Write Quick, Run Fast: Sparse Deep Neural Network in 20 Minutes of Development Time via SuiteSparse:GraphBLAS

Timothy A. Davis*, Mohsen Aznaveh†, and Scott Kolodziej‡
Dept. of Computer Science and Engineering
Texas A&M University
College Station, TX
Email: *davis@tamu.edu, †aznaveh@tamu.edu, ‡scottk@tamu.edu

***Abstract*—SuiteSparse:GraphBLAS is a full implementation of the GraphBLAS standard, which provides a powerful and expressive framework for creating graph algorithms based on the elegant mathematics of sparse matrix operations on a semiring. Algorithms written in GraphBLAS achieve high performance with minimal development time. Using GraphBLAS, it took a mere 20 minutes to write a first-cut computational kernel that solves the Sparse Deep Neural Network Graph Challenge. Understanding the problem description and file format, writing code to read in the files that define the problem, and comparing our results with the reference solution took a full day. The kernel consists of a single for-loop around 4 lines of code, all of which are calls to GraphBLAS, and it worked perfectly the first time it was compiled. The sequential performance of the GraphBLAS solution is 3x to 5x faster than the MATLAB reference implementation. OpenMP parallelism gives an additional 10x to 15x speedup on a 20-core Intel processor, 17x on an IBM Power8 system, and 20x on a Power9 system, for the largest problems. Since SuiteSparse:GraphBLAS does not yet employ MPI, this was added at the application level, a development effort that took one week, primarily because of difficulties in resolving a load-balancing issue in the MPI-based parallel algorithm.**

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

I. Introduction

The GraphBLAS standard [1] defines sparse matrix and vector operations on an extended algebra of semirings. The operations are useful for creating a wide range of graph algorithms. Kepner and Gilbert [2] provide a framework for understanding how graph algorithms can be expressed as matrix computations. This approach leads to high performance, since the library can then compute bulk operations on adjacency matrices. User code need not deal with individual nodes and edges. Writing graph algorithms with GraphBLAS reduces development time as well, as illustrated by the results in this paper.

To demonstrate the utility of GraphBLAS in solving large-scale computational problems quickly from both a performance and development standpoint, we used it to solve the Sparse Deep Neural Network Graph Challenge [3]. Not accounting for the software development time to understand the problem format, to write the code to read in the problem into GraphBLAS matrices and to check the results with the posted solutions, writing the computational kernel for solving the entire sparse deep neural network took only 20 minutes of programmer time. The kernel worked correctly the first time it was compiled.

Sparse Deep Neural Network Graph Challenge

Deep neural networks have become highly effective tools throughout the fields of artificial intelligence and machine learning. However, the computational time and effort required to solve these networks has increased as they have grown larger. To combat this, pruning and sparse coding methods were introduced to remove unimportant connections in these networks, decreasing the amount of computation required for forward propagation of values through the network [4]–[6].

The sparse deep neural network problem (Figure 1) involves computing an output vector (Y_n) based on an input vector (Y_0) and a deep neural network consisting of n layers of a fixed number of neurons in each layer. The connections between each layer of neurons are sparse, meaning that not all inter-neuronal connections are present (or, alternatively, many of the inter-neuronal connections have zero weight) [7]–[9]. The values of each layer are computed using a rectified linear unit (ReLU) computation. In practice, there are many independent input vectors, yielding a matrix of inputs; each layer computation is thus a sparse matrix-matrix multiplication. The Sparse Deep Neural Network Graph Challenge describes several instances of the sparse deep neural network problem, which we solve using GraphBLAS [3].

II. Overview of GraphBLAS objects, methods, and operations

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

With support from NSF CNS-1514406, NVIDIA, Intel, MIT Lincoln Lab, Redis Labs, and IBM. Portions of this research were conducted with the advanced computing resources provided by Texas A&M High Performance Research Computing.

