Performance Optimization of SpMV on Spark

1st Kun Xie
Dept. Computer Science
National Tsing Hua University
HsinChu, Taiwan
joseph520521@gmail.com

2nd Che-Rung Lee
Dept. Computer Science
National Tsing Hua University
HsinChu, Taiwan
cherung@cs.nthu.edu.tw

3rd Feng-Yuan Liu
Dept. Computer Science
National Tsing Hua University
HsinChu, Taiwan
smilepa3034@gmail.com

***Abstract*—Sparse matrix-vector multiplication (SpMV) is one of the most important computational kernels in solving large scale numerical problems for scientific computing, data analysis, machine learning, and many others. However, its performance optimization on various platforms remains a research problem owing to the diversity of matrix structures and architectural properties. In this paper, we present two performance optimization methods for SpMV on Spark. First, we proposed a new data format, called Block COO plus (BCOO+), which can significantly reduce the number of shuffles in Spark. Second, we designed a new deep convolutional neural network (CNN) to analyze the matrix structure, and automatically choose the right data format for SpMV on Spark. The experimental results show that our method can achieve 3.2 times performance improvement comparing to traditional CSC format.**

***Index Terms*—Spark, Matrix vector multiplication, performance optimization, deep convolution neural network, data structure**

I. Introduction

Sparse matrix-vector multiplication (SpMV) is one of the most important computational kernels in solving large scale numerical problems for scientific computing, data analysis, machine learning, and many others. The reason is for large scale problems, iterative algorithms, such as page-rank or Lanczos method, are more computationally effective, and their major computation cost is the matrix vector multiplication [1].

There are bunch of papers on accelerating sparse matrix vector multiplication on various platforms [2], [3]. Traditionally, the right data format for sparse matrix storage and computation is the key to accelerate the SpMV operation. Several classical data formats, such as Coordinate Format (COO), Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), and Block Compressed Sparse Row (BCSR), have been widely used in high performance computing applications [1]. New data formats that target specific computation devices, such as GPU or Intel MIC, are also introduced for performance enhancement [4]–[6]. Lately, new hybrid formats that combine different data structures [7]–[9] and adaptive methods [10]–[13] that dynamically change the data formats for target matrices and platforms were proposed. Recently, machine learning based methods are receiving increasing number of investigations [14]–[18]. The late success of deep learning techniques also were also utilized in solving this problem [19]–[21].

However, most of researches on the acceleration of SpMV focus on high performance devices, such as GPU. For big data processing systems, like Spark [22], the problem hasn't been well studied. And the results of previous studies cannot be applied directly because of their architectural differences. Nevertheless, this is an important problem. Many big data processing tools and libraries, such as Spark SQL [23], MLlib [24], and GraphX [25], were built on Spark, and many of them require the efficient implementation of SpMV. Take MLlib as an example. It provides four data formats for matrices: dense matrix, sparse matrix, row matrix, and block matrix. However, all of them are stored in the dense matrix format. In our preliminary comparison, the performance of those formats is much worse than COO and CSR.

In this paper, we proposed two methods to improve the performance of SpMV on Spark. The first one is a new data format, called Block COO plus (BCOO+), which can significantly reduce the number of shuffles in Spark, comparing to other block formats. Although BCOO+ performs well in most cases, it is not perfect for all the matrices. The variations of matrix structures and Spark node configurations may favor other kinds of data formats for sparse matrices. So our second method is an format selection algorithm, which utilize a deep learning model to classify the matrices based on their structures and the running configurations. Our algorithms are different from the previous studies of SpMV acceleration, because the architecture and performance bottleneck of Spark are not the same as GPU or other HPC devices.

