This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3006353, IEEE Access

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

`Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.Doi Number

On the Efficiency of Supernodal Factorization in Interior-Point Method using CPU-GPU Collaboration

Usman Ali Shah[1,2], Suhail Yousaf[1,3], Iftikhar Ahmad[1], and Muhammad Ovais Ahmad*[4]

[1]Department of Computer Science and Information Technology, University of Engineering and Technology, Peshawar, Pakistan
[2]Department of Computer Science, Islamia College, Peshawar, Pakistan
[3]Intelligent Systems Design Lab, National Center of Artificial Intelligence, University of Engineering and Technology, Peshawar, Pakistan
[4]Department of Mathematics and Computer Science, Karlstad University, Sweden

*Corresponding author: Muhammad Ovais Ahmad (e-mail: ovais.ahmad@kau.se).

ABSTRACT Primal-dual interior-point method (PDIPM) is the most efficient technique for solving sparse linear programming (LP) problems. Despite its efficiency, PDIPM remains a compute-intensive algorithm. Fortunately, graphics processing units (GPUs) have the potential to meet this requirement. However, their peculiar architecture entails a positive relationship between problem density and speedup, conversely implying a limited affinity of GPUs for problem sparsity. To overcome this difficulty, the state-of-the-art hybrid (CPU-GPU) implementation of PDIPM exploits presence of supernodes in sparse matrices during factorization. Supernodes are groups of similar columns that can be treated as dense submatrices. Factorization method used in the state-of-the-art solver performs only selected operations related to large supernodes on GPU. This method is known to underutilize GPU's computational power while increasing CPU-GPU communication overhead. These shortcomings encouraged us to adapt another factorization method, which processes sets of related supernodes on GPU, and introduce it to the PDIPM implementation of a popular open-source solver. Our adaptation enabled the factorization method to better mitigate the effects of round-off errors accumulated over multiple iterations of PDIPM. To augment performance gains, we also used an efficient CPU-based matrix multiplication method. When tested for a set of well-known sparse problems, the adapted solver showed average speed-ups of approximately 55X, 1.14X and 1.05X over the open-source solver's original version, the state-of-the-art solver, and a highly optimized proprietary solver known as CPLEX, respectively. These results strongly indicate that our proposed hybrid approach can lead to significant performance gains for solving large sparse problems.

INDEX TERMS Sparse matrices, GPU, GPGPU, linear programming, primal-dual interior-point method.

I. INTRODUCTION

Linear programming (LP) is the simplest optimization technique, which is widely used in fields as diverse as scientific research, economics and industry [1]. LP problem solvers also serve as drivers for solving nonlinear and mixed-integer linear programming problems [2], which are frequently encountered in highly active research areas like artificial intelligence, cloud computing and cryptography [3]–[12]. Fundamentally, there are two types of methods used for solving large LP problems, namely simplex method and interior-point method (IPM) [13]. Algorithms based on simplex method remain popular for solving LP problems [14]. However, the most widely used variant of IPM, called primal-dual interior-point method (PDIPM) [13], is generally preferred for solving more widely-occurring sparse problems [15].

The process of solving a large LP problem using PDIPM involves compute-intensive operations like matrix-matrix multiplication, matrix-vector multiplication and solving systems of normal equations. Cost-effective availability of immense computational power in graphics processing units (GPUs) has led to their widespread use as general-purpose processors for efficiently performing compute-intensive operations [16]. This general-purpose computing on GPU (GPGPU) approach has also been followed to parallelize the compute-intensive operations in dense versions of both simplex method and PDIPM [17], [18]. It is important to highlight that memory organization of GPUs suites efficient execution of dense linear algebra operations better as

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3006353, IEEE Access

compared to their sparse counterparts [19], which apparently diminishes the utility of these devices for solving sparse problems using PDIPM. Nevertheless, there are some GPU-friendly operations even in sparse implementations of PDIPM. These operations have already been leveraged to develop GPU implementations of PDIPM [20]–[22].

PDIPM is an iterative algorithm. During each iteration, it solves a system of normal equations defined by a symmetric positive-definite (SPD) matrix twice for different right-hand side (RHS) vectors [21]. Conjugate-gradient method (CGM) and Cholesky factorization (CF) are the two main methods used for solving systems of normal equations. CGM follows an iterative approach toward finding an approximation of the solution, where the number of iterations is positively related to the accuracy of the solution returned. This method becomes an effective option in case accuracy requirements for the solutions returned can be relaxed [21]. The entire CGM needs to be repeated even for the same SPD matrix if the RHS vector is changed. Its potential for GPU implementation in the context of solving sparse systems of equations lies in its need to repeatedly perform matrix-vector multiplications. However, its inability to return exact solutions may adversely affect the ability of PDIPM to converge quickly [22]. As opposed to CGM, CF can be used to obtain exact solutions. It decomposes SPD matrices into triangular factors. These factors can subsequently be used to find the solutions for different RHS vectors [23]. This ability to reuse the factors, along with higher accuracy of the solutions returned, makes CF a better choice for implementing PDIPM.

Unfortunately, potential of sparse CF for efficient GPU implementation is not immediately apparent. Nevertheless, it has previously been discovered that presence of structures called supernodes in many real-world SPD matrices can be leveraged for efficient GPU implementation of sparse CF. A supernode is a set of columns with (nearly) identical nonzero pattern, which can be treated as a dense submatrix. The established supernodal CF method, subsequently referred to as root CF (RCF), has purely-CPU as well as hybrid (CPU-GPU) implementations [24], [25]. Hybrid RCF performs only two of the three so-called block operations involved in the processing of large supernodes on GPU. This strategy of performing only some operations related to large supernodes on GPU, while executing all other operations on CPU, increases delays in launching GPU routines and CPU-GPU communication overhead. To reduce these losses in efficiency, Rennich *et al.* [19] later proposed another supernodal CF algorithm, which is subsequently referred to as hybrid subtree CF (SCF). It transfers a set, consisting of a large supernode and all its *related* supernodes, to GPU as a batch. All the block operations on each of these supernodes are performed on GPU, while calls to some GPU routines processing smaller supernodes are further batched. This reduces waiting times involved in the processing of supernodes on GPU while improving its utilization. In addition, hybrid SCF uses GPU streams to overlap the execution of some operations.

During literature review, we came across a hybrid implementation of PDIPM as early as 2008 reported by Jung and O'Leary [18]. However, their implementation focused only on dense LP problems. Smith *et al.* [20] later developed GPU-friendly sparse matrix-vector multiplication (SPMV) kernels to accelerate the process of solving systems of normal equations using CGM. Gade-Nielsen *et al.* [21] also contributed towards the development of hybrid implementations of PDIPM. They focused on a specific class of problems arising in the field of model predictive control. They *tried* to use hybrid RCF implementation of a popular library known as CHOLMOD [24], [26], but admitted that they were unable to get it working, which forced them to exclusively use CGM for solving systems of normal equations. More recently, Maggioni [22] developed a hybrid implementation of PDIPM for sparse real-world problems. His solver used a combination of CHOLMOD's hybrid RCF implementation and CGM for solving systems of normal equations. CGM was used during the initial iterations, whereas CHOLMOD's hybrid RCF was used during later iterations when accuracy of normal equations' solution is critical for converging to LP problem's solution. Maggioni also developed GPU-based SPMV kernels to accelerate CGM.

Our survey of existing hybrid implementations of PDIPM revealed that hybrid SCF proposed by Rennich *et al.*, available in CHOLMOD beta v. 4.6.0 since 2016 [26], has not been used previously. Existing solvers either use variants of CGM exclusively or use them in combination with hybrid RCF implementation of CHOLMOD's release version. This encouraged us to adapt PDIPM implementation of a popular open-source solver, known as GNU linear programming kit (GLPK) [15], [27], by replacing its simple (non-supernodal) CF with CHOLMOD beta version's hybrid SCF. We made changes to the new CF algorithm that made it compatible with GLPK's PDIPM implementation and ensured that accumulation of round-off errors did not cause premature termination of PDIPM. The adapted CF version was also numerically more stable. We also replaced calls to GLPK's original sparse matrix-matrix multiplication (SPMM) routine with calls to the corresponding routine available in CHOLMOD [28], which exploits the sparsity of input matrices more extensively to reduce the amount of processing required.

The introduction of our modified version of hybrid SCF and calls to the new SPMM routine resulted in significant performance gains for the adapted solver. During our experiments, we used the same set of test problems that was used previously to test the performance of the state-of-the-art solver. Our results show that the proposed solver achieved average speed-ups of around 55.458X, 1.136X and 1.053X over GLPK's original version, the state-of-the-art solver and IBM's highly optimized solver known as CPLEX [29], respectively.