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

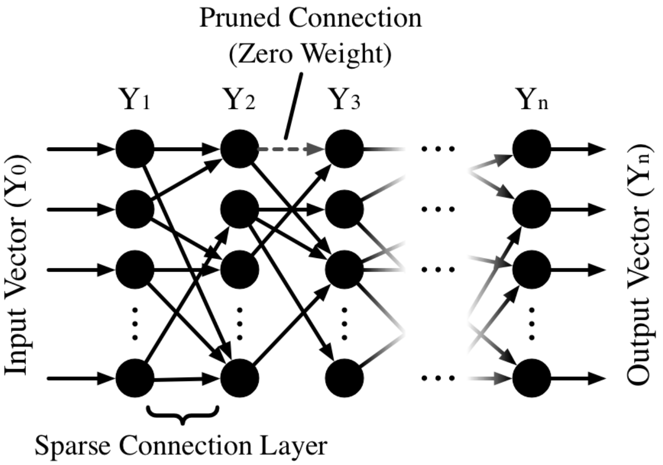

Fig. 1. Sparse Deep Neural Network Problem

described below. User applications can also define their own data types, operators, monoids, and semirings.

(1) Types: A GraphBLAS type (GrB_Type) can be any of 11 built-in types (Boolean, integer and unsigned integers of sizes 8, 16, 32, and 64 bits, and single and double precision floating point). In addition, user-defined scalar types can be created from nearly any C typedef. The sparse deep neural network problem is solved in GraphBLAS using single precision floating-point.

(2) Unary operators: A unary operator (GrB_UnaryOp) is a function $z = f(x)$. The sparse deep neural network problem requires a user-defined unary operator as part of the ReLU threshold.

(3) Binary operators: Likewise, a binary operator (GrB_BinaryOp) is a function $z = f(x, y)$, such as $z = x + y$ or $z = xy$. The floating-point plus and times operators were used, but only as part of the two semirings, for the sparse DNN.

(4) Select operators: The GxB_SelectOp operator is a SuiteSparse extension to the GraphBLAS API. It is used in the GxB_select operation to select a subset of entries from a matrix, like L=tril(A) in MATLAB. This operator was also used for the ReLU phase on the sparse deep neural network.

(5) Monoids: The scalar addition of conventional matrix multiplication is replaced with a *monoid*. A monoid (GrB_Monoid) is an associative and commutative binary operator $z = f(x, y)$ where all three domains are the same (the types of x, y, and z) and where the operator has an identity value o such that $f(x, o) = f(o, x) = x$. Performing matrix multiplication with a semiring uses a monoid in place of the "add" operator, scalar addition being just one of many possible monoids. The sparse DNN relies only on the plus monoid.

(6) Semirings: A *semiring* (GrB_Semiring) consists of a monoid and a "multiply" operator. Together, these operations define the matrix "multiplication" $\mathbf{C} = \mathbf{AB}$, where the monoid is used as the additive operator and the semiring's "multiply" operator is used in place of the conventional scalar multiplication in standard matrix multiplication via the plus-times semiring. The sparse DNN requires two different semirings: the conventional plus-times, and a plus-plus semiring, both over the single-precision floating-point type.

TABLE I
SUITESPARSE:GRAPHBLAS OPERATIONS USED IN SOLVING THE SPARSE DEEP NEURAL NETWORK GRAPH CHALLENGE

function name	description	GraphBLAS notation
GrB_mxm	matrix-matrix mult.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{AB}$
GrB_apply	apply unary op.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C}\odot f(\mathbf{A})$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w}\odot f(\mathbf{u})$
GxB_select	apply select op.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C}\odot f(\mathbf{A}, \mathbf{k})$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w}\odot f(\mathbf{u}, \mathbf{k})$
GrB_eWiseMult	element-wise, set-union	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \otimes \mathbf{B})$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$
GrB_reduce	reduce to vector reduce to scalar	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w}\odot[\oplus_j \mathbf{A}(:, j)]$ $s = s \odot [\oplus_{ij} \mathbf{A}(i, j)]$

(7) Descriptors: A *descriptor* GrB_Descriptor with parameter settings for GraphBLAS operations.

(8) Vectors: A sparse vector, GrB_Vector.

(9) Matrices: A sparse matrix, GrB_Matrix.