Our deep learning model dose not only recognize matrix structures, but also the execution configuration and other matrix parameters. The image recognition is based on a convolution neural network (CNN), fitNet-4 [26]. The configuration data is recognized by a single layer of fully connected network. To combine those two submodels, we add few more layers to mixed their outputs. Several fine tuning techniques have been also performed to improve the accuracy of the model. We used the matrices from Florida Sparse matrix collection for training and testing. Based on the density of block submatrices, all matrices are summarized as 32x32 images to capture their structure. The results show that our model can achieve 81% accuracy to find the best data format, and 94% to find the top two data formats. The performance of SpMV can be improved 3.2 time of our adaptive method, comparing to single CSC format.

The rest of paper is organized as follows. Section II presents our algorithms, model, and implementations. Section III shows the experimental results. Section **??** gives a brief literature review on the performance enhancement for SpMV on various platforms. Conclusion and future work are given the last section.

II. Algorithms and Implementations

A. Block Coordinate Plus Format (BCOO+)

A matrix is called sparse as long as it can take advantages, computation or storage-wise, from its zero elements. The sparse matrix-vector multiplication (SMV) can be represented as $y = Ax$, where A is $M \times N$ matrix, x is a vector of length N, and y is a vector of length M. A parameter, the number of non-zeros N_{nz}, is also important to SpMV because most data structures only store non zero elements.

There are four commonly used data structures for sparse matrix representation: Coordinate Format (COO), Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), and Block Compressed Sparse Row (BCSR). The COO format stores a tuple (row index, column index, and value) for each non-zero value. Normally, COO stores three arrays: a row index array R, a column index array C, and a value array V. The Compressed Sparse Row (CSR) format also uses three arrays to store the non-zeros, but the row array were compressed into the compressed row array R_c. The Compressed Sparse Column (CSC) is the same as CSR, except it compresses the column array. The Block Compressed Sparse Row (BCSR) format takes advantage of the clumpy structures of sparse matrices. The matrix is partitioned in small dense sub-blocks with a block size. It stores the row and column indices of the top left of the block and stores the values of the block in the row major order. This format could store extra zero elements to maintain the block structure. However, the continuous storage could leverage the cache effect and reduce the latency of data movement, and therefore improve the performance if the number of extra zeros is few.

We propose a new sparse data format BCOO+ for SpMV on Spark to improve the performance of SpMV on Spark. Our designs are based the empirical results, which will be shown first. We randomly select ten matrices, whose number of rows and number of columns are larger than 5000, from the University of Florida Sparse Matrix Collection [27]. We implemented three commonly used data structures for sparse matrices: COO, CSR, and CSC.

The Figure1 shows the count of the best data format among ten randomly selected benchmarks matrices and six node configurations, which are the combinations of two memory sizes and three iteration numbers. The result shows that the performance of data formats could be different for various matrices and configurations.

Hence, it is expected that the CSR could get better performance when the number of iteration grows because of the cache mechanism in Spark at the multiplication step. Moreover, the COO could have better performance in small number of iterations, because its initial step is the fastest. Nevertheless, the CSC could have the best performance for some matrices.

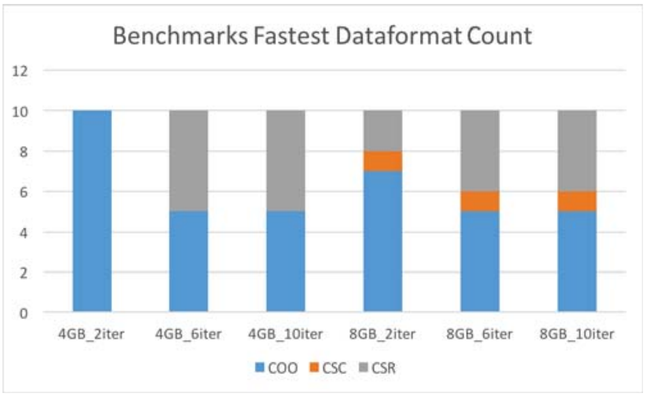


Fig. 1. The Count of the best data format for ten benchmark matrices.

