply $\mathbf{q}^\mathsf{T}\langle\neg s(\mathbf{p}^\mathsf{T}), \mathrm{r}\rangle = \mathbf{q}^\mathsf{T}$ min.secondi $\mathbf{A}$ must be replaced with an apply operation with the unary rowindex operator ($\mathbf{q} = \mathsf{rowindex}(\mathbf{q})$), followed by a vector-matrix multiply using the min.first semiring instead of min.secondi, where $\mathsf{first}(x, y) = x$.

Creating a `GrB_IndexBinaryOp` in the GraphBLAS enables the simpler BFS in Algorithm 1 by through the secondi operator, making GraphBLAS more expressive. This does not, however, improve BFS performance. Our benchmarks found essentially

**Algorithm 1:** Parents BFS (push-only).

**Input:** $\mathbf{A}$, $startVertex$

**Function** *ParentsBFS*

- $\mathbf{p}(startVertex) = startVertex$
- $\mathbf{q}(startVertex) = startVertex$
- **while** $\mathbf{q}$ *is not empty* **do**
  - $\mathbf{q}^{\mathsf{T}}\langle\neg s(\mathbf{p}^{\mathsf{T}}), \mathrm{r}\rangle = \mathbf{q}^{\mathsf{T}}$ min.secondi $\mathbf{A}$
  - $\mathbf{p}\langle s(\mathbf{q})\rangle = \mathbf{q}$

the same performance between the two approaches. As we will see in Section IV-B, however, this is not the the case for the argmax algorithm where changes in both expressiveness and performance are high.

### B. ArgMin and ArgMax

The argmax method computes the maximum element in each row and the column index in which that element appears.

Using an IndexBinaryOp, argmax is very simple (Algorithm 2); a single matrix-vector multiply $\mathbf{x} = \mathbf{A1}$ where $\mathbf{1}$ is a vector of all ones. The multiplicative operator is $z = \text{maketuple}(a_{ik}, b_{kj}) = (a_{ik}, k)$ where the result $(a_{ik}, k)$ is a tuple containing a numerical value from $\mathbf{A}$, and its column index $k$. The monoid operates on this tuple domain, and returns the tuple with the largest numerical value. The monoid itself does not depend on an IndexBinaryOp. This gives us the argmax.maketuple semiring. Note that the content of the $\mathbf{1}$ vector is not accessed. SuiteSparse:GraphBLAS can create such a vector in $O(1)$ time and space. The result is a vector $\mathbf{x}$, where $x_i$ is the tuple $(\max(\mathbf{A}_{i*}), \text{argmax}(\mathbf{A}_{i*}))$ and $\mathbf{A}_{i*}$ is the $i^{th}$ row of $\mathbf{A}$.

In both algorithms presented here, entries that do not appear in the sparsity pattern of $\mathbf{A}$ do not take part in the computation. This is unlike the typical argmax of a matrix in conventional linear algebra where the missing entry is assumed to be zero. The latter can be obtained with additional computations but this is not considered here to keep things simple.

**Algorithm 2:** Row-wise ArgMax (using an IndexBinaryOp)

**Input:** $\mathbf{A}$

**Function** *ArgMax_IndexBinOp*

- $\mathbf{x} = \mathbf{A}$ argmax.maketuple $\mathbf{1}$

Without the argmax.maketuple semiring, computing the argmax is rather difficult, as shown in Algorithm 3. It returns its results as two vectors $\mathbf{x}$ (with the maximum value in each row), and $\mathbf{p}$, where $\mathbf{p}(i) = \text{argmax}(\mathbf{A}_{i*}) = j$ if $a_{ij}$ is the largest entry in the $i$th row of $\mathbf{A}$ (excluding entries not present, which do not take part in the computation).

Algorithm 3 must compute a matrix $\mathbf{G}$ that records where each maximum entry appears in each row, where $g_{ij} = 1$ if $a_{ij}$ is equal to the largest value of the entries in the $i$th row of $\mathbf{A}$. Next, it replaces the 1's in the matrix $\mathbf{G}$ with their