GraphBLAS methods and operations

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

Table I lists a subset of GraphBLAS operations used in solving the Sparse Deep Neural Network Challenge in the GraphBLAS notation where $\mathbf{AB}$ denotes the multiplication of two matrices over a semiring. Upper case letters denote a matrix, and lower case letters are vectors. The two operations in the latter part (GrB_eWiseMult and GrB_reduce) were used to compare the results with the reference solution. An optional accumulator operator ($\odot$) and mask matrix ($\mathbf{M}$) can be specified, written as $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$ where $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$ denotes the application of the accumulator operator, and $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$ denotes the mask operator via the Boolean matrix $\mathbf{M}$. The mask matrix is used to selectively write results into the output matrix. The mask was not used in our solution, but it could be used in conjunction with two calls to GrB_apply to implement the ReLU operator, in place of GxB_select.

III. OPENMP PARALLELISM IN SUITESPARSE:GRAPHBLAS

SuiteSparse:GraphBLAS Version 2.3.4 is to appear as a Collected Algorithm of the ACM [10]. While its sequential performance is good, as illustrated in that paper and in the results in this paper, it does not exploit any parallelism at all. An OpenMP version is in progress, and is robust enough to use for these experiments. A CUDA-accelerated version is also in progress, and an MPI version is planned in the more future. While all major operations in SuiteSparse:GraphBLAS

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

(V3.0.0 Draft, July 9) have been parallelized, the subset of operations used in solving the Sparse Deep Neural Network Graph Challenge (and the method by which they were parallelized) are described below.

A. GrB_mxm: Parallel matrix multiplication

The sequential version of SuiteSparse:GraphBLAS includes three different forms of matrix-matrix multiply: Gustavson's method [11], a heap-based method [12], and a dot-product based method. Each of these has a masked variant to compute $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}$, and the dot product variant can also compute $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{AB}$.

By default, all matrices in SuiteSparse:GraphBLAS are held in compressed-sparse row (CSR) format, but the matrices can also be held in compressed-sparse column format (CSC). This discussion assumes the default CSR format.

Gustavson's method and the heap-based method are both *saxpy*-based, where the ith row is computed as a sum of scaled sparse vectors. In MATLAB notation, assuming the conventional plus-times semiring:

```
for k = find (A(i,:))
    C(i,:) = C(i,:) + A(i,k) * B(k,:)
end
```

Gustavson's method uses a size-n gather/scatter workspace, where C is m-by-n. In the parallel case, each thread requires its own workspace. SuiteSparse:GraphBLAS keeps a set of workspaces that can be used in subsequent operations, to reduce the time to initialize this space. However, for large numbers of threads, the Gustavson method does not scale well. The heap-based method avoids this problem, but (at least in our current implementation) it is not as fast as Gustavson's method. It merges the vectors of B using a heap of size nnz(A(i,:)). In both methods, all rows of C can be computed in parallel.

Our current implementation divides the work into a single task for each thread, where the tasks are chosen to balance the operation count in each task. Each submatrix of C is computed in parallel, and then the resulting submatrices are concatenated together.

The dot-product method takes a different approach. It efficiently computes C=A*B', if the matrices are in CSR format. Each entry C(i,j) is computed independently. The method is not well-suited for general matrix-matrix multiplication, since all mn dot products must be computed. The time is thus $\Omega(mn)$. However, if the mask is present, only entries in the mask need be computed. In this case, the dot product method can be much faster than Gustavson's method or the heap-based method.

SuiteSparse:GraphBLAS automatically selects the method to use, although the user application can make this selection instead. The sparse deep neural network solution spends the bulk of its time in the parallel Gustavson matrix-matrix multiplication.

The SuiteSparse implementation of GrB_mxm also includes two specialized matrix multiplication methods, in which one of the matrices A or B are diagonal. These two methods are easy to parallelize, and are fast sequentially as well. The sparse deep neural network problem uses one of these methods for its second GrB_mxm, to apply the bias to each neuron, using the plus-plus semiring.

B. GrB_apply: Parallel unary operators

The GrB_apply operation applies a unary operator to each entry in A, as C=f(A). It is easy to parallelize, since dense vectors of A can easily be split into multiple tasks. Unlike sparse matrix addition (GrB_eWiseAdd) or element-wise multiplication (GrB_eWiseMult), which require a nested binary search to split dense columns, GrB_apply can directly split a dense vector amongst several tasks. All the tasks are the same size. The GrB_apply operation is used in the sparse deep neural network solution for the YMAX threshold, to limit the values in Y to 32.

