To appear in the *International Journal of Computer Mathematics*
Vol. 00, No. 00, February 2013, 1–17

RESEARCH ARTICLE

A Computational Comparison of Scaling Techniques for Linear Optimization Problems on a GPU

Nikolaos Ploskas[a] and Nikolaos Samaras[a]*

[a]*Department of Applied Informatics, School of Information Sciences, University of Macedonia, 156 Egnatia Str., 54006 Thessaloniki, Greece*

(*v1.0 released March 2013*)

Preconditioning techniques are important in solving linear problems, as they improve their computational properties. Scaling is the most widely used preconditioning technique in linear optimization algorithms and is used to reduce the condition number of the constraint matrix, improve the numerical behavior of the algorithms and reduce the number of iterations required to solve linear problems. Graphical Processing Units (GPUs) have gained a lot of popularity in the recent years and have been applied for the solution of linear optimization problems. In this paper, we review and implement ten scaling techniques with a focus on the parallel implementation of them on GPUs. All these techniques have been implemented under the MATLAB and CUDA environment. Finally, a computational study on the Netlib set is presented to establish the practical value of GPU-based implementations. On average the speedup gained from the GPU implementations of all scaling methods is about 7x.

Keywords: linear programming; scaling techniques; GPU; MATLAB; CUDA

AMS Subject Classification: 65F35; 90C05; 90C99

1. Introduction

Linear Programming (LP) is the process of minimizing or maximizing a linear objective function $z = \sum_{j=1}^{n} c_j x_j$ subject to a number of linear equality and inequality constraints. Simplex algorithm is the most widely used method for solving Linear Programming problems (LPs). Consider the following LP (LP.1) in the standard form:

$$\begin{aligned} min \quad & c^T x \\ s.t. \quad & Ax = b \\ & x \geq 0 \end{aligned} \qquad (LP.1)$$

where $A \in \mathbb{R}^{m \times n}$, $(c, x) \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, and T denotes transposition. We assume that A has full rank, $rank(A) = m, m < n$. Consequently, the linear system $Ax = b$ is consistent. The simplex algorithm searches for an optimal solution by moving from one feasible solution to another, along the edges of the feasible set. The dual problem associated with the (LP.1) is presented in (DP.1):

*Corresponding author. Email: samaras@uom.gr

$$\begin{aligned} min \quad & b^T w \\ s.t. \quad & A^T w + s = c \\ & s \geq 0 \end{aligned} \qquad (DP.1)$$

where $w \in \mathbb{R}^m$ and $s \in \mathbb{R}^n$.

The increasing size of real life LPs demands more computational power and parallel computing capabilities. The explosion in computational power (CPUs and GPUs) has made it possible to solve large and difficult LPs in a short amount of time. Also, parallel computing has grown rapidly during the past 20 years. As in the solution of any large scale mathematical system, the computational time and numerical accuracy for large LPs are major concerns. Preconditioning techniques can be applied to LPs prior to the application of a solver in order to improve their computational properties. Scaling is the most widely used preconditioning technique in linear optimization solvers. A matrix is badly-scaled if its nonzero elements are of different magnitudes. Scaling is an operation in which the rows and columns of a matrix are multiplied by positive scalars and these operations lead to nonzero numerical values of similar magnitude. Scaling is used prior to application of a linear programming algorithm for three reasons [17]: (a) to produce a compact representation of the bounds of variables, (b) to reduce the number of iterations required to solve LPs, (c) to simplify the setup of the tolerances, and (d) to reduce the condition number of the constraint matrix A and improve the numerical behavior of the algorithms.

Tomlin [17] presented a thesis on scaling LPs and a computational study comparing arithmetic mean, geometric mean, equilibration, Curtis and Reid [3] scaling technique, Fulkerson and Wolfe [7] scaling technique, and various combinations on six test problems of varying size. Tomlin concluded that the geometric mean scaling, optionally followed by equilibration, or the Curtis-Reid method are the best combined scaling techniques. Larsson [12] expanded on Tomlin's study by presenting and comparing entropy, $L_p$norm [9], and de Buchet [4] scaling models over 135 randomly generated problems of varying size. Larsson remarked that the entropy model is often able to improve the conditioning of the randomly generated LPs. Elble and Sahinidis [6] performed a computational study comparing arithmetic mean, de Buchet scaling model, entropy scaling technique, equilibration, geometric mean, IBM MPSX method, $L_p$norm scaling method, binormilization scaling technique, and various combinations over Netlib and Kennington set. Elble and Sahinidis used four measures to evaluate the quality of each scaling technique: (a) scaling time, (b) solution time, (c) solution iterations, and (d) maximum condition number. Elble and Sahinidis concluded that on average no scaling technique outperforms the equilibration scaling technique despite the added complexity and computational cost.

Graphical Processing Units (GPUs) have gained a lot of popularity in the past decade and High Performance Computing applications have already started to use them. GPUs offer good computational performance and efficient memory bandwidth. There are currently two major programming models available for programming on GPUs, CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language). NVIDIA introduced CUDA in late 2006 and is only available with NVIDIA GPUs, while Apple introduced OpenCL in 2008 and is available on GPUs of different vendors and even on CPUs. We have implemented the GPU code in this paper using CUDA, because it is currently more mature. CUDA enables users to execute codes on their GPUs and it is based on a Single Instruction Multiple Threads (SIMT) programming model. Any

performance improvements in the parallelization of scaling techniques would be of great interest. GPUs have been already utilized for the solution of LPs using simplex-type methods (Bieling et al. [2], Lalami et al. [11], Meyer et al. [13], Spampinato and Elster [16]) and Interior Point Methods (Jung and OLeary [10], Smith et al. [15]). Furthermore, Elble and Sahinidis [5] have proposed a parallelization of the binormilization scaling technique using a GPU that outperformed the CPU implementation.

This paper reviews and implements ten scaling schemes: (i) arithmetic mean, (ii) de Buchet for the case $p = 1$, (iii) de Buchet for the case $p = 2$, (iv) entropy, (v) equilibration, (vi) geometric mean, (vii) IBM MPSX, (viii) L_p-norm for the case $p = 1$, (ix) L_p-norm for the case $p = 2$, and (x) L_p-norm for the case $p = \infty$ and de Buchet for the case $p = \infty$. Then, we propose a parallel implementation of these schemes, which is based on MATLAB and CUDA. Finally, we carry out a detailed computational study on the Netlib set (optimal, Kennington and infeasible LPs) in order to highlight the superiority of the GPU over the CPU based implementations. Until now these scaling techniques have been tested only on CPUs. To the best of our knowledge, this is the first paper that implements all these scaling methods on a GPU and presents a corresponding computational study. The results are promising and provide hope for a fast implementation of simplex type algorithms.

The structure of the paper is as follows. In Section 2, ten scaling methods are presented and analyzed. Section 3 presents the parallel implementations of these scaling methods on a GPU. In Section 4, the computational comparison of the CPU and GPU based implementations are presented over the Netlib set (optimal, Kennington and infeasible LPs). Finally, the conclusions of this paper are outlined in section 5.

2. Scaling Techniques

2.1 *Mathematical preliminaries*

Some necessary mathematical preliminaries should be introduced, before the presentation of the scaling techniques. We have adopted notations and terminology similar to [6]. Let A be an m x n matrix. Let r_i be the row scaling factor for row i and s_j be the column scaling factor for column j. Let $N_i = \{j|A_{ij} \neq 0\}$, where $i = 1, ..., m$, and $M_j = \{i|A_{ij} \neq 0\}$, where $j = 1, ..., n$. Let n_i and m_j be the cardinality numbers of the sets N_i and M_j, respectively. Furthermore, we assume that there are no zero rows or columns in matrix A (or if exist, we remove them with the application of a presolve technique). The scaled matrix is expressed as $X = RAS$, where $R = diag(r_1...r_m)$ and $S = diag(s_1...s_n)$. All scaling methods presented in this paper, perform first a scaling of the rows and then a scaling of the columns (Gauss - Seidel based versions). A Jacobi based version of binormilization scaling technique has been proposed by Elble and Sahinidis [5] and showed smaller convergence rate, but high speed up in dense large scale LPs. In this paper, we perform a computational study over sparse LPs, so we preferred to implement Gauss - Seidel based versions of the scaling techniques.

Row and column scales are applied iteratively. Let $X^0 = A$. Let k be the index of the scaling iterations, X^k be the scaled matrix after k iterations, $X^{k+1/2}$ be the scaled matrix only by row scaling factors in iteration $k+1$, X^{k+1} be the scaled matrix both by row and column scaling factors in iteration $k+1$, r^{k+1} be the row scaling factors in iteration $k+1$, and s^{k+1} be the column scaling factors in iteration $k+1$. Then, each iteration is given by:

$$\begin{aligned} X^{k+1/2} &= R^{k+1} X^k, \\ X^{k+1} &= X^{k+1/2} S^{k+1} \end{aligned} \tag{1}$$

where:

$$\begin{aligned} R &= \prod_{k=1}^{t} R^k, \\ S &= \prod_{k=1}^{t} S^k \end{aligned} \tag{2}$$

where t is the number of iterations for a scaling method to converge to the scaled matrix.

