Enabling Runtime SpMV Format Selection through an Overhead Conscious Method

Weijie Zhou, *Student Member, IEEE,* Yue Zhao, *Student Member, IEEE,*
Xipeng Shen, *Senior Member, IEEE,* Wang Chen

Abstract—Sparse matrix-vector multiplication (SpMV) is an important kernel and its performance is critical for many applications. Storage format selection is to select the best format to store a sparse matrix; it is essential for SpMV performance. Prior studies have focused on predicting the format that helps SpMV run fastest, but have ignored the runtime prediction and format conversion overhead. This work shows that the runtime overhead makes the predictions from previous solutions frequently sub-optimal and sometimes inferior regarding the end-to-end time. It proposes a new paradigm for SpMV storage selection, an overhead-conscious method. Through carefully designed regression models and neural network-based time series prediction models, the method captures the influence imposed on the overall program performance by the overhead and the benefits of format prediction and conversions. The method employs a novel two-stage lazy-and-light scheme to help control the possible negative effects of format predictions, and at the same time, maximize the overall format conversion benefits. Experiments show that the technique outperforms previous techniques significantly. It improves the overall performance of applications by 1.21X to 1.53X, significantly larger than the 0.83X to 1.25X upper-bound speedups overhead-oblivious methods could give.

Index Terms—SpMV, High Performance Computing, Program Optimization, Sparse Matrix Format, Prediction Model

1 Introduction

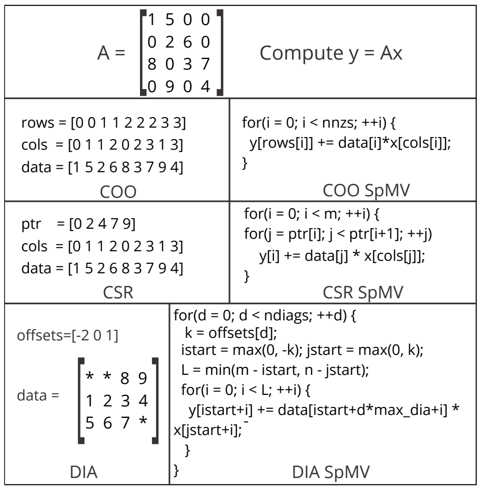
Sparse matrix-vector multiplication (SpMV) is an important kernel, and is widely used in many applications, such as linear equation system solvers [1], machine learning [2], [3] and graph analytics [4], [5], [6]. Since its performance is essential for these applications, it has motivated a large volume of research on optimizing its performance [7], [8], [9], [10], [11].

For the sparsity of matrices in SpMV, the matrices are usually represented in a compact format in memory with only non-zero elements stored. It saves space and also reduces memory footprint. For instance, Coordinate (COO) format uses three arrays to represent a sparse matrix, explicitly storing the row indices and column indices and values of all non-zero entries in the sparse matrix. Figure 1 illustrates the format and several other formats.

Through the years, various storage formats have been proposed for diverse application scenarios and computer architectures [9], [13], [14], [15], [16], [17], [18], [19], [20]. As observed in numerous studies [9], [12], [18], [19], [21], [22], [23], 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 [12]).

Storage format selection is to the selection of the proper format to represent sparse matrices in memory. The problem is challenging. There is no single format that has been found optimal for all sparse matrices. The proper format of a matrix depends on many factors, including the characteristics of the sparse matrix, the hardware architecture, the implementation of the SpMV library, the application, and so on.

The problem has drawn some prior research efforts. Li et al. [12] have developed a decision tree-based classifier that, for a given sparse matrix, predicts the storage format on which SpMV runs fastest. Follow-up studies have built similar predictors through other machine learning models [22], [24], [25].



$$A = \begin{bmatrix} 1 & 5 & 0 & 0 \\ 0 & 2 & 6 & 0 \\ 8 & 0 & 3 & 7 \\ 0 & 9 & 0 & 4 \end{bmatrix}$$

```
rows = [0 0 1 1 2 2 2 3 3]
cols  = [0 1 1 2 0 2 3 1 3]
data = [1 5 2 6 8 3 7 9 4]
```

```
for(i = 0; i < nnzs; ++i) {
  y[rows[i]] += data[i]*x[cols[i]];
}
```

```
ptr   = [0 2 4 7 9]
cols  = [0 1 1 2 0 2 3 1 3]
data = [1 5 2 6 8 3 7 9 4]
```

```
for(i = 0; i < m; ++i) {
for(j = ptr[i]; j < ptr[i+1]; ++j)
   y[i] += data[j] * x[cols[j]];
}
```

offsets=[-2 0 1]

$$data = \begin{bmatrix} * & * & 8 & 9 \\ 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & * \end{bmatrix}$$

```
for(d = 0; d < ndiags; ++d) {
  k = offsets[d];
 istart = max(0, -k); jstart = max(0, k);
 L = min(m - istart, n - jstart);
 for(i = 0; i < L; ++i) {
   y[istart+i] += data[istart+d*max_dia+i] *
x[jstart+i];
  }
}
```

Fig. 1: Three sparse matrix storage formats and their corresponding SpMV pseudo-code (adapted from [12]); *rows*, *cols*, *data* record the row indices, column indices and values of each nonzero element; *ptr* records the beginning positions of each row in *cols*; *offsets* records the offsets of each diagonal from the principal diagonal; *nnzs*: # non-zeros.

Despite promise results these work has shown, they are all subject to an important limitation. They are overhead oblivious—that is, none of them have considered the implications of the overhead in employing the predictors and in adopting their predictions including the substantial overhead of the required format conversions.

We use Figure 2 to illustrate the principled issue. In prac-

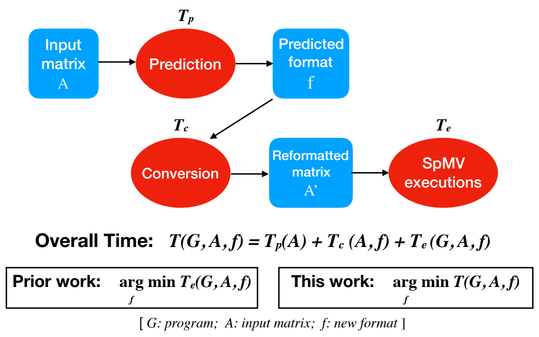

T_p
Input matrix A
Prediction
Predicted format f
T_c
Conversion
Reformatted matrix A'
T_e
SpMV executions
Overall Time: $T(G,A,f) = T_p(A) + T_c(A,f) + T_e(G,A,f)$
Prior work: $\arg\min_f T_e(G,A,f)$
This work: $\arg\min_f T(G,A,f)$
[G: program; A: input matrix; f: new format]

Fig. 2: The typical workflow for an application to benefit from improved matrix formats, and the objectives targeted by the previous studies and by this work.

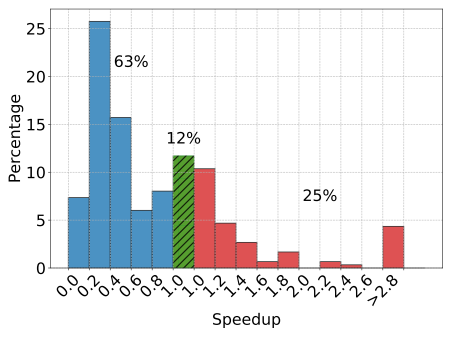


Fig. 3: The histogram of the overall speedups of program `PageRank` when it uses a previous decision tree-based predictor with an idealized 100% accuracy (for minimizing T_e in Figure 2). The annotated numbers indicate percentage of samples with speedup < 1, $= 1$, > 1 respectively; < 1 means slowdown. CSR is the default format.

tical settings, the input matrix to an SpMV-based application often comes with only one default storage format. To use a better format, two steps have to happen, the prediction on which format to use, and the conversion of the matrix into that new format. In many cases, these two steps need to happen during the execution of the application. The overall time is hence the sum of these overheads (T_p for prediction and T_c for conversion) and the execution time (T_e) of the SpMV on the new format.

Prior studies have all tried to build predictors to predict the format that minimizes T_e, ignoring the influence of T_p and T_c, which together can be even longer than the execution time of SpMV. As a result, even though previous predictors appear to have a quite good prediction accuracy, the formats predicted by them frequently give the inferior performance in practical usage.

The problem is fundamental. As Figure 3 shows, even if the prior method could achieve a perfect prediction accuracy, the results from them could still cause significant slowdowns to the overall executions.

The goal of this work is to solve the problem by putting the overhead into consideration for SpMV storage format prediction. The solution would need to create predictors that can accurately predict the overall effects (rather than just SpMV performance) of a new storage format.

