Machine learning for optimal selection of Sparse Triangular System Solvers on GPUs

Ernesto Dufrechou[a], Pablo Ezzatti[a], Manuel Freire[a], Enrique S. Quintana-Ortí[b]

[a] *Instituto de Computación (INCO), Facultad de Ingeniería, Universidad de la República, Montevideo, Uruguay.*

[b] *Depto. de Sistemas de Informática de Sistemas y Computadores , Universitat Politècnica de València. Spain.*

Abstract

Many numerical algorithms for science and engineering applications require the solution of sparse triangular linear systems (SPTRSV) as their most costly stage. For this reason, considerable research has been dedicated to produce efficient implementations for almost all high performance computing platforms. In the case of graphics processing units (GPUs), there are several strategies to perform this operation, which translate into a handful of different routines. In general, it is difficult to establish a priori which is the best routine for a given problem, and thus, an automatic procedure able to select the best solver for each matrix can entail large performance benefits.

This work extends a previous effort, in which we relied on machine learning techniques to predict the best SPTRSV routine for each matrix, by improving both the accuracy and the speed of the selection procedure. Specifically, we focus on the most efficient machine learning techniques regarding the speed of their training and prediction stages; evaluate the artificial generation of sparse matrices to expand our dataset; and propose heuristics to compute approximations of some expensive features. The experimental results show that we can strongly improve the runtime of our procedure without compromising the quality results.

Keywords: graphics processors; sparse triangular linear systems; high performance; machine learning.

*Corresponding author

Email address: edufrechou@fing.edu.uy (Ernesto Dufrechou)

1. Introduction

The numerical solution of linear systems of equations through direct or iterative methods often involves the decomposition of the original problem into triangular factors. Well-known examples of this are the LU factorization, and the Incomplete LU used as preconditioner in iterative methods [1, 2]. For this reason, the solution of sparse triangular linear systems (SPTRSV) is one of the most important building blocks in sparse numerical linear algebra.

Throughout the years, the SPTRSV kernel has been implemented for almost all relevant parallel platforms [3, 4, 5, 6, 7]. However, the data dependencies arising in this particular operation complicate the design of efficient parallel routines. Moreover, as many other sparse kernels, the operation presents a low computational intensity and irregular data access pattern, which make it difficult to attain high performance [8, 9]. GPUs are no exception, and since their adoption as commodity HPC hardware, there have been numerous efforts to produce high performance implementations of SPTRSV for these platforms. From the review of literature and existing software libraries, it can be concluded that the most successful of these efforts generally belong to two main classes, the *level-set* [10] and the *self-scheduled* [11] paradigms.

The performance results reported by different authors indicate that any of these two paradigms are better suited for some linear systems than for others. In [12] we took a first step in the direction of identifying matrix properties that allowed predicting which solver or paradigm offers the smallest execution time for a given problem. However, the analysis was performed on a small set of matrices, and the conclusions were limited to the most evident cases.

Later, in [13], we moved on to study the use of black-box machine learning models to automatically select among several SPTRSV routines, based on the characteristics of the sparse matrix. Specifically, we evaluated the application of the techniques bundled with the *Classification Learner App* of MATLAB® [14]. With Krylov subspace solvers in mind, we targeted the case where one analysis phase (if applicable) and 10 solution phases are performed for each triangular linear system. Furthermore, we explored the effect of 10 different matrix features on the selected classifiers, studying the quality of their predictions using different subsets. The experimental evaluation carried out previously reveals that some procedures are able to attain predic-

tions with values of 80% of accuracy. A similar approach has been recently followed by other authors [15].

This effort extends [13] in several lines with the purpose of improving the performance of the whole selection procedure. The main contributions of this work are:

1. The extension of the number of SPTRSV realizations addressed in our study, including two extra methods. Specifically, we added the solvers bundled with the latest versions of the CUSPARSE library: *cuspv2*$_{nl}$ is the novel solver based on a domino (or synchronization-free) strategy, and *cuspv2* that is a domino-like method enhanced with level set information.
2. A study of the fastest machine learning techniques. In other words, we select the machine learning strategies that consume less runtime for the training and prediction stages among those that present an acceptable level of accuracy.
3. The inclusion of an additional metric for the experimental evaluation of the machine learning techniques. In previous work, we used the accuracy (number of correct predictions over the total number of instances) and the ARE (average relative difference of runtime between the predicted and the best method). In this work we add the Total Relative Runtime Error (TRRE), which is the ratio between the aggregated runtimes of the predicted methods and the aggregated runtimes of the best method in each case. This new metric can be used to detect when a given model performs badly for large instances.
4. Artificial sparse matrix generation (data augmentation). As the sparsity pattern completely determines the performance of the different triangular solvers, we aimed to train our models using realistic datasets (from SuiteSparse collection). However, the number of available sparse matrices derived from real problems is too small for machine learning technologies. Thus, to count with a larger training set, we create new matrices by performing random perturbations of the original ones. With this approach, we generate larger training sets, while maintaining some characteristics of the sparsity patterns found on real problems.
5. Design of heuristics to reduce the runtime related with the computation of costly matrix features.
6. We apply and evaluate our procedure to select the best SPTRSV solver as a part of a method to address real-world problems. In particu-

lar, as a part of a Preconditioned Conjugate Gradient (PCG) iterative method, with an incomplete factorization with 0 fill-in (ILU0) as preconditioner.

The rest of the paper is structured as follows. In Section 2 we review the main aspects of the GPU implementations of the sparse triangular linear systems. Section 3 revisits our first steps with the use of machine learning, and its application in the context of our problem. Later, in Section 4, we detail our experimental evaluation and highlight the most important results. This is followed by the description of our extensions in Section 5, and the application of our automatic selection procedure in the context of preconditioned Krylov subspace solvers in Section 6. Finally, Section 7 offers several concluding remarks and summarizes a few lines of future work.