2.2 *Arithmetic Mean*

Arithmetic mean aims to decrease the variance between the nonzero elements in the coefficient matrix A. In this method, each row is divided by the arithmetic mean of the absolute value of the elements in that row. The row scaling factors are presented in equation 3:

$$r_i^{k+1} = \left(n_i / \sum_{j \epsilon N_i} \left| X_{ij}^k \right| \right) \tag{3}$$

Similarly, each column is divided by the arithmetic mean of the absolute value of the elements in that column. The column scaling factors are presented in equation 4:

$$s_j^{k+1} = \left(m_j / \sum_{i \epsilon M_j} \left| X_{ij}^{k+1/2} \right| \right) \tag{4}$$

2.3 *de Buchet*

The de Buchet scaling model is based on the relative divergence and is formulated as shown in equation 5:

$$\min_{(r,s>0)} \left(\sum_{(i,j) \epsilon \overline{Z}} \left\{ A_{ij} r_i s_j + 1 / \left(A_{ij} r_i s_j \right) \right\}^p \right)^{1/p} \tag{5}$$

where p is a positive integer and $\overline{Z}$ is the number of nonzero elements of A.
We focus now our attention on the cases $p = 1$, 2 and ∞. For the case $p = 1$, the relation 5 is formulated as shown in equation 6:

$$\min_{(r,s>0)} \sum_{(i,j) \epsilon \overline{Z}} A_{ij} r_i s_j + 1 / \left(A_{ij} r_i s_j \right) \tag{6}$$

The row scaling factors are presented in equation 7:

$$r_i^{k+1} = \left\{ \left(\sum_{j \epsilon N_i} 1/ \left| X_{ij}^k \right| \right) \left(\sum_{j \epsilon N_i} \left| X_{ij}^k \right| \right) \right\}^{1/2} \tag{7}$$

Similarly, the column scaling factors are presented in equation 8:

$$s_j^{k+1} = \left\{ \left(\sum_{i \epsilon M_j} 1/ \left| X_{ij}^{k+1/2} \right| \right) \left(\sum_{i \epsilon M_j} \left| X_{ij}^{k+1/2} \right| \right) \right\}^{1/2} \tag{8}$$

For the case $p = 2$, the relation 5 is stated as shown in equation 9:

$$\min_{(r,s>0)} \left(\sum_{(i,j) \epsilon \overline{Z}} \left\{ A_{ij} r_i s_j + 1/ \left(A_{ij} r_i s_j \right) \right\}^2 \right)^{1/2} \tag{9}$$

The row scaling factors are presented in equation 10:

$$r_i^{k+1} = \left\{ \left(\sum_{j \epsilon N_i} \left(1/ \left| X_{ij}^k \right| \right)^2 \right) \left(\sum_{j \epsilon N_i} \left(\left| X_{ij}^k \right| \right)^2 \right) \right\}^{1/4} \tag{10}$$

Similarly, the column scaling factors are presented in equation 11:

$$s_j^{k+1} = \left\{ \left(\sum_{i \epsilon M_j} \left(1/ \left| X_{ij}^{k+1/2} \right| \right)^2 \right) \left(\sum_{i \epsilon M_j} \left(\left| X_{ij}^{k+1/2} \right| \right)^2 \right) \right\}^{1/4} \tag{11}$$

Finally, for the case $p = \infty$, the relation 5 is formulated as shown in equation 12:

$$\min_{(r,s>0)} \max_{(i,j) \epsilon \overline{Z}} \left| \log \left(A_{ij} r_i s_j \right) \right| \tag{12}$$

The row scaling factors are described in equation 13:

$$r_i^{k+1} = 1/ \left\{ \left(\max_{j \epsilon N_i} \left| X_{ij}^k \right| \right) \left(\min_{j \epsilon N_i} \left| X_{ij}^k \right| \right) \right\}^{1/2} \tag{13}$$

Similarly, the column scaling factors are presented in equation 14:

$$s_j^{k+1} = 1/ \left\{ \left(\max_{i \epsilon M_j} \left| X_{ij}^{k+1/2} \right| \right) \left(\min_{i \epsilon M_j} \left| X_{ij}^{k+1/2} \right| \right) \right\}^{1/2} \tag{14}$$

2.4 *Entropy*

The entropy model was first presented by Larsson [12] and is attributed to Dantzig and Erlander. This technique solves the model presented in equation 15, in order to identify

a scaling X, with all $x_{ij} \neq 0$ of magnitude one:

$$\begin{aligned} \min & \sum_{(i,j)\epsilon\overline{Z}} X_{ij} \left(\log \left(X_{ij}/A_{ij}\right) - 1\right) \\ s.t. & \sum_{j\epsilon N_i)} X_{ij} = n_i \qquad i = 1, ..., m \\ & \sum_{i\epsilon M_j} X_{ij} = m_j \qquad j = 1, ..., n \\ & X_{ij} \geq 0 \qquad \forall \left(i,j\right) \epsilon\overline{Z} \end{aligned} \tag{15}$$

According to Larsson [12], any algorithm for solving entropy programs can be used for scaling matrices, but it is recommended to apply the row and column scaling factors presented in equations 16 and 17:

$$r_i^{k+1} = n_i / \sum_{j\epsilon N_i} \left|X_{ij}^k\right| \tag{16}$$

$$s_j^{k+1} = m_j / \sum_{i\epsilon M_j} \left|X_{ij}^{k+1/2}\right| \tag{17}$$

2.5 *Equilibration*

For each row of the coefficient matrix A the largest element in absolute value is found, and the specified row of matrix A and the corresponding element of vector b is multiplied by the inverse of the largest element. Then, for each column of the coefficient matrix A that does not include 1 as the largest element in absolute value, the largest element in absolute value is found, and the specified column of matrix A and the corresponding element of vector c is multiplied by the inverse of the largest element. Consequently, all the elements of matrix A will have values between -1 and 1.

2.6 *Geometric Mean*

Like arithmetic mean, geometric mean also aims to decrease the variance between the nonzero elements in the matrix. In this method, the row scaling factors are calculated as shown in equation 18:

$$r_i^{k+1} = \left(\max_{j\epsilon N_i} X_{ij}^k \min_{j\epsilon N_i} X_{ij}^k\right)^{-1/2} \tag{18}$$

Similarly, the column scaling factors are presented in equation 19:

$$s_j^{k+1} = \left(\max_{j\epsilon M_j} X_{ij}^{k+1/2} \min_{j\epsilon M_j} X_{ij}^{k+1/2}\right)^{-1/2} \tag{19}$$

2.7 *IBM MPSX*

The method was proposed by Benichou et al. [1] and was later adopted by IBM, which used this method in IBMs MPSX linear optimization solver. This method combines geometric mean and equilibration scaling techniques. Initially, geometric mean is performed four times or until the relation 20 is true.

$$\frac{1}{\left|\overline{Z}\right|}\left(\sum_{(i,j)\epsilon\overline{Z}}\left(X_{ij}^{k}\right)^{2}-\left(\sum_{(i,j)\epsilon\overline{Z}}\left(X_{ij}^{k}\right)^{2}\right)^{2}/\left|\overline{Z}\right|\right)<\varepsilon \tag{20}$$

where $\left|\overline{Z}\right|$ is the cardinality number of nonzero elements of X^k and ε is a tolerance, which often is set below ten. Then, a equilibration scaling is applied.

2.8 *Lp-norm*

The L_p-norm scaling model is formulated as shown in equation 21:

$$\min_{(r,s>0)}\left(\sum_{(i,j)\epsilon\overline{Z}}\left|\log\left(A_{ij}r_is_j\right)\right|^{p}\right)^{1/p} \tag{21}$$

where p is a positive integer and $\left|\overline{Z}\right|$ is the cardinality number of nonzero elements of A. We focus now our attention on the cases $p = 1$, 2 and ∞. For the case $p = 1$, the model is formulated as shown in equation 22:

$$\min_{(r,s>0)}\sum_{(i,j)\epsilon\overline{Z}}\left|\log\left(A_{ij}r_is_j\right)\right| \tag{22}$$

We divide each row and column by the median of the absolute value of the nonzero elements. The row scaling factors are presented in equation 23:

$$r_i^{k+1}=1/\text{median}\left\{X_{ij}^{k}|j\epsilon N_i\right\} \tag{23}$$

Similarly, the column scaling factors are presented in equation 24:

$$s_j^{k+1}=1/\text{median}\left\{X_{ij}^{k+1/2}|i\epsilon M_j\right\} \tag{24}$$

For the case $p = 2$, the model is stated as shown in equation 25:

$$\min_{(r,s>0)}\left(\sum_{(i,j)\epsilon\overline{Z}}\left|\log\left(A_{ij}r_is_j\right)\right|^{2}\right)^{1/2} \tag{25}$$

The row scaling factors are calculated as shown in equation 26:

$$r_i^{k+1}=1/\prod_{j\epsilon N_i}\left(X_{ij}^{k}\right)^{1/n_i} \tag{26}$$

Similarly, the column scaling factors are presented in equation 27:

$$s_j^{k+1} = 1/\prod_{i \epsilon M_j} \left(X_{ij}^{k+1/2}\right)^{1/m_j} \tag{27}$$

Finally, for the case $p = \infty$, the model is equivalent to the de Buchet method (case $p = \infty$).

