Accelerating DNN Inference with GraphBLAS and the GPU

Xiaoyun Wang
Department of Computer Science
University of California, Davis
Davis, California 95616
Email: xiywang@ucdavis.edu

Zhongyi Lin Carl Yang John D. Owens
Department of Electrical & Computer Engineering
University of California, Davis
Davis, California 95616
Email: {zhylin, ctcyang, jowens}@ucdavis.edu

***Abstract*—This work addresses the 2019 Sparse Deep Neural Network Graph Challenge with an implementation of this challenge using the GraphBLAS programming model. We demonstrate our solution to this challenge with GraphBLAST, a GraphBLAS implementation on the GPU, and compare it to SuiteSparse, a GraphBLAS implementation on the CPU. The GraphBLAST implementation is 1.94× faster than SuiteSparse; the primary opportunity to increase performance on the GPU is a higher-performance sparse-matrix-times-sparse-matrix (SpGEMM) kernel.**

I. Introduction

The newest GraphChallenge (2019) targets inference using large sparse deep neural networks. Neural networks are ubiquitous in a wide range of modern machine learning workloads. Inference is the process of using a trained network to evaluate an input. The larger the network, the better the quality of the evaluation; but network size is limited by the size of processor memory. Thus an emerging area of focus is to prune the network by removing network connections with small weights, making the networks sparse and thus able to either achieve similar accuracy and better performance with less memory or superior accuracy and similar performance with the same amount of memory as dense networks.

Dense networks are straightforward to parallelize as any reasonable decomposition of the network across parallel processors results in uniform workloads per processor. Sparsifying the network will likely result in load imbalances across processors, so implementing high-performance inference using sparse networks is a more challenging task.

Our implementation treats the network as a *graph* and thus can leverage the significant investment in high-performance graph computation frameworks to address this problem. High-performance graph frameworks are well-suited to address parallel workloads with fine-grained load imbalance. Our framework of choice is based on the GraphBLAS [5], an open standard specification that expresses graph computation in the language of linear algebra. The initial mapping of inference on large sparse deep neural networks to the GraphBLAS was the work of Kepner et al. [6], who demonstrated how the mathematics of inference map to the GraphBLAS. The GraphChallenge problem that we address here has some important differences from their work—different matrix sizes, varying layer counts, and a more specified testing methodology—but the mathematics described by Kepner et al. is the core of our implementation.

We compare against the "SuiteSparse" GraphBLAS implementation of Davis [1], implemented on a CPU. Davis implemented this GraphChallenge problem, using his SuiteSparse GraphBLAS backend, in the LAGraph algorithm suite [8]. (For the remainder of this paper, we will refer to this work as "SuiteSparse".)

Our contributions in this work are:

1) We implement the GraphChallenge problem on the GPU using our "GraphBLAST" GraphBLAS backend [14].
2) We mitigate the problem of limited GPU memory using data parallelism. This allows us to complete the GraphChallenge using a GPU with 12 GB main memory that would otherwise not be able to fit the 16 384- and 65 536-neuron models.
3) We perform a thorough performance comparison with SuiteSparse and MATLAB baseline that indicates we get a 1.94× geomean (3.17× peak) and 43.3× geomean (56.0× peak) speedups respectively.
4) We highlight the importance of one specific form of load-balancing: given sparse matrices Y and W, deciding whether to perform the multiplication using the matrices in CSR (compressed sparse row) format. We note that on these datasets choosing the correct format yields a 5.80× geomean (175.5× peak) speedup.
5) Our performance analysis shows the source of our speedup over SuiteSparse: (1) parallelizing the filtering out of zeroes from the activation matrix (SuiteSparse does this sequentially), and (2) avoiding one level of memory indirection by having rank promotion (i.e., Numpy-style broadcasting) that allows elementwise operations between a matrix and a vector in which the vector is replicated along either the row or column direction.