**Algorithm 3:** Row-wise ArgMax (without any IndexBinaryOp)

**Input:** $\mathbf{A}$

**Function** *ArgMax_Difficult*

- // compute the max entry in each row:
- $\mathbf{x} = \mathbf{A}$ max.first $\mathbf{1}$
- // find where each max entry appears in each row:
- $\mathbf{D} = diag(\mathbf{x})$
- $\mathbf{G} = \mathbf{D}$ eq.eq $\mathbf{G}$
- $\mathbf{G} = (\mathbf{G} \neq 0)$
- // find index of all maximum entries:
- $\mathbf{H} = \text{rowindex}(\mathbf{G})$
- // find index of first max entry in each row:
- $\mathbf{p} = \mathbf{H}$ min.first $\mathbf{1}$

column index using the IndexUnaryOp, rowindex, obtaining the $\mathbf{H}$ matrix. Finally, the smallest such index in each row is computed with $\mathbf{p} = \mathbf{H}$ min.first $\mathbf{1}$, obtaining the desired argmax result $\mathbf{p}$.

The secondi IndexBinaryOp can be used to remove the requirement for computing the $\mathbf{H}$ matrix, and this saves a little time. We have such an algorithm in LAGraph, but for these experiments we consider just two cases: a future algorithm relying on a complete implementation of the IndexBinaryOp (Algo 2), and one without any IndexBinaryOp (Algo 3). The latter can be implemented and its performance is considered below.

The IndexBinaryOp supports a very simple argmax algorithm, thus the expressiveness is far better. The performance is hard to judge, so we consider a proxy. MATLAB can compute both max and argmax of a sparse matrix, using single-threaded algorithms that should have reasonable performance. The relative performance of the two methods can be used as a proxy for the performance we expect when Algorithm 2 can be implemented.

Algorithms 2 and 3 compute the row-wise argmax. MATLAB stores its matrices column-wise. SuiteSparse:GraphBLAS uses row-wise storage by default but this is changed to column-wise when used via its MATLAB interface, so our performance results below consider the column-wise argmax instead. The column-wise methods are very similar to the algorithms above.

We consider a single large test matrix, the `GAP-twitter` matrix [17], [4], with 61.6 million rows and columns, and 1.47 billion entries. To ensure the MATLAB and GraphBLAS results are identical, we add an identity matrix to the test matrix (otherwise, rows and columns with no entries are handled differently). The resulting matrix has 1.52 billion entries. Results are shown in Table I. MATLAB uses a single thread; GraphBLAS uses 1 or 40 threads. All results are for computing the max or argmax of each column of this matrix.

The argmax in MATLAB takes only 1.08x the time as its column-wise max. In GraphBLAS, the ratio is very high (4.75x) because of the complexity of Algorithm 3. This ratio would drop to 1.08x with the introduction of the `GrB_IndexBinaryOp`, since computing the max in Graph-

| method | time 1 thread | time 40 threads |
|---|---|---|
| MATLAB max | 3.95 | - |
| MATLAB argmax | 4.28 | - |
| GraphBLAS max | 6.65 | 0.52 |
| GraphBLAS argmax (no index binary op) | 22.42 | 2.47 |
| GraphBLAS argmax (estimated, with index op) | 7.18 | 0.56 |

TABLE I: MATLAB R2021a and GraphBLAS v9.3.0 max and argmax on a 20-core (40 thread) Intel Xeon E5-2698v4 (2.2Ghz). Time is wallclock (sec).

BLAS is also done with a single matrix-vector multiply (just line 2 of Algorithm 3). The extra information in the tuple data structure should not have a great effect on the run time. With this assumption, the estimated time for the 40-thread argmax (Algorithm 2) would be 0.56 seconds (1.08 times the 0.52 seconds to compute the max reduction with 40-threads), a speedup of over 4x.

### C. Hierarchical Navigable Small World (HNSW)

The Hierarchical Navigable Small World algorithm is a graph-based approximate nearest neighbor search technique for searching for the nearest vector embeddings in a corpus of embeddings to a given query vector. These vector embeddings are commonly produced by tools such as word2vec and AI/ML models. They encode and compress the meaning of text or features of images into smaller, uniformly sized vector representations that can be processed and searched quickly.

Vector databases store millions to billions of embeddings in their databases, and users search for the nearest vectors stored in the database to a given query. The proximity metric is typically a vector distance function such as Euclidean L2 distance or Cosine distance. Given the very high dimensionality of the data, traditional database indexing tools such as trees and hash tables provide no useful benefit for searching along so many possible dimensions.

Without getting too deep into the details of this relatively complex algorithm, the nodes of an HNSW graph are the corpus of vectors themselves. The graph is constructed by incrementally creating several edges to existing nodes in the graph that are close to newly inserted vectors. The algorithm uses multiple graph layers of increasing edge density, using sparser granularity layers to quickly approach approximate nearest neighbors and then using denser layers to refine nearest candidates.

