Bridging the Gap between Deep Learning and Sparse Matrix Format Selection

Yue Zhao
Computer Science, North Carolina State University
Raleigh, NC, USA
yzhao30@ncsu.edu

Jiajia Li
CSE, Georgia Institute of Technology
Atlanta, GA, USA
jiajiali@gatech.edu

Chunhua Liao
CASC, Lawrence Livermore National Laboratory
Livermore, CA, USA
liao6@llnl.gov

Xipeng Shen
Computer Science, North Carolina State University
Raleigh, NC, USA
xshen5@ncsu.edu

Abstract

This work presents a systematic exploration on the promise and special challenges of deep learning for sparse matrix format selection—a problem of determining the best storage format for a matrix to maximize the performance of Sparse Matrix Vector Multiplication (SpMV). It describes how to effectively bridge the gap between deep learning and the special needs of the pillar HPC problem through a set of techniques on matrix representations, deep learning structure, and cross-architecture model migrations. The new solution cuts format selection errors by two thirds, and improves SpMV performance by 1.73× on average over the state of the art.

CCS Concepts • **Mathematics of computing → Computations on matrices**; • **Computing methodologies →** *Neural networks*; *Modeling methodologies*;

Keywords SpMV, Sparse matrix, Format selection, Convolutional neural network, Deep learning

ACM Reference Format:
Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. 2018. Bridging the Gap between Deep Learning and Sparse Matrix Format Selection. In *PPoPP '18: PPoPP '18: 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3178487.3178495

PPoPP '18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-4982-6/18/02…$15.00
https://doi.org/10.1145/3178487.3178495

1 Introduction

Deep Neural Networks (DNN) has already shown high impact on various applications via its several distinctive appealing properties, such as much less demand for feature extraction and highly accurate prediction. Its special efficacy is demonstrated in assisting perceptions and decision makings.

In High Performance Computing (HPC), there are many cases calling for decision makings that resemble some problems in other domains on which DNN has shown effectiveness. However, HPC problems also feature some distinctive attributes that pose some special challenges to DNN (detailed below). It remains an open question how to bridge that gap to tap into the potential of DNN for HPC. In this work, we take one of the fundamental HPC problems, sparse matrix storage format selection, as the focus to explore this direction.

Sparse matrix vector multiplication (SpMV) is one of the most important, widely used kernels in many scientific applications (e.g., linear equation system solvers) [9, 16]. It is also often the performance bottlenecks of their executions [8, 13]. Optimizing its performance is difficult because of the irregular indirect data access patterns.

One of the most important factors people have observed for the SpMV performance is the selection of the proper format to represent sparse matrices in memory. Various storage formats have been proposed for diverse application scenarios and computer architectures [6, 10, 17, 19, 23, 24, 33, 38, 42, 43]. As observed in numerous studies [6, 10, 20, 22, 26, 32, 38, 40], the different formats may substantially affect the data locality, cache performance, and ultimately the end-to-end performance of SpMV (for as much as several folds [20]).

The problem is a good candidate for DNN, especially Convolutional Neural Networks (CNN), to solve, for several reasons. First, selecting the proper format is a challenging task for programmers. The proper format of a sparse matrix depends on its matrix size, nonzero distribution, architecture characteristics, and so on. Second, it has been also difficult for traditional machine learning techniques to solve. Due to

the difficulties in coming up with the right features of matrices for learning and the complex relations between SpMV performance and the proper format of a sparse matrix, so far, traditional machine learning techniques have achieved an average of 85% [20] and 78% [32] prediction accuracy [1]. Considering the misprediction may lead to several-fold performance loss, a more accurate prediction model is needed. Third, the problem resembles some other tasks that CNN has proved effective. Particularly, it is akin to image classification—such as, to tell whether an image contains a dog or a cat; in both problems, the right decisions are primarily determined by the spatial patterns of the elements in an input. For image classification, the patterns are of pixels, and for sparse matrix format selection, they are of non-zero elements. As CNN has shown good efficacy on image classifications, the similarity of the two problems suggests some promise for it to work for sparse matrix format selection.

On the other hand, the problem poses some special challenges to CNN. Three of them are especially prominent.

The first is input representation. CNN typically requires all input data to be of the same size (as it usually has a fixed number of visible nodes.) For a dataset that contains data of different sizes, some data normalization methods are usually applied to transform them into a fixed size. For images, the transformation could be cropping, scaling or sampling. That approach works in general for image processing because it keeps the major patterns of the objects in the image. But sparse matrix format selection is sensitive to some subtle features of a matrix, which can get lost by those traditional transformations. For instance, as Section 4 will elaborate, scaling creates some diagonals for a non-diagonal matrix; as having diagonals critically affects format selection for SpMV, the scaled images mislead CNN learning and prediction. Therefore, new research explorations are necessary for understanding the effects of the various transformations for keeping the important features of sparse matrices, and for finding the fixed-size representations that properly suit the need of CNN for sparse matrix format selection.

The second special challenge is the design of the suitable CNN structures. CNN structure refers to the number of network layers, the type of network of each layer, and the number of nodes on each layer. Differences in CNN structures affect the quality of the learning results significantly. For image processing, researchers and practitioners have used different CNN structures for some public image sets. However, because of the different representations of the input data, the prior explored structures for image processing may not work effectively for sparse matrix format selection. New research is hence needed for identifying the structures of CNN that fit the needs of sparse matrix format selection.

[1]The accuracy number of [20] is the average of different platforms and precisions, and that of [32] is the average of different platforms using their "Advanced2" feature set.

The third challenge is the architectural dependence of sparse matrix format selection. As prior studies have shown [10, 38, 41], many factors of a machine (e.g., memory bandwidth, cache size, number of cores) could affect the performance of SpMV on a particular matrix format and the best format for a given matrix. A prediction model built for one machine rarely works well for another. Re-training CNN is time-consuming. How to efficiently migrating a model across systems is a problem specifically important for HPC problems.

The paper presents our research results for addressing these challenges. It makes several major contributions.

First, it describes a set of fixed-size representations of sparse matrices that we have designed, reports their influences, and identifies a histogram-based representation as the most effective choice. (Section 4)

Second, it empirically reveals the influence of CNN structures on sparse matrix format selection and identifies a *late-merging structure* as a CNN structure that suites the needs of Sparse Matrix format selection. The structure delays the integration of the information from different parts of the input representation to a late stage of the CNN processing, making it better match the input representations of sparse matrices. (Section 5)

Third, it introduces a concept in machine learning, *transfer learning*, into HPC, and reveals its potential for alleviating the cross-architecture migration difficulties for CNN-based models to serve for matrix format selection. It proposes two ways to materialize *transfer learning* in this new context, empirically compares their effectiveness, and demonstrates the large savings of the model migration overhead the technique brings. (Section 6)

Fourth, using an expanded set of sparse matrices, it compares the CNN-based method with the state of the art of sparse matrix format selection. The results indicate that the new method improves the accuracy of the best matrix format selection from 85% to 93%. The predictions rectified by the CNN model yields 1.73 average (up to 5.2) speedups to SpMV. (Section 7)

Finally, the paper crystallizes all the explorations into a set of novel findings on the applications of CNN to sparse matrix format selection, which could shed insights for bridging the gap between CNN and other HPC problems. It also discusses the ways that the developed CNN-based selector can be adopted in practice, through the integrations into a compiler, a library, or serving as a standalone tool. (Section 8)

2 Background

This section provides the necessary background on sparse matrix formats, their usage in SpMV, and CNN.

