Platform and Data-Aware Execution of Sparse Triangular Solve on CPU-GPU Heterogeneous Systems

by

Najeeb Ahmad

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Doctor of Philosophy

in

Computer Science and Engineering

KOÇ ÜNİVERSİTESİ

April 19, 2021

Platform and Data-Aware Execution of Sparse Triangular Solve on CPU-GPU Heterogeneous Systems

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a doctoral dissertation by

Najeeb Ahmad

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____

Assist. Prof. Dr. Didem Unat (Advisor)

_____

Prof. Dr. Attila Gürsoy

_____

Assist. Prof. Dr. Ismail Arı

_____

Prof. Dr. Öznur Özkasap

_____

Assist. Prof. Dr. Kamer Kaya

Date: _____

To my parents, wife Amna, and children Mahnoor and Saad

ABSTRACT

Platform and Data-Aware Execution of Sparse Triangular Solve on CPU-GPU Heterogeneous Systems
Najeeb Ahmad
Doctor of Philosophy in Computer Science and Engineering
April 19, 2021

Sparse triangular solve (SpTRSV) is an important computational kernel used in many scientific and numerical linear algebra applications such as direct methods, iterative solvers and least square problems. In comparison with other sparse kernels such as sparse matrix-vector multiplication (SpMV), SpTRSV is an inherently sequential operation owing to the presence of dependencies among computations of different unknowns. Often, it has also been observed to be one of the most time consuming operations in an application. The performance of a given algorithm highly depends upon the sparsity characteristics of the input matrix and the underlying hardware. Several SpTRSV algorithms and their implementations are available for the CPUs and the GPUs. Unfortunately, there is no single algorithm or hardware platform that has been shown to achieve the best performance for all input matrices.

In this dissertation, we propose tools and techniques aimed at extracting higher SpTRSV performance for a given input matrix on modern CPU-GPU heterogeneous systems. Towards this end, we adopt a two-pronged approach: Depending upon the sparsity characteristics of the input matrix, we propose to (i) automatically select the best CPU or GPU SpTRSV algorithm for the matrix, (ii) split a single SpTRSV execution into parallel and sequential parts such that the parallel part is executed with a parallel-friendly algorithm and the sequential part is executed with a sequential-friendly algorithm. The aim is to achieve a higher SpTRSV performance than using a single algorithm. The two algorithms can also potentially execute on two different platforms (CPU and GPU).

For the SpTRSV algorithm selection, we propose a supervised machine learning-based prediction framework to predict the SpTRSV implementation giving the fastest execution time for a given sparse matrix based on its structural features.

The framework works by extracting matrix features, collecting algorithm performance data, and training a prediction model with around 1000 real square matrices from the SuiteSparse Matrix Collection. Once trained on a given machine, the model can predict the fastest SpTRSV implementation for a given matrix by paying a one-time matrix feature extraction cost. The framework is also capable of taking into account CPU-GPU communication overheads that might be incurred in scientific applications such as iterative solvers. We test the framework on a modern CPU-GPU machine. Experimental results on a modern CPU-GPU platform (Intel Xeon Gold + NVIDIA Tesla V100 GPU) show that the fastest algorithm is selected with a reasonable accuracy (87%) as well as the predicted SpTRSV implementation achieves significant speedups compared (1.4-2.7× harmonic mean) with a lazy choice of a single algorithm.

Secondly, for dividing the SpTRSV execution between the CPU and GPU, we propose an SpTRSV split execution model. The model has been designed based on the empirical evidence from the existing research that; (i) a highly parallel algorithms perform well for SpTRSV requiring few sequential steps, with high number of unknowns computed per step, (ii) a sequential algorithms perform better for SpTRSV requiring more steps with few unknowns computed per step. For matrices having a mix of highly parallel and sequential steps, a parallel algorithm is expected to provide benefits for steps with large number of unknowns, its performance is expected to deteriorate for steps with few unknowns. The converse can be said about the performance of sequential algorithms for such matrices. With the aim of performing split-execution for such matrices using a highly parallel and a sequential algorithm, our split execution model can automatically determine the suitability of an SpTRSV for split-execution, find the appropriate split point, and execute SpTRSV in a split fashion using two SpTRSV algorithms while automatically managing any required inter-platform communication. The model is implemented as a C++/CUDA library supporting multiple CPU-GPU algorithms. Experimental evaluation of the model on two CPU-GPU 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 the matrices on Intel Xeon Gold + NVIDIA Tesla V100 and 83% of the matrices on Intel Core I7 + NVIDIA G1080 Ti platform achieving speedups up to 10× and 6.36×, respectively.

We expect that the tools and methods proposed in this dissertation will benefit

both academia and industry while at the same time will pave the way for further research in the area of efficient exploitation of CPU-GPU systems for sparse linear algebra computations.

ÖZETÇE

Doktora Tez Başlığı
Najeeb Ahmad
Bilgisayar Bilimleri ve Mühendisliği, Doktora
19 Nisan 2021

Seyrek üçgen çözüm (SpTRSV), doğrudan yöntemler, yinelemeli çözücüler ve en küçük kare problemleri gibi birçok bilimsel ve sayısal doğrusal cebir uygulamalarında kullanılan önemli bir hesaplama çekirdeğidir. Seyrek matris vektör çarpımı (SpMV) gibi diğer seyrek çekirdeklerle karşılaştırıldığında, SpTRSV doğası gereği farklı bilinmeyenlerin hesaplamaları arasındaki bağımlılıkların varlığı nedeniyle çoğu zaman, bir uygulamada en çok zaman alan işlemlerden biri olduğu da gözlemlenmiştir. CPU'lar ve GPU'lar için çeşitli SpTRSV algoritmaları ve uygulamaları mevcuttur. Verilenin performansı algoritması, giriş matrisinin seyreklik özelliklerine ve temeldeki donanıma büyük ölçüde bağlıdır. Ne yazık ki, tüm girdi matrisleri için en iyi performansı elde ettiği gösterilen tek bir algoritma veya donanım platformu yoktur.

Bu tezde, modern CPU-GPU heterojen sistemlerinde belirli bir giriş matrisi için daha yüksek SpTRSV performansı elde etmeyi amaçlayan araçlar ve teknikler öneriyoruz. Bu amaçla, iki yönlü bir yaklaşım benimsiyoruz: Bir girdi matrisinin seyreklik özelliklerine bağlı olarak, (i) matris için en iyi CPU veya GPU SpTRSV algoritmasını otomatik olarak seçmek, (ii) paralel parça paralel uyumlu bir algoritma ile yürütülecek ve sıralı parça sıralı dostu bir algoritma ile yürütülecek şekilde SpTRSV'yi bölmek için yöntemler ve araçlar öneriyoruz. Amaç, tek bir algoritma kullanmaktan daha yüksek bir SpTRSV performansı elde etmektir. İki algoritma potansiyel olarak iki farklı platformda (CPU ve GPU) çalışabilir.

SpTRSV algoritma seçimi için, yapısal özelliklerine dayalı olarak belirli bir seyrek matris için en hızlı yürütme süresini veren SpTRSV uygulamasını tahmin etmek için denetimli makine öğrenimi tabanlı bir tahmin çerçevesi öneriyoruz. Çerçeve, matris özelliklerini çıkararak, algoritma performans verilerini toplayarak ve SuiteSparse Matrix Koleksiyonundan yaklaşık 1000 gerçek, kare matrisli bir tahmin modeli

eğiterek çalışır. Model, belirli bir makine üzerinde eğitildikten sonra, belirli bir matris için en hızlı SpTRSV uygulamasını tahmin eder. Çerçeve, yinelemeli çözücüler gibi bilimsel uygulamalarda ortaya çıkabilecek CPU-GPU iletişim ek yüklerini de hesaba katabilir. Çerçeveyi modern bir CPU-GPU makinesinde test ediyoruz. Modern bir CPU-GPU platformunda deneysel sonuçlar (Intel Xeon Gold + NVIDIA Tesla V100 GPU), en hızlı algoritmanın makul bir doğrulukla (% 87) seçildiğini ve tahmin edilen SpTRSV uygulamasının, tek bir algoritmanın tembel bir seçimiyle karşılaştırıldığında (1.4-2.7 × harmonik ortalama) önemli hız artışlarına ulaştığını göstermektedir.

İkinci olarak, SpTRSV yürütmesini CPU ve GPU arasında bölmek için bir SpTRSV bölünmüş yürütme modeli öneriyoruz. Model, mevcut araştırmalardan elde edilen ampirik kanıtlara dayanılarak tasarlanmıştır; (i) SpTRSV için yüksek oranda paralel algoritmalar, adım başına hesaplanan çok sayıda bilinmeyenle birkaç ardışık adım gerektiren, (ii) adım başına hesaplanan birkaç bilinmeyenle daha fazla adım gerektiren sıralı bir algoritma SpTRSV için daha iyi performans gösterir. Oldukça paralel ve sıralı adımların bir karışımına sahip olan matrisler için, paralel bir algoritmanın çok sayıda bilinmeyenli adımlar için fayda sağlaması beklenir, performansının birkaç bilinmeyenli adımlar için kötüleşmesi beklenir. Sohbet hakkında söylenebilir bu tür matrisler için sıralı algoritmaların performansı. Oldukça paralel ve sıralı bir algoritma kullanarak bu tür matrisler için bölünmüş yürütme gerçekleştirmek amacıyla, bölünmüş yürütme modelimiz bir SpTRSV'nin uygunluğunu otomatik olarak belirleyebilir. Bölünmüş yürütme için, uygun bölme noktasını bulur ve gerekli herhangi bir platformlar arası iletişimi otomatik olarak yönetirken iki SpTRSV algoritmasını kullanarak SpTRSV'yi bölünmüş bir şekilde yürütür. Model, birden çok CPU-GPU algoritmasını destekleyen C ++ / CUDA kitaplığı olarak uygulanmaktadır. Modelin deneysel değerlendirmesi SuiteSparse Matrix Collection'dan 327 matrislik bir matris veri kümesine sahip iki CPU-GPU'da (Intel Xeon Gold + NVIDIA Tesla V100 ve Intel Core I7 + NVIDIA G1080 Ti) matrislerin% 88'i ve% 83'ü için en hızlı SpTRSV yöntemini doğru şekilde seçtiğini göstermektedir. Bu platformlarda sırasıyla 10 × ve 6.36 × aralığında hızlanma sağlar.

Bu tezde önerilen araçların ve yöntemlerin hem akademiye hem de endüstriye fayda sağlayacağını ve aynı zamanda seyrek doğrusal cebir hesaplamaları için CPU-GPU sistemlerinin verimli kullanımı alanında daha fazla araştırmanın yolunu açacağını umuyoruz.

ACKNOWLEDGMENTS

First, I would like to extend my sincerest and heartiest gratitude to my mentor and advisor *Assist. Prof. Didem Unat*. I could not have wished for a better mentor or advisor for my Ph.D than her. Not only did she support me with her knowledge, wisdom and patience but also helped me evolve as a researcher by allowing me to work in my own way. Without her constant guidance, support and encouragement, this work could not have been proposed or completed.

Special thanks to my thesis progress committee members, *Prof. Attila Gürsoy* from Koç University and *Assist. Prof. Ismail Arı* from Özyeğin University for their critical comments and suggestions to improve this work. I would also like to thank *Prof. Öznur Özkasap* from Koç University and *Assist. Prof. Kamer Kaya* from Sabancı University for being part of my thesis jury committee.

I am thankful to *Larry S. Fung*, *Ali H. Doğru* and *Abdulrahman Al-Mana* from ARAMCO for their technical discussions and suggestions for the proposition and completion of this work.

I have been very lucky to have some of the finest research colleagues at ParCoreLab from whom I have learnt a lot during this endeavor. I would like to thank them for their thought-provoking discussions, knowledge sharing, and for the fun moments we had together.

I am also grateful to *Mr. Ufuk Yılmaz* from Koç University Advanced Computing Center for timely provision of the HPC resources. Thanks to *Xing Cai*, *Tore H. Larsen* and *James D. Trotter* from Simula Research Laboratory, Norway for access and support with eX3 HPC facility.

And last but not the least, I would like to express my deepest gratitude to my family, especially my wife and children for their patience and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

AP	Average Parallelism
ARL	Average Row Length
API	Application Programming Interface
ATLAS	Automatically Tuned Linear Algebra Software
BLAS	Basic Linear Algebra Subprograms
CPU	Central Processing Unit
CSC	Compressed Sparse Column sparse matrix storage format
CSR	Compressed Sparse Row sparse matrix storage format
CUDA	Compute Unified Device Architecture
cuBLAS	BLAS library for NVIDIA GPUs
DAG	Directed-Acyclic-Graph
DoP	Degree of Parallelism
DGEMM	Double-Precision General Matrix-Matrix Multiplication
ELLPack	ELLPack matrix storage format
ELMC	Element scheduling in CSC format SpTRSV algorithm
ELMR	Element scheduling in CSR format SpTRSV algorithm
GEMM	General Matrix-Matrix Multiplication
GMRES	Generalized Minimal Residual method
GPU	Graphics Processing Unit
HPL	High Performance Computing Linpack Benchmark
HTS	Hybrid Triangular Solve
ID	Identity number
ILU	Incomplete LU
ILUT	Incomplete LU with thresholding
L1	Level 1

L2	Level 2
L3	Level 3
LAPACK	Linear Algebra Package
LR	Logistic Regression model
MAGMA	Matrix Algebra on GPU and Multicore Architectures
MCL	Maximum Column Length
MKL	Intel Math Kernel Library
MRL	Maximum Row Length
NL	Number of Levels
OSKI	Optimized Sparse Kernel Interface
PCIe	Peripheral Component Interconnect Express
PETSc	Parallel Extensible Toolkit for Scientific Computation
PFR	Parallel-Friendly Levels
PFR	Parallel-Friendly Rows
QR	Matrix decomposition into Q and R
ROC	Receive Operating Characteristic curve
SAS	Single-Algorithm-Split
SELL-C-α	Sliced ELLPack format with row sorting
SFL	Serial-friendly Levels
SpTRSV	Sparse Triangular Solve
SpMV	Sparse Matrix-Vector Multiplication
SPGEMM	Sparse General Matrix-Matrix Multiplication

Chapter 1

INTRODUCTION

In the modern world, scientific computing has emerged as the *third pillar of science* [3]. Golub and Ortega define scientific computing as *"the collection of tools, techniques, and theories required to solve on computer mathematical models of problems in science and engineering"* [4]. In general, scientific computing relies on the collaboration of computer science and mathematics to develop methods to solve scientific and engineering problems on modern computers. Numerical linear algebra, an integral part of the domain of scientific computing, helps in answering problems such as solving systems of linear equations, Eigenvalue problems, and least square optimization problems, to name a few. These problems can be solved using *Direct* or *Indirect* methods [5]. These methods, in turn, rely on basic linear algebra operations such as matrix-vector multiplication, vector operations and solving triangular systems [6].