Creating such a predictor is challenging. It faces a strand of new complexities.

First, unlike in prior work, when the objective becomes minimizing the overall time, matrix features are not sufficient any more, because we need to compare the benefits with runtime overhead. The quantitative benefits from a new format of a matrix depending on how many times SpMV gets called on the matrix, which depends on both the matrix and the program code itself. A sparse matrix is already difficult to characterize; adding program code makes the development of the solution even harder.

Second, the creation of *overhead-conscious* predictors requires an appropriate design strategy to deal with the overhead. The overhead could be dealt with explicitly by, for instance, building up one predictor for each kind of overhead and then subtracting it from the predicted benefits. It could also be dealt with implicitly by, for instance, building up a single predictor that takes the features of the input matrix and the program and predicts which format gives the overall best performance. Different strategies have different pros and cons. Understanding them and finding a suitable strategy is the second open research problem.

Finally, no matter what strategy is used, we eventually create some kind of predictor such that it can predict the best format to use for a given sparse matrix. The problem is that because such a predictor often has to run at the program runtime and its overhead is non-trivial (as it often requires extracting matrix features), its own overhead could also get in the way. If its own overhead already exceeds the benefits, the program would suffer slowdowns. If we create another predictor Y' to predict whether it is worth to run the format predictor, how do we deal with the overhead of that new predictor Y'? Creating yet another predictor Y''? How about its own overhead? This could lead us into a chicken-egg dilemma. How to effectively address this dilemma is the third research problem.

The rest of this paper presents our solutions to these challenges. Section 2 analyzes different strategies in creating an *overhead-conscious predictor*. It points out their pros and cons and gives an overview of our designed solution. The solution treats overhead explicitly through three predictors, respectively for conversion overhead, benefits to SpMV, and loop trip counts predictions. It lists the major challenges for materializing such a design effectively.

Section 3 describes our solution in detail. It presents a simple method called *lazy-and-light* to address the aforementioned chicken-egg dilemma. The proposed method features a loop trip count predictor and two regression-based predictors in a two-stage scheme. The paper describes our efforts for overcoming the various difficulties in constructing each of the three predictors by exploring different methods for building up these predictors, including the use of both modern Deep Learning methods (Long-Short Term Memory [26], CNN [27]) as well as traditional methods.

Section 4 reports the detailed assessment of the quality of the constructed predictors as well as the speedups the technique brings to the executions of some real-world applications. Our predictors predict the normalized format conversion time and the SpMV time with an average accuracy greater than 88% in most cases. The proposed overhead-conscious method improves the overall performance of applications by 1.21X to 1.53X, significantly larger than the 0.83X to 1.25X upper-bound speedups overhead-oblivious methods could give.

In summary, this paper makes the following major contributions:

- It points out the importance and challenges of being overhead-conscious for sparse matrix format selection.
- It analyzes the pros and cons of different design choices for creating an overhead-conscious solution, and then explains the reasons for favoring an explicit treatment of the overhead.
- It proposes the first end-to-end overhead-conscious solution for sparse matrix format selection, featuring a lazy-and-light scheme for slowdown prevention.
- It presents the use of both deep learning and traditional time series and machine learning methods for the creations of lightweight loop trip count predictors and describes the construction of an ensemble of predictive models for the effective cost-benefit analysis for SpMV format selection.
- It gives a comprehensive evaluation of the technique on 2757 matrices and four real-world applications, demonstrating the significant benefits of the proposed overhead-conscious method.

2 Strategies and Challenges

As Figure 2 in Section 1 has mentioned, the runtime of an SpMV-based program execution consists of the time to predict what format of matrices is desirable, the time to convert the matrix into that format, and the time to run the SpMV-based program on the matrix in the new format. The first two parts are runtime overhead.

We have considered various designs to treat the overhead for minimizing the overall execution time.

2.1 Implicit Versus Explicit Treatment

The first set of designs we considered treat the overhead implicitly. They directly predict the overall execution time of the SpMV-based program.

For instance, one design we had is to train a predictor p that takes a program code, the original matrix, and a certain matrix format as input and predicts the overall runtime of the program. It can be represented as the following $T_{overall} = p(G, A, f)$. The difficulty is that the predictor must apprehend the influence of the features of the program G, the matrix A, and the format f at the same time. All three components could have many dimensions to consider, with the program being the most complex component.[1]

An alternative design is to build a predictive model p_G specific to each given program G. As it is specific to a particular program, it needs to learn only about the influence of A and f: $T_{overall} = p_G(A, f)$, which simplifies the construction process. However, the catch is the loss of the generality of the constructed predictor. One would need to go through the time-consuming process of predictor construction for each program.

1. A variant of the predictor is to directly predict which format is the best (rather than time), which is still subject to the influence of all the components.

Overall, methods with an implicit treatment to overhead suffer a tension between model generality and construction complexity.

To address the tension, we use a design that takes an explicit treatment to the overhead. This design is based on a more detailed view of the overall program runtime. For a given matrix used by one or more SpMV statements in a program, if we allow format conversion of the matrix based on some format selector, the whole program execution time can be regarded as follows:

$$T_{overall} = T_{predict} + T_{convert} + (\sum_i T_{spmv(i)} * N_i) + T_{other}, \tag{1}$$

where,

- $T_{overall}$: the overall execution time of the whole program.
- $T_{predict}$: the time to predict what format to use for a given matrix.
- $T_{convert}$: the time to convert the matrix into the desired format.
- $T_{spmv(i)}$: the runtime of the ith SpMV statement on the matrix.
- N_i: the number of times the ith SpMV statement is invoked on the matrix.
- T_{other}: the time spent by other parts of the program.

In all SpMV-based applications that we have examined, T_{other} is largely independent to the matrix format as those parts of code tend to use the SpMV results rather than the matrix itself. For them, T_{other} can be ignored in the format selection; we need to consider only the first three terms on the right-hand-side of $T_{overall}$. So the goal becomes to minimize

$$T_{affected} = T_{predict} + T_{convert} + (\sum_i T_{spmv(i)} * N_i), \tag{2}$$

Our design is to build separate predictors to directly predict the overhead, single SpMV time $T_{spmv(i)}$, and N_i respectively. They each work across programs, matrices, and formats, providing good generality. At the same time, they avoid many complexities the implicit designs face. For instance, the overhead and $T_{spmv(i)}$ are not affected by the features of the program G; $T_{convert}$ is determined by the matrix and formats only; $T_{spmv(i)}$ is determined by the matrix and the used format only (for a given SpMV library); N_i is influenced by the features of the program G and the matrix, but is independent of the matrix formats.

Because of the strengths in both simplicity and generality of the explicit design, we use it as the base for our developed solution.

2.2 Challenges

To effectively materialize the design, there are three major challenges.

(1) The first is that unlike previously built predictors that give out qualitative prediction results (i.e., which format works the best), the three predictors we need to build are all quantitative predictors, giving out numerical

predictions. Consequently, the machine learning methods that showed effectiveness in the previous predictors cannot be applied to our problem, and the features of matrices/formats/programs may need to be reexamined for the needs of predictors for overhead-conscious methods. Identifying suitable machine learning methods and features to use is the first question we must answer. This challenge relates to the constructions of all our predictors.

(2) The second challenge is specific to the prediction time $T_{predict}$. It is the chicken-egg dilemma mentioned in the introduction. The prediction time $T_{predict}$ could be substantial as it typically requires the extraction of matrix features. Because the prediction happens during runtime, if the prediction result is to keep the format unchanged, the prediction would incur only runtime overhead and result in slowdowns to the program execution. If we add a predictor for predicting $T_{predict}$, that added prediction itself suffers the same problem.

(3) The third challenge is specific to the prediction of N_i. The value of N_i is usually determined by the tripcount(s) of the loop(s) surrounding the call of SpMV. Predicting the tripcount of a loop is not easy as it depends on the algorithm implemented in the loop and the data involved in the loop execution. The former varies from program to program, and the latter varies even across the different runs of the same program. How to automatically model the relations between programs and loop tripcounts is a difficult problem—it is in general equivalent to the halting problem the answer to which is undecidable.

In the next section, we present our solution in detail and explain how it addresses all the challenges.

3 Overhead-Conscious Predictor in Detail

Previous format selection systems are oblivious to the overhead terms, and only need to make qualitative predictions (i.e., which format gives the shortest SpMV time). Hence, they all formalize the problem as a classification problem. In contrast, as the previous section has explained, when taking the overhead into consideration, we need the predictors to provide quantitative predictions. We can no longer model it as a classification problem. Instead, we need to build regression models to predict numerical values. Regression models are models that take in some feature values (which could be numerical or categorical) and output some numerical predictions. We next explain our regression-based solution. We first present our two-stage lazy-and-light scheme and how it helps with risk control, and then explain some other important details in the predictors' construction.