2.1 Sparse Matrix Storage Format

To efficiently store and process a sparse matrix, compressed data structures (a.k.a. storage formats) are used which store

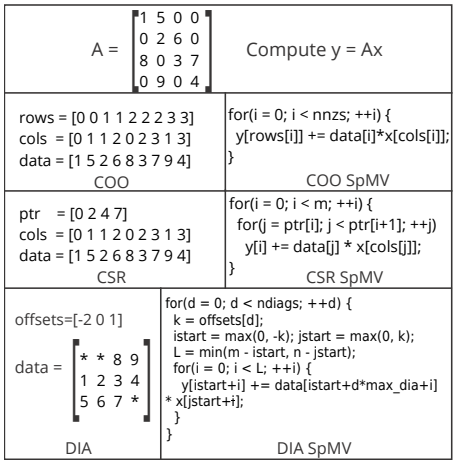
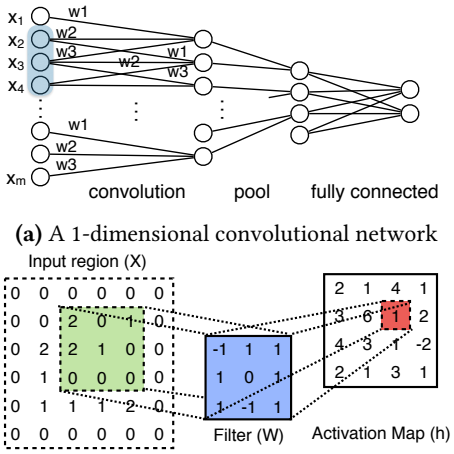


Figure 1. Sparse matrix storage formats and their corresponding SpMV pseudo-code (adapted from [20]).



(a) A 1-dimensional convolutional network

(b) A 2-D convolutional operation

Figure 2. Illustration of CNN.

only nonzero entries. Various storage formats have been proposed [6, 10, 17, 19, 23, 26, 33, 38].

As examples, Figure 1 shows a sparse matrix represented in COO, CSR, DIA formats respectively and their corresponding SpMV algorithms. *m*, *n*, and *nnzs* are used to represent the number of rows, columns, and nonzero entries of the sparse matrix respectively. Coordinate (COO) format explicitly stores the row and column indices and the values of all nonzero entries in *rows*, *cols*, and *data* arrays separately. Compressed sparse row (CSR) format retains the same *cols* and *data* arrays of COO but compresses the row indices into *ptr*, elements of which are the beginning positions of all rows in the *cols*/*data*. Diagonal (DIA) format stores non-zeros along the diagonal direction (from top left to bottom right). In the DIA example in Figure 1, the first row of *data* contains the two elements on the left bottom diagonal of matrix A, the second row is for the principal diagonal of matrix A, and the third row is for the diagonal on the top right of the matrix. The array *offsets* records the offsets of each diagonal from the principal diagonal. Please refer to prior literature [28, 31] for more details.

2.2 Convolutional Neural Networks (CNN)

CNN is a class of networks that finds non-linear models of patterns in inputs and makes data classifications. The most popular type of CNNs used in image recognition are convolutional neural networks (CNNs) [15, 18].

A CNN usually consists of a stack of layers of nodes. In Figure 2 (a), the leftmost layer is the *input level*, with each node representing one element in the input vector (e.g., the gray value of a pixel in an input image), the rightmost is the *output level*, with each node representing the predicted probability for the input to belong to one of two classes.

The output layer of a CNN gives the final prediction, while the other layers gradually extract out the critical features from the input. A CNN may consist of mixed types of layers, some for subsampling results (*pooling* layers), some for non-linear transformations. Convolution layers are of the most importance, in which, convolution shifts a small window (called a *filter*) across the input, and at each position, it computes the dot product between the filter and the input elements covered by the filter, as Figure 2 (b) shows in a 2-D case. In Figure 2 (a), the weights of every three edges connecting three input nodes with one layer-2 node form the filter $< w_1, w_2, w_3 >$ at that level. The result of a convolution layer is called an *activation map*. Multiple filters can be used in one convolution layer, which will then produce multiple activation maps. The last layer (i.e., the output layer) usually has a full connection with the previous layer.

Part of the CNN training process is to determine the proper values of the parameters in the filters (i.e., weights on the edges of the networks). In training, all the parameters in the network are initialized with some random values, which are refined iteratively by learning from training inputs. Each training input has a label (e.g., the ground truth of its class). The forward propagation on an input through the network gives a prediction; its difference from its label gives the prediction error. The training process (via back propagation) revises the network parameters iteratively to minimize the overall error on the training inputs. In using CNN, only forward propagation is needed to get the prediction.

Note that the size of the inputs to a CNN is typically fixed, equaling to the number of nodes in its input layer. If the raw inputs are of different sizes, they have to be normalized to the unified size.

3 Overview

In this work, by overcoming some major difficulties, we successfully construct a CNN-based sparse matrix format selector for SpMV. This section gives a high-level overview

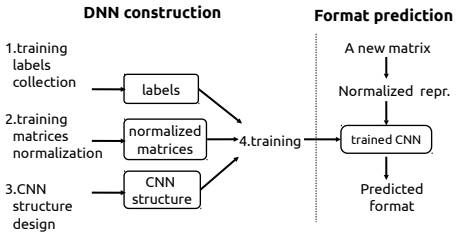
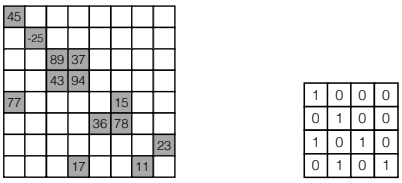


Figure 3. Overview.



(a) An example original matrix.

(b) Normalized binary matrix.

Figure 4. Image scaling loses some subtle but critical information for matrices.

of the construction process, and lists the major challenges which will be discussed in detail in later sections.

As the left half of Figure 3 shows, the construction process consists of four steps. It assumes that there are already a large set of sparse matrices S that the construction can use as its training inputs, and a target computing platform P (where the executions of SpMV happen.)

(1) The first step collects training labels. It runs SpMVs for all matrices on P multiple times, using a different matrix format each time. By measuring the execution times, for each matrix, it finds out the format that SpMV runs fastest and labels that matrix with the ID of that format. (2) The second step normalizes each of the matrices into a fixed size such that they can be fed into the input layers of CNN. (3) The third step designs the structure of CNN. The parameters of the designed CNN are initialized to some random values. The output layer is composed of K output nodes with each corresponding to one of the K matrix formats to choose from. (4) The fourth step runs the standard CNN training algorithm on the collected labels and the normalized matrices to finally determine the value of each parameter of the CNN, and concludes the construction process.

Inference with the trained CNN model is easy. For a given matrix, it is first normalized to the fixed size required by the CNN. The normalized representation is then fed into the trained CNN, the output nodes of which give the probabilities for each of the formats to be the best choice for that input matrix to use.

Steps one and four are easy to realize. Steps two and three face some research challenges special to the sparse matrix format selection problem. Additionally, the resulting CNN is specific to the training platform P as it uses the labels collected on P. How to quickly migrate the learned CNN to another platform is another research challenge. The next three sections separately describe our solutions to these three major research issues.