II. Algorithm

Algorithm 1 shows the pseudocode of how each step of the problem is mapped to operations in the GraphBLAST implementation of the GraphBLAS. The implementation in SuiteSparse is similar. The core of this algorithm is the

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

Algorithm 1 Pseudocode of the Sparse Deep Neural Network Graph Challenge's mapping to GraphBLAS.

Inputs:

- Y_0, an MNIST image as a sparse matrix;
- W, a list of sparse weight matrices;
- b, a list of bias vectors;
- TrueCategories, a true category vector; and
- L, the number of layers

Output:

Categories (rows) in the final matrix with entries > 0

▷ Part 1: Evaluate DNN for all layers (timed)
for l from 0 to $L-1$ **do**
▷ Maps to mxm with PlusMultipliesSemiring in GraphBLAST.
$Y_{l+1} \longleftarrow Y_l W_l$
▷ Maps to eWiseMult with plus binary operation.
$Y_{l+1} \longleftarrow Y_{l+1} + b_l$
▷ Maps to eWiseMult with maximum binary operation.
$Y_{l+1} \longleftarrow \text{ReLU}(Y_{l+1})$
▷ (Optional) Filter out zeroes in Matrix Y_{l+1}.
$Y_{l+1} \longleftarrow \text{rebuild}(Y_{l+1}, 0)$
▷ Maps to eWiseMult with minimum binary operation.
$Y_{l+1} \longleftarrow \text{clip}(Y_{l+1}, 32)$
end for

▷ Part 2: Identify categories in final matrix (timed)
▷ Maps to reduce with PlusMonoid.
$C \longleftarrow \text{Rowsum}(Y_L)$
▷ Maps to assign.
Categories $\longleftarrow$ Boolean(C)

▷ Correctness checking (not timed).
Check correctness by comparing Categories with TrueCategories.

multiplication of the sparse inference weight matrix with the input sparse feature matrix.

In Part 1, we map the sparse matrix multiplication of the input matrix and the weight matrix to a matrix multiplication operation. In GraphBLAST, this step is $m \times m$ with the semiring PlusMultipliesSemiring. In SuiteSparse, this semiring is specified with `LAGraph_PLUS_TIMES`. GraphBLAST does not support allows in-place computation (i.e., $Y = YW$), so we use Y for the input matrix and a second matrix Y_{swap} for the output matrix. At each step, after the $m \times m$ operation, we swap Y and Y_{swap}. The next steps are adding a bias and applying a rectifier activation function (a "ReLU"), which we implement as two eWiseMult operations with plus and maximum binary operations. After the ReLU, we have a Matrix::rebuild method that filters out the zeroes from matrix Y. Finally, we clip ReLU values above 32 with another eWiseMult operation using the minimum binary operation.

The result of Part 1 is the matrix Y. We compute the sum of each row of this matrix Y with a reduce operation with the PlusMonoid and store it into a sparse vector C. We then extract the category pattern of C into a Boolean dense vector, where each false entry corresponds to a zero value in C and each true entry corresponds to a non-zero value. This concludes the computation steps; we stop timing at this point, then verify category correctness by extracting tuples of value and index from this dense vector and verify correctness.

III. Experiments

We compare three implementations of the GraphChallenge benchmark:

- A single-threaded CPU MATLAB implementation, running on one core of a 2.2 GHz 20-core Intel Xeon E5-2698 v4 CPU;
- A 32-thread[1] CPU GraphBLAS implementation on SuiteSparse, running on all cores of a 2.2 GHz 20-core Intel Xeon E5-2698 v4 CPU; and
- A GPU GraphBLAS implementation on GraphBLAST, running on an NVIDIA Titan V. The sparse-matrix-times-sparse-matrix kernel in GraphBLAST is currently implemented using NVIDIA's CUDA 10.0 and cuSPARSE (10.0) sparse-matrix library [11].