At search time, a multi-frontal BFS starting from random nodes is compared to the query vector to see if traversing any outgoing edges gets the search to a candidate neighbor vector that is closer to the query vector. This is done for each graph layer in turn until there are no edges that get any closer to the query. The resulting set of vector embeddings represent the approximate nearest neighbors to the query. Since these vectors can have thousands of dimensions, the distance computation to determine if an edge gets closer to a query can be quite expensive.

A commonly applied optimization to HNSW is called *Locality Sensitive Hashing* (LSH). It compresses the relatively large vector embedding into a small hash that can be compared to other hashes using a Hamming distance metric, where hashes with more dissimilar bits are more distant from each other. Computing the Hamming distance between two small hashes is very fast, requiring only an ‘xor‘ and ‘popc‘ instruction and is much more efficient than computing the distance between two large vectors. Accuracy of the hash distance can be tuned depending on hash size.

Given the large bit size for indices in the GraphBLAS and support for hypersparse index ID spaces, this makes GraphBLAS indices an ideal way to store these hashes. Support for IndexBinaryOp provides an efficient way to compute Hamming distance between source and destination nodes at edge traversal time without having to involve any other data structures to lookup vectors for each node while traversing the graph.

The User Defined Operator code for this computation is quite simple. The row identifier of the left side of the multiplication is the node identifier of the edges source, the column identifier of the right side is the node identifier of the edges destination.

```
(*z) = __builtin_popcountll(aiˆ(*theta))
     - __builtin_popcountll(bjˆ(*theta));
```

By xoring those identifiers with the theta query hash, the results contain only the different bits between them. The compiler intrinsic `__builtin_popcountll` is then used to count the number of differing bits which is the Hamming distance between the nodes and the query vectors hashes. These values are then subtracted to get a relative distance. If the result is positive, the distance to the destination from the query vector is greater than the distance from the source, and that edge will get the search closer to the goal and should be traversed. If the result is negative the edge will not get closer and should be discarded.

### D. Finite-Element Assembly

Creating the sparsity pattern of an assembled finite-element matrix is simple and fast using GraphBLAS. A finite-element analysis creates a sparse $n$-by-$n$ matrix $\mathbf{A}$ that is the sum of $m$ individual finite-elements, $\mathbf{A} = \sum_{k=1}^{m} \mathbf{A}_k$. The matrix $\mathbf{A}_k$ is a very sparse $n$-by-$n$ matrix that can be thought of as the adjacency matrix of a single small clique in a graph of $n$ nodes. In terms of a matrix, $\mathbf{A}_k$ has nonzero entries only in the positions defined by the Cartesian product of two small lists of integers, typically of length 9 to 30. If the sparsity pattern of $\mathbf{A}_k$ is symmetric then only one such list is needed. The size of this list depends on the kind of discretization (2D or 3D, polynomial degree of finite element ansatz, continuous or discontinuous) and the kind of differential equations being modeled by the finite-element method.

Let $\mathbf{F}$ be an $m$-by-$n$ boolean matrix that represents $m$ finite elements. The structure of the $k$th finite-element is represented by the $k$th row, $\mathbf{F}_{k*}$, and the list of column indices of entries present in this row gives the list of integer indices that define the sparsity pattern of $\mathbf{A}_k$. The outer product of the two vectors $\mathbf{F}_{k*}^{\mathsf{T}} \vee . \wedge \mathbf{F}_{k*}$ gives the sparsity pattern of the $k$th finite-element,

and thus the sparsity pattern of $\mathbf{A}$ is $\mathbf{F}^{\mathsf{T}} \vee . \wedge \mathbf{F}$, via a single call to `GrB_mxm`. Applying this technique in the deal-II finite-element package [3] yields a speedup of about 4x for the overall strategy, and 15x when comparing only the portion handled by GraphBLAS, on an 18-core shared-memory system, since the existing assembly of the sparsity pattern is handled with a single-threaded algorithm in deal-II version 9.5. For matrices with unsymmetric sparsity pattern and rectangular finite-elements, we can replace $\mathbf{F}^{\mathsf{T}}$ with a different matrix $\mathbf{G}$ and compute $\mathbf{A} = \mathbf{G} \vee . \wedge \mathbf{F}$ instead.