3. Parallel Implementation of Scaling Techniques on a GPU

In this section, we present the GPU-based implementations of the aformentioned scaling methods, taking advantage of the power that modern GPUs offer. The parallel implementations of these scaling techniques are implemented on MATLAB and CUDA. MATLAB's GPU support was added in version R2011a. The scaling methods are built using CUDA MEX files and invoked from MATLAB.
All methods take as input the constraint matrix A, the right-hand side vector b and the objective function coefficients c. The outputs are the scaled matrices A, b and c, and the row and column scaling factors, r and s respectively.

3.1 *GPU Architecture*

This section briefly describes the architecture of a NVIDIA GPU in terms of hardware and software. The GPU is a multi-core processor having thousands of threads running concurrently. The GPU has many cores aligned in a particular way forming a single hardware unit. Data parallel algorithms are well suited for such devices, since the hardware can be classified as SIMT (Single-Instruction, Multiple Threads). GPUs outperform CPUs in terms of GFLOPS (Giga Floating Point Operations per Second). For example a high-end Core I7 processor with 3.46 GHz delivers a peak of 55.36 GFLOPs, while a high-end NVIDIA Quadro 6000 delivers a peak of 1030.4 GFLOPs.
NVIDIA CUDA is an architecture that manages data-parallel computations on a GPU. A CUDA program includes two portions, one that executes on the CPU and another that executes on the GPU. The code that can be parallelized is executed on the GPU, as kernels, while the rest is executed on the CPU. CPU starts the execution of each portion of code and invokes a kernel function, so the execution is moved to the GPU. The connection between CPU memory and GPU memory is through a fast PCIe 16x point to point link. Each code that is executed on the GPU is divided into many threads. Each thread executes the same code independently on different data. A thread block is a group of threads that cooperate via shared memory and synchronize their execution to coordinate their memory accesses. A grid consists of a group of thread blocks and a kernel is executed on a group of grids. A kernel is the resulting code after the compilation. NVIDIA Quadro 6000, which was used in our computational experiment, consists of 14 stream processors with 32 cores each, resulting in 448 total cores.

3.2 *Notations*

Some necessary notation should be introduced before the presentation of the pseudocode of each scaling method. Let r be a 1xm vector with row scaling factors and s be a 1xn vector with column scaling factors. Let sum_row be a 1xm vector with the sum of each

row's elements and *sum_col* be a 1xn vector with the sum of each column's elements. Furthermore, *row_max* be a 1xm vector with each row's maximum element and *col_max* be a 1xn vector with each column's maximum element. Finally, let *count_row* be a 1xm vector with the number of each row's nonzero elements and and *count_col* be a 1xn vector with the sum of each column's nonzero elements.
The pseudocode of arithmetic mean and equilibration scaling techniques are presented in the following sub-section of this section. Pseudocodes include "do parallel" and "end parallel" sections, in which the workload is divided into warps that are executed sequentially on a multiprocessor. The warp size in NVIDIA Quadro 6000 is 32 threads. The pseudocodes presented in the following sub-sections do not work as they are for arrays larger than the warp size, because the scan of matrix A and vectors b and c is performed in place. The results of one warp will be overwritten by threads in another warp. We assume, without loss of generality, that matrix A and vectors b and c are double-buffered. This assumption is made merely to simplify the presentation of the subsequent pseudocodes. Although, the scaling process can be applied iteratively, pseudocodes present only one iteration. Finally, all pseudocodes are presented according to the equations of section 2.

3.3 *Implementation of GPU-Based Scaling Techniques*

In this sub-section, the GPU-Based implementation of two scaling methods is presented: (i) arithmetic mean, and (ii) equilibration. Similar steps are performed in each of the aforementioned scaling techniques. Figure 1 presents the process that is performed in the GPU-based implementation of arithmetic mean. Firstly, the CPU reads the matrix A and vectors b and c from the input file. In the second step, the GPU gathers the input data and calculates the number and the sum of nonzero elements of each row. In step 3, the GPU calculates the row scaling factors as the number of nonzero elements of each row to the sum of the same row. Then, in the fourth step, the GPU updates matrix A and vector b according to the row scaling factors. In the fifth step, the GPU calculates the number and the sum of nonzero elements of each column. In step 6, the GPU calculates the column scaling factors as the number of nonzero elements of each column to the sum of the same column. Then, in step 7, the GPU updates matrix A and vector c according to the column scaling factors. Finally, the CPU gathers the results and save matrix A and vectors b and c to the output file.

Table 1 shows the pseudocode of the implementation of the arithmetic mean scaling technique on a GPU. In the first for-loop (lines 2:14), the row scaling factors are calculated in parallel as the number of nonzero elements of each row to the sum of the same row (line 10). If the absolute value of the sum and the inverse sum of a row are not zero (line 9), then matrix A and vector b are updated (lines 11:12). Finally, in the second for-loop (lines 17:31), the column scaling factors are calculated in parallel as the number of nonzero elements of each column to the sum of the same column (line 25). If the absolute value of the sum and the inverse sum of a column are not zero (line 24), then matrix A and vector c are updated (lines 26:27).

Figure 2 presents the process that is performed in the GPU-based implementation of equilibration. Firstly, the CPU reads the matrix A and vectors b and c from the input file. In the second step, the GPU gathers the input data and calculates the maximum element of each row. In step 3, the GPU calculates the row scaling factors as the inverse of the maximum element of each row. Then, in the fourth step, the GPU updates matrix A and vector b according to the row scaling factors. In the fifth step, the GPU calculates

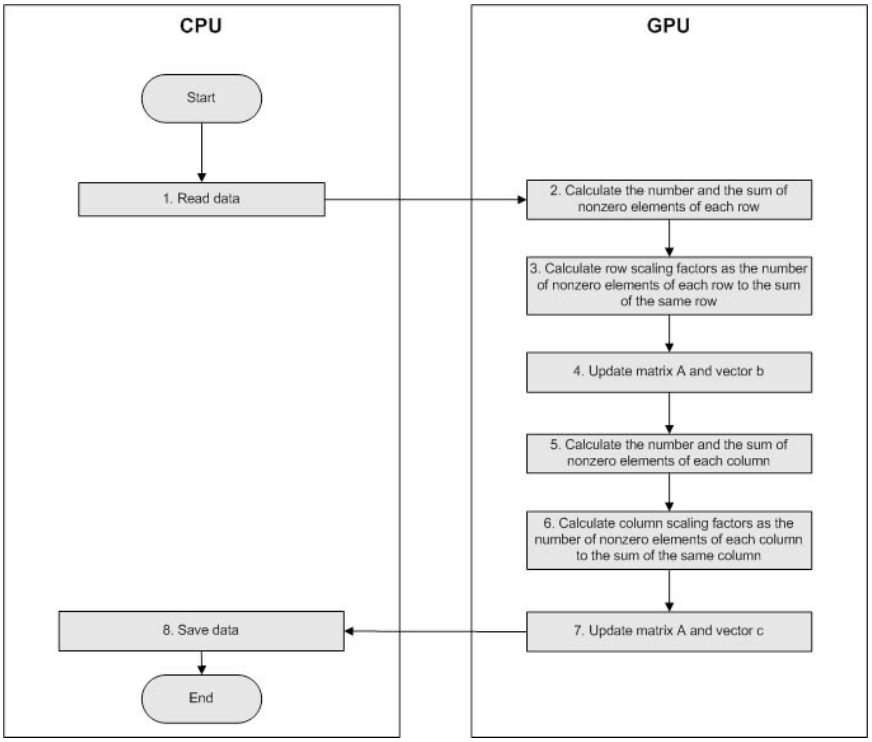

Figure 1. Flow Chart of the GPU-based Arithmetic Mean

the maximum element of each column. In step 6, the GPU calculates the column scaling factors as the inverse of the maximum element of each column. Then, in step 7, the GPU updates matrix A and vector c according to the column scaling factors. Finally, the CPU gathers the results and save matrix A and vectors b and c to the output file.

