Efficient GPU-based Implementations of Simplex Type Algorithms

Nikolaos Ploskas, Nikolaos Samaras*

---

*Corresponding author
Address: Department of Applied Informatics, School of Information Sciences, University of Macedonia, 156 Egnatia Str., 54006 Thessaloniki, Greece
Email Address: samaras@uom.gr
Phone: +30 2310 891866; Fax: +30 2310 891879

Preprint submitted to Applied Mathematics and Computation *September 19, 2014*

Efficient GPU-based Implementations of Simplex Type Algorithms

Nikolaos Ploskas, Nikolaos Samaras*

Abstract

Recent hardware advances have made it possible to solve large scale Linear Programming problems in a short amount of time. Graphical Processing Units (GPUs) have gained a lot of popularity and have been applied to linear programming algorithms. In this paper, we propose two efficient GPU-based implementations of the revised simplex algorithm and a primal-dual exterior point simplex algorithm. Both parallel algorithms have been implemented in MATLAB using MATLAB's Parallel Computing Toolbox. Computational results on randomly generated optimal sparse and dense linear programming problems and on a set of benchmark problems (netlib, kennington, Mészáros) are also presented. The results show that the proposed GPU implementations outperform MATLAB's interior point method.

Keywords: Linear Programming; Simplex Type Algorithms; Graphical Processing Unit; Parallel Computing; MATLAB

1. Introduction

Linear Programming (LP) is perhaps the most important and well–studied optimization problem. Lots of real world problems can be formulated as Linear Programming problems (LPs). LP is the process of minimizing or maximizing a linear objective function $z = \sum_{j=1}^{n} c_j x_j$ to a number of linear equality and inequality constraints. Simplex algorithm is the most widely used method for solving LPs. Consider the following linear program (LP.1) in the standard form:

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

*Corresponding author
Address: Department of Applied Informatics, School of Information Sciences, University of Macedonia, 156 Egnatia Str., 54006 Thessaloniki, Greece
Email Address: samaras@uom.gr
Phone: +30 2310 891866; Fax: +30 2310 891879

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 region. The dual problem associated with the (LP.1) is presented in (DP.1):

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

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

The explosion in computational power of hardware has made it possible to solve large LPs in a short amount of time in PCs. In the past two decades, many parallel implementations of LP algorithms have been proposed. Recently, Graphical Processing Units (GPUs) have gained a lot of popularity and researchers implemented various GPU-based LP algorithms. Using GPU computing for solving large-scale LPs is a great challenge due to the capabilities of GPU architectures. Jung and O'Leary [17] proposed a CPU-GPU implementation of the Interior Point Method for dense LPs and their computational results showed a speedup up to 1.4 on medium-sized Netlib LPs [11] compared to the corresponding CPU implementation. Spampinato and Elster [41] presented a GPU-based implementation of the revised simplex algorithm with NVIDIA CUBLAS [27] and NVIDIA LAPACK libraries [28]. Their implementation showed a maximum speedup of 2.5 on large random LPs compared to the corresponding CPU implementation. Bieling et al. [3] also proposed a parallel implementation of the revised simplex algorithm on GPU. They compared their GPU-based implementation with the serial GLPK solver and reported a maximum speedup of 18 in single precision. Lalami et al. [19] proposed a parallel implementation of the tableau simplex algorithm on a CPU-GPU system. Their computational results on randomly generated dense problems showed a maximum speedup of 12.5 compared to the corresponding CPU implementation. Lalami et al. [20] extended their previous work [19] on a multi-GPU implementation and their computational results on randomly generated dense problems showed a maximum speedup of 24.5. Li et al. [22] presented a GPU-based parallel algorithm, based on Gaussian elimination, for large scale LPs that outperforms the CPU-based algorithm. Meyer et al. [25] proposed a mono- and a multi-GPU implementation of the tableau simplex algorithm and compared their implementation with the serial CLP solver. Their implementation outperformed CLP solver on large sparse LPs. Gade-Nielsen and Jorgensen [10] presented a GPU-based interior point method and their computational results showed a speedup of 6 on randomly generated dense LPs and a speedup of 2 on randomly generated sparse LPs compared to the MATLAB's built-in function linprog. Smith et al. [40] proposed a GPU-based interior point method and their computational results showed a maximum speedup of 1.27 on large sparse LPs compared to the corresponding multi-core CPU implementation.

