A Split Execution Model for SpTRSV

Najeeb Ahmad, Buse Yilmaz, and Didem Unat

Abstract—Sparse Triangular Solve (SpTRSV) is an important and extensively used kernel in scientific computing. Parallelism within SpTRSV depends upon matrix sparsity pattern and, in many cases, is non-uniform from one computational step to the next. In cases where the SpTRSV computational steps have contrasting parallelism characteristics– some steps are more parallel, others more sequential in nature, the performance of an SpTRSV algorithm may be limited by the contrasting parallelism characteristics. In this work, we propose a split-execution model for SpTRSV to automatically divide SpTRSV computation into two sub-SpTRSV systems and an SpMV, such that one of the sub-SpTRSVs has more parallelism than the other. Each sub-SpTRSV is then computed using different SpTRSV algorithms, which are possibly executed on different platforms (CPU or GPU). By analyzing the SpTRSV Directed Acyclic Graph (DAG) and matrix sparsity features, we use a heuristics-based approach to (i) automatically determine the suitability of an SpTRSV for split-execution, (ii) find the appropriate split-point, and (iii) execute SpTRSV in a split fashion using two SpTRSV algorithms while managing any required inter-platform communication. Experimental evaluation of the execution model on two CPU-GPU machines with a matrix dataset of 327 matrices from the SuiteSparse Matrix Collection shows that our approach correctly selects the fastest SpTRSV method (split or unsplit) for 88% of matrices on the Intel Xeon Gold (6148) + NVIDIA Tesla V100 and 83% on the Intel Core I7 + NVIDIA G1080 Ti platform achieving speedups up to $10\times$ and $6.36\times$, respectively.

Index Terms—Sparse triangular solve, CPU-GPU computing, heterogeneous computing, sparse linear systems, SpTRSV, SpTS

1 Introduction

Sparse triangular solve (SpTRSV) is an important computational kernel used in many scientific and engineering applications and appears in direct and iterative solvers and least square problems. When compared with other sparse kernels such as sparse matrix-vector multiplication (SpMV) and sparse matrix transposition [1], SpTRSV is an inherently sequential operation. Consequently, it has been observed to be one of the major time-consuming operations in many numerical applications [2], [3].

The sequential nature of SpTRSV stems from inter-dependencies among SpTRSV computations, forcing it to be completed in several sequential steps (or *levels*). These dependencies can be represented as a direct-acyclic-graph (DAG). Empirical evidence from existing research has shown that while highly parallel algorithms perform exceedingly well for DAGs having few levels with a high number of unknowns, sequential algorithms perform better for matrices with DAGs having a high number of levels with few unknowns [3], [4], [5], [6]. The number of levels and unknowns computed within each level depends upon the sparsity pattern of the input triangular matrix. For systems with matrices having a mix of highly parallel and sequential levels, a parallel algorithm is expected to provide benefits for levels with a large number of unknowns, but the performance is expected to deteriorate for levels with few unknowns [7]. The converse can be said about the performance of sequential algorithms for such matrices. This observation leads to an interesting research question of whether one can split the single SpTRSV into two parts and use a different algorithm for each part to solve the system.

- *Najeeb Ahmad and Didem Unat are with the Department of Computer Science and Engineering, Koç University, Istanbul, 34450, Turkey. Buse Yilmaz is with the Department of Computer Science, Istinye University, Istanbul, 34010, Turkey.*

There are several challenges associated with developing such a split execution model to improve overall performance. Firstly, the model must decide whether a given DAG is suitable for split execution, in other words, whether the matrix has enough disparity in the degree of parallelism in various parts of its DAG such that one part of the DAG is more parallel and the other is more serial. Once a DAG is found to be suitable for split execution, the split point and affinity of each part of the DAG to the respective algorithms and platforms (CPU or GPU) are to be determined. Then the model must manage data structures such that each SpTRSV algorithm receives all its inputs in the desired form while keeping the overheads as low as possible. Finally, the CPU-GPU communication must be managed so that the communication costs do not cancel out the attained performance gains.

To address these challenges, we develop an extensible and fully automatic execution framework that uses light-weight statistical DAG analysis and a heuristic-based algorithm to distribute SpTRSV computations between two algorithms and potentially execute them on two different platforms. We leverage an existing portfolio of SpTRSV algorithms and choose the appropriate one for each part. Our framework transparently handles all the data structure changes and data movement required for migrating the work from one platform to another during execution. The framework is implemented as a C++/CUDA library and evaluated on two state-of-the-art CPU-GPU platforms.

To the best of our knowledge, the closest work to ours is presented in [8]. The author proposes a strategy, called HTS, to partition SpTRSV computation into parts and perform SpTRSV using a level-scheduling algorithm on one part [9] and a recursive blocking algorithm on the other. The strategy is designed to work on multi-core CPU systems. Another closely related work by Lu et al. [10] uses a recursive blocking strategy for SpTRSV on the GPUs in which

they recursively split a triangular matrix into parts and use SpMV and SpTRSV on the sub-matrices. Their method aims to equally divide (same number of rows) the triangular matrix based on the pre-selected recursion depth and adaptively select SpMV and SpTRSV kernels for the resulting sub-matrices. Unlike our work, they always split the matrix, decide split points based on the recursion depth (instead of matrix characteristics), and target GPUs only. While our framework utilizes the HTS library for SpTRSV execution on the CPU, we target a broader range of algorithms and platforms (e.g., GPU). Moreover, we develop heuristics to assess the suitability of a matrix for split-execution as split-execution does not always provide performance benefits. Lastly, we explore cases where split execution can enhance the performance of certain matrices even if the same algorithm is used for each split part.

The contributions of this work are summarized below:

- We devise a split execution model for SpTRSV to improve the SpTRSV performance.
- We develop heuristics to determine the suitability of a sparse triangular system for split execution and determine the split-point for such systems.
- We select appropriate algorithms and platforms for each split part for systems found suitable for split execution.
- We demonstrate how split execution can enhance the performance of an SpTRSV solver for certain matrices even if the same algorithm is used for each sub-system.
- We evaluate the performance and overhead of the approach on two CPU-GPU platforms using a diverse set of matrices.
- Finally, the split SpTRSV framework is available at https://github.com/ParCoreLab/Split_SpTRSV.

2 Background and Motivation

SpTRSV refers to the solution of linear systems of the form $Lx = b$ or $Ux = b$, where L and U are strictly lower and upper triangular sparse matrices, respectively, and b is a dense vector. In this paper, we consider square L and U matrices of sizes $n \times n$ and vector b of size n. As the solution of the lower triangular system ($Lx = b$) is similar to the solution of the upper triangular system ($Ux = b$), subsequent discussions will focus on the lower triangular system.

Algorithm 1 shows serial SpTRSV for a lower triangular system where matrix L is stored in CSR (compressed sparse row) format with row pointer array $rowPtr$, column indices array $colIdx$, and values array val. As evident, the computation of an unknown x (line 6) depends upon previously calculated x values (line 4). This makes SpTRSV an inherently sequential operation. In the case of sparse matrices, however, as most of the matrix entries are zero, some unknowns may depend upon none or the same group of previously calculated unknowns rendering their parallel computation possible. To elaborate on this, consider the sparsity pattern of a sample lower triangular matrix L shown in Figure 1(a). Here, the red dots represent non-zero values. The relationship between computations of different unknowns for this matrix is represented by a DAG in Figure 1(b) where

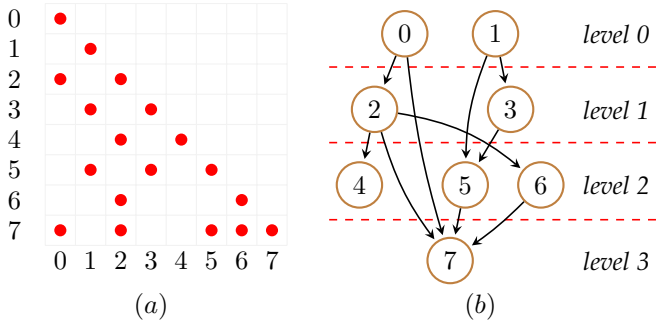

Fig. 1. (a) Sparsity pattern of a sample lower triangular matrix L and (b) its dependency graph (DAG)

Algorithm 1 SpTRSV in CSR format

```
1: for i = 0, ..., n − 1 do
2:     sum = 0
3:     for j = rowPtr[i], ..., rowPtr[i + 1] − 2 do
4:         sum+ = val[j] × x[colIdx[j]]
5:     end for
6:     x[i] = (b[i] − sum)/val[rowPtr[i + 1] − 1])
7: end for
```

the nodes represent unknowns and edges represent their inter-dependencies. Nodes in the DAG are arranged from top to bottom in *levels* such that nodes within a level have no inter-dependencies. Consequently, the parallel SpTRSV is intra-level parallel and inter-level sequential.

For the parallel solution of sparse triangular systems, two algorithm classes are widely used, namely 1) *level-scheduling* algorithms [11], and 2) *self-scheduling* or *synchronization-free* algorithms [12]. Many implementations of these algorithms are available for the CPUs [9], [13], [14] and the GPUs [4], [5], [6], [15]. These algorithms are comprised of two phases, an *analysis*, and a *solve* phase. In level-scheduling algorithms, the analysis phase discovers levels [11] in the DAG while in self-scheduling algorithms indegrees of the nodes are calculated [5]. Some variants of self-scheduling algorithms also discover levels in the DAG [6], [16]. In practical settings, a solver can have the same triangular matrix for tens of iterations and the same nonzero pattern for hundreds or more iterations. Based on this observation, the analysis phase can be further classified into *symbolic analysis* and *numerical* phases [8]. Here, the symbolic analysis phase is the same as the previously described analysis phase, while the numerical phase refers to modifying the matrix nonzero values without changing its nonzero pattern. In the solve phase, the triangular system is solved using the information gathered during the symbolic analysis phase. During the solve phase, level-scheduling algorithms require a synchronization barrier between two consecutive levels, i.e. computations on the next level can only begin once all unknowns within the current level have been completed. In self-scheduling algorithms, computations on an unknown can start as soon as all its dependencies are met.