Many problems in scientific computing and computational science are modeled as partial differential equations. The discretization of these models for implementation on digital computers results in *sparse linear systems*. Sparse linear algebra is a subbranch of linear algebra dealing with the solution of sparse linear systems. The sparse triangular solve (SpTRSV) is an important sparse linear algebra kernel that serves as the basic building block [4] in many direct methods [7, 8] and preconditioned iterative solvers [5] such as ILU-preconditioned GMRES solvers [9]. Being an inherently serial operation, triangular solve is a challenging kernel to implement on parallel computers. The reason is the existence of dependencies among different unknowns such that an unknown can only be solved once all its dependencies are resolved. While these dependencies are generally predictable in the case of dense

matrices, they depend upon the distribution of nonzeros or the matrix *sparsity pattern* [10] in the case of sparse matrices. Therefore, for sparse matrices, the missing dependencies among unknowns make it possible to compute some unknowns in parallel. For instance, if the triangular matrix contains only the diagonal entries, all the unknowns can be potentially computed in parallel in a single step. On the other hand, a lower triangular matrix with a main diagonal and first diagonal below the main diagonal represent the case in which each unknown should be in a separate step. These two examples represent matrices on the opposite ends of the parallelism spectrum with the diagonal matrix representing 100% parallelism and the matrix with a main diagonal and first diagonal representing 0% parallelism for SpTRSV. Between these two parallelism extremes, depending upon its sparsity pattern, a given sparse matrix may require any number of sequential steps with a possibly varying number of unknowns solved in parallel in each step for SpTRSV. The vast spectrum of sparsity patterns and resultant parallelism characteristics make it very challenging for a single SpTRSV implementation on modern computers to performs the best for all input matrices.

As the semiconductor scaling limits and power and thermal constraints curtailed the growth of single-core processors, computer hardware vendors resorted to the development of computing systems with multiple cores, enabling computations to be performed in parallel. The result is the emergence of multicore CPU and many-core Graphics Processing Unit (GPU) based parallel systems. A common execution model for a single node such parallel systems is to use a multicore CPU with one or more GPUs to form a so-called *heterogeneous computing system*. In such systems, the CPU has fewer but more powerful, heavyweight processing cores and is more friendly for performing serial computations. On the other hand, GPUs have a large number of less powerful, lightweight cores capable of processing huge amounts of data in parallel. The combination of the two platforms (CPU and GPU) with such diverse and complementary processing characteristics make them suitable for use in numerous practical applications including scientific computing.

Efficient use of CPU-GPU based heterogeneous systems for optimal performance

is not a trivial task. Program performance on such platform depends upon the computational characteristics of the application, platform architecture, and the programmer skills. To make the programmer's life easier, many frameworks and tools have been proposed in literature to enable programmers to write performance portable programs that once written automatically tune themselves aiming to give optimal performance on various underlying hardware platforms [11]. Autotuning is still an active research area as new CPU and GPU architectures and algorithms are developed from time-to-time to solve scientific and numerical computing problems.

For the parallel solution of SpTRSV, many algorithms and their implementations are available for the CPUs [12–15] and GPUs [1, 16–19] are available. Empirical evidence from the existing research has shown that while highly parallel algorithms perform exceedingly well for matrices having few computational steps with a high number of unknowns per step, sequential algorithms perform better for matrices with high number of steps with fewer unknowns per step [1, 18, 20]. As the number of steps and unknowns computed per step is a function of sparsity pattern of the input matrix, there is *no single algorithm or platform that gives the best SpTRSV performance for all input matrices (*).* Moreover, for systems with matrices having a mix of highly parallel and sequential steps, *a parallel algorithm is expected to provide benefits for steps with large number of unknowns, but its performance is expected to deteriorate for levels with few unknowns(**)* [21]. The converse can be said about the performance of sequential algorithms for such matrices.

Based on the first observation in the previous paragraph (*), it is easy to conclude that by selecting an appropriate SpTRSV implementation for a given input matrix, one can achieve higher SpTRSV performance. However, given the abundance of SpTRSV implementations for both the CPUs and GPUs, making this selection can be complex task. An obvious approach to select the fastest SpTRSV implementation is to run different implementations in turn and make the selection based on the empirical results. Given each SpTRSV implementation has its own data structures, API and matrix analysis requirements, selection of SpTRSV implementation based on empirical results is non-trivial and time-consuming. Existing research dealing with

SpTRSV algorithm selection on a single platform (CPU or GPU) exists [22, 23]. However, to the best of our knowledge, no existing research targets both the CPU and GPU for SpTRSV algorithm selection. The second observation (**) leads to an interesting research question of whether one can split the single SpTRSV into parts and use a different algorithm for each part to solve the system. The idea is to use highly parallel algorithm for the parallel part(s) and sequential algorithm for the sequential part(s). However, for such an SpTRSV execution model, multiple decisions are to be made and mechanisms have to be developed such as, i) determining whether a given matrix is suitable for split SpTRSV execution, ii) selection of split point, iii) assignment of appropriate algorithm, iv) management of data structures, and v) management of inter-platform data communication. Existing works dealing with the split SpTRSV execution [15, 24] are available. However, these works have the following limitations, i) they target a single execution platform, CPU [15] or GPU [24], and ii) they always perform split execution even if unsplit execution might be a better option.

In this dissertation, we propose tools and techniques to achieve higher SpTRSV performance for a given input matrix on modern CPU-GPU heterogeneous systems comprising of a single multicore CPU and one or more GPUs. The major contributions of this dissertation can be summarized as below:

- We propose a machine learning based framework for the prediction of the fastest SpTRSV algorithm for CPU-GPU heterogeneous systems.

- We devise an execution model for SpTRSV for split execution on CPU-GPU platforms. The split execution model is implemented as C++/CUDA library supporting multiple SpTRSV implementations. The model is capable of making decisions about split/unsplit SpTRSV execution, split point, and algorithm selection for split or unsplit execution.

- We provide performance evaluation results for our SpTRSV prediction framework on a modern CPU-GPU platform (Intel Xeon Gold CPU + NVIDIA Tesla V100 GPU).

- We provide performance evaluation results for our split execution models on two CPU-GPU machines of varying characteristics (Machine 1: Intel Core I7 CPU + GTX 1080 Ti GPU, Machine 2: Intel Xeon Gold CPU + NVIDIA Tesla V100 GPU).

The rest of the dissertation is organized as follows: Chapter 2 covers some background on working of the SpTRSV kernel, its importance in iterative solvers, various SpTRSV implementations on CPU and GPU, and motivation for this research. Chapter 3 presents some related work on the general problem of algorithm selection, algorithm selection for SpTRSV and hybrid CPU-GPU computing approaches in scientific computing. Our SpTRSV prediction framework is presented in Chapter 4 followed by the split SpTRSV execution model which is presented in Chapter 5. Performance evaluation of the SpTRSV prediction framework and split execution model are presented in Chapter 6. Finally, the conclusion of this dissertation is presented in Chapter 7.

Chapter 2

BACKGROUND AND MOTIVATION

In this chapter, we present the required background on the sparse triangular solve kernel, its various algorithms and implementations, its importance in numerical and scientific computing and the motivation for our research. We begin with the introduction of the basic triangular solve in Section 2.1, followed by the introduction of sparse triangular solve as a method to solve sparse linear systems in Section 2.2. In Section 2.3, we elaborate the importance of SpTRSV in some preconditioned iterative solvers. Next, in Section 2.4, we introduce necessary methodology and terminology necessary to understand the existing SpTRSV algorithms and new methods proposed in this research. Section 2.5 gives an overview of existing CPU and GPU SpTRSV algorithms. Finally, in Section 2.6, we introduce our research topic and discuss factors motivating this research.

2.1 Sparse Triangular Solve (SpTRSV)

2.1.1 Basic Triangular Solve

Triangular solve is an operation used in linear algebra to solve a linear system of the form:

$$Ax = b \tag{2.1}$$

Where A is a matrix with m rows and n columns, x is a vector of unknowns to be computed and b is the right hand side. One of the methods to solve equations of type 2.1 is to factorize the matrix A into lower (L) and upper (U) triangular parts such that $A = LU$. Equation 2.1 can then be written as $LUx = b$ and, because of triangular nature of L and U, can be solved in the following two steps:

$$\begin{bmatrix} 3 & 3 & 1 \\ 3 & 2 & 2 \\ 1 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1/3 & 1 & 0 \\ 1/9 & 1/6 & 1 \end{bmatrix} \begin{bmatrix} 3 & 3 & 1 \\ 0 & -2 & 2/3 \\ 0 & 0 & 22/9 \end{bmatrix}$$

Figure 2.1: LU decomposition of a sample 3×3 matrix

1. Solve $Ly = b$

2. Solve $Ux = y$

The matrices L and U are such that L has all zero entries above the diagonal while U has zero entries below the diagonal. Generally the diagonal entries for L are set to 1. Due to the shape of non-zeros in L and U (resembling a triangle), the system is known as triangular system and the method to solve the triangular system is referred to as *Triangular Solve*. An example decomposition of a 3×3 matrix is shown in Figure 2.1.

2.2 Sparse Linear Systems and SpTRSV

Sparse matrices are the matrices in which most of the data entries are zero. They often occur in the field of scientific computing as as result of discretization of partial differential equations modeling a physical system [5]. Generally, sparse matrices representing physical systems can be very huge in size.

The solutions of linear system, $Ax = b$, where A is a sparse matrix can be obtained with two type of methods, i) *Direct methods* ii) *Iterative Methods*. In direct methods, solution of the linear system is obtained in a single step while in iterative methods, an approximate solution is approached in multiple steps. At each step, the solution obtained is more accurate than the previous step and the method stops when desired solution accuracy is achieved [5].

Most of the direct and iterative methods for sparse linear systems utilize LU decomposition and triangular solve for their operation. Unlike dense matrices, *fill-ins* is a matter of concern for sparse matrices when LU factorization is performed.

Fill-in refers to phenomenon whereby the L and U parts have more non-zeros than lower and upper parts of the matrix A. Fill-in can be a big concern in case of very large sparse matrices resulting in prohibitively large L and U matrices that may give rise to memory and computational time issues in computing systems. To avoid these issues, scientists have devised strategies to minimize LU factorization fill-in. Separate strategies exist for direct and iterative methods. In our research, we are limited to the LU factorization and SpTRSV for iterative methods only and therefore SpTRSV for iterative methods will be discussed in this document.

2.3 Preconditioning in iterative methods and SpTRSV

An important application of sparse triangular solver in iterative methods is for *preconditioning*. Preconditioning refers to a linear algebraic operation in iterative methods that either accelerates convergence rate of the iterative method or can make non-converging system converge. Various preconditioners are available in practice with different computational and convergence properties [5]. For instance, Jacobi preconditioners are easy to compute having high degree of parallelism but provide slower convergence compared with other preconditioners. On the other hand, the sparse triangular solve based Incomplete LU (ILU) preconditioners are comparatively more expensive to apply but give better convergence compared to Jacobi preconditioner. ILU preconditioners are widely used in practice.

As discussed in the previous section, fill-in in LU factorization can give rise to memory and computational issues. To overcome these, scientists and mathematicians have come up with alternate strategies for LU factorization suitable for SpTRSV application in sparse linear systems solution using iterative methods. Instead of performing complete LU factorization, methods have been devised to perform so-called *Incomplete LU factorization* with varying degree of fill-in. Such methods are generally referred to as $ILU(k)$ or $ILUT$ methods. The $ILU(k)$ methods allow fill-in based on the structure of input matrix A. The k in $ILU(k)$ refers to the degree of fill-in. For instance in $ILU(0)$ factorization, non-zero structure in L and U is same as upper and lower parts of matrix A. The level of fill-in increases with increasing

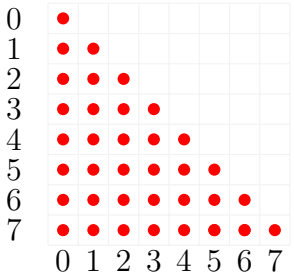


Figure 2.2: Sparsity pattern of a sample dense triangular matrix L

Algorithm 1 Dense triangular solve

1: **for** $i = 0, ..., n-1$ **do**
2: $y_i = b_i / l_{ii}$
3: **for** $j = i+1, ..., n-1$ **do**
4: $b_j = b_j - l_{ij} \times y_i$
5: **end for**
6: **end for**

value of k (1, 2, 3 etc.). $ILU(k)$ rely on structure of matrix A for fill-in, while in $ILUT$, the value to be dropped in LU factorization is decided based on a pre-selected threshold (T in $ILUT$). Similarly, while the zero pattern for $ILU(k)$ methods is pre-determined, it is dynamic for $ILUT$ methods. A more detailed discussion of these methods can be found in [5].

2.4 Solution of Triangular Systems

To explain the basic solution process of a triangular system, let us first consider the solution of a dense lower triangular system $Ly = b$ with sparsity pattern as shown in Figure 2.2(a). Here, the red dots represent the non-zero values. Algorithm 2 shows the steps involved in the solution of the dense triangular systems. As evident from this algorithm, the solution of each y_i depends on the previously calculated unknowns making it a highly serial operation.

For the case of sparse matrices and incomplete LU factorization, the dependen-

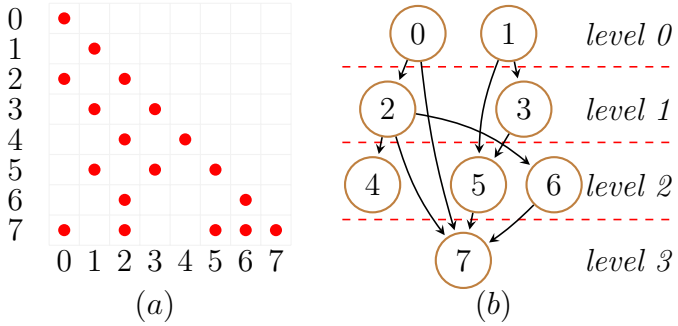


Figure 2.3: (*a*) Sparsity pattern of a sample lower triangular matrix L and (*b*) its dependency graph (DAG)

cies among rows may not be as sequential as for dense case. For instance, consider the sparsity pattern of a lower triangular matrices shown in Figure 2.3(a). Here, depending upon the sparsity pattern, a given y_i does not necessarily depend on all previously calculated unknowns. In this case, dependencies among unknowns can be represented in the form of a directed-acyclic-graph (DAG) as shown in Figure 2.3(b) for the matrix in Figure 2.3(a). The nodes in this graph 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. This makes it possible for the unknowns within a level to be computed in parallel. The average degree of parallelism (DoP) within a sparse triangular system can be computed using the equation:

$$Degree\ of\ parallelism = \frac{Number\ of\ matrix\ rows}{Number\ of\ levels\ in\ triangular\ system} \tag{2.2}$$

Equation 2.2 gives average number of rows in the triangular system that can solved in parallel.

Sparse matrices are generally stored in special formats that generally rely on storing non-zero values only. Compressed Sparse Row (CSR) is one of such storage formats. Algorithm 2 shows serial algorithm for the solution a lower triangular system where matrix L is stored in CSR format with array $rowPtr$, column indices array $colIdx$, and values array val. To exploit the parallelism in SpTRSV, many

Algorithm 2 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: $sum+ = val[j] \times x[colIdx[j]]$
5: **end for**
6: $x[i] = (b[i] - sum)/val[rowPtr[i+1] - 1])$
7: **end for**

algorithms have been developed for CPU and GPU machines which we will discuss in the next section.

2.5 Algorithms and Methods for SpTRSV

In this section, we will briefly give an overview of the existing algorithms and methods for SpTRSV on modern many and multicore architectures. The algorithms for the parallel SpTRSV can be widely classified into i) *Level-scheduling* algorithms, ii) *Synchronization-free* or *Self-scheduling* algorithms, iii) *Graph Coloring* algorithms, and (iv) *Partitioned Inverse* algorithms.

The level-scheduling algorithms [5] are design to take advantage of missing dependencies between unknowns in the same level. These methods were first introduced by Anderson and Saad [25] and then by Saltz [26]. The level-set methods are comprised of two phases; an *analysis* phase and a *solve* phase. In the analysis phase, the levels within the DAG are discovered while in the solve phase, the system is solved level-by-level. There a number of efforts dealing with level-scheduling implementations on the GPU with works in [1, 2] being the first ones to the best of our knowledge.