To the best of our knowledge, these are all the papers that proposed a GPU-

based implementation of a LP algorithm. No parallel implementation of the simplex algorithm has yet offered significantly better performance relative to an efficient sequential simplex solver [14]; at least not in all type of LPs (sparse or dense, randomly generated or benchmark). For this reason, there is a strong motivation for exploring how the simplex type algorithms exploit high performance computing architectures. Yet, there has been no attempt to implement a GPU-based exterior point simplex algorithm. The novelty of this paper is that we propose GPU-based implementations of the revised simplex algorithm and a primal-dual exterior point simplex algorithm. The computational results demonstrate that the proposed GPU implementations outperform MATLAB's interior point method on randomly generated optimal dense and sparse LPs and on a set of benchmark problems (netlib, kennington, Mészáros).
The structure of the paper is as follows. In Section 2, the background of this paper is presented. In Section 3, the revised simplex algorithm and a primal-dual exterior point simplex algorithm are described. Section 4 presents the GPU-based implementations of these algorithms. In Section 5, the computational comparison of the GPU-based implementations with MATLAB's interior point method on a set of randomly generated optimal dense and sparse LPs and on a set of benchmark problems (netlib, kennington, Mészáros) is presented. Finally, the conclusions of this paper are outlined in section 6.

2. Background

The increasing size of real life LPs demands more computational power and parallel computing capabilities. Recent hardware advances have made it possible to solve large LPs in a short amount of time. LP algorithms have been parallelized many times. Some of these implementations use dense matrix algebra (parallelization of the tableau simplex algorithm using dense matrix algebra or parallelization of the revised simplex algorithm using dense basis inverse), other use sparse matrix algebra, while some other use special LP algorithms variants. Furthermore, with the advances made in hardware, GPUs have been widely applied to scientific computing applications. GPU is utilized for data parallel and computationally intensive portions of an algorithm. Two major general purpose programming languages exist for GPUs, CUDA (Compute Unified Device Architecture) [26] and OpenCL (Open Computing Language) [43]. These programming languages are based on the stream processing model. CUDA was introduced in late 2006 and is only available with NVIDIA GPUs, while OpenCL was introduced in 2008 and is available on GPUs of different vendors and even on CPUs. Table 1 presents a representative list of parallel (CPU- and GPU-based) simplex implementations in chronological order.

Date	Author	Algorithm	Hardware Platform	Speedup and Comments	Sparse or Dense	CPU or GPU
1988	Stunkel and Reed [42]	Tableau simplex algorithm	16 processor Intel hyper cube	Between 8 and 12 on small Netlib LPs	Dense	CPU
1991	Cvetanovic et al. [6]	Tableau simplex algorithm	16 processor shared memory machine	Up to 12 on one large and one small Netlib LP	Dense	CPU
1994	Ho and Sundarraj [15]	Revised simplex algorithm	Intel iPSC/2 and Sequent Balance 8000	An average speedup of 1.5 on medium sized Netlib and other LPs	Sparse	CPU
1995	Eckstein et al. [9]	Tableau simplex algorithm with the most-negative reduced cost and with the steepest edge pivot rules and revised simplex algorithm with explicit inverse	Connection Machine CM-2 with thousands of processors	Iteration speed is superior to MINOS 5.4 on some Netlib LPs	Dense	CPU
1995	Lentini et al. [21]	Tableau simplex algorithm	8 transputers	Between 0.5 and 2.7 on small medium sized Netlib LPs and 5.2 on a problem with a relatively large column-row ratio compared to a sparse revised simplex solver running on a mainframe	Sparse	CPU
1995	Shu [39]	Revised simplex algorithm	Intel iPSC/2	Up to 17 on small Netlib LPs and up to 8 on large Kennington LPs	Sparse	CPU
1996	Hall and McKinnon [13]	Revised simplex algorithm with Devex pivot rule	6 processors of a Cray T3D	3 in terms of iteration speed on a medium sized Netlib LP	Sparse	CPU

1996	Thomadakis and Liu [44]	Tableau simplex algorithm with the steepest edge pivot rule	128 x 128 cores MasPar machine	Up to 1000 on large random LPs	Dense	CPU
1998	Hall and McKinnon [14]	Revised simplex algorithm	Cray T3D	Between 2.5 and 4.8 on medium sized Netlib LPs	Sparse	CPU
2000	Bixby and Martin [4]	Revised simplex algorithm with the steepest edge pivot rule and steepest edge variants	Different platforms, including heterogeneous workstation clusters, Sun S20-502, Silicon Graphics multi-processors and an IBM SP2	Between 1 and 3 on Netlib LPs	Sparse	CPU
2006	Badr et al. [1]	Tableau simplex algorithm	8 processors SMP	Up to 5 on random small dense LPs	Dense	CPU
2008	Jung and OLeary [17]	Interior point method (Mehrotras predictor-corrector algorithm)	NVIDIA GeForce 7800 GTX and Intel Xeon 3GHz	Up to 1.4 on medium-sized Netlib LPs compared to the corresponding CPU implementation	Dense	GPU
2009	Yarmish and Slyke [46]	Tableau simplex algorithm	7 workstations	7 in terms of iteration speed on a large random dense LP	Dense	CPU
2009	Spampinato and Elster [41]	Revised simplex algorithm with dense PFI basis update	NVIDIA GeForce GTX 280	Up to 2.5 on large random LPs compared to the corresponding CPU implementation running on an Intel Core 2 Quad 2.83 GHz	Dense	GPU
2010	Bieling et al. [3]	Revised simplex algorithm with the PFI basis update and the steepest edge pivot rule	NVIDIA GeForce 9600 GT	Up to 18 in single precision on random LPs compared to the serial GLPK solver running on an Intel Core 2 Duo 3GHz	Dense	GPU

