A Prediction Framework for Fast Sparse Triangular Solves

Najeeb Ahmad[1][0000−0002−3460−1256], Buse Yilmaz[1][0000−0001−5529−7188], and Didem Unat[1][0000−0002−2351−0770]

Department of Computer Science and Engineering, Koç University, Rumelifeneri Yolu, 34450 Sariyer, Istanbul, Turkey
{nahmad16,byilmaz,dunat}@ku.edu.tr

Abstract. Sparse triangular solve (SpTRSV) is an important linear algebra kernel, finding extensive uses in numerical and scientific computing. The parallel implementation of SpTRSV is a challenging task due to the sequential nature of the steps involved. This makes it, in many cases, one of the most time-consuming operations in an application. Many approaches for efficient SpTRSV on CPU and GPU systems have been proposed in the literature. However, no single implementation or platform (CPU or GPU) gives the fastest solution for all input sparse matrices. In this work, we propose a machine learning-based framework to predict the SpTRSV implementation giving the fastest execution time for a given sparse matrix based on its structural features. The framework is tested with six SpTRSV implementations on a state-of-the-art CPU-GPU machine (Intel Xeon Gold CPU, NVIDIA V100 GPU). Experimental results, with 998 matrices taken from the SuiteSparse Matrix Collection, show the classifier prediction accuracy of 87% for the fastest SpTRSV algorithm for a given input matrix. Predicted SpTRSV implementations achieve average speedups (harmonic mean) in the range of 1.4-2.7x against the six SpTRSV implementations used in the evaluation.

Keywords: Performance prediction · Sparse triangular solve · heterogeneous systems · Performance autotuning

1 Introduction

The sparse triangular solve (SpTRSV) is one of the important kernels used in direct and iterative methods for sparse linear systems and least square problems [24]. Efficient implementation of SpTRSV on CPU and GPU has been extensively studied and many SpTRSV implementations are available [15, 19, 20, 22, 24, 28, 30, 36]. However, there is no single execution platform or algorithm that gives the best SpTRSV performance for all input matrices. This is because, given a sparse matrix, the SpTRSV performance depends upon characteristics of the available parallelism in the matrix and implementation details of the algorithm (e.g. data structures, the sequence of operations etc.) [37]. Therefore, CPU has been observed to give better SpTRSV performance for some matrices

than GPU and vice versa [20, 28]. Also, different SpTRSV implementations on the same platform have been observed to achieve higher performance than others for different matrices [19, 28]. By selecting appropriate SpTRSV implementation for a given matrix, one can achieve higher SpTRSV performance. This can result in considerable performance gains for applications requiring multiple SpTRSV iterations, e.g., iterative solvers [33]. Given that many SpTRSV implementations are available for each platform, this selection can be a complex task. An obvious approach to select the fastest SpTRSV implementation is to run different implementations one-by-one and collect the empirical results. This, however, is a time-consuming and non-trivial process as each SpTRSV implementation has its own data structure, API, and matrix analysis requirements [24, 28].

In this paper, we propose a machine learning-based framework for predicting the fastest SpTRSV algorithm for a given matrix on CPU-GPU heterogeneous systems. The framework works by extracting matrix features, collecting algorithm performance data, and training a prediction model with 998 real matrices from the SuiteSparse Matrix Collection [9]. Once trained on a given machine, the model can predict the fastest SpTRSV implementation for a given matrix by paying a one-time matrix feature extraction cost. The framework is also capable of taking into account CPU-GPU communication overheads, which might be incurred in an iterative solver. We test our prediction framework for two CPU and four GPU algorithms on a modern Intel Xeon Gold CPU and NVIDIA Tesla V100 GPU systems. The model achieves an average prediction accuracy of 87% on our selected platform. Experimental results show predicted implementation achieving an average speedup (harmonic mean) in the range 1.4x-2.7x over a lazy choice of one of the six SpTRSV implementations used in this study.

The contributions of this work are summarized below:

- We provide comparative performance results of six SpTRSV implementations on a CPU-GPU platform.
- We identify an important set of features of a sparse matrix and develop a tool for efficiently extracting these features.
- We devise a framework to automatically extract matrix features, collect SpTRSV performance data, train machine learning model, and predict the fastest SpTRSV algorithm.
- We evaluate the performance, accuracy, and overhead of the framework on a modern CPU-GPU heterogeneous system.

2 Background and Motivation

The triangular solve refers to the solution of a linear system of the form $Ly = b$ or $Ux = y$, where L and U are lower and upper triangular matrices and x, y, and b are dense vectors. Due to the presence of dependencies among unknowns, triangular solve is an inherently sequential operation not easily lending itself to efficient parallelization [22]. When L and U are sparse, some of the dependencies may be missing thus offering an opportunity to calculate some unknowns in

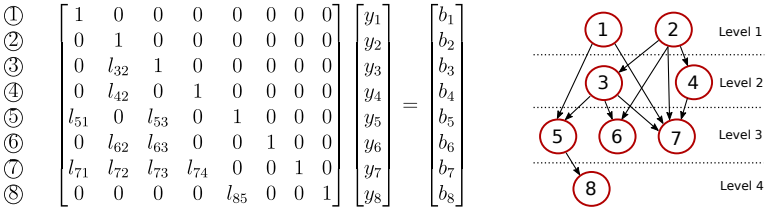

Fig. 1. A lower triangular system $Ly = b$ (left) and its dependency graph (right) [30]