Existing research has shown that while a highly parallel algorithm performs exceedingly well for DAGs having few levels with a high number of unknowns (*fat levels*), a sequential algorithm performs better for matrices with DAGs

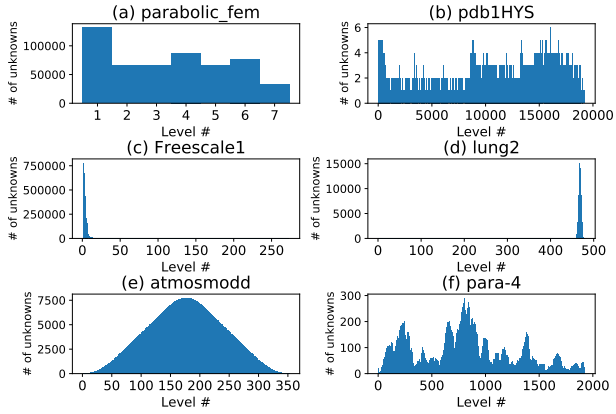

Fig. 2. Rows(Unknowns) per-level plots for SpTRSV DAGs of some matrices from SuiteSparse collection

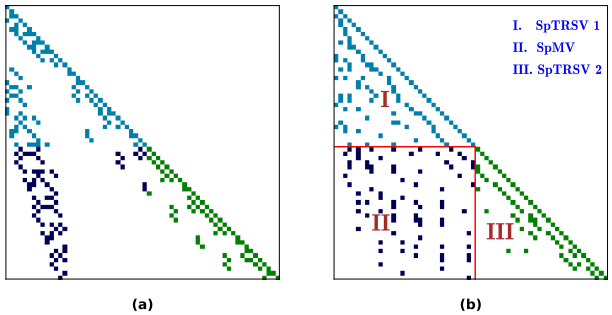

Fig. 3. (a) Sparsity pattern of the lower triangular part of matrix *bfwa62* from the SuiteSparse collection. (b) Sparsity pattern of the DAG rearranged version of the matrix in (a) and its division into two triangular matrices and a rectangular sparse matrix for split execution

having a high number of levels with few unknowns (*thin levels*) [3], [5], [17]. Figure 2 shows a representative set of the *unknowns-per-level* histograms for SpTRSV DAGs of some matrices taken from the SuiteSparse collection [18]. The DAG for matrix *parabolic_fem*, shown in Figure 2(a), has only 7 levels, each with tens of thousands of unknowns. Such SpTRSV DAGs are perfect candidates for execution using parallel algorithms. A less parallel DAG for *pdb1HYS* is shown in Figure 2(b) that has a large number of levels, each with less than ten unknowns. DAGs of this type are more suitable for execution using less parallel algorithms. The matrices *Freescale1* and *lung2* in Figures 2(c) and (d) have DAGs with fat levels on one side and thin levels on the other side of the histograms. For such matrices, while a highly parallel algorithm is expected to give better performance on fat levels, its performance will drop for computations on thin levels. Similar can be said about a less parallel SpTRSV algorithm whose better performance on thin levels will be offset by its performance degradation on fat levels. In short, the presence of contrasting types of level fatness in DAG results in attaining less than optimal SpTRSV performance. Intuitively, for such DAGs, SpTRSV performance is expected to improve if fat levels are executed using a highly parallel algorithm and thin levels using a sequential one.

Other DAGs may have multiple regions suitable for a highly parallel or a sequential algorithm such as symmetric DAG of *atmosmodd* in Figure 2(e) that has thin levels on either side of the DAG and fat levels in the middle. In the DAG of Figure 2(f), there is no clear distinction between fat and thin levels as the other examples. For simplicity and to reduce overheads, we limit the number of split points to one in this work, leaving the study of multi-split-point SpTRSV execution as future work.

3 The SpTRSV Split Execution Model

The split execution model of SpTRSV is based on the concept that a triangular matrix can be split at a level into multiple on-diagonal smaller sub-triangular matrices and many rectangular sub-matrices. Consider the lower triangular matrix corresponding to the ILU(0) factorization of a sparse matrix in Figure 3(a) and the row rearranged version of the same matrix shown in Figure 3(b). The rows are rearranged based on SpTRSV DAG such that the rows corresponding to level 1 are on top of the matrix, followed by rows in level 2 and so on. Figure 3(b) also shows how the row rearranged matrix can be subdivided into two sub-triangular matrices (I and III) and a rectangular sparse matrix (II). Then by applying SpTRSV individually on the triangular matrices and sparse matrix-vector multiplication (SpMV) on the rectangular sparse matrix in a specific order, the overall SpTRSV solution for the triangular matrix can be obtained. In particular, by (i) getting the solution of the first on-diagonal triangle (SpTRSV 1), (ii) performing SpMV of the SpTRSV 1 solution with the rectangular sparse matrix (II) and projecting the result to the right-hand-side of SpTRSV 2, and finally, (iii) solving the second triangular system, SpTRSV 2, one can obtain the overall solution. This split-execution strategy allows SpTRSV 1, SpMV, and SpTRSV 2 to be potentially solved by using different algorithms on different architectures. We utilize this splitting approach to distribute SpTRSV computation between parallel and sequential algorithms with our split SpTRSV execution model. Each of the SpTRSV 1, SpMV, SpTRSV 2 can execute on a CPU or on a GPU.

3.1 Two-algorithm vs Single-algorithm Split

The main thrust of this work is to enhance SpTRSV performance through split SpTRSV execution with different algorithms for SpTRSV 1 and SpTRSV 2 (one parallel, other less parallel or sequential). However, as an auxiliary case study, we also explore some cases where the split-execution using the same algorithm for SpTRSV 1 and SpTRSV 2 can enhance the SpTRSV performance versus the unified SpTRSV. Based on this observation, we further divide split execution into

- (i) **Two-algorithm-split execution**, in which different algorithms are used for SpTRSV 1 and SpTRSV 2.
- (ii) **Single-algorithm-split execution**, in which the same algorithm is used for SpTRSV 1 and SpTRSV 2.

To explain the performance gain from single-algorithm-split execution, we note that for some highly parallel SpTRSV implementations, a fixed number of threads or warps (for GPUs) are allocated for each row or column. For instance, some Sync-Free SpTRSV implementations on the NVIDIA GPUs [5], [6], [15], [16] assign a fixed number

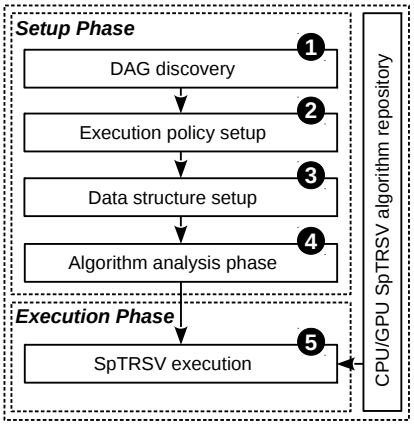

Fig. 4. SpTRSV split execution model block diagram. The model is comprised of two phases, a one-time setup phase and an execution phase that can execute multiple times, potentially with a different right-hand side

of warps per row or column. This may lead to serialization of some computations if the parallel operations to be performed per row or column are more than the assigned number of warps. In such cases, splitting the triangular system into smaller sub-triangular systems reduces row and column lengths thus removing the serialization bottleneck [10]. In addition, splitting replaces some SpTRSV computations with highly parallel SpMV computations thus resulting in better overall performance. However, like SpTRSV, some operations in an SpMV implementation, e.g. [19], may also be serialized if a row or column lengths of the matrix are more than the number of warps assigned per row or column. Therefore, one needs to consider both the row and column lengths in a matrix to decide whether the single-algorithm-split can improve the overall SpTRSV performance.

3.2 Execution Framework Overview

The split SpTRSV execution model is designed to automatically split and distribute computations of a single SpTRSV iteration based on the analysis of the input triangular matrix DAG. Figure 4 shows the block diagram of the execution framework. The framework works in two phases; a one-time *setup phase* and an *execution phase* that can be executed multiple times, potentially with a different right-hand-side each time. The setup phase is completed in four steps:

1) *DAG discovery* step calculates statistics such as the number of levels, number of rows, and nonzeros in each level.
2) *Execution policy setup* step, in which the DAG statistics from step (1) are used to analyze the DAG and decide the SpTRSV execution policy which can either be *unified* or *split*. The *unified* policy refers to SpTRSV computation over the non-split DAG on a single architecture using a single algorithm. The *split* policy refers to the division of the DAG re-arranged matrix into three parts as previously described. The algorithm for each part is also determined based on the DAG characteristics. For the *unified execution policy*, the SpTRSV algorithm is determined by a few simple heuristics.
3) The *data structure setup* step sets up the data structures for the selected SpTRSV algorithm(s) and SpMV in case of split-execution.
4) In the *algorithm analysis* step, the analysis phase of the selected algorithm(s) is executed. Once it is completed, the framework is ready to execute SpTRSV with the selected execution policy.

In the *execution phase*, the framework uses the execution policy and data structures from the setup phase to transparently execute SpTRSV on a CPU-GPU system from an extensible repository of existing SpTRSV algorithms. Any required CPU-GPU communication is automatically managed.

4 Methodology

In this section, we present our methodology for achieving an efficient split-execution model and explain details of the steps shown in Figure 4.

4.1 Step 1: DAG Discovery

The DAG discovery process is similar to the symbolic analysis phase of level-scheduling algorithms in which the input triangular matrix is analyzed for the order of unknowns (rows) in the DAG and levels within the DAG are calculated. In addition, the framework calculates the number of unknowns and non-zeros within each level and keeps them in a data structure that stores statistics for each level. For this purpose, a modified breadth-first search (BFS) algorithm similar to one proposed in [4] is used. Other SpTRSV DAG features (explained in Section 4.2.1) are also computed in this step.

4.2 Step 2: Execution Policy Setup

Figure 5 shows the flowchart of the execution policy setup procedure. In the Splitting Decision phase (labeled as (1)), we decide whether the triangular system would benefit from split execution or not. This phase checks whether the DAG can be divided into regions of disparate parallelism characteristics and executed with CPU-GPU split execution policy. If not, the DAG is further analyzed to determine if it is suitable for CPU-CPU, GPU-GPU split execution, or unified SpTRSV on the CPU or on the GPU. To keep the overhead low, each phase of the Splitting Decision heuristic is modeled as a simple binary selection or binary classification problem, using cheap-to-calculate matrix features. In the DAG slicing and algorithm mapping phases (labeled as (2) and (3), respectively), we set one of the split or unified execution policies based on the output of the splitting decision.