2011	Lalami et al. [19]	Tableau simplex algorithm	NVIDIA GeForce GTX 260	Up to 12.5 on large random dense LPs compared to the corresponding CPU implementation running on an Intel Xeon 3 GHz	Dense	GPU
2011	Lalami et al. [20]	Tableau simplex algorithm	2 NVIDIA Tesla C2050	Up to 24.5 on large random dense LPs compared to the corresponding CPU implementation running on an Intel Xeon 2.66 GHz	Dense	GPU
2011	Li et al. [22]	Tableau simplex algorithm	NVIDIA GeForce 9800GT	Up to 120 on large LPs compared to the corresponding CPU implementation running on an Intel Core 2 Duo 1.6 GHz	Dense	GPU
2011	Meyer et al. [25]	Tableau simplex algorithm	4 NVIDIA Tesla S1070	The speedup is not explicitly stated. Compared a multi-GPU implementation to the serial CLP solver on dense LPs	Dense	GPU
2012	Gade-Nielsen and Jorgensen [10]	Interior point method (Mehrotras predictor-corrector algorithm)	NVIDIA Tesla C2050	6 on dense random LPs and 2 on sparse random LPs compared to the MATLAB's built-in function linprog running on an Intel Core i7 2.8GHz	Sparse	GPU
2012	Smith et al. [40]	Interior point method (matrix free method)	NVIDIA Tesla C2070	Up to 1.27 on large sparse LPs compared to the corresponding multi-core CPU implementation running on an AMD Opteron 2378	Sparse	GPU

Table 1: LP Algorithms' Parallelization

3. Simplex Type Algorithms

In this section, the revised simplex algorithm and a primal-dual exterior point simplex algorithm are described.

3.1. Revised Simplex Algorithm

The most well-known method for the optimization problem is the simplex algorithm developed by George B. Dantzig [8]. The simplex algorithm begins with a primal feasible basis and uses pricing operations until an optimum solution is computed. It also guarantees monotonicity of the objective value. It has been proved that the expected number of iterations in the solution of a linear problem is polynomial [5]. Moreover, the worst case complexity has exponential behavior [18].
Using a basic partition (B, N), the linear problem in (LP.1) can be written as shown in (LP.2).

$$\begin{array}{ll} min & c_B^T x_B + c_N^T x_N \\ subject\ to & A_B x_B + A_N x_N = b \\ & x_B, x_N \geq 0 \end{array} \qquad (LP.2)$$

In (LP.2), A_B is an $m \times m$ non-singular sub-matrix of A, called basic matrix or basis. The columns of A which belong to subset B are called basic and those which belong to N are called non basic. The solution $x_B = (A_B)^{-1} b, x_N = 0$ is called a basic solution. A solution $x = (x_B, x_N)$ is feasible iff $x > 0$. Otherwise, (LP.2) is infeasible. In order to initialize the simplex algorithm, a basic feasible solution must be available. The solution of (DP.1) is computed by the relation $s = c - A^T w$, where $w = (c_B)^T (A_B)^{-1}$ are the simplex multipliers and s are the dual slack variables. The basis A_B is dual feasible iff $s \geq 0$.
In each iteration, simplex algorithm interchanges a column of matrix A_B with a column of matrix A_N and constructs a new basis $A_{\overline{B}}$. A formal description of the revised simplex algorithm is given in Table 2.