2. Parallel solution of sparse triangular linear systems

Consider the linear system

$$Lx = b, \tag{1}$$

where $L \in \mathbb{R}^{n \times n}$ is a lower-triangular matrix, $b \in \mathbb{R}^n$ is the right-hand side vector, and $x \in \mathbb{R}^n$ is the sought-after solution. Typically, the sparse matrix L is stored in some sparse storage format, among which CSR (Compressed Sparse Row) is the most common [2].

The importance of this computational kernel motivates the development of efficient routines for the many processor architectures. However, as usual in sparse linear algebra, reaching high levels of performance in the SPTRSV is a challenging task due to the combined effect of irregular data access patterns and low computational intensity [16]. Additionally, data dependencies and load imbalance due to the triangular structure of the problem [1] makes the parallelization of the operation quite challenging.

With the widespread adoption of massively parallel processors such as GPUs by the HPC community, the exploitation of parallelism for SPTRSV has motivated a considerable number of research efforts [6, 17, 18, 19, 20, 21]. The experimental analysis performed by different authors confirms that any of these proposals offer high performance for some linear systems but low for others.

The most successful routines, considering the numerical results presented in the literature and their integration into widely-used software libraries, belong to two main approaches, referred to as *level-set* [10] and *self-scheduled*

methods [11], respectively. A detailed description of both strategies, as well as the study of some of the most remarkable implementations of each paradigm for the CSR sparse matrix format, was presented in our previous work [21]. There, the *cuspv1* routine [6], which is the most representative implementation of the level-set strategy, is compared with four self-scheduled variants: *sf_ nan*, *sf_ order* and *sf_ mr* [21].

In this work we include two additional SPTRSV routines. Specifically, we include the solvers that are bundled with the latest versions of the CUSPARSE library: *cuspv2* is a solver based on a domino (or synchronization-free) strategy that relies on level set information, implemented in routines `csrsv2_analysis` and `csrsv2_solve` with policy *USE_ LEVEL*; $cuspv2_{nl}$ results from invoking the same routines with policy *NO_ LEVEL*.

3. Automatic selection

In this effort we continue relying on the *Classification Learner App* in MATLAB® in order to obtain the supervised machine learning models that will be applied to determine the best SPTRSV implementation. This MATLAB® module includes 5 types of techniques, namely, *Decision Tree*, *Discriminant Analysis*, *Support Vector Machines*, *K-Nearest Neighbor*, and *Ensemble* classifiers.

Along this line, we revisit our previous work and consider the same 10 features: **Arithmetic precision** ($prec$), **Matrix dimension** (n), **Number of nonzeros** (nnz), **Maximum number of non-zeros per row** (nnz_{max}), **Average number of non-zeros per row** (nnz_{avg}), **Standard deviation of non-zeros per row** (nnz_{stdev}), **Average bandwidth** (bw_{avg}), **Number of level-sets** ($levels$), **Locality** ($locality$), and **Multirow locality** ($locality_{mr}$).

As we stated previously, some of the features are more costly to compute than others. While n, nnz and nnz_{avg} can be obtained with cost $\mathcal{O}(1)$ from the sparse matrix data structure, computing nnz_{max}, nnz_{stdev}, bw_{avg} and $locality$ has a cost of $\mathcal{O}(n)$, where n is the dimension of the matrix. Furthermore, the computation of $levels$ and $locality_{mr}$ needs to be performed during or after the analysis phase of the *sf_ order* or *sf_ mr* variants, whose computational cost is similar to that of the solution phase of the linear system. Therefore, it is interesting to evaluate the performance of the models trained only with the features that are cheaper to extract.

After selecting these features, we trained the set of available machine learning models and performed a 5-fold cross validation to estimate the accuracy of the trained models on unseen data.

As our dataset is imbalanced (see Section 4.2), the accuracy of the models, which is the percentage of correctly classified instances, is not sufficient to determine the performance of the classifiers, because it can hide a selection delivering low performance for the minority classes. However, at this stage this metric provides some valid insights, as it reveals which type of machine learning models are best suited for our problem and deserve a more detailed analysis. Additionally, we also take into account the training and inference speed of each technique.

The result of this preliminary evaluation, employing the dataset described in Section 4.2, is shown in Table 1. In general, we note that the *Decision Tree* classifiers appear to be the best performing group of classifiers, especially if the training and prediction speeds of the models are considered. *The K-Nearest Neighbor* classifiers present a reasonably accuracy, with *Fine KNN* offering an accuracy of 74% but slow prediction speed. In comparison, *Weighted KNN* presents the same accuracy with a much faster prediction speed. Although in both cases, the training time is substantially higher than that required by Decision Trees, their magnitude is too small to make a difference in our application.

Some of the *Ensemble* classifiers included in this analysis also show good accuracy, though they present slower prediction speeds than *Decision Trees* and *Weighted KNN* classifiers.

On the other side, *the Discriminant Analysis* classifiers and *Support Vector Machines* perform badly in general.

To perform a more exhaustive evaluation, we next focus on the best performing models from the *Decision Tree*, *Weighted KNN*, considering a trade off between accuracy and prediction speed. Specifically, we set a lower bound of 70% for the accuracy and 10,000 observations per second for the speed, selecting the Fine Tree and Weighted KNN models. Before advancing to the experimental evaluation, we provide a small description of the *selected* models.

3.1. (Binary) Decision Tree

A binary decision tree is a tree structure where the inner nodes recursively split the dataset into two partitions according to the value of a certain

Table 1: Accuracy, prediction speed, and training time of the models included in the *Classifier Learner App* for our dataset, considering all the predictors.

	Model	Acc. (%)	Pred. speed (obs/sec.)	Train time (sec.)
Decision Trees	Fine tree	80.7	13000	5.41
	Medium tree	73.2	14000	5.81
	Coarse tree	69.6	47000	5.59
Discriminant	Lin. discr.	59.3	19000	6.12
SVM	Linear SVM	65.1	15000	9.76
	Quadratic	72.3	8200	73.82
	Cubic	59.9	21000	150.68
	Fine Gauss.	69.2	8300	75.24
	Medium Gauss.	67.6	9300	76.29
	Coarse Gauss.	58.4	9500	77.65
KNN	Fine	74.3	1600	78.11
	Medium	66.3	21000	78.54
	Coarse	60.5	13000	79.04
	Cosine	65.2	19000	78.91
	Cubic	65.7	3000	80.32
	Weighted	74.1	25000	80.75
Ensemble	Boosted Trees	76.6	3700	85.23
	Bagged Trees	84.6	3300	88.92
	Subspace discr.	59.8	3200	91.83
	Subspace KNN	79.0	2600	94.79
	RUSBoost. Trees	61.2	3600	98.14

attribute, and the terminal nodes (called leaves) represent the outcome classification of an observation that complies with the splitting attributes of the corresponding branch.