The rest of this paper is organized as follows. In Section II, we introduce CF and the process of solving LP problems using

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3006353, IEEE Access

PDIPM. In Section III, we discuss supernodal CF variants used in different solvers considered in this paper. In Section IV, we provide implementation details related to our proposed solver. In Section V, we describe the experimental setup. In Section VI, we discuss performance gains achieved by our proposed solver. We conclude the paper in Section VII by summarizing our work and possibilities for its future extension.

II. BACKGROUND

A. CF BASICS

CF is used to decompose an SPD matrix as a product LL^T, where L and L^T are lower and upper triangular matrices, respectively [23]. Consequently, only one matrix L is sufficient to describe both the factors. Algorithm 1 shows one of the simplest forms of CF. This algorithm runs in $O(n^3)$ time but this high cost can be reduced by exploiting symmetric nature of SPD matrices. Moreover, if the same SPD matrix is used to solve for more than one RHS vectors, then the lower time complexity ($O(n^2)$) of these solutions further compensates for the high cost of factorization [22]. Further details related to CF are provided later in Section III.

Algorithm 1. Simple left looking CF

Procedure: *choleskySimple(S, m)*
Inputs: Matrix, $S_{m\times m} = [s_{i,j}]$
Output: Lower triangular matrix, $L_{m\times m} = [l_{i,j}]$
1. for $i = 1 : m$ do
2. for $j = i : m$ do
3. for $k = 1 : i - 1$ do
4. $s_{j,i} \mathrel{-}= (l_{j,k} \times l_{i,k})$
5. end for
6. end for
7. $l_{i,i} = \sqrt{s_{i,i}}$
8. for $j = i+1 : m$ do
9. $l_{j,i} = s_{j,i}/l_{i,i}$
10. end for
11. end for

Algorithm 2. Block CF

Procedure: *choleskyBlock(S, m, bs)*
Inputs: Matrix, $S_{m\times m} = [s_{i,j}]$; block size, bs
Output: Lower triangular matrix, $L_{m\times m} = [l_{i,j}]$
1. for $i = 1 : bs : m$ do
2. for $j = i : m$ do
3. for $k = 1 : i - 1$ do
4. $S_{j,i:i+bs-1} \mathrel{-}= L_{j,k} \times L^T_{i:i+bs-1,k}$
5. end for
6. end for
7. $L_{i:i+bs-1,i:i+bs-1} = choleskySimple(S_{i:i+bs-1,i:i+bs-1}, bs)$
8. for $j = i + bs : m$ do
9. $L_{j,i:i+bs-1} = S_{j,i:i+bs-1} \times (L^T_{i:i+bs-1,i:i+bs-1})^{-1}$
10. end for
11. end for

B. SOLVING LP PROBLEMS USING PDIPM

LP involves optimization, minimization or maximization, of the value of a given linear function, called objective function, subject to linear constraints [30]. Any LP problem can be brought into a standard minimization form, having only equality constraints by scaling the objective function, and adding appropriate slack, surplus and artificial variables to its constraints. A minimization problem in standard form, having n number of variables and m number of constraints, can be mathematically represented as follows [15], [17].

$$Minimize: z = c^T x + c_0$$
$$s.t: Ax = b, x \geq 0$$

where; x represents the n number of variables in the objective function and constraints, c contains coefficients of the variables in the objective function, c_0 is a constant representing a shift in the value of the objective function; $A_{m\times n} = [a_{ij}]$ is the constraint matrix where $a_{ij}, i = \{1,2,\cdots,m\}, j = \{1,2,\dots,n\}$ represents coefficient of the jth variable in the ith constraint.

The formulation given above shows an LP problem represented in its primal form. An alternative form also exists, called the dual form, which is obtained by transposing the constraint matrix of the primal problem. Some algorithms, called primal-dual algorithms, simultaneously solve both primal and dual forms of an LP problem to get quicker convergence to the solution [15].

Since its development in 1953, simplex method was the only algorithm available for solving LP problems. Work during late 1970s and early 1980s led to the development of a new class of algorithms called IPM. These algorithms have a polynomial worst-case time-complexity, which is better than the exponential worst-case complexity of the simplex method [31]. Conceptually, an IPM begins from any point inside the region bounded by constraints, and moves towards the optimal value (vertex) in a stepwise manner; adjusting magnitude and direction of the movement at each step [13]. The most widely-used variation of IPM, called PDIPM, simultaneously solves both primal and dual versions of the problem for quicker convergence [13]. In this paper, we consider PDIPM as implemented in GLPK, which is based on Mehrotra's predictor-corrector method [32].

GLPK's PDIPM implementation needs to solve a system of normal equations described by the SPD matrix $S_{m\times m} = ADA^T$ twice during each iteration, where D is an $n \times n$ diagonal matrix that changes from one iteration to another. In addition, the same system of normal equations, for $D = I_{n\times n}$, needs to be solved twice during initialization. In an implementation of PDIPM for sparse problems, the nonzero pattern of the matrix S remains constant, which can be used to save the cost of computing its memory requirements during each iteration. During a run of PDIPM over N number of iterations, the system of normal equation needs to be solved $2(N + 1)$ times, whereas the product $S = ADA^T$ and factorization of S need to be computed $N + 1$ times. It is shown in Section IV that these two operations constitute bulk of the overall computational cost of PDIPM.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3006353, IEEE Access

TABLE I
DEPENDENCY AMONG BLOCK OPERATIONS

Block operation	Depends on
1. *update(i, j, k)*	a. *solve(k, i)* b. *solve(k, j)*
2. *solve(i, j)*	a. *factorize(i)* b. *update(i, j, k)*, for $k = 1, 2, 3, .. , i - 1$
3. *factorize(i)*	a. *update(i, i, k)*, for $k = 1, 2, 3, .. , i - 1$

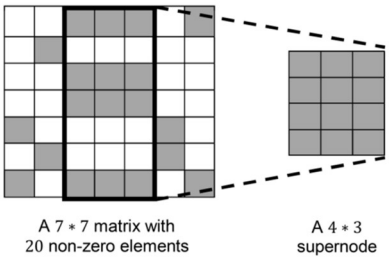

FIGURE 1. An illustration of a supernode present in a sparse matrix.

III. SUPERNODAL CF

We begin this section by introducing supernodal CF. Afterwards, we describe the supernodal CF variants, CPU-based RCF, hybrid RCF and hybrid SCF, as implemented in different versions of CHOLMOD.

A. BASICS

CF, as shown previously in Algorithm 1, is called a left-looking algorithm because it moves from left to right in a column-wise manner to factorize the input matrix lying toward the left of each column. This algorithm provides limited acceleration opportunities only in operations shown in Line 4 and Line 9. However, efficiency of this algorithm for factorizing large matrices can be greatly enhanced by traversing the input matrix in a block-wise manner [22]. Algorithm 2 shows one such method, where a fixed block size is used to process blocks of consecutive columns in a single iteration [23]. In Algorithm 2, three operations, referred to as block operations, are performed iteratively. In the following paragraphs, we briefly describe these operations.

1. *update:* This operation updates the current block of the input submatrix using the previously computed rows of the factor, lying to the left of the current block. The operation *update(i, j, k)* represents the *update* operation applied to j^{th} row in i^{th} block of the input matrix using a product of ith and j^{th} rows in previously computed k^{th} block of the factor. This operation is shown in Line 4 of Algorithm 2.
2. *factorize:* This operation uses Algorithm 1 to factorize a relatively small square submatrix containing the diagonal of current block. The operation *factorize(i)* represents the factorization of the input submatrix containing diagonal of the ith block. This is shown in Line 7 of Algorithm 2.
3. *solve:* This operation computes factorization of the ith block by using the results of *update* and *factorize* operations. The operation *solve(i, j)* denotes factorization of jth row of the ith block, which is performed by using the results of *update* and *factorize* operations performed earlier for the ith block. Line 12 of Algorithm 2 shows this operation.

Hogg [23], in his detailed survey of factorization techniques for symmetric matrices, described two types of acceleration opportunities available while implementing Algorithm 2. First, the block-wise approach of Algorithm 2 can achieve high levels of performance by using cache-friendly linear algebra operations available in highly efficient versions of basic linear algebra subprograms (BLAS) and linear algebra package (LAPACK) libraries [33], [34]. It is important to mention that GPU implementations of these libraries can also be used to achieve high degrees of fine-grained parallelization for processing large dense matrices. Second, Hogg suggested that further performance gains, in terms of coarse-grained parallelization, can be achieved by overlapping the execution of block operations that are not dependent on each other.