Table 2 shows the pseudocode of the implementation of the equilibration scaling technique on a GPU. In the first for-loop (lines 2:9), the row scaling factors are calculated in parallel as the inverse of the maximum element of each row (line 5). If the the maximum element of a row is not zero (line 4), then matrix A and vector b are updated (lines 6:7). Similarly, in the second for-loop (lines 12:19), the column scaling factors are calculated in parallel as the inverse of the maximum element of each column (line 15). If the the maximum element of a column is not zero (line 14), then matrix A and vector c are updated (lines 16:17).

4. Computational Results

Computational studies have been widely used, in order to examine the practical efficiency of an algorithm or even compare algorithms. The computational comparison of the aforementioned updating schemes has been performed on a quad-processor Intel Core i7 3.4 GHz with 32 Gbyte of main memory and 8 cores, a clock of 3700 MHz, an L1 code cache of 32 KB per core, an L1 data cache of 32 KB per core, an L2 cache of 256 KB per core, an L3 cache of 8 MB and a memory bandwidth of 21 GB/s, running under Microsoft Windows 7 64-bit and on a NVIDIA Quadro 6000 with 6 GB GDDR5 384-bit memory, a core clock of 574 MHz, a memory clock of 750 MHz and a memory bandwidth of 144 GB/s. It consists of 14 stream processors with 32 cores each, resulting

Table 1. GPU-based Arithmetic Mean

```
1. do parallel
2. for i=1:m
3.     for j=1:n
4.         if A[i][j] != 0
5.             sum_row[i] = sum_row[i] + |A[i][j]|
6.             count_row[i] = count_row[i] + 1
7.         end if
8.     end for
9.     if count_row[i] != 0 AND sum_row[i] != 0
10.        r[i] = count_row[i] / sum_row[i]
11.        A[i][:] = A[i][:] * r[i]
12.        b[i] = b[i] * r[i]
13.    end if
14. end for
15. end parallel
16. do parallel
17. for i=1:n
18.    for j=1:m
19.        if A[i][j] != 0
20.            sum_col[i] = sum_col[i] + |A[i][j]|
21.            count_col[i] = count_col[i] + 1
22.        end if
23.    end
24.    if count_col[i] != 0 AND sum_col[i] != 0
25.        s[i] = count_col[i] / sum_col[i]
26.        A[:][i] = A[:][i] * s[i]
27.        c[i] = c[i] * s[i]
30.    end if
31. end for
32. end parallel
```

in 448 total cores. The graphics card driver installed in our system is NVIDIA 64 kernel module 306.23. The serial algorithms have been implemented using MATLAB Professional R2012b. MATLAB (MATrix LABoratory) is a powerful programming environment and is especially designed for matrix computations in general. The mex files of the parallel implementations have been implemented using CUDA SDK 4.2 and Microsoft Visual Studio 2012.

Serial scaling techniques automatically execute on multiple computational threads, in order to take advantage of the multiple cores of the CPU. The execution time of all serial scaling techniques already includes the performance benefit of the inherent multithreading in MATLAB. MATLAB supports multithreaded computation for some built-in functions. These functions automatically utilize multiple threads without the need to specify commands to handle the threads in a code. Of course, MATLAB's inherent multithreading is not so efficient as a pure parallel implementation. Execution times both on CPU and GPU-based scaling techniques have been measured using tic and toc MATLAB's built-in functions.

The test set used in the computational study was the Netlib set (optimal, Kennington and infeasible LPs) [8]. The Netlib library is a well known suite containing many real world LPs. Ordóñez and Freund [14] have shown that 71% of the Netlib LPs are

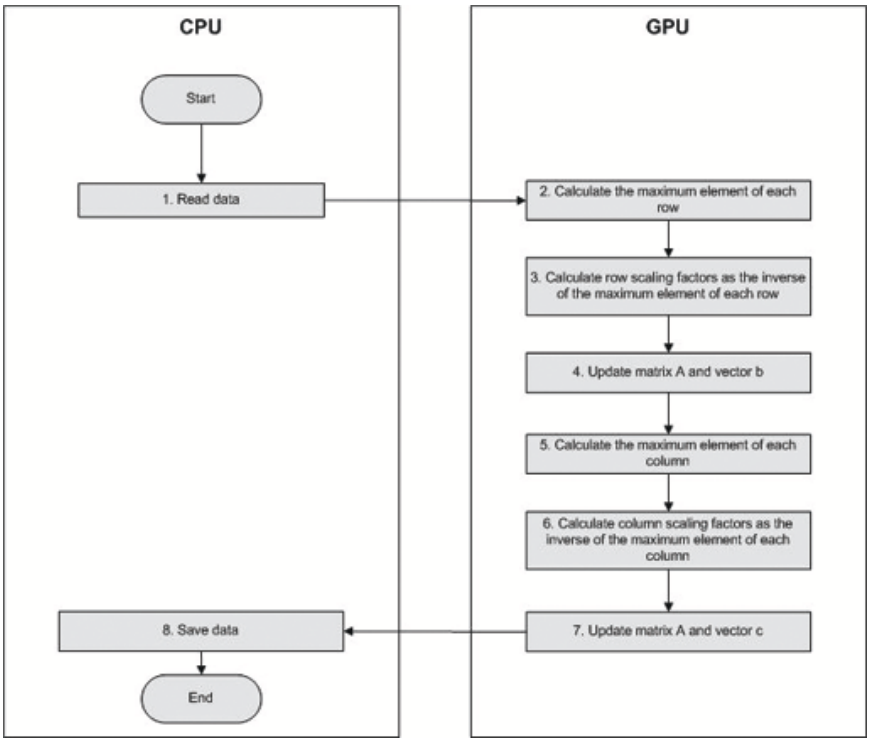

Figure 2. Flow Chart of the GPU-based Arithmetic Mean

ill-conditioned. Hence, numerical difficulties may occur. Table 3 presents some useful information about the test bed, which was used in the computational study. The first column includes the name of the problem, the second the number of constraints, the third the number of variables, the fourth the nonzero elements of matrix A and the fifth the objective value. The test bed includes 53 optimal and 7 infeasible LPs from Netlib and 5 Kennington LPs that do not have ranges and bounds sections in their mps files.

Table 4 presents the results from the execution of the serial implementations of the scaling methods, while Table 5 presents the results of the parallel ones. Table 6 presents the speedup obtained by the GPU-based implementations. In Tables 4 - 6, the following abbreviations are used: (i) S1 - arithmetic mean, (ii) S2 - de Buchet for the case p = 1, (iii) S3 - de Buchet for the case p = 2, (iv) S4 - entropy, (v) S5 - equilibration, (vi) S6 - geometric mean, (vii) S7 - IBM MPSX, (viii) S8 - L_p-norm for the case $p = 1$, (ix) S9 - L_p-norm scaling technique for the case $p = 2$, and (x) S10 - L_p-norm scaling technique for the case $p = \infty$ and de Buchet for the case $p = \infty$. For each instance, we averaged times over 10 runs. All times in the Tables 4 - 5 are measured in seconds.

We must highlight that the computational results observed for all scaling techniques are very promising. From the above results, we observe that: (i) both the serial and parallel implementations of all scaling techniques are very close in terms of time except entropy and IBM MPSX scaling methods, (ii) GPU outperforms CPU in all problems and for all scaling techniques, and (iii) on average the speedup gained from the GPU implementations of all scaling methods is about $7x$. The application is memory-bound, so if we compare GPU main memory bandwidth of 144 GB/s to 21 GB/s available per processor on CPU motherboards, we will end up with a ratio of about 6.85. The slightly greater speedup can be explained by the use of shared memory. Furthermore, we observe a slightly smaller speedup in large scale LPs. This occurs for two main reasons. First,

Table 2. GPU-based Equilibration

```
1.  do parallel
2.  for i=1:m
3.     find the maximum element in row i and store it to row_max[i]
4.     if row_max[i] != 0
5.         r[i] = 1 / row_max[i];
6.         A[i][:] = A[i][:] * r[i]
7.         b[i] = b[i] * r[i]
8.     end if
9.  end for
10. end parallel
11. do parallel
12. for i=1:n
13.    find the maximum element in column i and store it to col_max[i]
14.    if col_max[i] != 0
15.        s[i] = 1 / col_max[i]
16.        A[:][i] = A[:][i] * s[i]
17.        c[i] = c[i] * s[i]
18.    end if
19. end for
20. end parallel
```

the communication and synchronization cost for small linear problems, e.g. AFIRO (28 constraints and 32 variables), is almost zero. On the other hand, the communication and synchronization cost for large scale LPs is expensive. For example, the communication cost for OSA-30 ($4,351$ constraints and $100,024$ variables) is 3.12 seconds, approximately 1/6 of the GPU's total execution time. The second factor affecting the speedup in large scale LPs is that MATLAB's inherent multithreading in CPU performs better in large scale LPs. Finally, the results of the GPU are very accurate, because NVIDIA Quadro 6000 is fully IEEE 764-2008 compliant 32- and 64-bit fast double-precision.