Table 1. SpTRSV winning algorithm breakdown for 37 matrices in Figures 2

Arch.	SpTRSV implementation	Winner for # of matrices	Percentage
CPU	MKL(seq)	11	29.73%
	MKL(par)	2	5.41%
GPU	cuSPARSE(v1)	7	18.92%
	cuSPARSE(v2)(level-sch.)	7	18.92%
	cuSPARSE(v2)(no level-sch.)	2	5.41%
	Sync-Free	8	21.62%

parallel. Figure 1 shows a lower triangular system and its dependency graph. The nodes of the graph represent row numbers thus unknowns and edges represent dependencies of unknowns. The horizontal dashed lines separate the set of nodes into levels where nodes in each level can potentially be calculated in parallel [33]. The levels are numbered sequentially in the order in which computations on them can begin. As the number of levels, the number of unknowns in a level, and dependencies among unknowns is a function of matrix sparsity pattern, it is hard to devise an efficient SpTRSV algorithm for all possible input matrices.

The parallel SpTRSV algorithms proposed in the literature can be broadly categorized into (i) Level-scheduling [2] (ii) Synchronization-free [14, 18, 22, 24] (iii) Graph coloring [19, 29] (iv) Partitioned inverse [1], and (v) Iterative algorithms [4]. Most of these algorithms are comprised of two phases, an *analysis phase* in which parallelism in the matrix is discovered, and a *solve phase* in which the solver solves the linear system in parallel [19]. In level-scheduling algorithms, the analysis phase constitutes discovering the levels and unknowns within each level. In the solve phase, the algorithm proceeds level-by-level, synchronizing before starting computations on a new level. In synchronization-free methods, the number of dependencies per unknown [24] and in certain variations, the levels and unknowns within each level [19] are calculated in the analysis phase. Unlike level-scheduling approach, computations on an unknown can start as soon as its dependencies are met. The rest of the methods provide an approximate solution of the triangular system and are not the focus of this study.

For CPUs and GPUs, many implementations for the exact solution of SpTRSV are available. For CPUs, Intel MKL library [15] provides parallel *(MKL(par))* and sequential *(MKL(seq))* SpTRSV implementations. An implementation based on dependency graph sparsification has been developed by Park et al. [30] for

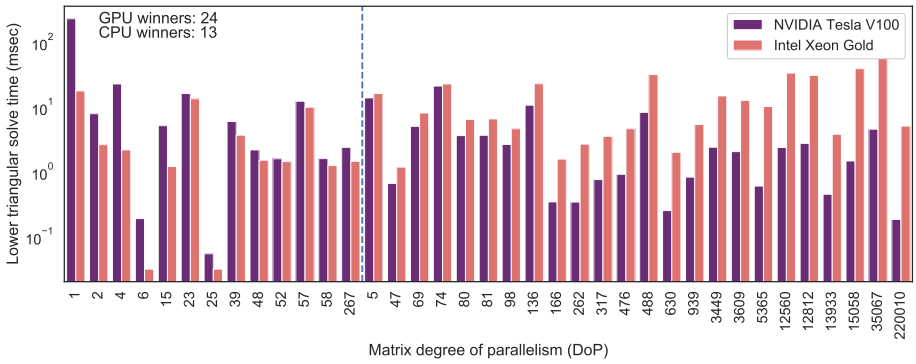

Fig. 2. SpTRSV performance on Intel Xeon Gold (6148) CPU and an NVIDIA V100 GPU (32GB, PCIe).

multicore CPUs. For NVIDIA GPUs, the cuSPARSE library provides SpTRSV based on the level-scheduling approach with their legacy API (*cuSPARSE(v1)*) [28]. The newer cuSPARSE SpTRSV (*cuSPARSE(v2)*) works with or without level information, depending upon the user's choice [7]. Weifeng et al. [22–24] developed a synchronization-free SpTRSV algorithm (*Sync-Free*) with multiple right-hand sides for GPUs. In [19,21], the author proposes variations of SpTRSV based on synchronization-free algorithms for GPUs.

Table 1 shows the breakdown of winning algorithms among six SpTRSV implementations (2 CPU, 4 GPU implementations) for a set of 37 matrices from the SuiteSparse matrix collection. Figure 2 shows the comparison for CPU and GPU winners for each of these matrices. The dashed vertical line separates the matrices into two groups; matrices attaining the best performance on CPU are on the left and on GPU, on the right. The x-axis shows the matrix degree of parallelism (DoP), which equals the average number of rows per level. Results show that no single algorithm or platform performs best for all matrices.

To find the best SpTRSV implementation for a given matrix, one option is to test each algorithm individually and select the best performing one. This requires the programmer to learn new APIs, manage data structures, and perform data format conversions for each implementation, which is tedious and error-prone. Moreover, some algorithms require non-trivial analysis time and necessitate multiple iterations to get stable performance numbers. To aid the programmer, this work proposes a framework that hides all the mentioned complexities and reports the predicted fastest algorithm by analyzing the matrix features. This can substantially improve the programmer's productivity and solver performance.