Algorithm 1 SpMV implementation of BCOO+ on Spark

```
Require: Hadoop NLineInputFormat — Split to different partitions by block size;
1: Split the input matrix to $(row, (col, value))$ tuple format;
2: Glom — Group each partition to block arrays as RDD $M$;
3: Cache the $M$ RDD;
4: Broadcast the initial HashMap $N$ to all slaves;
5: for $iter = 1; iter < iterationNum; iter++$ do
6:     $flatMap$ Function:
7:     for each element in iterator of $(row, (col, value))$ do
8:         Get the $value$ from vector HashMap $N$ by $col$;
9:         Do multiplication and add result to HashMap by key — $row$;
10:    end for
11:    $flatMap$ Output iterator $(row_i, subSum_i)$ converted from HashMap;
12:    ReduceByKey — Sum all sub-summation of the same $row$ from different partition;
13:    CollectAsMap — Collect the RDD to HashMap $N$;
14:    Broadcast the HashMap $N$ to all slaves;
15: end for
```

According to the result of preliminary test, we design a new data structure, called Block Coordinate Plus format (BCOO+), specifically for the architectural properties of Spark. We have observed two performance bottlenecks from the result. The first one is the action of shuffle in the initial step, and the second one is the number of RDDs in the multiplication step. Our strategy is based on the implementation of COO, which need not be shuffled in the initial step, and improve its performance in the multiplication step.

Given a matrix A, it is first converted into the COO format, $(row, (col, val))$ without any ordering constrains. Then those tuples are partitioned and grouped into blocks of the same size. Each block is then stored into one RDD. Note such partition need not base on the structure of the matrices so that no shuffles are required in the initial phase. However, if the input matrix is ordered by row, BCOO+ can get better performance. The main reason of grouping is to avoid the

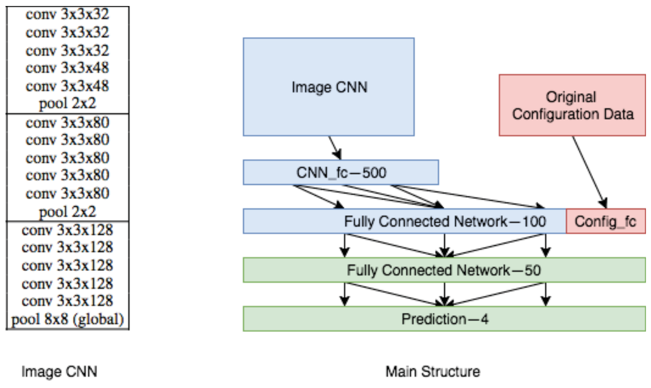


Fig. 2. The network architecture of the fine-tuned model.

shuffle in the initial step. Moreover, it can use `combiner` to sum the multiplication result in different partitions, which could also improve the performance. Algorithm 1 sketches the implementation of SpMV for the BCOO+ format on Spark.

B. Deep Learning Format Selector

Although BCOO+ outperforms other data formats in most cases, it is not the winner of all. To further improve the performance, we proposed an adaptive method to select the best format based on the matrix structure and system configurations. Thus, the input is the matrix structure and the system configuration, and the output the best data format in terms of performance.

Our algorithm utilizes a deep learning model, which considers the matrix structure and the machine configurations. It consists of three sub-networks: a network for matrix structure, a network for configuration data, and a network to combine the results from two other networks. The matrix structure is analyzed by the convolution neural network (CNN), the most popular deep learning model for the image recognition [28]. First, each matrix is represented as a 32×32 image for input data, whose details will be presented in the next section. Then the network of Fitnet4-LSUV [26] is used to classify the image data. For the configuration data, we also added some information about the matrix, including the number of columns and the number of non-zeros. Since the configuration part only has a few features, it cannot be a hierarchical network similar to CNN. So we applied fully connected network for the configuration data. However, due to the limited configuration data features, the network should not be too deep. Empirical result shows that one single fully connected layer of size 25 can achieve the best accuracy.