4.2.1 Matrix Features for Execution Policy

The following features and parameters are used for our heuristics and binary classification:

- Matrix size (*rows*): Number of rows in the matrix,
- NNZs: Number of nonzeros in the triangular matrix,
- NL: Number of levels in the SpTRSV DAG,
- Average parallelism (AP): Ratio of *rows* and *NL*,

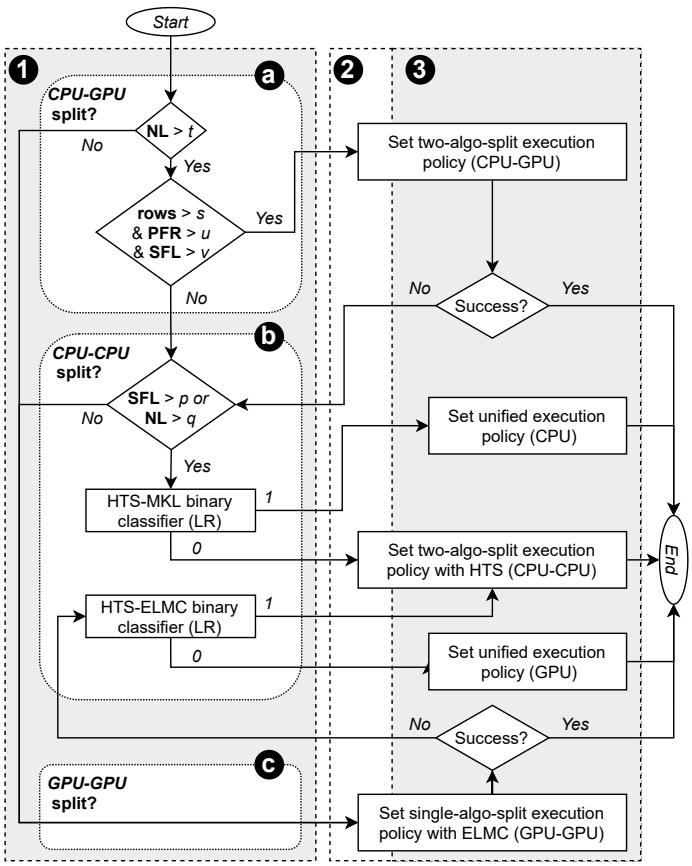

Fig. 5. Flowchart for setting up the execution policy. (1) The splitting decision heuristic determines whether the triangular matrix is suitable for split execution. (2) DAG slicing determines split-point and (3) algorithm mapping chooses the SpTRSV algorithm for each part. LR: Logistic regression model, NL: Number of levels, PFR: Parallel-friendly rows, SFL: Serial-friendly levels

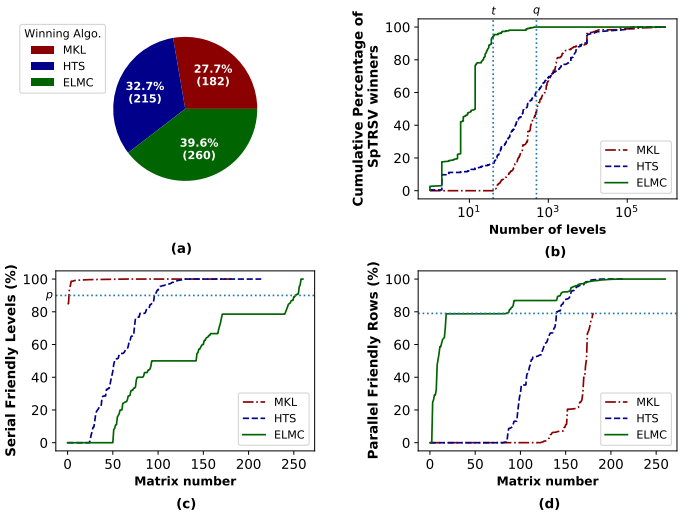

Fig. 6. Statistical analysis of training data set. (a) SpTRSV winning algorithm (algorithm with least execution time) breakdown for the 657 matrices (b) Cumulative percentage of SpTRSV winners plotted versus SpTRSV DAG levels in the input matrix (c) For each winning algorithm, percentage of serial friendly levels (SFL) in SpTRSV DAG (sorted in ascending order) (d) For each winning algorithm, percentage of parallel friendly rows(PFR) in SpTRSV DAG (sorted in ascending order). Blue dashed lines indicate the chosen thresholds for the Splitting Decision algorithm. The value of m for (c) and (d) is 200.

- Parallel-friendly levels *(PFL)*: Number of levels with at least m rows, as a percentage of the total number of levels. The remaining levels are categorized as serial-friendly levels (SFL),
- Parallel-friendly rows (PFR): Sum of rows in all PFLs as a percentage of the total number of rows,
- Maximum column length (*MCL*): Maximum column length per level, where column length refers to the number of nonzeros in a column,
- Maximum row length (*MRL*): Maximum row length per level, where row length refers to the number of nonzeros in a row,
- Average row length (*ARL*): Ratio of *nnzs* and *rows*

4.2.2 Splitting Decision

In this section, we present our splitting decision algorithm and the underlying statistical analysis. The basic criteria for the design of the algorithm are to make each decision step a binary decision (Yes or No) or binary classification problem using as few features as possible to keep the overheads low. The statistical analysis is performed on SpTRSV performance data of three SpTRSV implementations using a training data set of 657 matrices from the SuiteSparse Matrix Collection [18]. These three implementations are MKL [13], HTS [8], and ELMC [6]. MKL is Intel's Math Kernel Library, HTS is a library that performs SpTRSV in a split fashion on the CPU, and ELMC is the state-of-the-art SpTRSV algorithm designed for GPUs. In the analysis, it is assumed that the two-algorithm split uses MKL (sequential) for the sequential-friendly part and ELMC for the parallel-friendly part of the DAG.

(1-a) CPU-GPU split: The Splitting Decision algorithm first examines the suitability of the DAG for CPU-GPU split execution, labeled as (1-a) in Figure 5. Our decision process is based on several observations, also presented in Figure 6. The first observation is that below a certain number of levels $NL \leq t$, MKL never gives better performance than the other two algorithms, and splitting such DAGs is not expected to provide any benefits. Therefore, we only consider SpTRSV DAGs with number of levels greater than t for CPU-GPU split execution. t is selected to be 40 and indicated by the first vertical dotted line in Figure 6 (b).

The second observation is that all the matrices for which MKL wins have a high percentage of SFLs (85% and above as shown in Figure 6 (c)) while most of the ones for which ELMC wins have a high percentage of PFRs (80% and above as shown in Figure 6 (d)). Therefore, SFLs (above a certain threshold (u)) and PFRs (above another threshold (v)) can help identify parallelism disparity in SpTRSV DAG. Additionally, the algorithm checks for the matrix size to exclude matrices whose size is too small to be considered for CPU-GPU split execution. The thresholds u and v are tunable parameters that have been empirically selected as 10% and 60%, respectively, in this study. The minimum matrix size threshold (s) is set to 10K.

(1-b) CPU-CPU split:

For the matrices crossing the minimum NL threshold (t) but found unsuitable for CPU-GPU split execution due to insignificant or mixed parallelism disparity, we evaluate

their suitability for CPU-CPU execution. According to Figure 6 (c), whenever SFLs are above a certain threshold (p), either MKL (sequential) or HTS wins, except for some very few and small outliers ($<$ 3K rows) for which ELMC wins. Similarly, for the matrices for which NL is above threshold q (the second vertical dotted line to the right in Figure 6(b)), the winning algorithm is either HTS or MKL. For both these cases, we devise an HTS-MKL binary classifier based on a Logistic Regression model. For this purpose, we use HTS and MKL SpTRSV performance data for our training dataset of 657 matrices and select the features of average parallelism (AP), mean row length (ARL), and maximum column length (MCL) in the matrix to attain 10-fold cross-validation scores as shown in Figure 7 (a). The model is tested with our matrix test set of 327 matrices for which it achieves an accuracy of 83% with area under the ROC curve as shown in Figure 7 (b). Based on this analysis, we either employ unified execution with MKL or split execution with HTS on the CPU.

From Figure 6 (b), we observe that for matrices with $NL \leq t$, either HTS or ELMC SpTRSV algorithms win (have the least execution time). For such matrices, we select between HTS and ELMC algorithms using a simple binary classifier based on the Logistic Regression model (LR). As features of the model, we use the number of levels (NL), average parallelism (AP), mean row length (ARL), and maximum column length (MCL) in the matrix to attain 10-fold cross-validation scores as shown in Figure 7(c). The model attains an accuracy of 90% with the area under the ROC curve as shown in Figure 7(d). Based on the analysis, we either employ unified execution with ELMC on GPU or split execution with HTS on the CPU.

(1-c) GPU-GPU split: For the matrices with $NL \leq t$ or the decision step in Figure 5 (1-b) results in a "No", we first check whether maximum column length MCL is becoming a serialization bottleneck for these matrices for the GPU execution. If the splitting algorithm succeeds, the GPU-GPU split execution policy is used for such matrices. Otherwise, these matrices are tested by the HTS-ELMC binary classifier.

4.2.3 DAG Slicing and Algorithm Mapping

Setting the split execution policy requires determining the level at which the SpTRSV DAG is to be split into two parts, a process we refer to as *DAG slicing*. Both split and unified execution policies require appropriate algorithms to be assigned for the SpTRSV(s), referred to as *algorithm mapping*. The framework uses separate heuristics-based algorithms to set two-algorithm-split or single-algorithm-split execution policies. These algorithms are discussed next.