On all implementations, both the input and output are stored in the memory of the processor that is performing the computation. While it may be argued that the GPU implementation's data should begin and end in the CPU's memory, we submit that it is most likely that an inference operation would be only one stage in a multi-stage pipeline that is increasingly implemented entirely on the GPU (cf. NVIDIA's RAPIDS initiative), and thus our methodology likely represents the common case. We note, however, that GraphBLAST's overall performance would decrease if it included the time to copy input and output data between CPU and GPU.

A. Results

We record runtimes for both the matrix manipulation part (Part 1 of Algorithm 1) and the identification of results greater than zero part (Part 2 of Algorithm 1). Table I contains the results for each implementation and Table II summarizes the rate metric specified by the challenge (inputs $\times$ DNN connections/runtime). The runtimes of Part 1 are at least 3 orders of magnitude greater than Part 2 so we concentrate on Part 1 runtimes in our analysis.

The amount of memory required to store the largest case (65 536 neurons and 1920 layers) does not fit into our GPU's memory and hence our results do not include that case.

As expected, increasing the number of neurons or increasing the number of layers increases the runtime roughly proportionally for all implementations. We observe the following geomean speedups on overall runtime:

- SuiteSparse over MATLAB: 21.84×
- GraphBLAST over MATLAB: 43.32×
- GraphBLAST over SuiteSparse: 1.94×

[1]For SuiteSparse, we ran all thread counts from 1 to 40 and found 32 threads was the fastest.

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

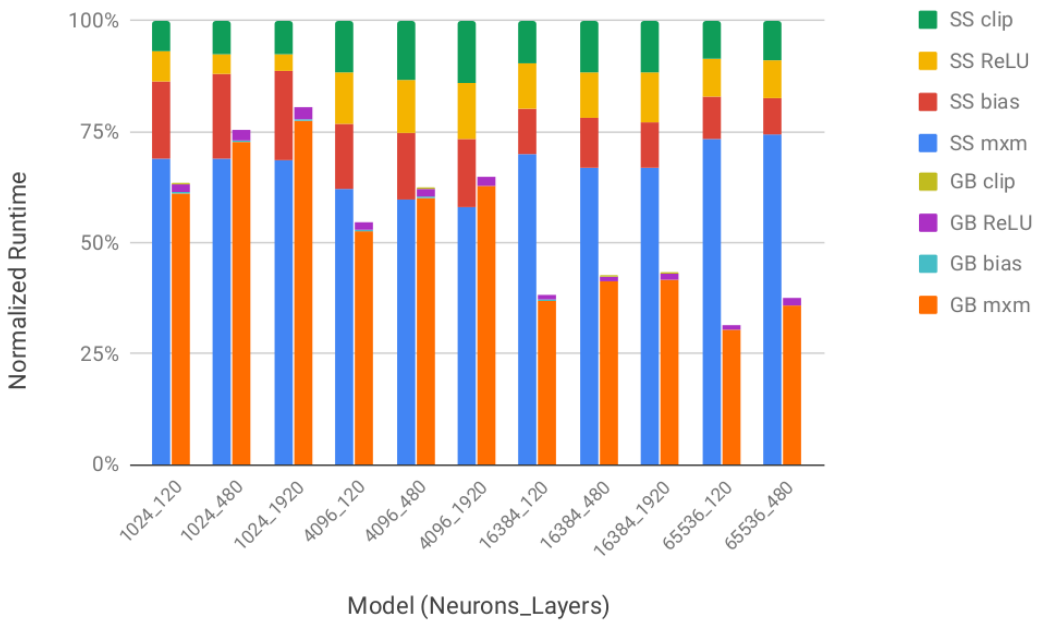

Fig. 1: Normalized runtime (SuiteSparse = 1) on various model sizes using SuiteSparse (SS) and GraphBLAST (GB).

B. Performance Analysis of Each Operation

In terms of the main matrix-matrix operation, we do not see as marked a difference in performance.

a) Multiply with weights: In terms of matrix-matrix multiplication, we use cuSPARSE's "csrgemm2" routine [11]. Compared to SuiteSparse, we are between 1.13× slower to 2.41× faster. In the geomean, we are 1.34× faster. SuiteSparse uses a multithreaded implementation of Gustavson's algorithm [1], [4].