The third network that combines the results from image data and configuration is a fully connected network. But before the combination, we found that some fully connected layers between the image CNN and the combination network can improve the overall accuracy. Table I lists different combinations of dense networks appending the image CNN, where $\star$ means the concatenate layer. The best result is adding two fully connected layers of size 500 and 100 after the image CNN, and using a fully connected layer of size 50 as the combination network. The detail architectural design is shown in Figure 2.

C. Data Preprocessing and Model Training

1) Matrix Data: We obtained the data set from the SuiteSparse Matrix Collection [27], which is a large and actively growing set of sparse matrices from real applications. There are 2,047 sparse matrices in the collection. Three different data format implementations on Spark, COO, CSR, and CSC, are evaluated. In the collection, each matrix has a $308 * 308$ image to represent it. The image size is too large to use so we re-sized them to 32×32 using linear scaling. If a matrix is not square, the image will be aligned to the top left, and zeros will be padded to the rest area.

2) System Configuration Data: As the result of preliminary testing shown in Figure 1, the best data format does not only rely on the matrix structure, but also the system configuration. We have identify two of the most important parameters, which are (1) the number of iterations for SpMV, and (2) the memory size. We performed a similar evaluation as the preliminary test, but this time we used all 2047 matrices, and put the BCOO+ format for evaluation. We say a data format is the fastest if its execution time is the shortest comparing to others.

Figure 3 shows the counts of the fastest data format for different numbers of iterations and different memory sizes respectively. As can be seen, BCOO+ is the fastest data format in most cases. However, there are considerable cases that COO and CSR win the testing. And there also few matrices favoring the CSC format. A more important observation is that for different system configuration, the execution time may be varied a lot. That is the major reason we need to put those two system configurations into account.

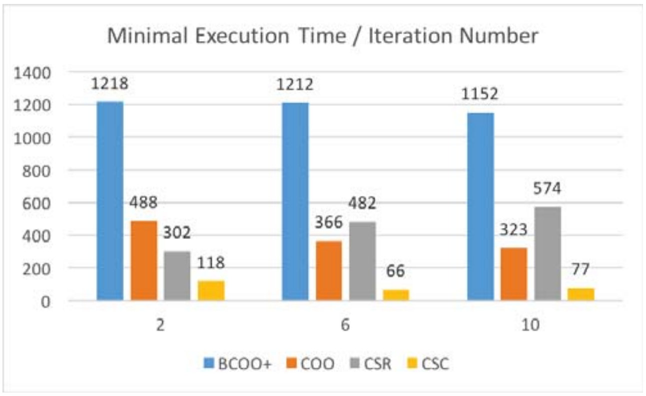
III. Experiments and Results

This section introduces two sets of experiments and their results. The first set of experiments evaluate the performance of BCOO+. The second set of experiments show the effectiveness of the format selection algorithm.

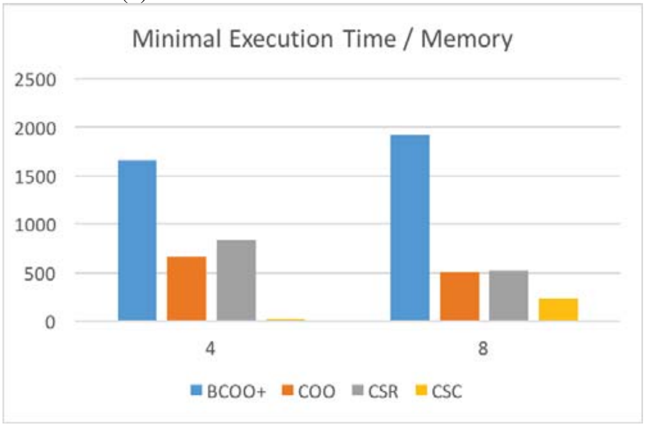
We run the Spark programs on a cluster, which has seven physical nodes. Each physical node equips with two Intel Xeon CPU E5-2670 v2 (10 cores) for 2.50 GHz , 252 GB memory, and the OS is CentOS Linux release 7.3.1611 (Core). The Hadoop version is 2.7.3. We used the Hadoop Distributed File System (HDFS) for the Spark implementations. The Spark

TABLE I
The accuracy of prediction for different network designs.