3.1 Two-Stage Lazy-and-Light Scheme

As the previous section has mentioned, one of the barriers for deploying format predictors during runtime is that the predictor's own overhead could already cause large slowdowns to the overall execution if the SpMV is called for only a few times on a given matrix. It is primarily due to the time needed for the predictor to extract the features from the given matrix (running the predictor takes little time as mentioned in the previous subsection.)

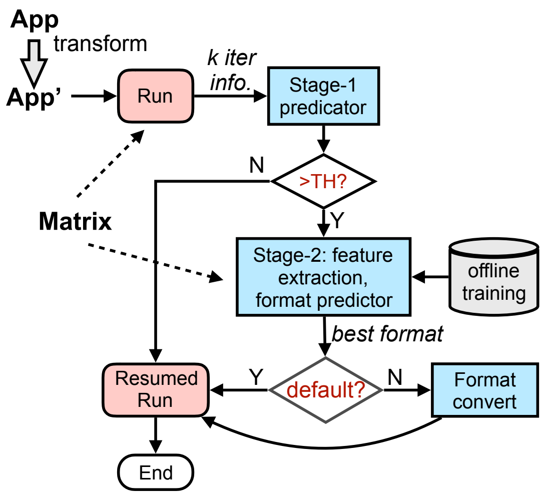


Fig. 4: The two-stage prediction system.

Our solution is a two-stage lazy-and-light scheme. As Figure 4 shows, the scheme consists of two stages of predictions. The first one is a lightweight predictor of the number of times the SpMV gets called on a matrix. The second stage does more sophisticated predictions and decides what format is the best format to use. The first predictor serves two purposes. First, by comparing the predicted value LC to a threshold TH, it decides whether it is worthwhile to invoke the second stage of prediction. If $LC < TH$, no further prediction will be done and the program runs with the default format; otherwise, the second-stage prediction will be done, and depending on the prediction result, the matrix may remain as it is or be converted to a new format and used in the rest of the program execution. Second, if the second stage is invoked, the predicted value LC is useful for computing N_i (number of times SpMV runs on a certain matrix) in the Equation (2), which is required in the format selection.

Such a scheme is designed to prevent the large prediction overhead from causing significant slowdowns when the executions are short-run executions. For the scheme to work effectively, it is important to ensure that the first stage takes virtually no time but still can provide reasonably prediction accuracy, and at the same time, the influence of its prediction errors could be controlled as much as possible.

We have explored several ways to implement the first-stage predictor. One method is to build it by modeling the relations between the features of matrices and the tripcounts of the SpMV-containing loops, such that the model can predict the tripcounts for an arbitrary matrix given to that loop.

However, the tripcounts are usually determined by the convergence of the program and the convergence of many applications depends on the dynamic value of the data. Our attempts on directly using matrix features to build predictors (through XGBoost) showed poor prediction results for tripcounts. The method also incurs substantial runtime overhead as it needs the collection of matrix features.

We eventually settled on the following time-series–based lazy predictor and found it meets our needs well. We observe that the loop surrounding SpMV in an SpMV-based

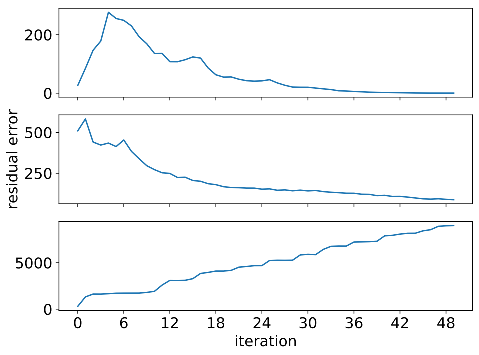


Fig. 5: Examples of residual error for the first 20 iterations.

application is often a loop for convergence, thus the loop tripcount is determined by the convergence of the algorithm and implementation. In a linear solver, for instance, each iteration of the loop computes the error from the current solution and the loop terminates when the error is smaller than a predefined threshold. We call the error the *progress indicator* of the loop. Other kinds of applications may have other kinds of *progress indicators*.

For example, in Figure 5, a CG linear solver is applied to solve the equation $\mathbf{Ax} = \mathbf{b}$ for the unknown $\mathbf{x}$. Figure 5 shows the residual error along the iteration number for three instances of the $(\mathbf{A}, \mathbf{b})$ pairs. In this example, the progress indicators are the residual error traces. In the first and second examples, the algorithm converges within a loop tripcount of 83 and 487, respectively, while in the third example, the algorithm diverges until the maximum tripcount 10000 is reached. So the intuition is that the convergence may be implied by the patterns and trends of the progress indicators.

Our solution is to build up a predictor that predicts the total number of iterations based on the sequences of the *progress indicators* of the first k iterations of the loop. Its usage will be lazy in the sense that it is invoked in execution of the target program only after the first k iterations of the loop have passed and their *progress indicators* are revealed.

Such a lazy design has three benefits. First, it avoids the slowdowns that the prediction (and prediction errors) may cause to the loop if the loop has very few (less than k) iterations. For such short loops, even small runtime overhead could have some significant impact. Second, as the collected *progress indicators* reflect the dynamic behaviors of this current execution, they provide the predictors with clues specific to this run. Finally, the constructed predictor runs quickly. It does not need extracting features from matrices. It only needs to record several values of the *progress indicator* and then do several linear algebraic calculations. They take virtually no time compared to an SpMV execution.

3.2 The First-Stage Prediction

In this part, we will describe the loop tripcount prediction model in detail. The input of the prediction model is a series of values that the progress indicator of the loop takes at the first number of iterations of the loop. We treat the problem as a time series prediction problem. Our initial design [28] employs auto-regressive integrated moving average (ARIMA) model [29], which fits linear models to time series data to predict future points in the series.

ARIMA does not give us accurate predictions (detailed in Section 4.3). We later resort to Deep Neural Networks (DNNs), specifically, the recurrent neural networks (RNNs) and the convolutional neural networks (CNNs), which both have shown promising results in other time series prediction problems [26], [27], [30]. Thanks to their high learning capacity for modeling nonlinear systems, these DNN methods give much better prediction results. We hence concentrate the description to these DNN-based models.

3.2.1 Prediction with CNNs

A CNN usually consists of a stack of layers of nodes. In Figure 6, the leftmost layer is the *input level*, with each node representing one element in the input vector (e.g., one point in the time series), 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. In Figure 6, the weights of every three edges connecting three input nodes with one layer-2 node form the filter $\langle w_1, w_2, w_3 \rangle$ at that level. The size of the filter is then 3 in this example. 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.

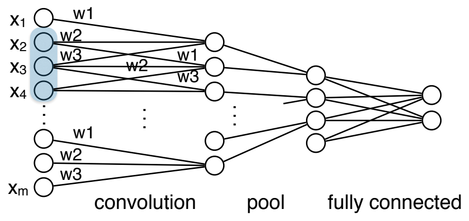


Fig. 6: A 1-dimensional convolutional network.

Our CNN predicator is a regressor since it predicts the continuous value instead of the classification label. The input is the progress indicators of the length of k and the output is the predicted tripcount. We build the CNN model based on [30] and an example structure is shown in Table 1. The CNN model has two one-dimensional convolutional layers and a fully connected layer. The size of the filter is constantly 5 for all the convolutional layers. The k is a tunable parameter and is chosen as 32.

3.2.2 Prediction with RNNs

An RNN architecture is a repetition of the same structures. Each unit structure corresponds to a one-time step. Figure 7 shows the diagram of a basic RNN layer. The $\langle x_1, x_2, \ldots, x_t \rangle$ is the input sequence of t time steps and

TABLE 1: The structure of the CNN prediction model. Layer 1 is the input layer and layer 6 is the output layer.

Layer No.	Layer	Output Shape	No. of filters	No. of Params
1	Conv1D	32	8	48
2	MaxPooling	16	8	0
3	Conv1D	16	8	328
4	MaxPooling	8	8	0
5	Flatten	64	N/A	0
6	Dense	1	N/A	65
-	Total	N/A	N/A	441

$\langle y_1, \ldots, y_t \rangle$ is the output of this RNN layer. The h_t in each unit is called the memory cell and holds the state for that time step.

At each time step, the h_t is calculated using the previous h_{t-1}, current input x_t, and the associated weights corresponding to the connections. After that, y_t is calculated using h_t and the associated weights. In practice, RNN layers are commonly stacked to form a multilayer RNN. Long Short Term Memory networks(LSTMs) [26] are efficient implementations of RNNs to deal with the gradient vanishing problem of the plain RNNs.

