IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received February 3, 2020, accepted March 2, 2020, date of publication March 12, 2020, date of current version March 25, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2980309

Accelerating Revised Simplex Method Using GPU-Based Basis Update

USMAN ALI SHAH[1], SUHAIL YOUSAF[1,2], IFTIKHAR AHMAD[1], SAFI UR REHMAN[3], AND MUHAMMAD OVAIS AHMAD[4]

[1]Department of Computer Science and Information Technology, University of Engineering and Technology at Peshawar, Peshawar 25120, Pakistan
[2]Intelligent Systems Design Lab, National Center of Artificial Intelligence, University of Engineering and Technology at Peshawar, Peshawar 25120, Pakistan
[3]Department of Mining Engineering, Karakoram International University, Gilgit 15100, Pakistan
[4]Department of Mathematics and Computer Science, Karlstad University, 65188 Karlstad, Sweden

Corresponding author: Muhammad Ovais Ahmad (ovais.ahmad@kau.se)

ABSTRACT Optimization problems lie at the core of scientific and engineering endeavors. Solutions to these problems are often compute-intensive. To fulfill their compute-resource requirements, graphics processing unit (GPU) technology is considered a great opportunity. To this end, we focus on linear programming (LP) problem solving on GPUs using revised simplex method (RSM). This method has potentially GPU-friendly tasks, when applied to large dense problems. Basis update (BU) is one such task, which is performed in every iteration to update a matrix called basis-inverse matrix. The contribution of this paper is two-fold. Firstly, we experimentally analyzed the performance of existing GPU-based BU techniques. We discovered that the performance of a relatively old technique, in which each GPU thread computed one element of the basis-inverse matrix, could be significantly improved by introducing a vector-copy operation to its implementation with a sophisticated programming framework. Second, we extended the adapted element-wise technique to develop a new BU technique by using three inexpensive vector operations. This allowed us to reduce the number of floating-point operations and conditional processing performed by GPU threads. A comparison of BU techniques implemented in double precision showed that our proposed technique achieved 17.4% and 13.3% average speed-up over its closest competitor for randomly generated and well-known sets of problems, respectively. Furthermore, the new technique successfully updated basis-inverse matrix in relatively large problems, which the competitor was unable to update. These results strongly indicate that our proposed BU technique is not only efficient for dense RSM implementations but is also scalable.

INDEX TERMS Dense matrices, GPU, GPGPU, linear programming, revised simplex method.

I. INTRODUCTION

Contemporary graphics processing units (GPUs) can easily perform thousands of giga floating-point operations per second (GFLOPS) for efficient real-time processing of high-definition (HD) graphics [1]. This capability of GPUs has encouraged high-performance-computing community to explore general-purpose computing on GPUs (GPGPU) for solving an ever-growing set of compute-intensive problems [2]. NVIDIA has become a leading manufacturer of GPUs used for general-purpose computing [3]. It provides a convenient programming environment based on C++ language, which is specifically tailored for its GPUs [4].

Linear programming (LP), a technique used to find the optimal value of a linear objective function subject to a set of linear constraints, is extensively used in a wide range of fields like scientific research, engineering, economics and industry [5]. LP problem solvers also serve as drivers for optimizing more complex nonlinear and mixed integer programming (MIP) problems [6]. The two basic algorithms available for solving LP problems, simplex method and interior-point method [7], have at least some phases with potential to benefit from extensive computational capability of modern GPUs [8], [9]. An LP problem can be categorized as being either dense or sparse, depending on the density of matrix containing coefficients of its constraints. Interior-point method is the preferred algorithm for solving sparse problems, which are more prevalent in real-world applications [10]. GPU implementations of interior-point method for

The associate editor coordinating the review of this manuscript and approving it for publication was Juan Touriño.

sparse problems have been proposed, which use the GPU either for processing sets of *similar* columns that can be treated as dense submatrices, called supernodes, present in an otherwise sparse matrix or for multiplying sparse matrices with dense vectors [9]–[11]. However, the peculiar memory organization of GPUs makes them more efficient at processing dense data structures [8]. This encouraged us to consider in this work an adaptation of simplex method that is suitable for solving dense LP problems. Dense LP problems are less widely used than sparse problems, yet they remain useful in fields like digital filter circuit design, data analysis and classification, financial planning and portfolio management [12]. Dense problems also arise while solving LP problems having special structural properties using Benders or Dantzig-Wolfe decomposition [13].

GPU implementations of standard simplex method have already been proposed [13], [14]. Revised simplex method (RSM) is an extension of the standard algorithm, which is suitable for developing efficient computer implementations owing to its use of matrix manipulation operations [15], [16]. Even though, relatively advanced GPU-based solvers have been proposed, like multi-GPU and batch-processing implementations of standard simplex method [14], [17], [18], we decided to focus on the traditional approach of solving a single large LP problem using a single GPU. We selected this approach because our analysis revealed that there still was potential for improving the performance of existing single-problem-single-GPU solvers. Furthermore, any resultant enhancement could be incorporated into future multi-GPU and batch-processing implementations.

RSM is an iterative algorithm. During each iteration, it requires solution of a system of linear equations for different right-hand-side (RHS) vectors, but the same coefficient matrix, called basis matrix (B). Three main methods are available for solving these systems of linear equations. The first method, called Gaussian Elimination, is relatively expensive in terms of its time complexity [19]. The second method, called LU factorization, is less expensive than Gaussian-Elimination method. It is widely used in RSM implementations for sparse problems. However, in the context of solving dense problems using RSM, LU factorization is outperformed by variants of a method that involves updating inverse of the basis matrix (B^{-1}) [19]. The third method for solving systems of linear equations involves multiplication of matrix B^{-1} and the RHS vector. However, computing inverse of a matrix may itself be an expensive operation. Fortunately, there is a specific relationship between instances of matrix B in successive iterations of RSM. This relationship leads to a basis-update (BU) technique called the product form of inverse (PFI), which requires a matrix-matrix multiplication as its primary operation. This multiplication also has a high time complexity [19]. However, a more efficient adaptation of PFI exists, which requires a vector product and a matrix addition. This technique, called modified product form of inverse (MPFI), is less expensive than original PFI [8].

During literature review, we came across five single-problem-single-GPU implementations of RSM. Greeff [20] first proposed a GPU implementation of RSM as early as 2005. In 2009, Spampinato and Elster [21] extended Greeff's work by using NVIDIA's proprietary programming environment, known as compute-unified device architecture (CUDA) [4]. They implemented the classical PFI-based BU technique in single precision, using linear algebra functions from NVIDIA's highly optimized CUBLAS library [22]. In 2010, Bieling *et al.* [16] proposed another single-precision implementation of RSM using NVIDIA's Cg programming language, which has since become obsolete [23]. Instead of directly using either PFI or MPFI, they proposed a BU technique in which each GPU thread computed exactly one element of matrix B^{-1}. In 2013, Ploskas and Samaras [19] published the results of a comparative study of different BU techniques that could be used in GPU implementations of RSM. They concluded that MPFI was more efficient than both LU factorization and PFI. It is important to mention that they did not consider the element-wise BU technique proposed earlier by Bieling *et al.* Later in 2015, Ploskas and Samaras proposed a GPU implementation of RSM, which used MPFI to perform BU [8]. They used MATLAB to develop their implementation in double precision. The latest GPU implementation of RSM was proposed by He *et al.* in 2018 [24]. They followed a column-wise approach towards updating the matrix B^{-1} by exploiting symmetry among its columns. They developed their implementation using CUDA in single precision. However, they did not provide performance comparison of their technique against any of the previously proposed techniques.

We began by implementing all the BU techniques discussed in the preceding paragraph using CUDA. It turned out that the column-wise technique, proposed by He *et al.*, provided the best performance. However, it had a drawback in the form of its inability to update matrix B^{-1} in large problems. This encouraged us to explore possibilities for developing a BU technique that was efficient as well as scalable.

We discovered that our CUDA implementation of the element-wise technique, proposed by Bieling *et al.*, offered the most obvious enhancement opportunity in the form of an avoidable matrix-copy operation. Consequently, we adapted the original element-wise technique to exploit this opportunity. The adapted technique also had a significantly reduced memory requirement, which enabled it to update matrix B^{-1} in large problems. In this work, we propose a new BU technique that further extends our adaptation of the element-wise technique. The new technique gains performance by reducing the number of control-flow instructions and floating-point operations (FLOPs) required to update the matrix B^{-1}. We used ideas from MPFI and the column-wise technique to achieve this reduction in the amount of processing by introducing a preprocessing phase that included three inexpensive vector operations.

Our results show that the new BU technique implemented in double precision, outperformed all the existing

BU techniques for randomly generated LP problems as well as large test problems from a well-known problem set. For the selected well-known problems, it achieved an average speed-up of 13.3% over its closest competitor; the column-wise technique. An LP problem solver implementing our proposed BU technique showed an average solution-time speed-up of 1.56% over a version of the same solver implementing the column-wise technique. Furthermore, the new technique requires the least amount of space in GPU's global memory, except for the column-wise technique and our prior adaptation of element-wise technique. However, unlike the column-wise technique, our proposed technique is not limited by the amount of shared memory that can be allocated to a block of GPU threads. Consequently, as opposed to the column-wise technique, the new technique was able to update matrix B^{-1} for two of the largest test problems as well.

The rest of this paper is organized as follows. In Section II, we provide background knowledge by introducing the process of solving LP problems using RSM and relevant BU techniques. In Section III, we first provide details about our implementation of existing GPU-based techniques using CUDA. Afterwards, we present an adaptation of the element-wise technique followed by our proposed BU technique. In Section IV, we describe the experimental environment under which we performed our experiments. In Section V, we provide a detailed performance comparison of all the BU techniques. We conclude the paper in Section VI by summarizing our contributions and proposing directions for future research.

II. BACKGROUND

A. SOLVING LP PROBLEMS USING RSM

Linear programming (LP) involves optimization, minimization or maximization, of the value of a given linear function, called objective function, subject to linear constraints [25]. 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 [21], [26].

$$\begin{aligned} &Minimize : z = c^T x + c_0 \\ &s.t : Ax = b, \quad x \geq 0 \end{aligned}$$

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; $A = [a_{ij}]$ is the constraint matrix where $a_{ij}, i = \{1, 2, \ldots, m\}, j = \{1, 2, \ldots, n\}$ represents coefficient of the j^{th} variable in the i^{th} constraint.

Standard simplex method provides a tableau-based method for solving an LP problem following an iterative approach towards computing the optimal value of the objective function. The algorithm begins by setting up a basis, representing an initial feasible solution. It then iteratively performs three major tasks to improve the solution: (1) Search for an *entering variable*, whose inclusion in the basis improves the solution; (2) Find the *leaving variable* to be excluded from the basis to accommodate the *entering variable*; (3) Updating the basis to represent the new feasible solution. These tasks are repeatedly performed, until the feasible solution cannot be further improved [7].

Simplex method can be represented as an iterative algorithm for a straightforward computer implementation [25]. However, it needs to maintain an updated copy of the entire tableau in the memory, making it inefficient for solving large LP problems. On the other hand, RSM reduces standard simplex method's processing overhead by using matrices to represent only relevant portions of the tableau in the memory. It applies linear algebra operations on these matrices to perform the required tableau manipulation tasks of simplex method [15], [21].

B. BASIS-UPDATE METHODS

As mentioned previously in Section I, RSM needs to solve a system of linear equations defined by matrix B, which has m number of rows and columns, four times (in some cases three times) during each iteration. Two of these solutions, called forward transformation (FTRAN), can be mathematically represented as follows.

$$By = x, \quad or \tag{1}$$
$$y = B^{-1}x \tag{2}$$

The remaining two solutions (or one solution in some cases) of the system of linear equations, called backward transformation (BTRAN), can be mathematically represented as follows.

$$B^T y = x, \quad or \tag{3}$$
$$y = \left(B^T\right)^{-1} x = \left(B^{-1}\right)^T x \tag{4}$$

As briefly discussed in Section I, there are three main options for performing FTRAN and BTRAN in the context of RSM. The first option of using Gaussian Elimination method requires $O(n^3)$ time for performing each instance of FTRAN and BTRAN [19], making it inefficient for solving large LP problems using RSM. The second option is to decompose the matrix B into its lower and upper triangular factors using a technique called LU factorization. This technique also runs in $O(n^3)$ time. However, FTRAN and BTRAN can be performed in only $O(n^2)$ time by performing substitution using triangular factors computed during the factorization phase. LU factorization has been used to efficiently factorize the matrix B in RSM for sparse LP problems [26]. However, Ploskas and Samaras experimentally showed that in the context of solving dense LP problems using RSM, LU factorization was the least efficient method among all the methods tested by them [19].