During the training of the model the nodes are split using some impurity measure computed over the subset of the training dataset that corresponds to the node. The subset is then partitioned accordingly, and the resulting partition is used as the training dataset for the respective children nodes.

In this work we train a decision tree based on the CART algorithm [22], which uses the Gini's diversity index [23] as impurity measure for the split criterion.

3.2. Weighted K-Nearest Neighbors

K-Nearest Neighbors is a simple yet powerful machine learning technique. Given some distance function, it maps an observation into the class corresponding to the majority of its k nearest neighbors, where k is an algorithm parameter.

A variation of this approach is to assign a heavier weight to nearby observations, so that they have more influence on the classification than those that are further away [24].

4. Experimental evaluation

In this section we revisited the initial experiments performed in Section 3 with the aim of evaluating the efficiency of the classification algorithms to select the best sparse triangular linear system solver.

4.1. Hardware platform

The runtime evaluation of the SPTRSV variants for the linear systems included in the dataset for the machine learning methods was performed on a server equipped with an Intel Xeon Gold 6138 CPU (20 cores at 2.00 GHz) and 128 GB of RAM, connected to a NVIDIA P100 GPU with 12 GB of RAM. The operating system is CentOS Linux 7 (Core) and the CUDA Toolkit for the GPU is version 9.2.

We employed MATLAB® 2018a for the classification experiments.

4.2. Initial dataset

In order to perform the experimental evaluation, we selected a set of square, real matrices, of medium and large dimension, from the SuiteSparse Matrix Collection[1] (formerly known as the University of Florida Matrix Collection –UFMC–). In particular, we considered the lower triangular part of a total of 981 sparse square matrices with dimension n between 1,916 and 55,042,369 and up to 113 million nonzeros.

As we stated previously, in addition to the triangular solvers described in [13] we add two more solvers that are bundled with the latest versions of the cuSPARSE library, *cuspv2* and $cuspv2_{nl}$.

For those solvers that require an initial analysis phase, we perform experiments replacing the runtimes of the solvers by the runtime taken by the execution of a single analysis phase plus 10 or 1000 iterations of the solver. For one-phase solvers, we only consider the time of 10 or 1000 iterations of the solver. This has the purpose of emulating the application of a Krylov subspace iterative solver in two different scenarios: one where the cost of the analysis may have a significant impact; and an alternative where that cost is completely overshadowed by the cost of 1000 executions of the solution phase.

We employ the IEEE floating point representations, in double precision for all the experiments.

4.3. Experimental Results

In Section 3 we selected the classification methods which offered better initial results to perform a complete analysis. In particular, we chose two methods, from the Decision Tree (*Tree*) and a K-Nearest Neighbor (*wKNN*) families, that presented a good compromise between accuracy and speed of training or prediction.

The *Tree* model was trained with a maximum depth of 100, and the Gini diversity index was applied as split criteria. The *wKNN* was trained using the Euclidean distance, the squared inverse of the distance as weights, and 10 nearest neighbors.

The cost of the computation of some matrix features, such as the number of level sets or the maximum number of nonzeros in one row, can be high in some cases, which can blur the gains obtained by the prediction. It is

[1] http://faculty.cse.tamu.edu/davis/suitesparse.html

therefore interesting to consider which features exert a stronger influence on the accuracy of the machine learning models. To study the impact of each feature, and assess how to diminish the number of predictors without affecting the accuracy of the models significantly, we start by removing the $locality, locality_{mr}$ and bw_{avg} predictors. Later, we also discard the number of $levels$. Next, we remove the nnz_{stdev} predictor. Finally, we eliminate nnz_{max} keeping only those predictors that can be computed with $\mathcal{O}(1)$ cost (n, nnz and nnz_{avg}). The ordering is based on the considerable computational cost of these predictors. In particular, $locality$, $locality_{mr}$ and $levels$ imply the computation of a costly level-set analysis of the sparse matrix.

Table 2 summarizes the effect of the exclusion of some features in the attained accuracy of the trained models. The accuracy of these models is computed using a 5-fold cross validation on the original dataset.

Besides the number of correct predictions over the total number of instances (accuracy), we are interested in analyzing the prediction from the point of view of our particular problem. In other words, in our application it may not be relevant to misclassify a solver when the difference in runtime between the predicted solver and the best routine is small.

For this reason we introduce two more metrics to assess the quality of the models. We define the Average Runtime Error (ARE) computed as:

$$ARE = \frac{1}{d} \sum_{i=1}^{d} \frac{T_{pred}(i) - T_{opt}(i)}{T_{opt}(i)},$$

and the Total Relative Runtime Error (TRRE) computed as:

$$TRRE = \frac{\sum_{i=1}^{d} T_{pred}(i)}{\sum_{i=1}^{d} T_{opt}(i)},$$

where T_{pred} is the runtime taken by the predicted method, T_{opt} is the runtime required by the best method, and d is the number of observations in the dataset. The best values for ARE and TRRE are 0 and 1, respectively.

While the ARE aims to reflect the average relative error in the prediction of each individual instance, it can hide cases where the models perform badly for large instances, which will have a high impact on TRRE.

The results summarized in Table 2 show that the effect of discarding these features is not uniform. In some cases, removing a certain feature deteriorates the results, but in other cases it improves the accuracy of the trained model.