Table I shows the mutual dependencies of different block operations. It is customary for any implementation of CF to perform a preliminary *analysis* phase to build an elimination tree representing dependencies of all its block operations [23], [24]. The elimination tree can be used to find the dependency of a block operation on the results obtained by the processing of other blocks as follows. (1) If two blocks lie on different branches of the elimination tree then there is no dependency between them. (2) The processing of an ancestor node depends on the results obtained by processing of its descendants. Workflow represented as an elimination tree helps in determining parallelization possibilities in executing block operations [23].

A further advantage of the block-based approach is that it can be used to efficiently factorize real-world sparse matrices by leveraging the presence of supernodes. Fig. 1 illustrates how columns with the same nonzero pattern can be grouped together to form a supernode. The advantage of using supernodes is that they can be represented as dense blocks in the input matrix. Efficient implementations of dense BLAS and LAPACK routines can subsequently be used to accelerate block operations for supernodes [25]. Furthermore, the dependencies of supernodes upon each other can be determined during the *analysis* phase, which enables execution of independent block operations in parallel [23].

B. RCF

The CPU implementation of CF available in CHOLMOD's release versions, i.e. RCF, exploits acceleration opportunities by using CPU-based BLAS and LAPACK routines. It uses general matrix-matrix multiplication (GEMM) and symmetric rank-k update (SYRK) BLAS routines to perform the *update* block operation, while using the triangular solve (TRSM) BLAS routine to perform the *solve* operation. In addition,

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3006353, IEEE Access

TABLE II
SUMMARY OF CF ALGORITHMS CONSIDERED IN THIS WORK

CF Algorithm	Implemented in	Summary
Simple CF	GLPK	Non-supernodal implementation as illustrated in Algorithm 1
CPU-based RCF	CHOLMOD release v. 5.4.0	Uses CPU-based BLAS and LAPACK
Hybrid RCF	CHOLMOD release v. 5.4.0	Uses CUBLAS and CPU-based LAPACK
Hybrid SCF	CHOLMOD beta v. 4.6.0	Uses CUBLAS, CUSOLVER and batched processing

dense CF (POTRF) LAPACK routine is used to perform the *factorize* block operation [25].

A widely-used GPU-based BLAS library, NVIDIA CUBLAS [35], provides more efficient implementations of BLAS routines as compared to its CPU-based counterparts for processing large dense matrices. Hybrid RCF implementation in CHOLMOD transfers a large supernode to GPU for accelerating its *update* and *solve* block operations using CUBLAS' versions of GEMM, SYRK and TRSM. Unfortunately, it actively processes only one supernode on GPU at a time, which leads to underutilization of GPU's compute units and increased latencies in launching CUBLAS routines while a supernode is transferred to GPU [19]. Once GPU finishes performing *update* and *solve* operations, the results of these operations are written back to host's memory. This division of workload between CPU and GPU for a single supernode increases communication overhead. Moreover, the only type of coarse-grained parallelization is achieved in the form of overlapping block operations' execution with CPU-GPU data transfers.

C. SCF

As mentioned previously in Section I, CHOLMOD beta v. 4.6.0 implements hybrid SCF algorithm proposed by Rennich *et al.*, which takes advantage of both fine-grained and coarse-grained parallelization opportunities. Like hybrid RCF, it uses CUBLAS to accelerate the execution of *update* and *solve* block operations for large supernodes on GPU. However, in contrast to hybrid RCF, it exploits the independence of supernodes belonging to different branches of the elimination tree to achieve coarse-grained parallelization. Under this strategy, once a supernode is selected for processing on GPU, the entire subtree rooted at the selected node is copied to GPU's memory. All the three block operations, including *factorize*, for all these supernodes are performed on GPU. The POTRF routine in a GPU implementation of LAPACK called CUSOLVER is used to perform the *factorize* block operation for large supernodes [36]. Calls to GPU routines for processing smaller supernodes are batched together to reduce kernel-launch latencies. Once the entire subtree has been processed, only then its results are copied back to host's memory. In addition to overlapped execution between CPU and GPU, hybrid SCF also allows for concurrent processing of block operations on GPU using different CUDA streams.

TABLE III
COUNT AND WEIGHTAGE OF MAJOR OPERATIONS IN THE PDIPM IMPLEMENTATION OF GLPK

Operation	Operation count	Operation time vs. total time
Matrix multiplication: $S = ADA^T$	(N + 1) times	63.65%
Factorization of the matrix S	(N + 1) times	36.00%
All other operations combined	-	0.35%

In summary, hybrid SCF significantly reduces overheads in terms of both kernel-launch latencies and CPU-GPU communication, while achieving a better utilization of GPU's computational resources. Rennich *et al.* showed that this method provides significant speed-up over hybrid RCF [19]. When we examined the source code of CHOLMOD release v. 5.4.0 and beta v. 4.6.0, we found out the default criteria to select a (root) supernode for processing on GPU to be; (1) the number of columns in the supernode to be greater than 512, or (2) the number of rows in the supernode to be greater than 256 and its number of columns to be greater than 32 . In the rest of this paper, we refer to a supernode as being GPU-friendly if it meets either of these criteria. CF algorithms considered in this paper are summarized in Table II.

IV. IMPLEMENTATION DETAILS

To make our results reproducible, we selected GLPK's PDIPM implementation as a baseline solver. We decided to adapt it by modifying its CF and SPMM routines. Modification of GLPK's CF implementation also necessitated changing the routine used for solving systems of normal equations. The decision to focus only on CF and SPMM was based on our observation that more than 99% of the overall solution time was contributed by these two operations, as shown in Table III. The weightages shown in Table III were obtained by averaging execution times of different operations in the original GLPK implementation for the selected test problems specified later in Section V. It is obvious that even though SPMM is the most time-consuming operation, any enhancement in terms of CF can also result in significant performance gains.

In the rest of this section, we explain the procedure adopted for introducing CHOLMOD's SPMM implementation to GLPK followed by description of a simple modification that made CPU-based and hybrid versions of RCF compatible with GLPK. Afterwards. we explain modifications introduced to hybrid SCF that allowed us to use it in our proposed GLPK-based solver. To the best of our knowledge, GLPK has previously not been adapted in such a way. We have made the modified CHOLMOD and GLPK source code, along with detailed usage instructions, available publicly [37].

A. ADAPTING GLPK

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3006353, IEEE Access

We began by undertaking the relatively simple task of exploring efficient alternatives to GLPK's original SPMM routine used to compute the product $S = ADA^T$, where, as mentioned previously, D represents a diagonal matrix. If the matrix D is temporarily ignored for ease of illustration, then the following linear combination represents how GLPK's SPMM routine computes each nonzero element (s_{ij}) of the matrix S.

$$s_{ij} = \sum_{k=0}^{n-1} (a_{ik}a_{jk}), a_{jk} \neq 0$$

It is evident that all the terms having their second operand (a_{jk}) equal to zero are eliminated, thus reducing the number of floating-point multiplications. We subsequently learnt that CHOLMOD also provides a CPU implementation of SPMM, proposed by Davis [28], which can performs the same product more efficiently as follows.

$$s_{ij} = \sum_{k=0}^{n-1} (a_{ik}a_{jk}), a_{ik} \neq 0, a_{jk} \neq 0$$

In contrast to GPLK's SPMM routine, this implementation eliminates all the terms in which either of the two operands is equal to zero, which further reduces the number of floating-point operations. It is shown later in Section VI-A that the replacement of GLPK's original SPMM routine with its counterpart from CHOLOMD led to significant overall performance gains for our proposed solver. GLPK uses compressed sparse row (CSR) format to store the matrix A, which is the exact transpose of the compressed sparse column (CSC) format used by CHOLMOD. This relationship greatly simplified conversion of matrices between storage formats used by GLPK and CHOLMOD.

We also considered the option of performing SPMM using popular GPU-based sparse linear algebra libraries like CUSP and CUSPARSE [38], [39]. In addition to these libraries, we came across other works related to GPU implementation of SPMM as well [40]–[42]. However, we preferred CHOLMOD over all these GPU implementations in this work because both hybrid RCF and SCF process only large supernodes on GPU and require the matrix S to be stored in the host's memory.