Apart from Gaussian Elimination method and LU factorization, the remaining option for performing FTRAN and

BTRAN operations is to use (2) and (4), respectively. The use of these equations requires multiplication of the matrix B^{-1} or $(B^{-1})^T$ with the RHS vector x. This matrix-vector multiplication operation runs in only $O(n^2)$ time. Unfortunately, as mentioned previously, matrix inversion is an expensive operation. However, the property that the instances of matrix B in two consecutive iterations of RSM differ from each other in only one column makes it possible to update the matrix B^{-1} cheaply, using a method called MPFI. In the following paragraphs, we provide mathematical background for MPFI.

The simplest formulation of a BU technique, which exploits the relationship between the instances of matrix B in two consecutive iterations, is called PFI, which works as follows. Suppose that

$$B_N^{-1} = \begin{bmatrix} b_{00} & b_{01} & \cdots & b_{0k} & \cdots & b_{0(m-1)} \\ b_{10} & b_{11} & \cdots & b_{1k} & \cdots & b_{1(m-1)} \\ \vdots & \vdots & & \vdots & & \vdots \\ b_{k0} & b_{k1} & \cdots & b_{kk} & \cdots & b_{k(m-1)} \\ \vdots & \vdots & & \vdots & & \vdots \\ b_{(m-1)0} & b_{(m-1)1} & \cdots & b_{(m-1)k} & \cdots & b_{(m-1)(m-1)} \end{bmatrix} \tag{5}$$

represents the matrix B^{-1} during the N^{th} iteration of RSM. Also suppose that index k of the *leaving variable* and the vector α (pivoting column) have both been already computed. We then set the variable $\theta = \alpha_k$, and calculate the elements of vector ω using the following rule.

$$\omega = \begin{cases} 1/\theta, & i = k \\ -\alpha_i/\theta, & i \neq k \end{cases} \tag{6}$$

PFI also requires another $m \times m$ matrix called Eta matrix (E). For the N^{th} iteration of RSM, matrix E can be constructed by replacing the k^{th} column of an identity matrix with vector ω, as shown below.

$$E_N = \begin{bmatrix} 1 & 0 & \cdots & \omega_0 & \cdots & 0 \\ 0 & 1 & \cdots & \omega_1 & \cdots & 0 \\ \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & \cdots & \omega_k & \cdots & 0 \\ \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & \cdots & \omega_{(m-1)} & \cdots & 1 \end{bmatrix} \tag{7}$$

Matrix B_{N+1}^{-1}, basis-inverse matrix for iteration number $(N+1)$, can then be computed by performing a straightforward matrix multiplication as follows.

$$B_{N+1}^{-1} = E_N B_N^{-1} \tag{8}$$

Both the matrices, B and B^{-1}, during the first iteration of RSM are known to be identity matrices.

The operation shown in (8) is a general matrix-matrix multiplication having a time-complexity of $O(n^3)$. However, this cost can be significantly reduced if we use an alternative formulation; MPFI. This formulation represents BU as a combination of an outer product between two vectors and a matrix addition, as shown below.

$$B_{N+1}^{-1} = \bar{B}_N^{-1} + \omega \otimes \bar{b}_k \tag{9}$$

In the above equation, $\bar{b}_k$ represents the k^{th} row of the matrix B_N^{-1}:

$$\bar{b}_k = \begin{bmatrix} b_{k0} & b_{k1} & \cdots & b_{kk} & \cdots & b_{k(m-1)} \end{bmatrix} \tag{10}$$

The matrix $\bar{B}_N^{-1}$ is obtained by setting all the elements belonging to the k^{th} row of matrix B_N^{-1} equal to zero, as shown below.

$$\bar{B}_N^{-1} = \begin{bmatrix} b_{00} & b_{01} & \cdots & b_{0k} & \cdots & b_{0(m-1)} \\ b_{10} & b_{11} & \cdots & b_{1k} & \cdots & b_{1(m-1)} \\ \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots & & \vdots \\ b_{(m-1)0} & b_{(m-1)1} & \cdots & b_{(m-1)k} & \cdots & b_{(m-1)(m-1)} \end{bmatrix} \tag{11}$$

As shown in (9), MPFI requires a matrix addition and an outer product of two vectors, which results in a time-complexity of $O(n^2)$. This cost is significantly lower than the time complexity of PFI, explicit matrix inversion and LU factorization. However, we must emphasize that both PFI and MPFI result in the same updated matrix as shown in (12) at the bottom of the next page.

If we substitute the values of ω in the above matrix, then we get its more detailed representation, as shown in (13) at the bottom of the next page.

Different equations given in this section correspond to each of the existing four BU techniques. Spampinato and Elster used (8) to implement PFI, whereas Ploskas and Samaras implemented MPFI using (9). Bieling *et al.* exploited symmetry among elements of the matrix B^{-1}, as shown in (13). Finally, He *et al.* based their technique on symmetry among columns of the matrix B^{-1}, as shown in (12). In the next section, we provide implementation details of all these BU techniques. We also describe an adaptation of the element-wise technique as well as our proposed BU technique.

III. IMPLEMENTAION DETAILS

To implement and compare the performance of different BU techniques, we decided to use an existing GPU-based LP problem solver as a baseline implementation. For this purpose, we selected the solver developed by Spampinato and Elster, since its entire source code was publicly available [7]. We began by debugging the source code of the selected baseline solver. We discovered that the minimum-element search routine used in the baseline solver did not always return the index of the first instance of the minimum element. This was contrary to the behavior exhibited by the CPU-based benchmark developed by Spampinato and

Elster. Consequently, we replaced the original search routine with the corresponding function from an open-source library known as Thrust [27]. The new function exhibited the desired behavior of always returning the index of the first instance of the minimum element. Our choice of Thrust library's function was driven by the fact that CUBLAS library, already used in the baseline solver, only provided search routines based on absolute values [22], which could not be *directly* used for implementing all the required search operations. We also modified the baseline solver so that it was able to perform double-precision FLOPs.

Since the BU technique in the baseline solver already called functions from the CUBLAS library, we decided to use it while implementing other techniques as well, wherever possible. Table 1 shows a summary of double-precision versions of CUBLAS functions used in this work. Other than using functions of CUBLAS library, we kept our implementation of each technique identical to the method proposed in its respective publication. Furthermore, we avoided the use of low-level optimizations to keep our comparison limited to the fundamental operations related to each BU technique. In Section IV, we describe the approach we followed towards selecting appropriate values of block size (*BS*), number of threads per block, for each kernel.

In addition to the baseline solver, we introduced implementations of all the BU techniques to an adapted version of a popular open-source solver called GNU linear programming kit (GLPK) [26], [28]. This allowed us to test the performance of all the BU techniques for well-known test problems, which could not be correctly solved by the cut-down version of RSM implemented in the baseline solver. Details related to our GPU implementation of GLPK are provided in Section III-G. We have made the source code, along with necessary instructions, required to reproduce all the solvers discussed in this paper publicly available [29].

TABLE 1. Functions used from the CUBLAS library.

CUBLAS function	Usage in this work
cublasDcopy()	Copying vectors and matrices
cublasDscal()	Scaling vector elements
cublasDgemv()	Matrix-vector multiplication
cublasDgemm()	Matrix-matrix multiplication and outer product of vectors
cublasDgeam()	Matrix-matrix addition

While implementing different BU techniques, we also considered the implications of selecting one of the two main orders in which elements of matrices can be stored in GPU's global memory; column-major (CM) order and row-major (RM) order. A matrix stored in either of these orders can be converted to the other by simply transposing it. The baseline solver used CM order, which as shown in the next subsection, suited the PFI-based BU technique implemented in it. However, we discovered that storage of matrix B^{-1} in RM order better suited the GPU implementation of most of the other BU techniques. Therefore, we proceeded to adapt the baseline solver so that it was able to process the matrix B^{-1}, and other matrices needed to update it, stored in RM order. It turned out that by simply interchanging instances of matrix-vector multiplication routine (*cublasDgemv()*) corresponding to FTRAN and BTRAN, the original baseline solver

$$B_{N+1}^{-1} = \begin{bmatrix} b_{00}+\omega_0 b_{k0} & b_{01}+\omega_0 b_{k1} & \cdots & b_{0k}+\omega_0 b_{kk} & \cdots & b_{0(m-1)}+\omega_0 b_{k(m-1)} \\ b_{10}+\omega_1 b_{k0} & b_{11}+\omega_1 b_{k1} & \cdots & b_{1k}+\omega_1 b_{kk} & \cdots & b_{1(m-1)}+\omega_1 b_{k(m-1)} \\ \vdots & \vdots & & \vdots & & \vdots \\ \omega_k b_{k0} & \omega_k b_{k1} & \cdots & \omega_k b_{kk} & \cdots & \omega_k b_{k(m-1)} \\ \vdots & \vdots & & \vdots & & \vdots \\ b_{(m-1)0}+\omega_{(m-1)} b_{k0} & b_{(m-1)1}+\omega_{(m-1)} b_{k1} & \cdots & b_{(m-1)k}+\omega_{(m-1)} b_{kk} & \cdots & b_{(m-1)(m-1)}+\omega_{(m-1)} b_{k(m-1)} \end{bmatrix} \tag{12}$$

$$B_{N+1}^{-1} = \begin{bmatrix} b_{00}-\frac{\alpha_0}{\theta} b_{k0} & b_{01}-\frac{\alpha_0}{\theta} b_{k1} & \cdots & b_{0k}-\frac{\alpha_0}{\theta} b_{kk} & \cdots & b_{0(m-1)}-\frac{\alpha_0}{\theta} b_{k(m-1)} \\ b_{10}-\frac{\alpha_1}{\theta} b_{k0} & b_{11}-\frac{\alpha_1}{\theta} b_{k1} & \cdots & b_{1k}-\frac{\alpha_1}{\theta} b_{kk} & \cdots & b_{1(m-1)}-\frac{\alpha_1}{\theta} b_{k(m-1)} \\ \vdots & \vdots & & \vdots & & \vdots \\ \frac{1}{\theta} b_{k0} & \frac{1}{\theta} b_{k1} & \cdots & \frac{1}{\theta} b_{kk} & \cdots & \frac{1}{\theta} b_{k(m-1)} \\ \vdots & \vdots & & \vdots & & \vdots \\ b_{(m-1)0}-\frac{\alpha_{(m-1)}}{\theta} b_{k0} & b_{(m-1)1}-\frac{\alpha_{(m-1)}}{\theta} b_{k1} & \cdots & b_{(m-1)k}-\frac{\alpha_{(m-1)}}{\theta} b_{kk} & \cdots & b_{(m-1)(m-1)}-\frac{\alpha_{(m-1)}}{\theta} b_{k(m-1)} \end{bmatrix} \tag{13}$$

could be made to process BU-related matrices stored in RM order. As shown respectively in (2) and (4), FTRAN uses the matrix B^{-1} in a non-transposed form, whereas, BTRAN uses it in a transposed form. The version of RSM implemented in the baseline solver performed one BTRAN and two FTRAN operations per iteration. Therefore, in order to enable the baseline solver to correctly process the matrix B^{-1} stored in RM order, we were required to replace one transposed and two non-transposed matrix-vector multiplications with one non-transposed and two transposed multiplications. This slightly degraded the combined performance of the two operations, FTRAN and BTRAN, while having a negligible effect on the overall performance of the solver. Fortunately, the more sophisticated version of RSM present in GLPK performed exactly two FTRAN and two BTRAN operations per iteration. Therefore, in our GLPK-based implementations, there was no difference in the combined performance of FTRAN and BTRAN, if either of the two matrix-storage orders was used.