C. GxB_select: Parallel selection operators

SuiteSparse:GraphBLAS adds the GxB_select operation as an extension to the API Specification. It selects a subset of entries from a matrix, keeping only those for which the select operator is true. In this way, the select operator acts much like a functional mask. The selector can depend on the row and column index, the dimension of A, and the value of the entry. There are two kinds of built-in operators: those that depend solely on the position of the entry (like tril and triu), and those that depend on the value (such as keeping only nonzero values). User-defined select operators can combine both tests. The parallelism for the two kinds of operators is slightly different. Both are split into an analysis phase that counts the number of entries in each vector of the result, and a execution phase that constructs the result. Methods that depend only on the position require only a binary search for each vector in the analysis phase. Computation is divided into tasks in much the same as GrB_apply.

GxB_select was introduced into SuiteSparse:GraphBLAS for the 2018 Graph Challenge, since it was needed to compute the lower triangular part of a matrix for the triangle counting problem [13], or L=tril(A) in MATLAB notation. It now finds use in the sparse deep neural network problem, as the ReLU function, which must drop entries at each layer, keeping only those greater than zero. Since it proved useful for both the 2018 and 2019 Graph Challenges, GxB_select is a good candidate to consider for adding to a future GraphBLAS C Specification.

D. GrB_*_build: Parallel matrix and vector build

GrB_Matrix_build creates a CSR or CSC matrix from a list of unsorted tuples (each with a row index, column index, and value). The parallel method divides into five phases. Phase 1 makes a copy of the user input. Phase 2 sorts the tuples, using a parallel quicksort. Phase 3 finds the non-empty vectors and the duplicate entries in $O(e/p)$ time. Phase 4 constructs the vector pointers, and list of non-empty vectors. The final phase assembles the tuples. All phases except the parallel quicksort in phase 2 take $O(e/p)$ time, with p threads and an input of e tuples. GrB_Vector_build creates an analogous sparse vector.

This method is well suited for constructing *hypersparse* matrices, since no part of the time or memory complexity depends on the matrix dimensions. In a hypersparse CSR matrix, the list of row vectors itself becomes sparse, and the total memory required is $O(e)$ for a matrix with e entries. The output is always constructed as hypersparse, and then converted to standard CSR or CSC format, if appropriate. An m-by-n matrix in standard CSR format requires $O(m + e)$ memory. If $e > m$, the standard format is faster, but it uses too much memory and takes too much time to construct if $e << m$. In that case, the matrix is constructed in hypersparse form.

GrB_Matrix_build was used to construct the input matrices for the sparse deep neural network, from the input files, although this time was not included in the total computation.

IV. SuiteSparse:GraphBLAS development

SuiteSparse:GraphBLAS has undergone intense development since February 2017, and the sequential version appears in the ACM Transactions on Mathematical Software, consisting of 28K lines of code. The OpenMP version was started in February 2019, and is still in progress. As of July, 2019, the draft Version 3.0.0 is 41K lines in size, all close to robust, library quality code ready for inclusion in a production application. It is nearly fully parallel. Overall, this reflects a total effort of about 18 months, spanning 2.5 years, to create the parallel GraphBLAS library used to obtain the results presented in this paper. SuiteSparse:GraphBLAS depends on no other libraries except for the standard ANSI C run-time library.

V. Sparse Deep Neural Network solution in GraphBLAS

Compared with *writing* GraphBLAS, the level of effort required to create a working solution to the Sparse Deep Neural Network Graph Challenge *using* SuiteSparse:GraphBLAS stands in stark contrast:

Twenty lines in twenty minutes.