We propose using the `GrB_IndexBinaryOp` to create the numerical values of $\mathbf{A}$ as well, with a semiring that constructs each entry of $\mathbf{A}_k$ inside its multiplicative operator. Suppose the entries $f_{ki}$ and $f_{kj}$ of $\mathbf{F}$ contain a user-defined data type with enough information, along with the scalar $\theta$, so that an index binary operator can compute $(\mathbf{A}_k)_{ij} = \otimes(f_{ki}, f_{kj}, i, k, j, \theta)$. This operator would have access to all three indices $i$, $j$, and $k$, as well as the user-defined scalar $\theta$. The data type of $\theta$ could be simply a pointer to a complex user-provided data structure, and could be dereferenced with $k$ and perhaps the *global* indices $i$ and $j$ to find any information need for the $k$th finite-element.

It may also be useful for this $\otimes$ operator to have access to the *local* indices, $i_{\text{local}}$ and $j_{\text{local}}$ of the $(\mathbf{A}_k)_{ij}$ entry. The local index $i_{\text{local}}$ is an index into a small dense finite-element; that is, if $f_{ki}$ is the third entry present in the $k$th row of $\mathbf{F}_{k*}$, then $i_{\text{local}} = 3$ (or 2 if zero-based indexing is used). This local index is not part of the proposed `GrB_IndexBinaryOp`, so if needed it would be added to the user-defined data type and encoded in the value of $f_{ki}$. We anticipate other uses for this local index in LAGraph such as in the Connected Components algorithm which requires the selection of the leftmost 4 entries in each row. This suggests future work on an indexed binary operator that includes such local indices.

## V. Discussion

We have shown how an indexed binary operator can simplify the expression of algorithms in the GraphBLAS. Indexed binary operators allow for a particularly simple and elegant BFS but with no change in performance. For argmin/argmax the algorithms were both simpler and much faster.

More significantly, the indexed binary operators open up new classes of applications for the GraphBLAS. For example, the deal-II finite element package currently uses the SuiteSparse implementation of the GraphBLAS for symbolic finite-element assembly. This speeds up their package considerably (4x). As shown in this paper, with indexed binary operators, we would be able to carry out the numerical assembly of the finite element matrices as well.

It is well established that we can use the GraphBLAS to build graph database systems [7]. In this paper, we've shown that by reinterpreting the meaning of the indices in a GraphBLAS matrix, we can support vector search using the HNSW algorithm. This innovative algorithm opens up wide areas of exploration for the use of GraphBLAS in data management applications. Research on these sorts of applications, however, is only possible if we have indexed binary operators in GraphBLAS.

This paper is largely speculative. We have not implemented the indexed binary operator described in this paper and have not decided how they will appear in a future version of the GraphBLAS. In this paper, we have described one approach, one that exploits the fact that GraphBLAS operators (and more generally, GraphBLAS objects) are opaque. Opacity is a key design philosophy in the GraphBLAS. Opaque objects let us use modified versions of objects seamlessly across the API. We must define new methods to create and manage the objects that reference indexed binary operators, but then we can use them almost anywhere a binary operator can be used in the GraphBLAS.

As we've suggested in this paper, however, there are limitations to the indexed binary operator that follow from this design choice. Our design is at odds with how the existing indexed unary operator is defined. The appearance of the scalar $\theta$ inside the operator may not work well with algorithms that need to change the scalar from one invocation of a method to the next. Alternative designs for the operator are possible and will be actively discussed within the GraphBLAS community as we decide how these operators will appear in a future version of the GraphBLAS C API.

## VI. Conclusion

In this paper we laid out the case for a new indexed binary operator in the GraphBLAS. Our goal is to tell the GraphBLAS community about this new operator and to give them some ideas about how it might be defined. To complete the design, however, we need use-cases and feedback on the proposed semantics. The results in this paper are mostly speculative, but not purely so. LAGraph [13] using the SuiteSparse [8] implementation of the GraphBLAS has implemented a small number of indexed binary operators. This has shown that the concept works and can deliver both simplicity and improved performance.

Our results show that in terms of expressiveness, an operator in a semiring that has access to the indices of matrix elements is powerful. What is particularly interesting is how this opens up new types of applications for the GraphBLAS. In particular, with indexed binary operators, we are moving closer to our goal of supporting data management (such as HNSW), differential equation solvers, and graph analytics all from within a single framework. GraphBLAS ... one algebra to rule them all!