3.2. Primal-Dual Exterior Point Simplex Algorithm

Since Dantzig's initial contribution, researchers have made many efforts in order to enhance the performance of simplex algorithm. In the 1990s a totally different approach arose; namely Exterior Point Simplex Algorithm (EPSA). The first implementation of an EPSA was introduced for the assignment problem [29]. The main idea of EPSA is that it moves in the exterior of the feasible region and constructs basic infeasible solutions instead of feasible solutions calculated by the simplex algorithm. Although EPSA outperforms the original

Table 2: Revised Simplex Algorithm

Step 0. *(Initialization).*
Start with a feasible partition (B, N). Compute $(A_B)^{-1}$ and vectors x_B, w and s_N.
Step 1. *(Test of optimality).*
if $s_N \geq 0$ then STOP. (LP.2) is optimal.
else
Choose the index l of the entering variable using a pivoting rule.
Variable x_l enters the basis.
Step 2. *(Minimum ratio test).*
Compute the pivot column $h_l = (A_B)^{-1} A_l$.
if $h_l \leq 0$ then STOP. (LP.2) is unbounded.
else
Choose the leaving variable $x_{B[r]} = x_k$ using the following relation:

$$x_{B[r]} = \frac{x_{B[r]}}{h_{il}} = min \left\{ \frac{x_{B[i]}}{h_{il}} : h_{il} < 0 \right\}$$

Step 3. *(Pivoting).*
Swap indices k and l. Update the new basis inverse $(A_{\overline{B}})^{-1}$, using a basis update scheme.
Go to Step 1.

simplex algorithm, it also has some computational disadvantages. The main disadvantage is that in many LPs, EPSA can follow a path, which steps away from the optimal solution. This drawback can be avoided if the exterior path is replaced with a dual feasible simplex path. The most effective types of EPSA algorithms are the primal-dual versions. It has been observed that replacing the exterior path of an EPSA with a dual feasible simplex path results in an algorithm free from the computational disadvantages of EPSA [30]. A more effective approach is the Primal-Dual Exterior Point Simplex Algorithm (PDEPSA) [38]. PDEPSA can deal with the problems of stalling and cycling more effectively and as a result improves the performance of the primal dual exterior point algorithms. The advantage of PDEPSA stems from the fact that it uses an interior point in order to compute the leaving variable in contrast to primal dual exterior point algorithms which use a boundary point. A formal description of the revised simplex algorithm is given in Table 3. For a full description of PDEPSA see [38].

PDEPSA needs a dual feasible basic partition (B, N) to start. If the initial basis B is not dual feasible, then we can apply the algorithm to a big-M problem. More information about this procedure can be found in [30].

4. GPU-based Implementations

This section presents the GPU architecture of the implementations and the GPU-based implementations of the simplex type algorithm presented in Section 3.

Table 3: Primal-Dual Exterior Point Simplex Algorithm

Step 0. *(Initialization).*
A) Start with a dual feasible basic partition (B, N) and an interior point $y > 0$ of (LP.2).
Set:

$$P = N, Q = \varnothing$$

and compute

$$x_B = (A_B)^{-1} b,\ w^T = (c_B)^T (A_B)^{-1},\ s_N = (c_N)^T - w^T A_N$$

B) Compute the direction d_B from the relation: $d_B = y_B - x_B$
Step 1. *(Test of optimality and choice of the leaving variable).*
if $x \geq 0$ then STOP. (LP.2) is optimal.
else
Choose the leaving variable $x_k = x_{B[r]}$ from the relation:

$$a_l = \frac{x_{B[r]}}{-d_{B[r]}} = \max\left\{\frac{x_{B[r]}}{-d_{B[r]}} : d_{B[i]} > 0 \wedge x_{B[i]} < 0\right\}$$

Step 2. *(Computation of the next interior point).*
Set:

$$a = \frac{a_l + 1}{2}$$

Compute the interior point: $y_B = x_B + a d_B$
Step 3. *(Choice of the entering variable).*
Set: $H_{rN} = \left(A_B^{-1}\right)_{r.} A_{.N}$.
Choose the entering variable x_l from the relation:

$$\frac{-s_l}{H_{rN}} = \min\left\{\frac{-s_l}{H_{rN}} : H_{rj} \wedge j \in N\right\}$$

Compute the pivoting column: $h_l = (A_B)^{-1} A_{.l}$
if $l \in P$ then
$P \leftarrow P \setminus \{l\}$
else
$Q \leftarrow Q \setminus \{l\}$
Step 4. *(Pivoting).*
Set:

$$B[r] = l \text{ and } Q \leftarrow Q \cup \{k\}$$

Using the new partition (B, N) where $N = (P, Q)$, compute the new basis inverse A_B^{-1} and the variables x_B, w, and s_N.
Go to step $0B$.

4.1. GPU Architecture

This section briefly describes the architecture of an NVIDIA GPU in terms of hardware and software. GPU is a multi-core processor having thousands of threads running concurrently. 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, concerning the equipment utilized in the computational study presented in Section 5, a high-end Core i7 processor with 3.46 GHz delivers up to a peak of 55.36 GFLOPs, while a high-end

NVIDIA Quadro 6000 delivers up to 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 is executed on the CPU and another that is executed on the GPU. The CPU should be viewed as the host device, while the GPU should be viewed as co-processor. 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 grid of thread blocks. A kernel is the resulting code after the compilation. NVIDIA Quadro 6000, which was used in our computational experiment, consists of 14 stream processors (SP) with 32 cores each, resulting to 448 total cores. A typical use of a stream is that the GPU schedules a memory copy of results from CPU to GPU, a kernel launch and a copy of results from the GPU to CPU. A high level description of the GPU architecture is shown in Figure 1.