In self-scheduling algorithms, a pool of threads or processes is assigned to work on number of matrix rows and when a given thread or processes is done with the row(s), it checks if there are any unassigned rows. If so, the row(s) is assigned to the thread [27]. The algorithm continues until the solution of the whole triangular

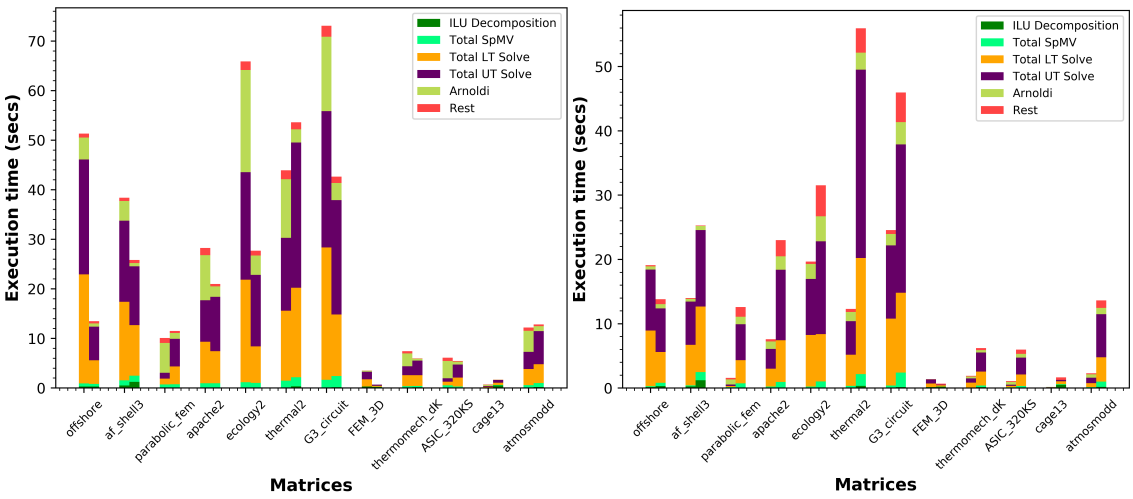
system is obtained. Like level-scheduling algorithms, self-scheduling algorithms are also comprised of an analysis and a solve phase. In the analysis phase, the indegrees of the nodes are calculated. A number of works dealing with self-scheduling SpTRSV on multiprocessor and distributed systems are available in [27–30]. For the GPUs, Liu et al. have implemented self-scheduling SpTRSV for a single [31] and multiple right-hand-sides [16].

In [17,18], Li et al. present different variants of level-scheduling and self-scheduling algorithms for the GPUs. An approach for SpTRSV on the multicore systems using both level-scheduling schemes and self-scheduling schemes is presented by Park et al. [13]. Vendor libraries, like Intel MKL [12] and NVIDIA cuSPARSE library [19], readily support solution of sparse triangular systems. Specifically, cuSPARSE library also allows the programmer to choose the underlying solution algorithm through their API (e.g., level-scheduling or self-scheduling).

An important difference between level-scheduling and self-scheduling is that level-scheduling algorithms have synchronization between two levels. Due to this global synchronization, the jobs that are ready in next level have to wait till the completion of the current level. The self-scheduling algorithms work by avoiding the global synchronization and can start the jobs as soon as they are ready to go.

Research efforts on sparse triangular solve performance improvement using graph coloring and matrix reordering based approaches can be found in [2, 32–34]. The essence of sparse triangular solve using coloring and reordering is that the number of levels in the triangular matrix can be significantly reduced using matrix reordering [2]. This, however, has the side effect of changing the triangular system [17].

In partitioned inverse methods, the inverse of the triangular matrix is represented as a product of few sparse matrices reducing triangular solve operation to a few matrix-vector multiplications [35,36]. Finally, there also exist methods using iterative techniques for approximate solution of triangular systems for preconditioning in iterative solvers [37,38]. The focus of our research is limited to the exact solution of sparse linear systems using level-scheduling and self-scheduling based triangular solve methods.



(a) Intel IvyBridge vs NVIDIA K80 (b) Intel Xeon Gold vs NVIDIA V100

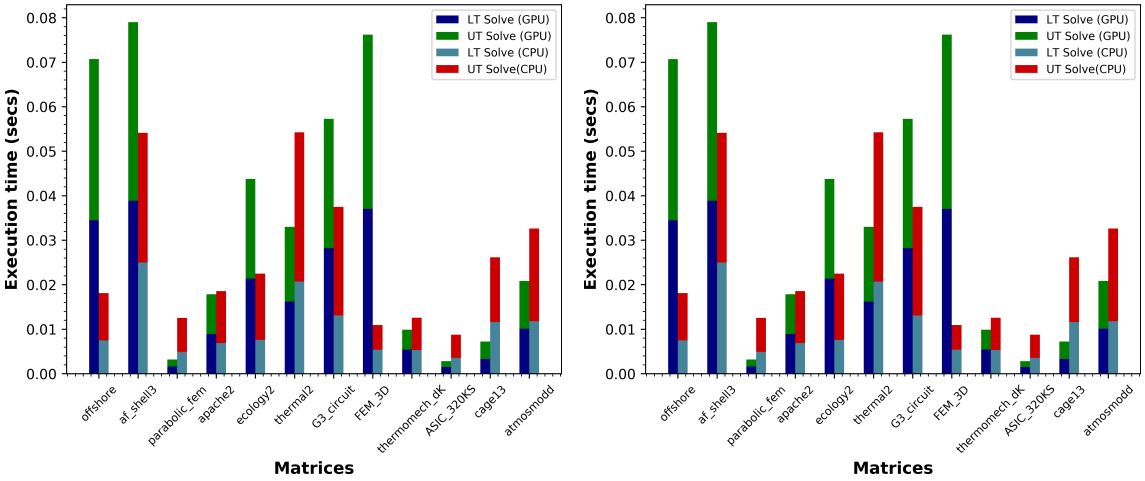
Figure 2.4: GMRES profiling results for matrix dataset used in [1]. For each matrix, first stacked bar represents execution time on GPU, second one on CPU. LT: Lower triangular, UT: Upper triangular

2.6 Research Focus and Motivation

As summarized in Chapter 1, the main focus of this research is to *efficiently use CPU, GPU or both for the solution of sparse triangular systems based on the analysis of the characteristics of the input matrix on modern heterogeneous systems.* Here, the term *efficient* refers to striving to attain best SpTRSV performance, specifically in terms of execution time, for a given input matrix. In this context, we are targeting heterogeneous systems consisting of a single CPU and one or more GPUs. The motivation for this research comes from the factors discussed in the following subsections.

2.6.1 SpTRSV in ILU(k)-based GMRES

Generalized Minimal Residual (GMRES) [9] is an iterative algorithm for solving sparse linear systems that often uses ILU(k)-based preconditioning. It is widely used iterative method because of its property that it works for both symmetric and



(a) Intel IvyBridge vs NVIDIA Tesla K80 (b) Intel Xeon Gold vs NVIDIA Tesla V100

Figure 2.5: SpTRSV performance on CPU and GPU for matrix dataset from [1, 2]. SpTRSV has winners (minimum execution time) on both the CPU and the GPU

unsymmetric systems. The GMRES with ILU(k) preconditioning requires SpTRSV for its working. Other linear algebraic operations in the GMRES algorithm include dot vector product, vector norms and sparse matrix-vector multiplication (SpMV). To test and compare the performance of various operations in GMRES on both the CPU and the GPU, we implemented the GMRES algorithm presented in [6] on CPU and GPU using Intel MKL (Intel Parallel Studio 2019) and NVIDIA CUDA (version 10.1), respectively. The profiling results of various constituent on two CPU-GPU machines (Intel Ivy Bridge E5-2695V2 + NVIDIA Tesla K80, Intel Xeon Gold (6148) + NVIDIA Tesla V100) for a matrix dataset used in [1] are shown in 2.4. Evidently, SpTRSV is the most time-consuming operation for almost all these matrices. This observation has also been confirmed in [39] by Jamal et al. This finding implies that *if we can improve the performance of SpTRSV on a CPU-GPU machine, we can considerably improve performance of iterative solvers* such as GMRES as a result.

2.6.2 SpTRSV performance, CPU versus GPU

If we focus only on the SpTRSV performance in Figure 2.4 (Figure 2.5, the SpTRSV performance results on modern CPU and the GPU architectures confirm another interesting finding from previous research [1, 2] that *SpTRSV performs better on the CPU for some matrices and it performs better on the GPU for others.* We repeated the experiment with a bigger matrix dataset (37 matrices) taken from SuiteSparse collection [40] and used in [1, 2] to observe that both the CPU and GPU have almost the same number of winners. This observations leads us to the conclusion that *on a CPU-GPU platform, either of the CPU or GPU can win for SpTRSV depending on the sparsity structure of the input matrix.* This implies that *choosing the right platform for ILU execution can enhance SpTRSV performance.* As SpTRSV executions multiple times in a single run of iterative solvers such as ILU-preconditioned GMRES, *SpTRSV performance improvement can lead to potentially significant execution time improvements in such solvers.*

2.6.3 SpTRSV performance with Algorithm choice

It has been shown in the previous research that the highest performing SpTRSV algorithm on a given platform (CPU or GPU) depends on the sparsity pattern of the input matrix [10]. In other words, although there are many promising implementations available for each platform, no single algorithm has been shown to achieve the best SpTRSV performance for all input matrices [1, 13, 16–18, 31]. This means a mechanism to choose the fastest SpTRSV algorithm for a given input matrix can help achieve better SpTRSV performance and hence the numerical method employing it.

The discussions in Subsections 2.6.1, 2.6.2 and 2.6.3 serve as the main motivations for this work. Based on these motivating factors, the methods and tools proposed in this thesis target heterogeneous CPU-GPU platforms to achieve higher SpTRSV performance, by using CPU only, GPU only or both, compared with using a single algorithm or a single platform.

Chapter 3

RELATED WORK

In this Chapter, we present a review of the related work for algorithm selection and CPU-GPU hybrid computing for linear algebra kernels used in scientific computing. We also compare and contrast the existing research with our approach and summarize the differences.

3.1 Algorithm and Implementation Selection

In this section, we present an overview of the existing research in the domain of algorithm or implementation selection for solving numerical computing problems and contrast it with our prediction framework presented in Chapter 4.

The general problem of algorithm selection [41] has been previously studied in the domain of numerical computing using statistical [42, 43], empirical and machine learning techniques. ATLAS [44] is an autotuning library for dense linear algebra operations. It works by exploring performance of a linear algebra kernel (e.g. GEMM) on a given machine by inspecting machine properties (like cache-size and floating-point pipeline length), and then applying standard performance engineering techniques (like blocking, loop unrolling) resulting in multiple implementations of the kernel. After eliminating unlikely choices by heuristic search, ATLAS selects the best implementation based on empirical performance results of remaining implementations. ATLAS does not take into account characteristics of the input data for autotuning and therefore no decisions are made at runtime. In case of sparse matrices, performance of the linear algebra kernels not only depend on the characteristics of the underlying hardware but also on the non-zero structure of the input matrix [10]. Optimized Sparse Kernel Interface (OSKI) [22] is an autotuning library based on statistical techniques for sparse linear algebra kernels on CPU plat-

forms, particularly the sparse matrix-vector multiplication (SpMV) and SpTRSV. The library can transparently tune kernels at runtime using the machine input matrix characteristics. The library only selects best kernel for execution on the CPU. PetaBricks [45] is a language and compiler that allows multiple implementations of multiple algorithms for the same problem and automatically selects the best algorithm by building and using the so-called choice dependency graph. Sequoia [46] autotunes an application based on memory hierarchy of the underlying machine. While PetaBricks and Sequoia do not take input data characteristics except the data set size, Nitro [47] framework allows programmers to guide the algorithm selection process by letting them specify characteristics of the input data they want to be considered for algorithm selection. These frameworks require programmers to learn new APIs and procedures to guide he algorithm selection process. In [39], Jamal et al. propose a hybrid CPU-GPU approach for their parallel algebraic multilevel solver (pARMS) in which they use both CPU and GPU in preconditioning phase. For the case of ILU(k) preconditioning, they always perform LU factorization on the CPU and always perform SpTRSV on the GPU. A previous work dealing with SpTRSV execution choice between CPU and GPU is presented in [48] for the MAPS reservoir simulation systems. It includes a highly tuned collection of solvers and preconditioners for both the CPU and the GPU, mostly based on Incomplete LU (ILU). The framework orchestrates these components on a heterogeneous system based on some heuristics geared towards enhanced overall simulation performance. Although targeting heterogeneous systems for preconditioning and SpTRSV, the framework is specifically tuned to work for reservoir simulations with input matrices exhibiting characteristics particular to such simulations. Also, the preconditioner, once selected, always executes on the CPU or the GPU. A recent work by Dufrechou et al. [23] uses supervised machine learning to automatically select SpTRSV algorithms on the GPUs. They tested their model for selection among cuSPARSE library-based SpTRSV and three variants of a CSR-based self-scheduling algorithm [49]. Their model managed to achieve an accuracy of close to 81%.

A number of works exist dealing with the selection of solvers and preconditioner-

solver pairs for numerical software. Lighthouse [50] allows users to select the right solver and generate corresponding code for PETSc applications. It uses machine learning to analyze the matrix features and select the solver accordingly. The framework currently supports selection of solvers and preconditioners from PETSc [51–53] and Trilinos libraries [54]. Both PETSc and Trilinos provide libraries and tools for scientific computing and are widely used in practice. Lighthouse framework generates machine learning input dataset for both PETSc and Trilinos on a CPU-based supercomputer. It means that the solver-preconditioner pairs reported by Lighthouse currently are valid for CPU-based architectures. Motter et al. [55] utilize machine learning techniques for selecting solver-preconditioner pairs for the PETSc framework [52] taking into account machine characteristics. For their prediction framework, their methodology constitutes benchmarking the hardware with hardware benchmarking tools (e.g. HPL, DGEMM, STREAM etc.) to obtain machine features, matrix feature extraction with Anamod software package [56] and computing runtimes for various solver-preconditioner combinations (from PETSc library). They train a machine learning model using matrix and machine features and computing runtimes for a collection of matrices from SuiteSparse Matrix Collection [40]. They can then predict PETSc solver-preconditioner for the selected matrix and machine based on its numerical and structural features and characteristics of the underlying machine.

The algorithm selection strategies in this section can be classified into the, i) ones targeting a single platform (e.g. CPU, GPU), and ii) ones targeting both the platforms. Our SpTRSV prediction framework (Chapter 4 is fundamentally different from the first set of works in that we target both the CPU and the GPU for algorithm selection. For the second set of strategies where both the CPU and GPU and considered for algorithm selection and execution, the process either requires significant programmer intervention, always execute specific algorithms on specific platforms (e.g. LU factorization on the CPU and SpTRSV on the GPU) or target a specific application area (e.g. reservoir simulation). In contrast, our prediction framework is designed for minimum programmer intervention, extensibility and flex-

ibility in platform selection for SpTRSV execution. The work closest to our approach is presented in [23] that selects SpTRSV algorithms on the GPU. It, however, only targets GPUs, and shows lesser accuracy than our prediction framework.

3.2 Hybrid CPU-GPU computing

In this section, we present an overview of existing research for solving scientific and numerical computing problems by simultaneously using multiple execution platforms and algorithms. In [57], 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 [58], 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 [59], the authors devise a static scheduling mechanism to divide matrix computations between a CPU and a GPU for landform attributes representation, a mathematical approach to efficiently represent geophysical resources. Agullo et al. [60] accelerate QR factorization on a CPU-GPU machine by expressing QR factorization as a set of tasks. Benner et al. [61] use a sequential version of code with a multi-threaded version of BLAS to accelerate the computation of matrix sign function on a CPU-GPU platform. Muramatsu et al. [62] 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 sub-tasks between CPU and GPU based on the nature of the sub-tasks such as parallelism characteristics [63–71].