4.2.3.1 *DAG Slicing for CPU-GPU Execution*: Figure 8 shows the DAG slicing algorithm used by the CPU-GPU split policy. The algorithm starts by identifying whether most of matrix rows are covered by the first or the second half of the DAG levels (Figure 8(1-a)). Accordingly, the search for the parallel-friendly part begins in the first or second half of the DAG levels, respectively. When searching in the first half (Figure 8(1-b)), the algorithm marks level 0 as the start of the parallel-friendly part (*split1Start*) and keeps searching for the end level until either all PFLs are covered or the number of SFLs reaches its threshold (a% of the total levels). In the former cases, the last PFL is marked

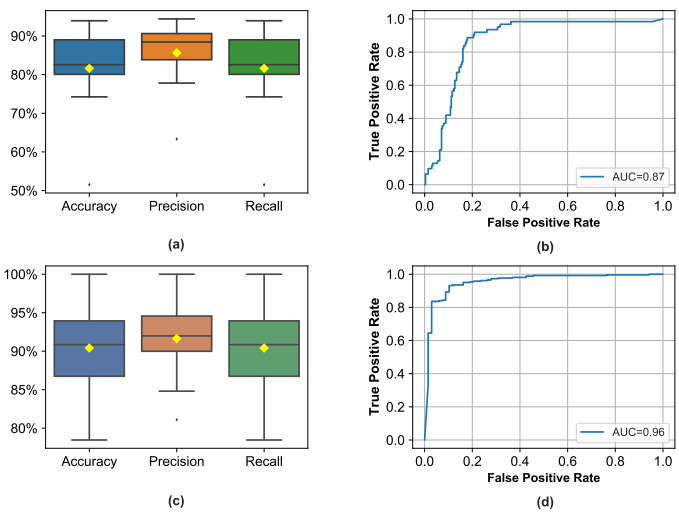

Fig. 7. (a), (b) HTS-MKL Logistic Regression-based binary classifier performance. Mean 10-fold cross-validation scores for accuracy, precision, and recall are 81%, 85%, and 81%, respectively. The area under the ROC curve is 0.87. (c), (d) HTS-ELMC Logistic Regression-based binary classifier performance. Mean 10-fold cross-validation scores for accuracy, precision, and recall are 90%, 92%, and 90%, respectively. The area under the ROC curve is 0.96

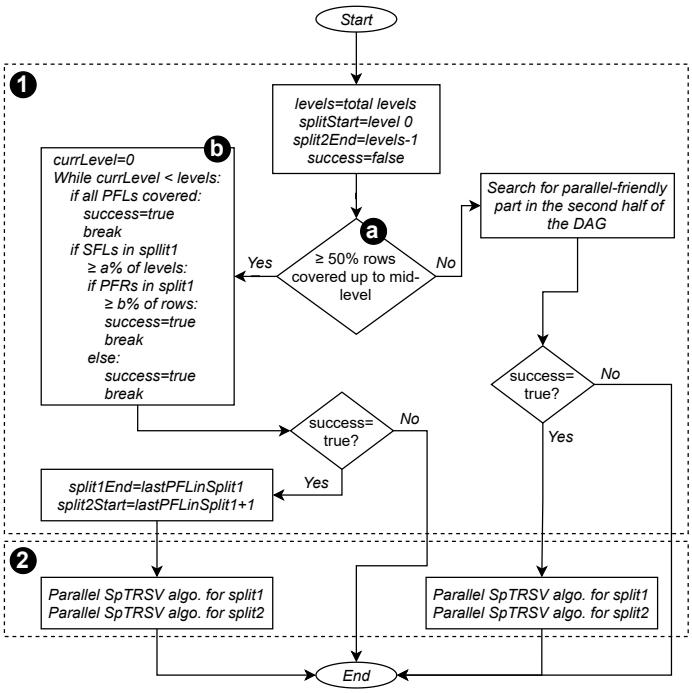

Fig. 8. (1) DAG slicing and (2) algorithm mapping for two-algorithm-split execution (CPU-GPU). *split1Start, split2Start* are start, and *split1End, split2End* are end levels for first and second splits, respectively. *lastPFLinSplit1* refers to the ID of the last parallel-friendly level in split 1.

as the end of the parallel-friendly part (*split1End*) and all the subsequent levels are declared as the serial-friendly part. In the latter cases, an additional check is made to ensure that there are still enough PFRs covered by the parallel-friendly part before labeling the last PFL in the search. In case this check fails, DAG slicing is flagged as unsuccessful. For the sake of simplicity, Figure 8 shows the search algorithm for the parallel-friendly part in the left part of the DAG only.

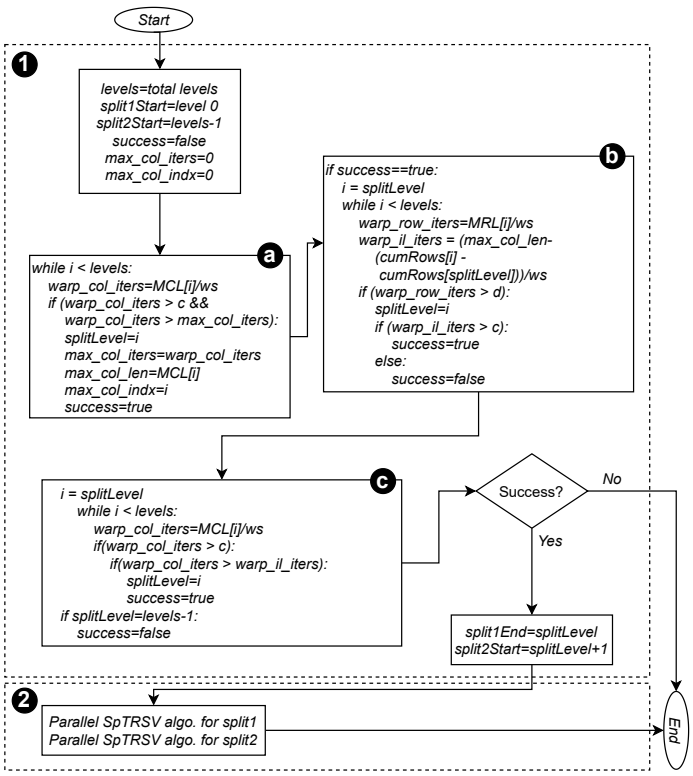

Fig. 9. (1) DAG slicing and (2) algorithm mapping for single-algorithm-split. split1Start, split2Start are start, and split1End, split2End are end levels for the first and second split, respectively. splitLevel refers to the level number for DAG slicing. ***ws*** refers to warp size

The values of the tunable parameters a and b are selected as 10% and 85%, respectively for this study.

Once the DAG slicing succeeds, assigning an algorithm to each part of the split (algorithm mapping) is a straightforward task. Depending on whether the search for the parallel-friendly level took place in the first or second half of the DAG levels, the parallel algorithm is assigned to the first or second part of the split, respectively, while the less parallel algorithm is assigned to the remaining part. Even though other options are possible, the parallel-friendly part is solved using the ELMC algorithm on the GPU while the serial-friendly part is solved using MKL-serial on the CPU.

4.2.3.2 *DAG Slicing for CPU-CPU Execution*: For the cases where the splitting decision algorithm selects the HTS algorithm, an existing Hybrid Triangular Solve (HTS) library is used for two-algorithm-split SpTRSV execution on the CPU. This library handles DAG slicing on behalf of our framework as follows. First, the matrix rows are re-arranged based on the level ordering. Assuming N_i is the number of rows in level i, level i is considered *good* if $N_i \geq n_{good}$, otherwise it is considered as *bad*. The fraction of rows in bad levels is represented by f_{bad}. Now, if C_i is the cumulative sum of rows up to level N_i and C_i^{bad} is the sum of rows in all bad levels up to level i, the DAG slicing algorithm selects the largest good level i as the split level such that $C_i^{bad} \leq f_{bad} C_i$ [8]. Once the DAG is sliced, HTS uses level-scheduling and recursive blocking algorithms for executing SpTRSV 1 and SpTRSV 2, respectively.

4.2.3.3 *DAG Slicing for GPU-GPU Execution*: The idea behind single-algorithm-split execution is to reduce serialization bottlenecks for some parallel algorithms due to large system matrix column or row lengths. The serialized execution occurs when the matrix row or column length is more than the number of threads assigned to that row or column [5], [6], [15], [16]. In such cases, dividing the matrix into three parts can reduce this bottleneck by having smaller triangular systems (I and III in Figure 3(b)) and replacing some operations with SpMV. For the evaluation in this study, we assume parallel SpTRSV algorithms assign a fixed number of threads per column [5], [6], [15], [16], and SpMV with a fixed number of threads per row [19], which is typically the case.

We devise a heuristic-based DAG slicing algorithm for the single-algorithm-split policy that works for GPU-GPU execution as shown in Figure 9. The slicing algorithm aims to select a split point that results in maximum savings in serial iterations required by a warp to process large columns. This is achieved by making the SpMV part contain as many nonzeros of the large columns as possible. The decision is subject to the following criteria: i) The savings in serial iterations should be $\geq c$, ii) Any row with the number of nonzeros requiring warp serial iterations $\geq d$ should be part of SpTRSV 1 to avoid computational bottleneck for SpMV, iii) If there is more than one column satisfying the criterion listed in (i), the split point should be selected targeting the maximum savings in serial iterations. In Figure 9 (a), the algorithm first determines if any column(s) satisfies criterion (i) and in the case of more than one such column, it initially selects a split point resulting in maximum savings in serial iterations. Next, in Figure 9 (b), the split point is adjusted to satisfy criterion (ii). Finally, in Figure 9 (c), further adjustments (if required) are made to the split point so that the criteria (i) and (iii) still hold. When DAG slicing is successful, the same SpTRSV algorithm is used for solving the two sub-triangular systems. The threshold values for the tunable parameters *c* and *d* have been empirically chosen as 100 and 75, respectively.

4.3 Step 3: Data Structure Setup

Once the execution policy is chosen, necessary data structures are set up for the SpTRSV execution. For this purpose, the model creates three sub-matrices corresponding to SpTRSV 1, SpMV, and SpTRSV 2 based on the DAG of the re-arranged matrix (Figure 3(b)) in either CSR or CSC matrix storage format as required by the corresponding algorithm. For instance, if SpTRSV 1 is to be executed using the MKL library, the sub-matrix for SpTRSV 1 is created in CSR format. Similarly, if SpTRSV 2 is to be executed on GPU with a Sync-free algorithm using CSC matrix format, a sub-matrix in CSC format is created. Likewise, for SpMV, a CSR matrix is created on the CPU or GPU depending on whether SpMV is to be performed on the CPU or the GPU. In addition to these sub-matrices, the framework initializes all the data structures required by the selected algorithms for their analysis and execution phases and for inter-platform communication.

4.4 Step 4: Algorithm Analysis

As described in Section 2, SpTRSV algorithms are comprised of two phases, namely analysis (symbolic analysis or numerical) and solve phases. In this work, the analysis phase refers

to symbolic analysis. The investigation of the analysis phase separately as symbolic and numerical is left as future work. Once the data structures are set up, the analysis phases for each of the selected SpTRSV algorithms (SpTRSV 1 and SpTRSV 2) are performed for the corresponding sub-matrices (Step 4 in Figure 4). In the case of a unified execution, the analysis phase for the sub-matrix corresponding to SpTRSV 1 is performed.