Dense Network	Accuracy
(BaseLine) fc_500($\star$) $\rightarrow$ fc_25	74%
fc_500 $\rightarrow$ fc_100($\star$) $\rightarrow$ fc_25	81%
fc_500 $\rightarrow$ fc_100($\star$) $\rightarrow$ fc_50	82%
fc_500 $\rightarrow$ fc_200($\star$) $\rightarrow$ fc_25	80%
fc_500 $\rightarrow$ fc_200($\star$) $\rightarrow$ fc_50	80%
fc_500 $\rightarrow$ fc_300($\star$) $\rightarrow$ fc_25	80%
fc_500 $\rightarrow$ fc_300($\star$) $\rightarrow$ fc_50	82%
fc_500 $\rightarrow$ fc_200($\star$) $\rightarrow$ fc_50 $\rightarrow$ fc_25	79%
fc_500 $\rightarrow$ fc_200($\star$) $\rightarrow$ fc_100 $\rightarrow$ fc_25	79%



(a) for different numbers of iterations.



(b) for different memory sizes.

Fig. 3. The count of the fastest data format.

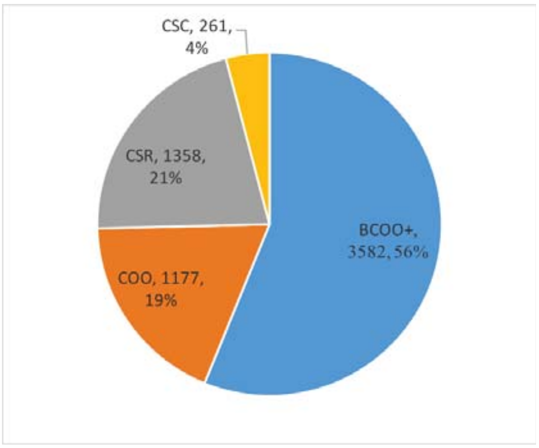


Fig. 4. The fastest data format for different matrices and configurations.

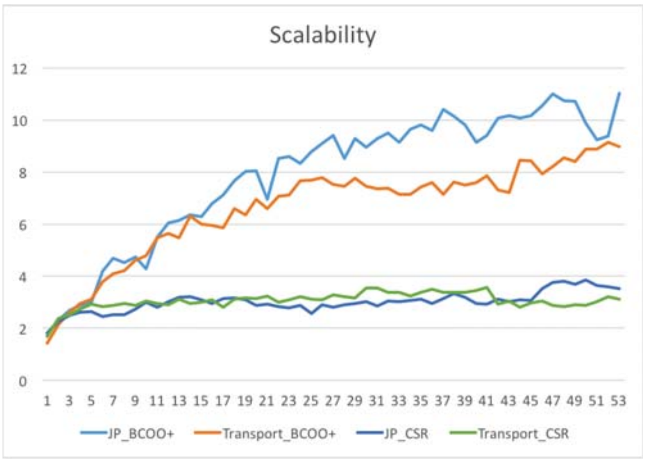


Fig. 5. The Speedup of BCOO+ format

version is 2.0.1. The deploy mode of spark implementations is standalone mode, which means managing the resources by spark standalone master. The python version is Python 2.7.12. We used the Keras framework, version 2.0.3, to design the deep learning model, and used Tensorflow as the back-end of the Keras. In addition, the CUDA version is 9.0.176 and the CUDNN version is 7.0.3.

The data-set is described in Section II-C1. However, we only targeted on integer and real number matrices, and the matrix column size is larger than five thousands. Therefore, only 1,063 of them were selected.