Several works dealing with CPU-GPU hybrid computations of sparse matrix-vector multiplication (SpMV) are available in literature. In [72], the authors use design patterns to distribute SpMV computations between CPUs and GPUs. In [73], 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 [74], 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 [75], 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. [76] utilizes a heuristic-based approach for work division of sparse matrix-matrix computations between CPU and GPU. They achieve 6× speedup over an optimized Intel MKL library implementation of SPGEMM. In [77], the authors propose SELL-Cα 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 el al. [78]. 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 [79]. 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. [80] 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 nd 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). They split the matrix of size M at a row that is equal to the next power of $2 \geq M/2$ and $< M$.

For the solution of sparse triangular systems, the works closest to our SpTRSV split execution model are presented in [15, 24]. In [15], the author proposed SpTRSV split execution on CPU in which the sub-triangular parts are solved using a level-scheduling [13] and a recursive blocking algorithm, respectively. Their approach works based on the observation that level-scheduling induced reordering 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. In [24], the proposed approach uses recursive blocking for solving SpTRSV on the GPUs. Like the approach in [80] 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. To the best of our knowledge, our split execution model for SpTRSV is the only SpTRSV execution framework that is capable of splitting SpTRSV computation between CPU and GPU based on the analysis of the input matrix.

Chapter 4

A PREDICTION FRAMEWORK FOR SPTRSV

4.1 Introduction

The sparse triangular solve (SpTRSV) is one of the important kernels used in direct and iterative methods for sparse linear systems and least square problems [16]. Efficient implementation of SpTRSV on CPU and GPU has been extensively studied and many SpTRSV implementations are available [1, 2, 12, 13, 15–17, 22, 31]. However, there is no single execution platform or algorithm that gives the best SpTRSV performance for all input matrices. This is because, given a sparse matrix, the SpTRSV performance depends upon characteristics of the available parallelism in the matrix and implementation details of the algorithm (e.g. data structures, the sequence of operations etc.) [10]. Therefore, CPU has been observed to give better SpTRSV performance for some matrices than GPU and vice versa [1,2]. Also, different SpTRSV implementations on the same platform have been observed to achieve higher performance than others for different matrices [1, 17]. By selecting appropriate SpTRSV implementation for a given matrix, one can achieve higher SpTRSV performance. This can result in considerable performance gains for applications requiring multiple SpTRSV iterations, e.g., iterative solvers [33]. Given that many SpTRSV implementations are available for each platform, this selection can be a complex task. An obvious approach to select the fastest SpTRSV implementation is to run different implementations one-by-one and collect the empirical results. This, however, is a time-consuming and non-trivial process as each SpTRSV implementation has its own data structure, API, and matrix analysis requirements [1, 16].

In this work, we propose a machine learning-based framework for predicting the fastest SpTRSV algorithm for a given matrix on CPU-GPU heterogeneous sys-

tems. The framework works by extracting matrix features, collecting algorithm performance data, and training a prediction model with 998 real matrices from the SuiteSparse Matrix Collection [40]. Once trained on a given machine, the model can predict the fastest SpTRSV implementation for a given matrix by paying a one-time matrix feature extraction cost. The framework is also capable of taking into account CPU-GPU communication overheads, which might be incurred in an iterative solver. We test our prediction framework for two CPU and four GPU algorithms on a modern Intel Xeon Gold CPU and NVIDIA Tesla V100 GPU systems. The model achieves an average prediction accuracy of 87% on our selected platform. Experimental results show predicted implementation achieving an average speedup (harmonic mean) in the range 1.4x-2.7x over a lazy choice of one of the six SpTRSV implementations used in this study.

The contributions of this work are summarized below:

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

4.2 Motivation

As discussed in Chapter 2, existing research has shown that no single algorithm or platform performs best for all matrices. To demonstrate this, we test six SpTRSV implementations (2 CPU, 4 GPU implementations) for a set of 37 matrices from the

Table 4.1: SpTRSV winning algorithm breakdown for 37 matrices in Figures 4.1

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

SuiteSparse Matrix Collection [40]. Table 4.1 shows the breakdown of the winners (minimum execution time) among these six implementations for these matrices. Figure 4.1 shows the comparison for CPU and GPU winners for each of these matrices. The dashed vertical line separates the matrices into two groups; matrices attaining the best performance on CPU are on the left and on GPU, on the right. The x-axis shows the matrix degree-of-parallelism (DoP), which equals the average number of rows per level. Results show that no single algorithm or platform performs the best for all matrices.

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

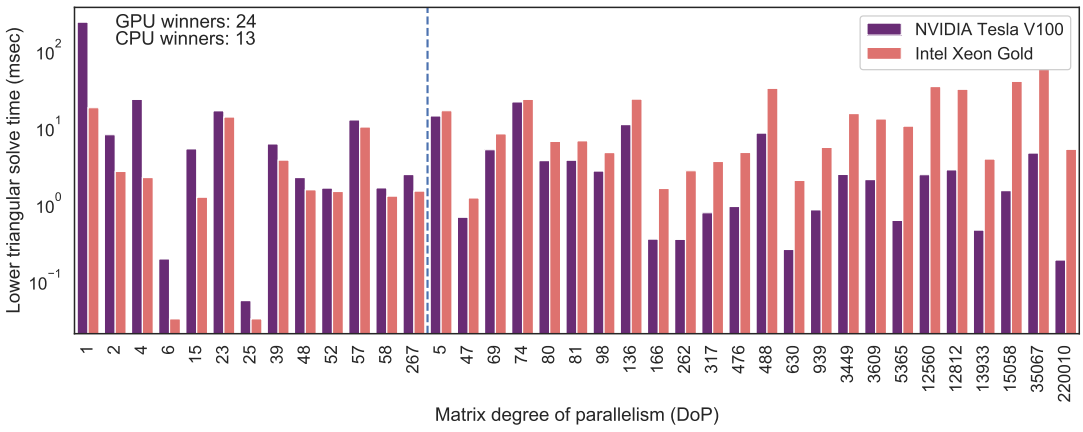


Figure 4.1: SpTRSV performance on Intel Xeon Gold (6148) CPU and an NVIDIA V100 GPU (32GB, PCIe).

4.3 Design and Implementation

The prediction framework is designed to automate the SpTRSV algorithm se- lection process for a given machine and matrix. It is composed of five main components (Figure 4.2); (1) An automatically downloadable set of matrices from the SuiteSparse Matrix Collection, (2) A matrix feature extractor, (3) An SpTRSV algorithm repository, (4) An SpTRSV performance data collector, which works by automatically downloading matrices, running each SpTRSV algorithm in the repository for each matrix, and logging the SpTRSV execution time. (5) A trainer and tester for the machine learning algorithm prediction model, which uses matrix features from the feature extractor as the input data and ID of the winning algorithm from performance data collector as the target for the model training and testing. Once the model is trained and tested, it can predict the fastest SpTRSV algorithm for a given matrix based on its features.

Figure 4.3 shows the design flow for our prediction model. For training the model, feature and algorithm performance data for the matrix data set is split into training and testing sets. Once trained with the training set, the model is tested with the matrices in the test set. Next, we discuss the important parts of the framework:

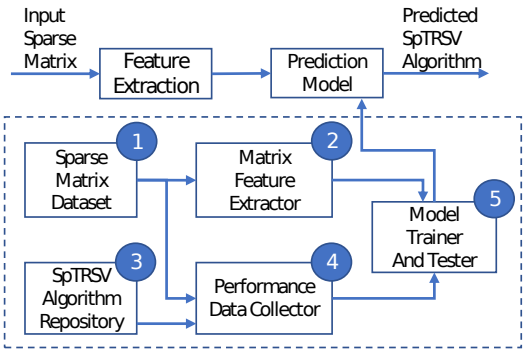


Figure 4.2: The prediction framework block diagram

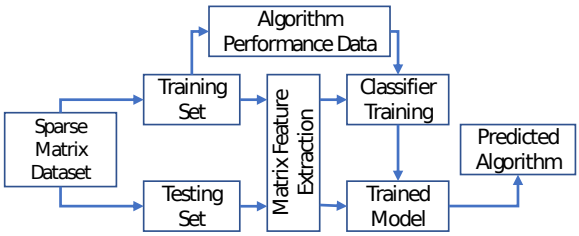


Figure 4.3: Prediction model design flow

(1) feature set selection, (2) feature extraction, and (3) machine learning model for prediction.

4.4 Feature Set Selection

SpTRSV performance is mainly affected by the sparsity pattern (i.e. the dis- tribution of nonzero (*nnz*) elements in the matrix) [10]. The sparsity pattern is described by matrix structural data such as the number of rows, columns, nnzs, row and column lengths etc. We initially started with a set of around 50 structural features. After feature correlation analysis and feature score comparison, 30 structural features are finalized. We choose not to reduce the number of features further because reducing the number of features from 30 to, say, 10 negligibly improves the overhead of the feature extraction process but results in up to 10% drop in prediction accuracy. This is because many of the top-scoring features require per-level information, which in turn requires level calculation, which is generally the most time-consuming part of the matrix analysis phase [1]. The majority of the other features can be cheaply collected as a part of the level calculation process. Table 4.2 lists the final feature set used by the framework. The last column in the table also shows the score rank for each feature, where the lower score rank means a higher impact on performance.

Table 4.2: Selected feature set for the prediction framework

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

4.5 Feature Extraction

Feature extraction is an overhead for the SpTRSV algorithm prediction and its execution time should be kept minimum. To achieve this, we employ both CPU and GPU in our feature extraction tool. This process completes in three steps. In the first step, row dependencies (row lengths) for lower/upper triangular matrices are calculated on GPU. Then, we use a slightly modified CUDA implementation of Kahn's algorithm [81] presented in [17] to construct levels in a triangular matrix. The algorithm calculates levels and rows in a level by performing topological sorting on the dependency graph. It recursively finds rows with zero dependencies, saves the row IDs of the current level into a queue, and then removes these rows and their

outgoing edges from the graph until no more rows to process. In addition to level calculation, we also collect some statistics such as the number of rows per level, row and column lengths per level, and the nnzs per level. Finally, the remaining features listed in Table 4.2 are calculated using the NVIDIA Thrust library [82]. For this purpose, while CPU iterates over levels, GPU is used to calculate features for that level.

4.6 Machine Learning Model and Training

For training the model, we use the Scikit-learn machine learning library in Python [83]. As the matrix data set, we choose 998 real square matrices with 1000 or more rows (up to 16M rows) from the SuiteSparse Matrix Collection. We train the model with two CPU SpTRSV algorithms, namely *MKL(seq)* and *MKL(par)*, and four GPU algorithms, namely *cuSPARSE(v1)*, *cuSPARSE(v2) with* and *without level-scheduling* and synchronization-free algorithm (*Sync-Free*) [16].

We assign a unique integer ID to each of these algorithms and collect features and SpTRSV performance data for each matrix in our data set in an automated fashion using the libufget library [84] and our feature extraction tool. The matrix features and the ID of the fastest SpTRSV implementation then serve as input and target, respectively, for training the machine learning model.

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

accuracy.

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

4.7 Effects of CPU-GPU Data Transfers

In a CPU-GPU system, executing the fastest SpTRSV algorithm may require data transfers between CPU and GPU. For instance, GMRES solver with preconditioning performs sparse-matrix multiplication and vector products in addition to SpTRSV in each iteration [9]. With data transfer overheads, the fastest SpTRSV algorithm may no longer be the fastest as another implementation may require no data transfer.

To elaborate on this, consider a lower triangular system $Ly = b$ to be solved with SpTRSV (see Chapter 2). For iterative methods, matrix L is generally fixed while b and y are updated every iteration. Consider the scenarios shown in Figure 4.4, where computations just before and after SpTRSV, execute on different platforms. In Figure 4.4, H->D and D->H represent host-to-device and device-to-host data transfers, respectively. As shown in the figure, the data transfer for either the right-hand side or solution vector is inevitable. Therefore, it is always beneficial to choose the fastest SpTRSV algorithm irrespective of whether it runs on the CPU or on the GPU. For the scenarios where computations, just before and after SpTRSV, execute on the same platform and SpTRSV executes on a different platform (Figure 4.5), two data transfers are incurred; (the right-hand side and the solution vector). Consequently, this data transfer overhead may change the algorithmic choice. To

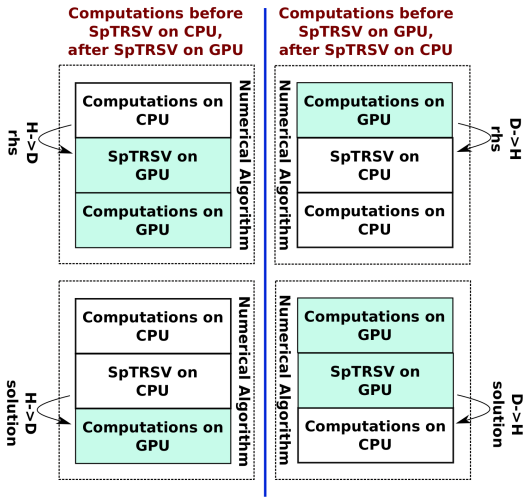


Figure 4.4: CPU-GPU data exchange when computations just before and after SpTRSV execute on different platforms

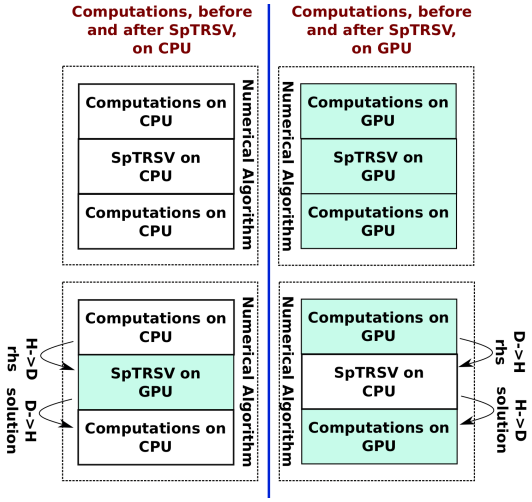


Figure 4.5: CPU-GPU data exchange when computations just before and after SpTRSV execute on the same platforms

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

Evaluation of the prediction framework is presented in Chapter 6.

Chapter 5

A SPLIT EXECUTION MODEL FOR SPTRSV

5.1 Introduction

In comparison with other sparse kernels such as sparse matrix-vector multiplication (SpMV) and sparse matrix transposition [85], Sparse Triangular Solve is an inherently sequential operations. Consequently, it has been observed to be one of the major time consuming operations in many numerical applications [20, 39].

The sequential nature of SpTRSV stems from inter-dependencies among SpTRSV computations, forcing it to be completed in a number of sequential steps (or *levels*). These dependencies can be represented as a direct-acyclic-graph (DAG). 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 the 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 [1, 16, 18, 20]. 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 [21]. 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.

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, inter CPU-GPU communication must be managed so that its cost does 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 [15]. The authors propose a strategy, called HTS, to partition SpTRSV computation into parts and perform SpTRSV using a level-scheduling algorithm on one part [13] 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. [24] 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 the 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.

5.2 Motivation

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 having a high number of levels with few unknowns (*thin levels*) [16, 20, 86]. Figure 5.1 shows a representative set of the *unknowns-per-level* histograms for SpTRSV DAGs of some matrices taken from the SuiteSparse collection [40].