4 Input Representations

Matrices are of various sizes. For them to work with CNN, they have to be represented in a single size as the input layer of a CNN requires. This process is called matrix normalization. It is important that the normalization keeps the features of the original matrix that are crucial for determining the appropriate format for the matrix to use. In this section, we first describe the common normalization method used in image processing, and then present two novel methods we propose to overcome the limitations of the traditional methods in the context of sparse matrix format selection.

Representation from Traditional Methods In CNN-based image processing, input normalization is through image scaling, which down-samples large images or interpolates small images. We tried to apply the same method to matrices. In down sampling, for instance, a block in the original matrix maps to one element in the new matrix. Because the spatial patterns of non-zero elements rather than their exact values are relevant to SpMV performance, the values of the elements of the normalized matrix are set to binary. The new element is set to zero if the original block contains all zeros, and 1 otherwise. It results in a binary matrix.

The scaling method's results are not satisfying, reflected by low prediction accuracies (88%) by its constructed CNN (detailed in Section 7). The main reason is that although scaling keeps the coarse-grained patterns of objects in an image, it loses some subtle info that is critical for matrix format selection.

Figure 4 shows such an example. The original matrix contains irregular diagonals, but after down-sampling, the normalized matrix becomes a perfect regular diagonal matrix. Being a diagonal matrix is an important property for matrix format selection: Some matrix storage formats (e.g., DIA) are designed particularly for efficiently storing diagonal matrices; the normalization result hence causes confusions to the CNN construction as well as the format prediction. Such cases happen frequently on large sparse matrices.

We explored a number of representations to help address the limitations, two of which showed good promise, described next.

Augmentation with Density Representation In *density representation*, instead of producing zero or one for each block of the original matrix, it produces a decimal value between 0 and 1, equaling the number of nonzero entries in a block divided by the block size, as Figure 5 (a) illustrates for the original matrix in Figure 4 (a).

0.5	0	0	0
0	1	0	0
.25	0	.75	0
0	.25	0	0.5

(a) The density representation.

2	0	0	0
4	0	0	0
3	0	1	0
2	0	1	0

(b) The row histogram representation.

Figure 5. The density representation and histogram representation of the matrix in Figure 4 (a).

Compared to the binary representation, the density representation captures more detailed variations among the different regions of the original matrix. Binary representation is still useful for capturing the overall spatial patterns of the non-zero elements in the original matrix. Using both of them in one CNN could possibly get the best of both worlds. We will return to this point when discussing the structure design of CNN in Section 5. The density and binary representations can be produced by one traversal of the original matrix.

Algorithm 1 Normalized through Histogram Sampling.

```
1:  procedure HISTNORM(A, r, BINS)
2:  /* create a row histogram for an input matrix A */
3:  /* the target representation is a r × BINS matrix R*/
4:      initialize a r × BINS empty matrix R
5:      ScaleRatio = A.height/r
6:      MaxDim = max(A.height, A.width)
7:      for each non-zero entry e in A do
8:          int row = e.row / ScaleRatio
9:          bin = BINS ×|e.row − e.col| / MaxDim
10:         R[row][bin]++
11:     return R
```

Distance Histogram Representation Our second proposal is *distance histogram* or called *histogram representations*, which stores the spatial distribution of non-zero elements in a matrix through histograms. It consists of two matrices, with one storing the histograms for the rows of the original matrix, and the other for the columns. The histogram is based on the distance between an element and the principal diagonal of the original matrix.

Algorithm 1 outlines the algorithm for constructing the row histogram from a given matrix. We take the original matrix in Figure 4 (a) as an example for explanation. In this example, we try to construct a 4-row histogram matrix R for the matrix. Every two consecutive rows in the original matrix produce one row in the histogram matrix. Suppose that we want the histogram to have 4 bins regarding the distances from the elements to the principal diagonal of the original matrix. Now consider the bottom two rows (rows 6 and 7 with 0-base index) in Figure 4 (a). Row 6 contains only one non-zero element (with value 23), whose distance from the principal diagonal is 1; the histogram bin number for that distance is $\lfloor 1/2 \rfloor = 0$. It hence causes $R[3][0]$ to increase by 1 (lines 9 and 10 in Algorithm 1). Row 7 contains two non-zero elements (with values 17 and 11), and their distances from the principal diagonal are 4 and 1 respectively. Their histogram bins are 2 and 0 respectively (calculated by line 9 in Algorithm 1), causing $R[3][2]$ and $R[3][0]$ each to increase by one. Hence, the result of the bottom row of R is [2, 0, 1, 0].

In the same vein, one can construct a histogram for the columns of the original matrix. Together they form the histogram representation for the matrix. The values in both matrices are then normalized to the range of [0,1] by dividing the largest value in each.

Compared to the binary and density representations, the histograms—using numerical values and leveraging distances rather than direct spatial locations—tend to capture richer information about the distribution of non-zero elements in the matrix. Meanwhile, their sizes are more flexible to adjust. For binary and density representations, because matrices can be larger in either dimension, the representations are typically made square to strike a tradeoff. For histograms, there is no such a need; the number of histograms can differ from the number of rows or columns. A benefit of the relaxed constraint is that the size of the histograms could be smaller. For instance, in our experiments reported later, 128×128 is the size that the binary and density representations should take to get a good prediction accuracy, while 128×50 already works well for histograms.

5 CNN Structure Designs

A CNN network may take various structures with different depths or widths or types of layers. Different structures have distinct modeling strengths and generalities. For instance, a more complex network with deeper and wider layers can typically capture more complex relations between the inputs and outputs, but at the same time, it would require more data to train than a simpler network needs.

Insufficiency of Common Structures Despite the large varieties, in image processing, the usual structure of a CNN is as Figure 6 illustrates. The values in different channels of an image (e.g., Red, Green, Blue values of pixels) together form a single input layer, and the other layers all work on the values combined from all these input nodes. The differences among different CNNs for image processing have been mostly on the configurations of each of the layers after the input layer.

In our exploration, we start with a set of CNNs of the similar type of structure as used in image processing. However, after training each of the CNNs, we find the resulting prediction accuracies unsatisfying, regardless of which input representation is used (detailed in Section 7.)

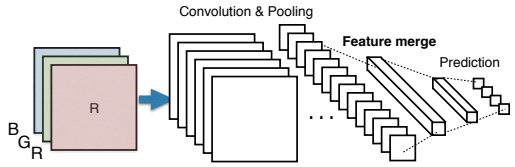


Figure 6. Traditional CNN structure that merges the info from different channels at the early stage of the networks.

Through examinations of the intermediate computation results from the CNNs, we found that the fundamental reason was the combinations of the info from different sources in the early stages of the networks. In image processing, the different sources of information are just different channels of the input image, and have a homogeneous semantic. For instance, for an RGB image, the j^{th} element in each of the three channels all corresponds to the j^{th} pixel in the image, and the values are all about the numerical values of the pixel's color, just in different color channels.

However, in our problem, the different sources of the info do not have such homogeneous semantic or one-to-one matching relations. Consider the binary representation and the density representation. Even though each element is derived from one region in the original matrix, they have different types of values. In the histogram case, even though the elements in the two histogram representations have the same value type, they do not have a one-to-one matching relation: one for rows, the other for columns. As a result, the combinations of different sources of info at the early stage of CNN work well for image processing, but not for our problem.