In this paper, both parallel algorithms have been implemented in MATLAB using MATLAB's Parallel Computing Toolbox [24]. MATLAB's Parallel Computing Toolbox provides adequate tools for the solution of computationally and data-intensive problems using multicore processors, GPUs and computer clusters. The toolbox provides a data structure called GPUArray, which is a special array that let users perform computations on CUDA-enabled NVIDIA GPUs. Furthermore, existing CUDA-based GPU kernels can be executed directly from MATLAB. Finally, multiple GPUs can be utilized using MATLAB workers in Parallel Computing Toolbox and Distributed Computing Server.
Prior to the presentation of the GPU-based implementations of the revised simplex algorithm and primal-dual exterior point simplex algorithm, we should describe two specific steps that are part of both implementations. The first one is the step where the algorithm determines if the linear problem is optimal in order to terminate its' execution. The GPU calculates reduced costs and stores in a flag variable the minimum of them. This flag variable is transferred to the CPU and the CPU determines if the linear problem is optimal. If the linear problem is optimal (i.e. the flag variable contains a positive value), the algorithm terminates, while if it is not the GPU continues finding the index of the entering variable. Similarly, the second step is the one where the algorithm determines if the linear problem is unbounded in order to terminate its' execution. The GPU finds the index of the entering variable and then transfers to the CPU another flag variable. The CPU checks the flag variable to determine if the linear problem is unbounded (i.e. the flag variable is null) in order to terminate the algorithm. Otherwise, the GPU continues finding the index of the leaving variable. On both steps, we preferred to implement the if statements in the