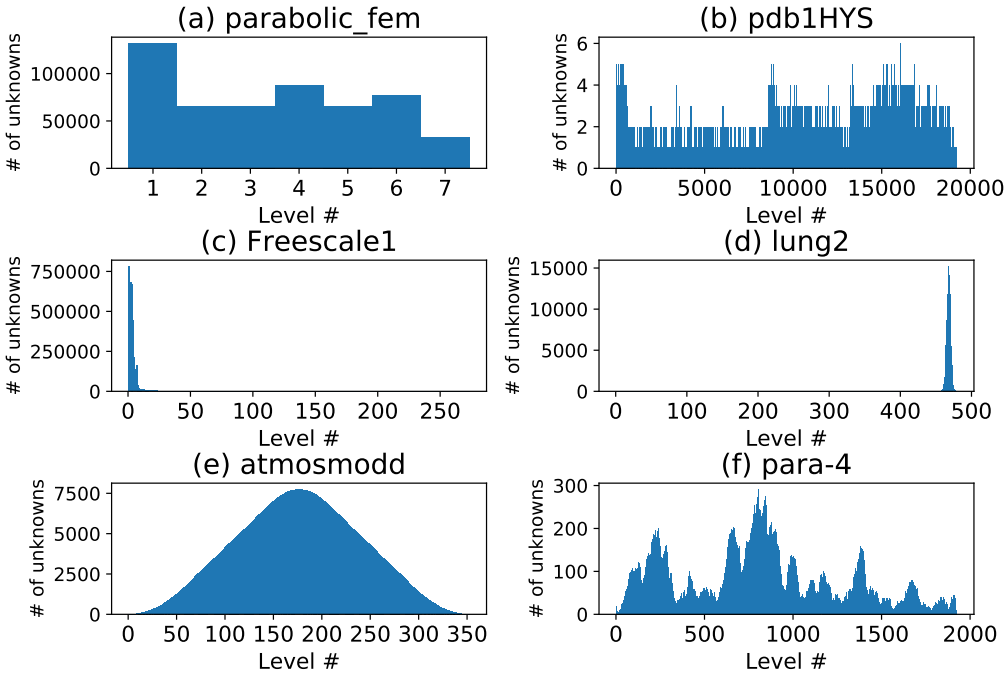


Figure 5.1: Rows(Unknowns) per-level plots for SpTRSV DAGs of some matrices from SuiteSparse collection

The DAG for matrix *parabolic_fem*, shown in Figure 5.1(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 5.1(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 5.1(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 5.1(e) that has thin levels on either side of the DAG and fat levels in the middle. In the DAG of Figure 5.1(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.

5.3 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 5.2(a) and the row rearranged version of the same matrix shown in Figure 5.2(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 5.2(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.

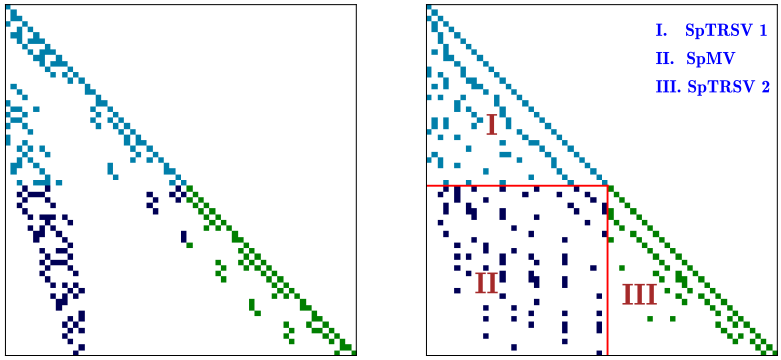


Figure 5.2: (a) Sparsity pattern of the lower triangular part of matrix *bfwa62* from the SuiteSparse collection. (b) Sparsity pattern of the DAG re-arranged version of the matrix in (a) and its division into two triangular matrices and a rectangular sparse matrix for split execution

5.3.1 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 performance gain with 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 [16–18,31] assign a fixed number of

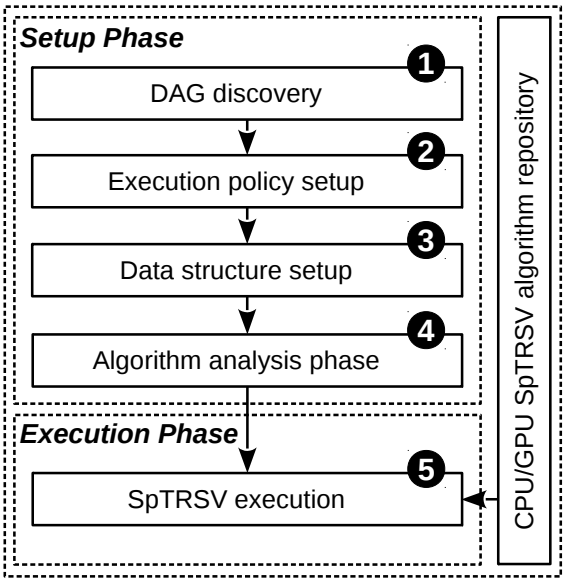


Figure 5.3: SpTRSV split execution model block diagram. The model is comprised of two phases, a one-time setup phase and an execution phase that can execute multiple times, potentially with a different right-hand side

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 [24]. 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., [87], 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.

5.4 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 5.3 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 inter CPU-GPU communication is automatically managed.

5.5 Methodology

In this section, we present our methodology for achieving an efficient split-execution model and explain details of the steps shown in Figure 5.3.

5.5.1 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 [1] is used. Other SpTRSV DAG features (explained in Section 5.5.3) are also computed in this step.

5.5.2 Step 2: Execution Policy Setup

Figure 5.4 shows the flowchart of the execution policy setting up procedure. In the Splitting Decision phase (labeled as (1)), we decide whether the triangular system would benefit from the 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, 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.

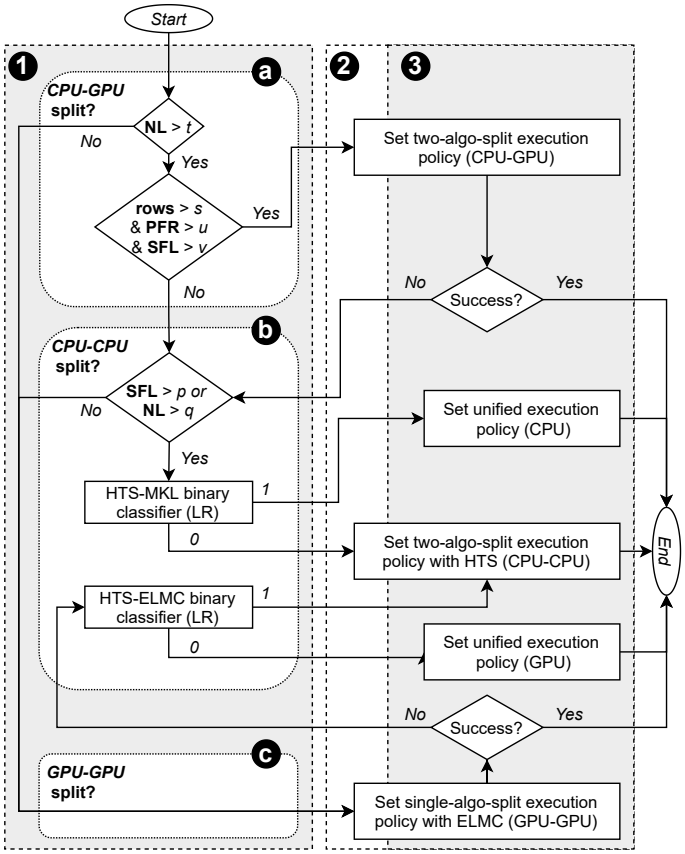


Figure 5.4: Flowchart for setting up the execution policy. (1) The splitting decision heuristic determines whether the triangular matrix is suitable for split execution. (2) DAG slicing determines split-point and (3) algorithm mapping chooses the SpTRSV algorithm for each part. LR: Logistic regression model, NL: Number of levels, PFR: Parallel-friendly rows, SFL: Serial-friendly levels

5.5.3 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*
- 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*

5.5.4 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 [40]. These three implementations are MKL [12], HTS [15], and ELMC [18]. 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 5.4. Our decision process is based on several observations, also presented in Figure 5.5.

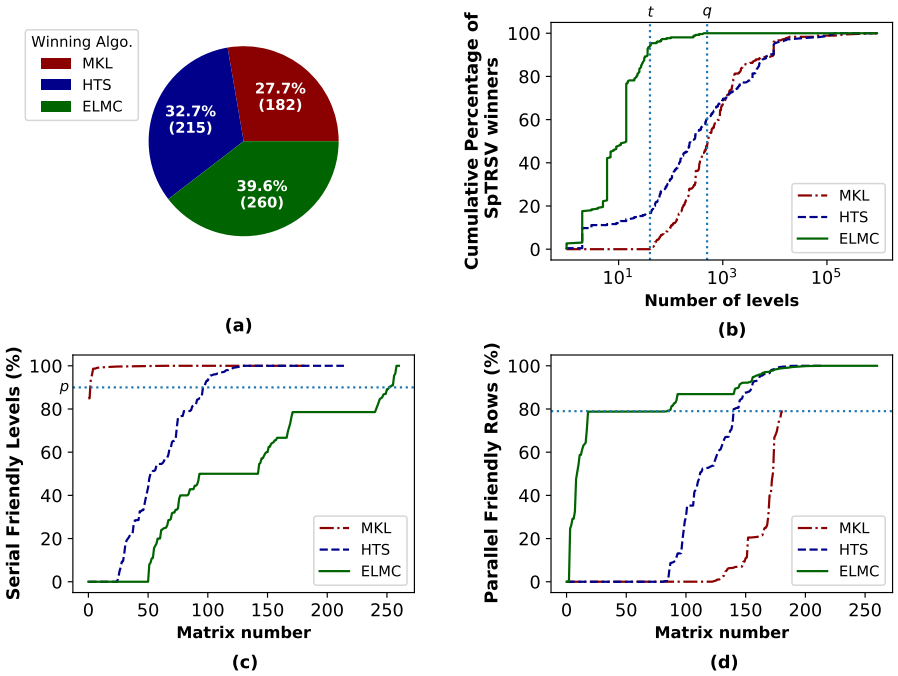


Figure 5.5: Statistical analysis of training data set. (a) SpTRSV winning algorithm (algorithm with least execution time) breakdown for the 657 matrices (b) Cumulative percentage of SpTRSV winners plotted versus SpTRSV DAG levels in the input matrix (c) For each winning algorithm, percentage of serial 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.

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 greater than t for CPU-GPU split execution. t is selected to be 40 and indicated by the first vertical dotted line in Figure 5.5 (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 5.5 (c)) while most of the ones for which ELMC wins have a high percentage of PFRs (80% and above as shown in

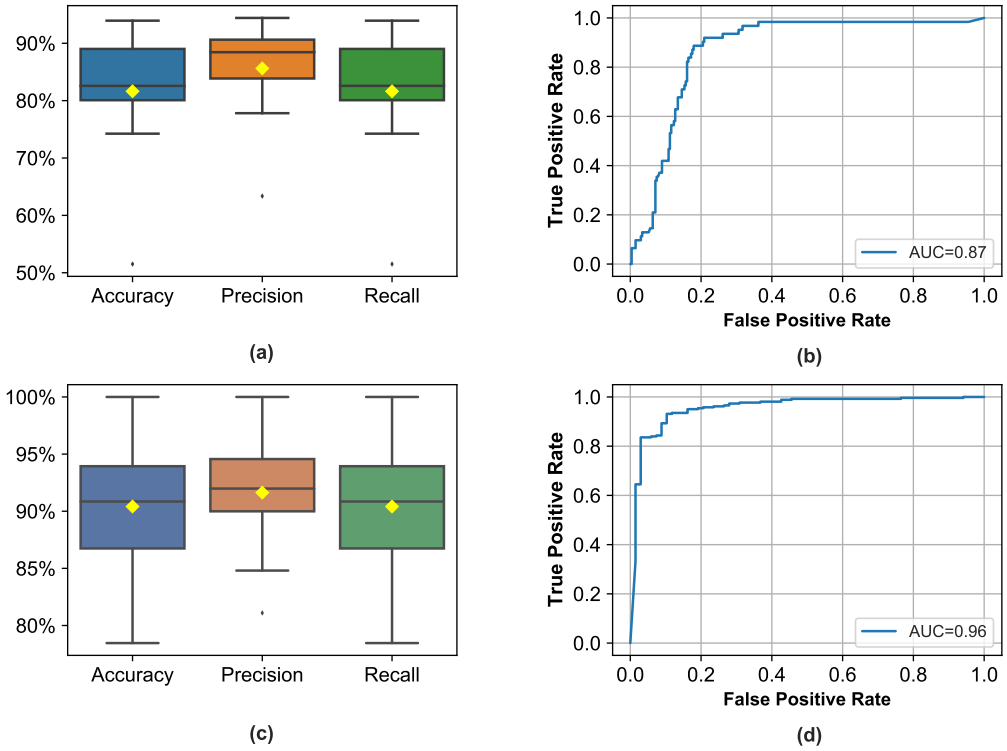


Figure 5.6: (a), (b) HTS-MKL Logistic Regression-based binary classifier performance. Mean 10-fold cross-validation scores for accuracy, precision, and recall are 81%, 85%, and 81% respectively. The area under the ROC curve is 0.87. (c), (d) HTS-ELMC Logistic Regression-based binary classifier performance. 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

Figure 5.5 (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 5.5 (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 5.5(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 5.6 (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 5.6 (b). Based on this analysis, we either employ unified execution with MKL or split execution with HTS on the CPU.

From Figure 5.5 (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 5.6(c). The model attains an accuracy of 90% with the area under the ROC curve as shown in Figure 5.6(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 5.4 (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.

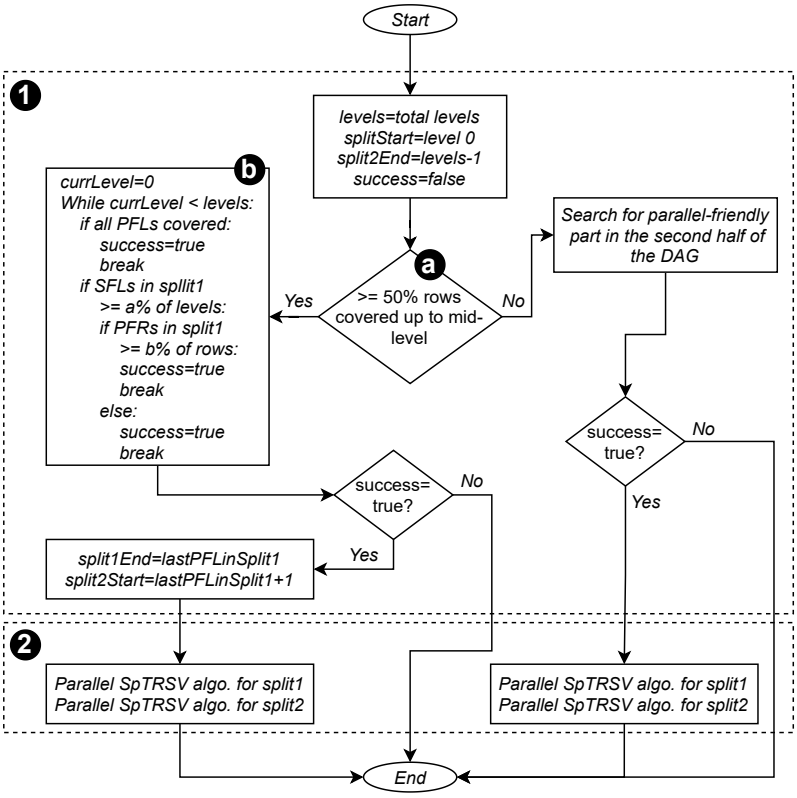


Figure 5.7: (1) DAG slicing and (2) algorithm mapping for two-algorithm-split execution (CPU-GPU). *split1Start, split2Start* are start, and *split1End, split2End* are the end levels for first and second splits, respectively. *lastPFLinSplit1* refers to the ID of the last parallel-friendly level in split 1.