A. *Performance of BCOO+*

We run the experiments on all combinations of matrices and configurations, and labeled the fastest data format for each test case. There are four classes: BCOO+, COO, CSR, and CSC. Each test case is executed five times and their average is used. Figure 4 shows the statistics.

The second experiment evaluates the scalability of BCOO+. Two square matrices, JP and Transport, are used for the evaluation. Matrix JP has 67,320 rows/columns and 13,734,559 non-zeros; matrix Transport has 1,602,111 rows/columns and 23,487,281 non-zeros. The memory used is 8GB, and the number of iterations is 6. We used virtual machines for Spark execution. Each VM has 8GB memory and one core of CPU.

We compare the scalability of BCOO+ and CSR. Figure 5 shows the result of speedup from 1 core to 54 cores. The speedup is computed based on the 1 core CSR format. As can be seen, as the number of cores increases, the performance of BCOO+ grows linearly before the number of cores equal to 20. After that, the performance improvement is slowdown. On the other hand, the scalability of CSR is poor, which stagnates after the number of cores is larger than 5.

The third experiment investigates the influence of the block size in BCOO+. The block size means the size of each partition in BCOO+. We examined all the 1,063 matrices and 6 configurations (total 6,378 cases). Each case runs eight different block sizes: 50K, 100K, 500K, and 1M, 20K, 80K, 120K, and 1.28M. The experiment runs on 8 nodes with 8GB memory for 6 iterations. For each case, the block size of the best performance is recorded.

Figure 6 shows the histogram of the eight different block sizes, which shows no clear trend. The reason is the best block size depends not only the number of non-zeros in a matrix,

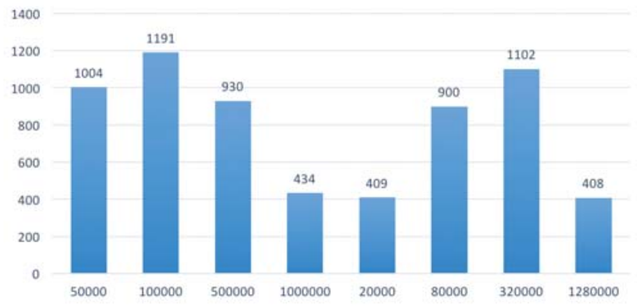


Fig. 6. The counts of the best block-size for BCOO+.

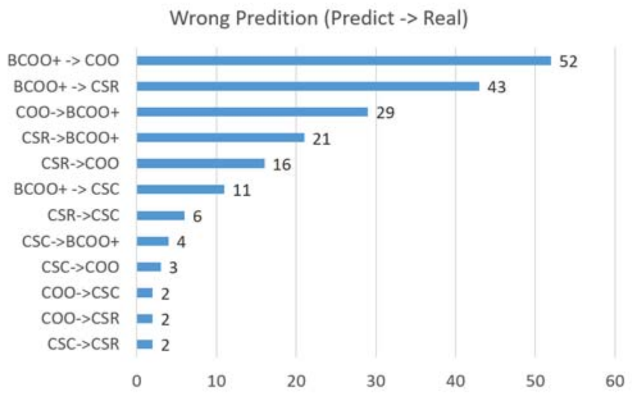


Fig. 7. The analysis of wrong prediction.

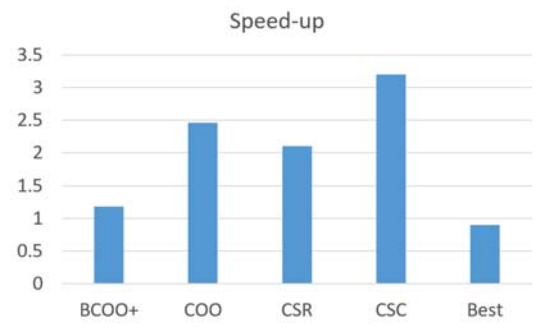


Fig. 8. The speedup of the adaptive method.

but also the number of nodes for Spark to execution. If the block size is too small, the number of RDDs will be large, which causes a significant overhead for aggregation, as COO does. On the other hand, a large block size can cause the load imbalance problem, because the number of RDD is small, and the effect of parallelization is diminished. Thus, a large block size can also damage the performance. Empirically, we found the number of RDDs per node should be in a proper range to achieve the best performance.