3 Related work

The general problem of algorithm selection [32] has been previously studied using statistical [13, 35], empirical [38] and machine learning techniques. OSKI [36] is

an autotuning library based on statistical techniques for sparse linear algebra kernels on CPU platforms, particularly the sparse matrix-vector multiplication (SpMV) and SpTRSV. The library can transparently tune kernels at runtime using the machine and input matrix characteristics. PetaBricks [3] is a language and compiler that allows multiple implementations of multiple algorithms for the same problem and automatically selects the best algorithm by building and using the so-called choice dependency graph. Sequoia [12] autotunes an application based on the memory hierarchy of the underlying machine. While PetaBricks and Sequoia do not take input data characteristics except the data set size, Nitro [27] framework allows programmers to guide the algorithm selection process by letting them specify characteristics of the input data they want to be considered for algorithm selection. These frameworks require programmers to learn new APIs and procedures to guide the algorithm selection process. A previous work dealing with SpTRSV execution choice between CPU and GPU is presented in [16] for the MAPS reservoir simulation system. The approach is, however, specific to reservoir simulation systems. A recent work by Dufrechou et al. [11] uses supervised machine learning to automatically select a sparse triangular solver on the GPUs. They tested their model for selection among cuSPARSE library-based SpTRSV and three variants of a CSR-based self-scheduling algorithm [10]. Their model managed to achieve an accuracy of close to 81%.

A number of works exist dealing with the selection of solvers and preconditioner-solver pairs for numerical software. Lighthouse [26] allows users to select the right solver and generate corresponding code for PETSc applications. It uses machine learning to analyze the matrix features and select the solver accordingly. Motter et al. [25] utilize machine learning techniques for selecting solver-preconditioner pairs for the PETSc framework [5] taking into account machine characteristics.

Unlike many of the previous works [11, 25, 26, 36], our prediction framework targets heterogeneous CPU-GPU systems. Compared with other frameworks targeting heterogeneous systems [3, 27], it does not require programmer guidance or target a specific application area [16]. In comparison to the similar work on the GPUs [11], our framework achieved higher accuracy (87% versus 81%) using a larger set of features for a wider (6 versus 4) and diverse set of SpTRSV algorithms. In addition, our framework is extensible allowing the inclusion of new SpTRSV algorithms as they become available [21].

4 Design and Implementation

The prediction framework is designed to automate the SpTRSV algorithm selection process for a given machine and matrix. It is composed of five main components (Figure 3); (1) An automatically downloadable set of matrices from the SuiteSparse Matrix Collection, (2) A matrix feature extractor, (3) An SpTRSV algorithm repository, (4) An SpTRSV performance data collector, which works by automatically downloading matrices, running each SpTRSV algorithm in the repository for each matrix, and logging the SpTRSV execution time. (5) A trainer and tester for the machine learning algorithm prediction model, which

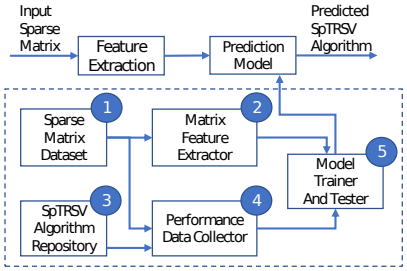

Fig. 3. The prediction framework

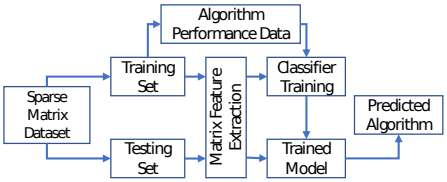

Fig. 4. Prediction model design flow

uses matrix features from the feature extractor as the input data and ID of the winning algorithm from performance data collector as the target for the model training and testing. Once the model is trained and tested, it can predict the fastest SpTRSV algorithm for a given matrix based on its features.

Figure 4 shows the design flow for our prediction model. For training the model, feature and algorithm performance data for the matrix data set is split into training and testing sets. Once trained with the training set, the model is tested with the matrices in the test set. Next, we discuss the important parts of the framework: (1) feature set selection, (2) feature extraction, and (3) machine learning model for prediction.

4.1 Feature Set Selection

SpTRSV performance is mainly affected by the sparsity pattern (i.e. the distribution of nonzero (nnz) elements in the matrix) [37]. The sparsity pattern is described by matrix structural data such as the number of rows, columns, nnzs, row and column lengths etc. We initially started with a set of around 50 structural features. After feature correlation analysis and feature score comparison, 30 structural features are finalized. We choose not to reduce the number of features further because reducing the number of features from 30 to, say, 10 negligibly improves the overhead of the feature extraction process but results in up to 10% drop in prediction accuracy. This is because many of the top-scoring features require per-level information, which in turn requires level calculation, which is generally the most time-consuming part of the matrix analysis phase [28]. The majority of the other features can be cheaply collected as a part of the level calculation process. Table 2 lists the final feature set used by the framework. The last column in the table also shows the score rank for each feature, where the lower score rank means a higher impact on performance.

4.2 Feature Extraction

Feature extraction is an overhead for the SpTRSV algorithm prediction and its execution time should be kept minimum. To achieve this, we employ both CPU and GPU in our feature extraction tool. This process completes in three