In the tripcount prediction problem, the input sequence is the progress indicator trace. Therefore, the trace length k determines the number of time steps. In our solution, the loop tripcount predictor is a two-layer LSTM network with the specification shown in the Table 2. The k is also chosen as 32 here.

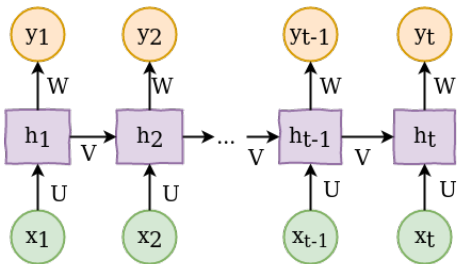


Fig. 7: A basic recurrent neural network.

TABLE 2: The structure of an LSTM prediction model.

Layer No.	Layer	Output Shape	No. of Params
1	Input	32, 1	0
2	LSTM	32, 40	6720
3	LSTM	1, 40	12960
4	Dense	1	41
-	Total	N/A	19721

3.2.3 DNN Model Training

The training process for both DNN models is similar. The main task is to determine the proper values of the parameters in the connections or 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 target. For a classification problem, it is the ground truth of its class, for regression problem, it is the value of the target output. The forward propagation on input through the network gives a prediction; its difference from its target 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 the DNNs, only forward propagation is needed to get the prediction.

3.2.4 Discussion

Thanks to the fact that the progress indicator trace is just a one-dimensional sequence of short length, our DNN-based predictors are lightweight in terms of the number of parameters and thus the execution time.

We make three notes. First, the goal of the two-stage scheme is not to ensure no slowdown (which could be easily achieved by avoiding any prediction or format conversion) but to maximize the overall speedups while avoiding large slowdowns in the unfavorable cases. The second stage of the prediction gets used only if the first stage predicts, after the first k iterations of the loop, that the loop will run for at least another TH iterations. We empirically set both k and TH to 15 in our implementations. Second, we do not claim the novelty of the time-series method for loop tripcount prediction. The approach shares lots of commonality with many other time-series–based program behavior predictions. The main contribution in this part is that we go around the prediction overhead dilemma through this lazy-and-light two-stage design. Finally, our discussion focuses on convergence loops. For regular `for` loops, the problem is easier; the loop tripcount can be directly attained from the loop bounds at runtime.

3.3 The Second-Stage Prediction and Feature Selection

The second stage predicts the best format to use. Recall that our original goal is to find the format to minimize $T_{affected}$ in Equation 2. By this time, the first-stage prediction has already decided to run the second-stage prediction. So the same $T_{predict}$ is incurred no matter what format the predictor chooses. Therefore, the original goal is equivalent to minimize

$$T_{convert} + \sum_i T_{spmv(i)} * N_i.$$

We observe that if we divide the goal formula with a constant (normalization), the result of minimizing the normalized value is equivalent to minimizing the original formula.

This observation is useful because, in our explorations, we found that directly predicting $T_{convert}$ and $T_{spmv(i)}$ is not as easy as predicting their normalized values.In our work, we use the $T_{spmv}(CSR)$ as the normalizer. The plausible reason is that some environment biases common to the three times are canceled out by the divisions.

So, we build two predictors for the second stage prediction. They respectively predict the normalized conversion time and the normalized SpMV time on each matrix format. For a given old matrix format and a new format, both normalized times are primarily determined by the matrix features. The predictors are regression models on matrix features, constructed with XGBoost. XGBoost [31] is one of the most widely used tree boosting packages, and has repeatedly proven competitive in many previous machine learning tasks [32], [33]. Meanwhile, it is based on a collection of simple tree structures and runs efficiently.

The construction process is straightforward. For instance, for the predictor of normalized $T_{convert}$ from given old to new formats, we collect the normalized $T_{convert}$ values on many matrices. For each, we create a tuple $(feature_1, feature_2, \cdots, feature_m, normalized\ T_{convert})$, where $feature_i$ is the value of the ith feature of the matrix. XGBoost then takes these tuples as training data and automatically constructs the predictor that predicts the normalized $T_{convert}$ from the features of an arbitrarily given matrix.

The main challenge in building the regression models is on the identification of the important features of a sparse matrix. The feature should capture the characteristics of the matrix for the studied problem well. On the other hand, more features may increase the feature extraction overhead and artificially increase the needed number of training data to form the predictor. So the optimal choice of features is a trade-off among expressiveness, cost, and simplicity.

Prior studies [7], [12], [34] have proposed more than 60 features to store the meta information about a matrix. It is essential to find those that are important for our predictors. Fortunately, a benefit of using ensembles of decision tree methods like XGBoost is that they can automatically determine feature importance from a trained predictive model. To be specific, an XGBoost model is built using the whole feature set in Table 3. As a side product, the importance score of each feature can be calculated by the algorithm, allowing features to be ranked and compared to each other. The importance score indicates how useful each feature is in the construction of the tree-boosting model. Features with low importance score can be automatically pruned until the minimal set of features is retained without sacrificing the prediction performance.

In addition to the feature selection, some other standard methods are used in our predictors' constructions, including grid search for parameter auto-tuning and cross-validation for overfitting prevention.

The deployment of the overhead-conscious method is currently through library. For a given application, to use the method for runtime format selection and conversion, the user just needs to replace the original SpMV call with our customized SpMV call, and insert code to record the values of the *progress indicator* of the surrounding loop. The customized SpMV call adds a wrapper to SpMV such that when the appropriate conditions are met, it calls the construction of Stage-1 predictor, or calls the Stage-2 predictor (which needs to be built only once on the system of interest) and the format conversion when necessary. The simplified pseudocode in Algorithm 1 demonstrates how to deploy the SpMV format selection process on an application `PageRank`.

4 EVALUATION

In this section, we report the evaluations of the efficacy of the proposed overhead-conscious format selection for SpMV-based applications.

TABLE 3: The full set of candidate features of a matrix.

Parameter	**Meaning**
M	number of rows
N	number of columns
NNZ	number of nonzeros (NZ)
Ndiags	number of diagonals
NTdiags_ratio	the ratio of "true" diagonals (occupied mostly by NZ) to total diagonals
aver_RD	average number of NZ per row
max_RD	maximum number of NZ per row
min_RD	minimum number of NZ per row
dev_RD	the deviation of number of NZ per row
aver_CD	average number of NZ per column
max_CD	maximum number of NZ per column
min_CD	minimum number of NZ per column
dev_CD	the deviation of number of NZ per column
ER_DIA	the ratio of NZs in DIA data structure
ER_RD	the ratio of NZs in row-packed structure
ER_CD	the ratio of NZs in column-packed structure
row_bounce	average diff. between NNZs of adjacent rows
col_bounce	average diff. between NNZs of adjacent cols
d	density of NNZ in the matrix
cv	normalized variation of NNZ per row
max_mu	$max\_RD - aver\_RD$
blocks	number of non_zero blocks
mean_neighbor	average number of NZ neighbors of an element

4.1 Experimental setup

4.1.1 Hardware

The evaluations are on a CPU-GPU platform as detailed in Table 4.

TABLE 4: Hardware platforms used in the experiments.

	Intel® CPU	NVIDIA® GPU
CPU	Xeon E5-1607	GeForce GTX TITAN X
Freq.	3.00 GHz	1.08 GHz
Cores	4	3072
Memory	16GB DDR3 1.9 GHz	12GB GDDR5 3.5 GHz
Memory Bandwidth	34.1 GB/s	168 GB/s
OS/Driver	Ubuntu 16.04	CUDA 8.0
Compiler	GCC (6.2)	NVCC (8.0)

4.1.2 Library, Formats, Applications

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 an SpMV library that can work with multiple matrix formats.

Algorithm 1 The demonstration of the deployment of the overhead-conscious format selection on an simplified application `PageRank`.

```
1: initialization
2: format = "CSR" // default
3: for $iter \leftarrow 0$ to $maxNumLoops$ do
4:     // original loop body
5:     kernel(format).spmv(inMatixPtr, inVectorPtr, outVectorCPtr);
6:     calcSum(damping, outVectorPtr, sum);
7:     err = l2Norm(inVectorPtr, sum);
8:     inVectorPtr = sum;
9:     if $err < tolerant$ then
10:        break
11:    // Start of deployment of format selection
12:    if $iter < k$ then
13:        indicators.append(err)
14:    if $iter == k$ then
15:        //stage 1 prediction
16:        N = loopCountPredict(indicators);
17:        if $N > NThreshold$ then
18:            // stage 2 prediction
19:            formatNew = formatPredict(inMatixPtr, N);
20:            if $formatNew != format$ then
21:                format = formatConvert(inMatixPtr);
22:    //End of deployment of format selection
```

In our evaluation, we focus on the CUDA-based SpMV libraries on the NVIDIA GPU. We adopt the NVIDIA® CUSP library [35], which supports COO, CSR, DIA, ELL, HYB formats. We supplement it with the cuSPARSE library [36] to support the BSR format. A previous work reports some promising performance of a new format CSR5 over some alternative formats and publishes the implementation [16]. We include its CUDA implementation to support CSR5 format. Format conversions are through the functions included in CUSP which runs on GPU. The set of formats covered in this study are limited to 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.