Table 2: Accuracy and Average Runtime Error (ARE) after performing a 5-fold cross validation with three of the trained ML models. The effect of excluding features during training is also displayed. set_1 includes all features. set_2 excludes $locality$, $locality_{mr}$ and bw_{avg}. set_3 excludes the features excluded in set_2 and $levels$. set_4 excludes the features excluded in set_3 and nnz_{stdev}. set_5 excludes the features excluded in set_4 and nnz_{max}.

Iters	Model	Feat.	Acc.	ARE	TRRE
10	*Tree*	set_1	67%	0.25	1.34
		set_2	65%	0.30	1.42
		set_3	62%	0.45	1.67
		set_4	58%	0.48	1.63
		set_5	56%	0.57	1.60
	wKNN	set_1	72%	0.21	1.28
		set_2	66%	0.38	1.34
		set_3	63%	0.43	1.49
		set_4	59%	0.47	1.40
		set_5	56%	0.55	1.56
1000	*Tree*	set_1	69%	0.27	1.15
		set_2	67%	0.30	1.21
		set_3	64%	0.29	1.27
		set_4	65%	0.26	1.25
		set_5	61%	0.49	1.39
	wKNN	set_1	71%	0.16	1.07
		set_2	72%	0.12	1.07
		set_3	70%	0.16	1.23
		set_4	63%	0.24	1.18
		set_5	64%	0.30	1.23

However, the *levels* and $nnz_{max/std}$ features are those that have a stronger impact on the predictions. This is more evident in the TRRE metric. For the *Tree* model, the runtime overhead of using the predicted routines to solve all the systems in the dataset increases by 19% in the 10-iteration case, and 20% in the 1000-iteration case, whereas for the *wKNN*, the corresponding increases are 28% and 17% when these features are excluded. Moreover, the exclusion of the $nnz_{max/std}$ features seems to have significant impact on ARE, while the effect that can be observed on TRRE is smaller. This suggests that the row length is more relevant for the performance of the solvers for matrices of relatively small size where, even if the relative difference between the runtime of the predicted and best solver is large, the absolute difference in runtime is small.

Table 3: ARE and TRRE obtained by solving the entire dataset with each triangular solver.

Solver	10 iters		1000 iters	
	ARE	TRRE	ARE	TRRE
cuspv1	56.97	24.74	2.96	3.51
cuspv2	1.18	1.82	1.70	2.16
$cuspv2_{nl}$	0.78	1.85	2.02	2.36
baseline	1.75	2.97	4.16	3.81
sf_ nan	1.04	2.20	2.83	2.82
sf_ order	1.15	1.73	0.47	1.49
sf_ mr	0.98	1.51	0.22	1.22

The TRRE of the best cases, which are obtained with the *wKNN* model, indicate that using the predicted solver to process the entire dataset at least takes 28% more runtime than the optimal using the 10-iteration variant and 7% than using the 1000-iteration one. However, reported in Table 3, using the same solver for all cases results in considerably worse values of ARE and TRRE. In fact, using *sf_ mr*, which is the best routine in general for this dataset, results at least, in 51% and 22% larger runtime than optimal for the 10- and 1000-iteration cases respectively.

5. Improvement strategies

In this section, we extend our previous work in two different lines. Specifically, we address the reduction of the runtime needed by the features com-

putation. In more detail, we employ an heuristic to estimate the feature that implied $\mathcal{O}(n)$ computations. Later, we study the effect of the data set extension through the use of a procedure for sparse matrix generation.

5.1. Improving the speed of feature detection

As previously mentioned, the aim of this effort is to reduce the total runtime required by applications that rely in the solution of sparse triangular linear systems by the automatic selection of the best SPTRSV routine for each matrix. For this procedure to be effective, it is not enough that the machine learning models are accurate in their predictions. The prediction itself needs to add as little overhead as possible to the target application.

In general, as the training stage can be used for a large number of problems, its cost becomes negligible. The overhead of the procedure is then composed of two main aspects: the computational cost of the prediction, and the runtime needed for the computation of the matrix features that serve as input to the machine learning models.

The first of these two aspects was contemplated by the selection of machine learning techniques that present a fast inference stage. Regarding the second, in Section 4 we showed that our methods require some features that involve a computational cost of $\mathcal{O}(n)$. To explore the possibility of reducing the runtime related to the computation of the features, in this section we evaluate the impact of substituting the most expensive ones by approximate values, proposing a number of heuristics to compute those estimations.

5.1.1. Estimating the maximum and standard deviation of the number of nonzeros in a row

In order to obtain the maximum number of nonzero elements in a row for a sparse matrix stored in CSR format, it is necessary to traverse the vector of row pointers, subtracting each pair of consecutive entries to obtain the length of the corresponding row. This procedure requires $\mathcal{O}(n)$ time.

To compute an approximation of this value, with a reduced computational cost, we study two heuristics:

- The first one is to partition the vector of row pointers in k partitions of n/k rows. For each partition, the average number of nonzeros per row is computed, and the values obtained are reduced, while keeping the maximum, which is returned as the estimated maximum of nonzeros in a row. We evaluated the results obtained with three variants of

this heuristic: one with 1024 partitions (*max_p1024*), one with 128 partitions (*max_p128*), and one with the number of partitions equal to 10% of n (*max_p10%*).

- The second heuristic is a recursive binary search that, at each step, divides the rows into two groups and continues the recursion in the group containing a larger number of nonzero elements. The procedure stops when the partition has only one row, returning the number of non zero elements of that row. The algorithm also receives a parameter c that forces the procedure to explore the two branches of the recursion for the first c levels of the tree. We evaluated this heuristic for $c = 0, 2$ and 5, which are denoted as *max_b0*, *max_b2* and *max_b5*, respectively.

Although the first procedure implies $\mathcal{O}(1)$ and the second $\mathcal{O}(c \log n)$, if $k = 1024$ partitions are considered, the second procedure is faster for $c < 6$ for the tested matrices. With $c = 5$ the binary search algorithm obtains higher accuracy results, in general, while also outperforming the first procedure in execution time.

To estimate the standard deviation of nonzeros per row, we follow a procedure analogous to *max_p1024* and *max_p128*. We call these variants *std_p1024* and *std_p128*.