It was explained in Section III-B and Section III-C how different versions of supernodal CF available in CHOLMOD use CPU and GPU implementations of BLAS and LAPACK to accelerate execution of block operations related to large supernodes. Initially, we introduced these CF methods, without any modification, to GLPK's PDIPM implementation by replacing calls to its original CF-related routines with calls to appropriate routines from different versions of CHOLMOD. Unfortunately, CF implementations of GLPK and CHOLMOD handle the situation differently when matrix S stops being positive-definite because of accumulation of round-off errors over multiple iterations, which is indicated by the occurrence of non-positive diagonal elements ($s_{i,i}$). GLPK handles this situation by replacing Line 7 of Algorithm 1 with the following conditional statement in accordance with an approximation method proposed by Wright [43].

$$l_{i,i} = (s_{i,i} \leq 0.0) \; ? \; DBL_MAX : \sqrt{s_{i,i}}$$

Where, *DBL_MAX* represents a large real number defined in the C header file <*float.h*>, which approximates infinity. In contrast to GLPK's use of Wright's approximation method, CF implementations available in different versions of CHOLMOD consider the occurrence of a non-positive diagonal element to be an error condition and terminate the factorization process. In the next two subsections, we describe how we adapted RCF and SCF to make them compatible with GLPK's PDIPM implementation.

B. ADAPTING CPU-BASED AND HYBRID RCF

As mentioned previously, both CPU-based and hybrid versions of RCF available in CHOLMOD release v. 5.4.0 perform the *factorize* block operation using the POTRF routine of a CPU-based LAPACK implementation available on the system. On detecting a non-positive diagonal element, the default behavior of POTRF routines is to terminate after returning an error. To modify this behavior, we decided to use a LAPACK version that comes bundled with an open-source BLAS implementation called OpenBLAS [44]. We found that OpenBLAS was already referenced as a possible BLAS candidate in the main configuration file of CHOLMOD's parent package called SuiteSparse [26].

We initially introduced the same conditional statement given in the previous subsection to POTRF, which made both versions of RCF compatible with GLPK's PDIPM implementation. However, we observed that even after this modification, the solutions of some test problems suffered from numerical instability. Further experiments revealed that relaxation of the original condition as follows resolved these difficulties.

$$l_{i,i} = (s_{i,i} \leq \epsilon) \; ? \; DBL_MAX : \sqrt{s_{i,i}}$$

Where, ϵ represents a *very* small positive real number. We experimented with different values of ϵ and found $\epsilon = 1\times10^{-14}$ to be a suitable value that ensured convergence to the solution in minimal number of iterations while eliminating numerical instability. We also replaced the original conditional statement of GLPK with the new statement.

C. ADAPTING HYBRID SCF

As opposed to RCF, the modification of hybrid SCF available in CHOLMOD beta v. 4.6.0 was not as straightforward since it uses two different GPU routines for performing the *factorize* block operation:

1. Hybrid SCF calls CUSOLVER's double-precision version of POTRF function named *cusolverDnDpotrf()* to *factorize* large supernodes on GPU [36]. Hybrid SCF does not check for the presence of non-positive diagonal

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3006353, IEEE Access

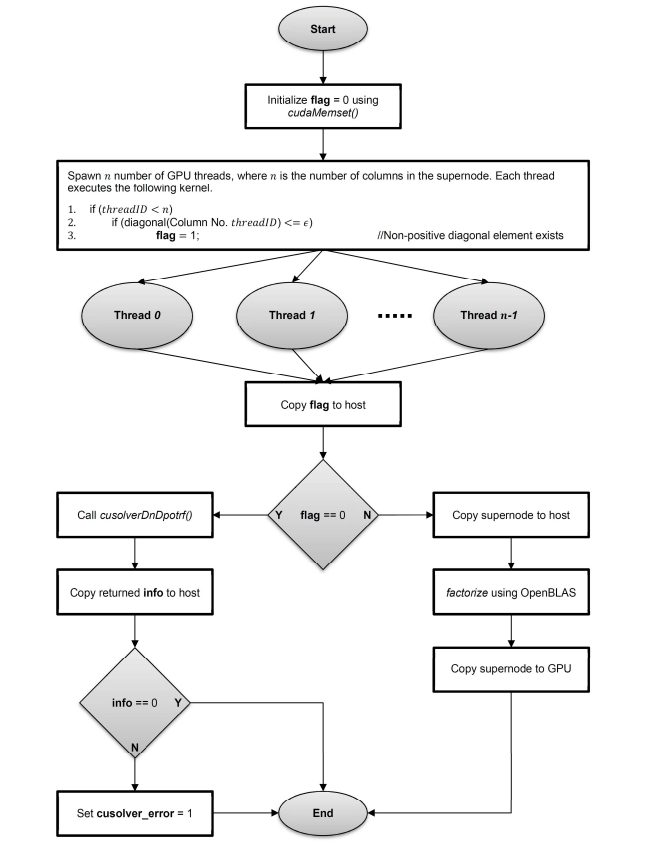

FIGURE 2. Adapted method to perform *factorize* block operation on large supernodes in hybrid SCF.

elements before calling this function. Moreover, it does not check the status returned by the function *cusolverDnDpotrf()* for the occurrence of errors. We could not modify the function *cusolverDnDpotrf()* since CUSOLVER is a proprietary library. Consequently, we developed a new CUDA kernel that determined if a non-positive diagonal element existed in the supernode. Subsequently, the function *cusolverDnDpotrf()* was called only if no non-positive diagonal element was found. Otherwise, the supernode was copied back to the host to be *factorized* using the same POTRF routine that was used by RCF. The status returned by the function *cusolverDnDpotrf()* was also checked to determine if it successfully performed the *factorize* operation. In case of an error, a flag was set for indicating to the calling PDIPM routine that it should switch to using CPU-based RCF for its remaining iterations. However, during our experiments, we found this error condition to be rare, which occurred only once during the final iterations of one test problem. Fig. 2 summarizes these adaptations.

2. Hybrid SCF uses a custom CUDA kernel that performs the *factorize* block operation on relatively small supernodes as a batch. We simply introduced the modified conditional statement, shown in the previous subsection, to this CUDA kernel for handling non-positive diagonal elements.

We also fixed some bugs related to memory management and creation of handles to NVIDIA libraries CUBLAS and CUSOLVER in hybrid SCF implementation, which originally caused PDIPM to terminate abnormally for some test problems. Relevant debugging details are available in the publicly available source code package [37].

TABLE IV
SUMMARY OF LP PROBLEM SOLVERS CONSIDERED IN THIS WORK

Solver	Summary
CPU-BASE	GLPK's original PDIPM implementation used as baseline
CPU-RCF	Modified GLPK implementation that uses our adaptation of CPU-based RCF
HYB-RCF	Modified GLPK implementation that uses our adaptation of hybrid RCF
HYB-SCF	Our proposed GLPK implementation that uses our adaptation of hybrid SCF
CPU-CPLEX	A simple C program that uses CPLEX's PDIPM routine
HYB-MAGGIONI	State-of-the-art hybrid solver proposed by Maggioni

D. GLPK-BASED SOLVERS

To compare the effects of using different CF implementations on the performance of PDIPM, we further modified GLPK's PDIPM implementation that used CHOLMOD's SPMM routine to get three new version by using the three CF adaptations discussed in the previous two subsections. The first version uses our adaptation of CPU-based RCF. In the rest of this paper, we refer to this solver as CPU-RCF. The second version uses our adaptation of hybrid RCF. In the rest of this paper, we refer to this solver as HYB-RCF. The third version, our proposed solver, uses our adaptation of hybrid SCF. In the rest of this paper, we refer to our proposed solver as HYB-SCF. We compared the performance of all these solvers against original PDIPM implementation of GLPK, which is referred to as CPU-BASE in the rest of this paper. These solvers are summarized in Table IV.

E. CPLEX-BASED SOLVER

In addition to the solvers discussed in the previous subsection, we also used a simple C program to invoke CPLEX's PDIPM (barrier method) routine with its default settings. In the rest of this paper, we refer to this program as CPU-CPLEX. Furthermore, we refer to the state-of-the-art hybrid solver, proposed by Maggioni, as HYB-MAGGIONI.

Maggioni reported results comparing the performance of his solver against CPLEX. For an approximate performance comparison between HYB-SCF and Maggioni's solver, we used; (1) results of our experiments, as discussed later in Section VI, comparing the performance of HYB-SCF against CPU-CPLEX, and (2) speed-ups reported by Maggioni for his

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3006353, IEEE Access

TABLE V
PROPERTIES OF CONSTRAINT MATRIX IN TEST PROBLEMS

Problem	Source	m	n	Size = $m \times n$	Number of non-zeros	Problem density
RAIL507	Mittleman	507	63009	31945563	409349	1.2814%
TSPALKO	Meszaros	22002	47235	1039264470	1076903	0.10362%
RAIL2586	Mittleman	2586	920683	2380886238	8008776	0.33638%
RAIL4284	Mittleman	4284	1092610	4680741240	11279748	0.24098%
KARTED	Meszaros	46502	133115	6190113730	1770349	0.0286%
DEGME	Meszaros	185501	659415	122322141915	8127528	0.00664%
TP6	Meszaros	142752	1014301	144793496352	11537419	0.00797%
WATSON2	Mittleman	352013	671861	236503806193	1841028	0.00078%

solver against CPLEX. It is important to mention that Maggioni relaxed the tolerances (relative primal infeasibility, relative dual infeasibility and primal-dual gap) used to determine convergence of the LP problem's solution. However, in the interest of a fairer performance comparison, we did not modify default tolerances in either GLPK-based solvers or CPU-CPLEX.