5. Conclusions

Scaling is a de facto preconditioning technique that is applied by linear optimization solvers prior to the execution of a linear programming algorithm. By scaling an LP, the condition number of the constraint matrix can be reduced and that can lead to the reduction of the iterations and the solution time of the simplex algorithm. In this paper, we reviewed and implemented ten scaling methods and proposed GPU-based implementations for them using MATLAB and CUDA. Nowadays, GPUs are used in scientific programming, because they outperform high-end CPUs in data intensive problems. We performed a computational study and found that GPU-based implementations outperform the serial ones. More specifically, the speedup gained from all scaling techniques is about 7x. The results are very promising and provide hope for fast GPU-based implementations for linear programming algorithms. In future work, we plan to port all steps of the revised simplex algorithm in a GPU-based implementation in order to fully exploit parallelism. Furthermore, we plan to implement composite-Jacobi based versions of these scaling techniques that may utilize better the GPU and outperform Gauss - Seidel based scaling techniques presented in this paper.

Table 3. Statistics of the Netlib LPs (optimal, Kennington and infeasible LPs)

Problem	Constraints	Variables	Nonzeros A	Objective value
25FV47	822	1,571	11,127	5.50E+03
ADLITTLE	57	97	465	2.25E+05
AFIRO	28	32	88	-4.65E+02
AGG	489	163	2,541	-3.60E+07
AGG2	517	302	4,515	-2.02E+07
AGG3	517	302	4,531	1.03E+07
BANDM	306	472	2,659	-1.59E+02
BEACONFD	174	262	3,476	3.36E+04
BGINDY	2,672	10,116	75,019	Infeasible
BGPRTR	21	34	90	Infeasible
BLEND	75	83	521	-3.08E+01
BNL1	644	1,175	6,129	1.98E+03
BNL2	2,325	3,489	16,124	1.81E+03
BRANDY	221	249	2,150	1.52E+03
CRE-A	3,517	4,067	19,054	2.35E+07
CRE-C	3,069	3,678	16,922	2.52E+07
D2Q06C	2,172	5,167	35,674	1.23E+05
DEGEN2	445	534	4,449	-1.44E+03
DEGEN3	1,504	1,818	26,230	-9.87E+02
E226	224	282	2,767	-1.88E+01
FFFFF800	525	854	6,235	5.56E+05
ISRAEL	175	142	2,358	-8.97E+05
ITEST2	10	4	17	Infeasible
ITEST6	12	8	23	Infeasible
KLEIN1	55	54	696	Infeasible
KLEIN2	478	54	4,585	Infeasible
KLEIN3	995	88	12,107	Infeasible
LOTFI	154	308	1,086	-2.53E+01
MAROS-R7	3,137	9,408	151,120	1.50E+06
OSA-07	1,119	23,949	167,643	5.35E+10
OSA-14	2,338	52,460	367,220	1.10E+06
OSA-30	4,351	100,024	700,160	2.13E+06
QAP12	3,193	8,856	44,244	5.23E+02
QAP15	6,331	22,275	110,700	1.04E+03
QAP8	913	1,632	8,304	2.04E+02
SC105	106	103	281	-5.22E+01
SC205	206	203	552	-5.22E+01
SC50A	51	48	131	-6.46E+01
SC50B	51	48	119	-7.00E+01
SCAGR25	472	500	2,029	-1.48E+07
SCAGR7	130	140	553	-2.33E+06
SCFXM1	331	457	2,612	1.84E+04
SCFXM2	661	914	5,229	3.67E+04
SCFXM3	991	1,371	7,846	5.49E+04
SCORPION	389	358	1708	1.88E+03
SCRS8	491	1,169	4,029	9.04E+02
SCSD1	78	760	3,148	8.67E+00
SCSD6	148	1,350	5,666	5.05E+01
SCSD8	398	2,750	5,500	9.05E+02
SCTAP1	301	480	2,052	1.41E+03
SCTAP2	1,091	1,880	8,124	1.72E+03
SCTAP3	1,481	2,480	1,0734	1.42E+03
SHARE1B	118	225	1,182	-7.66E+04
SHARE2B	97	79	730	-4.16E+02
SHIP04L	403	2,118	8,450	1.79E+06
SHIP04S	403	1,458	5,810	1.80E+06
SHIP08L	779	4,283	17,085	1.91E+06
SHIP08S	779	2,387	9,501	1.92E+06
SHIP12L	1,152	5,427	21,597	1.47E+06
SHIP12S	1,152	2,763	10,941	1.49E+06
STOCFOR1	118	111	474	-4.11E+04
STOCFOR2	2,158	2,031	9,492	-3.90E+04
TRUSS	1,001	8,806	36,642	4.59E+05
WOOD1P	245	2,594	70,216	1.44E+00
WOODW	1,099	8,405	37,478	1.30E+00

Acknowledgements

The authors thank NVIDIA for their support through the Academic Partnership Program. Furthermore, the authors are indebted to the anonymous referees whose comments and suggestions have improved the quality of the final paper.

References

[1] Benichou, M., Gauthier, J.M., Hentges, G., Ribiere, G.: The efficient solution of large-scale linear programming problems-Some algorithmic techniques and computational results. Math. Program.

Table 4. Scaling time of serial implementations (secs)