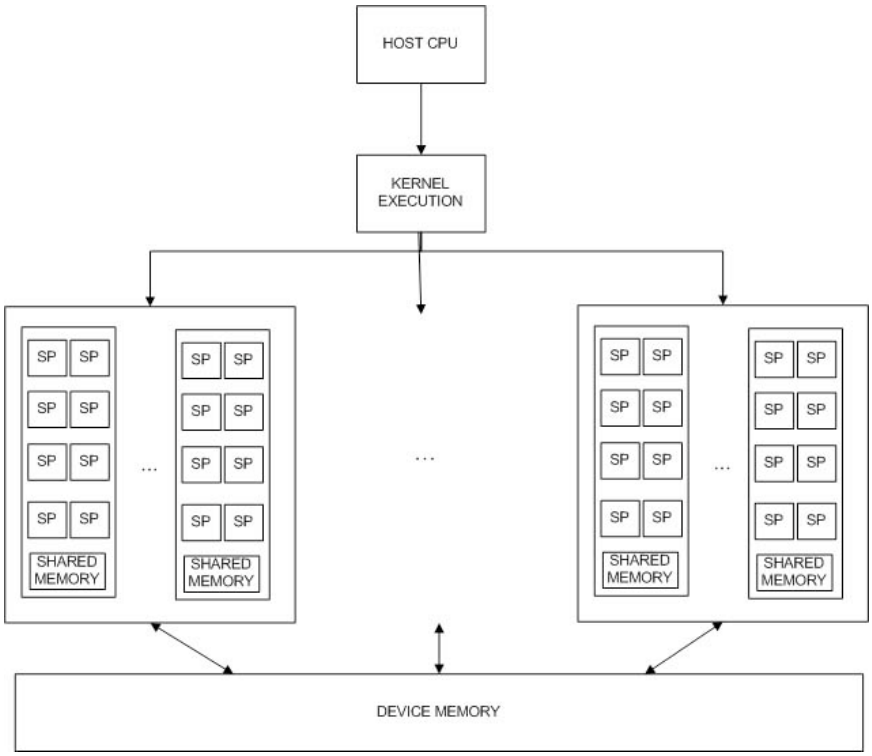

Figure 1: High Level Description of GPU Architecture

CPU, because the GPU is not suitable for logical branching [16]. This divergence problem exists in all modern GPUs [47]. We concluded to this decision by computationally comparing the other two alternatives: (i) The GPU calculates the reduced costs and sends the vector of the reduced costs to the CPU and the CPU determines if the linear problem is optimal, and (ii) The GPU calculates the reduced costs and determines itself if the linear problem is optimal. In order to justify this decision, both GPU-based implementations were executed for all three scenarios (the one chosen and the two alternatives) over a $4,000$ x $4,000$ randomly generated optimal dense LP. The total execution time for the revised simplex algorithm is: (i) 154.06, if we transfer the flag variable, (ii) 329.41, if the GPU calculates the reduced costs and sends the vector of the reduced costs to the CPU and the CPU determines if the linear problem is optimal, and (iii) 512.35, if the GPU calculates the reduced costs and determines itself if the linear problem is optimal. Respectively, the total execution time for the primal-dual exterior point simplex algorithm is: (i) 49.34, if we transfer the flag variable, (ii) 174.17, if the GPU calculates the reduced costs and sends the vector of the reduced costs to the CPU and the CPU determines if the linear problem is optimal, and (iii) 298.45, if the GPU calculates the reduced costs and determines itself if the linear problem is optimal.

4.2. Implementation of the GPU-Based Revised Simplex Algorithm

Figure 2 presents the process that is performed in the GPU-based implementation of the revised simplex algorithm. In the first step, the CPU initializes the algorithm by reading all the necessary data. In the second step, the CPU transfers the adequate variables (A, b, and c) to the GPU and the GPU scales the linear problem. In the third step, the GPU computes a feasible solution and transfers to the CPU a flag variable. The CPU checks the flag variable to determine if the linear problem is optimal. If the linear problem is optimal, the algorithm terminates, while if it is not the GPU finds the index of the entering variable in the fourth step and then transfers to the CPU another flag variable. The CPU checks the flag variable to determine if the linear problem is unbounded in order to terminate the algorithm. Otherwise, the GPU finds the index of the leaving variable in the fifth step and updates the basis and all the necessary variables in the sixth step. Then, the algorithm continues with the next iteration until a solution is found.

The aforementioned steps are executed in a serial manner either on the CPU or on the GPU. Data-parallel steps are executed on the GPU; these steps include: (i) scaling, (ii) pivoting, and (iii) basis update. In this point, we should describe the methods that we used for these three steps of the algorithm. Scaling is the most widely used preconditioning technique in linear optimization solvers. 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 the application of a linear programming algorithm for four reasons [45]: (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