A. PFI

The baseline solver used the implementation of a straightforward PFI-based BU technique, which we left unaltered. In the rest of this paper, we refer to this implementation as PFI-BU. Fig. 1 shows details of the four steps involved in PFI-BU. In Step 1, it spawned m^2 number of threads to initialize matrix E as an identity matrix. In Step 2, it spawned m number of threads to replace the k^{th} column of matrix E with vector ω. Storage of matrix E in CM order suited the implementation of this step, since it allowed threads to access elements of the k^{th} column in a coalesced manner. In Step 3, it called the function *cublasDgemm(),* for multiplying matrix E with the current instance of matrix B^{-1}, to get updated values of the matrix B^{-1}. Unfortunately, the function *cublasDgemm()* did not allow writing back directly to the input matrix. There were two consequences of this restriction. First, memory allocation for an additional $m \times m$ matrix was required. Second, an additional processing step (Step 4) was required, during which the original version of matrix B^{-1} was updated with the newly computed values.

In Section V, we show that even the use of highly optimized CUBLAS library could not significantly mitigate the high ($O(n^3)$) processing cost of PFI-BU. In addition, it required space for three $m \times m$ matrices (E, B_N^{-1} and B_{N+1}^{-1}) in GPU's global memory, which is the highest memory requirement among all the BU techniques. We subsequently refer to the original version of baseline solver, which implements PFI-BU, as PFI-SOL.

B. MPFI

Ploskas and Samaras also proposed a four-step GPU implementation of conventional MPFI. In each step, their implementation spawned T number of threads to perform the relevant task, where T represents the total number of cores available in a GPU (1024 in our case). In Step 1, it used the T number of threads to compute elements of vector ω using the

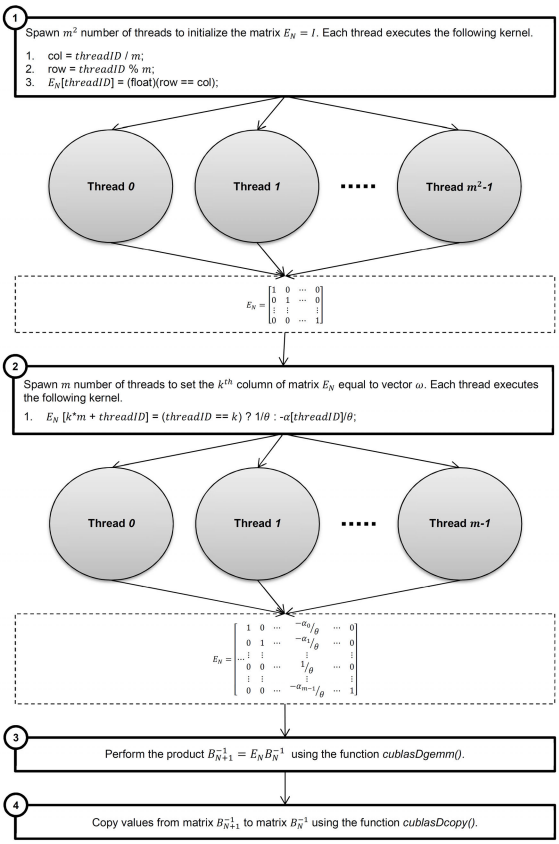

FIGURE 1. PFI-BU implemented for matrices stored in CM order.

rule given in (6). In Step 2, it performed the product of vector ω with the k^{th} row of matrix B^{-1}. In Step 3, all the elements belonging to the k^{th} row of matrix B^{-1} were set equal to zero. In the final step, output matrices of the preceding two steps were added to get the updated matrix B^{-1}.

Fig. 2 shows the details of our implementation of MPFI, which is subsequently referred to as MPFI-BU. We implemented the first step exactly as suggested by Ploskas and Samaras. We spawned $T = 1024$ threads as eight blocks of 128 threads each. Our choice of these values is explained later in Section IV. We determined that the remaining three steps could be implemented in a more generalized way using functions of the CUBLAS library, which did not require us to specify the number of threads to be spawned. Consequently, we implemented the outer product of vectors in Step 2 using the function *cublasDgemm()*, the row operation in Step 3 using the function *cublasDscal()* and matrix addition in the final step using the function *cublasDgeam()*. Use of the function *cublasDgeam()* in the final step had the additional advantage that, as opposed to the use of function *cublasDgemm()* in PFI-BU, it allowed MPFI-BU to update the matrix B^{-1} in place. Hence, MPFI-BU required space in GPU's global memory for only two $m \times m$ matrices (B_N^{-1} and output matrix of Step 2) and one vector having m number of elements (ω).

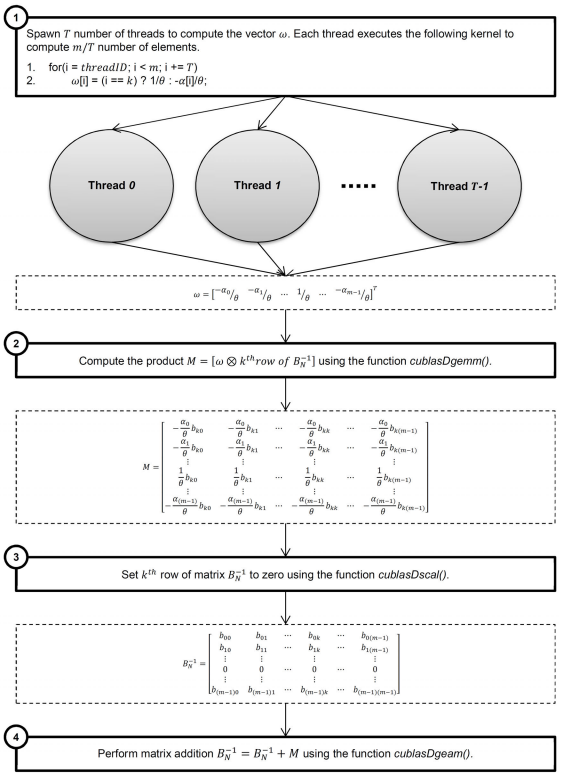

FIGURE 2. MPFI-BU implemented for matrices stored in RM order.

While implementing MPFI-BU, we observed that storage of matrix B^{-1} in CM order led elements in each row to be stored m locations apart. This caused operations in Step 2 and Step 3 to access elements of matrix B^{-1} in a non-coalesced manner. On the other hand, storing matrix B^{-1} in RM order resulted in coalesced access to its elements. Hence, we implemented MPFI-BU for matrices stored in RM order. This required us to interchange operands' positions in the function *cublasDgemm()*. This reordering was required because CUBLAS expects the matrices to be stored in CM order. Interchanging the positions of operands, causes the output of a product to be transposed, which is equivalent to representing the output in RM order.

In Section V, we show that lower time complexity of MPFI as compared to PFI also translated into significant performance gains for MPFI-BU over PFI-BU. In the rest of this paper, a version of the baseline solver having MPFI-BU as its BU technique is referred to as MPFI-SOL.

C. COLUMN-WISE TECHNIQUE

He *et al.* proposed a two-step implementation of their column-wise BU technique. In Step 1, their implementation computed vector ω using a procedure similar to the one discussed in the previous subsection for MPFI-BU. The only difference is that the new procedure used m number of threads, each computing one element of vector ω, as opposed to the use of T number of threads in MPFI-BU. We show in Section V, that the m-thread method provided slightly better performance as compared to the earlier T-thread method. However, the impact on overall performance was negligible.

In Step 2, again m number of threads were spawned, each updating elements of one column. The following equation shows the rule used by the j^{th} thread to update its corresponding column.

$$b_{ij} = \begin{cases} b_{kj}\omega_i, & i = k \\ b_{ij} + b_{kj}\omega_i, & i \neq k \end{cases} \tag{14}$$

It is evident from this rule that vector ω can be symmetrically used to update each column of matrix B^{-1}. The column-wise technique exploited this symmetry by storing a copy of vector ω in the shared memory, enabling threads in a single block to share it. To this end, all the threads belonging to a single block collaboratively copied vector ω from global memory to their corresponding multiprocessor's shared memory, before proceeding with actual computations. This allowed threads to subsequently read elements of vector ω cheaply from shared memory. However, there is a downside to following this approach as well, which limits the size of problems that can be successfully solved. Each block can be allocated a limited amount of shared memory, say S bytes, from the total available in its corresponding multiprocessor. Therefore, a block can store an instance of vector ω having $S/4$ elements represented in single precision and $S/8$ elements represented in double precision. Since the number of elements in vector ω is equal to the number of constraints in an LP problem, an implementation of the column-wise technique can only handle problems having $m \leq S/4$ using single-precision FLOPs and $m \leq S/8$ using double-precision FLOPs. For example, our GPU allowed a maximum of 48KB of shared memory per block, which meant that we could not solve any problem with $m > 12288$ using single-precision arithmetic and $m > 6144$ using double-precision arithmetic. A related observation regarding the use of shared memory is that even though we did not explicitly use shared memory earlier while implementing PFI-BU and MPFI-BU, we subsequently learnt that functions *cublasDgemm()* and *cublasDgeam()* were using shared memory at the back end. We show in Section IV that storage of vector ω in shared memory prevented our implementation of the column-wise technique from efficiently solving different problems using only a single value of BS.

The column-wise technique required the least amount of global memory among all the BU techniques. Like MPFI-BU, this technique also updated the matrix B^{-1} in place, resulting in a requirement to allocate memory for only one $m \times m$ matrix, along with a vector having m elements (ω), in global memory. However, this saving in terms of memory space led to the following restriction on the order in which elements in a column could be updated. Since, as shown in (14), all the elements in a column were computed using current value of that column's k^{th} element (b_{kj}), it was necessary that b_{kj} be updated last by each thread. We further discuss the

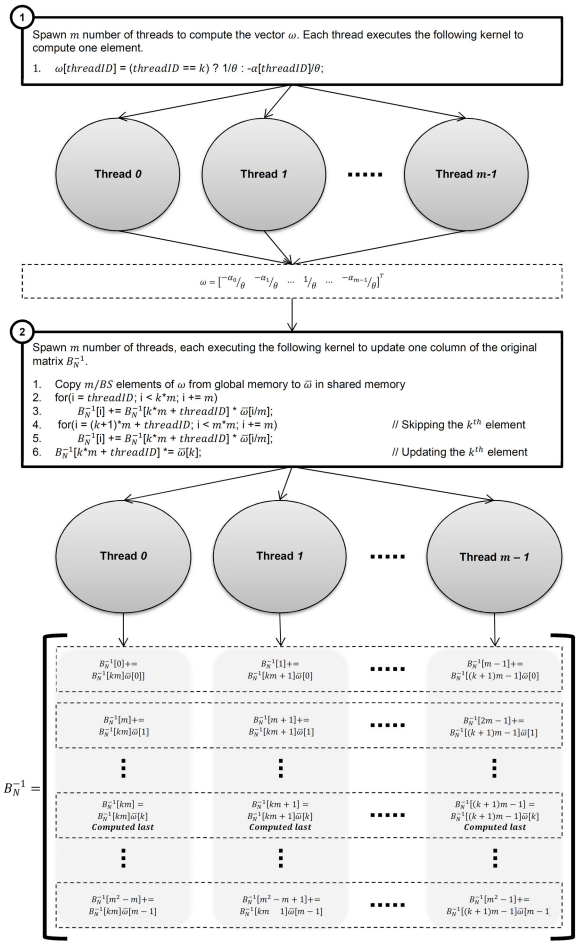

FIGURE 3. COL-WISE-ORIG-BU implemented for matrices stored in RM order.