Table 2. Selected feature set for the prediction framework

No.	Features	Description	Score rank
1	*nnzs*	Number of nonzeros	1
2-4	<*max, mean, std*>_*nnz_pl_rw*	<maximum, mean, std dev> nonzeros per level row-wise	2, 4, 5
5	*max_nnz_pl_cw*	maximum nonzeros per level column-wise	3
6	*m*	Number of rows/columns	6
7-10	<*max, mean, median, std*>_*rpl*	<maximum, mean, median, std dev> rows per level	7, 12, 13, 16
11-12	<*min, max*>_*cl_cnt*	<minimum, maximum> column length count	8, 10
13-14	<*max, min*>_*rl_cnt*	<maximum, minimum> row length count	9, 11
15-17	<*max,std,median*>_cl	<maximum, std dev, median> column length	14, 22, 29
18	*lvls*	Number of levels	15
19-21	*mean_*<*max, mean, std*>_*cl_pl*	mean <maximum, mean, std dev> columns per level	17, 18, 20
22-25	<*max,mean,median,std*>_rl	<maximum, mean, median, std dev> row length	19, 27, 28, 30
26-30	*mean_*<*max,std,mean ,median,min*>_*rl_pl*	mean <maximum, std dev, mean, median, minimum> row length per level	21, 23, 24, 25, 26

steps. In the first step, row dependencies (row lengths) for lower/upper triangular matrices are calculated on GPU. Then, we use a slightly modified CUDA implementation of Kahn's algorithm [8] presented in [19] to construct levels in a triangular matrix. The algorithm calculates levels and rows in a level by performing topological sorting on the dependency graph. It recursively finds rows with zero dependencies, saves the row IDs of the current level into a queue, and then removes these rows and their outgoing edges from the graph until no more rows to process. In addition to level calculation, we also collect some statistics such as the number of rows per level, row and column lengths per level, and the nnzs per level. Finally, the remaining features listed in Table 2 are calculated using the NVIDIA Thrust library [6]. For this purpose, while CPU iterates over levels, GPU is used to calculate features for that level.

4.3 Machine Learning Model and Training

For training the model, we use the Scikit-learn machine learning library in Python [31]. As the matrix data set, we choose 998 real square matrices with 1000 or more rows (up to 16M rows) from the SuiteSparse Matrix Collection. We train the model with two CPU SpTRSV algorithms, namely *MKL(seq)* and *MKL(par)*, and four GPU algorithms, namely *cuSPARSE(v1)*, *cuSPARSE(v2) with* and *without level-scheduling* and synchronization-free algorithm (*Sync-Free*) [24].

We assign a unique integer ID to each of these algorithms and collect features and SpTRSV performance data for each matrix in our data set in an automated fashion using the libufget library [17] and our feature extraction tool. The matrix

features and the ID of the fastest SpTRSV implementation then serve as input and target, respectively, for training the machine learning model.

For selecting appropriate classifier for the prediction model, we evaluated a number of supervised machine learning-based classifiers provided by the Scikit-learn library including Decision Trees, Random Forest, Support Vector Machines (with grid-search), K-Nearest Neighbors, and Multi-Layer Perceptron Classifier using the Scikit-learn `model_selection` class. Based on the cross-validation scores, we choose Random Forest classifier for prediction. The feature scores are calculated using `feature_selection` class (`SelectKBest` function) with chi-squared used as the score function. Although Deep Neural Networks are suitable for classification tasks and feature selection is done by the model itself, they take considerably amount of time to train and a large training set is required. Hence, we preferred classical supervised machine learning techniques mentioned above and obtained good prediction accuracy.

To evaluate the performance of the prediction model, we utilize cross-validation functionality provided by the Scikit-learn `model_selection` class. For this purpose, features in the input data set are first scaled using Standard Scaler and the data set is then split into test and training data with `train_test_split` function that randomly splits the data set into 75% training data and 25% test data by default. We keep the default split ratios for our evaluation. Next, we use k-fold cross-validation with k set to 10. In k-fold cross-validation, training data set is divided into k smaller sets. For each of k sets, *k-1* sets are used as training data while the remaining set is used for validating the model. The performance of the k-fold cross-validation is then the average of these results.

4.4 Effects of CPU-GPU Data Transfers

In a CPU-GPU system, executing the fastest SpTRSV algorithm may require data transfers between CPU and GPU. For instance, GMRES solver with preconditioning performs sparse-matrix multiplication and vector products in addition to SpTRSV in each iteration [34]. With data transfer overheads, the fastest SpTRSV algorithm may no longer be the fastest as another implementation may require no data transfer.

To elaborate on this, consider a lower triangular system $Ly = b$ to be solved with SpTRSV (see Section 2). For iterative methods, matrix L is generally fixed while b and y are updated every iteration. Consider the scenarios shown in Figure 5, where computations just before and after SpTRSV, execute on different platforms. In Figure 5, H->D and D->H represent host-to-device and device-to-host data transfers, respectively. As shown in the figure, the data transfer for either the right-hand side or solution vector is inevitable. Therefore, it is always beneficial to choose the fastest SpTRSV algorithm irrespective of whether it runs on the CPU or on the GPU. For the scenarios where computations, just before and after SpTRSV, execute on the same platform and SpTRSV executes on a different platform (Figure 6), two data transfers are incurred; (the right-hand side and the solution vector). Consequently, this data transfer overhead may change the algorithmic choice. To cater for such scenarios, our framework allows

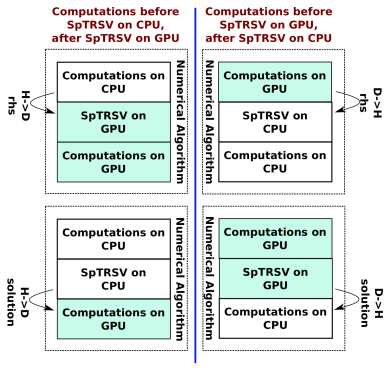

Fig. 5. CPU-GPU data exchange when computations just before and after SpTRSV execute on different platforms