4.5 Step 5: SpTRSV Execution

Once the setup phase is completed, the framework is ready to execute SpTRSV using the selected execution policy. The split SpTRSV execution is completed in the following steps:

- SpTRSV 1 is performed using the selected platform and algorithm.
- If SpTRSV 1 and SpMV are scheduled to be executed on different platforms, SpTRSV 1 solution is communicated to the SpMV platform using CUDA memory copy API call. Similarly, if SpTRSV 2 and SpMV are on different platforms, the right-hand-side corresponding to SpTRSV 2 is communicated to the SpMV platform.
- SpMV between the rectangular sparse matrix and SpTRSV 1 solution is performed using MKL (SpMV on CPU) or cuSPARSE SpMV (SpMV on GPU) routines that computes SpMV of the form:

$$b_{2(new)} := \alpha \cdot M \cdot x_1 + \beta \cdot b_{2(old)} \quad (1)$$

  Equation 1 computes SpMV of M and x_1 and projects its solution to the right-hand-side of SpTRSV 2 resulting in its new right-hand-side $b_{2(new)}$. M is the rectangular sparse matrix corresponding to II in Figure 3(b), x_1 is the solution of SpTRSV 1, α and β are -1 and 1 respectively, $b_{2(old)}$ is the existing right-hand-side corresponding to SpTRSV 2 and $b_{2(new)}$ is the updated right-hand-side.
- If SpMV and SpTRSV 2 are on different platforms, the updated right-hand-side $b_{2(new)}$ is communicated to the appropriate platform using CUDA memory copy API call.
- SpTRSV 2 is performed using the selected platform and algorithm.
- Finally, overall SpTRSV results are consolidated on the platform of choice (CPU or GPU) which can be selected by the programmer as part of the execution policy.

4.6 Supported SpTRSV and SpMV Algorithms

The framework supports several SpTRSV and SpMV algorithms for both the CPU and GPU architecture. For the CPU, we support SpTRSV and SpMV implementations using the Intel MKL library [13] in the CSR matrix storage format. In addition, for two-algorithm-split execution on CPU, an SpTRSV implementation using the HTS library [8] is supported. HTS uses a level-scheduling algorithm [9] for SpTRSV 1 and a recursive-blocking algorithm for SpTRSV 2. For GPUs, the framework supports SpTRSV and SpMV algorithms using cuSPARSE(v2) library [20], the Sync-Free SpTRSV algorithm by Liu et al. [5], and ELMC, ELMR SpTRSV algorithms by Li et al. [6]. For the framework evaluation in this study, we chose ELMC over the other supported GPU algorithms for its superior performance as reported in [6]. The framework is extensible with more SpMV and SpTRSV implementations and enriching the collection is left as future work.

4.7 Framework Overheads

There are several overheads associated with the execution model for both the setup and execution phases. The overhead for the setup phase, referred to as *setup overhead* in the subsequent discussion, is as follows:

$$\textit{Setup overhead} = T_{DD} + T_{EPS} + T_{DSS} + T_{AA} \quad (2)$$

where T_{DD}, T_{EPS}, T_{DSS}, and T_{AA} are the time spent in DAG discovery, execution policy setup, data structure setup, and algorithm analysis, respectively. Setup overhead is a one-time overhead incurred before the SpTRSV execution phase begins.

We define execution overhead as the time spent in inter-platform data communication when any of the SpTRSV 1, SpTRSV 2, and SpMV are performed on different platforms during the execution phase and is given by the following equation:

$$\textit{Execution overhead} = C_{SpTRSV1} + C_{SpMV} + C_{SpTRSV2} \quad (3)$$

$C_{SpTRSV1}$, C_{SpMV}, and $C_{SpTRSV2}$ are data communication overheads for SpTRSV 1, SpMV and SpTRSV 2, respectively. $C_{SpTRSV1}$ includes the time to communicate right-hand-side corresponding to SpTRSV 1 to the selected platform. C_{SpMV} accounts for the time to communicate the solution of the SpTRSV 1 and the initial right-hand-side corresponding to SpTRSV 2 to the SpMV platform. Finally, $C_{SpTRSV2}$ corresponds to the time required to communicate the updated right-hand-side by SpMV to the SpTRSV 2 platform. The execution overhead can be represented in terms of CPU-GPU communication bandwidth as follows:

$$\textit{Execution overhead} \leq ((2 \cdot r_1 + 2 \cdot r_2) \cdot FS)/BW_{interconnect} \quad (4)$$

where r_1 is the number of rows in the upper sub-triangular matrix (I in Figure 3), r_2 is the number of rows in the lower sub-triangular matrix (III in Figure 3), $BW_{interconnect}$ is the CPU-GPU bandwidth in bytes/second and FS is 4 or 8 bytes for single-precision or double-precision floating-point computations, respectively. Unlike the setup overhead which is incurred only once, execution overhead is incurred per SpTRSV iteration.

5 Evaluation

In this section, we present the performance of our framework on two CPU-GPU platforms (see Table 1 and 2) using matrices from the SuiteSparse Matrix Collection. Here are the highlights of the findings.

- Section 5.2 presents the speedups achieved by our framework versus the best of the MKL and ELMC

algorithms for a selected set of matrices. The framework has been shown to achieve $\geq$1 speedup for 91% and 86% of the matrices for machine 1 and machine 2, respectively.

- Section 5.3 shows the benefit of the GPU-GPU split execution using the same algorithm against unified execution. GPU-GPU split execution particularly provides the highest speedups for matrices having long rows and columns (e.g., matrices following power-law distribution).
- Section 5.4 evaluates overheads associated with the setup phase of the framework, which have been found to be well within acceptable limits so as to achieve overall performance gain.
- Section 5.5 explores the impact of CPU-GPU data transfers for the cases where split execution requires CPU-GPU communication. The data transfer overhead has been found to be negligible in comparison with the speedups achieved by the framework.
- Finally, Section 5.6 compares the framework performance against MKL, ELMC, and cuSPARSE(v2) for our test set of 327 matrices. The framework achieves ≥ 1 speedup for $> 85\%$ of the matrices versus each of these algorithms on both machines. Moreover, the framework selects the fastest algorithm (split or unsplit) for over 80% of the matrices.

TABLE 1
Specifications of the CPU machines used for the evaluation

	Machine 1	Machine 2
CPU Name	Intel Xeon Gold	Intel Core I7
Model	6148	8700K
Processor Base Frequency	2.4 GHz	3.7 GHz
Number of cores	40	12
Number of sockets	2	1
Cores per socket	20	12
L1 Data cache	32K	32K
L1 Instruction cache	32K	32K
L2 cache	1024K	256K
L3 cache	28160K	12288K
Main memory	512GB	64GB
Operating system	CentOS Linux release 7.4.1708	Ubuntu 18.04.1 (kernel ver 4.15.0-132)

TABLE 2
Specifications of the GPUs used for the evaluation

	Machine 1	Machine 2
GPU Name	NVIDIA Tesla V100	NVIDIA GTX1080 Ti
Number of cores	5120	3584
Main memory	32GB	11GB
Memory bandwidth	900 GB/sec	484 GB/sec
Interconnect bandwidth	32 GB/sec	32 GB/sec

5.1 Experimental Platforms, Dataset and Algorithms

Experimental Platforms: We evaluated the performance of the execution model on two CPU-GPU platforms. The first machine is an Intel Xeon Gold CPU with an NVIDIA Tesla V100 GPU (machine 1). The second machine is an Intel Core i7 CPU with NVIDIA GTX1080 Ti GPU (machine 2). The specifications of these machines are listed in Tables 1 and 2. The host code is compiled using Intel's *icpc* compiler provided as part of Intel Parallel Studio 2019 with `-O3` optimization and is dynamically linked with the sequential version of the Intel MKL library using the `-mkl=sequential` option. For compiling the device code, we use the `nvcc` compiler from CUDA version 10.1 with options `-gencode arch=compute_70,code=sm_70` for V100 and `-gencode arch=compute_61,code=sm_61` for GTX 1080 Ti GPUs, respectively.

Matrix Dataset: As the test set, we use 327 matrices from the SuiteSparse Matrix Collection. For model training and heuristics development (Section 4.2) a set of 657 matrices is selected such that there is no overlap between the test and training sets. For a detailed analysis, a subset of 67 matrices is selected from the test set. The subset contains matrices from diverse application domains with distinct characteristics. The matrices in the test dataset have rows ranging between 1K and 8.3M with a median of 36K, and nnzs ranging between 1.47K and 141.7M with a median of 313K.

Algorithms: For the CPU-GPU split execution case, we use the sequential version of the Intel MKL library for the sequential-friendly part and the state-of-the-art Sync-Free algorithm with level-scheduling, known as element scheduling in the CSC format (ELMC) [6] for the parallel-friendly part of the DAG. If the split execution requires a GPU, then SpMV is always performed on the GPU using cuSPARSE(v2) library [20]. For two-algorithm-split execution on the CPU, we use the HTS library. In the case of unified execution, GPU still employs the ELMC algorithm.

5.2 Speedup against Best CPU/GPU Algorithm

Table 3 shows the breakdown of the matrices achieving speedups $<$1 or $\geq$1 with our framework versus the best of the unified (unsplit) execution of SpTRSV using either ELMC or MKL(seq) algorithms.

Out of the 67 matrices used for the detailed analysis, 43 are selected for split execution with 23 employing CPU-GPU, 10 using CPU-CPU split execution with HTS and 10 utilizing GPU-GPU split execution policy. For the remaining matrices, unified execution policy on the CPU and GPU is selected for 16 and 8 matrices, respectively.

TABLE 3
Distribution of matrices according to achieved framework speedup over the best of ELMC or MKL unsplit execution for the 67-matrix subset S: Speedup. Machine 1: Intel Xeon Gold + NVIDIA Tesla V100. Machine 2: Intel Core I7 + NVIDIA GTX 1080 Ti

	S	Split Policy			Unified Policy		Total
		CPU-GPU	CPU-CPU	GPU-GPU	CPU	GPU	
Machine 1	$<$1	4	0	1	1	0	**6**
	=1	0	0	0	15	8	**23**
	$>$1	19	10	9	0	0	**38**
	Total	**23**	**10**	**10**	**16**	**8**	**67**
Machine 2	$<$1	5	4	0	0	0	**9**
	=1	0	0	0	16	8	**24**
	$>$1	18	6	10	0	0	**34**
	Total	**23**	**10**	**10**	**16**	**8**	**67**

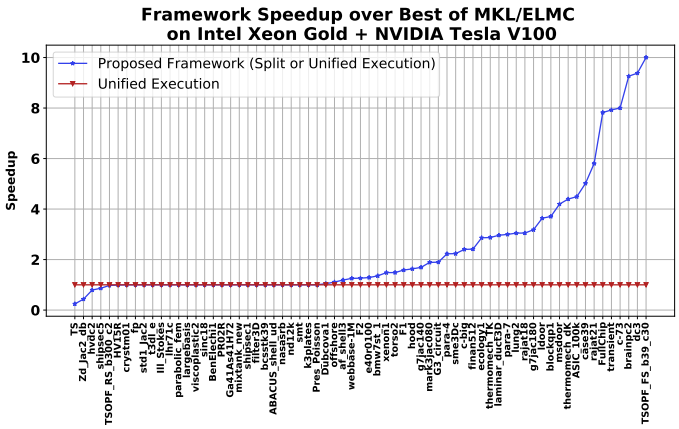

Fig. 10. Performance of proposed framework on Intel Xeon Gold + NVIDIA Tesla V100 machine for the 67-matrix subset. For each matrix, the speedup is calculated over the best of the MKL(seq) or ELMC algorithms (unit speedup line) for solving the lower triangular system.