5.5.5 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.

***DAG Slicing for CPU-GPU Execution*:** Figure 5.7 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 5.7(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 5.7(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 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 5.7 shows the search algorithm for the parallel-friendly part in the left part of the DAG only. 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 straight-forward 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.

***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

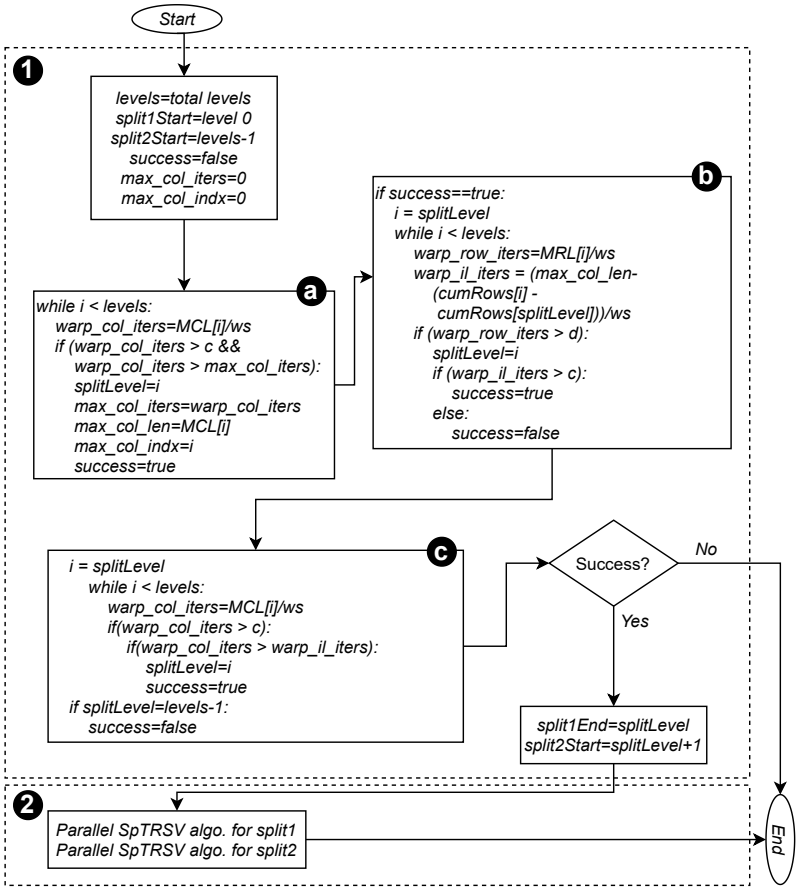


Figure 5.8: (1) DAG slicing and (2) algorithm mapping for single-algorithm-split. split1Start, split2Start are start, and split1End, split2End are end levels for the first and second split, respectively. splitLevel refers to the level number for DAG slicing.***ws*** refers to warp size

i, the DAG slicing algorithm selects the largest good level i as the split level such that the $C_i^{bad} \leq f_{bad}C_i$ [15]. Once the DAG is sliced, HTS uses level-scheduling and recursive blocking algorithms for executing SpTRSV 1 and SpTRSV 2, respectively.

***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 [16–18, 31]. In such cases, dividing the matrix into three

parts can reduce this bottleneck by having smaller triangular systems (I and III in Figure 5.2(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 [16–18, 31], and SpMV with a fixed number of threads per row [87], 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 5.8. 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 5.8 (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 5.8 (b), the split point is adjusted to satisfy criterion (ii). Finally, in Figure 5.8 (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.

5.5.6 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 5.2(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.

5.5.7 Step 4: Algorithm Analysis

As described in Chapter 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 5.3). In the case of a unified execution, the analysis phase for the sub-matrix corresponding to SpTRSV 1 is performed.

5.5.8 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:

$$b_{2(new)} := \alpha \cdot M \cdot x_1 + \beta \cdot b_{2(old)} \tag{5.1}$$

  Equation 5.1 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 5.2(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.

5.5.9 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 [12] in the CSR matrix storage format. In addition, for two-algorithm-split execution on CPU, an SpTRSV implementation using the HTS library [15] is supported. HTS uses a level-scheduling algorithm [13] for SpTRSV 1 and a recursive-blocking algorithm for SpTRSV 2. For GPUs, the framework supports SpTRSV and SpMV algorithms using cuSPARSE(v2) library [19], the Sync-Free SpTRSV algorithm by Liu et al. [16], and ELMC, ELMR SpTRSV algorithms by Li et al. [18]. For the framework evaluation in this study, we chose ELMC

over the other supported GPU algorithms for its superior performance as reported in [18]. The framework is extensible with more SpMV and SpTRSV implementations and enriching the collection is left as future work.

5.5.10 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:

$$\textit{Setup overhead} = T_{DD} + T_{EPS} + T_{DSS} + T_{AA} \tag{5.2}$$

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:

$$\textit{Execution overhead} = C_{SpTRSV1} + C_{SpMV} + C_{SpTRSV2} \tag{5.3}$$

$C_{SpTRSV1}$, 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:

$$\textit{Execution overhead} \leq ((2 \cdot r_1 + 2 \cdot r_2) \cdot FS)/BW_{interconnect} \tag{5.4}$$

where r_1 is the number of rows in the upper sub-triangular matrix (I in Figure 5.2), r_2 is the number of rows in the lower sub-triangular matrix (III in Figure 5.2),

$BW_{interconnect}$ 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.

Evaluation of the split execution model is presented in the next chapter.

Chapter 6

EVALUATION

In this chapter, we present and evaluate performance results of our Prediction Framework for SpTRSV discussed in Chapter 4 and our Split Execution Framework for SpTRSV discussed in Chapter 5.

6.1 Evaluation of SpTRSV Prediction Framework

In this section, we evaluate performance of different SpTRSV algorithms, our framework's prediction accuracy, its performance, and its overhead compared to the symbolic analysis phase of SpTRSV algorithms. The performance results were collected on a CPU-GPU machine with an Intel Xeon Gold (6148) CPU and NVIDIA Tesla V100 GPU. CPU has 2 sockets with 20 cores in each and comes with a 512 GB of memory. GPU has 32 GB of memory. The Intel MKL implementations are compiled with *icpc* compiler from Intel Parallel Studio 2019 with `-O3` optimization. For *MKL(par)*, all available CPU cores are used without hyperthreading. The cuSPARSE and Sync-Free implementations are compiled using `nvcc` compiler from CUDA version 10.1 with options `-gencode arch=compute_70,code=sm_70`. Statis-

Table 6.1: Number of rows and nonzero statistics for the 998 matrices from SuiteSparse

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

Table 6.2: SpTRSV winning algorithm breakdown for the 998 matrices from SuiteSparse

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

tics for the number of rows and nonzeros for the matrix data set are given in Table 6.1.

6.1.1 Performance of SpTRSV Algorithms

This section presents the experimental results for the six SpTRSV algorithms. For this purpose, each of the six SpTRSV implementations is run 100 times for each matrix in the data set and mean execution time is reported. The results presented here are for the solution of the lower triangular system. Table 6.2 shows the breakdown of the winning implementations for the entire matrix data set. As regards the number of times an SpTRSV implementation was the fastest for the data set, we observe that, in general, there is no clear GPU advantage over CPU. Intel *MKL(seq)* is the fastest for a high percentage of the matrices than any other implementation. This is possibly due to the fact that some matrices exhibit very low parallelism that can be exploited or variable degrees of parallelism. In general, Intel *MKL(par)* shows poor performance. The *cuSPARSE(v1)* surprisingly performs better than two variants of *cuSPARSE(v2)* combined. Moreover, on GPU, the *Sync-Free* implementation is dominant over cuSPARSE implementations.

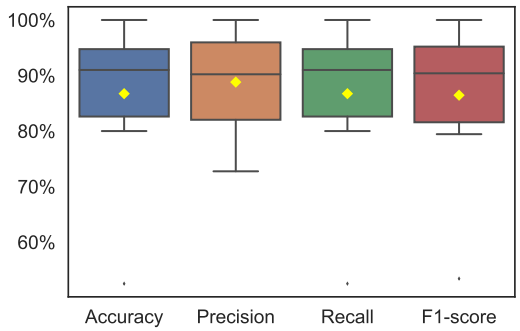


Figure 6.1: Model cross validation scores with 30 features in the feature set

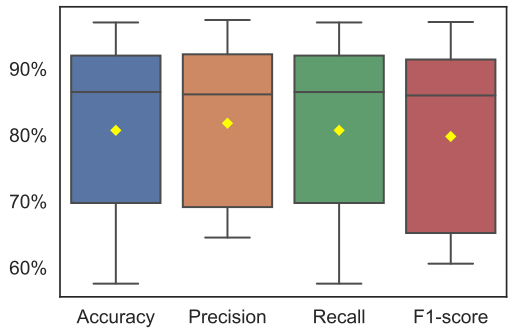


Figure 6.2: Model cross validation scores with 10 features in the feature set

6.1.2 Accuracy of the Framework

The performance of the machine learning model is measured using typical metrics of accuracy, precision, recall, and f1-score. Figure 6.1 shows the 10-fold cross-validation results for the Random Forest classifier with 300 forests and feature set with 30 features presented in Table 4.2. The yellow diamond shows the mean value for each parameter. The classifier achieves an average weighted score of 87% for accuracy, recall, f1-score, and 89% for precision. It means that our SpTRSV framework correctly predicts the best algorithm for 87% of the data set.

To evaluate the effect of reducing the number of features on the prediction model performance, we keep the top 10 features in the feature data set based on their feature scores (score rank in Table 4.2) and perform 10-fold cross-validation of the resultant model. As shown in Figure 6.2, there is a 7-10% drop in performance metrics with the reduced set of features. In addition, there is a wider spread of performance. Considering the possibility of inclusion of new algorithms into the framework in the future, we keep the 30 features listed in Table 4.2.

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

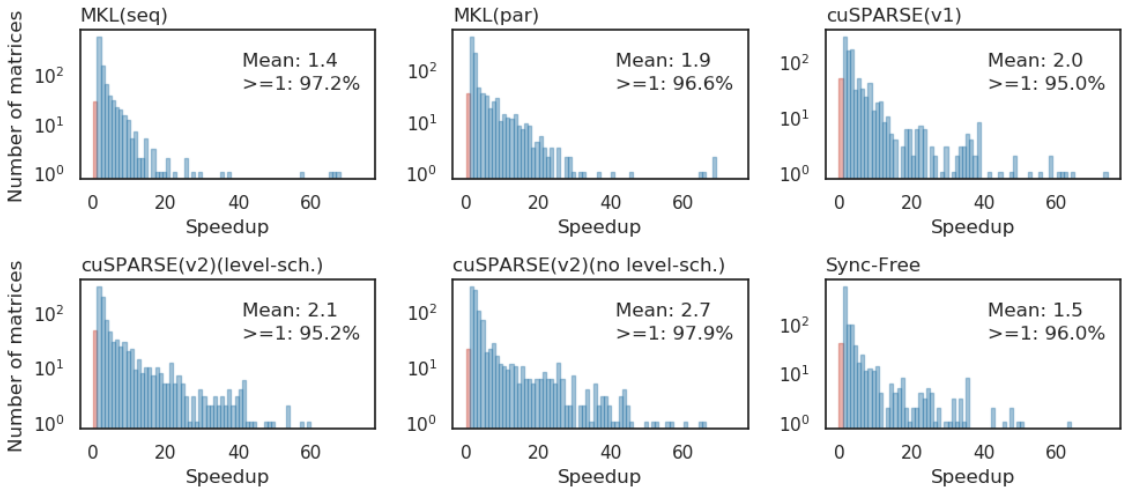


Figure 6.3: Speedup gained by predicted over lazy choice algorithm. >= 1 indicates speedup of greater or equal to 1. (Harmonic) mean refers to average speedup achieved by the framework over the lazy choice. Each bin covers a speedup range e.g. first bin covers speedups between 0-0.99, second one covers speedups between 1-1.99 and so on.

further investigate these reasons in the future.

6.1.3 Speedup Gained by the Framework

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

Figure 6.3 shows the histogram for the speedups achieved by the prediction framework over each of the six implementations for the entire data set. The figure

also shows the percentage when the predicted algorithm achieves equal or better performance than the lazy choice. The results show that the predicted fastest SpTRSV algorithm achieves the same or better performance for greater or equal to 95% of the matrices. Note that for the aggressive programmer, our prediction is 87%, which is presented in Section 6.1.2. We also observe that the speedup obtained by the prediction framework can reach tens or even hundreds for some SpTRSV implementations. Thus, using the framework is highly attractive than an arbitrary algorithmic choice for the SpTRSV execution.

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

6.1.4 Framework Prediction Overhead

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

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

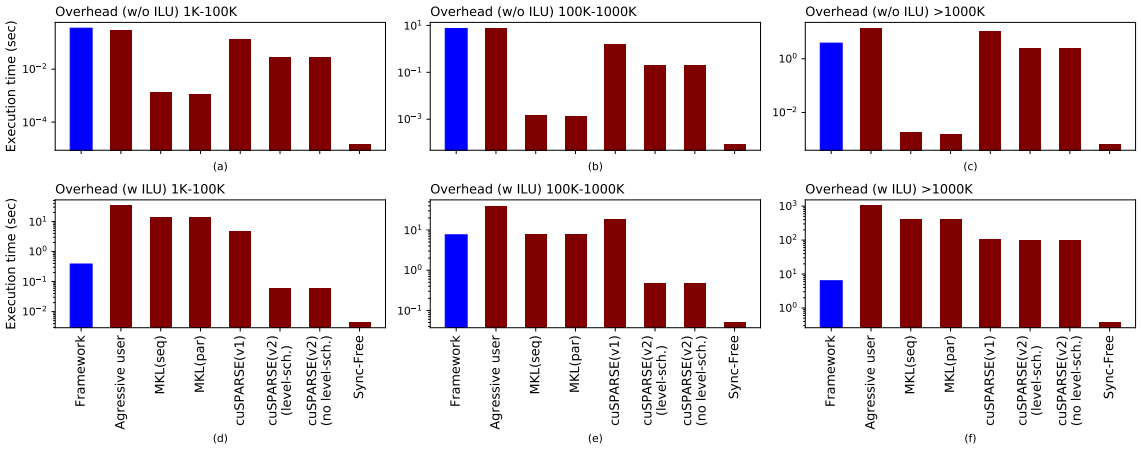


Figure 6.4: Mean overhead of framework versus mean empirical execution time for aggressive and lazy users. 1K-100K, 100K-1000K and >1000K refer to matrix size ranges.

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

$$\textit{Empirical Overhead} = \sum_{i=1}^{N} (A_i + 10 * (TS)_i) \tag{6.1}$$

where A_i and $(TS)_i$ are the matrix analysis phase and single SpTRSV iteration times for algorithm i, respectively. The factor 10 in Equation 6.1 refers to the approximate number of SpTRSV executions required to get a stable time estimate of a single SpTRSV iteration. For the lazy user, there is only a matrix analysis phase as the lazy user does not question the suitability of the algorithm.