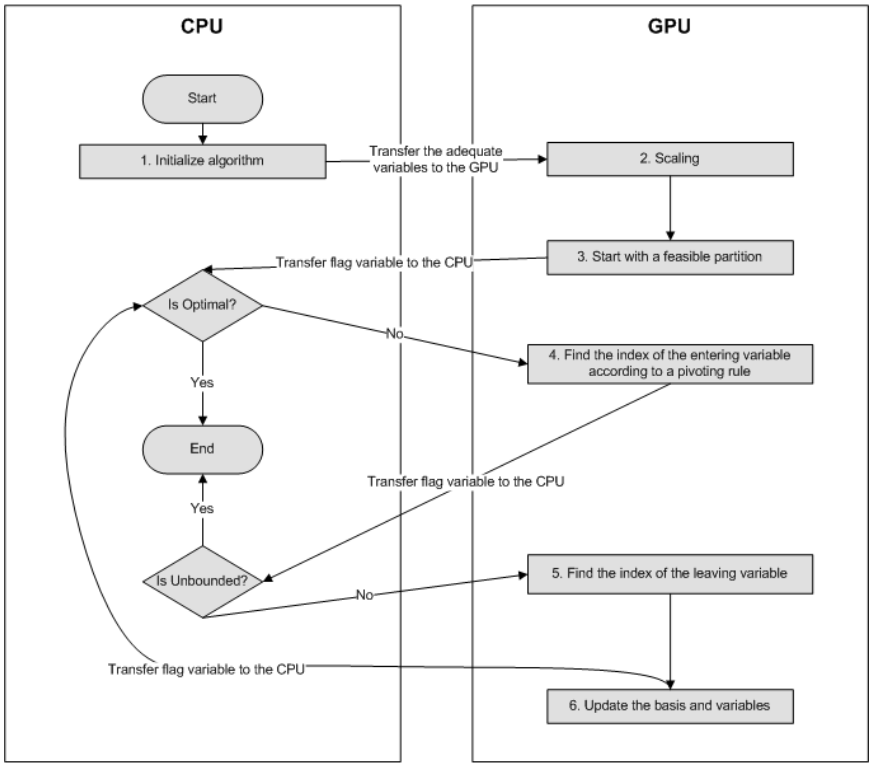

Figure 2: Flow Chart of the GPU-based Revised Simplex Algorithm

reduce the condition number of the constraint matrix A and improve the numerical behavior of the algorithms. Scaling has been proven to reduce both the number of iterations and the total execution time of the revised simplex algorithm [34]. In a previous paper [35], we have reviewed and implemented ten scaling techniques with a focus on the parallel implementation of them on GPUs under the MATLAB and CUDA environment. The computational study showed that the GPU-based arithmetic mean is the fastest scaling method. However, the equilibration scaling technique leads the revised simplex algorithm to less iterations than the other methods [34]. So, in this paper the scaling is performed using the arithmetic mean scaling method followed by the equilibration scaling method.
Some necessary notation should be introduced before the presentation of the pseudocode of each scaling method (more details can be found in [35]). 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.
Pseudocodes include "do parallel" and "end parallel" sections, in which the workload is divided into warps that are executed sequentially on a multiprocessor. Although, the scaling process can be applied iteratively, pseudocodes present only one iteration. Table 4 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).

Table 5 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).

A crucial step in solving a linear problem with the simplex algorithm is the selection of the entering variable. This step is performed in each iteration. Good

Table 4: 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
```

Table 5: 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
```

Table 6: GPU-based Steepest Edge

```
1.  do parallel
2.      Y = BasisInv * A(:,NonBasicList)
3.      dj = sqrt(1 + diag(Y' * Y))
4.      rj = Sn' ./ dj
5.      find the index of the minimum element of the vector rj
6.  end parallel
```

choices can lead to a fast convergence to the optimal solution, while poor choices lead to worse execution times or even no solutions of the LPs. A pivoting rule is one of the main factors that will determine the number of iterations that simplex algorithm performs [23]. In a previous paper [37], we have proposed six well-known pivoting rules for the revised simplex algorithm on a CPU-GPU computing environment. The computational study showed that the GPU-based steepest-edge is the fastest pivoting rule. So, in this paper the pivoting step is performed using the steepest-edge pivoting rule [12].
Some necessary notations should be introduced, before the presentation of the aforementioned pivoting rules. Let l be the index of the entering variable and $\overline{c_l}$ be the difference in the objective value when the non-basic variable x_l is increased by one unit and the basic variables are adjusted appropriately. Reduced cost is the amount by which the objective function on the corresponding variable must be improved before the value of the variable will be positive in the optimal solution. Steepest Edge Rule selects as entering variable the variable with the most objective value reduction per unit distance, as shown in Equation (1):

$$d_j = min\left\{\frac{c_l}{\sqrt{1+\sum_{i=1}^{m} x_{il}^2}} : l = 1, 2, ..., n\right\} \quad (1)$$

Table 6 shows the pseudocode of the implementation of the Steepest Edge on a GPU (more details can be found in [37]). The index of the incoming variable is calculated according to the Equation (1) (lines 2 - 5). $NonBasicList$ is an mx$(n-m)$ vector with the indices of the non basic variables and $BasisInv$ is an mxm matrix with the basis inverse. Sn is an 1xn vector with dual slack variables and ./ denotes the element–wise division of two vectors.

The total work of an iteration of simplex type algorithms is dominated by the determination of the basis inverse [32] [33]. This inverse, however, does not have to be computed from scratch during each iteration. Simplex type algorithms maintain a factorization of basis and update this factorization in each iteration. There are several schemes for updating basis inverse. In a previous paper [36], we proposed a GPU-based implementation for the Product Form of the Inverse (PFI) [7] and a Modification of the Product Form of the Inverse (MPFI)

[2] updating schemes. The computational study showed that the GPU-based MPFI outperformed the GPU-based PFI. So, in this paper the basis update is performed using the MPFI updating scheme.
Let $(A_B)^{-1}$ be the previous basis inverse, (h_l) be the pivot column, (k) be the index of the leaving variable and (m) the number of the constraints. Furthermore, let us assume that we have t GPU cores. Table 7 shows the steps that we used to compute the new basis inverse $(A_{\overline{B}})^{-1}$ with the MPFI scheme on the GPU (more details can be found in [36]).

Table 7: GPU-based MPFI

Step 0.
Compute the column vector:

$$v = \begin{bmatrix} -\frac{h_{1l}}{h_{rl}} & \cdots & \frac{1}{h_{rl}} & \cdots & -\frac{h_{ml}}{h_{rl}} \end{bmatrix}^T$$

Each core computes in parallel m/t elements of v. The pivot element is shared between all t cores.
Step 1.
Compute the outer product $v \otimes (A_{B_r})^{-1}$ with matrix multiplication. Each core will compute a block of the new matrix.
Step 2.
Set the r^{th} row of $(A_{\overline{B}})^{-1}$ equal to zero. Each core computes in parallel t/p rows of $(A_{\overline{B}})^{-1}$.
Step 3.
Add matrix $(A_{\overline{B}})^{-1}$ with the resulted matrix from step 1. Each core will compute a block of the new basis inverse.