Table 4 shows the accuracy results of the models trained with the features of set_3 (see Table 2), but replacing nnz_{max} and nnz_{stdev} by the approximate values computed by the combinations of heuristics that presented higher accuracy for each model and dataset. The results in this table show that the use of approximate values does not offer significant differences with respect to the results obtained for set_3 in Table 2. Considering the balance between accuracy and computational cost, the combination of heuristics *max_p128/std_p128* stands out. In fact, this combination achieves high accuracy results for all the tested cases, at a fraction of the computational cost than most other combinations. For this reason we employ *max_p128/std_p128* for the rest of the experiments in this work.

5.2. Improving the prediction accuracy

In the previous experiments, the dataset used for training consisted on a subset of the sparse matrices from the SuiteSparse Matrix collection. In general, the sparsity pattern of this kind of matrices completely determines the performance of the different triangular solvers (the numerical values play

Table 4: Performance of the models after replacing the features nnz_{max} and nnz_{stdev} by the approximate values returned by different heuristics.

Iters	Model	Heuristics	Acc	ARE	TRRE
10	*Tree*	*max_p128/std_p1024*	61%	0.42	1.50
		max_p10%/std_p1024	60%	0.41	1.64
		max_b5/std_p1024	61%	0.43	1.57
		max_p128/std_p128	60%	0.45	1.66
		max_p10%/std_p128	60%	0.37	1.57
		max_b0/std_p128	60%	0.43	1.55
	wKNN	*max_p128/std_p1024*	64%	0.45	1.58
		max_p10%/std_p1024	64%	0.42	1.51
		max_b5/std_p1024	62%	0.45	1.57
		max_p128/std_p128	64%	0.46	1.59
		max_p10%/std_p128	64%	0.38	1.56
		max_b5/std_p128	63%	0.44	1.46
1000	*Tree*	*max_p10%/std_p1024*	65%	0.32	1.26
		max_b5/std_p1024	63%	0.39	1.28
		max_b2/std_p1024	63%	0.33	1.22
		max_p128/std_p128	63%	0.46	1.29
		max_p10%/std_p128	64%	0.31	1.29
		max_b2/std_p128	62%	0.31	1.25
	wKNN	*max_p128/std_p1024*	68%	0.18	1.20
		max_b5/std_p1024	67%	0.23	1.18
		max_b2/std_p1024	68%	0.28	1.23
		max_p128/std_p128	69%	0.27	1.22
		max_p10%/std_p128	68%	0.26	1.23
		max_b0/std_p128	68%	0.28	1.20

no role in this sense) and, therefore, the purpose was to train our models with a realistic dataset, assuming that some nonzero patterns are more likely to appear in real-world scenarios than others.

However, the dimension of the training dataset is a critical point for machine learning techniques. In this sense, although the SuiteSparse collection is a comprehensive and widely used benchmark for the traditional validation of sparse routines, the number of sparse matrices in that collection is rather small compared to the usual dimension of training sets for machine learning applications.

To count with a larger training set (data augmentation), the insufficiency of real-world instances led us towards the generation of random sparse triangular matrices. However, we suspect that a purely random training set needs to be immensely large for the trained machine learning models to be effective on realistic nonzero patterns, which is our ultimate goal. For this reason, instead of generating random datasets with millions of instances, in this work we create new matrices by performing random perturbations of the original ones. In this way, we intend to generate larger training sets but maintaining some characteristics of the sparsity patterns found on real problems.

We implemented two matrix generators. The first one takes an integer seed as input to initialize the random number sequence and then assigns each non-diagonal nonzero a new column index, which is a random number between the column index of the previous and next nonzero in the row. The column indexes of all the elements of each row thus remain in increasing order. In order to achieve datasets with different degrees of randomness, a *step* parameter controls the fraction of the total nonzeros to modify.

Using this generator, we can create datasets where the matrices have a dimension, total number of nonzeros, and number of nonzeros per row that can be found in, at least, one instance of the original dataset. Other properties, such as the number of level sets or the bandwidth, can drastically vary.

The second generator commences by performing a similar procedure on the row pointers vector of the CSR structure that holds the matrix, moving the integer pointer to the beginning of each row to a random position between the beginning of the previous and subsequent rows. Then it assigns a random column index to each non-diagonal nonzero of the modified rows, respecting the increasing order of the indexes and the triangular structure of the matrix.

Besides the matrix features randomized by the first generator, the second one varies features such as the maximum and standard deviation of the

number of nonzeros per row.

Using this two generators, we then created three random datasets:

1. *rand_1%* uses the first generator and modifies the column index of only 1% of the nonzero entries.
2. *rand_100%* uses the first generator and modifies all the nonzero entries.
3. *rand_rows* uses the second generator and modifies all the rows and nonzero entries.

Each dataset contains five randomized versions of each matrix in the original one, using different seeds to initialize the random sequence.

Table 5: Performance of the models using cross validation on different datasets for variant *max_p128/std_p128*.

Dataset	Iters.	Model	Acc.	ARE	TRRE
rand_100%	10	*Tree*	91%	0.05	1.03
	10	*wKNN*	96%	0.02	1.00
	1000	*Tree*	82%	0.10	1.04
	1000	*wKNN*	93%	0.01	1.01
rand_1%	10	*Tree*	81%	0.11	1.07
	10	*wKNN*	95%	0.01	1.00
	1000	*Tree*	77%	0.11	1.10
	1000	*wKNN*	93%	0.02	1.02
rand_rows	10	*Tree*	91%	0.01	1.01
	10	*wKNN*	93%	0.01	1.01
	1000	*Tree*	93%	0.01	1.01
	1000	*wKNN*	94%	0.01	1.00

Table 5 shows the result of performing 5-fold cross validation on each new dataset, using the *wKNN* and *Tree* models, simulating 10 or 1000 iterations in the execution times of the triangular solvers.

The results in the table show that the models present a remarkably higher performance on all the extended datasets, with accuracy ratios that exceed 90% in most cases, and negligible ARE and TRRE. However, these results are largely explained by overfitting. It is possible that the difference in the features between two random versions of the same matrix is too small to add significant variety to the dataset. This causes that, when performing cross validation, the instances in the test sets are too similar to the instances of the training set.