B. Deep Learning Model for Format Selection

There are 6,372 cases from the all combination of 1,063 sparse matrices and 6 system configurations. We selected 1/6 of the cases for testing and 5/6 for training. The order of training cases are shuffled. Given a matrix and a system configuration, including memory size and number of iterations, the model outputs the data format that has the best performance from four categories: BCOO+, COO, CSR, and CSC. Since the number of each output category is imbalance, as shown in Figure 4, our selection of test cases, although is random, also follows the same distribution.

The top-1 accuracy of our model is 82%. More specifically, it makes 872 correct predictions out of 1,063 cases. The top-2 accuracy is about 94%, which means if our model can give two recommendation in each case, there are 999 cases, out of 1,063, whose fastest data format will be one of those two recommendations.

Figure 7 enumerates the all the incorrect top-1 prediction, sorted by their counts. As can be seen, the most often mis-predicted cases are the prediction of BCOO+, but the real winner are COO or CSR. The third and fourth mis-predicted cases are vice-versa: the prediction are COO or CSR, but the actual winner is BCOO+. The reason is the outputs of those three categories are over 95%. Therefore their change of mis-prediction are higher than other cases.

We further analyzed the execution time of the recommended format and that of the correct answer. We found that over 90% (175/192) of the cases, the relative differences between their execution time, as defined below, are within 3%,

$$\text{Relative Difference} = \frac{T_P - T_R}{T_R}, \tag{1}$$

where T_P is the execution time for the recommended format, and T_R is the time for the fastest one. In other word, even though our model cannot make a correct prediction, it still gives a good recommendation in most cases.

The last experiment evaluates the performance improvement made by using our adaptive method. We run the SpMV of all the test matrices for six iterations on seven nodes, and each node has 8GB memory. We compared the adaptive method with single format strategy, and measured the speedup made by the adaptive method. For example, if the time to run all the 1,063 test cases by using BCOO+ is T_B and the time by using the adaptive method is T_A, the speedup is defined as

$$T_B/T_A. \tag{2}$$

Figure 8 displays the speedups made by the adaptive method comparing to the strategy of using single data format for BCOO+, COO, CSC, and CSR. We also added another comparison for the adaptive method and the optimal, which means every test case uses its best data format.

The result shows our adaptive method can improve the performance up to 3.2 times comparing the single data format of CSC. And comparing to our BCOO+ format, the adaptive method can also enhance its performance by about 1.2 times. And our adaptive method can also achieve over 90% of the optimal performance.

IV. Conclusion and Future Work

In this paper, we presented two performance optimization methods for improve the performance of SpMV on Spark. The first one is a new data structure for sparse matrices, called BCOO+; and the second one is a format selection algorithm to choose the right data format based on matrix structure and the configuration of execution. The format selector utilizes a deep learning model to pick the data format. Our results show our deep learning model can achieve 82% accuracy for top 1 format, and 94% for top 2. With our methods, SpMV can be accelerated 3.2 times comparing to tradition CSC data format.

This is a new research direction that combines big data processing system, high performance computing, and artificial intelligence. We do realize that more environmental parameters, such as number of nodes, execution policy, should be put into consideration, and more AI models could be tried. However, we do provide a good foundation for the study in this direction. We collected all the performance data and will release to public. For the future work, we need to put the optimizer for some real applications, in which the number of iterations may not know. Second, we found the 32x32 images may be too small. But large images could cause the difficulties in training. We will try to extract features from the matrix structure, or find a more economic method to represent the matrix structures. Last, as mentioned, more numerical kernels, besides SpMV, would be studied to optimize their performance on Spark or other big data processing systems.