For overhead analysis, we divide the matrices into three groups based on their sizes (1K-100K, 100K-1000K, >1000K). For each group, we compare the mean time spent by the framework, by the aggressive user to select the fastest algorithm, and by the lazy user to run the analysis phase of their chosen algorithm. We assume, without loss of generality, that each SpTRSV implementation runs its own ILU factorization phase except *cuSPARSE(v2)(no level-sch.)* that can use ILU factorization from *cuSPARSE(v2)(level-sch.)*. In some cases, it might be possible for some implementations to use ILU factorization from another implementation. However, it will generally require extra effort from the programmer and might add its own pro-

cessing overhead (e.g. converting ILU factors from one data structure to another). For the sake of fairness, we provide an overhead comparison with ILU factorization time included (*w ILU*) and excluded (*w/o ILU*) ILU from A_i as well as framework overhead. For *Sync-Free* implementation, extraction of upper and lower triangular parts of the input matrix as ILU factorization time as it does not perform actual ILU factorization [88].

Figure 6.4(a), (b), (c) compare overhead for the three groups of matrices with ILU factorization time excluded. For matrix sizes less than 1000K, the average framework overhead is comparable with the time spent by the aggressive user. For matrix sizes >1000K, the framework overhead is on average 4 times less than the overhead of the aggressive user. Figure 6.4(d), (e), (f) compare overhead for the three groups of matrices with ILU factorization time included. For all matrix sizes, the average overhead of the framework is observed to be considerably less than aggressive user time by factors ranging between 5 (for 100K-1000K range) and 161 (for >1000K range). Overall, considerable time savings can be obtained by using our framework, especially for large matrices.

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

6.2 Evaluation of SpTRSV Split Execution Model

In this section, we present the performance of our our SpTRSV split execution model on two CPU-GPU platforms using matrices from the SuiteSparse Matrix Collection.

Here are the highlights of the findings.

- Section 6.2.2 presents the speedups achieved by our framework versus the best of the MKL, ELMC algorithms for a selected set of matrices. The framework has been shown to achieve >=1 speedup for 91% and 86% of the matrices for machine 1 and machine 2, respectively.

- Section 6.2.3 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 6.2.4 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 6.2.5 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 6.2.6 compares the framework performance against MKL, ELMC, and cuSAPRSE(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.

6.2.1 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

Table 6.3: Specifications of the CPU machines used for the evaluation

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)

CPU with NVIDIA GTX1080 Ti GPU (machine 2). The specifications of these machines are listed in Tables 6.3 and 6.4. 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 5.5.2) 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

Table 6.4: Specifications of the GPUs used for the evaluation

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

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) [18] 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 [19]. 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.

6.2.2 Speedup against Best CPU/GPU Algorithm

Table 6.5 shows the breakdown of the matrices achieving speedups <1 or >=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

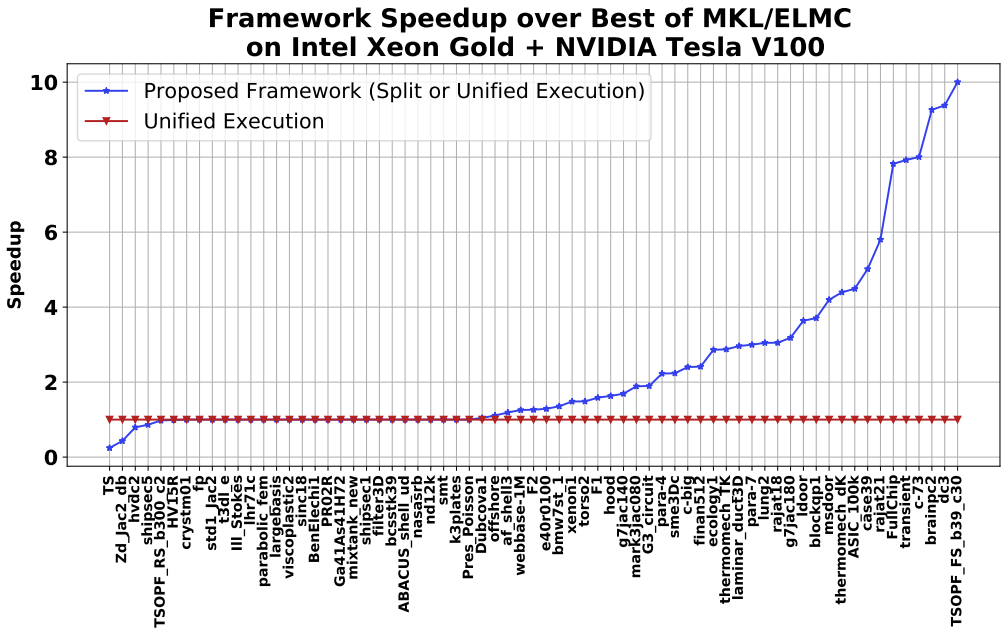


Figure 6.5: Performance of proposed framework on Intel Xeon Gold + NVIDIA Tesla V100 machine for the 67-matrix subset. For each matrix, the speedup is calculated over the best of the MKL(seq) or ELMC algorithms (unit speedup line) for solving the lower triangular system.

selected for 16 and 8 matrices, respectively. 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 6.5 shows the speedups achieved by the framework over the unified Sp-TRSV execution using the best of ELMC and MKL(seq) algorithms on machine 1. The speedups range between 1.04× and 10×. The top five matrices showing

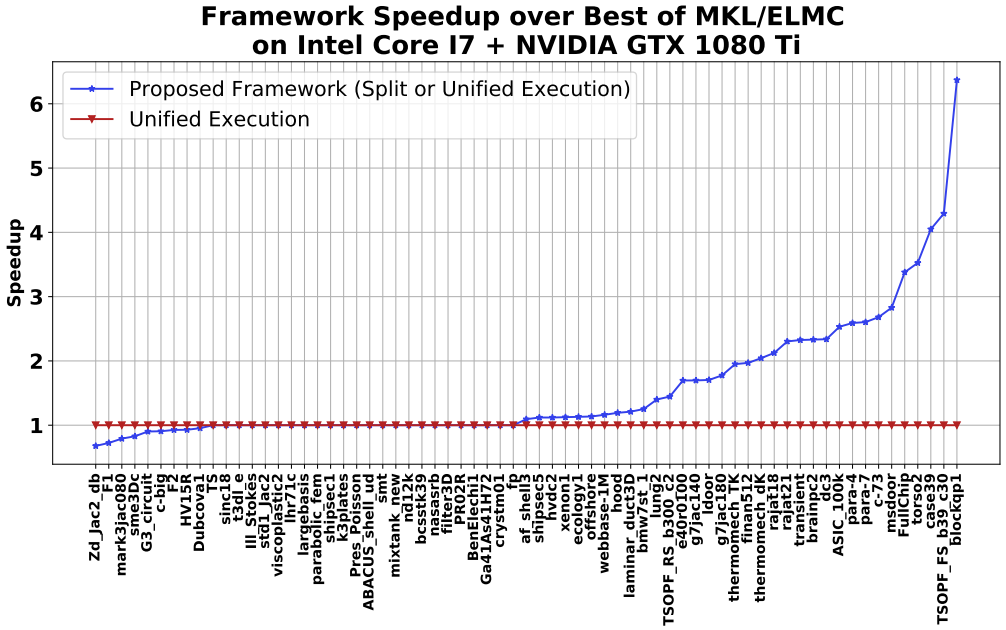


Figure 6.6: Performance of proposed framework on Intel Core I7 + NVIDIA GTX 1080 Ti machine. For each matrix, the speedup is calculated over the best of the MKL(seq) or ELMC algorithms (unit speedup line) for solving lower triangular system.

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 fall between the range 3.04× and 10.0×. The CPU-GPU split policy provides a speedup in the range of $1.04-7.82\times$ for 19 (out of 23 matrices). The speedup ranges between 1.48× and 3.70× for CPU-CPU split execution for 10 matrices. Comparable results hold on machine 2, with speedups ranging between 1.09× and 6.36× (Figure 6.6).

6.2.3 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 6.5) by our execution model. Figure 6.7 shows

Table 6.5: 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**

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 [24]. 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

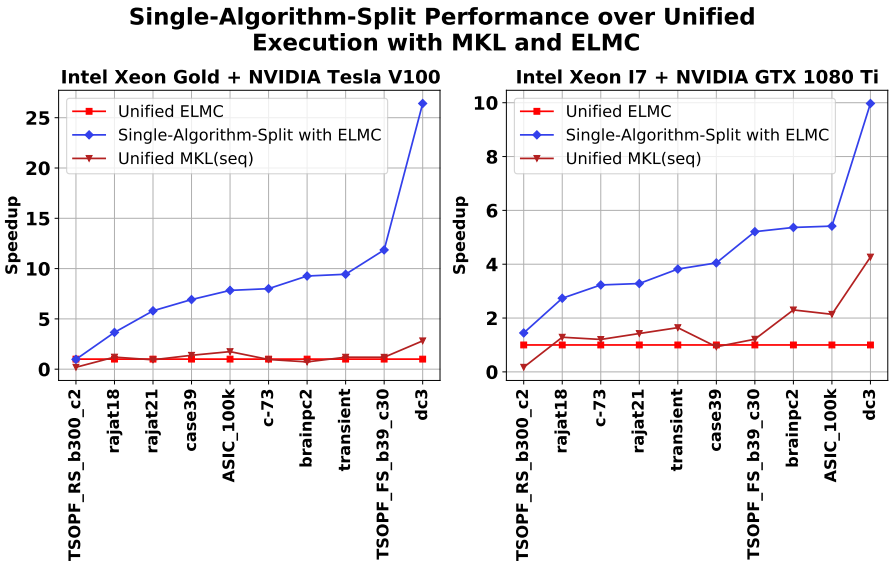


Figure 6.7: Performance comparison of single-algorithm-split execution using ELMC against MKL and unified ELMC for the ten matrices (out of the 67-matrix subset)

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.

6.2.4 Framework Setup Overhead

In this section, we present the framework setup overhead corresponding to Equation 5.2 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:

$$\textit{SpTRSV iterations} = \frac{\textit{Setup overhead (msec)}}{\textit{SpTRSV execution time (msec)}} \tag{6.2}$$

where *Setup overhead* is calculated as per Equation 5.2 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 6.8 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 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.

6.2.5 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 5.5.10, intermediate results need to be communicated between the host and device for such cases. Figure 6.9 shows the execution overhead for the 23 matrices (Table 6.5) 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

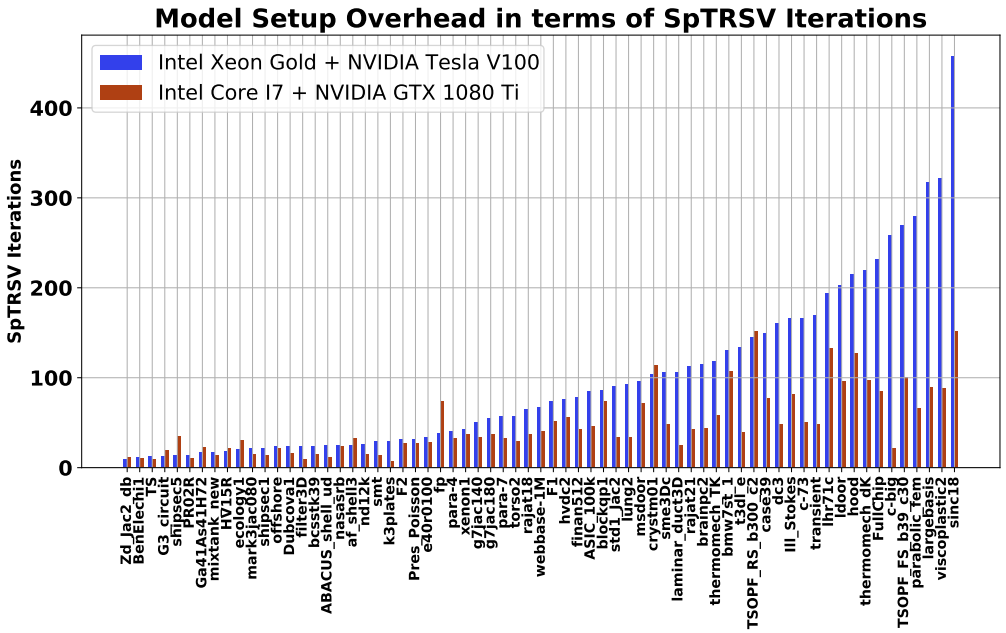


Figure 6.8: Number of SpTRSV iterations required to amortize framework setup overhead

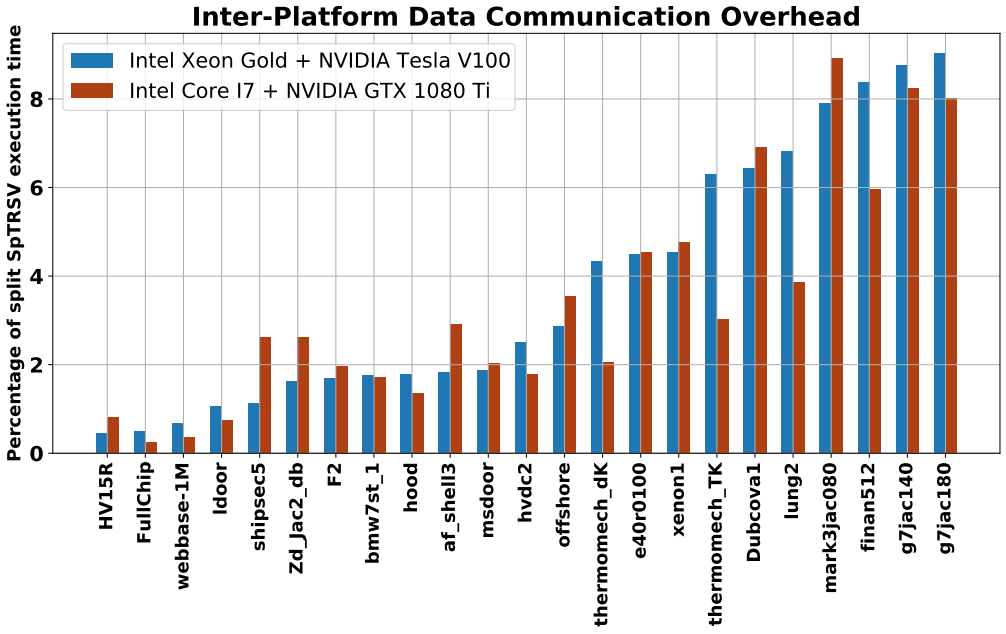


Figure 6.9: Inter-platform data transfer overhead for CPU-GPU split cases. The overhead is shown as a percentage of the total split execution time for a single iteration.

Table 6.6: 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

with a theoretical peak bandwidth of 32 GB/s. To reduce this overhead, we allocate corresponding host arrays in pinned memory [89]. 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.

6.2.6 Speedup against MKL, ELMC, and cuSPARSE

Figures 6.10 shows speedup distribution over MKL, ELMC, and cuSPARSE (v2) on machine 1 for the test set of 327 matrices. Figures 6.10(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 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× over ELMC and cuSPARSE (v2) at first, stays above 1× for most of the matrices, finally converging to 1× as

the matrices get larger and larger. As shown in Figure 6.11, comparable results are observed for machine 2.

Table 6.6 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 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 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.