V. EXPERIMENTAL SETUP

To test the relative performance of different solvers, we selected a set of eight well-known sparse problems, previously used by Maggioni to test the performance of the state-of-the-art hybrid solver [22]. This selection enabled us to fairly compare the performance of HYB-SCF against its most recent competitor (HYB-MAGGIONI). Furthermore, these problems vary in size and density (percentage of nonzero values), which provides an opportunity to study the effects of variations in these parameters on the performance of different solvers. Details of these problems are given in Table V. Except for (probably) one problem, RAIL507, all the problems can be easily categorized as being large LP problems. RAIL507 is also the least sparse problem, having a density of around 1.3%. On the other hand, WATSON2, which is hyper-sparse, is the largest problem in the set.

Table VI compares the hardware we used during our experiments against the hardware used by Maggioni to test the performance of his solver. These hardware specifications indicate that the speed-up reported later in Section VI-B, for our proposed solver (HYB-SCF) over the state-of-the-art solver (HYB-MAGGIONI), is not because of any *peculiar* hardware choices that could have unduly favored GPU over CPU or vice versa. Nevertheless, this speed-up remains the outcome of an *estimated* performance comparison.

TABLE VI
HARDWARE COMPARISON; HYB-SCF VS. HYB-MAGGIONI

Specification	Hardware used by us	Hardware used by Maggioni
CPU model	Intel Core i3-3220	Intel Xeon E5 2650
CPU frequency	3.30 GHz	2.80 GHz
No. of CPU cores	2	8
CPU cache	3 MB	20 MB
Host RAM	16 GB	64 GB
GPU model	GeForce GTX 960	Tesla K40
No. of GPU cores	1024	2880
GPU global memory	4 GB	12 GB

In terms of software, we modified PDIPM implementation of GLPK v. 4.65 as discussed in the previous section. We used compute unified device architecture (CUDA) toolkit v. 9.0 in Ubuntu 17.04 operating system to execute GPU-end routines. We modified LAPACK implementation available in OpenBLAS v. 0.3.6, as discussed in the previous section, to perform *factorize* block operation using CPU [44]. We used IBM ILOG CPLEX Optimization Studio v. 12.6 to implement CPU-CPLEX.

We solved the eight test problems five times using all the GLPK-based solvers, except for CPU-BASE which took unreasonably long (more than four hours) to solve some problems. For each run of these solvers, we measured overall solution time, average per-iteration solution time, average CF time and average SPMM time to compute $S = ADA^T$. We also solved the test problems five times using CPU-CPLEX.

VI. RESULTS

In this section, we first compare the performance of the four GLPK-based solvers; CPU-BASE, CPU-RCF, HYB-RCF and HYB-SCF. Afterwards, we compare HYB-SCF's performance against CPU-CPLEX as well as Maggioni's state-of-the-art hybrid solver.

A. COMPARISON OF GLPK-BASED SOLVERS

Due to a high degree of variation in timing measurements for different GLPK-based solvers, we opted to tabulate these measurements in Table VII and Table VIII. Table VII shows number of iterations and total solution time, whereas Table VIII shows per-iteration time, average CF time and average SPMM time. Speed-ups achieved by the three modified versions of GLPK (CPU-RCF, HYB-RCF and HYB-SCF) over the baseline version (CPU-BASE) in terms of total solution time, average CF time and average SPMM time are shown in Fig. 3a, Fig. 3b and Fig. 3c, respectively.

As mentioned previously, we solved each problem five times using CPU-RCF, HYB-RCF and HYB-SCF. Each run of a solver for a specific problem executed the same number of iterations, except for HYB-SCF's attempts to solve RAIL2586 and RAIL4284. We observed that HYB-SCF executed 109 to 112 iterations for solving RAIL2586 and 92 or 93 iterations for solving RAIL4284. These variations explain the presence of fractional parts in the number of iterations column in Table VII corresponding to HYB-SCF for these two problems. These variations can be attributed to the

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3006353, IEEE Access

TABLE VII
NUMBER OF ITERATIONS AND TOTAL SOLUTION TIME FOR CPU-BASE, CPU-RCF, HYB-RCF AND HYB-SCF

	CPU-BASE		CPU-RCF		HYB-RCF		HYB-SCF	
Problem	Number of iterations	Total solution time (s)	Number of iterations	Total solution time (s)	Number of iterations	Total solution time (s)	Number of iterations	Total solution time (s)
RAIL507	47	4.20	47	1.97	47	3.88	47	3.93
TSPALKO	31	857.25	31	62.91	31	64.81	31	35.43
RAIL2586	110	884.80	109	109.51	109	111.47	110.6	112.86
RAIL4284	94	3710.00	92	212.27	92	214.29	92.2	190.50
KARTED	29	3701.70	29	210.88	29	199.50	29	84.07
DEGME	46	17578.44	46	645.69	46	615.06	46	268.90
TP6	41	21104.06	41	288.14	41	271.70	41	139.48
WATSON2	72	35.35	72	27.64	72	27.62	72	28.10
Average	58.75	5984.47	58.375	194.88	58.375	188.54	58.6	107.91

TABLE VIII
PER-ITERATION, CF AND SPMM TIME FOR CPU-BASE, CPU-RCF, HYB-RCF AND HYB-SCF

	CPU-BASE			CPU-RCF			HYB-RCF			HYB-SCF		
Problem	Per-iteration time (s)	Average CF time (s)	Average SPMM time (s)	Per-iteration time (s)	Average CF time (s)	Average SPMM time (s)	Per-iteration time (s)	Average CF time (s)	Average SPMM time (s)	Per-iteration time (s)	Average CF time (s)	Average SPMM time (s)
RAIL507	0.08282	0.00875	0.06130	0.03619	0.00289	0.02117	0.03519	0.00295	0.02064	0.03981	0.00490	0.02254
TSPALKO	26.76029	24.41397	2.26204	1.93706	1.62705	0.25197	1.93569	1.62657	0.25158	1.02241	0.71109	0.25353
RAIL2586	7.92042	0.75025	6.86826	0.94543	0.08023	0.56435	0.94560	0.08031	0.56468	0.94590	0.07236	0.57246
RAIL4284	38.94828	7.80179	30.69964	2.19271	0.73644	1.00342	2.19131	0.73516	1.00308	1.93487	0.46600	1.01612
KARTED	123.32702	107.68082	15.44823	6.90700	6.44119	0.33029	6.46188	5.99940	0.32967	2.61965	2.15635	0.32919
DEGME	373.90775	192.30118	181.13393	13.58440	12.30164	0.91713	12.86711	11.58162	0.92068	5.52705	4.24233	0.92370
TP6	501.90203	79.66031	422.20675	6.68638	5.35274	0.92492	6.24839	4.91899	0.92467	3.10287	1.76304	0.93741
WATSON2	0.47043	0.24136	0.10833	0.36314	0.18039	0.06348	0.36288	0.18014	0.06341	0.36924	0.18418	0.06454
Average	134.16488	51.60730	82.34856	4.08154	3.34032	0.50959	3.88100	3.14064	0.50980	1.94523	1.20003	0.51494

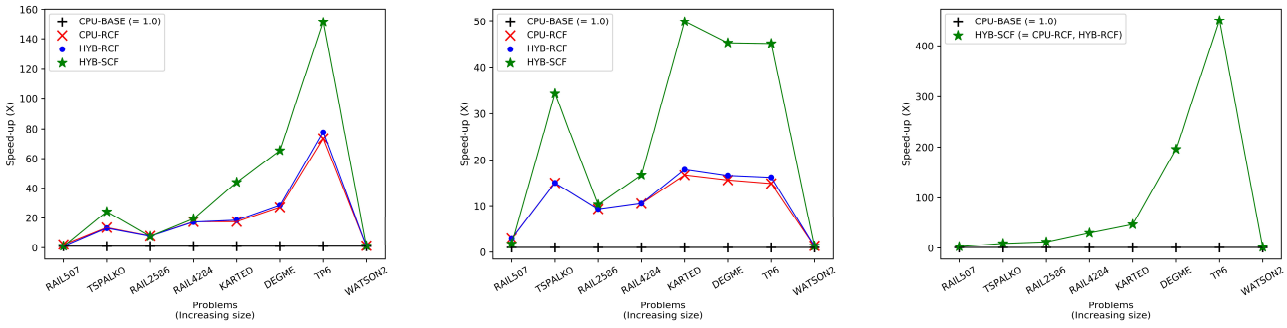

FIGURE. 3. Performance comparison of CPU-RCF, HYB-RCF and HYB-SCF against CPU-BASE. (a) Overall speed-up obtained by dividing CPU-BASE's solution time over each solver's solution time. (b) CF speed-up obtained by dividing CPU-BASE's CF time over each solver's CF time. (c) SPMM speed-up obtained by dividing CPU-BASE's SPMM time over HYB-SCF's average SPMM time.

relatively large number of iterations required to solve these problems. This increases the chances of differences in intermediate results of the same set of floating-point operations while performing POTRF using GPU, which is caused by an absence of any predefined order in the execution of GPU threads.