As Figure 1 illustrates, in terms of the non-matrix-matrix multiplication operations, we see a significant 9.79×–23.4× (16.6× geomean) speedup when compared with SuiteSparse. There are several differences between our implementation and SuiteSparse, which are outlined below:

b) Add bias: For adding the bias, SuiteSparse implements this addition as a matrix-matrix multiplication where the activation matrix is multiplied by the bias vector b represented as diagonal matrix $\text{diag}(b)$. However, since SuiteSparse stores sparse matrices in CSC format, this is equivalent as treating the bias vector as a sparse vector in which the CSC *col_ptr* array corresponds to the sparse vector indices and the CSC *values* array corresponds to the sparse vector values. This forces an unnecessary layer of indirection that harms performance. Instead of modeling this addition operation as a matrix-matrix multiplication, we treat it as a GraphBLAS extension method, namely an elementwise multiplication operation between the activation matrix and the bias vector in which the vector is broadcasted in Numpy fashion [12] (or rank-promoted [9]) into a matrix. Since the vector is dense, this allows avoiding one layer of indirection into the vector indices. In terms of adding bias, we attain a 50×–80.7× (59.2× geomean) speedup over SuiteSparse. Even though this method is not currently in the GraphBLAS standard, we provide evidence that Numpy-style broadcasting is both a useful convenience method and important for high performance.

c) Clipping at 32: For clipping at 32, SuiteSparse implements the operation as an apply operation using a user-defined unary operator ymax, which returns 32 if the input is equal or above 32 and returns the input value if below 32. Since this operation is user-defined, it cannot be inlined by the SuiteSparse GraphBLAS shared library. Instead, we opt to use the maximum binary operation together with an elementwise multiplication between a matrix and a scalar value 32 that is broadcasted into a matrix in Numpy fashion. The advantage of such an operation is that by using a standard maximum binary operation, the operation can be inlined in the inner loop. In terms of clipping at 32, we attain a 27.6×–93.6× (62.1× geomean) speedup over SuiteSparse.

d) ReLU and filtering nonzeroes at each layer: For performing the ReLU and filtering nonzeroes out, SuiteSparse uses an extension method GxB_select. What this operation does is allow the user to pass in either a user-defined or predefined SelectOp such as GxT_GT_ZERO. When given an input matrix, this operation will return an output matrix filled with only the input matrix elements that are greater than zero. We do a similar operation called Matrix::rebuild, except we implement this operation in less generality and in parallel. Our rebuild operation takes 3 arguments: input and output Matrix Y, zero element z, and descriptor. It is functionally equivalent to the following two GraphBLAS operations:

1) eWiseMult with equality binary operator, and tests each nonzero of the input matrix for equality with the zero

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

	Layers	Neurons 1024	4096	16384	65536
Part 1	120	1.67632	4.85655	19.1996	79.8156
	480	5.29933	17.6273	73.3141	335.672
	1920	19.9119	68.7398	307.507	
Part 2	120	0.000225136	0.0042545	0.00984867	0.0517773
	480	0.00230621	0.00413536	0.010087	0.0903038
	1920	0.00186198	0.00507398	0.0208721	

(a) GraphBLAST, $W^T Y_0^T$

Neurons 1024	4096	16384	65536
2.25538	16.4389	109.525	3002.55
7.44351	61.1379	430.739	58918.22
28.2826	235.94	3238.34	
0.00030256	0.000316992	0.000494944	0.000179123
0.000295968	0.000306368	0.000494592	0.00531384
0.000303872	0.000307232	0.00105958	

(b) GraphBLAST, $Y_0 W$

	Layers	Neurons 1024	4096	16384	65536
Part 1	120	2.65	8.95	50.03	252.94
	480	7.09	28.21	172.05	891.58
	1920	24.86	106.08	712.79	3577.19
Part 2	120	0.00262717	0.00796638	0.0252706	0.115172
	480	0.00224776	0.00426845	0.0269771	0.106592
	1920	0.00181377	0.0073598	0.0297972	0.117601