CSR is the most commonly used default format in SpMV-based applications. It is hence used as the default format in our experiments.

We create a simple software framework, named `SpMVframe`, to help with a focused study of SpMV performance. It consists of a loop with adjustable upperbound that surrounds a call to SpMV. In addition, we evaluate the technique on four real-world SpMV-based applications: `PageRank` [4] is the popular web page ranking algorithm, `BiCGSTAB` [37] implements the bi-conjugate gradient stabilized method, `CG` [38] is a conjugate gradient method, `GMRES` [39] is a linear equation system solver based on the generalized minimum residual method. The `PageRank` program is modified from an existing CUDA implementation [40] and the three linear solvers are based on the implementation from the Paralution project [41]. The modifications are for instrumenting the application to record the progress indicator and the iteration number at runtime.

TABLE 5: The conversion (CSR to other formats) time normalized by a single SpMV on CSR.

Other format	Conversion cost
COO	9
DIA	270
ELL	102
HYB	147
BSR	37
CSR5	26

Besides, the DNN-based loop tripcount predictors are built with the TensorFlow [42] framework.

4.1.3 Dataset

Our experiments use the dataset from the SuiteSparse matrix collection [43]. These matrices include the 2757 real-world matrices (which were also used in the previous studies [12], [22]).

In our evaluation of the prediction model, we run the SpMV on all the matrices of all formats[2]. However, not all runs are valid as some formats impose extra limitations on matrices. For example, the DIA and ELL require the fill ratio (the ratio of zeros to be padded in the storage) is within some threshold. Some applications (e.g., linear solver) works on only matrices meeting certain conditions. Only valid runs are considered in the performance comparisons; more details are given during our following result discussions.

4.1.4 Cross Validation

For the evaluations on the SpMV format prediction, we separate testing data from training data through 5-fold cross-validations.

4.2 Impact of the Overhead on Format Selection

We first measure the impact of the format conversion overhead on format selection. This part uses no prediction but actual performance measurements. As Table 5 shows, converting a matrix to a different format takes a lot of time, equaling 9-270 calls of SpMV. The time differs across different formats. As a result, the format minimizing SpMV often does not give the best overall performance. It is reflected by the largely different distributions of matrices shown in Table 6 in terms of their favorite formats in overhead-conscious (OC) and overhead-oblivious (OO) cases. Table 6 also shows that when overhead is considered, the best format changes with the number of iterations of the SpMV-surrounding loop (format conversion is done only once for a matrix in a run). These results confirm the importance of being overhead conscious in selecting matrix storage format.

4.3 Performance Comparison of Loop Tripcount Predictors

As mentioned, our overhead-conscious solution consists of two stages. The first stage is a gateway mostly based on loop

2. For BSR, we choose the block size of 4 which shows statistically best performance

TABLE 6: The number of matrices that favor each of the formats in the conversion overhead-oblivious (*OO*) and overhead-conscious cases (when the loop has 100 or 1000 iterations).

Format	*OO*	*OC* (Iter=100)	*OC* (Iter=1000)
CSR	195	1490	965
DIA	30	6	27
ELL	107	0	76
HYB	54	4	13
BSR	943	201	632
CSR5	582	210	198

tripcount prediction. It is application specific. The second stage consists of our primary predictors that deal with the trade-off between format conversion overhead and conversion benefits. In this part, we will evaluate the performance of the loop tripcount predictor in the first-stage.

Algorithm 2 shows the pseudo code of the `PageRank` application. The most time-consuming part is the SpMV operation in line 4, surrounded by a loop. Algorithm 3 shows the pseudo code for the BiCGSTAB linear solver. Similarly, the SpMV operations in line 5, 6, 7, 9, 13 dominate the execution time. Such code patterns are representative for SpMV-based applications and also appear in the other two applications we have tested.

Algorithm 2 PageRank algorithm

```
Input: Sparse matrix $\mathbf{A}^{N\times N}$, damping factor $d$, convergence error $\epsilon$
Output: Vector $\mathbf{R}^{(n)}$
1: procedure PAGERANK
2:     $\mathbf{R}^{(0)}[i] \leftarrow 1/N, \forall i \in \{1...n\}$
3:     repeat
4:         $\mathbf{R}^{(k+1)} \leftarrow (1-\alpha)\mathbf{R}^0 + d(\mathbf{A}^T \times \mathbf{R}^{(k)})$
5:     until $|\mathbf{R}^{(k+1)} - \mathbf{R}^{(k)}| < \epsilon$
```

Algorithm 3 BICGSTAB algorithm

```
1: procedure BICGSTAB
2:     $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$, choose $\mathbf{r}'_0$: $\mathbf{r}_0 \cdot \mathbf{r}'_0 \neq 0$
3:     Set $\mathbf{p}_0 = \mathbf{r}_0$
4:     for $j = 0, 1, \ldots$ do
5:         $\alpha_j = (\mathbf{r}_j \cdot \mathbf{r}'_0)/((\mathbf{A}\mathbf{p}_j) \cdot \mathbf{r}'_0))$
6:         $\mathbf{s}_j = \mathbf{r}_j - \alpha_j \mathbf{A}\mathbf{p}_j$
7:         $\omega_j = ((\mathbf{A}\mathbf{s}_j) \cdot \mathbf{s}_j)/((\mathbf{A}\mathbf{s}_j) \cdot \mathbf{A}\mathbf{s}_j)$
8:         $\mathbf{x}_{j+1} = \mathbf{x}_j + \alpha \mathbf{p}_j + \omega_j \mathbf{s}_j$
9:         $\mathbf{r}_{j+1} = \mathbf{s}_j - \omega_j \mathbf{A}\mathbf{s}_j$
10:        if $\|\mathbf{r}_{j+1}\|/\|\mathbf{r}_0\| < \epsilon_0$ then
11:            break
12:        $\beta_j = (\alpha_j/\omega_j) \times (\mathbf{r}_{j+1} \cdot \mathbf{r}'_0)/(\mathbf{r}_j \cdot \mathbf{r}'_0)$
13:        $\mathbf{p}_{j+1} = \mathbf{r}_{j+1} + \beta_j(\mathbf{p}_j - \omega_j \mathbf{A}\mathbf{p}_j)$
14:    $x = x_{j+1}$
```

The convergence checking statements in those applications give the *progress indicators* for the Stage-1 predictor to use for its prediction of loop tripcounts. The four applications exhibit different patterns in loop tripcounts. `PageRank` has a quite stable pattern, with loop tripcounts in the range of [1, 93], while `BiCGSTAB` shows a much larger range [1, 100000] (100000 is the preset upper bound).

4.3.1 Data Normalization

Since the range of values of the progress indicators and tripcounts vary widely, they need to be normalized to make the deep learning models work properly. Following the common practice in DNN, we normalize the values to the range [-1, 1] through Algorithm 4. The normalization is asymmetric such that small values can get zoomed in after normalization; for many applications (e.g., linear solvers), the values of interest are residual errors in a converging process, and those values tend to be small for most iterations of the convergence loop. The asymmetric normalization helps preserve their influence when predictors are being constructed. For instance, Figure 8 depicts the residual errors of the CG linear solver applied on the BCSSTK01 matrix [43] before and after the normalization. The segment with residuals smaller than the initial residual becomes more prominent after the normalization, increasing their influence in the DNN learning process. This normalization algorithm is used in Algorithm 3 for normalizing progress indicator values.

Algorithm 4 Normalization of residual errors in Algorithm 3. k error points is recorded in the first k iterations.

```
1: Initial residual error $r_0 = 1$
2: procedure NORMALIZE
3:     for all $r_i$ do
4:         if $r_i \leq r_0$ then
5:             $r'_i = r_i/r_0 - 1$
6:         else
7:             $r'_i = (r_i - r_0)/(\max_{j=1,k}(r_j) - r_0)$
```

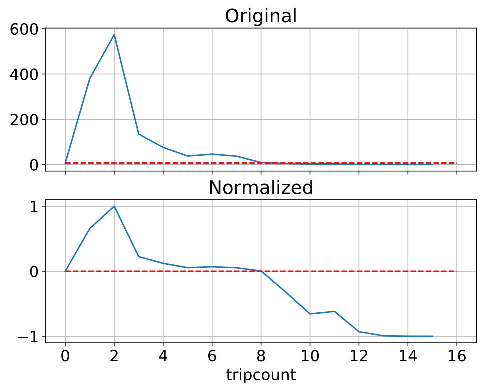


Fig. 8: Example of the residual error normalization.