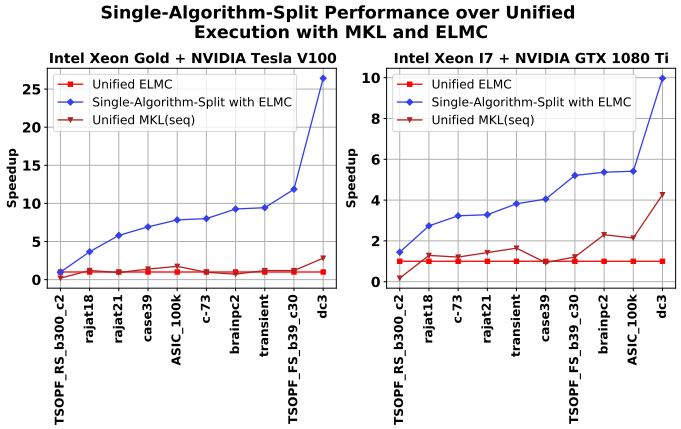

Fig. 11. Performance comparison of single-algorithm-split execution using ELMC against MKL and unified ELMC for the ten matrices (out of the 67-matrix subset)

For machine 1, the framework achieves >1 speedups for a total of 38 (out of 67 matrices) while the best CPU or GPU algorithm for unified execution is correctly selected for 23 matrices. In total, the framework achieves a speedup of ≥ 1 for 61 (91%) matrices used in this analysis. Although the execution policy remains unchanged and the same HTS-MKL and HTS-ELMC models trained for machine 1 are used, on machine 2 the framework achieves equal or better performance for 58 (87%) of the 67 matrices. For 34 of these matrices, the framework achieves a speedup of > 1 while the best unified execution policy (speedup=1) is selected for the rest (24 matrices).

Figure 10 shows the speedups achieved by the framework over the unified SpTRSV execution using the best of ELMC and MKL(seq) algorithms on machine 1. The speedups range up to $10\times$. The top five matrices showing the highest speedups on machine 1 use GPU-GPU split execution policy, showing the immense potential of the GPU-GPU execution policy in overcoming performance bottlenecks for some matrices. The speedups achieved by GPU-GPU split policy range up to $10.0\times$. The CPU-GPU split policy provides speedups up to $7.82\times$ for 19 (out of 23 matrices). The speedup ranges up to $3.70\times$ for CPU-CPU split execution for 10 matrices. Comparable results hold on machine 2, with speedups ranging up to $6.36\times$.

5.3 Performance of Single-Algorithm-Split (SAS)

In this section, we present the performance of single-algorithm-split (SAS) execution of ELMC and compare it against MKL(seq) and unified ELMC for the ten matrices selected for the SAS execution (Table 3) by our execution model. Figure 11 shows the results for the two machines. Except for the one matrix on machine 1 for which performance is nearly the same as the unified ELMC execution, SAS execution performs better than both the unified ELMC and MKL executions for all matrices on both machines.

Not surprisingly, most of the matrices achieving higher performance with SAS belong to the circuit simulation or power network problems as such matrices obey the power-law distribution. For such matrices, long rows and columns are expected to become a performance bottleneck for SpTRSV [10]. Out of these matrices, *TSOPF_RS_b300_c2* and *dc3* have relatively the smallest and largest column lengths, respectively. These two matrices achieve the minimum and maximum speedups with SAS execution over the unified ELMC on both machines.

In addition to the maximum column length, the overall speedups also depend on characteristics of the corresponding matrices such as the distribution of nonzeros, matrix size, and the characteristics of the underlying machine. For instance, among the two matrices (*TSOPF_RS_b300_c2* and *brainpc2*) having the same savings in warp serial iterations, *brainpc2* achieves higher speedup, possibly due to having a higher percentage of nonzeros in the SpMV part. In conclusion, although column length alone does not determine the exact speedup, single-algorithm-split execution can provide significant speedups for matrices with large column lengths for ELMC or similar sync-free implementations.

5.4 Framework Setup Overhead

In this section, we present the framework setup overhead corresponding to Equation 2 in terms of SpTRSV iterations required to amortize the overhead for that matrix. The setup overhead includes the time spent in DAG discovery, DAG slicing, algorithm mapping, data structure setup, and algorithm analysis phase of SpTRSV algorithm(s) used. We calculate the equivalent number of SpTRSV iterations for the framework setup overhead using the equation:

$$SpTRSV\ iterations = \frac{Setup\ overhead\ (msec)}{SpTRSV\ execution\ time\ (msec)} \quad (5)$$

where *Setup overhead* is calculated as per Equation 2 and *SpTRSV execution time* is the time required to complete one SpTRSV iteration for the matrix using the selected execution policy. The SpTRSV execution time is the average of 10 iterations.

Figure 12 plots setup overhead in terms of SpTRSV iterations for both machines for the dataset of 67 matrices. For machine 1, the setup overhead ranges between 10 to 458 SpTRSV iterations with a median of 68 iterations. For

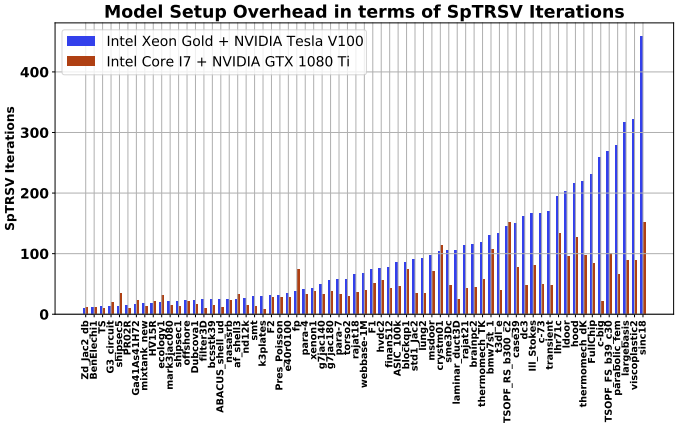

Fig. 12. Number of SpTRSV iterations required to amortize framework setup overhead

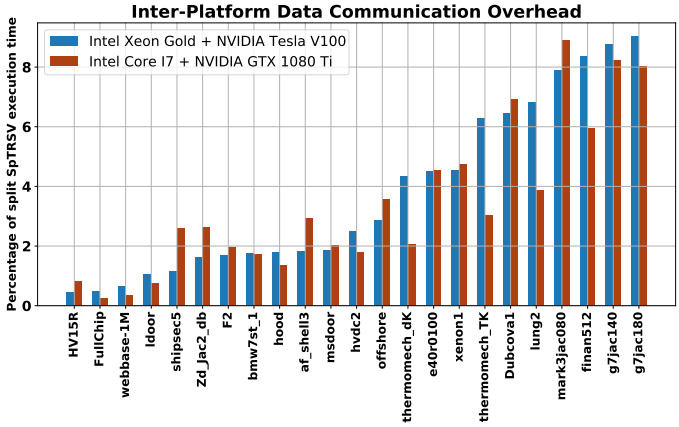

Fig. 13. Inter-platform data transfer overhead for CPU-GPU split cases. The overhead is shown as a percentage of the total split execution time for a single iteration.

machine 2, the range is between 8 to 151 SpTRSV iterations with a median of 37 iterations. In terms of absolute values, the setup overhead takes an average of 167 msec and 112 msec for the dataset on machines 1 and 2, respectively. The maximum setup overhead observed is 3.76 and 2.40 seconds for the matrix *HV15R* on machines 1 and 2, respectively.

For applications such as iterative solvers, SpTRSV is typically executed for 100s of iterations per symbolic analysis phase. Given the reasonable number of SpTRSV iterations to amortize the overhead, the framework can potentially provide significant performance gains for such applications. Moreover, for a given matrix and machine, this is a one-time overhead incurred at the beginning of the execution phase.

5.5 Inter-platform Data Transfer Overhead

The MKL(seq) and ELMC SpTRSV algorithms we have chosen for two-algorithm-split execution operate on two different platforms, i.e., CPU and GPU, respectively. As discussed in Section 4.7, intermediate results need to be communicated between the host and device for such cases. Figure 13 shows the execution overhead for the 23 matrices (Table 3) that use a two-algorithm-split execution policy using MKL-ELMC. This overhead is presented as the percentage of the time required for one SpTRSV iteration for each matrix. Both the machines use PCIe 3.0 interconnect with a theoretical peak bandwidth of 32 GB/s. To reduce this overhead, we allocate corresponding host arrays in pinned memory [21]. For all matrices, the maximum data transfer overhead is less than 10% of the split execution time on both machines. This shows that migrating the execution from one platform to another does not have a significant impact on the overall performance. However, having more than one split point might have an increasing impact on the execution time as each split may require a data transfer.