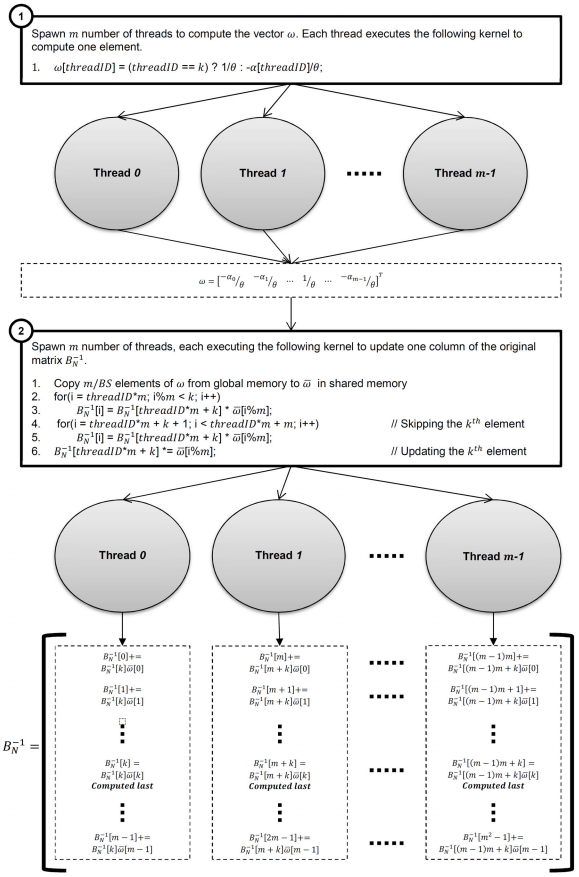

FIGURE 4. COL-WISE-ORIG-BU implemented for matrices stored in CM order, included only for illustration.

consequences of this restriction in Section III-E. It is evident that, like MPFI-BU, the time complexity of this technique is also $O(n^2)$.

We implemented the column-wise technique by following the two-step approach proposed by He *et al.* In Step 1, we spawned m number of threads to compute vector ω. In Step 2, we again spawned m number of threads and dynamically allocated the required amount of shared memory per block; $(8 \times m)$ bytes. Each thread first copied m/BS number of elements of vector ω from global memory to the shared memory. We ensured that vector ω was read from the global memory in a coalesced manner using an appropriate stride after each iteration of the for-loop. Afterwards, each thread computed all the elements of its corresponding column in the following order. Firstly, it updated elements in rows $\{0, 1, \ldots, (k-1)\}$. Second, it updated elements in rows $\{(k+1), (k+2), \ldots, (m-1)\}$. Finally, the element belonging to the k^{th} row was updated, when it was no longer required to update other elements.

Fig. 3 shows the details of our implementation of the column-wise technique. It is evident that successive threads during each iteration access (read from and write to) elements of the matrix B^{-1} stored in contiguous memory locations. This leads to a coalesced memory-access pattern. For the purpose of clarity, we also show an alternative implementation in Fig. 4, which assumes storage of matrix B^{-1} in CM order. It shows that successive threads access elements m locations apart, resulting in a non-coalesced memory-access pattern. Consequently, in the rest of the paper, we only consider the version shown in Fig. 3, which we refer to as COL-WISE-ORIG-BU. A version of the baseline solver having COL-WISE-ORIG-BU as its BU techniques is subsequently referred to as COL-WISE-ORIG-SOL.

D. ORIGINAL ELEMENT-WISE TECHNIQUE

Bieling *et al.* developed a Cg shader to implement their element-wise BU technique. The shader defined all the operations required to compute one element of the updated matrix. Therefore, there was a one-to-one mapping between threads and elements of matrix B^{-1}. The following equation summarizes the combined effect of operations defined in the shader.

$$b'_{ij} = \begin{cases} \dfrac{b_{kj}}{\theta}, & i = k \\ b_{ij} - \left(\dfrac{b_{kj}}{\theta}\right)\alpha_i, & i \neq k \end{cases} \tag{15}$$

Since, Cg language did not allow threads to update the matrix B^{-1} in place, we use b'_{ij} in the above equation to distinguish newly computed values of the matrix from current values (b_{ij}). It is clear from (15) that computation of all the elements requires the result of a division (b_{kj}/θ). Further multiplication and subtraction are required while computing elements that do not belong to the k^{th} row. Consequently, each thread is required to evaluate a condition ($i \neq k$), to determine if it needs to perform the multiplication and subtraction.

As mentioned in the previous subsection, the current value of the k^{th} element in each column is required to compute other values of that column. However, unlike COL-WISE-ORIG-BU, there is no straightforward way to delay the computation of elements belonging to the k^{th} row in the element-wise technique. Therefore, Bieling *et al.* did not update the matrix B^{-1} in place. This resulted in the requirement of performing an additional matrix-copy operation to update the matrix B^{-1} with newly computed values. However, it is important to reiterate that the use of Cg language did not allow writing back to the same matrix in any case [16], [24].

We implemented the original element-wise technique by simply porting source code of the Cg shader to a CUDA kernel. Fig. 5 shows our two-step implementation, which is subsequently referred to as ELE-WISE-ORIG-BU. In Step 1, it spawned m^2 number of threads. Each thread computed one element of the updated matrix using the rule given in (15). In Step 2, the function *cublasDcopy()* was called to write back newly computed values to the original matrix. ELE-WISE-ORIG-BU required space for two $m \times m$ matrices (B_N^{-1} and B_{N+1}^{-1}) in the global memory. Like MPFI and COL-WISE-ORIG-BU, this technique has a time complexity of $O(n^2)$.

In this paper, we only consider an implementation of ELE-WISE-ORIG-BU for matrices stored in RM order, even though the same coalesced memory access pattern could also be achieved for CM order. We prefer RM order because adaptations of ELE-WISE-ORIG-BU proposed in the next two subsections gained a slight advantage if matrix B^{-1} was stored in RM order. ELE-WISE-ORIG-BU and BU techniques discussed in the next two subsections do not show any obvious potential for benefiting from using shared memory because, unlike sharing of vector ω among threads in COL-WISE-ORIG-BU, their element-wise approach does not allow threads to share multiple values. A version of the baseline solver having ELE-WISE-ORIG-BU as its BU techniques is referred to as ELE-WISE-ORIG-SOL in the rest of this paper.

E. MODIFIED ELEMENT-WISE TECHNIQUE

In this section, we describe our initial adaptation of ELE-WISE-ORIG-BU, which was made possible by our use of CUDA. Fig. 6 shows the two steps of the new implementation. We refer to this implementation as ELE-WISE-MOD-BU in the rest of this paper. To exploit flexibility offered by CUDA that allowed the matrix B^{-1} to be updated in place, we needed to address the restriction imposed on the order in

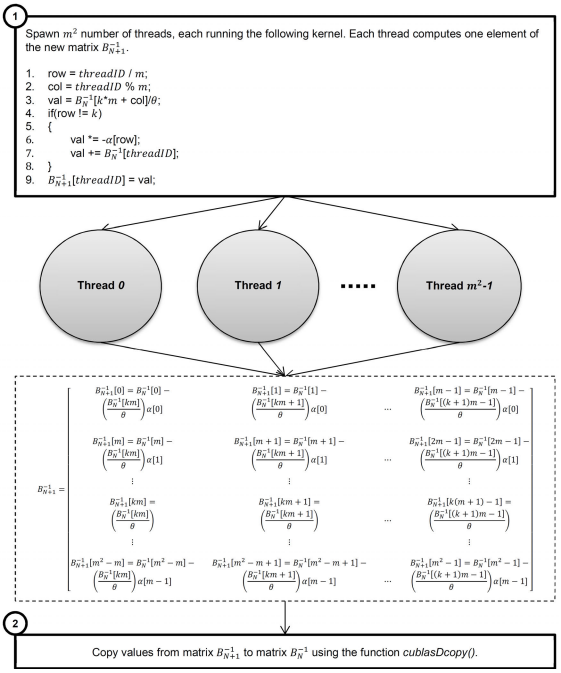

FIGURE 5. ELE-WISE-ORIG-BU implemented for matrices stored in RM order.

which elements could be updated. To address this restriction, we copied the k^{th} row of matrix B^{-1} to a separate vector ($\bar{b}_k$) in Step 1. This resulted in (15) to be transformed as follows.

$$b_{ij} = \begin{cases} \dfrac{\bar{b}_{k_j}}{\theta}, & i = k \\ b_{ij} - \left(\dfrac{\bar{b}_{k_j}}{\theta}\right)\alpha_i, & i \neq k \end{cases} \tag{16}$$

It is clear from this equation that elements belonging to the k^{th} row could be updated before computing other elements in Step 2, since current values of elements belonging to the k^{th} row were preserved in vector $\bar{b}_k$. Hence, it was possible to update the matrix B^{-1} in place, which subsequently removed the requirement of performing the expensive matrix-copy operation.

In both (15) and (16), division is performed before multiplication for updating elements that do not belong to the k^{th} row. We also implemented a version of the new technique in which multiplication was performed before division, as shown below.

$$b_{ij} = \begin{cases} \dfrac{\bar{b}_{k_j}}{\theta}, & i = k \\ b_{ij} - \dfrac{(\alpha_i \bar{b}_{k_j})}{\theta}, & i \neq k \end{cases} \tag{17}$$

We initially considered this reordering of operations to be a possible optimization opportunity, since our preliminary experiments indicated that the new implementation required, on average, fewer iterations to solve a given problem as compared to the original implementation, if both used single-precision FLOPs. Unfortunately, we could not

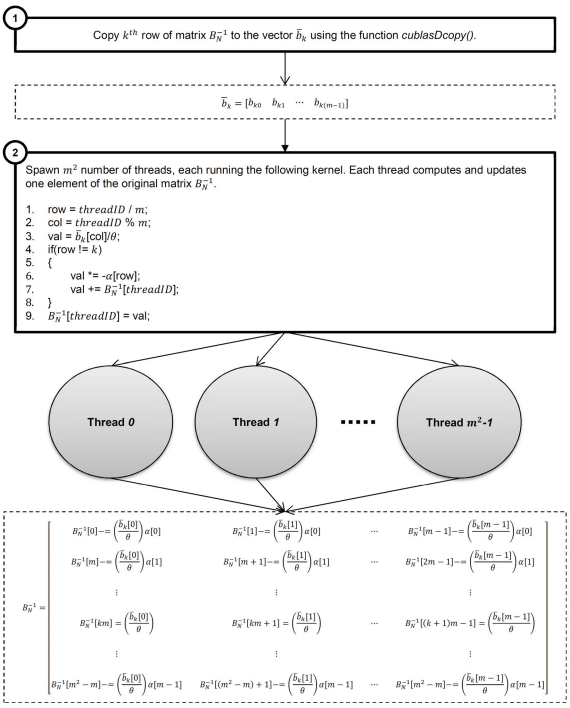

FIGURE 6. ELE-WISE-MOD-BU implemented for matrices stored in RM order.

reproduce the effect for double-precision implementations. Moreover, the reordering of operations in the new implementation made it slightly less efficient because, as shown in (17), it could not use the same value of the fraction $(\bar{b}_{k_j}/\theta)$ for both types of rows; (row $= k$) and (row $\neq k$). Consequently, we decided to continue using the original ordering of operations, as shown in (16).

ELE-WSIE-MOD-BU required space in global memory for only one $m \times m$ matrix (B_N^{-1}) and a vector having m number of elements $(\bar{b}_k)$. While implementing this technique, we observed that storage of matrix B^{-1} in CM order resulted in access to noncontiguous memory locations while copying its k^{th} row to the vector $\bar{b}_k$. On the other hand, its storage in RM order led to a coalesced memory-access pattern. Therefore, we implemented ELE-WISE-MOD-BU, along with the implementation discussed in the next subsection, for matrices stored in RM order. A version of the baseline solver, having ELE-WISE-MOD-BU as its BU technique, is referred to as ELE-WISE-MOD-SOL in the rest of this paper.