The code is shown in Figure 2. The time to write the code was carefully monitored. This first version did not include the ymax operator and the call to GrB_apply. The function compiled and worked perfectly, the first time it was compiled. The time of 20 minutes does not include the time it took to read the problem definition, to understand the non-standard file format of the problem, and to write the code to read in the files, call the dnn function, and check the result with the reference solution. All of that took about a full working day, for an additional of 220 lines of code. This full-working day level of effort points to the need for more standard file I/O formats in either GraphBLAS, or LAGraph [14]. The input files for the graph challenge were not in a standard input format, such as the Matrix Market format.

```
#include "GraphBLAS.h"
void ymax_fp32 (float *z, const float *x)
{
    (*z) = fminf ((*x), (float) 32.0) ;
}
void dnn        // solve a sparse deep neural network
(
    GrB_Matrix *Yhandle, // Y, created on output
    GrB_Matrix *W,       // W [0..nlayers-1]
    GrB_Matrix *Bias,    // Bias [0..nlayers-1]
    int nlayers,         // # of layers
    GrB_Matrix Y0        // nfeatures-by-nneurons
)
{
    GrB_Matrix Y = NULL ;
    GrB_UnaryOp ymax = NULL ;
    GrB_Index nfeatures, nneurons ;
    GrB_Matrix_nrows (&nfeatures, Y0) ;
    GrB_Matrix_ncols (&nneurons,  Y0) ;
    GrB_Matrix_new (&Y, type, nfeatures, nneurons) ;
    GrB_UnaryOp_new (&ymax, ymax_fp32, GrB_FP32, GrB_FP32) ;
    // propagate the features through the neuron layers
    for (int layer = 0 ; layer < nlayers ; layer++)
    {
        // Y = Y * W [layer]
        GrB_mxm (Y, NULL, NULL, GxB_PLUS_TIMES_FP32,
            ((layer == 0) ? Y0 : Y), W [layer], NULL) ;
        // Y(i,j) += Bias [layer] (j,j) for each Y(i,j)
        GrB_mxm (Y, NULL, NULL, GxB_PLUS_PLUS_FP3
            2, Y, Bias [layer], NULL) ;
        // delete entries; keep only those > 0
        GxB_select (Y, NULL, NULL, GxB_GT_ZERO, Y,
            NULL, NULL) ;
        // threshold maximum values: Y (Y > 32) = 32
        GrB_apply (Y, NULL, NULL, ymax, Y, NULL) ;
    }
    GrB_free (&ymax) ;  // free the unary operator
    (*Yhandle) = Y ;    // return result
}
```

Fig. 2. A complete solution to the Sparse Deep Neural Network in GraphBLAS, requiring 20 minutes to write.

```
function Y = inferenceReLUvec (W, bias, Y0)
% Performs ReLU inference  using input feature
%  vector(s) Y0, DNN weights W, and constant bias
    Y = Y0 ;
    nlayers = length (W) ;
    % Loop through each weight layer W{layer}
    for layer = 1:nlayers
        % Propagate through layer.
        Z = Y * W{layer} ;
        % Apply bias to non-zero entries.
        Y = Z + (double(logical(Z)) .* bias {layer}) ;
        % Threshold negative values.
        Y (Y < 0) = 0 ;
        % Threshold maximum values.
        Y (Y > 32) = 32 ;
    end
```

Fig. 3. A complete solution to the Sparse Deep Neural Network in MATLAB.

For comparison, the MATLAB reference implementation posted at graphchallenge.org is shown in Figure 3, slightly edited for style, to align it more closely with Figure 2. The edits do not affect performance or the length of code, just the spacing and variable names. Each of the four lines of MATLAB in the innermost loop correspond to a single line of GraphBLAS in Figure 2.

In most respects, the MATLAB implementation is simpler and more elegant that the GraphBLAS solution, but in one aspect it is more complex. The application of the bias for