Fig. 3a and Table VII show that HYB-SCF outperformed CPU-BASE for all the problems. Fig. 3a also shows that HYB-SCF's speed-up generally improved with an increase in the size of problems' constraint matrices ($m \times n$). A lack of significant speed-up for the largest problem, WATSON2, is because of its hyper-sparse nature. As shown in Table IX, the

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3006353, IEEE Access

TABLE IX
PROPERTIES OF MATRIX S IN TEST PROBLEM, PROBLEMS SORTED BY THE RATIO τ

Problem	Number of rows and columns	Number of non-zeros (NNZ)	Number of dense columns (10% or more non-zeros)	Number of supernodes	Number of GPU-friendly supernodes	Ratio τ: (%age of GPU-friendly supernodes / %age of dense columns) [1]	Cumulative size of GPU-friendly supernodes
WATSON2	352013	1871146	0	0	0	0	0
RAIL507	507	25309	213	38	1	0.062639	112896
RAIL4284	4284	1091677	854	231	3	0.065148	11518719
RAIL2586	2586	237092	223	207	4	0.224085	2076114
TP6	142752	4100789	1	3480	12	492.2483	39904761
DEGME	185501	9613196	0	2890	17	5.9×10^{13}	64806897
KARTED	46502	4961208	0	479	205	4.3×10^{15}	58445085
TSPALKO	22002	4096671	0	126	68	5.4×10^{15}	27117729

[1] For avoiding divide-by-zero errors, we added 1×10^{-14} to denominator during the calculation of this ratio.

matrix S in this problem does not contain supernodes, which eliminates any possibility of gaining speed-up by using supernodal CF.

In terms of HYB-SCF's performance gains over the solvers that use RCF, i.e. CPU-RCF and HYB-RCF, it clearly outperformed them for five of the eight test problems. Among the remaining three problems, matrix S in RAIL507 was obviously not large enough to contain any substantial supernodes suitable for processing on GPU. Similarly, the hyper-sparse nature of WATSON2 meant that its matrix S also did not have any supernodes for supernodal factorization to achieve any speed-up, whether on CPU or on GPU. Lastly, RAIL2586 is interesting in the sense that it is the second-least sparse problem among all the test problems. As shown in Table IX, the matrix S in this problem has only four supernodes large enough to be suitable for processing on GPU. Furthermore, the cumulative size of these four supernodes is the smallest among all the problems having supernodes, except for the *very small* matrix S in problem RAIL507. This relatively small number and size of GPU-friendly supernodes in RAIL2586 inhibited the ability of both the hybrid solvers, HYB-RCF and HYB-SCF, to gain performance over the CPU-based solver (CPU-RCF). However, it is encouraging to note that even for the hyper-sparse problem, WATSON2, the performance of HYB-SCF did not significantly drop below that of the other solvers.

HYB-SCF achieved overall average speed-ups of 55.458X, 1.806X and 1.747X over CPU-BASE, CPU-RCF and HYB-RCF, respectively. It achieved maximum speed-ups of 151.301X, 2.508X and 2.373X over CPU-BASE, CPU-RCF and HYB-RCF, respectively. It is interesting to note that, as opposed to HYB-SCF, HYB-RCF could manage to achieve discernible speed-ups over CPU-RCF for only three of the test problems; KARTED, DEGME and TP6. This (at least partially) explains the preference given to CGM over CF during previous attempts to use GPU technology for accelerating PDIPM [21], [22].

Fig. 3b shows the speed-up curves of CF in the four GLPK-based solvers, which follow reasonably similar trends as shown in Fig. 3a for overall speed-up. It demonstrates that hybrid SCF had a significant impact in enhancing the overall performance of its corresponding solver; HYB-SCF. We calculated the average contribution of CF towards overall reduction in solution time achieved by HYB-SCF against CPU-BASE to be around 38.12% [37]. Fig. 3c shows the speed-up achieved by CHOLMOD's SPMM routine used in the three adapted solvers (CPU-RCF, HYB-RCF and HYB-SCF) against the corresponding routine used in CPU-BASE. It is clear that except for a relatively small speed-up for WATSON2, which is a hyper-sparse problem, CHOLMOD's SPMM routine provided significant speed-ups over the corresponding routine of GLPK for all other test problems. We calculated the average contribution of SPMM towards overall reduction in solution time achieved by HYB-SCF against CPU-BASE to be around 61.88% [37]. We subsequently discovered that if only GLPK's SPMM routine was replaced with the corresponding routine of CHOLMOD, without replacing GLPK's original CF routine, then average speed-up achieved by the proposed solver over CPU-BASE would drop from 55.458X to 2.711X.

B. HYB-SCF VS. CPU-CPLEX AND HYB-MAGGIONI

Performance comparison of HYB-SCF against CPU-CPLEX and HYB-MAGGIONI is summarized in Table X and Fig. 4. Table X shows that HYB-SCF achieved a reduction of 5.68s over CPU-CPLEX in terms of average solution time, which translates into an average speed-up of 1.053X. However, the speed-up curve of HYB-SCF, as shown in Fig. 4a, lacks any meaningful trend with an increase in problem size. The irregular nature of this curve can be attributed to the following two factors. First, the CF algorithm used in HYB-SCF leverages the presence of GPU-friendly (large) supernodes to gain performance. Consequently, performance gained by HYB-SCF over CPU-CPLEX is positively related to the percentage of GPU-friendly supernodes in the matrix S. Second, CPLEX is a highly optimized CPU-based solver, which contains techniques to efficiently process dense columns, in addition to other features to enhance its performance [29]. On the other hand, HYB-SCF is based on PDIPM as implemented in GLPK, which does not have any method for efficient processing of dense columns [15]. Consequently, speed-up achieved by HYB-SCF over CPU-CPLEX is negatively related to the percentage of dense columns in the matrix S. To assign appropriate weightages to

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3006353, IEEE Access

TABLE X
PERFORMANCE COMPARISON OF HYB-SCF AGAINST CPU-CPLEX AND HYB-MAGGIONI, PROBLEMS SORTED BY THE RATIO τ

Problem	HYB-SCF solution time (s)	CPU-CPLEX solution time (s)	Approximate speed-up achieved by HYB-MAGGIONI over CPU-CPLEX	HYB-MAGGIONI estimated solution time (s)
WATSON2	28.10	7.97	0.18X	44.30
RAIL507	3.93	0.47	1.29X	0.37
RAIL4284	190.50	31.10	0.82X	37.92
RAIL2586	112.86	13.25	1.28X	10.35
TP6	139.48	168.98	0.92X	183.68
DEGME	268.90	299.97	0.83X	361.41
KARTED	84.07	243.95	1.28X	190.58
TSPALKO	35.43	143.02	0.94X	152.15
Average	107.91	113.59	-	122.59