Late-merging Structure To address the problem, we employ an alternative *late-merging structure*. As Figure 7 illustrates, the structure consists of two separate convolutional networks with each processing the info from one source, and only at the very last stage, the outputs of the two networks are put together as joint features, fed to the fully connected layer for the final output. The two convolution networks can be regarded as processes to extract the critical features from each of the two sources of input information. The final layer combines these features for the final prediction.

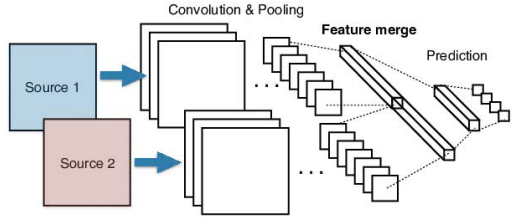


Figure 7. Proposed late-merging CNN structure.

Such a structure avoids the early mixing of the influence from the different sources in the early-merging structure, and gives two-fold benefits. First, it leads to a simpler problem. In the early-merging case, the convolution network has to simultaneously consider the influence from both sources of info and extract out their combined features. In comparison, each of the convolution networks in the late-merging structure only needs to extract the features from one source of info. The simplicity entails a simpler structure needed for the convolutional networks, and hence the reduced demands for the amount of training data and time. Second, the late-merging structure avoids the complexities from the different value types and semantics of the different sources of info.

The way that the *late-emerging structure* combines features from different sources is similar to feature concatenations in other models (e.g., Inception [34]). It fits the special properties of sparse matrices in the context of format selection.

For either the *early-merging* or *late-merging* structure, there are numerous possible designs of the CNN with different numbers of layers and other configurations. Following the common practice, we choose the configuration through the trial-and-error method by training and testing many CNNs with different configurations and select the best one.

6 Cross-Architecture Adaptations

The need for cross-architecture migrations is another special aspect of sparse matrix format selection compared to CNN-based image processing. In image processing, data labels are oblivious to computing systems: A dog in a picture is a dog on whatever machines. But it is not the case for sparse matrix format selection. The labels are the best formats for the input matrices, which could differ significantly from one machine to another. As a result, the CNN trained on one machine cannot predict the formats for another machine well (as Section 7.4 shows).

The implication of the architecture-dependence is that a new CNN has to be built on each different machine, which includes the collection of labels by rerunning SpMV on each matrix on the new machine and rerunning the CNN training algorithm to determine the appropriate parameters. Both parts take a lot of time. Such a process takes about 75 hours in our experiments for about 9200 matrices.

6.1 Concept and Two-fold Contributions

To alleviate this problem, we explore the use of *transfer learning*. Transfer learning is an idea existing in the deep learning field [30, 44]. In image recognition, people can train a CNN on a large dataset, and then use the pre-trained network as the base in training a CNN for a new dataset. In the previous studies, the idea is only explored for speeding up the training of CNN across datasets.

Our contributions are two-fold. First, we introduce this idea to HPC to support cross-architecture portability of

CNNs, and point out its benefits in saving both data collection time and CNN training time. Second, we empirically examined the options for materializing the idea in this new context and identified the suitable method. We next describe the explored options and the rationale of each; Section 7 will provide the empirical comparisons.

6.2 Options Explored

We have primarily considered two ways to materialize the transfer learning idea for sparse matrix format selection.

Continuous evolvement. This method treats the existing CNN as an intermediate state of the new CNN, based on which, it feeds the new set of training data collected on the new machine to the CNN to continue training the CNN until a convergence. This method reuses the structure and parameters of the previous CNN. Because these parameters have reflected certain important features of input matrices, reusing them could provide a better starting point than random values for the CNN construction for the new dataset. Meanwhile, even though the labels on two machines could differ, they usually do not differ completely. By inheriting the parameters of the previous CNN, the training of the new CNN could benefit from certain relations that those parameters have already captured between the features of matrices and the suitable formats.

Top evolvement. This second method inherits all parts of the previous CNN and keeps both the structure and the parameters unchanged, except the parameters of the top fully connected layer. We call the output of the reused part of the CNN the *CNN codes*. Because of the reuse, the CNN codes of a matrix stay the same across machines. Therefore, in the construction of the new CNN, this method just feeds these codes as inputs to the top layer of the CNN and uses the standard back propagation (on the labels collected on the new machine) to learn the suitable parameters of this top layer. After that layer is trained, we get the updated CNN by putting that layer on top of the reused part of the previous CNN. The rationale of the *top evolvement* method is that the major part of a CNN can be considered as the extraction of critical features of inputs. Because the tasks on the two machines share a similar nature, the set of critical features of the inputs could be similar—hence, the reuse. The top layer combines these features together and makes the final decision on the output and is hence closely related to the labels of data—hence, the update.

Qualitative Comparisons Qualitatively, the *top evolvement* involves a much smaller set of parameter changes than the first option does. As a result, it needs fewer training data, which entails a shorter time needed for collecting new labels and also a shorter time for the CNN training algorithm to converge. On the other hand, the *continuous evolvement* allows a much larger freedom for adjustment; as a result, theoretically speaking, the best CNN it can provide (infinite training data are allowed) shall be no worse than the one provided by the *top evolvement* method. However, if the objective is to achieve a high-quality CNN at a minimum training cost, the answer may not be that clear. Section 7 will offer some quantitative comparison results.

Table 1. Hardware platforms used in the experiments.

	Intel® CPU	AMD® CPU	NVIDIA® GPU
CPU	Xeon CPU E5-4603	A8-7600 Radeon R7	GeForce GTX TITAN X
Freq.	2.4 GHz	3.1 GHz	1.08 GHz
Cores	24	4	3072
Memory	64GB DDR3 1.9 GHz	12GB DDR3 2.1 GHz	12GB GDDR5 3.5 GHz
Memory Bandwidth	103 GB/s	25.6 GB/s	168 GB/s
OS/Driver	SUSE Linux Server 11	Ubuntu 16.04	CUDA 8.0
Compiler	icc (17.01)	gcc (6.2)	nvcc (8.0)

7 Evaluations

To evaluate the efficacy of the technique, we run a series of experiments and compare the CNN-based method with the state-of-the-art method in both prediction accuracies and resulting SpMV speeds. In addition, we report the influence of the three kinds of input representations, the benefits of the *late-merging* CNN structure, the impact of the two methods of *transfer learning*, and the sensitivity of the learning results to the granularity of the input representations.

7.1 Methodology

Baseline for Comparison The state of the art in sparse matrix format prediction is a traditional machine learning model produced by Li et al. [20] and Sedaghati et al. [32]. They manually came up with a set of features of matrices, and built up a decision tree based on the features and labels of a set of training matrices. They showed that the decision tree model outperforms previous methods in prediction accuracies and the resulting SpMV performance.

Hardware We construct format prediction models for two CPU systems and one GPU platform, detailed in Table 1. The trainings of CNNs all happen on the GPU.

Software Platforms and Formats to Select As Figure 1 has shown, different formats require SpMV to be coded differently. To evaluate the speedups of SpMV brought by format predictions, we need to use a SpMV library that can work with multiple matrix formats. A SpMV library contains only a set of procedures with each working with one of a small set of matrix formats, which determine the set of formats we can use in our experiments.