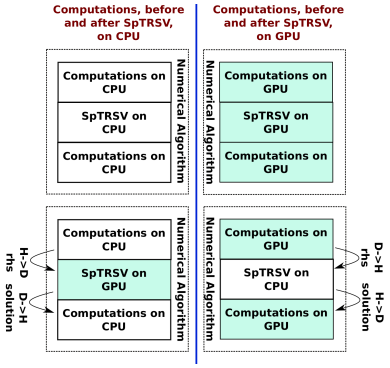

Fig. 6. CPU-GPU data exchange when computations just before and after SpTRSV execute on the same platforms

Table 3. SpTRSV winning algorithm breakdown for the 998 matrices from SuiteSparse

Arch.	SpTRSV Implementation	Winner for # of matrices	Percentage
CPU	MKL(seq)	411	41.18%
	MKL(par)	11	1.10%
GPU	cuSPARSE(v1)	111	11.12%
	cuSPARSE(v2)(level-sch.)	61	6.12%
	cuSPARSE(v2)(no level-sch.)	15	1.50%
	Sync-Free	389	38.98%

users to specify whether the rest of the numerical solvers executes on a CPU (CPU-centric) or a GPU (GPU-centric). For the CPU-centric scenario, the data transfer time (for the right-hand side and solution vector) is added to each of the GPU algorithms during the training phase. Similarly, for the GPU-centric scenario, the data transfer time is added to each of the CPU algorithms before training the model. Thus, the prediction framework can identify the fastest SpTRSV in presence of data communication overheads.

5 Evaluation

This section evaluates the performance of different SpTRSV algorithms, our framework's prediction accuracy, its performance, and its overhead compared to the analysis phase of SpTRSV algorithms. The performance results were collected on a CPU-GPU machine with an Intel Xeon Gold (6148) CPU and NVIDIA Tesla V100 GPU. CPU has 2 sockets with 20 cores in each and comes

Table 4. Number of rows and nonzero statistics for the 998 matrices from SuiteSparse

	Minimum	Median	Maximum
Number of rows	1K	12.5K	16.24M
Number of nonzeros	1.074K	105.927K	232.232M

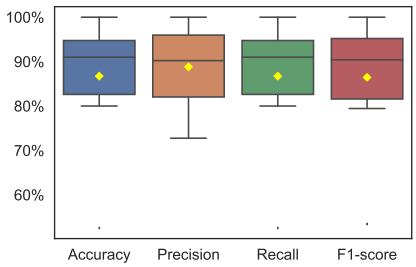

Fig. 7. Model cross validation scores with 30 features in the feature set

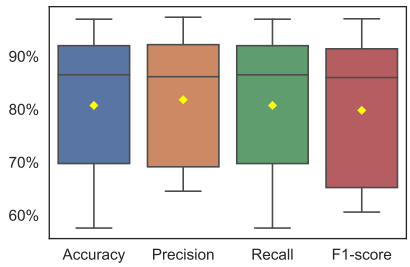

Fig. 8. Model cross validation scores with 10 features in the feature set

with a 512GB of memory. GPU has 32GB of memory. The Intel MKL implementations are compiled with `icpc` compiler from Intel Parallel Studio 2019 with `-O3` optimization. For *MKL(par)*, all available CPU cores are used without hyper-threading. The cuSPARSE and Sync-Free implementations are compiled using `nvcc` compiler from CUDA version 10.1 with options `-gencode arch=compute_70,code=sm_70`. Statistics for the number of rows and nonzeros for the matrix data set are given in Table 4.

5.1 Performance of SpTRSV Algorithms

This section presents the experimental results for the six SpTRSV algorithms. For this purpose, each of the six SpTRSV implementations is run 100 times for each matrix in the data set and mean execution time is reported. The results presented here are for the solution of the lower triangular system. Table 3 shows the breakdown of the winning implementations for the entire matrix data set. As regards the number of times an SpTRSV implementation was the fastest for the data set, we observe that, in general, there is no clear GPU advantage over CPU. Intel *MKL(seq)* is the fastest for a high percentage of the matrices than any other implementation. This is possibly due to the fact that some matrices exhibit very low parallelism that can be exploited or variable degrees of parallelism. In general, Intel *MKL(par)* shows poor performance. The *cuSPARSE(v1)* surprisingly performs better than two variants of *cuSPARSE(v2)* combined. Moreover, on GPU, the *Sync-Free* implementation is dominant over cuSPARSE implementations.

5.2 Accuracy of the Framework

The performance of the machine learning model is measured using typical metrics of accuracy, precision, recall, and f1-score. Figure 7 shows the 10-fold cross-validation results for the Random Forest classifier with 300 forests and feature set with 30 features presented in Table 2. The yellow diamond shows the mean value for each parameter. The classifier achieves an average weighted score of 87%