## Acknowledgements

T. Davis was supported by NSF CNS-1514406, FalkorDB, MIT Lincoln Lab, NVIDIA, and Intel. This material is also based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center [DM21-0298]. M. Bezbaruah and M. Maier were partially supported by NSF DMS-2045636.

## REFERENCES


[1] "GraphBLAS Forum," https://graphblas.github.io.

[2] "LAGraph GitHub repository," https://github.com/GraphBLAS/LAGraph.

[3] D. Arndt, W. Bangerth, M. Bergbauer, M. Feder, M. Fehling, J. Heinz, T. Heister, L. Heltai, M. Kronbichler, M. Maier, P. Munch, J.-P. Pelteret, B. Turcksin, D. Wells, and S. Zampini, "The `deal.II` library, version 9.5," *Journal of Numerical Mathematics*, vol. 31, no. 3, pp. 231–246, 2023. [Online]. Available: https://dealii.org/deal95-preprint.pdf

[4] S. Beamer *et al.*, "The GAP Benchmark Suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[5] B. Brock *et al.*, "The GraphBLAS C API specification v2.1," 2023, https://graphblas.org/docs/GraphBLAS_API_C_v2.1.0.pdf.

[6] A. Buluç *et al.*, "Design of the GraphBLAS API for C," in *GrAPL at IPDPS*. IEEE Computer Society, 2017, pp. 643–652. [Online]. Available: https://doi.org/10.1109/IPDPSW.2017.117

[7] P. Cailliau *et al.*, "RedisGraph GraphBLAS enabled graph database," in *GrAPL at IPDPS*. IEEE, 2019, pp. 285–286. [Online]. Available: https://doi.org/10.1109/IPDPSW.2019.00054

[8] T. A. Davis, "Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra," *ACM Trans. Math. Softw.*, 2019. [Online]. Available: https://doi.org/10.1145/3322125

[9] T. A. Davis *et al.*, "Write quick, run fast: Sparse deep neural network in 20 minutes of development time via SuiteSparse:GraphBLAS," in *HPEC*. IEEE, 2019, pp. 1–6. [Online]. Available: https://doi.org/10.1109/HPEC.2019.8916550

[10] J. Kepner, T. Davis, C. Byun, W. Arcand, D. Bestor, W. Bergeron, V. Gadepally, M. Hubbell, M. Houle, M. Jones, A. Klein, P. Michaleas, L. Milechin, J. Mullen, A. Prout, A. Rosa, S. Samsi, C. Yee, and A. Reuther, "75,000,000,000 streaming inserts/second using hierarchical hypersparse GraphBLAS matrices," in *GrAPL at IPDPS*. IEEE, 2020, pp. 207–210. [Online]. Available: https://doi.org/10.1109/IPDPSW50202.2020.00046

[11] J. Kepner *et al.*, "Mathematical foundations of the GraphBLAS," in *HPEC*. IEEE, 2016. [Online]. Available: https://doi.org/10.1109/HPEC.2016.7761646

[12] J. Kepner and J. R. Gilbert, Eds., *Graph Algorithms in the Language of Linear Algebra*. SIAM, 2011. [Online]. Available: https://doi.org/10.1137/1.9780898719918

[13] T. Mattson, T. A. Davis, M. Kumar, A. Buluç, S. McMillan, J. E. Moreira, and C. Yang, "LAGraph: A community effort to collect graph algorithms built on top of the GraphBLAS," in *GrAPL at IPDPS*, 2019. [Online]. Available: https://doi.org/10.1109/IPDPSW.2019.00053

[14] T. Mattson *et al.*, "Standards for graph algorithm primitives," in *HPEC*. IEEE, 2013. [Online]. Available: https://doi.org/10.1109/HPEC.2013.6670338

[15] G. Szárnyas, D. A. Bader, T. A. Davis, J. Kitchen, T. G. Mattson, S. McMillan, and E. Welch, "Lagraph: Linear algebra, network analysis libraries, and the study of graph algorithms," in *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2021, Portland, OR, USA, June 17-21, 2021*. IEEE, 2021, pp. 243–252. [Online]. Available: https://doi.org/10.1109/IPDPSW52791.2021.00046

[16] X. Wang *et al.*, "Accelerating DNN inference with GraphBLAS and the GPU," in *HPEC*. IEEE, 2019, pp. 1–6. [Online]. Available: https://doi.org/10.1109/HPEC.2019.8916498

[17] J. Yang and J. Leskovec, "Temporal variation in online media," in *ACM Intl. Conf. on Web Search and Data Mining (WSDM '11)*, 2011.