On CPU, we experiment with two SpMV libraries. One is Intel® MKL [39], which supports COO, CSR, DIA, and several other formats. The other is a multi-purposed SpMV benchmarking program (called *SMATLib* in this paper) from the previous work [20], which supports COO, CSR, DIA, ELL formats. Both libraries are multithreaded and run in parallel.

On Intel® MKL [39], the speedups from our prediction over the use of the default format (CSR) is up to 53.7× (1.46× on average). We concentrate our CPU result discussions on the SMATLib platform rather than MKL because SMATLib was used in the state-of-the-art study on matrix format prediction [20]. Reusing it allows for a direct head-to-head comparison with the previous work. The framework repeats a performance measurement for 50 times. The average is taken as the performance. We observe some small variances across the repeated trials; they are negligible compared to the significant performance differences between the different formats.

On GPU, we experiment with the NVIDIA® cuSPARSE library [28], which supports COO, CSR, ELL, HYB and BSR formats. A previous work reports some promising performance of a new format CSR5 over some alternative formats and publishes the implementation [21]. We append to cuSPARSE with the CUDA implementation from that work to make it also work with CSR5 format.

The set of formats covered in this study are restricted by the formats supported by these existing libraries. The support of these covered formats by these (commercial) libraries indicates their competitiveness and general applicability. The set unavoidably leaves some formats uncovered. With the idea verified, the approach can be easily extended to the selection of other formats.

Dataset Our experiments use a set of 9200 matrices (total size around 400GB). These matrices include the 2757 real-world matrices from the SuiteSparse matrix collection [11] (which was also used in the previous studies [20, 32]), and some extra matrices derived from them. The derivation attempts to create some variations of the existing matrices, and at the same time, do not deviate too much from the real-world matrices. To do this, we use some simple heuristics like cropping, transforming and randomized combinations of the original matrices.

Cross Validation For evaluations, we separate testing data from training data through 5-fold cross validations. This is a method commonly used in statistical learning for evaluations. It takes 20% of dataset out to form a test set and uses the remaining for training. It repeats the process for 5 times with a different subset of the dataset taken out as the test set. In all experiments, the results are in single precision, which was also used in the previous work [20]. Our observations on double precision showed similar prediction accuracy improvements and relative speedups as those on single precision.

In the rest of this section, Sections 7.2 to 7.5 first report the prediction accuracies of the machine learning models and the speedups they brought to SpMV. Section 7.6 then discusses the runtime overhead and the related practical usage issues.

7.2 Prediction Accuracy

This part compares the CNN models (with late merging) and the previous Decision Tree (DT) model [20, 32] in the quality of their predictions. We use three metrics. The first is *overall accuracy*, defined as the number of correct predictions (i.e., the predicted best format is indeed the best) over the total number of matrices. The other two assess the quality of the predictions for each format: *precision* on format X is the fraction of all predicted X that is correct; *recall* on format X is the fraction of S_x that are predicted as X, where S_x is the set of matrices on which format X is indeed the best.

Table 2 reports the results of our three CNN-based models (with Binary, Binary+Density, or Histogram representations) and the previous Decision Tree-based model (DT) [20, 32] on the Intel® CPU platform. The "Binary" and "Binary+Density" use 128×128 as the size of the representations, while the "Histogram" uses 128×50 (these sizes were picked empirically.) The second column in Table 2 gives the number of matrices of each format label from the overall 9200 sparse matrices. The results show that all three CNN models outperform the DT model, in almost all metrics. CNN with histograms achieves the best results, with an overall accuracy of 93%; DT gets only 85% accuracy. CNN also shows higher recall rates and precisions for all the formats.

The results differ for different formats. There are two major factors: the amount of training data and the complexity of patterns. CSR, for instance, has much better prediction results than other formats have thanks to the largest number of matrices carrying the CSR label, which allows CNN to learn it more sufficiently. COO and ELL have the similar amounts of data, but COO has much worse prediction results than ELL has. Our detailed analysis shows that matrices favoring ELL tend to have a similar pattern (rows in the matrix have similar numbers of non-zeros), while COO does not show clear patterns.

Table 3 reports the results on the GPU platform. As GPU memory is more limited, only 4218 of the matrices can run on it. As GPU cuSPARSE supports more formats than SMATLib, the table shows results on six formats. For space limit, we include only CNN with histograms and DT results. Among the six formats, format COO never wins on GPU, while the other five all win on some matrices[2]. Overall, CNN again outperforms DT significantly in both the overall accuracy (90% versus 83%) and the per-format metrics.

[2]BSR uses a 4×4 block size [37].

Table 2. Prediction quality on Intel® CPU (bottom decimals show overall accuracies)

Format	Ground Truth	CNN+Binary		CNN+Binary+Density		CNN+Histogram		DT	
		Recall	Precis.	Recall	Precis.	Recall	Precis.	Recall	Precis.
COO	667	0.53	0.70	0.71	0.74	0.71	0.78	0.53	0.61
CSR	6947	0.94	0.92	0.94	0.94	0.97	0.96	0.90	0.88
DIA	894	0.82	0.83	0.82	0.85	0.93	0.93	0.83	0.75
ELL	692	0.79	0.78	0.82	0.80	0.90	0.85	0.71	0.85
Overall	9200	0.88		0.90		0.93		0.85	

Table 3. Prediction results on the GPU platform.

Format	Ground Truth	CNN+Histogram		DT	
		Recall	Precis.	Recall	Precis.
CSR	1340	0.87	0.89	0.86	0.83
ELL	282	0.73	0.73	0.58	0.58
HYB	170	0.61	0.64	0.38	0.45
BSR	1806	0.93	0.90	0.90	0.90
CSR5	620	0.87	0.91	0.83	0.88
COO	0	-	-	-	-
Overall	4218	0.90		0.83	

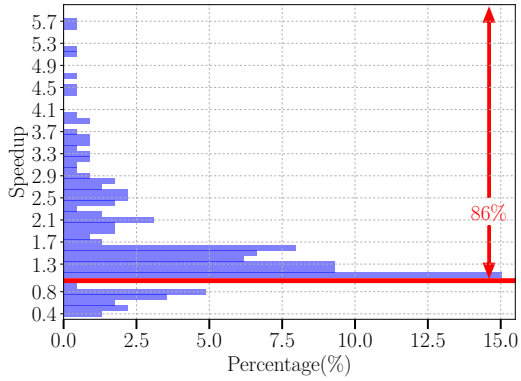


Figure 8. Speedups from CNN over DT-based predictions.

7.3 Speedup

This section reports the speedups of SpMVs by using our CNN model predicted formats over using the DT model's predictions. Figure 8 shows the speedup distribution over testing matrices on which the two models give different predictions of the suitable formats. The horizontal line at 1 explicitly separates the matrices achieving speedups using CNN model and those not showing speedups compared to the DT model. The CNN model helps improve the SpMV performance on 86% matrices over the DT model. The SpMVs using the CNN model predicted formats achieve an average of 1.73× and the maximum of 5.2× speedups over those of the DT model. This result further confirms that sparse formats are critical to SpMV performance. This comparison shows the performance improvement of our work over the state of the art[20]. Furthermore, we also tested the SpMV speedups with our CNN model over the default CSR format, which are 2.23× on average and 14.9× in maximum. For the GPU platform, the CNN model achieves an average of 1.7× and the maximum of 22.5× speedup of the default CSR format.