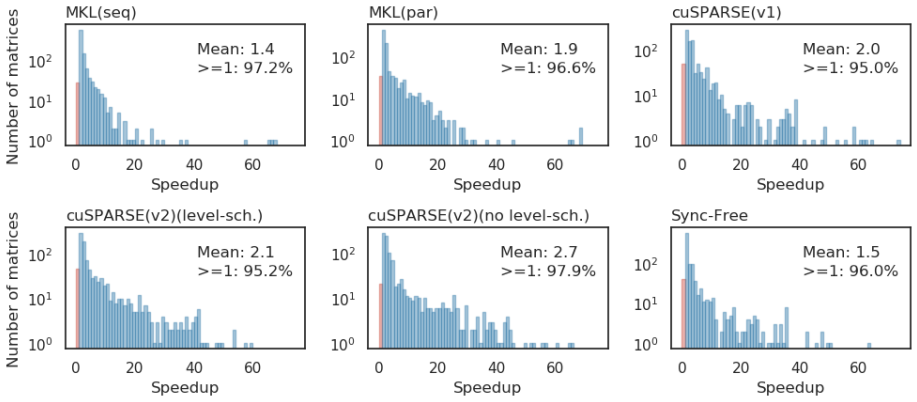

Fig. 9. Speedup gained by predicted over lazy choice algorithm. >= 1 indicates speedup of greater or equal to 1. (Harmonic) mean refers to average speedup achieved by the framework over the lazy choice. Each bin covers a speedup range e.g. first bin covers speedups between 0-0.99, second one covers speedups between 1-1.99 and so on.

for accuracy, recall, f1-score, and 89% for precision. It means that our SpTRSV framework correctly predicts the best algorithm for 87% of the data set.

To evaluate the effect of reducing the number of features on the prediction model performance, we keep the top 10 features in the feature data set based on their feature scores (score rank in Table 2) and perform 10-fold cross-validation of the resultant model. As shown in Figure 8, there is a 7-10% drop in performance metrics with the reduced set of features. In addition, there is a wider spread of performance. Considering the possibility of inclusion of new algorithms into the framework in the future, we keep the 30 features listed in Table 2.

Possible reasons for incorrect predictions by the framework include (1) limited diversity in matrix data set (2) comparable algorithm performance for a matrix so that incorrect prediction does not really matter (3) limited feature set. We will further investigate these reasons in the future.

5.3 Speedup Gained by the Framework

To evaluate the performance benefits of our framework, we compare the speedup over the *lazy* choice made by the user for an SpTRSV implementation. Unlike an *aggressive* programmer, who may test all the algorithms to find the best performing algorithm, the lazy programmer always uses the same SpTRSV algorithm regardless of the input matrix. The speedup is defined as $s = T_l/T_p$, where T_l is the execution time of the algorithm that the programmer lazily uses, and T_p is the predicted algorithm by the framework, which may or may not be the fastest algorithm. The speedup is calculated based on the SpTRSV running times and does not include the analysis phase for the algorithms for T_l or T_p.

Figure 9 shows the histogram for the speedups achieved by the prediction framework over each of the six implementations for the entire data set. The figure

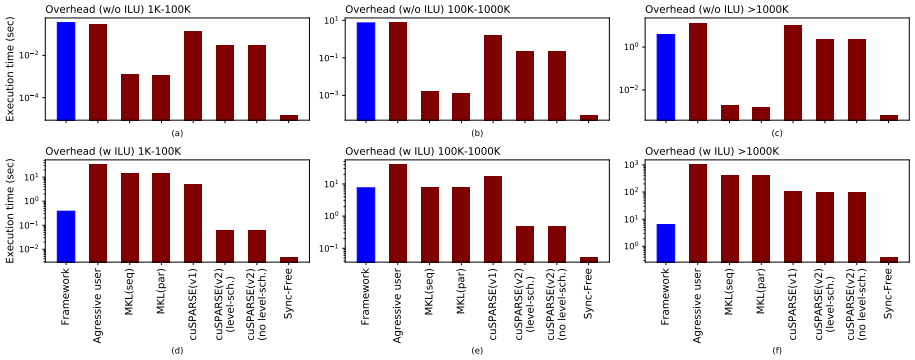

Fig. 10. Mean overhead of framework versus mean empirical execution time for aggressive and lazy users. 1K-100K, 100K-1000K and >1000K refer to matrix size ranges.

also shows the percentage when the predicted algorithm achieves equal or better performance than the lazy choice. The results show that the predicted fastest SpTRSV algorithm achieves the same or better performance for greater or equal to 95% of the matrices. Note that for the aggressive programmer, our prediction is 87%, which is presented in Section 5.2. We also observe that the speedup obtained by the prediction framework can reach tens or even hundreds for some SpTRSV implementations. Thus, using the framework is highly attractive than an arbitrary algorithmic choice for the SpTRSV execution.

To evaluate performance loss incurred by incorrect predictions, we compared the actual fastest SpTRSV against the incorrectly predicted implementation by our model. The results show that for roughly $3/4^{th}$ of the incorrect predictions, the predicted implementation is less than 2 times slower. Considering the prediction accuracy, speedups achieved with correct predictions, and programming benefits of the framework, we believe this performance loss is reasonable.

5.4 Framework Prediction Overhead

In this section, we evaluate the overhead associated with our algorithm prediction framework. This overhead includes the time spent in feature extraction and for the model to predict the fastest implementation. The feature extraction time depends on matrix sparsity pattern and its size while prediction time is constant for all matrices. Feature extraction includes computing dependencies in triangular matrices, calculating levels, collecting matrix statistics (e.g. row per level etc.), and calculating the final feature set from these statistics. This phase is very similar to the analysis phase of the SpTRSV algorithms based on the level-set method such as *cuSPARSE(v1)* and *(v2)* with levels.

