Overhead-Conscious Format Selection for SpMV-Based Applications

Yue Zhao, Weijie Zhou, Xipeng Shen
CSC, North Carolina State University
{yzhao30, wzhou9, xshen5}@ncsu.edu

Graham Yiu
IBM Toronto Software Lab
gyiu@ca.ibm.com

***Abstract*—Sparse matrix vector multiplication (SpMV) is an important kernel in many applications and is often the major performance bottleneck. The storage format of sparse matrices critically affects the performance of SpMV. Although there have been previous studies on selecting the appropriate format for a given matrix, they have ignored the influence of runtime prediction overhead and format conversion overhead. For many common uses of SpMV, such overhead is part of the execution times and may outweigh the benefits of new formats. Ignoring them makes the predictions from previous solutions frequently suboptimal and sometimes inferior. On the other hand, the overhead is difficult to consider, as it, along with the benefits of having a new format, varies from matrix to matrix, and from application to application. This work proposes a solution. It first explores the pros and cons of various possible treatments to the overhead in the format selection problem. It then presents an explicit approach which involves several regression models for capturing the influence of the overhead and benefits of format conversions on the overall program performance. It proposes a two-stage lazy-and-light scheme to help control the risks in the 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 1.14X to 1.43X, significantly larger than the 0.82X to 1.24X upperbound speedups overhead-oblivious methods could give.**

***Keywords*-SpMV, High Performance Computing, Program Optimizations, Sparse Matrix Format, Prediction Model**

I. Introduction

Sparse matrix vector multiplication (SpMV) is one of the most important, widely used kernels in many scientific applications (e.g., linear equation system solvers) [1], [2]. It is also often the performance bottlenecks of those applications [3], [4]. Maximizing the performance of SpMV is essential.

One of the 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 [5]–[11]. As observed in numerous studies [9]–[14], 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]). Meanwhile, 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.

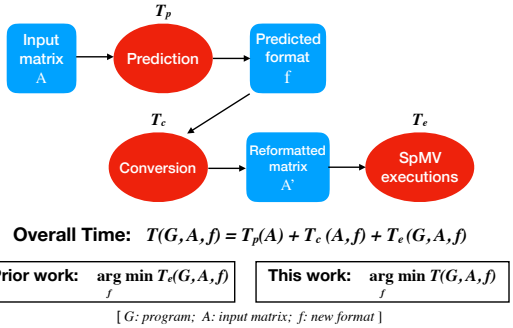

Fig. 1: The typical workflow for an application to benefit from improved matrix formats, and the objectives targeted by the previous studies and by this work.

There have been some efforts for creating automatic format selector for SpMV [12], [13], [15]. For instance, Li and others [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 [13], [15].

Although these studies have given some promise and valuable insights, 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.

Figure 1 illustrates the principled issue. In practical settings, the input matrix to a 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 overhead (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 quite good prediction accuracies, the formats predicted by them frequently give the inferior performance in practical usage. The problem is fundamental. As Figure 2 shows, even if

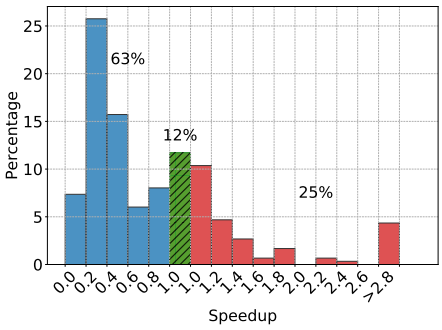

Fig. 2: The histogram of the overall speedups of program `PageRank` when it uses a previous decision tree-based predictor with an idealized 100% accuracy (for minimizing T_e in Figure 1). The annotated numbers indicate percentage of samples with speedup < 1, $= 1$, > 1 respectively; < 1 means slowdown.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

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, it has more factors to consider. Previous predictive models make predictions based on only the features of the input matrix because those are the only factor determining which format makes SpMV run the fastest on the given architecture. However, when our objective becomes minimizing the overall time, matrix features are not sufficient anymore, 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. 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 the 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. After some background knowledge in Section II, Section III 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 IV describes our solution in detail. It presents the methods for overcoming the various difficulties in constructing each of the three predictors. It also presents *lazy-and-light*, a simple method we develop to address the aforementioned chicken-egg dilemma through a lazy scheme and a lightweight loop trip count prediction.

Section V 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 1.14X to 1.43X, significantly larger than the 0.82X to 1.24X upperbound 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.
- It presents the first end-to-end overhead-conscious solution for sparse matrix format selection, including an ensemble of predictive models for cost-benefit analysis and a lazy-and-light scheme for slowdown prevention.
- It evaluates the technique on 2757 matrices and four real world applications, demonstrating the significant benefits of the proposed overhead-conscious method.