7.4 Model Migrations

Figure 9 reports the effect of the two transfer learning methods. The methods try to migrate the CNN model trained on Intel® platform to AMD® platform. The benefits of the methods are obvious. On less training data, they achieve much better accuracy than the "from scratch" method does. Note that the time to migrate a model consists of the retraining time of the CNN model as well as the time to collect the labels on the new platform. The latter takes most of the time. In our experiments, the collection of the labels for the 9200 matrices takes about 75 hours as it requires the executions of SpMV on each of the matrices many times.

Therefore, the large reduction of the needed retraining data by the transfer learning methods significantly shortens the model migration process. Suppose that we want to get a model on the new platform with a 90% accuracy. From Figure 9, we can see that with "top evolvement", it takes only about a quarter of the time the "from scratch" method takes, and two thirds of the time the "continuous evolvement" method takes. Overall, "continuous evolvement" achieves a slightly higher accuracy than "top evolvement" after relearning 4000 inputs, while "top evolvement" provides a faster learning process.

7.5 Impact of CNN Structures

This part compares late-merging and early-merging (Figures 6 and 7 in Section 5.) Figure 10 shows the CNN of the late-merging structure. It involves 13 layers. The early-merging model has the same structure except that the networks merge the use of the different channels at the beginning of the networks (the input layer uses one INPUT(128 x 128 x 2) rather than two separate INPUTs(128 x 128)).

Figure 11 shows the loss function curves of the two CNN models. Here, loss function is defined as the *cross-entropy* between the true labels and the predicted ones [27]. Cross-entropy is a measure of the similarity between the distributions; the smaller it is, the closer the two distributions

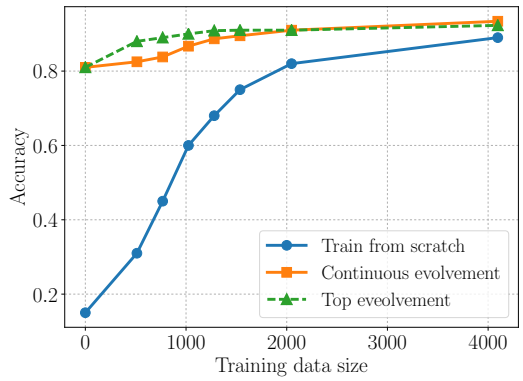


Figure 9. Prediction accuracies of different retraining methods on a new platform (AMD® platform). (Accuracies at x=0 are without retraining.)

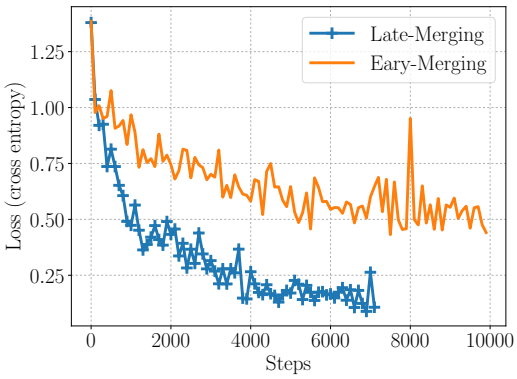


Figure 11. Comparison of the loss function convergence for two different structural designs.

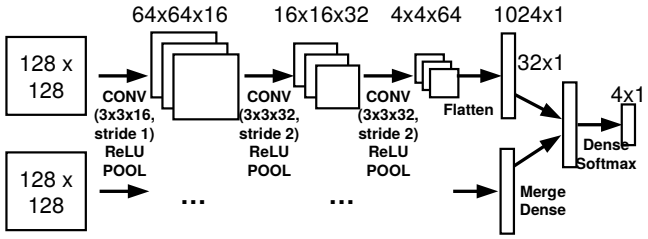


Figure 10. CNN late-merging structure (layer shapes in parentheses).

are. The loss function of late-merging structure decreases much faster than that of the early-merging structure. The late-merging structure uses about 7000 steps to converge to the loss value around 0.1, while early-merging structure converges to 0.4 after 10000 steps. By using only 1000 time steps, the loss function's value of late-merging is half of that of early-merging. In addition, the late-merging structure behaves more steadily than early-merging does. These results confirm that the late mixing of the different sources of information simplifies the prediction problem, and avoids the negative impact from the mixture.

7.6 Overhead and Uses in Practice

Training a CNN model based on collected data is about 27min. It is a one-time effort for a given platform.

Prediction overhead is worth more discussions. In both our CNN method and the previous DT-based method, the prediction for a matrix includes two steps: 1) input representation or feature extraction; 2) feeding the feature vector to the predictive model to get the prediction. In terms of the time taken by one iteration of SpMV (on the CSR format), the overhead for CNN method is (on average) 0.96× for step 1 and 0.13× for step 2, 1.09× in total. The overhead for the DT method is 3.4× for step 1 and 0.0085× for step 2, 3.4× in total. Another source of overhead is the format conversions. Converting one existing format of a matrix to another format could take a number of SpMV iterations' time.

These overhead entails different ways to use the CNN method in different settings. First, the runtime prediction overhead of the CNN method is much smaller than the DT-based method, which gives the CNN method an extra edge in addition to its better prediction accuracy. However, because SpMV is often called on a matrix repeatedly (e.g., a linear solver takes hundreds or thousands of iterations [5]), the 1–3 iterations of overhead is negligible compared to the time the better formats help save (see more proof in [35]).

Both methods are subject to format conversion overhead, which could be more significant than the prediction overhead. If the collection of matrices are to be used by many users (on the same or different platforms) for many SpMV-based computations, a one-time process can be used to generate and store all the candidate formats of the matrices. That can save the runtime needs for format conversions, and hence avoid the influence of format conversion overhead on the usage of the predictive models.

In a setting where new sparse matrices are generated and consumed throughout an execution, the predictions and storage conversions may both need to be invoked on the fly. A way to deal with it is to count these overhead when measuring the performance of a format during the label collection stage (i.e., step 1 in Figure 3). In this way, the predictive model will try to predict the format that minimizes the overall time including the overhead and the SpMV time.

To implement the on-the-fly usage, we may leverage either a compiler-based approach or a library-based approach. In the former, a compiler may transform some existing code by inserting invocations of the prediction model into the code along with calls to some procedures to do the matrix format conversion at runtime. In the latter, the predictive model can be also integrated into a sparse matrix library; when the procedure is called by a program, the model takes effects. The detailed explorations are left for the future.

8 Insights

This section summarizes three key insights and their general implications for other HPC tasks. The first is that CNN has some good potential for helping improve the decisions in HPC. Beyond the sparse matrix format selection problem, we envision that CNN may help other problems: selecting the proper solvers for solving a linear system (which also typically take sparse matrices as inputs)[3, 4, 7], choosing the proper compilation flags, finding good scheduling policies, and so on. Using CNN for them could face some challenges as those encountered in this work, such as cross-architecture portability, CNN structure designs, input representation issues. The findings reported in this work can shed some light on solving the issues in these other HPC scenarios.

The second insight is the importance of tailoring the representations of HPC data inputs according to the nature of the problem in the HPC domain. The representations we have proposed in this work could be useful for other sparse matrix algorithms beyond SpMV. Meanwhile, this work also shows the importance of thinking out of box when applying CNN to HPC, reflected by the benefits of the unconventional CNN structure (late-merging structure) driven by the special properties of the input representations.

The third insight is the benefits of *transfer learning* in saving time required for porting learned models across architectures. This technique is especially useful for HPC due to the widely existing architectural sensitivity.