Sparse DNN Problem			MATLAB on (1)		GraphBLAS on (1)				GraphBLAS on (2)			GraphBLAS on (3)				GraphBLAS on (4)	
Neurons Per Layer	Layers	Total Neurons	1 Thread		1 Thread		40 Threads		Best of 1-160 Threads			1 Node (20 Threads)		2 Nodes (40 Threads)		64 Threads	
			Time (s)	Rate (10^4/s)	Time (s)	Rate (10^9/s)	Time (s)	Rate (10^9/s)	Time (s)	Rate (10^9/s)		Time (s)	Rate (10^9/s)	Time (s)	Rate (10^9/s)	Time (s)	Rate (10^9/s)
1,024	120	1.23×10^5	179	2.19	21	11.3	2	**153.3**	2	110.8	←40 Threads→	2	118.6	3	92.6	4	57.5
	480	4.92×10^5	678	2.32	57	16.4	4	**239.5**	7	139.0		5	176.7	7	128.4	14	67.7
	1920	1.97×10^6	2681	2.35	203	18.5	14	**275.4**	25	151.0		20	190.2	28	134.6	53	70.6
4,096	120	4.92×10^5	818	1.92	100	9.5	9	100.4	8	125.0		7	**138.8**	9	103.1	8	115.2
	480	1.97×10^6	3240	1.94	303	12.5	30	126.9	23	164.2		22	**174.3**	30	127.5	27	140.3
	1920	7.86×10^6	12738	1.98	1108	13.6	106	143.1	83	182.5		81	**186.3**	109	138.0	101	149.5
16,384	120	1.97×10^6	3452	1.82	769	4.9	50	75.1	37	101.9	←160 Threads→	36	**104.8**	42	89.1	38	98.8
	480	7.86×10^6	13987	1.80	2630	5.7	183	82.7	135	111.7		131	**115.6**	142	106.1	131	115.1
	1920	3.15×10^7	59354	1.70	10182	5.9	689	87.7	531	113.8		599	100.9	-	-	502	**120.3**
65,536	120	7.86×10^6	15472	1.63	3697	4.1	251	60.1	144	**105.0**		167	90.4	-	-	186	81.4
	480	3.15×10^7	62678	1.61	13103	4.6	885	68.3	553	**109.1**		-	-	-	-	682	88.6
	1920	1.26×10^8	259919	1.55	51387	4.7	3721	64.9	2143	**112.7**		-	-	-	-	-	-

TABLE II
SPARSE DEEP NEURAL NETWORK CHALLENGE COMPUTATIONAL RESULTS

each neuron can be done with a single multiplication by a diagonal matrix, `Bias [layer]`, in GraphBLAS, using the `PLUS_PLUS_FP32` semiring. This is difficult to do in MATLAB, since MATLAB does not provide this semiring.

Using purely built-in operators, SuiteSparse:GraphBLAS has 1040 unique semirings that can be used in a wide variety of graph algorithms (independent set, breadth-first search, centrality metrics, and so on). MATLAB has just two semirings that it can apply to its sparse matrices: `PLUS_TIMES_FP64` and `PLUS_TIMES_COMPLEX`. Along with the masking operation, this gives GraphBLAS a distinct edge (pun intended) in writing complex graph algorithms, although this particular problem does not require the GraphBLAS mask, and the bulk of the work is done with a conventional linear algebra semiring (`Z=Y*W{layer}`).

VI. MPI PARALLELISM

The sparse deep neural network problem is embarrassingly parallel due to its many independent input vectors. We used GraphBLAS as a basis to add further parallelism at the user application level, in addition to the internal OpenMP parallelism, to improve performance.

Our first parallel implementation uses MPI, where each process computes the matrix multiplication using a subset of independent rows. There is no communication between processes, and each process checks its result with the expected categories. While this method only requires another 20 minutes to implement and provides a reasonable performance improvement, it suffers from load balancing difficulties, as some rows may require significantly more or less operations to compute the final result.

To address this problem, our second parallel implementation uses a manager/worker strategy. In this algorithm, tasks have a finer granularity and are scheduled by the manager; once any worker finishes their part, it calls the manager and another job is assigned to the worker until no jobs remain. This schema adds another 40 lines of code, but scales significantly better.

Both MPI implementations require very little time to implement. However, as with any MPI application, cross-platform compilation and performance varies greatly. Note that both implementations use independent memory spaces, which could be further improved using a shared memory model for increasingly large problems.

VII. PERFORMANCE RESULTS

Experiments were performed on four systems:

(1) An NVIDIA DGX Station (256GB RAM, Intel Xeon E5-2698 v4, with 2.2 GHz, 20 hardware cores (40 threads)), with the Intel `icc` compiler (19.0.3.199). The GPUs were not used since the CUDA-based parallelism is still in progress.
(2) An IBM Minsky system, with 1TB of RAM and 160 hardware threads (IBM Power8 8335-GTB, 4GHz, 20 hardware cores with 8-way threading on each core, `gcc` v7.2.0 compiler).
(3) A 4-node IBM Power9 cluster, with each node having 256GB of RAM and 160 hardware threads (40 hardware cores at 2.4GHz).
(4) A Lenovo x86 cluster with two CPU sockets (34-core Intel Xeon Phi CPU 7250 1.40GHz) with `icc` compiler (19.0.1.144) and 96GB RAM per node. Results for 64 threads (on a single node) are reported.