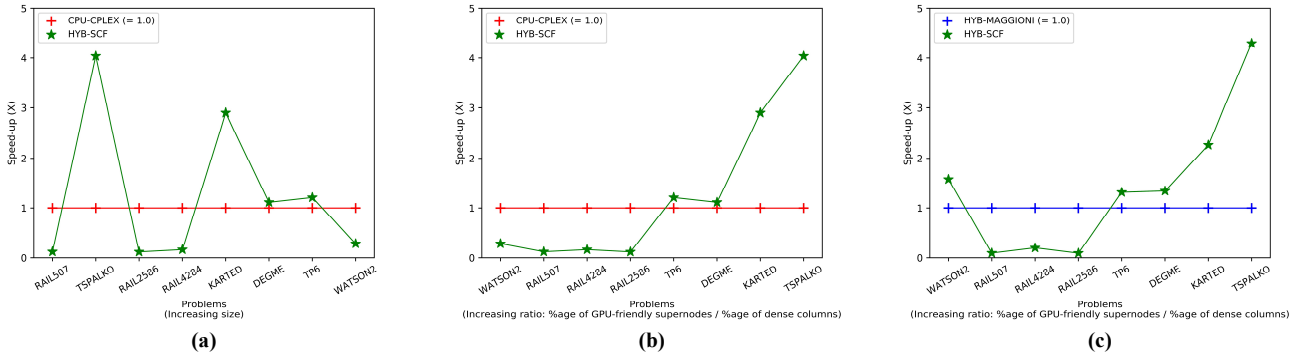

FIGURE 4. Performance comparison of HYB-SCF against CPU-CPLEX and HYB-MAGGIONI. (a) Speed-up achieved by HYB-SCF over CPU-CPLEX. Problems sorted in increasing order of size of matrix A. (b) Speed-up achieved by HYB-SCF over CPU-CPLEX. Problems sorted in increasing order of ratio τ. (c) Speed-up achieved by HYB-SCF over HYB-MAGGIONI. Problems sorted in increasing order of ratio τ.

these factors while comparing performance of HYB-SCF against CPU-CPLEX, we devised the following ratio to sort test problems.

$$\tau = \frac{Percentage\ of\ GPU\text{-}friendly\ supernodes\ in\ the\ matrix\ S}{Percentage\ of\ dense\ columns\ in\ the\ matrix\ S}$$

Fig. 4b shows that the speed-up achieved by HYB-SCF over CPU-CPLEX generally follows an upward trend as the ratio τ increases. We have also used the same ratio to sort problems in Table X.

These results signify the true impact of our proposed approach, which made the performance of a primitive implementation of PDIPM competitive with CPLEX's highly optimized industry-standard CPU implementation. It must be emphasized that the speed-up achieved by our proposed solver over CPU-CPLEX depends on performance gains achieved by the adapted CF implementation as well as CHOLMOD's highly efficient SPMM routine. We also compared the solutions returned by HYB-SCF and CPU-CPLEX, which closely matched each other. In concrete terms, there was an average percentage error of 5.80×10^{-7} % and a maximum percentage error of 2.74×10^{-6} % in the solutions returned by HYB-SCF as compared to CPU-CPLEX.

Finally, we compare the performance of HYB-SCF against HYB-MAGGIONI. Table X shows approximate speed-up achieved by HYB-MAGGIONI over CPU-CPLEX for each test problem, as reported by Maggioni [45]. It is evident from these speed-up measurements that HYB-MAGGIONI outperformed CPU-CPLEX for only three test problems; RAIL507, RAIL2586 and KARTED. The maximum speed-up achieved was around 1.29X. Subsequently, we used these speed-up measurements in conjunction with our measurements of CPU-CPLEX's solution times to estimate HYB-MAGGIONI's solution times, as shown in Table X and plotted in Fig. 4c. As discussed earlier in Section IV-E and Section V, this comparison is not biased towards HYB-SCF.

The speed-up curve of HYB-SCF in Fig. 4c follows similar trends as shown in Fig. 4b. The reasons for this symmetry between speed-up curves of HYB-SCF over CPU-CPLEX and HYB-MAGGIONI are twofold. First, as discussed previously in Section I, Maggioni used CHOLMOD's hybrid RCF routine only during some iterations of PDIPM. Furthermore, CHOLMOD's hybrid RCF routine provides meaningful speed-up over its CPU-based counterpart only for three of the test problems. Consequently, as opposed to HYB-SCF, the presence of GPU-friendly supernodes did not result in significant performance gains for HYB-MAGGIONI. Second, HYB-MAGGIONI used a sparse representation of matrices that was apparently able to handle the presence of dense columns efficiently [22], [45]. On the other hand, HYB-SCF did not have any such mechanism. The net effect is that the speed-up of HYB-SCF generally increases with an increase in the ratio τ. If we aggregate solution times for HYB-SCF and

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3006353, IEEE Access

HYB-MAGGIONI for all the test problems, we get an estimated average speed-up value of 1.136X. Interestingly, HYB-SCF also outperformed HYB-MAGGIONI for the hyper-sparse problem, WATSON2, which contains neither dense columns nor any supernodes. This observation is significant, because it implies that HYB-SCF outperformed Maggioni's state-of-the-art hybrid solver for all the test problems, except for the three problems having large number of dense columns.

VII. CONCLUSION AND FUTURE WORK

This work began with the premise that a modern GPU is potentially a high-performance computing resource for developing efficient LP problem solvers. We were particularly interested in PDIPM. However, the affinity of GPUs for dense linear algebra operations and the suitability of PDIPM for solving sparse LP problems apparently lead to a contradiction. Nevertheless, presence of supernodes, which can be treated as dense submatrices, in sparse matrices provides opportunities for GPU acceleration of PDIPM.

In this paper, we proposed a new LP problem solver based on the PDIPM implementation of GLPK. The new solver significantly accelerated the process of solving real-world sparse problems. Like the state-of-the-art solver, our proposed solver also used a CF method that exploited the presence of supernodes in sparse matrices. However, as opposed to the state-of-the-art solver, our proposed solver used an adaptation of a more efficient CF method that performs batched processing of related supernodes on GPU, which has not been used previously for accelerating PDIPM. Our adaptation made the new CF method more resilient to the effects of round-off errors that accumulate in the elements of the matrix S over multiple iterations of PDIPM. In addition to the new CF method, we used a CPU-based SPMM routine that required fewer floating-point operations as compared to GLPK's original routine. For a set of well-known sparse problems, our proposed solver was able to achieve average speed-ups of around 55X and 1.05X over original PDIPM routines of GLPK and CPLEX, respectively. Moreover, we estimated that it achieved an average speed-up of around 1.14X over the state-of-the-art hybrid solver.

Finally, we provide a couple of future research possibilities related to this work. First, effects of using GPU-based SPMM methods to compute the matrix S, mentioned in Section IV-A, need further investigation. Second, during the course of this work, we came across an enhanced version of the hybrid SCF algorithm, proposed by Tang *et al.* [46], which uses multiple CPU threads and pipelining to achieve further performance gains. Potential of this CF method for accelerating PDIPM may also be studied in a future work.

REFERENCES

[1] G. Dantzig, *Linear Programming and Extensions*. Princeton: Princeton University Press, 2016.

[2] P. Tar, B. Stágel, and I. Maros, "Parallel search paths for the simplex algorithm," *Central European Journal of Operations Research*, vol. 25, no. 4, pp. 967–984, 2017.

[3] S. Dutta, S. Jha, S. Sankaranarayanan, and A. Tiwari, "Output Range Analysis for Deep Feedforward Neural Networks," in *NASA Formal Methods*, Cham, 2018, pp. 121–138, doi: 10.1007/978-3-319-77935-5_9.

[4] M. Fischetti and J. Jo, "Deep neural networks and mixed integer linear optimization," *Constraints*, vol. 23, no. 3, pp. 296–309, 2018.

[5] T. Zhang, X. Wang, X. Xu, and C. P. Chen, "GCB-Net: Graph convolutional broad network and its application in emotion recognition," *IEEE Transactions on Affective Computing*, 2019.

[6] T. Zhang, G. Su, C. Qing, X. Xu, B. Cai, and X. Xing, "Hierarchical lifelong learning by sharing representations and integrating hypothesis," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2019.

[7] M. I. K. Khalil, I. Ahmad, and A. A. Almazroi, "Energy Efficient Indivisible Workload Distribution in Geographically Distributed Data Centers," *IEEE Access*, vol. 7, pp. 82672–82680, 2019, doi: 10.1109/ACCESS.2019.2924085.

[8] G. Li, J. Yan, L. Chen, J. Wu, Q. Lin, and Y. Zhang, "Energy consumption optimization with a delay threshold in cloud-fog cooperation computing," *IEEE Access*, vol. 7, pp. 159688–159697, 2019.

[9] T. Chen, A. G. Marques, and G. B. Giannakis, "DGLB: Distributed stochastic geographical load balancing over cloud networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 1866–1880, 2016.

[10] M. Cao and W. Zhang, "Related-Key Differential Cryptanalysis of the Reduced-Round Block Cipher GIFT," *IEEE Access*, vol. 7, pp. 175769–175778, 2019, doi: 10.1109/ACCESS.2019.2957581.

[11] C. Zhang, B. Han, W. He, X. Peng, and C. Xu, "A Novel Compressive Optical Encryption via Single-Pixel Imaging," *IEEE Photonics Journal*, vol. 11, no. 4, pp. 1–8, 2019.