Table 6: Performance of the models trained with the extended datasets when predicting instances of the original dataset.

Dataset	Iters.	Model	Acc.	ARE	TRRE
rand_100%	10	*Tree*	51%	0.85	2.06
	10	*wKNN*	51%	0.86	2.07
	1000	*Tree*	37%	1.73	2.05
	1000	*wKNN*	40%	1.68	2.14
rand_1%	10	*Tree*	69%	0.33	1.52
	10	*wKNN*	76%	0.22	1.39
	1000	*Tree*	65%	0.58	1.50
	1000	*wKNN*	76%	0.52	1.55
rand_rows	10	*Tree*	31%	0.89	1.84
	10	*wKNN*	34%	0.85	1.82
	1000	*Tree*	41%	1.27	2.02
	1000	*wKNN*	41%	1.28	2.05

To assess the performance of the models trained with the extended datasets on real problems, Table 6 shows the results using the original dataset as test for the three random training sets. The performance of the models drops radically in comparison with the cross validation case, showing worse results the more random the dataset. A possible improvement of this strategy can be to create training sets with a much larger number of matrices, combining different degrees of randomness.

6. Evaluation on real cases

To assess the effectiveness of our approach on real-world applications we shall use the Preconditioned Conjugate Gradient (PCG) iterative method, with an incomplete factorization with 0 fill-in (ILU0) as preconditioner [2]. This implies that two sparse triangular linear systems need to be solved in each iteration of the PCG. We will analyze the runtime for the entire iteration, obtained for each of the SPTRSV implementations considered previously in this work, and compare it with the runtime obtained by using the SPTRSV implementation predicted as optimal by the Weighted-KNN model in each case.

For this experiment, we selected 25 symmetric positive-definite (SPD) matrices from the original dataset. To train the machine learning method, we

used the original dataset, with all but these 25 SPD matrices. We produced two models, labeling the observations with the SPTRSV implementation that takes less time to perform one analysis phase (if it is required) plus 10 and 1000 solution phases, respectively.

The features used in the training and prediction phases are the dimension n, the number of nonzeros (nnz), and the estimation of the maximum number and standard deviation of nonzeros per row given by *max_p128* and *std_p128*. The characteristics of these matrices are reported in Table 7.

Table 7: Characteristics of the SPD matrices employed to test our machine learning approach in the context of a PCG iterative solver.

id	matrix	n	nnz	nnz_avg	nnz_max	nnz_std
1	af_shell3	504,855	9,046,865	17.92	18	5.26
2	af_shell4	504,855	9,046,865	17.92	18	5.26
3	af_shell7	504,855	9,046,865	17.92	18	5.26
4	af_shell8	504,855	9,046,865	17.92	18	5.26
5	bmw7st_1	141,347	3,740,507	26.46	44	14.32
6	hood	220,542	5,494,489	24.91	32	11.54
7	ship_003	121,728	4,103,881	33.71	49	16.94
8	shipsec1	140,874	3,977,139	28.23	39	10.54
9	shipsec5	179,860	5,146,478	28.61	43	11.12
10	shipsec8	114,919	3,384,159	29.45	43	11.15
11	thermal2	1,228,045	4,904,179	3.99	5	1.77
12	G2_circuit	150,102	438,388	2.92	3	0.53
13	G3_circuit	1,585,478	4,623,152	2.92	3	0.52
14	apache2	715,176	2,766,523	3.87	3	0.34
15	boneS01	127,224	3,421,188	26.89	32	11.38
16	Dubcova3	146,689	1,891,669	12.90	14	3.54
17	parabolic_fem	525,825	2,100,225	3.99	6	2.06
18	ecology2	999,999	2,997,995	3.00	2	0.04
19	2cubes_sphere	101,492	874,378	8.62	10	3.74
20	thermomech_TC	102,158	406,858	3.98	6	2.04
21	thermomech_TK	102,158	406,858	3.98	6	2.04
22	thermomech_dM	204,316	813,716	3.98	6	2.04
23	offshore	259,789	2,251,231	8.67	10	3.12
24	Emilia_923	923,136	20,964,171	22.71	23	4.96
25	Geo_1438	1,437,960	32,297,325	22.46	23	4.44
26	bundle_adj	513,351	10,360,701	20.18	39	15.74

The implementation of the PCG method is that of the *conjugateGradi-*

entPrecond code distributed with NVIDIA CUDA Toolkit 9.2, adapted to use double precision. The stopping criteria for the iteration is $\|r_i\|/\|r_0\| < 10^{-8}$ or a maximum of 5000 iterations, where r_i is the norm of the residual at iteration i.

Table 8: Runtime (in seconds) of the PCG using each solver to perform the SPTRSV that correspond to the computation of the preconditioned residual in every iteration. In the columns Pred 10 and Pred 1000, the triangular solver used is that predicted to be the best by the *wKNN* model trained with the runtimes of 10 and 1000 solution phases respectively.