Problem	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
25FV47	0.1872	0.2028	0.2028	4.9296	0.1872	0.1872	0.3588	0.2496	0.2184	0.2028
ADLITTLE	0.0001	0.0001	0.0156	0.0001	0.0001	0.0156	0.0156	0.0001	0.0156	0.0001
AFIRO	0.0001	0.0001	0.0001	0.0156	0.0001	0.0001	0.0001	0.0156	0.0001	0.0001
AGG	0.0156	0.0156	0.0156	0.0156	0.0156	0.0156	0.0312	0.0312	0.0156	0.0312
AGG2	0.0312	0.0624	0.0624	0.0312	0.0468	0.0468	0.0936	0.0624	0.0624	0.0468
AGG3	0.0468	0.0468	0.0624	0.0312	0.0468	0.0624	0.0936	0.0780	0.0468	0.0468
BANDM	0.0312	0.0156	0.0156	0.0780	0.0156	0.0156	0.0624	0.0312	0.0156	0.0312
BEACONFD	0.0001	0.0001	0.0156	0.0001	0.0001	0.0001	0.0156	0.0156	0.0156	0.0001
BGINDY	3.8376	3.9312	4.0560	3.8532	3.8064	3.9624	7.9093	4.2588	4.2120	3.9156
BGPRTR	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0156	0.0001
BLEND	0.0001	0.0001	0.0001	0.0156	0.0156	0.0156	0.0156	0.0156	0.0001	0.0001
BNL1	0.0780	0.0936	0.1092	2.7612	0.0780	0.0936	0.1716	0.1248	0.1092	0.0936
BNL2	0.4836	0.5616	0.6084	106.6579	0.4680	0.4992	0.9672	0.6240	0.5616	0.5148
BRANDY	0.0156	0.0156	0.0156	0.0156	0.0001	0.0156	0.0312	0.0312	0.0156	0.0001
CRE_A	1.1232	1.2012	1.2948	178.3559	1.1388	1.1324	1.8746	1.3016	1.2014	1.1354
CRE_C	0.8736	0.9204	0.9828	126.3140	0.8268	0.9032	1.5016	1.0013	0.9845	0.9514
D2Q06C	1.2948	1.3260	1.4040	162.3346	1.2636	1.3572	2.6208	1.4976	1.5132	1.3260
DEGEN2	0.0468	0.0624	0.0468	0.4524	0.0468	0.0468	0.0936	0.0624	0.0624	0.0468
DEGEN3	0.7176	0.7176	0.7176	19.8589	0.6708	0.6864	1.3416	0.7644	0.8268	0.6708
E226	0.0156	0.0312	0.0156	0.0312	0.0001	0.0312	0.0312	0.0312	0.0156	0.0156
FFFFF800	0.0468	0.0468	0.0468	0.3276	0.0468	0.0312	0.0780	0.0624	0.0468	0.0468
ISRAEL	0.0001	0.0001	0.0156	0.0156	0.0001	0.0156	0.0312	0.0156	0.0156	0.0001
ITEST2	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0156	0.0001	0.0156
ITEST6	0.0001	0.0001	0.0156	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
KLEIN1	0.0001	0.0156	0.0001	0.0001	0.0001	0.0001	0.0156	0.0001	0.0001	0.0001
KLEIN2	0.0468	0.0312	0.0468	0.0468	0.0312	0.0468	0.0624	0.0468	0.0468	0.0312
KLEIN3	0.1716	0.1872	0.1872	0.1716	0.1716	0.1872	0.3432	0.2184	0.2340	0.1872
LOTFI	0.0156	0.0001	0.0156	0.0312	0.0156	0.0156	0.0312	0.0156	0.0156	0.0156
MAROS-R7	2.9016	3.0576	3.1668	2.9172	2.9484	3.0576	15.2881	3.2448	3.8220	2.9640
OSA-07	4.0872	4.2588	4.4304	4.0872	3.9936	4.2717	8.3617	5.1168	5.1697	4.4176
OSA-14	19.6561	19.7497	20.2177	19.6717	18.9541	19.8277	39.2343	21.6529	22.3614	20.0015
OSA-30	81.5573	85.0049	83.3045	81.6977	79.9193	81.9824	140.0167	89.1428	91.1643	83.4710
QAP12	2.3244	2.4180	2.5116	2.3712	2.2776	2.4960	4.8828	2.7768	2.9016	2.4024
QAP15	11.5441	11.5285	11.8873	11.4661	11.1229	11.6221	23.6342	12.7141	14.6485	12.1993
QAP8	0.1248	0.1560	0.1560	0.1248	0.1248	0.1404	0.2652	0.2028	0.1560	0.1404
SC105	0.0001	0.0001	0.0001	0.0156	0.0001	0.0001	0.0001	0.0156	0.0156	0.0001
SC205	0.0156	0.0156	0.0156	0.0312	0.0001	0.0156	0.0156	0.0156	0.0001	0.0156
SC50A	0.0001	0.0001	0.0156	0.0001	0.0156	0.0001	0.0156	0.0156	0.0156	0.0001
SC50B	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
SCAGR25	0.0312	0.0312	0.0312	0.5304	0.0156	0.0312	0.0624	0.0468	0.0312	0.0312
SCAGR7	0.0001	0.0156	0.0156	0.0156	0.0156	0.0156	0.0000	0.0156	0.0000	0.0156
SCFXM1	0.0312	0.0312	0.0312	0.0936	0.0156	0.0312	0.0780	0.0468	0.0156	0.0156
SCFXM3	0.1404	0.1560	0.1716	3.9936	0.1404	0.1404	0.4368	0.2028	0.1560	0.1404
SCORPION	0.0156	0.0156	0.0156	0.1248	0.0156	0.0156	0.0312	0.0312	0.0312	0.0156
SCRS8	0.0468	0.0624	0.0624	0.0468	0.0312	0.0624	0.0936	0.0780	0.0468	0.0624
SCSD1	0.0156	0.0156	0.0312	0.0624	0.0312	0.0156	0.0312	0.0468	0.0312	0.0156
SCSD6	0.0312	0.0468	0.0468	0.3588	0.0312	0.0468	0.0780	0.0936	0.0312	0.0468
SCSD8	0.1092	0.1248	0.1404	7.3632	0.0936	0.1092	0.2184	0.2184	0.1248	0.1092
SCTAP1	0.0156	0.0312	0.0312	0.0156	0.0156	0.0312	0.0468	0.0312	0.0156	0.0312
SCTAP2	0.1404	0.1560	0.1872	0.1716	0.1404	0.1560	0.2964	0.2184	0.1716	0.1560
SCTAP3	0.2652	0.2808	0.2964	26.0522	0.2496	0.2808	0.5304	0.3588	0.2652	0.2652
SHARE1B	0.0001	0.0001	0.0156	0.0001	0.0156	0.0156	0.0156	0.0156	0.0156	0.0156
SHARE2B	0.0001	0.0156	0.0001	0.0156	0.0001	0.0001	0.0001	0.0156	0.0001	0.0001
SHIP04L	0.0468	0.0624	0.0780	0.0468	0.0468	0.0624	0.1248	0.1248	0.0780	0.0780
SHIP04S	0.0468	0.0468	0.0468	0.0468	0.0468	0.0468	0.0780	0.0780	0.0468	0.0468
SHIP08L	0.1092	0.1560	0.1716	11.1385	0.1092	0.1404	0.2496	0.2340	0.1560	0.1248
SHIP08S	0.0624	0.0624	0.0624	1.6536	0.0468	0.0624	0.1092	0.1092	0.0624	0.0624
SHIP12L	0.1716	0.2184	0.2496	0.1716	0.1716	0.2028	0.3900	0.3432	0.2340	0.2028
SHIP12S	0.0624	0.0780	0.0780	2.9796	0.0624	0.0780	0.1248	0.1404	0.0780	0.0780
STOCFOR1	0.0156	0.0001	0.0156	0.0001	0.0001	0.0001	0.0156	0.0156	0.0156	0.0001
STOCFOR2	0.2808	0.3120	0.3276	90.6210	0.2808	0.2964	0.5616	0.3900	0.3432	0.2964
TRUSS	0.6240	0.6864	0.7332	211.3190	0.6084	0.6708	1.2792	0.9828	0.7800	0.6396
WOOD1P	0.1872	0.2184	0.2184	0.1872	0.1872	0.2028	0.9360	0.2340	0.1716	0.1872
WOODW	0.3588	0.4056	0.4368	0.3432	0.3432	0.3900	0.7488	0.5304	0.3900	0.3900
Average	2.0953	2.1704	2.1745	16.9700	2.0463	2.1235	4.0007	2.3449	2.4033	2.1557

13, 280–322 (1977)

[2] Bieling, J., Peschlow, P., Martini, P.: An efficient GPU implementation of the revised Simplex method. In: Proceddings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010), Atlanta, USA (2010)

[3] Curtis, A.R., Reid, J.K.: On the automatic scaling of matrices for Gaussian elimination. J. Inst. Math. Appl. 10, 118–124 (1972)

[4] de Buchet, J.: Experiments and statistical data on the solving of large-scale linear programs. In: Hertz, D.A., Melese, J. (eds.) Proceedings of the Fourth International Conference on Operational Research, pp. 3–13. Wiley-Interscience, New York (1966)

[5] Elble, J.M., Sahinidis, N.V.: Sparse Matrix Binormalization on a GPU. In: Proceedings of the 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing, Trondheim, Norway (2008)

[6] Elble, J.M., Sahinidis, N.V.: Scaling linear optimization problems prior to application of the simplex method. Computational Optimization and Applications 52:2, 345–371 (2012)

[7] Fulkerson, D.R., Wolfe, P.: An algorithm for scaling matrices. SIAM Rev. 4, 142-146 (1962)

[8] Gay, D.M.: Electronic mail distribution of linear programming test problems. Mathematical Pro-

Table 5. Scaling time of GPU-based implementations (secs)