Computational results for solving the Sparse Deep Neural Network Challenge are tabulated in Table II. GraphBLAS on (1) uses the OpenMP-based parallelism described in Section III, while the other GraphBLAS results use both the OpenMP and MPI-based parallelism described in Section VI to take advantage of multiple nodes within a cluster. A serial MATLAB implementation is provided for comparison. Best results for a given sparse DNN problem are shown in bold; generally, larger problems can be more efficiently solved using more parallelism (combining both OpenMP and MPI-based parallelism). Wall times (in seconds) and connection rates (defined by inputs × DNN connections per unit time) are reported for all approaches.

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

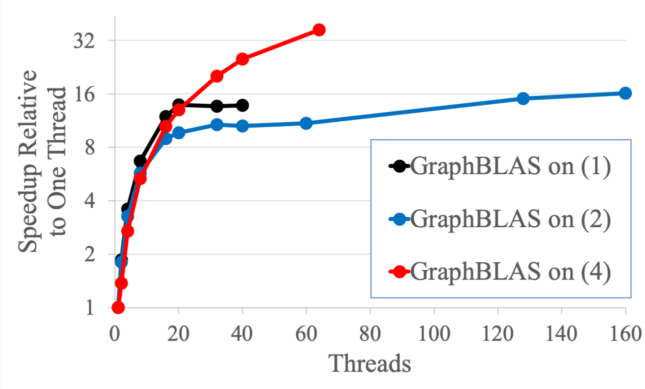

Fig. 4. Scaling Behavior of Parallel GraphBLAS Approaches

The scaling behavior of the various approaches on the 65,536 neuron/1920 layers sparse DNN is shown in Figure 4. Note that GraphBLAS on (3) was not included, as it was not able to solve the problem on the system (3) due to time constraints, and GraphBLAS on (4) is shown solving the 65,536 neuron/480 layer DNN, as it was unable to solve the larger 1920 layer problem. Generally, all approaches scale well up to the maximum number of hardware threads per node, beyond which performance scales more modestly. Future experiments taking advantage of more nodes are necessary to better define the scaling behavior.

VIII. Conclusions

These results demonstrate that GraphBLAS can be an efficient library that allows end users to write simple yet fast code. All codes used in this paper will appear at http://suitesparse.com.

References

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

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

[3] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, R. Robinett, and S. Samsi, "Sparse deep neural network graph challenge," MIT Lincoln Laboratory Supercomputing Center, Tech. Rep., 2019, http://graphchallenge.mit.edu/sites/default/files/documents/SparseDNN-GraphChallenge-2019-06-13-DRAFT.pdf.

[4] H. Lee, A. Battle, R. Raina, and A. Y. Ng, "Efficient sparse coding algorithms," in *Advances in neural information processing systems*, 2007, pp. 801–808.

[5] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[6] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[7] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, "Exploring the regularity of sparse structure in convolutional neural networks," *arXiv preprint arXiv:1705.08922*, 2017.

[8] X. Liu, J. Pool, S. Han, and W. J. Dally, "Efficient sparse-winograd convolutional neural networks," *arXiv preprint arXiv:1802.06367*, 2018.

[9] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 27–40.

[10] T. A. Davis, "Algorithm 9xx: SuiteSparse:GraphBLAS: graph algorithms in the language of sparse linear algebra," *ACM Trans. Math. Softw.*, vol. to appear, 2019.

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

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

[13] T. A. Davis, "Graph algorithms via SuiteSparse:GraphBLAS: triangle counting and k-truss," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, Sep. 2018, pp. 1–6.

[14] T. Mattson, T. A. Davis, M. Kumar, A. Buluc, S. McMillan, J. Moreira, and C. Yang, "LAGraph: A community effort to collect graph algorithms built on top of the GraphBLAS," in *Proc. GrAPL'19, Workshop on Graphs, Architectures, Programming, and Learning*, 2019.