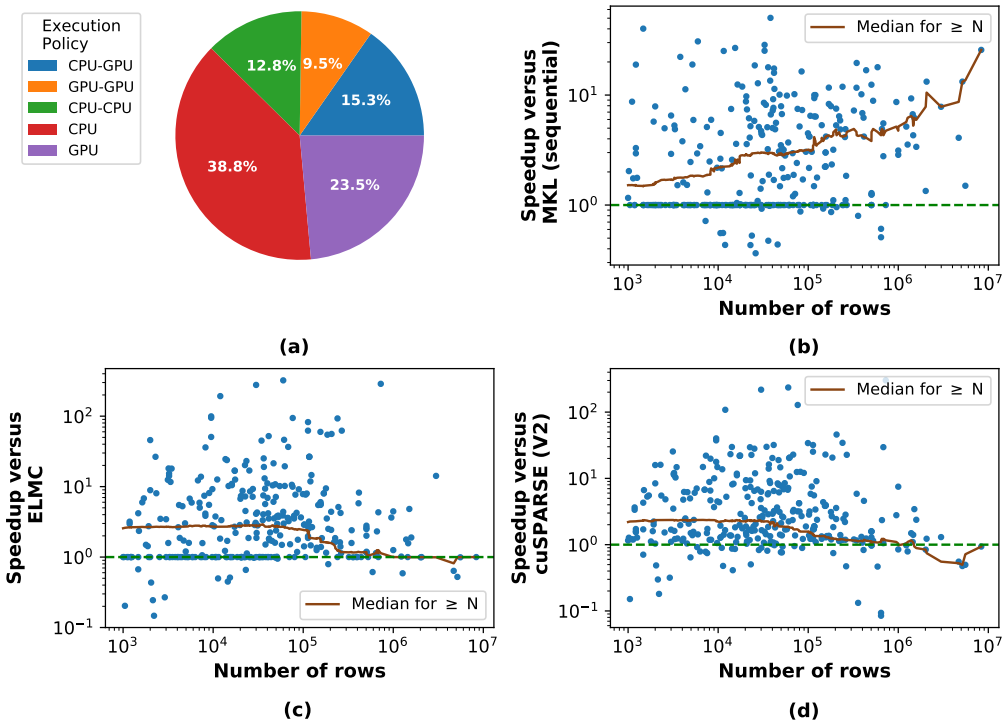


Figure 6.10: Framework performance for the test set of 327 matrices on Intel Xeon Gold + NVIDIA Tesla V100 machine. (a) Distribution of execution policy for the test set. Each slice of the pie shows percentage of matrices chosen for the corresponding execution policy. Speedup achieved by the framework 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.

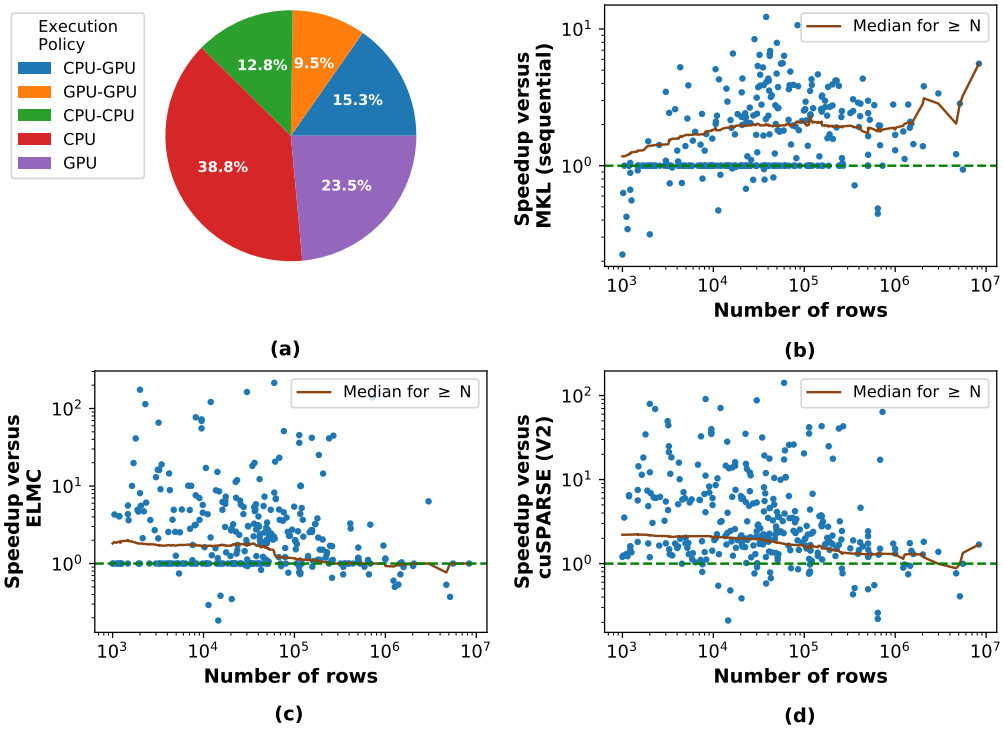


Figure 6.11: Performance of proposed framework for 327 matrices on Intel Core I7 + NVIDIA GTX 1080 Ti machine. (a) Distribution of execution policy for the matrix dataset. Each slice of the pie shows percentage of matrices chosen for the corresponding execution policy. Speedup achieved by the framework 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 represents speedup, y-axis is number of matrix rows, each dot represents a matrix. At each point N, red line shows median speedup for matrices with at least N rows.

Chapter 7

CONCLUSION AND FUTURE WORK

Sparse triangular solve is an important and often the most time-consuming computational kernel in many scientific and numerical computing applications. Better SpTRSV performance could mean an overall performance boost for these applications. There are many SpTRSV algorithms and implementations available in literature for both CPU and GPU platforms. Unfortunately, there is no single algorithm or execution platform that can guarantee best performance for all input matrices.

In this dissertation, we propose tools and techniques to extract best SpTRSV performance for a given input matrix on modern CPU-GPU heterogeneous computing systems. The main motivation of this work comes from the fact that, depending on the sparsity pattern of the input matrix, the CPUs or GPUs may outperform each other. To extract best SpTRSV performance, it is therefore important to decide whether to use CPU or GPU or both. Towards this end, this dissertation makes contributions in the form of a SpTRSV prediction framework and an SpTRSV split execution model.

In our SpTRSV prediction framework, we propose a machine learning-based method for predicting the fastest SpTRSV implementation for a given input matrix on CPU-GPU heterogeneous systems. The framework is trained with a carefully selected features of sparse matrices with an extensible set square, real matrices and multiple CPU and GPU algorithms. The framework is fully automatic and extensible with new SpTRSV algorithms as they become available. The framework has been shown to achieve considerable prediction accuracy (87%) and achieves average speedups (harmonic mean) in the range 1.4-2.7× over the scenario where the programmer always chooses the same algorithm for every input matrix.

Th SpTRSV split execution model is capable of automatically dividing and ex-

ecuting SpTRSV computation as two sub-SpTRSV systems, one suitable for less parallel while the other for highly parallel execution. Thus, the model may utilize two algorithms of 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 for CPUs, 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. The split execution model is implemented as C++/CUDA library with multiple CPU and GPU SpTRSV algorithms. Experimental results on two modern CPU-GPU machines with different characteristics show considerable speedups for a matrix test set selected from the SuiteSparse matrix collection.

While the work presented in this dissertation focuses on a single CPU and GPU pair, we are also currently working on developing multi-GPU SpTRSV split execution model. This model will be integrated into our existing SpTRSV split execution model in the future. The updated SpTRSV split execution model is, therefore, expected to support split SpTRSV execution on single node with a single CPU and multiple GPUs. In addition, we are also planning to investigate SpTRSV implementation on modern Intelligent Processing Units (IPUs) from Graphcore.

BIBLIOGRAPHY

[1] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu," tech. rep., NVIDIA Tech. Rep. NVR-2011-001, 2011.

[2] R. Li and Y. Saad, "Gpu-accelerated preconditioned iterative linear solvers," *The Journal of Supercomputing*, vol. 63, pp. 443–466, Feb 2013.

[3] B. Skuse, "the third pillar," *Physics World*, vol. 32, pp. 40–43, Mar 2019.

[4] G. Golub and J. Ortega, *Scientific Computing and Differential Equations: An Introduction to Numerical Methods*. Elsevier Science, 2014.

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

[6] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. D. Vorst, "Templates for the solution of linear systems: Building blocks for iterative methods," 1994.

[7] T. A. Davis, *Direct methods for sparse linear systems*, vol. 2. Siam, 2006.

[8] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*. Oxford University Press, 2017.

[9] Y. Saad and M. H. Schultz, "Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM J. Sci. Statist. Comput.*, vol. 7, no. 3, pp. 856–869, 1986.

[10] R. W. Vuduc, *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley, 2003.

[11] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc, "Autotuning in high-performance computing applications," *Proceedings of the IEEE*, vol. 106, pp. 2068–2083, Nov 2018.

[12] Intel Incorporated, "Intel MKL | Intel Software." https://software.intel.com/content/www/us/en/develop/articles/intel-math-kernel-library-documentation.html, 2017 (accessed March 3, 2021).

[13] 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.

[14] B. Yilmaz, B. Sipahioglu, 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. 188–198, Association for Computing Machinery, 2020.

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

[16] 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.

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

[18] 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.

[19] N. Corporation, "NVIDIA cuSPARSE library." `https://docs.nvidia.com/cuda/cusparse/index.html`, 2019.

[20] R. Li and Y. Saad, "Gpu-accelerated preconditioned iterative linear solvers," *J. Supercomput.*, vol. 63, p. 443–466, Feb. 2013.

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

[22] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," *Journal of Physics: Conference Series*, vol. 16, pp. 521–530, Jan 2005.

[23] E. Dufrechou, P. Ezzatti, and E. S. Quintana-Orti, "Automatic selection of sparse triangular linear system solvers on gpus through machine learning techniques," in *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 41–47, 2019.

[24] 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.

[25] E. Anderson and Y. Saad, "Solving sparse triangular linear systems on parallel computers," *Int. J. High Speed Comput.*, vol. 1, p. 73–95, Apr. 1989.

[26] J. H. Saltz, "Aggregation methods for solving sparse triangular systems on multiprocessors," *SIAM J. Sci. Stat. Comput.*, vol. 11, pp. 123–144, Jan. 1990.

[27] A. George, M. T. Heath, J. Liu, and E. Ng, "Solution of sparse positive definite systems on a shared-memory multiprocessor," *International Journal of Parallel Programming*, vol. 15, pp. 309–325, Aug 1986.

[28] M. Heath and C. Romine, "Parallel solution of triangular systems on distributed-memory multiprocessors," *SIAM Journal on Scientific and Statistical Computing*, vol. 9, no. 3, pp. 558–588, 1988.

[29] S. Eisenstat, M. Heath, C. Henkel, and C. Romine, "Modified cyclic algorithms for solving triangular systems on distributed-memory multiprocessors," *SIAM Journal on Scientific and Statistical Computing*, vol. 9, no. 3, pp. 589–600, 1988.

[30] G. Li and T. F. Coleman, "A parallel triangular solver for a distributed-memory multiprocessor," *SIAM J. Sci. Stat. Comput.*, vol. 9, pp. 485–502, May 1988.

[31] 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.

[32] B. Suchoski, C. Severn, M. Shantharam, and P. Raghavan, "Adapting sparse triangular solution to gpus," in *2012 41st International Conference on Parallel Processing Workshops*, pp. 140–148, Sep. 2012.

[33] T. Iwashita, H. Nakashima, and Y. Takahashi, "Algebraic block multi-color ordering method for parallel multi-threaded sparse triangular solver in iccg method," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 474–483, May 2012.

[34] M. Naumov, P. Castonguay, and J. Cohen, "Parallel graph coloring with applications to the incomplete-lu factorization on the gpu," tech. rep., Nvidia White Paper, 2015.

[35] F. L. Alvarado and R. Schreiber, "Optimal parallel solution of sparse triangular systems," *SIAM Journal on Scientific Computing*, vol. 14, no. 2, pp. 446–460, 1993.

[36] F. L. Alvarado, A. Pothen, and R. Schreiber, "Highly parallel sparse triangular solution," in *Graph Theory and Sparse Matrix Computation* (A. George, J. R. Gilbert, and J. W. H. Liu, eds.), (New York, NY), pp. 141–157, Springer New York, 1993.

[37] E. Chow and A. Patel, "Fine-grained parallel incomplete lu factorization," *SIAM Journal on Scientific Computing*, vol. 37, no. 2, pp. C169–C193, 2015.

[38] H. Anzt, E. Chow, and J. Dongarra, "Iterative sparse triangular solves for preconditioning," in *European Conference on Parallel Processing*, pp. 650–661, Springer, 2015.

[39] 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.

[40] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *Association for Computing Machinery*, vol. 38, Dec. 2011.

[41] J. R. Rice, "The algorithm selection problem," *Advances in Computers*, vol. 15, pp. 65 – 118, 1976.

[42] R. Vuduc, J. W. Demmel, and J. A. Bilmes, "Statistical models for empirical search-based performance tuning," *The Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 65–94, 2004.

[43] H. Guo, "A bayesian approach for automatic algorithm selection," in *Proceedings of the International Conference on Artificial Intelligence, Mexico*, pp. 1–5, 2003.

[44] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the IEEE Conference on Supercomputing*, SC '98, pp. 1–27, 1998.

[45] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: A language and compiler for algorithmic choice," *SIGPLAN Not.*, vol. 44, pp. 38–49, June 2009.

[46] K. Fatahalian and et al., "Sequoia: Programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pp. 4–4, Nov 2006.

[47] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro, "Nitro: A framework for adaptive code variant tuning," in *IEEE 28th IPDPS 2014*, pp. 501–512, IEEE, 2014.

[48] H. Klie, H. Sudan, R. Li, and Y. Saad, "Exploiting capabilities of many core platforms in reservoir simulation," in *Society of Petroleum Engineers - SPE Reservoir Simulation Symposium 2011*, Society of Petroleum Engineers - SPE Reservoir Simulation Symposium 2011, pp. 264–275, 6 2011.

[49] E. Dufrechou and 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)*, pp. 196–203, 2018.

[50] P. Motter, K. Sood, E. Jessup, and N. Boyana, "Lighthouse: an automated solver selection tool," in *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, pp. 16–24, 2015.

[51] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing* (E. Arge, A. M. Bruaset, and H. P. Langtangen, eds.), pp. 163–202, Birkhäuser Press, 1997.

[52] S. Balay and et al., "PETSc users manual," Tech. Rep. ANL-95/11 - Revision 3.11, Argonne National Laboratory, 2019.

[53] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, "PETSc Web page," 2019.

[54] M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams, "An Overview of Trilinos," Tech. Rep. SAND2003-2927, Sandia National Laboratories, 2003.

[55] P. Motter, *Hardware Awareness for the Selection of Optimal Iterative Linear Solvers*. PhD thesis, University of Colorado at Boulder, 2017.

[56] V. Eijkhout and E. Fuentes, "A standard and software for numerical metadata," *ACM Trans. Math. Softw.*, vol. 35, pp. 25:1–25:20, Feb. 2009.

[57] 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.

[58] G. Bernabé, J. Cuenca, L.-P. García, and D. Giménez, "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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

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

[69] 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.

[70] 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.

[71] 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. 1–12, Association for Computing Machinery, 2013.

[72] 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.

[73] 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.

[74] 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.

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

[76] K. Matam, S. Bharadwaj, and K. Kothapalli, "Sparse matrix matrix multiplication on hybrid cpu+gpu platforms," *Proceedings of the High Performance Computing Conference (HiPC'12)*, 2012.

[77] 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.

[78] 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.

[79] 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.

[80] 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, p. e4187, 2017. e4187 cpe.4187.

[81] T. Cormen, C. Leiserson, Rivest, R.L., and C. Stein, *Introduction To Algorithms*. MIT Press, 2001.

[82] N. Bell and J. Hoberock, *Thrust: A Productivity-Oriented Library for CUDA*. Boston: Morgan Kaufmann, 2012.

[83] F. Pedregosa and et al., "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[84] M. Köhler, "libufget - the uf sparse collection c interface," Sept. 2017.

[85] 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.

[86] 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.

[87] N. Bell and M. Garl, "Efficient sparse matrix-vector multiplication on cuda," tech. rep., NVIDIA Corporation, 2008.

[88] W. Liu and et al., "Benchmark SpTRSM using CSC," September 2017.

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