Before concluding this subsection, it is important to mention that we also applied the method discussed in this subsection, used for removing restriction on the order in which elements could be updated, to COL-WISE-ORIG-BU. Unfortunately, it had a slightly negative effect on the performance, since matrix B^{-1} was already being updated in place using coalesced memory accesses by COL-WISE-ORIG-BU. Therefore, there was no advantage to be gained from an additional vector-copy operation. However, we noticed that the removal of this restriction from an implementation of COL-WISE-ORIG-BU for matrices stored in CM order led to some improvement in its performance. Unfortunately, the improvement in performance was not enough to counter the negative effects caused by non-coalesced memory accesses in it. Hence, we persisted with the unmodified implementation of COL-WISE-ORIG-BU for matrices stored in RM order.

F. PROPOSED ELEMENT-WISE TECHNIQUE

In this subsection, we present our proposed BU technique, which was developed by further adapting ELE-WISE-MOD-BU. As explained in the previous subsection, each thread in Step 2 of ELE-WISE-MOD-BU was required to evaluate a condition to determine the row index of its corresponding element. We discovered that the need to evaluate this condition could be eliminated, if we used ideas from MPFI-BU and COL-WISE-ORIG-BU. In the following paragraphs, we discuss evolution of the new technique.

Equation (16), representing the operations performed by each thread in ELE-WISE-MOD-BU, can be rewritten in terms of ω as follows.

$$b_{ij} = \begin{cases} \mathbf{0} + \bar{b}_{k_j}\omega_i, & i = k \\ b_{ij} + \bar{b}_{k_j}\omega_i, & i \neq k \end{cases} \tag{18}$$

This equation shows that the product $P_{ij} = \bar{b}_{k_j}\omega_i$ is used to compute every element, whether it belongs to the k^{th} row or not. The only difference in the process of updating different elements is as follows. For updating elements belonging to the k^{th} row, nothing (zero) is added to the product P_{ij}. Remaining elements are updated by adding their respective current values to the product P_{ij}. To exploit this symmetry, we decided to use a method discussed earlier for Step 3 of MPFI-BU. Using the same method, we set all the elements belonging to the k^{th} row equal to zero. This row operation enables us to rewrite (18) in the following simplified form.

$$b_{ij} = b_{ij} + \bar{b}_{k_j}\omega_i \tag{19}$$

This equation shows that if vector ω has been precomputed, in addition to setting the elements belonging to the k^{th} row to zero, then elements of the matrix B^{-1} can be updated without evaluating any condition. Furthermore, one less FLOP (division) is performed while updating elements that do not belong to the k^{th} row.

Fig. 7 shows our implementation of the proposed BU technique, which is referred to as ELE-WISE-PRO-BU in the rest of this paper. In Step 1, ELE-WISE-PRO-BU spawned m number of threads. These threads performed three operations. Firstly, they copied the k^{th} row of matrix B^{-1} to the vector $\bar{b}_k$. Second, they set elements belonging to the k^{th} row equal to zero. Finally, they computed vector ω using the method discussed earlier for COL-WISE-ORIG-BU. In Step 2, ELE-WISE-PRO-BU spawned m^2 number of threads to update the matrix using (19). Since matrix B^{-1} was updated in place, this technique required space in global memory for one $m \times m$ matrix (B_N^{-1}). In addition, space for two vectors

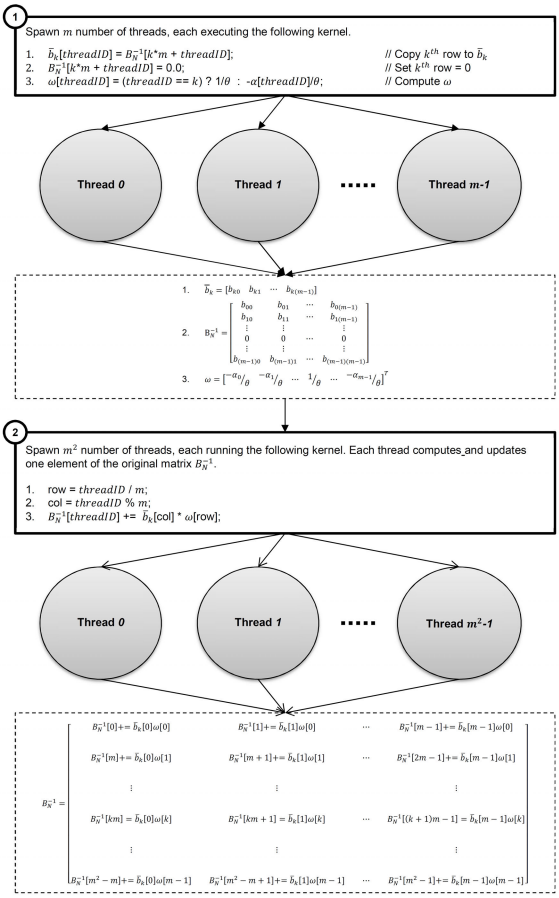

FIGURE 7. ELE-WISE-PRO-BU implemented for matrices stored in RM order.

(ω and $\bar{b}_k$), having m number of elements each, was also required. In Section V, we show that the reduced number of operations performed in Step 2, enabled ELE-WISE-PRO-BU to outperform all the other techniques. A version of the baseline solver, having ELE-WISE-PRO-BU as its BU technique, is referred to as ELE-WISE-PRO-SOL in the rest of this paper.

G. GLPK-BASED IMPLEMENTATION

We initially implemented all the BU techniques in the baseline solver. Unfortunately, the algorithm implemented in it suffered from the following three shortcomings. Firstly, it only provided the so-called *second phase* of RSM, which assumes prior availability of a feasible solution. Second, it did not allow the inclusion of surplus or artificial variables in constraints. Finally, it lacked any mechanism to handle arbitrary bounds on any variable. These shortcomings prevented the baseline solver from successfully solving *real* LP problems available in the well-known Netlib problem set [30], except for some small problems. In order to rapidly develop a test GPU implementation able to solve a wide range of well-known problems, we decided to modify the RSM implementation present in GLPK. We began by modifying configuration files of GLPK, which enabled us to compile it using the CUDA compiler (NVCC) [4]. Subsequently, we undertook the relatively simple task of developing GPU versions of vector operations used in GLPK's original implementation of RSM. Afterwards, we proceeded with the task of replacing GLPK's original CPU-based method of updating the basis matrix with our GPU implementations of BU techniques.

Instead of using explicit inverse of the matrix B to perform FTRAN and BTRAN, GLPK maintains a variation of sparse LU factorization of this matrix. By default, GLPK computes a fresh LU factorization of the matrix B after every 100 iterations. During the intervening iterations, it updates the LU factorization using the so-called Forrest-Tomlin method [31]. This method leads to a highly efficient CPU implementation of RSM for solving sparse problems. We replaced GLPK's sparse LU factorization with the six GPU-based BU techniques discussed in the preceding subsections. We also replaced the original implementations of FTRAN and BTRAN with appropriate calls to the function *cublasDegemv()* to perform the required matrix-vector multiplications on GPU. We must emphasize that our purpose was only to develop a solver through which we could test the performance of BU techniques for well-known problems. We also adapted appropriate source code files of GLPK so that it could be compiled with any of the six BU techniques by defining appropriate macros at the command line.

A summary of all the BU techniques discussed in this section is available in Table 2. In Section V, we report on a detailed performance comparison of all the techniques by using both the solvers; baseline solver and GLPK.

IV. EXPERIMENTAL SETUP

In order to test the performance of BU techniques implemented in the baseline solver, we generated random LP problems using a modified version of Spampinato and Elster's problem generating utility [7]. Their original utility generated only non-negative coefficients for the constraint matrix. This resulted in trivial problems, whose solution process generally concluded in a few (less than 100) iterations. We modified their utility so that one-third of the coefficients generated were negative, which ensured a higher degree of variation among coefficients. This resulted in nontrivial problems, which required the baseline solver to execute at least in the order of hundreds of iterations to solve them. We generated twelve such problems, having equal number of constraints (m) and variables (n), excluding slack variables. The problems varied in size from $m = 500$ in the smallest problem to $m = 6000$ in the largest problem. We have also made the source code of the modified problem-generating utility publicly available [29].

To make our results reproducible, we also provide performance comparison of BU techniques implemented in GLPK using problems available in the Netlib problem set. Most of the problems in this problem set are relatively small, which diminishes their utility for any meaningful performance

TABLE 2. Summary of BU techniques.

BU Technique	Description	Number of steps	Supported storage order of matrix B^{-1}	CUBLAS used for matrix addition or multiplication?	Space required in global memory	Shared memory used?	Spampinato and Elster's solver	GLPK-based solver
PFI-BU	PFI as implemented by Spampinato and Elster	4	CM	Yes	$3m^2$	Indirectly, by CUBLAS functions	PFI-SOL	PFI-GLPK
MPFI-BU	Our implementation of MPFI, originally proposed by Ploskas and Samaras	4	RM	Yes	$2m^2 + m$	Indirectly, by CUBLAS functions	MPFI-SOL	MPFI-GLPK
COL-WISE-ORIG-BU	Our implementation of column-wise technique proposed by He et al.	2	RM	No	$m^2 + m$	Directly, to store vector ω	COL-WISE-ORIG-SOL	COL-WISE-ORIG-GLPK
ELE-WISE-ORIG-BU	Our implementation of element-wise technique proposed by Bieling et al.	2	RM	No	$2m^2$	No	ELE-WISE-ORIG-SOL	ELE-WISE-ORIG-GLPK
ELE-WISE-MOD-BU	Our initial adaptation of element-wise technique	2	RM	No	$m^2 + m$	No	ELE-WISE-MOD-SOL	ELE-WISE-MOD-GLPK
ELE-WISE-PRO-BU	Our proposed BU technique, developed by further extending ELE-WISE-MOD-BU	2	RM	No	$m^2 + 2m$	No	ELE-WISE-PRO-SOL	ELE-WISE-PRO-GLPK

TABLE 3. Properties of Netlib problems used.

Problem	m	n	Size
FIT2P	3000	16525	49575000
MAROS-R7	3136	12544	39337984
QAP12	3192	12048	38457216
DFL001	6071	18301	111105371
QAP15	6330	28605	181069650
STOCFOR3	16675	32370	539769750

comparison in our case. Therefore, we selected only its six largest problems in terms of m. Table 3 shows properties of these six problems. These properties belong to the so-called working problems. A working problem is created by the simplex routine in GLPK after adding appropriate slack, surplus and artificial variables to the original problem.

In terms of hardware, we used a system containing an Intel Core i3-3220 CPU having 3 MB of level-3 cache and a clock-speed of 3.30 GHz. RAM installed in the system had a capacity of 16 GB. An NVIDIA GeForce GTX 960 GPU, having a total of 1024 cores, distributed among its eight multiprocessors, was also available in the system. The GPU had 4GB of global memory. Each of its eight multiprocessors had 64KB of shared memory, of which only 48KB could be allocated to a single block. In terms of software, we installed CUDA toolkit v. 9.0 on Ubuntu 17.04 to execute GPU-end code. We used CUBLAS library v. 9.0 to implement various linear algebra operations, whereas Thrust library v. 1.9 was used to implement minimum-element search operations. GLPK v. 4.65 was modified as explained previously in Section III-G.

To compare the performance of BU techniques, we solved randomly generated problems five times using versions of the baseline solver, whereas GLPK-based solver was used to solve the selected Netlib problems five times. For each run of these solvers, we measured execution time of all the major tasks using the CPU clock. We ensured proper synchronization was maintained between CPU and GPU routines while taking timing measurements. All the measurements were averaged across the five runs of each solver's attempt to solve a problem. In addition, BU time was also averaged within each run (over hundreds or thousands of iterations). To gauge performance of individual steps of each technique, we used the CUDA profiler; NVPROF [32]. We obtained execution time of each step, which was averaged over 25 samples (5 samples in the case of PFI because of its lack of performance). We ensured that the overall execution time measured earlier using CPU clock agreed with the aggregated measurements obtained using NVPROF. In addition to measuring execution times, we also used NVPROF to obtain metrics for each step related to its utilization of GPU's resources. In the next section, we mainly consider four metrics; occupancy achieved, multiprocessor activity, instructions per warp and peak-FLOP efficiency [32].