(c) SuiteSparse, $W^T Y_0^T$

Neurons 1024	4096	16384	65536
59.5722	243.8983	1034.8135	4470.8053
169.8355	750.7988	3416.2158	15283.4712
602.5248	3254.8743	13867.5775	60059.1867
0.008529	0.015045	0.063369	0.26472
0.004314	0.015823	0.064495	0.27719
0.004242	0.015132	0.061923	0.2704

(d) Matlab

TABLE I: Results, in seconds, for Part 1 and Part 2 runtimes as a function of the number of neurons and the number of layers for GraphBLAST (both YW and $W^T Y^T$), SuiteSparse, and Matlab.

Layers	Neurons 1024	4096	16384	65536
120	1.407×10^{11}	1.943×10^{11}	1.966×10^{11}	1.892×10^{11}
480	1.781×10^{11}	2.141×10^{11}	2.060×10^{11}	1.799×10^{11}
1920	1.896×10^{11}	2.197×10^{11}	1.964×10^{11}	

(a) GraphBLAST, $W^T Y_0^T$

Neurons 1024	4096	16384	65536
1.046×10^{11}	5.741×10^{10}	3.447×10^{10}	5.029×10^{9}
1.268×10^{11}	6.174×10^{10}	3.505×10^{10}	1.025×10^{9}
1.335×10^{11}	6.400×10^{10}	1.865×10^{10}	

(b) GraphBLAST, $Y_0 W$

Layers	Neurons 1024	4096	16384	65536
120	8.903×10^{10}	1.054×10^{11}	7.545×10^{10}	5.970×10^{10}
480	1.331×10^{11}	1.338×10^{11}	8.776×10^{10}	6.774×10^{10}
1920	1.518×10^{11}	1.423×10^{11}	8.473×10^{10}	6.754×10^{10}

(c) SuiteSparse, $W^T Y_0^T$

Neurons 1024	4096	16384	65536
3.960×10^{9}	3.869×10^{9}	3.648×10^{9}	3.377×10^{9}
5.557×10^{9}	5.028×10^{9}	4.420×10^{9}	3.952×10^{9}
6.265×10^{9}	4.639×10^{9}	4.355×10^{9}	4.023×10^{9}

(d) Matlab

TABLE II: Rate (inputs $\times$ DNN connections/runtime) for GraphBLAST (both YW and $W^T Y^T$), SuiteSparse, and Matlab.

element z. In this case, the zero element used is 0. Call the temporary result Boolean matrix temp.

2) apply using structural complement of temp as mask, the identity unary operator, and matrix Y as the input and output.

In terms of implementation, our Matrix::rebuild operation is composed of the following GPU kernels. It can be thought of as an optimization of the above GraphBLAS operations when apply's unary operator is identity.

1) Flag array: Each thread writes 1 to the flag array if CSR *values* array element equal to identity, otherwise 0.
2) Segmented reduce: Run on flag output using CSR *row_ptr* as segments, generates number of nonzeroes in each row.
3) Prefix sum: Run on number of nonzeroes in each row, generates new CSR *row_ptr*.
4) Prefix sum: Run on flag array, generates indices to which we need to scatter.
5) Stream compact: Each thread scatters to its index if the flag output is equal to 1, else do nothing. This generates the new CSR *col_ind* and *values* arrays.

On ReLU and filtering out nonzeroes, we attain a 1.56–9.55$\times$ (5.44$\times$ geomean) speedup over SuiteSparse.