II. Background

Before presenting the solution, we first provide some background on sparse matrix formats and their usage in SpMV.

To efficiently store and process a sparse matrix, compressed data structures (a.k.a. storage formats) are used which store only nonzero entries. Various storage formats have been proposed [5]–[11], [14].

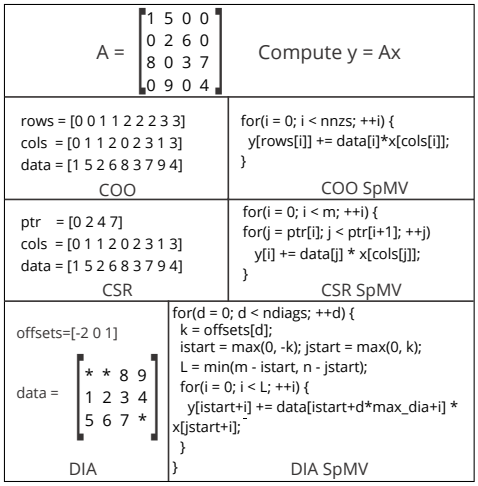

Fig. 3: Sparse matrix storage formats and their corresponding SpMV pseudo-code (adapted from [12]).

As examples, Figure 3 shows a sparse matrix represented in three formats respectively and their corresponding SpMV algorithms. Notations m, n, and $nnzs$ are used to represent the numbers 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 3, 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.

III. Design Choices and Challenges

As Figure 1 in Section I has mentioned, the runtime of a 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.

A. 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 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.

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:

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

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

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

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

[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.

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.

B. 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 four 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 in the same vein, the features of matrices/formats/programs found useful in the previous studies may not necessarily fit our predictors. Identifying the 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.

IV. Overhead-Conscious Predictor in Detail

This section first describes the choice of machine learning method we made for constructing the predictors, then presents our two-stage lazy-and-light scheme and how it helps with risk control, and finally explains some other important details in the predictors' construction.

A. Learning Method

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. There are many machine learning algorithms for constructing regression models, such as linear regression, Support Vector Regression (SVR), and so on. We select the learning method for our problem based on the following principles:

1) The algorithm should give good prediction results;

2) Due to the complexity of sparse matrices, the model should be robust and flexible in handling data with complex features;

3) As our predictors are for online predictions, the algorithm should be efficient;

4) It would be better if the produced models are interpretable.

Among the machine learning methods used in practice, regression tree-based models meet the four requirements well. They have advantages in simple data preparation, robust performance to nonlinear relationship, and easy interpretable results (as the created trees are composed of questions on input features). The regression models they produce are also fast to run as they involve only a small number of questions on the data features and several linear algebraic operations (the leaf nodes in the trees are typically linear functions of data features). Moreover, when combined with boosting methods, they have shown best prediction results and robustness in a variety of problems. In our work, we select the open-source package XGBoost [16], an efficient tree boosting system, to build our models. XGBoost is one of the most widely used tree boosting package, and has proven its effectiveness in many previous machine learning tasks [17], [18] [2].

B. 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.)

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

[2]Although deep learning is now popular, it is best for classification rather than regression.

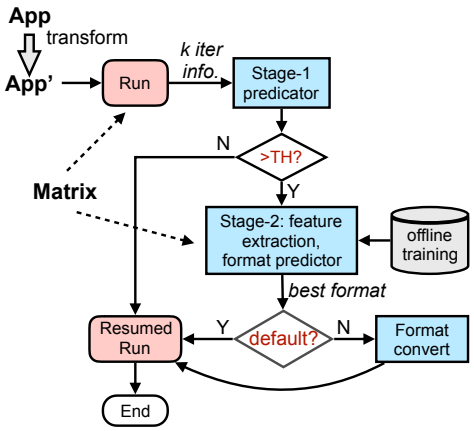

Fig. 4: The two-stage prediction system.

best format to use. The first predictor serves as a gateway; 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.

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 reasonable prediction accuracies, 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. The relations are too complicated to capture; our results based on XGBoost show disappointing prediction accuracies. 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. It is based on an observation that the loop surrounding SpMV in a SpMV-based application is often a loop for convergence. 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*. 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 an 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 a SpMV execution. The predictors are built with autoregressive integrated moving average (ARIMA) machine learning model [19], which is a simple but efficient time series analysis model.

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 contributions 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.

C. 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 (e.g., by $T_{spmv(0)}$ on the original matrix format). 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.

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 arbitrary 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 [12], [20], [21] 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 I. 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.