For the prediction target—that is, loop tripcounts, we normalize it to the range of [0, 1] using *Min-Max* normalization. Because the maximum tripcount can be very large, we apply log scaling before the normalization to help reduce the variation.

4.3.2 Prediction Accuracy

In collecting the dataset for training and testing the stage 1 predictors, we run the application of interest on the valid matrices. The target applications are instrumented so that the progress indicators for the first k iterations and the total loop tripcount l are recorded. For each matrix, we get a tuple $< e_1, e_2, ..., e_k, l >$, where the progress indicators e_i's form the k-sized feature vector and the ground-truth loop tripcount l is the label. These tuples form the dataset for stage 1. We further split it into the training set and testing set in the cross-validation.

It is worth noting that we train a predictor separately for each application, for two reasons. First, the ground-truth loop tripcount is determined by the convergence of the specific application. Second, many applications have special requirements for their input matrices (e.g., symmetric, positive definite, etc).

For both the CNN and DNN models, we use the classic design—a stack of layers or cells. The DNN models have some hyper-parameters including the size of DNN models (i.e., the number of layers and the size of each layer and filter) and k (the number of iterations before stage 1 prediction starts). Generally speaking, a larger network has more learning capacity but also requires more training data and incurs higher runtime overhead. Note that the value of k affects the structure (# of CNN layers or LSTM cells) of the best DNN for the prediction for its impact on the size of the inputs to the DNN models.

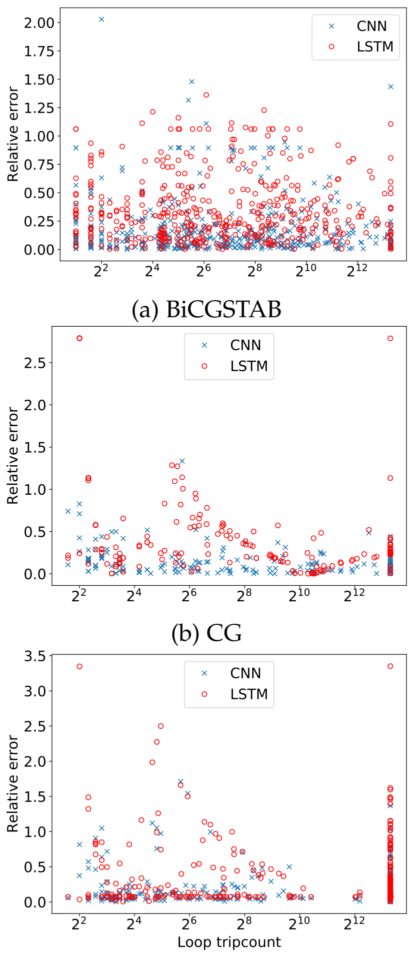
There is a tension on the value of k. The larger it is, the more data the predictors can use for more accurate prediction, but at the same time, the fewer iterations of the loop can benefit from the prediction. In addition, a larger k in general leads to a more complex DNN model which increases the runtime overhead. The best value of k is determined during the model construction process through an empirical search. By trying a list of options (5-30 in step 5), we observed 20 as an overall best choice.

To evaluate the performance of the predictors, we compare the prediction accuracy in terms of the *relative error* in Equation (3). Table 7 reports the average prediction errors of the CNN model, the LSTM model and the ARIMA model for each benchmark. *Relative error* in Equation (3) is used. The second column shows the number of tested matrices for each application. (Recall that only matrices meeting the input requirements of a benchmark are tested on that benchmark.) The third and fourth columns show the average and standard deviation of loop tripcounts respectively. The results are based on the 5-fold cross-validation, with 80% as the training data and the rest as the testing data in each round. The two DNN models outperform ARIMA significantly, confirming their advantages in modeling complex non-linear relations over traditional time series methods. Among all the benchmarks, `PageRank` is the easiest to predict for its small variations of loop tripcounts across runs on different matrices. To help examine the error distributions of the two DNN models on the other three benchmarks, Figure 9 plots their prediction errors on the three more challenging benchmarks.

Overall, the CNN model shows the best prediction accuracy for all benchmarks. In addition, the CNN model is faster to train and to use than the LSTM model, for its smaller size as Table 2 and Table 1 have shown. We hence choose the CNN model as our final loop tripcount predictor in our solution.

It is important to note that our ultimate goal is to decide whether to do the predictions and format conversions, rather than to get the precise loop tripcounts. Hence, there is a substantial amount of slack for tolerating tripcount prediction errors. For instance, even though a prediction of 10 for a 2-iteration loop means a 400% prediction error, the predictor still makes the right decision that no further predictions or conversions should be done as the loop is too small (smaller than the threshold). For a loop, tripcounts of 1000 and 2000 could lead to the same conclusion for our ultimate questions as they are both so large that the overhead in predictions and format conversions is trivial compared to the benefits. More specifically, for our Stage-1 predictor, the predicted tripcount is taken to compare with the threshold TH to decide whether it is worth proceeding into the more costly Stage-2 prediction. In this evaluation, both the pre-execution iteration number k and the threshold TH are chosen as 32 empirically. The optimal tuning of these parameters is left for future exploration.

So despite the sometimes quite big errors in the predicted tripcounts, our Stage-1 predictor correctly predicts whether the tripcounts (after the first stage) of the loop exceed the threshold in most of the times.



(c) GMRES

Fig. 9: Loop tripcount prediction errors. The x-axis is the actual loop tripcounts (in log2-scale).

TABLE 7: Prediction errors of different models. Both CNN and LSTM models are trained for 5000 epochs (LC: loop tripcount).

Application	No. of matrices	Avg. of LC	Std. of LC	Prediction error		
				CNN	LSTM	ARIMA
PageRank	1496	50.04	19.90	9.1%	9.7%	17%
BiCGSTAB	466	1203.80	2588.45	33.3%	38.0%	62%
CG	181	3634.37	4328.96	20.9%	34.2%	78%
GMRES	478	7816.46	4058.95	16.2%	21.3%	102%

4.4 Performance Comparison of Primary Predictors

In this part, we give a focused study on the quality of the primary predictors, and compare the format they select with those from previous overhead-oblivious methods. The next section will report the performance of the whole overhead-conscious solution on real applications.

We use our `SpMVframe` for this study; by allowing easy change of loop bounds, it makes it convenient to examine the tradeoff between conversion overhead and benefits.

4.4.1 Prediction Accuracy and Speedup Comparisons

Recall that the primary predictors predict the normalized format conversion time and the normalized SpMV time on the new format. We use *relative error* as the metric, defined as

$$\frac{|predicted\ value - actual\ value|}{actual\ value}. \tag{3}$$

Table 8 reports the relative errors of the predictions by our primary predictors on each of the studied matrix formats. The accuracy is over 88% in most cases.

TABLE 8: Prediction errors of normalized format conversion time and SpMV time

Format	No. of matrices	Error of conv. time	Error of SpMV time
COO	1911	9.6%	18.0%
CSR	1911	8.1%	7.0%
DIA	630*	8.8%	8.3%
ELL	1331*	8.6%	10.0%
HYB	1911	8.3%	8.0%
BSR	1911	10.7%	15.0%
CSR5	1911	13.9%	11.5%

*The library allows only matrices meeting certain conditions (e.g., number of diagonals exceeds 20% for DIA) for DIA and ELL.

Figure 10 shows that the predictions are sufficient for the primary predictors to select the best matrix formats correctly in most cases, and produces significant speedups. The bars in Figure 10 report the speedups obtained in three ways. The $Speedup_{OC}$ bars are the speedups from our primary predictors. The TB_{OC} bars are the upperbounds of the overhead-conscious method—that is, when the primary predictors have a perfect prediction accuracy. The TB_{OO} bars are the upperbounds of the results from overhead-oblivious methods, which are obtained by picking the format that actually minimizes SpMV time (without considering format conversion time).

As the results in Figure 10 show, because of the impact of the conversion overhead, the decisions from TB_{OO} cause large slowdowns when the SpMV-enclosing loop has a small number of iterations. When the number of iterations gets large, the speedups from that method is still lower than what the overhead-conscious method achieves. The difference between the $Speedup_{OC}$ bars and the TB_{OC} bars shows that the speedup loss due to the prediction error of our primary predictors is small across all different numbers of loop iterations. The results indicate the importance of being overhead-conscious, and suggest that the predictions from our primary predictors are accurate enough to keep most benefits of overhead-conscious matrix format selection.