5.6 Speedup against MKL, ELMC, and cuSPARSE

Figures 14 shows speedup distribution over MKL, ELMC, and cuSPARSE (v2) on machine 1 for the test set of 327 matrices. Figures 14(a) shows the distribution of the chosen execution policy for the test set. Overall, the split execution policy is selected for 38% of the matrices. In the remaining

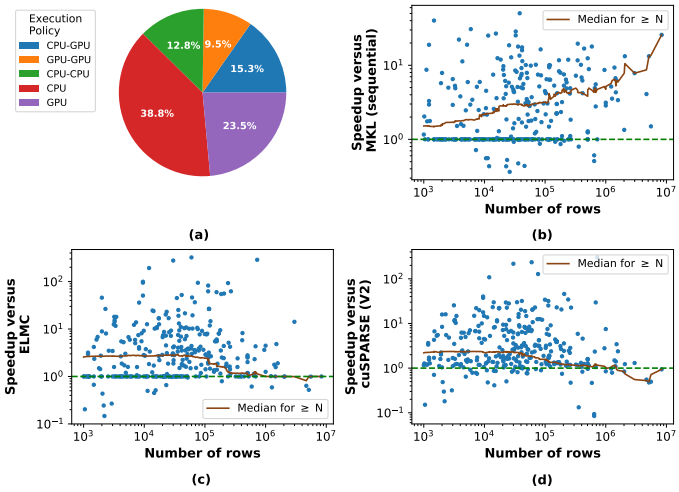

Fig. 14. Framework performance for the test set of 327 matrices on Intel Xeon Gold + NVIDIA Tesla V100 machine. (a) Distribution of execution policy for the test set. Each slice of the pie shows percentage of matrices chosen for the corresponding execution policy. Speedup achieved by the framework over (b) SpTRSV using Intel MKL (sequential) library, (c) SpTSRV using ELMC algorithm, and (d) SpTRSV using cuSPARSE (v2) library. For (b),(c),(d) x-axis is the number of matrix rows, each dot represents a matrix, y-axis represents speedup. At each point N, brown line shows median speedup for matrices with at least N rows.

figures, each plot point corresponds to a matrix showing speedup achieved against the matrix size. The brown line at a point N shows median speedup for matrices with rows $\geq N$. There is an upward trend in the framework's median speedup against MKL with increasing matrix size. On the other hand, the median speedup is around $2\times$ over ELMC and cuSPARSE (v2) at first, stays above $1\times$ for most of the matrices, finally converging to $1\times$ as the matrices get larger and larger. Comparable results are observed on machine 2, thus omitted for brevity.

Table 4 shows framework speedup breakdown versus MKL, ELMC, and cuSPARSE (v2). For machine 1, relative speedups achieved by the framework over MKL, ELMC, and cuSPARSE(v2) are ≥ 1 for over 94%, 93% and 86% of the matrices, respectively. For machine 2, these numbers are 91%, 90%, and 89%, respectively. As regards the selection of

TABLE 4
Distribution of 327 matrices based on achieved speedup over MKL, ELMC and cuSPARSE(v2). S: Speedup. Machine 1: Intel Xeon Gold + NVIDIA Tesla V100. Machine 2: Intel Core I7 + NVIDIA GTX 1080 Ti

	S	MKL	ELMC	cuSPARSE (v2)
Machine 1	<1	18	22	44
	=1	127	77	0
	>1	182	228	283
Machine 2	<1	27	32	34
	=1	127	77	0
	>1	173	218	293

the SpTRSV execution policy, the algorithm correctly selects the fastest algorithm (split or unsplit) for 88% and 83% of the matrices for machine 1 and machine 2, respectively.

Overall, our split execution framework has shown significant SpTRSV performance gains by using split execution policy, as well as by correctly selecting the best CPU or GPU algorithm for unified execution when split execution is found unsuitable for a given matrix.

6 Related Work

Solving scientific computing problems using multiple execution platforms and algorithms has been extensively studied in the literature. In [22], authors devise a CPU-GPU hybrid approach to perform LU factorization equivalent to *dgetrf* function in LAPACK library. Their solution works by representing the computations as a DAG and scheduling fine-grained tasks on the CPU and coarse-grained tasks on the GPU. In [23], empirical auto-tuning techniques are used to distribute and load balance the computations for dense matrix-matrix multiplication and LU factorization between CPUs and GPUs with satisfactory results. In [24], the authors devise a static scheduling mechanism to divide matrix computations between a CPU and a GPU for landform attributes representation, a mathematical modeling approach to efficiently represent geophysical resources. Agullo et al. [25] accelerate QR factorization on a CPU-GPU machine by expressing QR factorization as a set of tasks. Benner et al. [26] use a sequential version of code with a multithreaded version of BLAS to accelerate the computation of matrix sign function on a CPU-GPU system. Muramatsu et al. [27] use a hybrid CPU-GPU approach to accelerate Hessenberg reduction by running small-size BLAS operations entirely on CPU and distribute large-size BLAS operations between CPU and GPU to achieve speedup over CPU-only execution.

A considerable body of work exists that involves scheduling the subtasks between CPU and GPU based on the nature of the subtasks [28], [29], [30], [31], [32], [33], [34], [35], [36] such as parallelism characteristics. Like our two-algorithm-split execution policy for CPU-GPU, subtasks divided based on parallelism characteristics are mapped such that highly parallel subtasks run on the GPU while the sequential subtasks are mapped to the CPU.

Several works dealing with CPU-GPU hybrid computations of sparse matrix-vector multiplication (SpMV) are available in literature. In [37], the authors use design patterns to distribute SpMV computations between CPUs and GPUs. In [38], matrix rows are rearranged based on the probability mass function of the nonzeros in a row before distributing the rows between CPU and GPU to achieve higher SpMV performance than partitioning based on the original row order. In [39], SpMV is performed by dividing the input matrix into multiple blocks row-wise and assigning the best suited sparse storage format using a machine learning-based prediction model. Then, a mapping algorithm is used to assign different blocks to CPUs or GPUs. In [40], authors analyze CPU and GPU characteristics to devise a distribution function to partition the sparse matrix between CPU and GPU for SpMV calculation with noticeable performance improvement. A work by Matam et al. [41] utilizes a heuristic-based approach for work division of sparse matrix-matrix computations between CPU and GPU. They achieve $6\times$ speedup over an optimized Intel MKL library implementation of SPGEMM. In [42], the authors propose SELL-C-α, a unified matrix storage format, for the performance portability of SpMV on heterogeneous computing systems. A matrix splitting method based on the theoretical analysis of its nonzero distribution for SpMV is presented by Anzt et al. [43]. Based on the analysis, the matrix is partitioned into two parts and stored in two different formats (ELLpack and coordinate formats) for better load balancing and computational efficiency.

For the solution of a triangular system on CPU-GPU systems, a recursive algorithm for dense matrices is proposed in [44]. The algorithm works by recursively splitting the triangular solve into parts and assigning these parts to the CPUs and the GPUs with the aim of load balancing by taking the computational requirement of each part and processor speeds into account. The load balancing is performed based on the observation that for dense triangular solves, the performance of CPU and GPU is proportional to the size of the input matrix. A polynomial regression model is used for matrix size versus processor performance estimation. This partitioning principle, however, is not directly applicable to sparse matrices as the CPU or GPU performance is dictated by the input sparsity pattern, not the matrix size, thus it poses additional challenges compared to the dense cases. Charara et al. [45] use a split execution strategy for dense triangular systems on manycore CPUs and GPUs. Their method works by first splitting the matrix into two smaller triangular matrices and a rectangular matrix, and then recursively splitting the smaller triangular matrices in the same fashion. At the end of the recursion, the native triangular solvers are called. They report significant speedups for their implementation relative to the existing BLAS libraries (e.g., cuBLAS, MAGMA). Unlike our work where we determine split points based on matrix characteristics, they split the matrix of size M at a row that is equal to the next power of $2 \geq M/2$ and $< M$.

To the best of our knowledge, the closest work to ours is presented in [8]. The author proposes SpTRSV split execution on CPU in which SpTRSV 1 and SpTRSV 2 are always solved using a level-scheduling [9] and a recursive blocking algorithm, respectively. Their approach works based on the observation that level-scheduling induced re-ordering of the matrix densifies the rows towards the lower end of the triangular matrix and pushes the dense columns towards the right. As a result, using a level-scheduling algorithm on the upper sub-triangular part and a recursive blocking algorithm on the lower triangular part results in improved

performance. In contrast, our approach analyzes matrix sparsity pattern and variation in parallelism to divide the workload between serial and parallel-execution friendly algorithms. Also, our design approach allows new algorithms to be easily incorporated into the model, targets both the CPU and GPU platforms and shows how single-algorithm split execution can improve SpTRSV performance.

Recent work by Lu et al. [10] uses recursive blocking for solving SpTRSV on the GPUs. Like the approach in [45] for dense matrices, they recursively divide the triangular matrix into two smaller triangular and a rectangular part. When the recursion depth is reached, adaptively selected SpMV and SpTRSV kernels are used to solve the system. The kernel selection is based on features such as the average number of nonzeros per row and the number of levels. Unlike our work in which we may decide not to use split execution, they always perform the matrix partitioning, and the split point is decided based on the recursion depth instead of matrix characteristics.

7 Conclusion

We devise an SpTRSV split execution model that can automatically divide and execute SpTRSV computation as two sub-SpTRSV systems, one suitable for less parallel while the other is suitable for highly parallel execution. Thus, the framework may utilize two algorithms with contrasting computational characteristics to solve an SpTRSV system, where those algorithms may run on different platforms or on the same platform. Using statistical analysis of SpTRSV performance data for a parallel-friendly, sequential friendly, and an existing split execution algorithm, the framework can automatically determine the suitability of a sparse triangular system for split-execution and automatically determine the split point at which to partition the SpTRSV computation. A C++/CUDA library implements these heuristics together with multiple CPU and GPU SpTRSV algorithms to perform split SpTRSV execution.

Experimental evaluation of the framework on two CPU-GPU platforms using a diverse set of 327 matrices from the SuiteSparse Matrix Collection, shows that the framework achieves speedups up to $10\times$ on Intel Gold CPU + NVIDIA Tesla V100 GPU machine and up to $6.36\times$ on Intel Core I7 CPU + NVIDIA G1080 Ti GPU machine over the fastest SpTRSV algorithm.

Acknowledgments

The authors would like to thank Aramco Overseas Company and Saudi Aramco for funding this research.

References

[1] H. Wang, W. Liu, K. Hou, and W.-c. Feng, "Parallel transposition of sparse data structures," in *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, (New York, NY, USA), Association for Computing Machinery, 2016.

[2] A. Jamal, M. Baboulin, A. Khabou, and M. Sosonkina, "A hybrid CPU/GPU approach for the parallel algebraic recursive multilevel solver pARMS," in *2016 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pp. 411–416, 2016.

[3] R. Li and Y. Saad, "GPU-accelerated preconditioned iterative linear solvers," *J. Supercomput.*, vol. 63, p. 443466, Feb. 2013.

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

[5] W. Liu, A. Li, J. D. Hogg, I. S. Duff, and B. Vinter, "Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, p. e4244, 2017. e4244 cpe.4244.

[6] R. Li and C. Zhang, "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.

[7] S. Mittal and J. S. Vetter, "A survey of CPU-GPU heterogeneous computing techniques," *ACM Comput. Surv.*, vol. 47, July 2015.