TABLE I: The full set of feature candidates 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 to total diagonals, true diagonals represents one occupied mostly with NZ.
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 nonzeros in DIA data structure
ER_RD	the ratio of nonzeros in row-packed (ELL) structure
ER_CD	the ratio of nonzeros in column-packed structure
row_bounce	average difference between NNZs of adjacent rows
col_bounce	average difference between NNZs of adjacent columns
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

TABLE II: 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)

V. Evaluation

In this section, we report the evaluations of the efficacy of the proposed overhead-conscious format selection for SpMV-based applications. A quick summary is that it predicts the normalized format conversion time and the SpMV time with an average accuracy greater than 88% in most cases. It improves the overall performance of applications by 1.14X to 1.43X, significantly larger than the 0.82X to 1.24X upperbound speedups overhead-oblivious methods could give. We next describe the methodology and the full results.

A. Experimental setup

a) ***Hardware****:* The evaluations are on a CPU-GPU platform as detailed in Table II.

b) ***Library, Formats, Applications****:* As Figure 3 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.

In our evaluation, we focus on the CUDA-based SpMV libraries on the NVIDIA GPU. We adopt the NVIDIA® CUSP library [22], which supports COO, CSR, DIA, ELL, HYB formats. We supplement it with the cuSPARSE library [23] 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 [8]. 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 a loop with adjustable upperbounds that surrounds a call to SpMV. In addition, we also evaluate the technique on four real-world SpMV-based applications: `PageRank` [24] is the popular web page ranking algorithm, `BiCGSTAB` [25] implements the bi-conjugate gradient stabilized method, `CG` [26] is a conjugate gradient method, `GMRES` [27] is a linear equation system solver based on the generalized minimum residual method.

*c) **Dataset**:* Our experiments use the dataset from the SuiteSparse matrix collection [28]. These matrices include the 2757 real-world matrices (which were also used in the previous studies [12], [13]).

In our evaluation of the prediction model, we run the SpMV on all the matrices of all formats. 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. And 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.

*d) **Cross Validation**:* For the evaluations on the SpMV format prediction, 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 valid matrices out to form a test set and uses the remaining valid matrices for training. It repeats the process for 5 times with a different subset of the dataset taken out as the test set.

B. 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 III shows, converting a matrix to a different format takes a lot of time, equaling 9-270 calls of SpMV. And 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 IV in terms of their favorite formats in overhead-conscious (OC) and overhead-oblivious (OO) cases. Table IV 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.

TABLE III: 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

TABLE IV: 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

C. Performance Comparison of Primary Predictors

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

*a) **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

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

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

Figure 5 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 5 report the speedups obtained in three ways. The

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

Format	No. of matrices	Error of conversion 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.

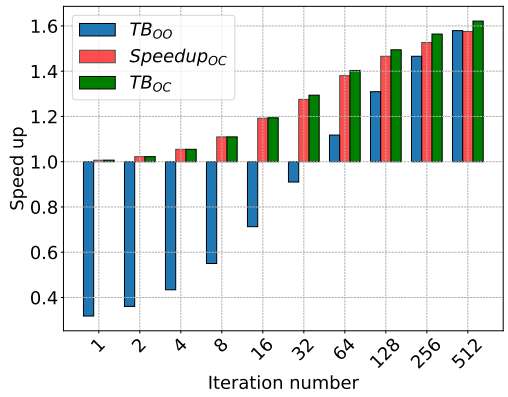

Fig. 5: The comparison of speedups based on *SpMVframe*. The baseline is the performance on the default CSR format.

$Speedup_{OC}$ 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 5 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 differences between the $Speedup_{OC}$ bars and the TB_{OC} bars show that the speedup loss due to the prediction errors 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.

D. Performance on Entire Applications

In this part, we report the overall speedups our solution brings to four real world applications, along with the evaluation of our first stage predictor.

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). The errors in the exact numbers of predicted loop iterations vary, from 17% average on `PageRank` to 62% on `BiCGSTAB`, 78% on `CG`, and 102% on `GMRES`. 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). And 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=15 to decide whether it is worth proceeding into the more costly Stage-2 prediction. 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: `PageRank` 93%, `BiCGSTAB` 82%, `CG` 76%, `GMRES` 65%.

TABLE VI: Speedup of applications.

Application	Speedup		
	TB_{OO}	TB_{OC}	$Speedup_{OC}$
PageRank	1.0762	1.4368	1.4307
BiCGSTAB	1.2454	1.3975	1.3375
CG	0.8246	1.1449	1.1416
GMRES	1.0136	1.2505	1.2034

The prediction errors have some influence on the overall benefits of the predictions, but the overall results still remain positive. Table VI (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 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 overhead-conscious method to a certain degree. It is most visible on `GMRES` which has the lowest prediction accuracies. 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 1.14X to 1.43X.