Problem	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
25FV47	0.03100	0.02970	0.02891	0.65986	0.03122	0.03018	0.06200	0.04091	0.03121	0.04018
ADLITTLE	0.00001	0.00001	0.00151	0.00001	0.00001	0.00198	0.00149	0.00001	0.00145	0.00001
AFIRO	0.00001	0.00001	0.00001	0.00138	0.00001	0.00001	0.00001	0.00125	0.00001	0.00001
AGG	0.00200	0.00201	0.00190	0.00229	0.00248	0.00191	0.00537	0.00440	0.00205	0.00365
AGG2	0.00371	0.00665	0.00625	0.00277	0.00480	0.00382	0.00943	0.00501	0.00645	0.00393
AGG3	0.00547	0.00524	0.00886	0.00372	0.00870	0.00837	0.01761	0.01147	0.00799	0.00672
BANDM	0.00482	0.00235	0.00278	0.01265	0.00278	0.00243	0.00980	0.00504	0.00255	0.00607
BEACONFD	0.00001	0.00001	0.00179	0.00001	0.00001	0.00001	0.00223	0.00180	0.00190	0.00001
BGINDY	0.83950	0.87065	0.91531	0.91778	0.74331	0.60835	1.98650	1.02990	0.93329	0.76881
BGPRTR	0.00001	0.00001	0.00179	0.00001	0.00001	0.00001	0.00223	0.00180	0.00190	0.00001
BLEND	0.00001	0.00001	0.00001	0.00131	0.00145	0.00124	0.00135	0.00112	0.00001	0.00001
BNL1	0.00816	0.00814	0.01243	0.28449	0.00928	0.01023	0.02368	0.01377	0.01182	0.01193
BNL2	0.12500	0.14531	0.18299	27.31438	0.12210	0.23583	0.23551	0.19031	0.09340	0.23715
BRANDY	0.00242	0.00265	0.00271	0.00276	0.00002	0.00236	0.00685	0.00502	0.00330	0.00002
CRE_A	0.22409	0.24709	0.31339	43.40088	0.22181	0.22239	0.47963	0.34241	0.28947	0.27043
CRE_C	0.15557	0.19787	0.24019	32.25917	0.16531	0.18131	0.37619	0.25022	0.24357	0.18973
D2Q06C	0.24900	0.19874	0.25630	23.55395	0.17434	0.19170	0.30062	0.19092	0.16268	0.14766
DEGEN2	0.00592	0.00952	0.00546	0.07014	0.00596	0.00672	0.01091	0.00937	0.00868	0.00621
DEGEN3	0.06300	0.06846	0.07157	1.70134	0.06922	0.05070	0.15392	0.05400	0.09665	0.05439
E226	0.00187	0.00304	0.00153	0.00314	0.00001	0.00306	0.00285	0.00287	0.00127	0.00139
FFFFF800	0.00647	0.00591	0.00602	0.03454	0.00625	0.00377	0.01210	0.00836	0.01018	0.00644
ISRAEL	0.00001	0.00001	0.00134	0.00141	0.00001	0.00143	0.00260	0.00157	0.00137	0.00001
ITEST2	0.00001	0.00001	0.00001	0.00001	0.00001	0.00001	0.00001	0.00156	0.00001	0.00208
ITEST6	0.00001	0.00001	0.00183	0.00001	0.00001	0.00001	0.00001	0.00001	0.00001	0.00001
KLEIN1	0.00001	0.00232	0.00001	0.00001	0.00001	0.00001	0.00278	0.00001	0.00001	0.00001
KLEIN2	0.00687	0.00447	0.00763	0.00910	0.00604	0.00586	0.00958	0.00759	0.00904	0.00556
KLEIN3	0.02792	0.03638	0.04499	0.02953	0.03321	0.04527	0.08300	0.04250	0.03583	0.03473
LOTFI	0.00242	0.00001	0.00296	0.00329	0.00400	0.00192	0.01055	0.00191	0.00990	0.00231
MAROS-R7	0.40600	0.55283	0.52233	0.51442	0.45905	0.55706	2.22572	0.85959	0.67973	0.59845
OSA-07	0.81430	0.86469	0.92810	1.34225	0.88147	0.81303	1.55536	1.24668	1.07228	1.08673
OSA-14	3.50249	5.21370	6.07788	4.97106	4.13408	4.18020	7.18940	5.00709	5.13130	4.32704
OSA-30	15.26764	17.85277	25.61921	19.02405	18.00210	15.16495	21.30879	17.96421	21.81985	18.01645
QAP12	0.78000	0.97736	1.27751	1.01333	4.48332	0.78392	3.75597	0.74726	1.67142	0.96327
QAP15	2.89943	3.02863	3.27338	2.84412	2.65684	2.60497	5.36347	2.66738	3.07322	2.39131
QAP8	0.01600	0.02146	0.01990	0.01871	0.01672	0.02059	0.03562	0.02443	0.02799	0.02186
SC105	0.00001	0.00001	0.00001	0.00168	0.00001	0.00001	0.00001	0.00145	0.00216	0.00001
SC205	0.00254	0.00304	0.00215	0.00607	0.00002	0.00392	0.00338	0.00501	0.00003	0.01352
SC50A	0.00001	0.00001	0.00136	0.00001	0.00123	0.00001	0.00107	0.00121	0.00098	0.00001
SC50B	0.00001	0.00001	0.00001	0.00001	0.00001	0.00001	0.00001	0.00001	0.00001	0.00001
SCAGR25	0.00804	0.00535	0.00828	0.06918	0.00876	0.00389	0.01681	0.00595	0.01391	0.00376
SCAGR7	0.00001	0.00159	0.00176	0.00145	0.00218	0.00130	0.00000	0.00131	0.00000	0.00118
SCFXM1	0.00366	0.00297	0.00464	0.00915	0.00213	0.00325	0.00895	0.00495	0.00155	0.00197
SCFXM3	0.03200	0.05455	0.04656	1.21183	0.06618	0.03744	0.10954	0.04719	0.02753	0.02265
SCORPION	0.00226	0.00269	0.00290	0.02504	0.00454	0.00236	0.01177	0.00413	0.00999	0.00275
SCRS8	0.00677	0.00937	0.01033	0.00989	0.00583	0.01158	0.02054	0.01781	0.00823	0.01046
SCSD1	0.00187	0.00213	0.00319	0.00708	0.00278	0.00202	0.00254	0.00783	0.00231	0.00200
SCSD6	0.00336	0.00425	0.00495	0.03095	0.00395	0.00357	0.01083	0.00768	0.00513	0.00360
SCSD8	0.01008	0.01396	0.01334	0.69999	0.00868	0.01182	0.02373	0.02468	0.01489	0.01221
SCTAP1	0.00184	0.00322	0.00330	0.00197	0.00148	0.00325	0.00462	0.00283	0.00168	0.00305
SCTAP2	0.03200	0.04538	0.07130	0.07595	0.08251	0.04657	0.69984	0.04641	-0.64884	0.03282
SCTAP3	0.06660	0.05087	0.05360	5.02743	0.05244	0.06257	0.16913	0.06608	0.21182	0.04411
SHARE1B	0.00001	0.00001	0.00154	0.00001	0.00157	0.00156	0.00157	0.00155	0.00135	0.00164
SHARE2B	0.00001	0.00141	0.00001	0.00151	0.00001	0.00001	0.00001	0.00152	0.00001	0.00001
SHIP04L	0.01600	0.01384	0.02318	0.00888	0.01075	0.01062	0.02009	0.01844	0.01760	0.01574
SHIP04S	0.00637	0.00583	0.00746	0.00494	0.00673	0.00470	0.00943	0.00819	0.00529	0.00487
SHIP08L	0.02617	0.03316	0.03363	1.73876	0.01629	0.02202	0.04672	0.03749	0.03070	0.02039
SHIP08S	0.00737	0.00932	0.00776	0.30476	0.00600	0.01800	0.01152	0.02516	0.00643	0.01393
SHIP12L	0.03466	0.06528	0.07758	0.03530	0.04687	0.03066	0.14144	0.07095	0.27420	0.03767
SHIP12S	0.01454	0.01762	0.01879	0.71014	0.01563	0.02286	0.02490	0.03909	0.01881	0.02383
STOCFOR1	0.00290	0.00002	0.00219	0.00002	0.00002	0.00003	0.00430	0.00415	0.00563	0.00003
STOCFOR2	0.05713	0.08173	0.05732	19.45784	0.04200	0.09733	0.07296	0.15298	0.05906	0.10596
TRUSS	0.17747	0.16288	0.17709	62.00310	0.14148	0.14678	0.29365	0.15760	0.25373	0.12771
WOOD1P	0.02297	0.02509	0.02685	0.01903	0.02965	0.02268	0.11958	0.02635	0.02064	0.01798
WOODW	0.10000	0.07539	0.16596	0.04957	0.46377	0.04625	0.33944	0.06574	0.21618	0.03914
Average	0.40762	0.4851	0.6354	3.9298	0.5198	0.4119	0.7408	0.4937	0.5625	0.4651

gramming Society COAL Newsletter 13, 10–12 (1985)
[9] Hamming, R.W.: Introduction to Applied Numerical Analysis. McGraw-Hill, New York (1971)
[10] Jung, J.H., OLeary, D.P.: Implementing an interior point method for linear programs on a CPU-GPU system. Electronic Transaction on Numerical Analysis 28, 174-189 (2008)
[11] Lalami, M.E., Boyer, V., El-Baz, D.: Efficient Implementation of the Simplex Method on a CPU-GPU System. In: Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW 2011), pp. 1999–2006. Washington, USA (2011)
[12] Larsson, T.: On scaling linear programsSome experimental results. Optimization 27, 335-373 (1993)
[13] Meyer, X., Albuquerque, P., Chopard, B.: A multi-GPU implementation and performance model for the standard simplex method. In: Proceedings 1st International Symposium and 10th Balkan Conference on Operational Research, pp. 312–319. Thessaloniki, Greece (2011)
[14] Ordóñez, F., Freund, R.: Computational experience and the explanatory value of condition measures for linear optimization. SIAM J. Optimization 14:2, 307–333 (2003)
[15] Smith, E., Gondzio, J., Hall, J.: GPU acceleration of the matrix-free interior point method. In: Parallel Processing and Applied Mathematics, pp. 681–689. Springer Berlin Heidelberg (2012)
[16] Spampinato, D.G., Elster, A.C.: Linear optimization on modern GPUs. In: Proceedings of the 23rd