9 Related Work

The work closest to this study is the SMAT work [20, 35] which builds up a decision tree for selecting the best storage format for a sparse matrix storage. The previous sections have provided quantitative comparisons with that work. A similar classification-tree–based model was built in another recent study [32] with a 65-84% accuracy. In addition, a previous work has proposed a hybrid format for sparse matrices [6, 33]. It allows the use of different formats for representing different subsets of a large sparse matrix. To select the format for each sub-matrix, it uses interpolation over a large offline collected performance database to estimate the performance if a certain format is used, based on its local features.

There are some other efforts trying to optimize the computations over sparse matrices, including hand-tuning input- or architecture-related features [6, 25, 41], designing new sparse formats [6, 17, 33], and building automatically performance tuning (auto-tuning) systems [10, 20, 32, 38].

In a broader scope, there has been a large body of work applying machine learning techniques to solve program optimization difficulties. Examples include some that focus on improving lower-level compiler optimizations [1, 2, 14, 29], some on algorithmic selections [3, 12], and some on dynamic compilations and adaptations [36].

To the best of our knowledge, this work is the first that explores the special challenges of applying CNN to HPC problems. Due to the special attributes of CNN, its usage in HPC faces some special complexities that prior applications of machine learning to HPC do not have, including the CNN structure designs and the input representations. It also brings some special opportunities, such as the use of *transfer learning* for alleviating cross-architecture portability difficulties. Explorations on these novel aspects are where the key contributions of this current work reside.

10 Conclusions

In this paper, we present a systematic exploration on closing the gap between CNN and sparse matrix format selection. It points out three-fold special challenges sparse matrix format selection poses to the applications of CNN: input matrix representation, CNN structure design, and the needs for cross-architecture migrations of the learned models. To tackle each of the challenges, it makes a set of innovations—including several novel matrix representations, use of late-merging CNN structure, and the use of *transfer learning* for reducing the large cost of cross-architecture model migrations. Meanwhile, it conducts a series of empirical measurements to unveil the influences of the various input representations, CNN structures, and transfer learning methods. The resulting predictive model demonstrates significant reductions of the prediction errors and brings substantial speedups for SpMV compared to the state-of-the-art technique. As one of the pioneering studies on bridging the gap between CNN and HPC, this work provides a set of insights that may help the adoption of CNN in other HPC problems.

Acknowledgments

We thank the reviewers for the helpful comments. This material is based upon work supported by DOE Early Career Award (DE-SC0013700), the National Science Foundation (NSF) under Grant No. CCF-1455404, CCF-1525609, CNS-1717425, and CCF-1703487, and IBM Ph.D. Fellowship Award. Besides, this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE, NSF, LLNL or IBM.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. 2006. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization*. 295–305.

[2] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. 2004. Finding effective compilation sequences.. In *LCTES'04*. 231–239.

[3] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: A Language and Compiler for Algorithmic Choice. In *PLDI*. Dublin, Ireland.

[4] H. Anzt, J. Dongarra, M. Kreutzer, G. Wellein, and M. KÃűhler. 2016. Efficiency of General Krylov Methods on GPUs – An Experimental Study. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 683–691. https://doi.org/10.1109/IPDPSW.2016.45

[5] Amir Beck and Marc Teboulle. 2009. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM journal on imaging sciences* 2, 1 (2009), 183–202.

[6] Nathan Bell and Michael Garland. 2009. Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*. ACM, New York, NY, USA, Article 18, 11 pages. https://doi.org/10.1145/1654059.1654078

[7] Sanjukta Bhowmick, Brice Toth, and Padma Raghavan. 2009. Towards low-cost, high-accuracy classifiers for linear solver selection. In *International Conference on Computational Science*. Springer, 463–472.

[8] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. 2003. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. In *ACM SIGGRAPH 2003 Papers (SIGGRAPH '03)*. ACM, New York, NY, USA, 917–924. https://doi.org/10.1145/1201775.882364

[9] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *Proceedings of the Seventh International Conference on World Wide Web 7 (WWW7)*. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 107–117. http://dl.acm.org/citation.cfm?id=297805.297827

[10] Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*. ACM, New York, NY, USA, 115–126. https://doi.org/10.1145/1693453.1693471

[11] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[12] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U. O'Reilly, and S. Amarasinghe. 2015. Autotuning Algorithmic Choice for Input Sensitivity. In *Proceedings of the 36th annual ACM SIGPLAN conference on Programming Language Design and Implementation*.

[13] R. D. Falgout. 2006. An Introduction to Algebraic Multigrid Computing. *Computing in Science Engineering* 8, 6 (Nov 2006), 24–33. https://doi.org/10.1109/MCSE.2006.105

[14] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle, Phil Barnard, Elton Ashton, Eric Courtois, and Francois Bodin. 2008. MILEPOST GCC: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385* (2015).

[16] Marat F. Khairoutdinov and David A. Randall. 2001. A cloud resolving model as a cloud parameterization in the NCAR Community Climate System Model: Preliminary results. *Geophysical Research Letters* 28, 18 (2001), 3617–3620. https://doi.org/10.1029/2001GL013552

[17] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. 2011. 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 (PPoPP '11)*. ACM, New York, NY, USA, 247–256. https://doi.org/10.1145/1941553.1941587

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[19] Daniel Langr and Pavel Tvrdik. 2016. Evaluation Criteria for Sparse Matrix Storage Formats. *IEEE Trans. Parallel Distrib. Syst.* 27, 2 (Feb. 2016), 428–440. https://doi.org/10.1109/TPDS.2015.2401575

[20] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: An Input Adaptive Auto-tuner for Sparse Matrix-vector Multiplication. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 117–126. https://doi.org/10.1145/2491956.2462181

[21] Weifeng Liu. 2016. Benchmark SpMV using CSR5. https://github.com/bhSPARSE/Benchmark_SpMV_using_CSR5. (2016).

[22] Weifeng Liu, Ang Li, Jonathan D. Hogg, Iain S. Duff, and Brian Vinter. 2017. Fast Synchronization-Free Algorithms for Parallel Sparse Triangular Solves with Multiple Right-Hand Sides. *Concurrency and Computation: Practice and Experience* 29, 21 (2017), e4244–n/a.

[23] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*. ACM, New York, NY, USA, 339–350. https://doi.org/10.1145/2751205.2751209

[24] Weifeng Liu and Brian Vinter. 2015. Speculative Segmented Sum for Sparse Matrix-vector Multiplication on Heterogeneous Processors. *Parallel Comput.* 49, C (Nov. 2015), 179–193.

[25] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. 2013. Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13)*. ACM, New York, NY, USA, 273–282. https://doi.org/10.1145/2464996.2465013

[26] Duane Merrill and Michael Garland. 2016. Merge-based Sparse Matrix-vector Multiplication (SpMV) Using the CSR Storage Format. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, New York, NY, USA, Article 43, 2 pages. https://doi.org/10.1145/2851141.2851190

[27] Kevin P Murphy. 2012. *Machine learning: a probabilistic perspective*. MIT press.

[28] M Naumov, LS Chien, P Vandermersch, and U Kapasi. 2010. CUSPARSE Library: A Set of Basic Linear Algebra Subroutines for Sparse Matrices. In *GPU Technology Conference*, Vol. 2070.