Table VII shows the distributions of the selected formats. Figure 6 shows the histogram of the speedup for `PageRank`. Compared with Figure 2, our selector largely avoids performance slowdowns caused by unnecessary format conversions.

TABLE VII: 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	320	32	114	36	122	117	292
o	DIA	12	0	8	3	8	5	9	5
r	ELL	38	0	9	5	27	23	17	8
m	HYB	24	0	4	0	12	5	8	0
a	BSR	367	129	166	96	115	45	291	164
t	CSR5	86	149	31	32	7	5	48	21

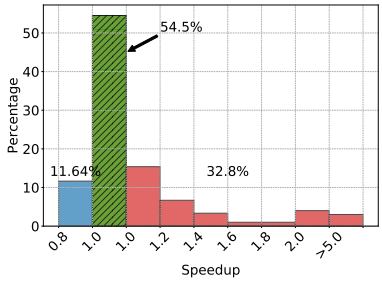

Fig. 6: The histogram of the overall speedups of program `PageRank` when it uses our OC format selector predictor. CSR is the default format.

Selected new formats work well for most cases.

Table VIII 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 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 will likely 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 some 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 0.82X–1.24X. (Recall that less than one means slowdown.) In contrast, our method, on average, improves the overall performance of applications by 1.14X–1.43X. 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], [29], 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 the time-series model and the XGBoost model is constant with value of $2ms$ and $5ms$, respectively. The feature extraction overhead varies with a range of 2X–4X of a SpMV call.

VI. Related work

There has been a number of studies on format selection for SpMV. For example, the SMAT work [12] built up a decision tree for selecting the best storage format for a sparse matrix storage. A similar classification-tree-based model was used in [13] and an SVM classification model was used in [15].

None of these studies have taken the overhead into account when predicting the best format. A recent work [29] 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 [9], [10], [12], [13], designing new sparse formats [6], [7], [11], and hand-tuning input- or architecture-related features [11], [30], [31].

In a broader scope, there have been a large body of work applying machine learning techniques to solve program optimization difficulties. Examples include some on algorithmic selections [32], [33], some on improving lower-level compiler optimizations [34]–[37], and some on dynamic compilations and adaptations [38].

VII. 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 method (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 [39].) How to unleashing the potential is left for future explorations.

Acknowledgment

We thank the anonymous 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, 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] M. F. Khairoutdinov and D. A. Randall, "A cloud resolving model as a cloud parameterization in the ncar community climate system model: Preliminary results," *Geophysical Research Letters*, vol. 28, no. 18, pp. 3617–3620, 2001.

[2] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 107–117, Apr. 1998.

[3] R. D. Falgout, "An introduction to algebraic multigrid computing," *Computing in Science Engineering*, vol. 8, no. 6, pp. 24–33, Nov 2006.

TABLE VIII: 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

[4] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse matrix solvers on the GPU: Conjugate gradients and multigrid," in *ACM SIGGRAPH 2003 Papers*, ser. SIGGRAPH '03. New York, NY, USA: ACM, 2003, pp. 917–924.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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: Springer-Verlag, 2005, pp. 807–816.

[10] 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. New York, NY, USA: ACM, 2010, pp. 115–126.

[11] 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, pp. 18:1–18:11.

[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] 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.

[14] 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. New York, NY, USA: ACM, 2016.

[15] 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)*. IEEE, aug 2016, pp. 496–505.

[16] 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.

[17] V. Sandulescu and M. Chiru, "Predicting the future relevance of research institutions - the winning solution of the KDD cup 2016," *CoRR*, 2016.

[18] 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.

[19] M. Hibon and S. Makridakis, "Arma models and the box–jenkins methodology," 1997.

[20] V. Eijkhout and E. Fuentes, "A proposed standard for numerical metadata," Tech. Rep., 2003.

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

[22] S. Dalton, N. Bell, L. Olson, and M. Garland, "Cusp: Generic parallel algorithms for sparse matrix and graph computations," 2014.

[23] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse library: A set of basic linear algebra subroutines for sparse matrices," in *GPU Technology Conference*, vol. 2070, 2010.

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

[25] 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, pp. 631–644, 1992.

[26] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*. NBS, 1952, vol. 49, no. 1.

[27] 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, pp. 856–869, 1986.

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

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

[30] 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, pp. 178 – 194, 2009, revolutionary Technologies for Acceleration of Emerging Petascale Applications.

[31] 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. New York: ACM, 2013.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] K. Tian, Y. Jiang, E. Zhang, and X. Shen, "An input-centric paradigm for program dynamic optimizations," in *the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2010.

[39] 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.