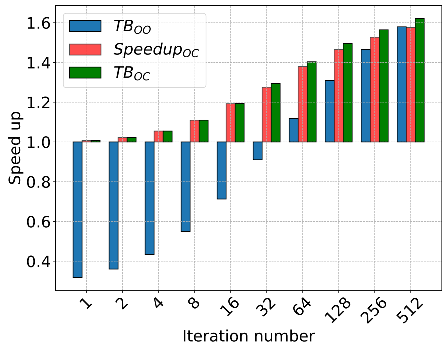


Fig. 10: The comparison of speedups based on *SpMVframe*. The baseline is the performance on the default CSR format.

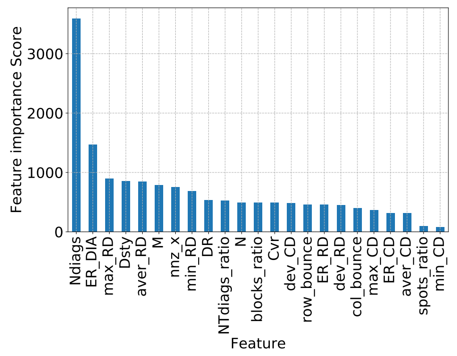
4.4.2 Feature Importance

Thanks to the tree-based structure of XGBoost, the predictor construction process can report the importance of each feature of the input matrix. Figure 11 shows the ranked list of the features that are the most important for the DIA and BSR formats in the T_{spmv} prediction. The importance of features varies from format to format. For the DIA format, the most important feature is Ndiags, while for the BSR format, the most important features are aver_RD and block_ratio due to its block based storage format.

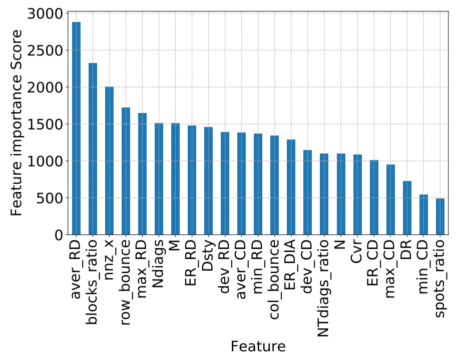
4.5 Performance on Entire Applications

In this part, we report the overall speedups our solution brings to four real-world applications.

The prediction errors raised in the first and second stages have some influence on the overall benefits of the predictions, but the overall results still remain positive. Table 9 (the $Speedup_{OC}$ column) shows the average speedups our method brings to the whole program executions (all runtime overhead is counted.) We also report the upperbound speedups (the TB_{OC} column) of our method when there



(a) DIA format



(b) BSR format

Fig. 11: Examples of feature importance ranking for T_{spmv}.

are no prediction errors and the upperbound speedups (the TB_{OO} column) of overhead-oblivious methods. Again, the performance of the default CSR format is used as the baseline. The prediction errors reduce the speedups of the overhead-conscious method to a certain degree. However, even with that influence, the overhead-conscious method still significantly outperforms the upperbounds of the overhead-oblivious methods. The average speedups are significant, ranging from 1.21X to 1.53X.

TABLE 9: Speedup of applications.

Application	Speedup		
	TB_{OO}	TB_{OC}	$Speedup_{OC}$
PageRank	1.08	1.57	1.53
BiCGSTAB	1.25	1.41	1.35
CG	0.83	1.24	1.21
GMRES	1.01	1.29	1.27

Table 10 shows the distributions of the selected formats. Figure 12 shows the histogram of the speedup for `PageRank`. Compared with Figure 3, our selector largely avoids performance slowdowns caused by unnecessary format conversions. Selected new formats work well for most cases.

Table 11 reports the details of the predictions on several matrices of different sizes and densities. The consideration of overhead leads our predictor to select entirely different formats from those selected by the ideal overhead-oblivious method on four out of the five matrices. For instance, on

TABLE 10: The numbers of matrices that favor each of the formats in the four applications.

	Applications	PageRank		BiCG		CG		GMRES	
	Schemes	OO	OC	OO	OC	OO	OC	OO	OC
F	CSR	71	313	32	117	36	119	117	271
o	DIA	12	1	8	1	8	5	9	7
r	ELL	38	3	9	6	27	19	17	10
m	HYB	24	1	4	0	12	4	8	3
a	BSR	367	129	166	90	115	47	291	167
t	CSR5	86	151	31	37	7	11	48	32

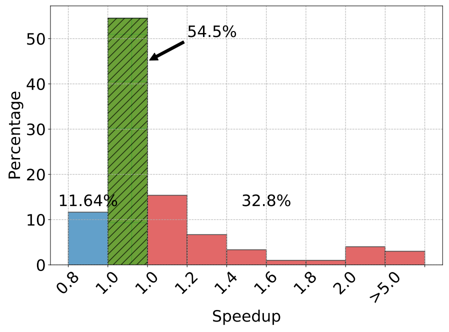


Fig. 12: The histogram of the overall speedups of program `PageRank` when it uses our OC format selector predictor. CSR is the default format.

matrix `shallow_water2`, even though HYB can make the SpMV run faster, our stage-1 predictor correctly predicts that it is not worthwhile because the runtime overhead would outweigh the benefits. By leaving the format unchanged, it avoids the substantial slowdowns the overhead-oblivious method incurs. Because stage-1 predictor takes virtually no time, it adds no noticeable overhead to the program execution. The stage-2 predictor has larger overhead (about 2X–4X SpMV time) as it needs to extract matrix features. However, the two-stage design and the lazy-and-light scheme ensure that it does not get invoked in short program executions, as in the `shallow_water2` case.

Overall, the upperbounds of speedups that previous overhead-oblivious methods can bring are only 0.83X–1.25X. (Recall that less than one means slowdown.) In contrast, our method, on average, improves the overall performance of applications by 1.21X–1.53X. The results demonstrate that the proposed method is effective in overcoming the limitations of prior methods in selecting the storage format for SpMV-based applications. The experiments focus on GPU, but as SpMV selection is also important for CPU [12], [44], we expect that the technique could help CPU executions as well; the exploration is left for future studies.

Regarding the prediction overhead, the time of both the DNN models and the XGBoost model is constantly less than $5ms$. Finally, the feature extraction overhead varies with a range of 2X–4X of an SpMV call.

5 Related Work

There has been a number of studies on format selection for SpMV. For example, the SMAT work [12] built a decision

TABLE 11: Speedup comparison for selected matrices.

Application	Matrix	NNZ	Row(Col)	Iter	$Format_{OO}$	$Format_{OC}$	$Speedup_{OO}$	$Speedup_{OC}$
PageRank	nlpkkt120	95117792	3542400	49	ELL	CSR5	0.89	1.297
PageRank	shipsec1	3568176	140874	48	CSR5	CSR5	1.17	1.17
PageRank	pwtk	11524432	217918	46	BSR	CSR5	0.95	1.245
BiCGSTAB	shallow_water2	81920	327680	11	HYB	CSR	0.03	1.0
CG	torso3	259156	4429042	45	ELL	CSR5	0.93	1.49

tree for selecting the best storage format for sparse matrix storage. A similar model was used in another study [22]. SVM classification was applied in some other work [24]. Another recent study [25] explored different machine learning models on matrix format selection and showed promising performance with XGBoost ensembles.

None of these studies have taken the overhead into account when predicting the best format. A recent work [44] discusses the "break-even point", which is the minimal number of SpMV calls needed for the conversion benefits to outweigh the overhead. This concept is overhead conscious, but the work does not integrate it into the prediction model, leaving the decision to users.

There are some other efforts trying to optimize the computations over sparse matrices, including building automatically performance tuning (auto-tuning) systems [12], [18], [19], [22], designing new sparse formats [9], [14], [15], and hand-tuning input- or architecture-related features [8], [9], [45].

In a broader scope, there has been a large body of work applying machine learning techniques to solve program optimization difficulties. Examples include some on algorithmic selections [46], [47], some on improving lower-level compiler optimizations [48], [49], [50], [51], and some on dynamic compilations and adaptations [52].

6 Conclusion

This paper has provided the first systematic exploration on how to construct overhead-conscious selectors of matrix format for SpMV-based programs. SpMV is important, but there are many other uses of sparse matrices. We foresee that the potential of the proposed techniques and methods (e.g., the two-stage lazy-and-light scheme, the explicit treatment of online overhead) may go well beyond SpMV format selection (e.g., selection of the best linear solvers for a given matrix [53].) How to unleashing the potential is left for future explorations.