C. Lessons Learned

a) Choice of YW or $W^T Y^T$ greatly impacts performance: We can choose to implement the matrix multiplication by performing either $Y_1 = Y_0 W + b$ or the transposed variant $Y_1^T = W^T Y_0^T + b^T$. As Figure 2 shows, this dataset produces a lot of load imbalance in matrix Y_0 (in blue) and consequentially, the activation matrix that is formed. On the other hand, W^T (shown in red) is perfectly load-balanced, because it has 32 nonzeroes in every row. Therefore one optimization we can make is that if we use W^T as the lefthand

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

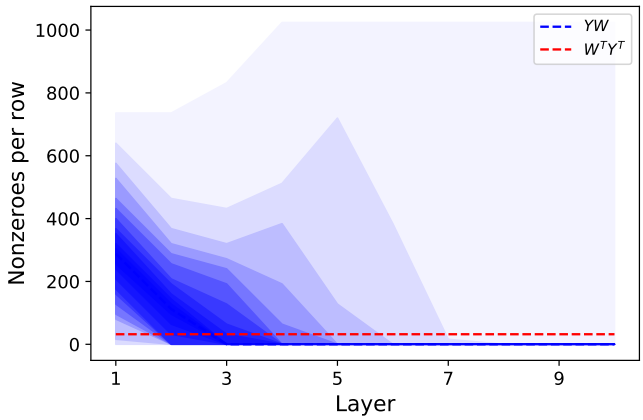

Fig. 2: Nonzeroes per row in the activation matrix in each layer of a 1024-neuron, 120-layer neural network with multiplications Y_iW and W^TY_i respectively. Darker shades of blue indicate nonzeroes per row closer to median, while lighter shades indicate nonzeroes per row at the 10th and 90th percentile.

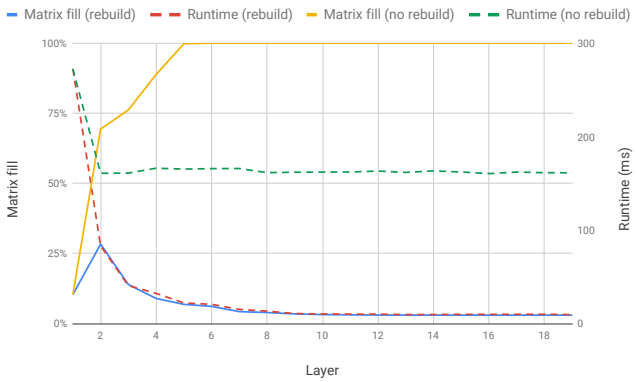

Fig. 3: Matrix fill ratio $\frac{nnz}{mn}$, where nnz is the number of nonzeroes in the $m \times n$ activation matrix Y_l after layer l; and matrix-matrix multiplication runtime in milliseconds after each layer of a 1024-neuron, 120-layer neural network with and without filtering out zeroes using Matrix::rebuild. Note that without filtering, all matrix entries rapidly become nonzero and the runtime is much higher than the filtered case as a result.

side matrix ($Y_1^T = W^TY_0^T + b^T$), we can ensure that W^T as the left matrix in matrix-matrix multiplication of every layer.

When both matrices are stored in CSR format, the performance of matrix multiplication is in large part driven by load imbalances imposed by the structure of the lefthand side matrix. In the GraphChallenge problem, weight matrix W^T always has exactly 32 nonzeroes per row, so we see no load imbalance and hence have better performance when we use W^T on the left. The geomean speedup of W^T over Y across all neuron/layer combinations is 5.80×, with speedups increasing with larger neuron count (peak speedup is 175.5× for the 65 536-neuron-480-layer case). If the weight matrices had more variability with the number of nonzeroes per row, the performance gap between having Y and W on the left would narrow. Although SuiteSparse uses the CSC storage format in which the righthand side matrix is the key determinant of load-balancing, they perform $Y_1 = Y_0W + b$. Functionally, this is equivalent as multiplying $Y_1^T = W^TY_0^T + b^T$ in CSR storage, so we speculate they are doing so for load-balancing reasons as well.

b) Filtering nonzeroes at each layer: At the end of each layer, the resulting matrix may have numerous zeroes. We can choose to leave that matrix unchanged (and pay the extra compute cost of computing on zeroes in the next layer) or run a filter step at the end of each layer's computation that removes all nonzeroes. We find that the filter step results in considerably higher performance (on one of our experiments, for instance, it reduces overall runtime from 40 s to 2.25 s).