References

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[2] R. W. Vuduc, "Automatic performance tuning of sparse matrix kernels," Ph.D. dissertation, 2003, aAI3121741.

[3] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on GPGPUs," *ACM Trans. Math. Softw.*, vol. 43, no. 4, pp. 30:1–30:49, Jan. 2017.

[4] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "CSX: An extended compression format for spmv on shared memory systems," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 247–256.

[5] R. Kannan, "Efficient sparse matrix multiple-vector multiplication using a bitmapped format," in *20th Annual International Conference on High Performance Computing*, Dec 2013, pp. 286–294.

[6] Y. Zhang, S. Li, S. Yan, and H. Zhou, "A cross-platform spmv framework on many-core architectures," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, pp. 33:1–33:25, Oct. 2016.

[7] N. Bell and M. Garl, "Efficient sparse matrix-vector multiplication on cuda," NVIDIA Corporation, Tech. Rep. NVIDIA Technical Report, NVR-2008-004, 2008.

[8] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Nov 2009, pp. 1–11.

[9] Z. Deng, "Sparse matrix-vector multiplication on Nvidia GPU," *International Journal of Numerical Analysis Modeling Series B*, vol. 3, no. 2, pp. 185–191, 2012.

[10] P. Guo, L. Wang, and P. Chen, "A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, pp. 1112–1123, May 2014.

[11] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2014, pp. 769–780.

[12] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaSpMV: Yet another SpMV framework on GPUs," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 107–118.

[13] P. Zardoshti, F. Khunjush, and H. Sarbazi-Azad, "Adaptive sparse matrix representation for efficient matrix–vector multiplication," *The Journal of Supercomputing*, vol. 72, no. 9, pp. 3366–3386, Sep 2016.

[14] J. Li, G. Tan, M. Chen, and N. Sun, "SMAT: An input adaptive autotuner for sparse matrix-vector multiplication," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 117–126.

[15] A. Benatia, W. Ji, Y. Wang, and F. Shi, "Machine learning approach for the predicting performance of SpMV on GPU," in *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, Dec 2016, pp. 894–901.

[16] I. Nisa, C. Siegel, A. S. Rajam, A. Vishnu, and P. Sadayappan, "Effective machine learning based format selection and performance modeling for SpMV on GPUs," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2018, pp. 1056–1065.

[17] S. Chen, J. Fang, D. Chen, C. Xu, and Z. Wang, "Adaptive optimization of sparse matrix-vector multiplication on emerging many-core architectures," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, June 2018, pp. 649–658.

[18] S. Usman, R. Mehmood, I. Katib, and A. Albeshri, "ZAKI+: A machine learning based process mapping tool for spmv computations on distributed memory architectures," *IEEE Access*, vol. 7, pp. 81 279–81 296, 2019.

[19] Y. Zhao, J. Li, C. Liao, and X. Shen, "Bridging the gap between deep learning and sparse matrix format selection," *SIGPLAN Not.*, vol. 53, no. 1, pp. 94–108, Feb. 2018.

[20] J. C. Pichel and B. Pateiro-López, "A new approach for sparse matrix classification based on deep learning techniques," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2018, pp. 46–54.

[21] M. Goetz and H. Anzt, "Machine learning-aided numerical linear algebra: Convolutional neural networks for the efficient preconditioner generation," in *2018 IEEE/ACM 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (scalA)*, Nov 2018, pp. 49–56.

[22] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016.

[23] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational data processing in Spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1383–1394.

[24] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "MLlib: Machine learning in Apache Spark," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 1235–1241, Jan. 2016.

[25] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 599–613.

[26] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, "FitNets: Hints for thin deep nets," in *In Proceedings of ICLR*, 2015.

[27] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.