The final consideration before proceeding with experiments was to select appropriate BS values for different kernels implemented in each BU technique. In this regard, selecting BS for the kernel implemented in Step 1 of MPFI-BU was straightforward. It required number of threads to be equal to the total number of GPU cores. Therefore, we spawned 1024 threads and divided them into eight blocks of $BS = 128$ threads each. This ensured that each of our GPU's eight multiprocessors was engaged. For other kernels, we began by calculating theoretical maximum occupancy for each kernel by using CUDA's occupancy calculator [33]. It turned out that all the remaining kernels, except for one, could theoretically achieve 100% occupancy per multiprocessor for a range of BS values from 64 to 1024. Consequently, as also suggested in occupancy calculator's *help*, we followed an experimental approach towards selecting a final value of BS for each kernel. We determined average execution time of each kernel

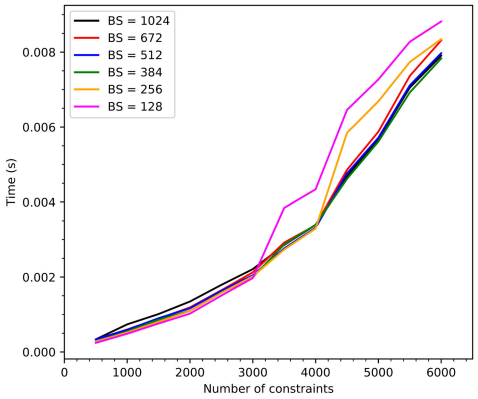

FIGURE 8. Effect of using different values of BS on the execution time of COL-WISE-ORIG-BU's Step 2.

using NVPROF for each of the twelve randomly generated problems. Interestingly, our experiments revealed that we could use any BS value suggested by the occupancy calculator without any noticeable effect on performance. Nevertheless, we decided to use $BS = 256$ for vector operations and $BS = 512$ for matrix operation in each solver, owing to a very slight improvement in performance as a result of using these values for problems having $m \geq 2000$.

In contrast to other kernels, performance of the kernel used in Step 2 of COL-WISE-ORIG-BU varied significantly for different values of BS. The occupancy calculator showed that 100% (theoretical) occupancy could be achieved only for problems having $m \leq 4000$. These problems had a shared memory requirement of less than 32KB; 8 bytes per element of vector ω. The maximum shared memory available per multiprocessor was 64KB. Hence each multiprocessor could potentially service the shared memory requirement of at least two blocks for these relatively small problems ($m \leq 4000$). The occupancy calculator provided $S = \{128, 256, 384, 512, 672, 1024\}$ as a set of values for which 100% occupancy could be achieved, depending on the shared memory requirement of each problem. However, for problems having $m > 4000$, the occupancy calculator showed 1024 as the only value of BS for which a maximum occupancy of only 50% could be achieved. We must emphasize that occupancy, which provides localized efficiency of a single multiprocessor, is not sufficient to reflect the combined efficiency of all the multiprocessors available in a GPU [32]. Consequently, we decided to compare the performance of the kernel for each of the twelve randomly generated problems using the set S. Values in the set S do not constitute an exhaustive set of all possible BS values. However, they represent a range broad enough to select a value of BS for each problem approaching the *best* value for that problem. Fig. 8 shows the results of the performance comparison conducted using values of the set S. It is evident that no single value could be used across the whole range of problems, without compromising on performance for some problem. Therefore, we proceeded to use different values of BS for solving different problems, as shown in Table 4. These values of BS cannot be generalized for other GPUs. We provide these values because COL-WISE-ORIG-BU turned out to be the closest competitor to our proposed technique. We wanted to ensure that COL-WISE-ORIG-BU was not at an undue disadvantage due to an unreasonably *poor* choice of BS, when we calculated ELE-WSIE-PRO-BU's speed-up over it.

TABLE 4. BS values used in step 2 of COL-WISE-ORIG-BU.

Problem	Value of BS
$m \leq 3000$	128 threads per block
$3000 < m \leq 4000$	256 threads per block
$m > 4000$	384 threads per block

V. RESULTS

We begin this section by comparing performance of different versions of the baseline solver for twelve randomly generated problems. We initially consider only the number of iterations, BU time and total solution time. Due to a high degree of variation in timing measurements, we have tabulated them in Table 5. Speed-up achieved by different BU techniques over PFI-BU is shown in Fig. 9a, whereas the overall speed-up achieved by each solver is shown in Fig. 9b.

Table 5 shows that each problem was solved in the same number of iterations by each solver. This is because of our use of double-precision FLOPs. During our preliminary experiments, when we used single-precision FLOPs, there was a significant degree of variation in the number of iterations performed by each solver. Another consequence of using double-precision arithmetic is that each solver showed an insignificant average percentage error of 2.02×10^{-15} % with respect to results returned by the original (CPU) implementation of RSM present in GLPK.

It is obvious from Fig. 9a that the relatively high time complexity of PFI-BU ($O(n^3)$) translated to its lack of performance as compared to all the other techniques. Table 6 shows that Step 3 of PFI-BU, in which matrix-matrix multiplication was performed, contributed primarily towards the execution time of this technique. Fig. 9a shows that the second worst-performing technique was ELE-WISE-ORIG-BU. The reason for its lack of performance was that it was the only technique, apart from PFI-BU, which did not update the matrix B^{-1} in place. As shown in Table 6, the matrix-copy operation performed in Step 2 of ELE-WISE-ORIG-BU consumed more than 30% of its overall time.

In terms of a performance comparison between MPFI-BU and ELE-WISE-MOD-BU, the former was slightly more efficient. When we profiled Step 2 and Step 4 of MPFI-BU, we observed that functions *cublasDgemm()* and *cublasDgeam()* called in these steps were both using shared memory, which can be at least partly attributed as the reason for performance gains achieved by MPFI-BU.

If we consider the T-thread method used by MPFI-BU to compute vector ω in Step 1 against the m-thread method used in the corresponding step of COL-WISE-ORIG-BU, it turns

TABLE 5. Performance of different versions of the baseline solver for randomly generated problems.

		PFI-SOL		MPFI-SOL		COL-WISE-ORIG-SOL		ELE-WISE-ORIG-SOL		ELE-WISE-MOD-SOL		ELE-WISE-PRO-SOL	
Number of Constraints	Number of iterations	BU time per iteration (s)	Total solution time (s)	BU time per iteration (s)	Total solution time (s)	BU time per iteration (s)	Total solution time (s)	BU time per iteration (s)	Total solution time (s)	BU time per iteration (s)	Total solution time (s)	BU time per iteration (s)	Total solution time (s)
500	335	0.00415	1.82	0.00012	0.41	0.00024	0.46	0.00017	0.43	0.00013	0.41	0.00006	0.39
1000	350	0.02493	10.73	0.00041	1.62	0.00047	1.64	0.00062	1.70	0.00043	1.63	0.00021	1.55
1500	761	0.08346	69.11	0.00089	4.74	0.00076	4.65	0.00137	5.10	0.00093	4.76	0.00045	4.39
2000	707	0.19661	147.79	0.00156	7.54	0.00103	7.16	0.00241	8.15	0.00164	7.61	0.00078	7.00
2500	1253	0.38436	505.12	0.00246	20.16	0.00148	18.94	0.00375	21.79	0.00255	20.28	0.00121	18.60
3000	956	0.65180	648.56	0.00354	21.84	0.00196	20.31	0.00539	23.61	0.00366	21.95	0.00174	20.12
3500	804	1.03857	864.05	0.00485	24.88	0.00274	23.21	0.00733	26.89	0.00497	24.99	0.00236	22.89
4000	860	1.54367	1367.59	0.00620	34.49	0.00330	32.02	0.00956	37.38	0.00648	34.75	0.00308	31.80
4500	1131	2.20501	2560.88	0.00813	57.73	0.00465	53.78	0.01211	62.27	0.00820	57.83	0.00391	52.99
5000	2034	3.02696	6304.47	0.01000	126.91	0.00565	118.08	0.01494	137.02	0.01012	127.18	0.00483	116.40
5500	3422	3.89590	13630.84	0.01233	258.15	0.00694	239.83	0.01806	277.89	0.01223	257.90	0.00583	236.01
6000	1490	5.07118	7711.17	0.01425	133.80	0.00786	124.32	0.02149	144.61	0.01455	134.26	0.00694	122.91

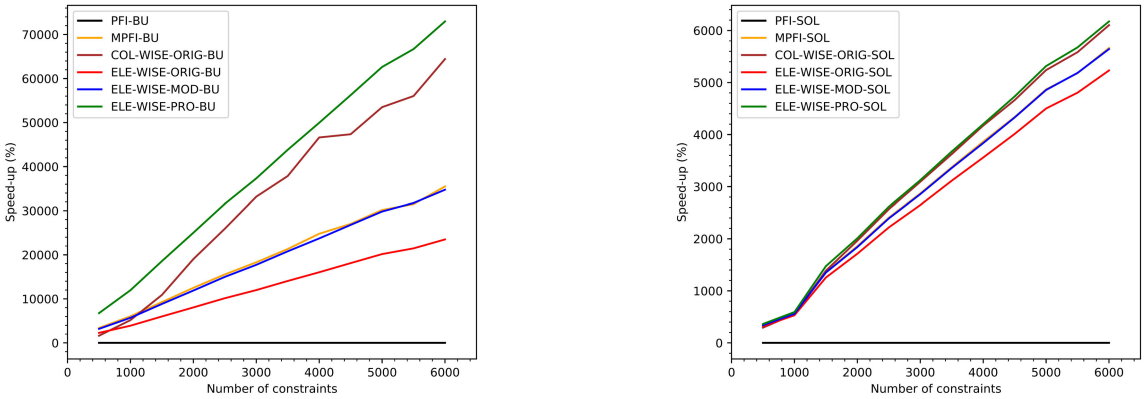

FIGURE 9. Speed-up, expressed in percentage, achieved by different versions of the baseline solver over PFI-SOL for randomly generated problems. (a) Speed-up achieved by different BU techniques with respect to PFI-BU. (b) Overall solution-time speed-up achieved by different versions of the solver with respect to PFI-SOL.

TABLE 6. Step-wise time complexity and average execution time of each BU technique.

	Step 1		Step 2		Step 3		Step 4	
BU Technique	Time complexity	Average execution time (s)	Time complexity	Average execution time (s)	Time complexity	Average execution time (s)	Time complexity	Average execution time (s)
PFI-BU	$O(n^2)$	0.001245	$O(n)$	0.000003	$O(n^3)$	1.508839	$O(n^2)$	0.002593
MPFI-BU	$O(n)$	0.000005	$O(n^2)$	0.001343	$O(n)$	0.000002	$O(n^2)$	0.003995
COL-WISE-ORIG-BU	$O(n)$	0.000003	$O(n^2)$	0.003083	-	-	-	-
ELE-WISE-ORIG-BU	$O(n^2)$	0.005733	$O(n^2)$	0.002596	-	-	-	-
ELE-WISE-MOD-BU	$O(n)$	0.000003	$O(n^2)$	0.005775	-	-	-	-
ELE-WISE-PRO-BU	$O(n)$	0.000004	$O(n^2)$	0.002598	-	-	-	-

out that the later provided better performance, as shown in Table 6. This is because of its superior occupancy, multiprocessor activity and peak-FLOP efficiency, as shown in Table 7. However, the share of execution time of both techniques' Step 1 in their total execution time was negligible.

If we consider the mutual performance comparison of the two adaptations of the element-wise technique, ELE-WISE-MOD-BU and ELE-WISE-PRO-BU, we observe that Step 1 in the later is slightly more expensive because of the two additional vector operations performed in it. However, this

TABLE 7. Average of GPU-based performance metrics for individual steps of each BU technique.