We compare the framework overhead with empirical execution overhead. For the empirical overhead, there are two different types of users: a *lazy* user, who conservatively uses the same algorithm and an *aggressive* user who tests all six algorithms and chooses the best performing SpTRSV implementation. The

empirical overhead for an aggressive user for N algorithms is calculated using the equation:

$$\textit{Empirical Overhead} = \sum_{i=1}^{N} (A_i + 10 * (TS)_i) \tag{1}$$

where A_i and $(TS)_i$ are the matrix analysis phase and single SpTRSV iteration times for algorithm i, respectively. The factor 10 in Equation 1 refers to the approximate number of SpTRSV executions required to get a stable time estimate of a single SpTRSV iteration. For the lazy user, there is only a matrix analysis phase as the lazy user does not question the suitability of the algorithm.

For overhead analysis, we divide the matrices into three groups based on their sizes (1K-100K, 100K-1000K, >1000K). For each group, we compare the mean time spent by the framework, by the aggressive user to select the fastest algorithm, and by the lazy user to run the analysis phase of their chosen algorithm. We assume, without loss of generality, that each SpTRSV implementation runs its own ILU factorization phase except *cuSPARSE(v2)(no level-sch.)* that can use ILU factorization from *cuSPARSE(v2)(level-sch.)*. In some cases, it might be possible for some implementations to use ILU factorization from another implementation. However, it will generally require extra effort from the programmer and might add its own processing overhead (e.g. converting ILU factors from one data structure to another). For the sake of fairness, we provide an overhead comparison with ILU factorization time included (*w ILU*) and excluded (*w/o ILU*) ILU from A_i as well as framework overhead. For *Sync-Free* implementation, extraction of upper and lower triangular parts of the input matrix as ILU factorization time as it does not perform actual ILU factorization [23].

Figure 10(a), (b), (c) compare overhead for the three groups of matrices with ILU factorization time excluded. For matrix sizes less than 1000K, the average framework overhead is comparable with the time spent by the aggressive user. For matrix sizes >1000K, the framework overhead is on average 4 times less than the overhead of the aggressive user. Figure 10(d), (e), (f) compare overhead for the three groups of matrices with ILU factorization time included. For all matrix sizes, the average overhead of the framework is observed to be considerably less than aggressive user time by factors ranging between 5 (for 100K-1000K range) and 161 (for >1000K range). Overall, considerable time savings can be obtained by using our framework, especially for large matrices.

We also compute the number of SpTRSV iterations of the predicted algorithm required to amortize the cost of the framework overhead. For all matrix sizes, the mean number of iterations required to amortize the framework overhead is within the range of hundreds. For instance, for the largest group of matrices (>1000K), a mean number of 127 SpTRSV iterations of the predicted algorithm are required to compensate for the framework overhead. Considering that an iterative solver generally requires several hundreds of iterations for convergence, we claim that the overhead of the framework is acceptable. For aggressive users, we provide an option to aggressively test each implementation and bypass the prediction, thus saving time and effort of manual implementation of each algorithm.

6 Conclusions

SpTRSV is an important and often most time consuming computational kernel with no single SpTRSV implementation shown to give the best performance for all matrices. In this work, we propose a machine learning-based framework for predicting the fastest implementation for SpTRSV for a given input matrix on heterogeneous systems. We train the prediction model with 30 features for each of the 998 square, real matrices selected from SuitSparse collection, and six SpTRSV algorithms. The experimental results, on an Intel Gold CPU with NVIDIA V100 GPU, show our framework achieving an average prediction accuracy of 87% and an average speedup (harmonic mean) in the range 1.4-2.7x over the *lazy* programmer scenario whereby the programmer always chooses the same alogrithm. The framework is extensible with new algorithms as they become available.

Acknowledgements

Authors would like to thank Aramco Overseas Company and SaudiAramco for funding this research.

References

1. Alvarado, F.L., Schreiber, R.: Optimal parallel solution of sparse triangular systems. SIAM Journal on Scientific Computing **14**(2), 446–460 (1993). https://doi.org/10.1137/0914027
2. Anderson, E., Saad, Y.: Solving sparse triangular linear systems on parallel computers. International Journal of High Speed Computing **1**(01), 73–95 (1989). https://doi.org/10.1142/S0129053389000056
3. Ansel, J., et al.: Petabricks: A language and compiler for algorithmic choice. SIGPLAN Not. **44**(6), 38–49 (Jun 2009). https://doi.org/10.1145/1543135.1542481
4. Anzt, H., Chow, E., Dongarra, J.: Iterative sparse triangular solves for preconditioning. In: European Conference on Parallel Processing. pp. 650–661 (2015). https://doi.org/10.1007/978-3-662-48096-0_50
5. Balay, S., et al.: PETSc users manual. Tech. Rep. ANL-95/11 - Revision 3.11, Argonne National Laboratory (2019), https://www.mcs.anl.gov/petsc
6. Bell, N., Hoberock, J.: Thrust: A productivity-oriented library for cuda. In: GPU Computing Gems Jade Edition, pp. 359 – 371. Morgan Kaufmann, Boston (2012). https://doi.org/10.1016/B978-0-12-385963-1.00026-5
7. Coporation, N.: Cusparse library, user's guide (November 2019)
8. Cormen, T., Leiserson, C., Rivest, R.L., Stein, C.: Introduction To Algorithms. MIT Press (2001)
9. Davis, T.A., Hu, Y.: The university of florida sparse matrix collection. ACM Trans. Math. Softw. **38**(1), 1:1–1:25 (Dec 2011). https://doi.org/10.1145/2049662.2049663
10. Dufrechou, E., Ezzatti, P.: Solving sparse triangular linear systems in modern gpus: A synchronization-free algorithm. In: 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP). pp. 196–203 (2018). https://doi.org/doi: 10.1109/PDP2018.2018.00034