[8] A. M. Bradley, "A hybrid multithreaded direct sparse triangular solver.," *SIAM Workshop on Combinatorial Scientific Computing*, Oct. 2016.

[9] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey, "Sparsifying synchronization for high-performance shared-memory sparse triangular solver," in *Supercomputing* (J. M. Kunkel, T. Ludwig, and H. W. Meuer, eds.), (Cham), pp. 124–140, Springer International Publishing, 2014.

[10] Z. Lu, Y. Niu, and W. Liu, "Efficient block algorithms for parallel sparse triangular solve," in *49th International Conference on Parallel Processing - ICPP*, ICPP '20, (New York, NY, USA), Association for Computing Machinery, 2020.

[11] E. Anderson and Y. Saad, "Solving sparse triangular linear systems on parallel computers," *Int. J. High Speed Comput.*, vol. 1, p. 7395, Apr. 1989.

[12] A. George, M. Heath, J. Liu, and E. Ng, "Solution of sparse positive definite systems on a shared-memory multiprocessor," *Int. J. Parallel Program.*, vol. 15, p. 309325, Oct. 1986.

[13] Intel Incorporated, "Intel MKL — Intel Software." https://software.intel.com/content/www/us/en/develop/articles/intel-math-kernel-library-documentation.html, 2017.

[14] B. Yılmaz, B. Sipahioğlu, N. Ahmad, and D. Unat, "Adaptive Level Binning: A new algorithm for solving sparse triangular systems," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPCAsia2020, (New York, NY, USA), p. 188198, Association for Computing Machinery, 2020.

[15] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter, "A synchronization-free algorithm for parallel sparse triangular solves," in *Euro-Par 2016: Parallel Processing* (P.-F. Dutot and D. Trystram, eds.), (Cham), pp. 617–630, Springer International Publishing, 2016.

[16] R. Li, "On parallel solution of sparse triangular linear systems in CUDA," *CoRR, abs/1710.04985*, 2017.

[17] N. Ahmad, B. Yilmaz, and D. Unat, "A prediction framework for fast sparse triangular solves," in *Euro-Par 2020: Parallel Processing* (M. Malawski and K. Rzadca, eds.), (Cham), pp. 529–545, Springer International Publishing, 2020.

[18] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, Dec. 2011.

[19] N. Bell and M. Garl, "Efficient sparse matrix-vector multiplication on CUDA," in *NVIDIA Technical Report NVR-2008-004,*, Dec. 2008.

[20] N. Corporation, "NVIDIA cuSPARSE library." https://docs.nvidia.com/cuda/cusparse/index.html, 2021.

[21] N. Corporation, "CUDA C++ best practices guide." https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, 2021.

[22] J. Kurzak, P. Luszczek, M. Faverge, and J. J. Dongarra, "LU factorization with partial pivoting for a multi-CPU, multi-GPU shared memory system," in *VECPAR 2012 - 10th International Meeting on High-Performance Computing for Computational Science*, (Kobe, Japan), July 2012.

[23] G. Bernab, J. Cuenca, L.-P. Garca, and D. Gimnez, "Tuning basic linear algebra routines for hybrid CPU+GPU platforms," *Procedia Computer Science*, vol. 29, pp. 30 – 39, 2014. 2014 International Conference on Computational Science.

[24] M. Boratto, P. Alonso, C. Ramiro, and M. Barreto, "Heterogeneous computational model for landform attributes representation on multicore and multi-GPU systems," *Procedia Computer Science*, vol. 9, pp. 47 – 56, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012.

[25] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov, "QR factorization on a multicore node enhanced with multiple GPU accelerators," in *2011 IEEE International Parallel Distributed Processing Symposium*, pp. 932–943, 2011.

[26] P. Benner, P. Ezzatti, E. S. Quintana-Ortí, and A. Remón, "Using hybrid CPU-GPU platforms to accelerate the computation of the matrix sign function," in *Euro-Par 2009 – Parallel Processing Workshops* (H.-X. Lin, M. Alexander, M. Forsell, A. Knüpfer, R. Prodan, L. Sousa, and A. Streit, eds.), (Berlin, Heidelberg), pp. 132–139, Springer Berlin Heidelberg, 2010.

[27] J. ichi Muramatsu, T. Fukaya, S.-L. Zhang, K. Kimura, and Y. Yamamoto, "Acceleration of Hessenberg reduction for nonsymmetric eigenvalue problems in a hybrid CPU-GPU computing environment," *International Journal of Networking and Computing*, vol. 1, no. 2, pp. 132–143, 2011.

[28] Y. Liu, A. Fedorov, R. Kikinis, and N. Chrisochoides, "Real-time non-rigid registration of medical images on a cooperative parallel architecture," in *2009 IEEE International Conference on Bioinformatics and Biomedicine*, pp. 401–404, 2009.

[29] P. Yao, H. An, M. Xu, G. Liu, X. Li, Y. Wang, and W. Han, "CuHMMer: A load-balanced CPU-GPU cooperative bioinformatics application," in *2010 International Conference on High Performance Computing Simulation*, pp. 24–30, 2010.

[30] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard, "Multi-GPU and multi-CPU parallelization for interactive physics simulations," in *Euro-Par 2010 - Parallel Processing* (P. D'Ambra, M. Guarracino, and D. Talia, eds.), (Berlin, Heidelberg), pp. 235–246, Springer Berlin Heidelberg, 2010.

[31] Q. Hu, N. A. Gumerov, and R. Duraiswami, "Scalable fast multipole methods on distributed heterogeneous architectures," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2011.

[32] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, pp. 78–88, 2011.

[33] P. Stpiczyski, "Solving linear recurrences on hybrid GPU accelerated manycore systems," in *2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pp. 465–470, 2011.

[34] P. C. Gao, Y. B. Tao, Z. H. Bai, and H. Lin, "Mapping the SBR and TW-ILDCs to heterogeneous CPU-GPU architecture for fast computation of electromagnetic scattering," *Progress In Electromagnetics Research*, vol. 122, pp. 137–154, 2012.

[35] Y. Wang, H. Du, M. Xia, L. Ren, M. Xu, T. Xie, G. Gong, N. Xu, H. Yang, and Y. He, "A hybrid CPU-GPU accelerated framework for fast mapping of high-resolution human brain connectome," *PLOS ONE*, vol. 8, pp. 1–14, 05 2013.

[36] C. Yang, W. Xue, H. Fu, L. Gan, L. Li, Y. Xu, Y. Lu, J. Sun, G. Yang, and W. Zheng, "A peta-scalable CPU-GPU algorithm for global atmospheric simulations," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, (New York, NY, USA), p. 112, Association for Computing Machinery, 2013.

[37] V. Cardellini, A. Fanfarillo, and S. Filippone, "Heterogeneous sparse matrix computations on hybrid GPU/CPU platforms.," in *Euro-Par 2016: Parallel Processing*, pp. 203–212, 2013.

[38] W. Yang, K. Li, Z. Mo, and K. Li, "Performance optimization using partitioned SpMV on GPUs and multicore CPUs," *IEEE Transactions on Computers*, vol. 64, no. 9, pp. 2623–2636, 2015.

[39] A. Benatia, W. Ji, Y. Wang, and F. Shi, "Sparse matrix partitioning for optimizing SpMV on CPU-GPU heterogeneous platforms," *The International Journal of High Performance Computing Applications*, vol. 34, no. 1, pp. 66–80, 2020.

[40] W. Yang, K. Li, and K. Li, "A hybrid computing method of SpMV on CPUGPU heterogeneous computing systems," *Journal of Parallel and Distributed Computing*, vol. 104, pp. 49 – 60, 2017.

[41] K. Matam, S. Bharadwaj, and K. Kothapalli, "Sparse matrix matrix multiplication on hybrid CPU+GPU platforms," *Proceedings of the High Performance Computing Conference (HiPC12)*, 2012.

[42] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for modern processors with wide SIMD units," *CoRR*, vol. abs/1307.6209, 2013.

[43] H. Anzt, T. Cojean, C. Yen-Chen, J. Dongarra, G. Flegar, P. Nayak, S. Tomov, Y. M. Tsai, and W. Wang, "Load-balancing sparse matrix vector product kernels on GPUs," *ACM Trans. Parallel Comput.*, vol. 7, Mar. 2020.

[44] R. Mahfoudhi, S. Achour, and Z. Mahjoub, "Parallel triangular matrix system solving on CPU-GPU system," in *2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)*, pp. 1–6, 2016.

[45] A. Charara, D. Keyes, and H. Ltaief, "A framework for dense triangular matrix kernels on various manycore architectures," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 15, 2017.

Najeeb Ahmad received his Ph.D. in Computer Science and Engineering at Koç University, Turkey in 2021. He received his BSc. in Electrical Engineering from University of Engineering and Technology, Peshawar, Pakistan (2002) and MEng degree in Signals and Information Processing from Beihang University, Beijing, China (2012). He has over 10 years industrial experience working mostly on embedded systems design and software development. His research interests include high performance computing, application optimization for scientific computing and machine learning. He is currently working on design and development of tools for optimizing sparse computations on heterogeneous computing platforms.

Buse Yilmaz obtained her Bachelors and MSc. degrees from Yeditepe University, Computer Engineering Department in 2009 and 2011. She obtained her PhD degree from the Department of Computer Science, Ozyegin University, in 2016, later where she worked as an instructor. Dr. Yimaz was a Postdoctoral Research Associate at Virginia Tech between 2016 and 2018 and at Koç University between 2018 and 2020. Since September 2020, she's been an assistant professor at the Department of Computer Science, Istinye University. Her research interests include high performance computing, compilers, and machine learning. Currently, she is working on designing and optimizing sparse solvers on heterogeneous computing platforms using machine learning and specialized code generation.

Didem Unat is a Professor at Koç University, Istanbul, Turkey. She is known for her work in designing programming models, performance tools, and runtime systems for parallel architectures. She received her Ph.D. at the University of California San Diego (UCSD) and later received the prestigious and highly competitive Luis Alvarez Postdoctoral Fellowship from the Lawrence Berkeley National Laboratory (LBNL). Since 2014 she leads her own Parallel and Multicore Computing group at Koç University. She is the recipient of the Marie Sklodowska-Curie Individual Fellowship from the European Commission in 2015 and the Young Scientists Award in 2019 from the Science Academy of Turkey. She is the very first researcher from Turkey, who has received the ERC grant in Computer Science from the European Commission.