BU Technique	Step No 1				Step No. 2				Step No. 3				Step No. 4			
	Occupancy per multiprocessor (%)	Multiprocessor Activity (%)	Instructions per warp	Peak FLOP efficiency (%)	Occupancy per multiprocessor (%)	Multiprocessor Activity (%)	Instructions per warp	Peak FLOP efficiency (%)	Occupancy per multiprocessor (%)	Multiprocessor Activity (%)	Instructions per warp	Peak FLOP efficiency (%)	Occupancy per multiprocessor (%)	Multiprocessor Activity (%)	Instructions per warp	Peak FLOP efficiency (%)
PFI-BU	76.44	99.16	72.00	-	20.29	30.25	41.37	9.09	12.50	99.73 [1]	274279	94.38	81.47	99.65	31.00	-
MPFI-BU	6.10	41.05	120.97	7.16	89.06	98.06	99.51	11.64	22.05	25.86	23.88	0.85	94.35	99.30	200.27	11.41
COL-WISE-ORIG-BU	20.85	29.35	38.61	9.11	20.24	94.11	113638	8.60	-	-	-	-	-	-	-	-
ELE-WISE-ORIG-BU	88.46	99.59	124.98	73.61	81.39	99.12	31.00	-	-	-	-	-	-	-	-	-
ELE-WISE-MOD-BU	22.37	27.72	29.80	-	88.41	99.59	118.99	73.59	-	-	-	-	-	-	-	-
ELE-WISE-PRO-BU	20.06	34.43	51.62	7.75	89.05	99.14	76.99	11.65	-	-	-	-	-	-	-	-

[1] NVPROF was unable to measure multiprocessor activity of PFI-BU's Step 3 for problems having $m \geq 5500$ because of overflow in its internal variables.

small cost paid in Step 1, allowed ELE-WISE-PRO-BU to update the matrix B^{-1} using fewer floating-point and control flow operations. This reduction in the number of operations per thread performed in Step 2 enabled ELE-WISE-PRO-BU to gain performance over ELE-WISE-MOD-BU. Table 7 also confirms that Step 2 of ELE-WISE-PRO-BU executed around 35% fewer instructions per warp. The reduction in peak-FLOP efficiency suffered by Step 2 of ELE-WISE-PRO-BU was a consequence of the reduced number of FLOPs required to be performed; not indicative of any flaws in our implementation.

It is evident from Fig. 9a and Fig. 9b that COL-WISE-ORIG-BU was closest to ELE-WISE-PRO-BU in terms of performance. Before comparing these two techniques, we consider the cause of variation in the speed-up curve of COL-WISE-ORIG-BU for problems having $m > 4000$, which is apparent in Fig. 9a. These variations were caused by different rates at which execution time of Step 2 varied between different pairs of problems, as shown in Fig. 8. For instance, the slackening of speed-up while moving from $m = 4000$ to $m = 4500$ was caused by a relatively steep increase in execution time of Step 2, which was performed using $BS = 256$ at $m = 4000$ and $BS = 384$ at $m = 4500$. Similarly, the lower rate of speed-up growth while moving from $m = 5000$ to $m = 5500$ corresponds to the steeper slope of $BS = 384$ curve in Fig. 8 between these values of m.

We begin the performance comparison of ELE-WISE-PRO-BU and COL-WISE-ORIG-BU by considering Step 1 of both the techniques. Table 6 shows that the Step 1 of ELE-WISE-PRO-BU was slower than the corresponding step of COL-WISE-ORIG-BU. However, it is important to note that ELE-WISE-PRO-BU performed three vector operations in this step. Nevertheless, the cost of Step 1 in both techniques was insignificant in relation to each technique's overall execution time. In terms of a comparison between Step 2 of these techniques, we note that COL-WISE-ORIG-BU could manage a significantly lower average occupancy as compared to the 89.05% occupancy achieved by the same step in ELE-WISE-PRO-BU. As explained previously in Section IV, the lower occupancy in Step 2 of COL-WISE-ORIG-BU was a consequence of its method of storing all the elements of vector ω in shared memory. Nevertheless, our effort towards selecting *almost-the-best* value of BS for solving each problem ensured that the average multiprocessor activity was more than 94%. However, it was still less than the average multiprocessor activity achieved by Step 2 of ELE-WISE-PRO-BU. The high instructions-per-warp count for Step 2 of COL-WISE-ORIG-BU reflects the fact that each thread computed m number of elements, as opposed to the computation of a single element by each thread in Step 2 of ELE-WISE-PRO-BU. The peak-FLOP efficiency of ELE-WISE-PRO-BU's Step 2 was also better than the corresponding metric for COL-WISE-ORIG-BU's Step 2. The cumulative effect, as shown in Table 6, was that ELE-WISE-PRO-BU's Step 2 was on average 18.67% faster than the corresponding step of COL-WISE-ORIG-BU.

An interesting relationship shown in Table 6 further convinced us regarding the efficacy of our proposed BU technique. It shows that the execution time of ELE-WISE-PRO-BU's Step 2 was *almost* the same as the time taken by Step 4 of PFI-BU and Step 2 of ELE-WISE-ORIG-BU. These steps of PFI-BU and ELE-WISE-ORIG-BU performed matrix-copy operations using the function *cublasDcopy()*. In addition, the cost of ELE-WISE-PRO-BU's Step 1 was relatively insignificant. Consequently, we feel safe in claiming that the overall cost of our proposed BU technique was comparable to the cost of copying values of an $m \times m$ matrix within our GPU's global memory using a highly optimized routine.

Finally, we compare the performance of GLPK-based solvers for the selected Netlib problems. We do not consider PFI-SOL during this comparison, since it is obvious that all the other solvers were significantly faster than it. Table 8 shows the number of iterations, BU time and solution time corresponding to each solver's attempt to solve test problems. As opposed to our earlier experiment for randomly generated problems, different solvers took different number of iterations for solving some of the test problems, despite

TABLE 8. Performance of different versions of GLPK-based solver for selected Netlib problems.

	MPFI-GLPK			COL-WISE-ORIG-GLPK			ELE-WISE-ORIG-GLPK			ELE-WISE-MOD-GLPK			ELE-WISE-PRO-GLPK		
Problem	Number of iterations	BU time per iteration (s)	Total solution time (s)	Number of iterations	BU time per iteration (s)	Total solution time (s)	Number of iterations	BU time per iteration (s)	Total solution time (s)	Number of iterations	BU time per iteration (s)	Total solution time (s)	Number of iterations	BU time per iteration (s)	Total solution time (s)
FIT2P	14692	0.0035	269.0	14874	0.0020	248.2	13831	0.0056	281.0	13831	0.0039	258.0	14874	0.0017	245.5
MAROS-R7	4262	0.0038	79.2	4262	0.0021	71.8	4262	0.0061	89.0	4262	0.0042	81.0	4262	0.0019	71.3
QAP12	63979	0.0040	1193.4	63813	0.0023	1080.1	71668	0.0063	1499.2	71668	0.0044	1359.4	63813	0.0020	1058.5
DFL001	71745	0.0148	3630.3	67303	0.0082	2962.7	67387	0.0227	3943.7	67387	0.0156	3474.0	67303	0.0071	2894.5
QAP15	100000	0.0160	5441.2	-	-	-	100000	0.0247	6300.8	100000	0.0170	5530.4	100000	0.0077	4617.6
STOCFOR3	-	-	-	-	-	-	-	-	-	15245	0.1183	4590.1	15108	0.0539	3575.1

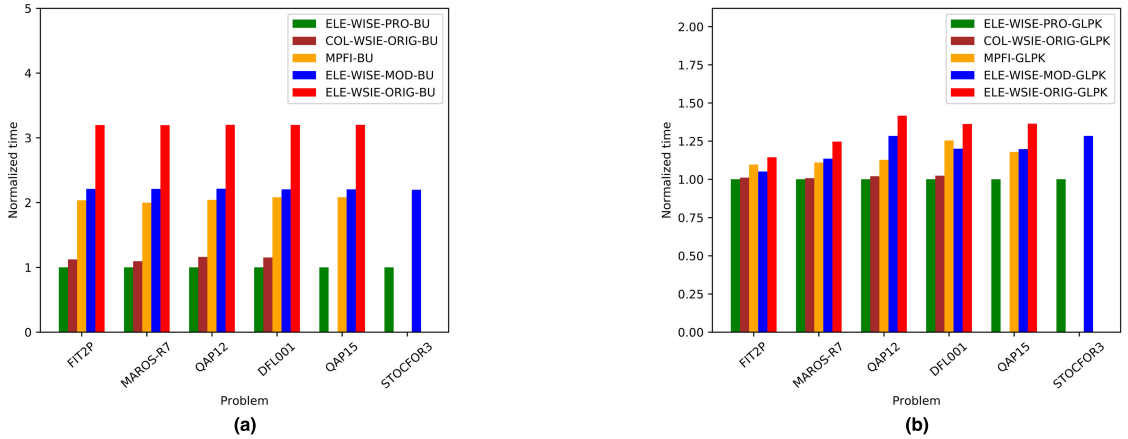

FIGURE 10. Performance comparison of GLPK-based solvers for selected Netlib problems. Normalized time for each implementation is obtained by dividing its execution time by execution time of the proposed implementation (ELE-WSIE-PRO-BU or ELE-WISE-PRO-GLPK). (a) BU time of different techniques as compared to ELE-WISE-PRO-BU. (b) Total solution time of different solvers as compared to ELE-WISE-PRO-GLPK. In both (a) and (b), bars within each group are sorted by BU time.

the use of double-precision FLOPs. This can be attributed to a general increase in the number of iterations required to solve *real* problems, which in turn increases the probability of difference in intermediate results produced by different solvers because of non-associativity of floating-point arithmetic. Fig. 10a and Fig. 10b show the performance comparison of these solvers in terms of BU time and overall solution time, respectively. ELE-WISE-MOD-GLPK and ELE-WISE-PRO-GLPK successfully solved the greatest number of problems. Therefore, we use BU technique implemented in one of these solvers, ELE-WISE-PRO-BU, as a benchmark in Fig. 10a. Similarly, we use ELE-WISE-PRO-GLPK as a benchmark in Fig. 10b.

ELE-WSIE-MOD-GLPK and ELE-WISE-PRO-GLPK were able to solve the largest Netlib problem; STOCFOR3 ($m = 16675$). They also concluded the solution of all the other problems within the limit of 100000 iterations set by us, except for QAP15. It is important to mention that even the original RSM routine present in GLPK could not solve QAP15 within 100000 iterations. Furthermore, COL-WISE-ORIG-BU could not even start the process of solving QAP15 and STOCFOR3. We subsequently calculated that COL-WISE-ORIG-BU could not have solved STOCFOR3 even if our GPU had an upper limit as high as 96KB on the allocation of shared memory per block. On the other hand, ELE-WISE-PRO-BU's only memory requirement was in terms of global memory ($m^2 + 2m$). It is also interesting to note that ELE-WISE-PRO-BU and COL-WISE-ORIG-BU took the same number of iterations to solve each of the four problems solved by the later.

We also computed average percentage error in solutions returned by each version of the GLPK-based solver with respect to solutions published by Netlib [34]. These error values for different solvers varied from 2.98×10^{-8} % for ELE-WISE-MOD-BU and ELE-WISE-ORIG-BU to 3.99×10^{-7} % for MPFI-BU. Interestingly, ELE-WISE-PRO-BU and COL-WISE-ORIG-BU both showed the same average percentage error of 1.96×10^{-7} %.

Before concluding this section, we aggregate the performance gains achieved by our proposed BU technique and solvers that implement it. Table 9 presents average speed-up achieved by ELE-WISE-PRO-SOL over other solvers, in terms of BU time and total solution time, for solving randomly generated problems having $m \geq 2000$.

TABLE 9. Speed-up achieved by the proposed technique for randomly generated problems having 2000 or more constraints.

BU Technique	BU speed-up	Solution-time speed-up
PFI-SOL	49578%	4169%
MPFI-SOL	105%	8.67%
COL-WISE-ORIG-SOL	17.4%	1.43%
ELE-WISE-ORIG-SOL	209%	17.4%
ELE-WISE-MOD-SOL	110%	9.13%

TABLE 10. Speed-up achieved by the proposed technique for selected Netlib problems.