11. Dufrechou, E., Ezzatti, P., Quintana-Orti, E.S.: Automatic selection of sparse triangular linear system solvers on gpus through machine learning techniques. In: 2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). pp. 41–47 (2019). https://doi.org/10.1109/SBAC-PAD.2019.00020
12. Fatahalian, K., et al.: Sequoia: Programming the memory hierarchy. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. pp. 4–4 (Nov 2006). https://doi.org/10.1109/SC.2006.55
13. Guo, H.: A bayesian approach for automatic algorithm selection. In: Proceedings of the International Conference on Artificial Intelligence, Mexico. pp. 1–5 (2003)
14. Heath, M., Romine, C.: Parallel solution of triangular systems on distributed-memory multiprocessors. SIAM J. Sci. Statist. Comput. **9**(3), 558–588 (1988). https://doi.org/10.1137/0909037
15. Intel Incorporated: Intel MKL — Intel Software (2019), https://software.intel.com/en-us/mkl/documentation/view-all
16. Klie, H., et al.: Exploiting capabilities of many core platforms in reservoir simulation. In: SPE Reservoir Simulation Symposium 2011. pp. 264–275 (6 2011). https://doi.org/10.2118/141265-MS
17. Köhler, M.: libufget - the uf sparse collection c interface (Sep 2017), https://doi.org/10.5281/zenodo.897632
18. Li, G., Coleman, T.F.: A parallel triangular solver for a distributed-memory multiprocessor. SIAM J. Sci. Stat. Comput. **9**(3), 485–502 (May 1988). https://doi.org/10.1137/0909032
19. Li, R.: On parallel solution of sparse triangular linear systems in cuda. ArXiv **abs/1710.04985** (2017)
20. Li, R., Saad, Y.: Gpu-accelerated preconditioned iterative linear solvers. The Journal of Supercomputing **63**(2), 443–466 (Feb 2013). https://doi.org/10.1007/s11227-012-0825-3
21. Li, R., Zhang, C.: Efficient parallel implementations of sparse triangular solves for gpu architectures. In: Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing. pp. 106–117 (2020). https://doi.org/10.1137/1.9781611976137.10
22. Liu, W., et al.: A synchronization-free algorithm for parallel sparse triangular solves. In: Euro-Par 2016. pp. 617–630. Springer (2016). https://doi.org/10.1007/978-3-319-43659-3_45
23. Liu, W., et al.: Benchmark SpTRSM using CSC (September 2017), https://github.com/bhSPARSE/Benchmark_SpTRSM_using_CSC
24. Liu, W., et al.: Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides. Concurrency and Computation: Practice and Experience **29**(21) (2017). https://doi.org/10.1002/cpe.4244
25. Motter, P.: Hardware Awareness for the Selection of Optimal Iterative Linear Solvers. Ph.D. thesis, University of Colorado at Boulder (2017)
26. Motter, P., Sood, K., Jessup, E., Boyana, N.: Lighthouse: an automated solver selection tool. In: Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering. pp. 16–24 (2015). https://doi.org/10.1145/2830168.2830169
27. Muralidharan, S., Shantharam, M., Hall, M., Garland, M., Catanzaro, B.: Nitro: A framework for adaptive code variant tuning. In: IEEE 28th IPDPS 2014. pp. 501–512. IEEE (2014). https://doi.org/10.1109/IPDPS.2014.59
28. Naumov, M.: Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu. NVIDIA Tech. Rep. NVR-2011-001 (2011)

29. Naumov, M., Castonguay, P., Cohen, J.: Parallel graph coloring with applications to the incomplete-lu factorization on the gpu. NVIDIA Tech. Rep. NVR-2015-001 (2015)
30. Park, J., Smelyanskiy, M., Sundaram, N., Dubey, P.: Sparsifying synchronization for high-performance shared-memory sparse triangular solver. In: ISC 2014. pp. 124–140. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-319-07518-1_8
31. Pedregosa, F., et al.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
32. Rice, J.R.: The algorithm selection problem. In: Advance in Computer, Advances in Computers, vol. 15, pp. 65 – 118. Elsevier (1976). https://doi.org/10.1016/S0065-2458(08)60520-3
33. Saad, Y.: Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edn. (2003). https://doi.org/10.1137/1.9780898718003
34. Saad, Y., Schultz, M.H.: Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM J. Sci. Statist. Comput. **7**(3), 856–869 (1986). https://doi.org/10.1137/0907058
35. Vuduc, R., Demmel, J.W., Bilmes, J.A.: Statistical models for empirical search-based performance tuning. The Int. J. High Perform. Comput. Appl. **18**(1), 65–94 (2004). https://doi.org/10.1177/1094342004041293
36. Vuduc, R., Demmel, J.W., Yelick, K.A.: Oski: A library of automatically tuned sparse matrix kernels. Journal of Physics: Conference Series **16**, 521–530 (2005). https://doi.org/10.1088/1742-6596/16/1/071
37. Vuduc, R.W.: Automatic Performance Tuning of Sparse Matrix Kernels. Ph.D. thesis, University of California, Berkeley (2003)
38. Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: Proceedings of the IEEE Conference on Supercomputing. pp. 1–27. SC '98 (1998). https://doi.org/10.1109/SC.1998.10004