[29] Eunjung Park, L.-N. Pouche, J. Cavazos, A. Cohen, and P. Sadayappan. 2011. Predictive modeling in a polyhedral optimization space. In *IEEE/ACM International Symposium on Code Generation and Optimization*. 119 –129.

[30] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. 2014. CNN Features off-the-shelf: an Astounding Baseline for Recognition. *CoRR* abs/1403.6382 (2014). http://arxiv.org/abs/1403.6382

[31] Yousef Saad. 1994. *SPARSKIT : a basic tool kit for sparse matrix computations*. Technical Report. University of Minnesota.

[32] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. 2015. Automatic Selection of Sparse Matrix Representation on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*. ACM, New York, NY, USA, 99–108. https://doi.org/10.1145/2751205.2751244

[33] Bor-Yiing Su and Kurt Keutzer. 2012. clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*. ACM, New York, NY, USA, 353–364. https://doi.org/10.1145/2304576.2304624

[34] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[35] Guangming Tan, Junhong Liu, and Jiajia Li. 2018. Design and Implementation of Adaptive SpMV Library for Multicore and Manycore Architecture. *ACM Trans. Math. Softw. (To appear)* (2018).

[36] K. Tian, Y. Jiang, E. Zhang, and X. Shen. 2010. An Input-Centric Paradigm for Program Dynamic Optimizations. In *the Conference on*

Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).

[37] Richard Wilson Vuduc. 2003. *Automatic Performance Tuning of Sparse Matrix Kernels*. Ph.D. Dissertation. AAI3121741.

[38] Richard W. Vuduc and Hyun-Jin Moon. 2005. Fast Sparse Matrix-vector Multiplication by Exploiting Variable Block Structure. In *Proceedings of the First International Conference on High Performance Computing and Communications (HPCC'05)*. Springer-Verlag, Berlin, Heidelberg, 807–816. https://doi.org/10.1007/11557654_91

[39] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.

[40] Xinliang Wang, Weifeng Liu, Wei Xue, and Li Wu. 2018. swSpTRSV: a Fast Sparse Triangular Solve with Sparse Level Tile Layout on Sunway Architectures. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (To appear) (PPoPP '18)*.

[41] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2009. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Comput.* 35, 3 (2009), 178 – 194. https://doi.org/10.1016/j.parco.2008.12.006 Revolutionary Technologies for Acceleration of Emerging Petascale Applications.

[42] Biwei Xie, Jianfeng Zhan, Zhen Jia, Wanling Gao, Lixin Zhang, and Xu Liu. 2018. CVR: Efficient SpMV Vectorization on X86 Processors. *The 2018 International Symposium on Code Generation and Optimization (To appear)* (2018).

[43] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. yaSpMV: Yet Another SpMV Framework on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, New York, NY, USA, 107–118. https://doi.org/10.1145/2555243.2555255

[44] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. How transferable are features in deep neural networks? *CoRR* abs/1411.1792 (2014). http://arxiv.org/abs/1411.1792

A Artifact appendix

A.1 Abstract

Our artifact provides Python programs implementing the CNN-based prediction model for sparse matrix format selection as described in the paper. With the included dataset, it can be used to evaluate the prediction performance. It also provides interface to predict the best format for a given matrix. We also provide shell script to automatically run some tests for fast evaluation convenience.

A.2 Artifact check-list (meta-information)

- **Algorithm: Automatic sparse matrix format selection for SpMV.**
- **Program: Tensorflow, sparse matrix (all sources and testing data included).**
- **Compilation: Not required.**
- **Binary: No binary needed, Python APIs are provided.**
- **Data set: Preprocessed matrices from the SuiteSparse matrix collection (they are included in the sources).**
- **Run-time environment: Python3 with required packages (included in the virtual machine image).**
- **Hardware: Platform supporting Virtualbox.**
- **Output: Prediction accuracy and predicted format.**
- **Workflow frameworks used?: Linux shell scripts.**
- **Publicly available?: Yes.**
- **Artifacts publicly available?: Yes. The submission includes a wrap-up for important models.**
- **Artifacts functional?: Yes. We provide Python API and wrap scripts for easy execution.**
- **Artifacts reusable?: Yes.**
- **Results validated?: Yes. Some major results will be validated.**

A.3 Description

A.3.1 How delivered

We provide a Virtual Machine image to include source and data for evaluation.

A.3.2 Hardware dependencies

We have measured the performance and collected the data on Nvidia GeForce TITAN X GPU. The collection of the performance data require 500G size of matrices and several days of running the experiment. So we provide the collected performance data in our delivery so the reviewers can skip the data collection part. As a result, there is no particular hardware restriction for the evaluation. Any platform which can run a recent Virtualbox can be used to evaluate it.

A.3.3 Software dependencies

VirtualBox We used VirtualBox 5.0.40 to prepare the image. It should also be compatible with similar versions.

Inside the image For the environment of virtual machine image, it mainly includes the following software:

- The guest OS is `Ubuntu 17.10`.
- `Python 3.6` is the main programming language.
- `Tensorflow 1.0` is the framework used for the deep learning model.
- Python packages `Numpy`, `cffi` are required for the execution.
- Datasets are generated from the original matrices and split as training data and testing data.

A.3.4 Data sets

The data sets are the representation extracted from the sparse matrix collection as well as the SpMV performance measured on each sparse matrix with four formats (COO, CSR, DIA, ELL). They are included in the `data` directory.

A.4 Installation

If the virtual machine image is used for evaluation, the image just needs to be imported to the installed Virtualbox software. Both the *username* and *password* for the GuestOS is *ubuntu*. The working directory is `/home/ubuntu/dnnspmv`, which is also the root directory for the source code and dataset.

A.5 Experiment workflow

For a quick evaluation, just run the provided shell script:

Listing 1. Run the wrapping script

```
cd /home/ubuntu/dnnspmv
./run.sh
```

The program `dnnspmv/dnnspmv/model/spmv_model.py` contains the main model and APIs for the selection model. It also provides wrapper interface to run as a script in three modes:

Training mode The script first trains the CNN-model with the training dataset

`dnnspmv/dnnspmv/data/train-data.npz`.

The computation graph and the state of the trained model will be saved for reuse since the training time is expensive. The command is:

Listing 2. Train the model

```
cd /home/ubuntu/dnnspmv
python3 dnnspmv/model/spmv_model.py train
```

Testing mode The script loads and restores the state of the model. Then it tests the model against the testing dataset

`dnnspmv/dnnspmv/data/test-data.npz`.

The command is:

Listing 3. Test the model

```
cd /home/ubuntu/dnnspmv
python3 dnnspmv/model/spmv_model.py test
```

Predict mode Given a sparse matrix in *matrix market* format (`.mtx`), the script predicts the best format and output the result. The command is:

Listing 4. Run prediction

```
cd /home/ubuntu/dnnspmv/dnnspmv
python3 model/spmv_model.py predict data/
    ↪ example.mtx
```

A.6 Evaluation and expected result

Run the wrapping script in Listing 1 will results in two output.

1. The accuracy of the prediction model on the test dataset. The accuracy should be larger than 90%.
2. The predicted format for a given matrix. It shows one of the COO, CSR, DIA, ELL. For the `example.mtx`, it should output CSR.

A.7 Note

The above virtual image is only for quick artifact evaluation. The complete workflow and dataset will be released on GitHub (https://github.com/yzhao30/dnnspmv) in the future.