Figure 3 illustrates the impact of filtering out zeroes. Without filtering out the nonzeroes, the activation matrix Y_l becomes dense after layer 4. However, if the zero entries are filtered out during each layer, the activation matrix is kept sparse and converges to 3% matrix fill.

c) Pruning neurons: We observe that the values of nonzeros in weights are all 0.0625 for all layers. We also observe that the number of nonzeroes stops changing after early layers, e.g., in the 1024-neuron, 120-layer case the number of nonzeroes remains 1855488 since the 29th layer. This inspires us that pruning the last layers might possibly speedup the inference hurt the correctness of the results. The following pruning scheme has been tried between the ReLU step and the rebuild step: an amount of 40% of the number of nonzeroes of random indices are generated using the cuRAND library and values corresponding to these indices in the Y matrix after layer 80 are zeroed out. We achieve 1.03× speedup with 1024-neuron, 120-layer and 1.10× with 16384-neuron, 1920-layer. However, the feasibility of this approach is due to the special characteristic of the given dataset. Without further improvements this approach may not generalize well in other similar contexts, which is beyond the scope of this paper, and thus we decide not to present the results with pruning here.

d) Running larger datasets: The 65 536-neuron-1920-layer case requires 38 GB of storage, which significantly exceeds the 12 GB of DRAM on our Titan V. Scaling to such a large dataset would require a different approach, almost certainly exploiting model parallelism. Possibilities include (a) loading a different subset of weights to the GPU, analogous to an out-of-memory graph framework, or (b) sending the intermediate computation to another GPU that holds a different subset of layers, analogous to a multi-GPU graph framework. We leave addressing this problem as future work.

e) Impact on GraphBLAS API: We implemented the following extension to the GraphBLAS API, which the GraphBLAS community may wish to consider for further study and possible additions to the standard. **Rank promotion**

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

(Numpy-style broadcasting): We see significant speed-up of 9.79×–23.4× (16.6× geomean) by using rank promotion on elementwise operation to avoid the use of user-defined unary operators and a layer of indirect memory access when doing elementwise multiply between matrix and vector instead of doing matrix-matrix multiplication with a diagonal matrix. In addition, the Numpy-style broadcasting may be more natural to Python users than needing to diagonalize a matrix in order to do an elementwise multiply. It may be an important addition in terms of convenience and performance to the GraphBLAS specification.

IV. Conclusion

In this work we have demonstrated a high-performance implementation of the 2019 Sparse Deep Neural Network Graph Challenge using a GPU implementation of the GraphBLAS standard. While our implementation shows a 1.94× speedup over the "SuiteSparse" CPU implementation of GraphBLAS, the most important kernel in any implementation of this challenge will be the SpGEMM operation, and we only show a 1.34× speedup over SuiteSparse on this kernel. In its marketing materials, cuSPARSE claims a 2–5× speedup over CPU competitors, and the raw computational and memory throughput of a GPU has a similar multiple over the CPU, so we believe this kernel represents the most significant opportunity to improve GPU performance. Recent GPU library implementations, including bhSPARSE [7], nsparse [10], and RMerge2 [3], have demonstrated significant speedups over cuSPARSE, and may be well-suited for the matrix operations we require in this challenge. cuSPARSE has the unenviable task of running effectively on any sparse matrix and thus its developers may have concentrated more on generality than performance. Nonetheless we hope that a future version of GraphBLAST—one that either implements its own kernels, that leverages other research libraries, or that incorporates an improved cuSPARSE—may be able to deliver higher performance in the future without any changes to the implementation of this graph challenge.