BU Technique	BU speed-up	Solution-time speed-up
MPFI-GLPK	105%	15.3%
COL-WISE-ORIG-GLPK	13.3%	1.56%
ELE-WISE-ORIG-GLPK	220%	30.7%
ELE-WISE-MOD-GLPK	121%	19.2%

It shows that ELE-WISE-PRO-SOL achieved an average speed-up of 17.4% in terms of BU time and 1.43% in terms of total solution time over COL-WISE-ORIG-SOL. Table 10 provides speed-up comparison of ELE-WISE-PRO-GLPK against other GLPK-based solvers for solving selected Netlib problems. It shows that ELE-WISE-PRO-GLPK achieved an average speed-up of 13.3% in terms of BU time and 1.56% in terms of total solution time over COL-WISE-ORIG-GLPK.

In summary, there are three advantages of our proposed technique. The most obvious advantage is the speed-up it achieves over other techniques. Second, and probably more important, advantage is its ability to solve large LP problems. The final advantage, albeit a minor one, is its consistent performance for a range of problems using a pair of constant BS values (one each for matrix and vector operations), which is not the case with its closest competitor.

VI. CONCLUSION

In this work, we began by conducting a detailed survey of existing BU techniques used in GPU implementations of RSM. We implemented all the existing GPU-based BU techniques using CUDA, and experimentally compared their performance. It turned out that the most recent technique, which updated the matrix B^{-1} in a column-wise manner, outperformed all the other techniques. However, it was limited by its inability to handle large problems. This motivated us to develop a BU technique that was not only efficient but was also able to update matrix B^{-1} in large problems. To this end, we extended a relatively old technique, which followed an element-wise approach towards updating the matrix B^{-1}, in two stages. During the first stage, we introduced an obvious enhancement using CUDA, which significantly saved both execution time and space required in GPU's global memory. However, these savings were not enough to make it more efficient than the column-wise technique in terms of execution time. During the second stage, we developed a new BU technique by further modifying the element-wise technique. These modifications reduced the amount of computation required to update the matrix B^{-1}. This reduction in computation enabled the new technique to achieve an average speed-up of 13.3% in terms of BU time, and 1.56% in terms of total solution time, over the column-wise technique for six of the largest Netlib problems. Moreover, it was able to fulfill memory requirements of all the six test problems. On the other hand, the column-wise technique could not satisfy the shared memory requirement of the two largest test problems. In our opinion, these results merit the use of our proposed BU technique in dense LP problem solvers implementing RSM.

The most obvious direction for future research that emerged as a result of our work is the apparent potential of our proposed technique to gain performance from its multi-GPU implementation. This assertion is based on the observation that for updating any element of the matrix B^{-1}, our technique only requires that element's current value and two other vectors. In addition, the effects of partial computation of columns by threads in Step 2 of the column-wise technique also need further investigation. The performance of our proposed BU technique in a batch-processing setup may also be considered in a future work.

REFERENCES

[1] D. B. Kirk and W. H. Wen-Mei, *Programming Massively Parallel Processors: A Hands-On Approach*. Burlington, MA, USA: Morgan Kaufmann, 2016.

[2] J. Lai, H. Li, Z. Tian, and Y. Zhang, "A multi-GPU parallel algorithm in hypersonic flow computations," *Math. Problems Eng.*, vol. 2019, pp. 1–15, Mar. 2019.

[3] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra, "Graphics processing unit (GPU) programming strategies and trends in GPU computing," *J. Parallel Distrib. Comput.*, vol. 73, no. 1, pp. 4–13, Jan. 2013.

[4] Nvidia. (2017). *CUDA C Programming Guide*. Accessed: Jul. 5, 2019. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[5] G. Dantzig, *Linear Programming and Extensions*. Princeton, NJ, USA: Princeton Univ. Press, 2016.

[6] P. Tar, B. Stágel, and I. Maros, "Parallel search paths for the simplex algorithm," *Central Eur. J. Oper. Res.*, vol. 25, no. 4, pp. 967–984, Dec. 2017.

[7] D. G. Spampinato, "Linear optimization with CUDA," Norwegian Univ. Sci. Technol., Trondheim, Norway, Fall Project Rep., Jan. 2009, pp. 29–65.

[8] N. Ploskas and N. Samaras, "Efficient GPU-based implementations of simplex type algorithms," *Appl. Math. Comput.*, vol. 250, pp. 552–570, Jan. 2015.

[9] N. F. Gade-Nielsen, B. Dammann, and J. B. Jørgensen, "Interior point methods on GPU with application to model predictive control," Tech. Univ. Denmark, Lyngby, Denmark, Tech. Rep., 2014.

[10] M. Maggioni, "Sparse convex optimization on GPUs," Ph.D. dissertation, Univ. Illinois, Chicago, Chicago, IL, USA, 2016.

[11] J. H. Jung and D. P. O'Leary, "Implementing an interior point method for linear programs on a CPU-GPU system," *Electron. Trans. Numer. Anal.*, vol. 28, nos. 174–189, p. 37, 2008.

[12] J. Eckstein, İ. İ. Boduroğlu, L. C. Polymenakos, and D. Goldfarb, "Data-parallel implementations of dense simplex methods on the connection machine CM-2," *ORSA J. Comput.*, vol. 7, no. 4, pp. 402–416, Nov. 1995.

[13] M. E. Lalami, V. Boyer, and D. El-Baz, "Efficient implementation of the simplex method on a CPU-GPU system," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops Phd Forum*, May 2011, pp. 1999–2006.

[14] X. Meyer, P. Albuquerque, and B. Chopard, "A multi-GPU implementation and performance model for the standard simplex method," in *Proc. 1st Int. Symp. 10th Balkan Conf. Oper. Res.*, 2011, pp. 312–319.

[15] G. B. Dantzig, *Alternate Algorithm for the Revised Simplex Method: Using a Product Form for the Inverse*. Santa Monica, CA, USA: Rand, 1953.

[16] J. Bieling, P. Peschlow, and P. Martini, "An efficient GPU implementation of the revised simplex method," in *Proc. IEEE Int. Symp. Parallel Distrib. Process., Workshops Phd Forum (IPDPSW)*, Apr. 2010, pp. 1–8.

[17] M. E. Lalami, D. El-Baz, and V. Boyer, "Multi GPU implementation of the simplex algorithm," in *Proc. IEEE Int. Conf. High Perform. Comput. Commun.*, Sep. 2011, pp. 179–186.
[18] A. Gurung and R. Ray, "Simultaneous solving of batched linear programs on a GPU," in *Proc. ACM/SPEC Int. Conf. Perform. Eng. (ICPE)*, 2019, pp. 59–66.
[19] N. Ploskas and N. Samaras, "A computational comparison of basis updating schemes for the simplex algorithm on a CPU-GPU system," *Amer. J. Oper. Res.*, vol. 3, no. 6, pp. 497–505, 2013.
[20] G. Greeff, "The revised simplex algorithm on a GPU," Univ. Stellenbosch, Stellenbosch, South Africa, Tech. Rep., 2005.
[21] D. G. Spampinato and A. C. Elster, "Linear optimization on modern GPUs," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, May 2009, pp. 1–8.
[22] *CUBLAS:: CUDA CUDA Toolkit Documentation V9.0*, Nvidia, Santa Clara, CA, USA, 2017.
[23] Nvidia Developer. (Mar. 7, 2011). *Cg FAQ*. Accessed: Jul. 5, 2019. [Online]. Available: https://developer.nvidia.com/cg-faq
[24] L. He, H. Bai, Y. Jiang, D. Ouyang, and S. Jiang, "Revised simplex algorithm for linear programming on GPUs with CUDA," *Multimedia Tools Appl.*, vol. 77, no. 22, pp. 30035–30050, Nov. 2018.
[25] I. Maros, *Computational Techniques of the Simplex Method*, vol. 61. New York, NY, USA: Springer, 2012.
[26] A. Makhorin, "GNU linear programming kit-reference manual for GLPK version 4.64," Moscow Aviation Inst., Moscow, Russia, Tech. Rep., 2017.
[27] *Thrust—Parallel Algorithms Library*. Accessed: Nov. 5, 2019. [Online]. Available: https://thrust.github.io/
[28] A. Makhorin, "GLPK—GNU project—Free software foundation (FSF)," Free Softw. Found., Boston, MA, USA, Tech. Rep., 2012.
[29] U. A. Shah, "Supplemental material (source code and results): Accelerating revised simplex method using GPU-based basis update," *IEEE Access*, 2020.
[30] D. M. Gay, "Electronic mail distribution of linear programming test problems," *Math. Program. Soc. COAL Newslett.*, vol. 13, pp. 10–12, Dec. 1985.
[31] J. J. H. Forrest and J. A. Tomlin, "Updated triangular factors of the basis to maintain sparsity in the product form simplex method," *Math. Program.*, vol. 2, no. 1, pp. 263–278, Feb. 1972.
[32] *Profiler: CUDA Toolkit Documentation*. Accessed: Jan. 16, 2020. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html
[33] *CUDA Occupancy Calculator*. Accessed: Jan. 18, 2020. [Online]. Available: http://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html
[34] *The NETLIB LP Test Problem Set*. Accessed: Jan. 18, 2020. [Online]. Available: http://www.numerical.rl.ac.uk/cute/netlib.html

USMAN ALI SHAH received the M.S. degree in computer engineering from the Sir Syed CASE Institute of Technology, formerly Center for Advance Studies in Engineering, Islamabad, Pakistan, in 2010. He is pursuing the Ph.D. degree with the Department of Computer Science and IT, University of Engineering and Technology at Peshawar, Pakistan. He is currently on study leave from the historic Islamia College Peshawar, Pakistan, where he has been serving as a Lecturer, since 2010. His area of research is to develop efficient parallel implementations of optimization algorithms using graphics processing units (GPU). He is also a member of the group working on developing efficient parallel algorithms for large-scale graph processing.

SUHAIL YOUSAF received the degree from Quaid-i-Azam University, Islamabad, Pakistan, in 2004. He was awarded with a fully funded scholarship by the government of Pakistan. He pursued the M.S. leading to Ph.D. degree with the Computer Systems Group, Vrije Universiteit Amsterdam, The Netherlands, under the supervision of Prof. Maarten van Steen. He is currently serving as an Assistant Professor with the Department of Computer Science and IT, University of Engineering and Technology at Peshawar, Pakistan. Apart from M.S. students, he is currently supervising a Ph.D. student. He is also Co-PI with the Intelligent Systems Design Lab, University of Engineering and Technology at Peshawar, under National Center of Artificial Intelligence, Pakistan. There, he is also leading three Research and Development projects. His research interests include the Internet of Things (IoT), high-performance computing (HPC), and graph theoretic approaches to the study of complex networks.

IFTIKHAR AHMAD received the master's degree in computer science from the University of Freiburg, Germany, and the 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 at Peshawar, Pakistan. He is also leading the Machine Learning Group, National Center of Big Data and Cloud Computing, University of Engineering and Technology at Peshawar, Peshawar. His research interests include theoretical computer science, machine learning, graph theory, and block-chain and crypto-currencies.

SAFI UR REHMAN received the Ph.D. degree in mining engineering from the University of Engineering and Technology at Peshawar, Peshawar, Pakistan. His Ph.D. research was in mine planning and design optimization. He is currently working as an Assistant Professor with Karakoram International University Gilgit and engaged in mentoring, and Research and Development activities. Besides that, he also served at his alma mater UET Peshawar as a faculty member for 7 year. His ongoing R&D endeavor is development of an open source mineral informatics system (MiSys). He is a Principal Investigator of this MiSys project funded by Technology Development Fund, Higher Education Commission, Government of Pakistan, and executed through academia-industry liaison. His research interests are in application of optimization, geospatial, and computation techniques in mineral industry.

MUHAMMAD OVAIS AHMAD received the master's and Ph.D. degrees in information processing science from the University of Oulu, Finland. He is currently a Senior Lecturer in software engineering with Karlstad University, Sweden. His research interests are focused on software development methodologies, evidence-based software engineering, process assessment and improvement, software engineering curriculum and pedagogy, and digital transformations.