			time PCG								
Id	Iters	Res. Norm	*cuspv1*	*cuspv2*	*cuspv2$_{nl}$*	*baseline*	*sf_ nan*	*sf_ order*	*sf_ mr*	Pred 10	Pred 1000
1	1056	7.06E-06	50.69	31.82	120.87	374.38	280.22	13.89	12.96	31.82	12.96
2	1056	7.06E-06	50.68	31.93	120.92	374.87	282.55	14.45	12.90	31.93	12.90
3	1056	7.00E-06	50.70	31.89	121.01	371.18	278.76	14.43	13.00	31.89	13.00
4	1056	7.00E-06	50.69	32.43	120.90	374.23	279.84	13.87	12.96	32.43	12.96
5	5001	8.89E+04	59.00	25.74	42.44	97.25	72.65	14.86	14.36	14.36	14.36
6	5001	1.82E+03	51.17	27.78	22.47	54.98	43.98	17.00	16.80	16.80	16.80
7	1739	2.92E-06	102.30	87.72	27.03	47.23	30.82	27.92	26.15	30.82	26.15
8	892	3.27E-06	27.44	22.29	8.21	15.71	11.00	6.61	6.12	6.12	6.12
9	764	4.18E-06	30.70	24.69	11.09	22.52	15.06	7.77	6.92	6.92	6.92
10	1806	2.85E-06	71.50	71.05	20.48	36.12	25.61	21.99	19.24	19.24	19.24
11	2289	1.10E-05	40.85	20.50	32.76	56.38	41.14	17.84	14.50	20.50	14.50
12	542	3.81E-06	4.74	1.72	5.41	14.95	11.38	1.37	2.06	11.38	2.06
13	906	1.25E-05	30.91	11.59	228.16	301.07	232.00	9.76	14.89	11.59	14.89
14	772	7.97E-06	8.10	3.72	36.28	134.35	103.40	3.17	3.33	36.28	3.33
15	601	3.52E-06	8.65	6.49	6.94	17.08	12.78	2.20	2.04	2.04	2.04
16	144	3.59E-06	5.88	3.79	1.69	4.57	3.26	1.41	1.69	3.79	1.69
17	1132	7.07E-06	3.01	3.36	3.11	3.38	2.82	2.94	1.51	3.36	1.51
18	1879	9.93E-06	45.13	19.50	493.95	1.133.15	853.89	16.29	19.97	19.50	16.29
19	10	2.23E-06	1.71	0.89	0.65	0.96	0.71	0.49	0.47	0.89	0.49
20	8	3.96E-07	0.26	0.01	0.01	0.01	0.01	0.02	0.01	0.01	0.02
21	5001	1.48E+01	14.14	7.61	4.51	5.41	4.41	6.82	3.92	4.41	6.82
22	8	5.60E-07	0.47	0.02	0.01	0.01	0.01	0.03	0.02	0.01	0.02
23	490	5.07E-06	20.54	12.97	15.91	34.67	25.72	5.50	5.91	12.97	5.91
24	454	9.33E-06	39.97	19.86	24.12	61.32	44.57	9.38	8.69	19.86	8.69
25	481	1.14E-05	48.95	25.96	26.18	66.20	49.10	12.24	11.68	25.96	11.68
26	5001	5.80E+04	46.59	44.08	43.47	39.27	35.87	36.43	36.73	44.08	36.73
total			864.77	569.44	1.538.61	3.641.26	2.741.57	278.68	268.83	438.97	268.07

Table 8 shows the result of this experiment. For the majority of cases, the iteration count of the PCG was rather large, much closer to 1000 than to 10 (the instances with 5001 iterations did not converge). This explains the higher performance of the model trained with the simulation of 1000 solution phases. Moreover, for this particular set of matrices, the *sf_ mr* and *sf_ order* routines render similar performance in general, and have lower execution times than the rest of the variants. The machine learning model predicts that the best variant will be one of these two, though it often fails to estimate which of them will be the best. This results in a rather low accuracy

(62%), but a fair performance, taking slightly less time to compute the whole set than that of the best method (*sf_ mr*), and attaining a significant speedup regarding the rest of the methods.

For the instances that took a small number of iterations (close to 10) the prediction of the model trained with the runtimes of 1000-solution-phase was not considerably worse (in one case it was better) than that of the 10-solution-phase counterpart. The results obtained for this set of matrices suggest that the former model can adapt better to the general case than the latter. However, the results can differ depending on the specific set of matrices, and a deeper study is in need. Ideally, it would be desirable to count with an estimation of the iteration count for a certain matrix, based on the convergence behavior of the iterative method in mind. This is an interesting line to explore as part of future work.

7. Final remarks and future work

In this work we proposed an automatic procedure to select the best performing GPU implementation of SPTRSV for a given sparse triangular linear system. Our proposal includes the evaluation of different machine learning strategies and a careful study of the linear system features considered to compute the selection.

To address this problem we performed a brief revision of the different techniques presented in previous efforts for the solution of this operation on GPUs. Specifically, we identified seven SPTRSV routines for the CSR matrix format and experimentally evaluate their performance on 981 different sparse triangular linear systems. To simulate the application of a Krylov subspace solvers we multiply the runtime of the solution phase by 10 and 1000 fictitious iterations, creating two initial datasets.

We applied the machine learning techniques bundled with MATLAB® to later perform a more detailed study on *Tree* and *wKNN*, which showed the best relation between accuracy and prediction speed. For the initial datasets, applying the solver predicted by the *wKNN* model for each matrix results in up to 23% lower runtime than applying always the *sf_ mr* solver, which has the best average performance.

To assess the importance of each feature on these two models, we tested their performance when progressively removing features from the training and prediction stages. Here we observed that the most relevant features are those related with the level sets and the length of the rows. The results

of this experiment also suggest that the later are particularly important for sparse matrices of small and medium scale.

The rest of the work aims to improve the performance of the models in terms of both accuracy and speed. To enhance their accuracy, we constructed larger datasets by generating additional sparse matrices through random perturbations of the original ones. Performing cross validation on the extended datasets yields accuracy results of up to 96%. However, training the models with the extended dataset to make predictions on the original one showed poor results.

To improve the speed of the procedure, we replaced two of the features which are more costly to compute, by approximate values obtained through cheaper heuristic procedures. The results showed that the use of these estimations does not significantly degrade the accuracy of the models. Regarding the heuristics, the *max_p128*/*std_p128* combination obtained remarkably good accuracy results requiring a smaller computational effort than most other combinations.

Finally, we tested the automatic selection procedure in the context of a Krylov subspace solver. We selected 25 SPD matrices from the SuiteSparse collection and compared the performance of the PCG method using the selected SPTRSV routines, with the performance obtained using the routine predicted by the *wKNN* model in each case. For this particular set of matrices, the *sf_mr* and *sf_order* routines, in general, present similar performance and have lower execution times than the rest of the variants. The machine learning model often fails to predict which routine will be the best, which results in a rather low accuracy result (62%), but a good result in terms of performance, taking slightly less time to compute the whole set than that of the best method (*sf_mr*), and significantly outperforming the rest of the methods.