In terms of future work, we note that due to GPU memory limitations, we were not able to run the 65 536-neuron-1920-layer model, which would have required an estimated 38 GB memory to run while the Titan V GPU we had access to only has 32 GB memory. In order to run larger sparse neural networks on GPUs, we will need to implement model parallelism, which would be interesting to address within the GraphBLAS specification. In this instance, the memory consumption is largely taken up by the 1920 layers, each having dimension $65\,536 \times 65\,536$ with ∼2M nonzeroes. If each layer were instead divided amongst 4 GPUs (e.g., layers 1–480 on GPU0, 481–960 on GPU1, 961–1440 on GPU2, 1441–1920 on GPU3), then each GPU could do local computation while only needing to communicate activations across GPUs at layer boundaries 480, 960 and 1440. Ideally, this can be combined with data parallelism in order to optimize the matrix dimensions for performance [2], [13].

V. Acknowledgments

We appreciate the funding support from the Defense Advanced Research Projects Agency (Awards # FA8650-18-2-7835 and HR0011-18-3-0007) and the National Science Foundation (Awards # OAC-1740333 and CCF-1629657). This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

[1] Timothy A. Davis. SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra. Submitted to *ACM Transactions on Mathematical Software (TOMS)*, 2018. http://faculty.cse.tamu.edu/davis/GraphBLAS_files/toms_graphblas.pdf. Accessed: 2019-05-01.

[2] Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydin Buluc. Integrated model, batch, and domain parallelism in training neural networks. In *Proceedings of the 30th Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 77–86. ACM, July 2018.

[3] Felix Gremse, Kerstin Küpper, and Uwe Naumann. Memory-efficient sparse matrix-matrix multiplication by row merging on many-core architectures. *SIAM Journal on Scientific Computing*, 40(4):C429–C449, January 2018.

[4] Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software*, 4(3):250–269, September 1978.

[5] Jeremy Kepner, Peter Aaltonen, David Bader, Aydın Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Jose Moreira, John D. Owens, Carl Yang, Marcin Zalewski, and Timothy Mattson. Mathematical foundations of the GraphBLAS. In *Proceedings of the IEEE High Performance Extreme Computing Conference*, HPEC 2016, September 2016.

[6] Jeremy Kepner, Manoj Kumar, José E. Moreira, Pratap Pattnaik, Mauricio J. Serrano, and Henry M. Tufo. Enabling massive deep neural networks with the GraphBLAS. In *Proceedings of the IEEE High Performance Extreme Computing Conference*, HPEC 2017, September 2017.

[7] Weifeng Liu and Brian Vinter. An efficient GPU general sparse matrix-matrix multiplication for irregular data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014.

[8] Tim Mattson, Timothy A. Davis, Manoj Kumar, Aydın Buluç, Scott McMillan, Jose Moreira, and Carl Yang. LAGraph: A community effort to collect graph algorithms built on top of the GraphBLAS. In *Proceedings of the Workshop on Graphs, Architectures, Programming, and Learning*, GrAPL 2019, May 2019.

[9] Timothy G Mattson, Carl Yang, Scott McMillan, Aydin Buluç, and José E Moreira. GraphBLAS C API: Ideas for future versions of the specification. In *IEEE High Performance Extreme Computing Conference*, HPEC 2017, pages 1–6, 2017.

[10] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. High-performance and memory-saving sparse general matrix-matrix multiplication for NVIDIA Pascal GPU. In *2017 46th International Conference on Parallel Processing*, ICPP-2017, August 2017.

[11] M. Naumov, L. S. Chien, P. Vandermersch, and U. Kapasi. CUSPARSE library: A set of basic linear algebra subroutines for sparse matrices. In *GPU Technology Conference*, GTC 2010, 2010. http://on-demand.gputechconf.com/gtc/2010/presentations/S12070-Cusparse-Library-a-Set-of-Basic-Linear-Algebra-Subroutines-for-Sparse-Matrices.pdf.

[12] Stefan Van Der Walt, S. Chris Colbert, and Gael Varoquaux. The NumPy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, March/April 2011.

[13] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference*, EuroSys '19, pages 26:1–26:17, March 2019.

[14] Carl Yang, Aydın Buluç, and John D. Owens. Graphblast: A high-performance linear algebra-based graph framework on the gpu. 2019.

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