Table 6. Speedup obtained by the GPU-based implementations

Problem	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
25FV47	6.0387	6.8287	7.0147	7.4707	5.9967	6.2027	5.7867	6.1007	6.9987	5.0467
ADLITTLE	10.0000	8.9340	10.3500	7.8540	10.9040	7.8740	10.5000	7.4800	10.7300	6.9740
AFIRO	10.0000	11.3980	10.8080	11.3080	11.9780	13.2640	11.1480	12.4960	9.2560	14.2500
AGG	7.8000	7.7720	8.2080	6.8020	6.2960	8.1840	5.8100	7.0840	7.6180	8.5580
AGG2	8.4000	9.3880	9.9920	11.2600	9.7440	12.2360	9.9220	12.4560	9.6680	11.8980
AGG3	8.5500	8.9300	7.0420	8.3920	5.3780	7.4580	5.3160	6.7980	5.8560	6.9600
BANDM	6.4700	6.6400	5.6160	6.1660	5.6140	6.4100	6.3680	6.1960	6.1120	5.1380
BEACONFD	10.0000	9.8440	8.7280	9.6720	8.5120	9.8360	6.9840	8.6740	8.1920	9.4120
BGINDY	4.5713	4.5153	4.4313	4.1984	5.1209	6.5134	3.9815	4.1352	4.5131	5.0931
BGPRTR	10.0000	10.0000	10.0000	10.0000	10.0000	8.7456	8.7145	10.0000	5.6190	10.0000
BLEND	10.0000	10.2420	9.3160	11.9360	10.7220	12.5440	11.5300	13.9640	12.0640	13.5220
BNL1	9.5600	11.4980	8.7860	9.7060	8.4060	9.1500	7.2460	9.0620	9.2380	7.8440
BNL2	3.8688	3.8648	3.3248	3.9048	3.8328	2.1168	4.1068	3.2788	6.0128	2.1708
BRANDY	6.4500	5.8840	5.7500	5.6480	5.0500	6.6000	4.5580	6.2120	4.7280	5.1940
CRE_A	5.0123	4.8614	4.1316	4.1095	5.1341	5.0919	3.9084	3.8013	4.1503	4.1985
CRE_C	5.6155	4.6515	4.0918	3.9156	5.0016	4.9814	3.9916	4.0016	4.0419	5.0145
D2Q06C	5.2000	6.6720	5.4780	6.8920	7.2480	7.0800	8.7180	7.8440	9.3020	8.9800
DEGEN2	7.9000	6.5520	8.5680	6.4500	7.8520	6.9640	8.5800	6.6600	7.1900	7.5400
DEGEN3	11.3905	10.4825	10.0265	11.6725	9.6905	13.5385	8.7165	14.1545	8.5545	12.3325
E226	8.3400	10.2520	10.2100	9.9380	9.8120	10.2100	10.9520	10.8640	12.2960	11.1880
FFFFF800	7.2300	7.9160	7.7680	9.4840	7.4880	8.2680	6.4480	7.4600	4.5960	7.2720
ISRAEL	10.0000	11.6960	11.6160	11.0940	11.0160	10.9360	11.9880	9.9120	11.3560	9.1300
ITEST2	10.0000	10.0000	10.0000	10.0000	10.0000	10.0000	10.0000	10.0000	6.7145	7.5145
ITEST6	10.0000	10.0000	8.5145	10.0000	10.0000	10.0000	10.0000	10.0000	10.0000	10.0000
KLEIN1	10.0000	6.7193	10.0000	10.0000	10.0000	10.0000	5.6189	10.0000	10.0000	10.0000
KLEIN2	6.8131	6.9814	6.1341	5.1416	5.1625	7.9810	6.5147	6.1620	5.1782	5.6134
KLEIN3	6.1452	5.1461	4.1613	5.8120	5.1679	4.1356	4.1349	5.1394	6.5303	5.3901
LOTFI	6.4590	8.4370	5.2730	9.4870	3.8990	8.1190	2.9570	8.1690	1.5750	6.7410
MAROS-R7	7.1468	5.5308	6.0628	5.6708	6.4228	5.4888	6.8688	3.7748	5.6228	4.9528
OSA-07	5.0193	4.9252	4.7736	3.0450	4.5306	5.2540	5.3760	4.1044	4.8212	4.0650
OSA-14	5.6120	3.7880	3.3264	3.9572	4.5848	4.7432	5.4572	4.3244	4.3578	4.6224
OSA-30	5.3418	4.7614	3.2516	4.2944	4.4394	5.4060	6.5708	4.9622	4.1780	4.6330
QAP12	2.9800	2.4740	1.9660	2.3400	0.5080	3.1840	1.3000	3.7160	1.7360	2.4940
QAP15	3.9815	3.8065	3.6315	4.0315	4.1865	4.4615	4.4065	4.7665	4.7665	5.1015
QAP8	7.8001	7.2681	7.8381	6.6721	7.4621	6.8181	7.4461	8.3021	5.5741	6.4221
SC105	10.0000	11.0840	9.5820	9.2820	7.7420	9.9380	6.7540	10.7240	7.2060	10.3000
SC205	6.1520	5.1300	7.2700	5.1420	5.3840	3.9820	4.6220	3.1160	3.3200	1.1540
SC50A	10.0000	9.6300	11.4580	10.9980	12.6960	11.8680	14.5700	12.8860	15.8820	12.6580
SC50B	10.0000	11.9940	10.8340	12.8600	11.4960	12.5740	12.8820	11.9620	10.9160	12.8820
SCAGR25	3.8790	5.8270	3.7690	7.6670	1.7810	8.0270	3.7110	7.8650	2.2430	8.3050
SCAGR7	10.0000	9.7960	8.8400	10.7820	7.1580	12.0400	7.3320	11.8760	8.9480	13.2460
SCFXM1	8.5230	10.5090	6.7270	10.2250	7.3410	9.6110	8.7170	9.4610	10.0650	7.9310
SCFXM3	4.3875	2.8595	3.6855	3.2955	2.1215	3.7495	3.9875	4.2975	5.6675	6.1975
SCORPION	6.9150	5.8070	5.3730	4.9850	3.4370	6.6010	2.6510	7.5570	3.1230	5.6810
SCRS8	6.9158	6.6578	6.0418	4.7298	5.3518	5.3898	4.5578	4.3798	5.6878	5.9658
SCSD1	8.3520	7.3180	9.7700	8.8100	11.2100	7.7260	12.2960	5.9800	13.5100	7.7860
SCSD6	9.2930	11.0230	9.4570	11.5930	7.9050	13.1250	7.2010	12.1850	6.0850	12.9890
SCSD8	10.8350	8.9370	10.5230	10.5190	10.7870	9.2370	9.2050	8.8510	8.3790	8.9410
SCTAP1	8.4630	9.6830	9.4470	7.9310	10.5110	9.6030	10.1310	11.0270	9.2690	10.2250
SCTAP2	4.3875	3.4375	2.6255	2.2595	1.7015	3.3495	0.4235	4.7055	-0.2645	4.7535
SCTAP3	3.9820	5.5200	5.5300	5.1820	4.7600	4.4880	3.1360	5.4300	1.2520	6.0120
SHARE1B	10.0000	9.1780	10.1200	10.3520	9.9540	9.9740	9.9600	10.0960	11.5600	9.5200
SHARE2B	10.0000	11.0680	8.0480	10.3360	7.5160	9.6080	8.4120	10.2340	7.6420	9.7040
SHIP04L	2.9250	4.5090	3.3650	5.2690	4.3550	5.8730	6.2110	6.7670	4.4310	4.9570
SHIP04S	7.3520	8.0300	6.2760	9.4780	6.9580	9.9480	8.2720	9.5200	8.8420	9.6140
SHIP08L	4.1720	4.7040	5.1020	6.4060	6.7020	6.3760	5.3420	6.2420	5.0820	6.1220
SHIP08S	8.4720	6.6980	8.0380	5.4260	7.7980	3.4660	9.4800	4.3400	9.7100	4.4800
SHIP12L	4.9514	3.3454	3.2174	4.8614	3.6614	6.6154	2.7574	4.8374	0.8534	5.3834
SHIP12S	4.2918	4.4258	4.1518	4.1958	3.9918	3.4118	5.0118	3.5918	4.1478	3.2738
STOCFOR1	5.3713	5.1493	7.1313	4.8893	5.5553	3.8153	3.6313	3.7593	2.7693	3.2053
STOCFOR2	4.9153	3.8173	5.7153	4.6573	6.6853	3.0453	7.6973	2.5493	5.8113	2.7973
TRUSS	3.5162	4.2142	4.1402	3.4082	4.3002	4.5702	4.3562	6.2362	3.0742	5.0082
WOOD1P	8.1492	8.7032	8.1332	9.8372	6.3132	8.9432	7.8272	8.8792	8.3132	10.4112
WOODW	3.5880	5.3800	2.6320	6.9240	0.7400	8.4320	2.2060	8.0680	1.8040	9.9640
Average	7.1795	7.2359	6.9249	7.3535	6.7668	7.5810	6.8010	7.4918	6.6324	7.4177

IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009), Rome, Italy (2009)

[17] Tomlin, J.A.: On scaling linear programming problems. Math. Program. Stud. 4, 146-166 (1975)