In future work we plan to extend this investigation along several lines. First of all, it is interesting to evaluate or develop new SPTRSV GPU methods, even relying in sparse storage formats different to the conventional CSR and CSC. We will explore different strategies to produce more effective randomized datasets that avoid the overfitting problem that arose in this effort. We also plan to develop heuristics for other costly features, such as the number of level sets, and evaluate their performance on the machine learning models. Additionally, we intend to provide parallel implementations of both the features and the model computations, so that all the procedure (at least the inference stage) can be performed on the GPU. Finally, it is interesting

to evaluate the incorporation of analytic or machine learning models that allow to estimate a priori the iteration count of a Krylov subspace solver, in order to apply our procedure more effectively.

Acknowledgments

The researchers from *UdelaR* were supported by PEDECIBA.

[1] T. A. Davis, Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2), Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

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

[3] J. H. Saltz, Aggregation methods for solving sparse triangular systems on multiprocessors, SIAM J. Sci. Stat. Comput. 11 (1) (1990) 123–144. doi:10.1137/0911008.

[4] E. Rothberg, A. Gupta, Parallel ICCG on a hierarchical memory multiprocessor - addressing the triangular solve bottleneck, Parallel Computing 18 (7) (1992) 719 – 741. doi:http://dx.doi.org/10.1016/0167-8191(92)90041-5.

[5] E. Totoni, M. T. Heath, L. V. Kale, Structure-adaptive parallel solution of sparse triangular linear systems, Parallel Computing 40 (9) (2014) 454 – 470. doi:https://doi.org/10.1016/j.parco.2014.06.006.
URL http://www.sciencedirect.com/science/article/pii/S0167819114000799

[6] M. Naumov, Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU, NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011 1 (2011).

[7] X. Wang, W. Liu, W. Xue, L. Wu, swsptrsv: A fast sparse triangular solve with sparse level tile layout on sunway architectures, SIGPLAN Not. 53 (1) (2018) 338–353. doi:10.1145/3200691.3178513.
URL http://doi.acm.org/10.1145/3200691.3178513

[8] M. M. Wolf, M. A. Heroux, E. G. Boman, Factors Impacting Performance of Multithreaded Sparse Triangular Solve, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 32–44. doi:10.1007/978-3-642-19328-6_6.
URL http://dx.doi.org/10.1007/978-3-642-19328-6_6

[9] R. Li, Y. Saad, Gpu-accelerated preconditioned iterative linear solvers, The Journal of Supercomputing 63 (2) (2013) 443–466.

[10] E. Anderson, Y. Saad, Solving sparse triangular linear systems on parallel computers, International Journal of High Speed Computing 01 (01) (1989) 73–95. arXiv:http://www.worldscientific.com/doi/pdf/10.1142/S0129053389000056, doi:10.1142/S0129053389000056.
URL http://www.worldscientific.com/doi/abs/10.1142/S0129053389000056

[11] A. George, M. T. Heath, J. Liu, E. Ng, Solution of sparse positive definite systems on a shared-memory multiprocessor, International Journal of Parallel Programming 15 (4) (1986) 309–325. doi:10.1007/BF01407878.

[12] D. Erguiz, E. Dufrechou, P. Ezzatti, Assessing sparse triangular linear system solvers on GPUs, in: 2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), 2017, pp. 37–42. doi:10.1109/SBAC-PADW.2017.15.

[13] E. Dufrechou, P. Ezzatti, E. S. Quintana-Ortí, Automatic selection of sparse triangular linear system solvers on gpus through machine learning techniques, in: 31st International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2019, Campo Grande, Brazil, October 15-18, 2019, IEEE, 2019, pp. 41–47. doi:10.1109/SBAC-PAD.2019.00020.
URL https://doi.org/10.1109/SBAC-PAD.2019.00020

[14] M. Paluszek, S. Thomas, MATLAB Machine Learning, 1st Edition, Apress, Berkely, CA, USA, 2016.

[15] N. Ahmad, B. Yilmaz, D. Unat, A prediction framework for fast sparse triangular solves (2020).

URL https://parcorelab.ku.edu.tr/sites/parcorelab.ku.edu.tr/files/project-images/Papers/Europar2020.pdf

[16] G. H. Golub, C. F. Van Loan, Matrix Computations, four Edition, The Johns Hopkins University Press, 2013.

[17] W. Liu, A. Li, J. Hogg, I. S. Duff, B. Vinter, A synchronization-free algorithm for parallel sparse triangular solves, in: European Conference on Parallel Processing, Springer, 2016, pp. 617–630.

[18] W. Liu, A. Li, J. D. Hogg, I. S. Duff, B. Vinter, Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides, Concurrency and Computation: Practice and Experience 29 (21) (2017).

[19] E. Dufrechou, P. Ezzatti, 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), 2018, pp. 196–203. doi:10.1109/PDP2018.2018.00034.

[20] E. Dufrechou, P. Ezzatti, A new GPU algorithm to compute a level set-based analysis for the parallel solution of sparse triangular systems, in: 2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Canada, 2018, IEEE Computer Society, 2018.

[21] E. Dufrechou, P. Ezzatti, Using analysis information in the synchronization-free GPU solution of sparse triangular systems, Concurr. Comput. Pract. Exp. 32 (10) (2020). doi:10.1002/cpe.5499.
URL https://doi.org/10.1002/cpe.5499

[22] L. Breiman, J. H. Friedman, R. A. Olshen, C. J. Stone, Classification and Regression Trees, Wadsworth and Brooks, Monterey, CA, 1984.

[23] G. P. Patil, C. Taillie, Diversity as a concept and its measurement, Journal of the American Statistical Association 77 (379) (1982) 548–561. doi:10.1080/01621459.1982.10477845.

[24] S. A. Dudani, The distance-weighted k-nearest-neighbor rule, IEEE Transactions on Systems, Man, and Cybernetics SMC-6 (4) (1976) 325–327. doi:10.1109/TSMC.1976.5408784.