Acknowledgments

We thank reviewers from IPDPS'18 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, CCF-1703487, and IBM CAS Fellowship. 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 or IBM.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Tech. Rep., 2006.

[2] A. Anandkumar, R. Ge, D. Hsu, S. M. Kakade, and M. Telgarsky, "Tensor decompositions for learning latent variable models," *The Journal of Machine Learning Research*, vol. 15, no. 1, 2014.

[3] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 77, 2017.

[4] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[5] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse matrix solvers on the gpu: conjugate gradients and multigrid," in *ACM transactions on graphics (TOG)*, vol. 22, no. 3. ACM, 2003, pp. 917–924.

[6] X. Yang, S. Parthasarathy, and P. Sadayappan, "Fast Sparse Matrix-Vector Multiplication on GPUs: Implications for Graph Mining," *Proceedings of the VLDB Endowment*, vol. 4, no. 4, pp. 231–242, 2011.

[7] R. W. Vuduc, "Automatic performance tuning of sparse matrix kernels," Ph.D. dissertation, Computer Science, University of California, Berkeley, 2003.

[8] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrixvector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, 2009.

[9] 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*, ser. SC '09. New York, NY, USA: ACM, 2009.

[10] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, "Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, vol. 2015-January, 2014.

[11] A. Li, W. Liu, M. R. Kristensen, B. Vinter, H. Wang, K. Hou, A. Marquez, and S. L. Song, "Exploring and analyzing the real impact of modern on-package memory on hpc scientific kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 26.

[12] J. Li, G. Tan, M. Chen, and N. Sun, "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*, ser. PLDI '13. New York,, USA: ACM, 2013.

[13] D. Langr and P. Tvrdik, "Evaluation criteria for sparse matrix storage formats," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 2, pp. 428–440, Feb. 2016.

[14] B.-Y. Su and K. Keutzer, "clSpMV: A cross-platform OpenCL SpMV framework on GPUs," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 353–364.

[15] 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*, 2011.

[16] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 339–350.

[17] W. L. N. S. Zhen Xie, Guangming Tan, "Ia-spgemm: An input-aware auto-tuning framework for parallel sparse matrix-matrix

multiplication," in *Proceedings of the 2019 International Conference on Supercomputing*. ACM, 2019.

[18] R. W. Vuduc and H.-J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *Proceedings of the First International Conference on High Performance Computing and Communications*, ser. HPCC'05, Berlin, Heidelberg, 2005.

[19] J. W. Choi, A. Singh, and R. W. Vuduc, "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*, ser. PPoPP '10, 2010.

[20] W. Liu and B. Vinter, "Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors," *Parallel Computing*, vol. 49, pp. 179–193, 2015.

[21] Y. Zhao, J. Li, C. Liao, and X. Shen, "Bridging the gap between deep learning and sparse matrix format selection," in *ACM SIGPLAN Notices*, vol. 53, no. 1. ACM, 2018, pp. 94–108.

[22] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on GPUs," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, USA: ACM, 2015.

[23] D. Merrill and M. Garland, "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*, ser. PPoPP '16, 2016.

[24] A. Benatia, W. Ji, Y. Wang, and F. Shi, "Sparse Matrix Format Selection with Multiclass SVM for SpMV on GPU," in *2016 45th International Conference on Parallel Processing (ICPP)*, 2016.

[25] 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.

[26] F. A. Gers, D. Eck, and J. Schmidhuber, "Applying lstm to time series predictable through time-window approaches," in *Neural Nets WIRN Vietri-01*. Springer, 2002, pp. 193–200.

[27] Y. LeCun, Y. Bengio *et al.*, "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[28] Y. Zhao, W. Zhou, X. Shen, and G. Yiu, "Overhead-conscious format selection for spmv-based applications," in *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018*, 2018, pp. 950–959.

[29] S. Makridakis and M. Hibon, "Arma models and the box–jenkins methodology," *Journal of Forecasting*, vol. 16, no. 3, 1997.

[30] A. Borovykh, S. Bohte, and C. W. Oosterlee, "Conditional time series forecasting with convolutional neural networks," *arXiv preprint arXiv:1703.04691*, 2017.

[31] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM, 2016, pp. 785–794.

[32] V. Sandulescu and M. Chiru, "Predicting the future relevance of research institutions-the winning solution of the kdd cup 2016," *arXiv preprint arXiv:1609.02728*, 2016.

[33] M. Volkovs, G. W. Yu, and T. Poutanen, "Content-based neighbor models for cold start in recommender systems," in *Proceedings of the Recommender Systems Challenge 2017*. ACM, 2017, p. 7.

[34] V. Eijkhout and E. Fuentes, "A proposed standard for numerical metadata," Technical Report ICL-UT-03-02, Innovative Computing Laboratory, University of Tennesse, Tech. Rep., 2003.

[35] N. Bell and M. Garland, "Cusp: Generic parallel algorithms for sparse matrix and graph computations," *Version 0.3. 0*, vol. 35, 2012.

[36] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse library," in *GPU Technology Conference*, 2010.

[37] H. A. Van der Vorst, "Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems," *SIAM Journal on scientific and Statistical Computing*, vol. 13, no. 2, 1992.

[38] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *Journal of research of the National Bureau of Standards*, vol. 49, pp. 409–436, 1952.

[39] Y. Saad and M. H. Schultz, "Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on scientific and statistical computing*, vol. 7, no. 3, 1986.

[40] Y. Liu and B. Schmidt, "Lightspmv: Faster csr-based sparse matrix-vector multiplication on cuda-enabled gpus," in *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*. IEEE, 2015, pp. 82–89.

[41] D. Lukarski and N. Trost, "Paralution project," *URL http://www.paralution. com. Accessed: December*, 2014.

[42] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning." in *OSDI*, vol. 16, 2016.

[43] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[44] B. Yilmaz, B. Aktemur, M. J. Garzarán, S. Kamin, and F. Kiraç, "Autotuning Runtime Specialization for Sparse Matrix-Vector Multiplication," *ACM Transactions on Architecture and Code Optimization*, vol. 13, no. 1, pp. 1–26, mar 2016.

[45] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13, 2013.

[46] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "PetaBricks: A language and compiler for algorithmic choice," in *PLDI*, Dublin, Ireland, Jun 2009.

[47] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *Proceedings of the 36th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, 2015.

[48] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman, "Finding effective compilation sequences." in *LCTES'04*, 2004, pp. 231–239.

[49] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather, C. Williams, M. O'Boyle, P. Barnard, E. Ashton, E. Courtois, and F. Bodin, "MILEPOST GCC: machine learning based research compiler," in *Proceedings of the GCC Developers' Summit*, Jul 2008.

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

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

[52] K. Tian, Y. Jiang, E. Z. Zhang, and X. Shen, "An input-centric paradigm for program dynamic optimizations," in *ACM Sigplan Notices*, vol. 45, no. 10. ACM, October 2010, pp. 125–139.

[53] S. Bhowmick, B. Toth, and P. Raghavan, "Towards low-cost, high-accuracy classifiers for linear solver selection," *Computational Science–ICCS 2009*, pp. 463–472, 2009.

Weijie Zhou received the BS degree in Communication Engineering from Jilin University and the MS degree in Electronic and Information Engineering from the Hong Kong Polytechnic University. She is working towards her Ph.D. degree in Computer Science at the North Carolina State University, with the research interest in the intersection of machine learning and computing systems.

Yue Zhao received the BS degree of Information Engineering from Shanghai Jiao Tong University and the Ph.D. degree of Computer Science from the North Carolina State University. His interest falls in Compilers, HPC program optimizations, Machine Learning, and so on. He is currently working for Facebook.

Xipeng Shen is a professor in Computer Science at the North Carolina State University. He is a receipt of the DOE Early Career Award, NSF CAREER Award, Google Faculty Research Award, and IBM CAS Faculty Fellow Award. He is an ACM Distinguished Member, ACM Distinguished Speaker, and a senior member of IEEE. He received his Ph.D. in Computer Science from the University of Rochester in 2006. His interest is in Programming Systems and Machine Learning.

Wang Chen is a senior manager for compiler development at IBM. He has extensive experience in compiler optimization for various workloads, including commercial, analytics, and High Performance Computing. He has led the IBM XL compiler team in development of C, C++, and Fortran OpenMP compilers to exploit NVIDIA GPU in the OpenPOWER systems, which enabled order of magnitude speed up for key US Department of Energy CORAL workloads. Wang is also a Research Collaboration Manager for Centre for Advanced Studies and was a co-chair of the program committee for CASCON 2018. His interest includes programming models for accelerated computing and cognitive based compiler optimization.