[12] T. Zhang, C. P. Chen, L. Chen, X. Xu, and B. Hu, "Design of highly nonlinear substitution boxes based on I-Ching operators," *IEEE Transactions on Cybernetics*, vol. 48, no. 12, pp. 3349–3358, 2018.

[13] R. Robere, "Interior point methods and linear programming," *University of Toronto*, 2012.

[14] Q. Huangfu and J. J. Hall, "Parallelizing the dual revised simplex method," *Mathematical Programming Computation*, vol. 10, no. 1, pp. 119–142, 2018.

[15] A. Makhorin, *GNU Linear Programming Kit–Reference Manual for GLPK Version 4.64. 2017.* .

[16] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2016.

[17] U. A. Shah, S. Yousaf, I. Ahmad, S. U. Rehman, and M. O. Ahmad, "Accelerating Revised Simplex Method Using GPU-Based Basis Update," *IEEE Access*, vol. 8, pp. 52121–52138, 2020.

[18] J. H. Jung and D. P. O'Leary, "Implementing an interior point method for linear programs on a CPU-GPU system," *Electronic Transactions on Numerical Analysis*, vol. 28, no. 174–189, p. 37, 2008.

[19] S. C. Rennich, D. Stosic, and T. A. Davis, "Accelerating sparse Cholesky factorization on GPUs," *Parallel Computing*, vol. 59, pp. 140–150, 2016.

[20] E. Smith, J. Gondzio, and J. Hall, "GPU acceleration of the matrix-free interior point method," in *International Conference on Parallel Processing and Applied Mathematics*, 2011, pp. 681–689.

[21] N. F. Gade-Nielsen, B. Dammann, and J. B. Jørgensen, "Interior point methods on GPU with application to model predictive control," 2014.

[22] M. Maggioni, "Sparse Convex Optimization on GPUs," PhD Thesis, The university of Illinois, Chicago, 2016.

[23] J. D. Hogg, "High performance Cholesky and symmetric indefinite factorizations with applications," 2010.

[24] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, "Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate," *ACM Transactions on Mathematical Software (TOMS)*, vol. 35, no. 3, p. 22, 2008.

[25] S. C. Rennich, T. A. Davis, and P. Vandermersch, "GPU Acceleration of Sparse Matrix Factorization in CHOLMOD," in GPU Technology Conference 2014. Nvidia Corp," 2014. http://on-demand.gputechconf.com/gtc/2014/presentations/S4201-gpu-acceleration-sparse-matrix-factorization-cholmod.pdf (accessed Apr. 14, 2020).

[26] "suitesparse : a suite of sparse matrix software." http://faculty.cse.tamu.edu/davis/SuiteSparse/ (accessed Sep. 02, 2019).

[27] A Makhorin and AI Moscow, "GLPK - GNU project - free software foundation (FSF)." 2012, Accessed: Jul. 01, 2019. [Online]. Available: https://www.gnu.org/software/glpk/.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3006353, IEEE Access

[28] T. A. Davis, *Direct methods for sparse linear systems*, vol. 2. Siam, 2006.
[29] *IBM ILOG CPLEX Optimization Studio CPLEX User's Manual, Version 12 Release 6.* IBM Corp, 2015.
[30] I. Maros, *Computational techniques of the simplex method*, vol. 61. Springer Science & Business Media, 2012.
[31] N. Karmarkar, "A new polynomial-time algorithm for linear programming," in *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, 1984, pp. 302–311.
[32] S. Mehrotra, "On the implementation of a primal-dual interior point method," *SIAM Journal on optimization*, vol. 2, no. 4, pp. 575–601, 1992.
[33] Netlib, "BLAS (Basic Linear Algebra Subprograms)," 2017. http://www.netlib.org/blas/ (accessed Jul. 05, 2019).
[34] J. Demmel, "LAPACK: A portable linear algebra library for high-performance computers," *Concurrency: Practice and Experience*, vol. 3, no. 6, pp. 655–666, 1991.
[35] Nvidia, "cuBLAS :: CUDA Toolkit Documentation," 2017. https://docs.nvidia.com/cuda/archive/9.0/cublas/index.html (accessed Jul. 05, 2019).
[36] Nvidia, "cuSOLVER :: CUDA Toolkit Documentation," 2017. http://docs.nvidia.com/cuda/cusolver/index.html (accessed Apr. 08, 2020).
[37] U. A. Shah, S. Yousaf, I. Ahmad, and M. O. Ahmad, "Supplemental-Material-On-the-Efficiency-of-Supernodal-Factorization-in-Interior-Point-Method," *GitHub*, 2020. https://github.com/ualishah/Supplemental-Material-On-the-Efficiency-of-Supernodal-Factorization-in-Interior-Point-Method (accessed Jun. 28, 2020).
[38] "CUSP: Main Page." https://cusplibrary.github.io/ (accessed Sep. 02, 2019).
[39] Nvidia, "cuSPARSE :: CUDA Toolkit Documentation," 2017. http://docs.nvidia.com/cuda/cusparse/index.html (accessed Jul. 05, 2019).
[40] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "SpArch: Efficient Architecture for Sparse Matrix Multiplication," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2020, pp. 261–274, doi: 10.1109/HPCA47549.2020.00030.
[41] M. Deveci, C. Trott, and S. Rajamanickam, "Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures," *Parallel Computing*, vol. 78, pp. 33–46, 2018.
[42] W. Liu and B. Vinter, "An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 370–381, doi: 10.1109/IPDPS.2014.47.
[43] S. J. Wright, "The Cholesky factorization in interior-point and barrier methods," *Preprint MCS ${$P600 ${$0596, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill*, 1996.
[44] "OpenBLAS : An optimized BLAS library." https://www.openblas.net/ (accessed Sep. 03, 2019).
[45] M. Maggioni and T. Berger-Wolf, "Optimization techniques for sparse matrix–vector multiplication on GPUs," *Journal of Parallel and Distributed Computing*, vol. 93, pp. 66–86, 2016.
[46] M. Tang, M. Gadou, S. Rennich, T. A. Davis, and S. Ranka, "Optimized sparse Cholesky factorization on hybrid multicore architectures," *Journal of computational science*, vol. 26, pp. 246–253, 2018.

USMAN ALI SHAH is serving as a Lecturer at the historic Islamia College Peshawar, Pakistan, since 2010. He holds an M.S. degree in Computer Engineering, which he obtained from Sir Syed CASE Institute of Technology, formerly Center for Advance Studies in Engineering, Islamabad, Pakistan in 2010. He has recently completed his Ph.D. research work at the department of Computer Science and IT, University of Engineering and Technology Peshawar, Pakistan. His major area of research is application of high-performance computing in the context of optimization algorithms. He was also a member of the group working on developing efficient parallel algorithms for large-scale graph processing.

SUHAIL YOUSAF graduated from Quaid-i-Azam University Islamabad, Pakistan in 2004. He is currently serving as an Assistant Professor at the department of Computer Science and IT, University of Engineering Technology Peshawar, Pakistan. He was awarded with a fully funded scholarship by the government of Pakistan to pursue his M.S. leading to Ph.D. endeavor at the Computer Systems Group, Vrije Universiteit Amsterdam, the Netherlands. There he worked under the supervision of Prof. Maarten van Steen. His research interests include Internet of Things (IoT), high-performance computing (HPC), and Graph Theoretic approaches to the study of Complex Networks. Apart from M.S. students, he is currently supervising a Ph.D. student. He is also a CoPI in Intelligent Systems Design Lab at University of Engineering and Technology Peshawar under National Center of Artificial Intelligence, Pakistan. There he is leading three R&D projects.

IFTIKHAR AHMAD received his master's degree in computer science from the University of Freiburg, Germany, and his Ph.D. degree in theoretical computer science from the University of Saarland, Saarbrücken, Germany. He is currently an Assistant Professor with the Department of Computer Science and Information Technology, University of Engineering and Technology Peshawar, Pakistan. He is also leading the Machine Learning Group at National Center of Big Data and Cloud Computing, University of Engineering and Technology at Peshawar, Pakistan. His research interests include theoretical computer science, machine learning, graph theory, and blockchain & crypto currencies.

MUHAMMAD OVAIS AHMAD is a Senior Lecturer at Karlstad University, Sweden. He received his Ph.D. in Information Processing Science from the University of Oulu in 2016. His research interests are focused on software development methodologies, evidence-based software engineering, process assessment and improvement, software engineering curriculum and pedagogy, optimization techniques, technology and digital services adoption and diffusion.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.