Finally, we have tried to optimize the use of GPU memory. The frequent and heavy back-and-forth transmission of data between the CPU and GPU will dominate the computational time, so we reduced the communication time as far as possible. CPU is used to control the whole iteration while GPU is used for computing intensive steps. In both GPU-based algorithms presented in this paper, the communication between the CPU and GPU occurs only in the following steps of the algorithm: (i) initially, the CPU transfers to the GPU the matrix A and the vectors b and c, (ii) in each iteration the GPU transfers a flag variable to the CPU in order to determine if the linear problem is optimal and terminate the algorithm, (iii) in each iteration the GPU transfers another flag variable to the CPU in order to determine if the linear problem is unbounded and terminate the algorithm, and (iv) finally, the GPU transfers the objective value and the iterations needed to find the solution to the CPU.
Table 8 shows the pseudocode of the implementation of the revised simplex algorithm on a GPU.

4.3. Implementation of the GPU-Based Primal-Dual Exterior Point Simplex Algorithm

Figure 3 presents the process that is performed in the GPU-based implementation of the PDEPSA. In the first step, the CPU initializes the algorithm

Table 8: GPU-based Revised Simplex Algorithm

```
1.  The CPU reads the linear problem and initialize all the necessary variables.
2.  The CPU transfers to the GPU the matrix A and the vectors b and c
3.  The GPU scales the linear problem using arithmetic mean followed by
    equilibration.
4.  The GPU calculates a feasible partition.
5.  The GPU transfers a flag variable to the CPU.
6.  while linear problem is not optimal
7.       The CPU checks the flag variable to determine if the linear problem
         is optimal. If the linear problem is optimal, the algorithm
         terminates.
8.       The GPU calculates the index of the entering variable using the
         steepest-edge pivoting rule.
9.       The GPU transfers a flag variable to the CPU.
10.      The CPU checks the flag variable to determine if the linear problem
         is unbounded. If the linear problem is unbounded, the algorithm
         terminates.
11.      The GPU calculates the index of the leaving variable.
12.      The GPU updates the basis using the MPFI updating scheme.
13.      The GPU transfers a flag variable to the CPU.
14. end
```

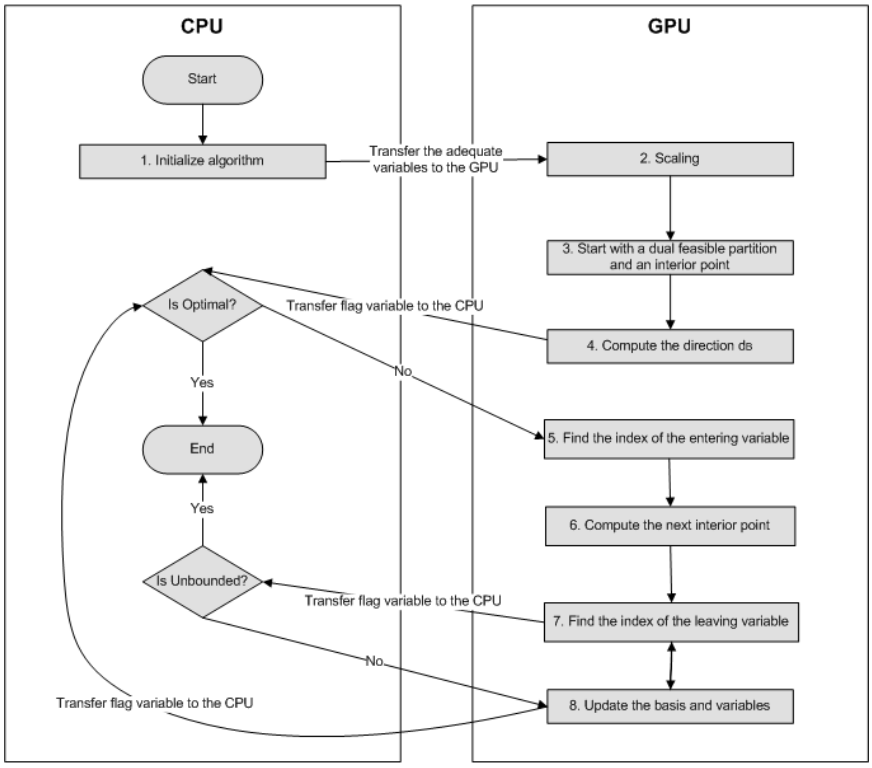

Figure 3: Flow Chart of the GPU-based Primal-Dual Exterior Point Simplex Algorithm

by reading all the necessary data. In the second step, the CPU transfers the adequate variables (A, b, and c) to the GPU and the GPU scales the linear problem. In the third step, the GPU computes a dual feasible solution and an interior point. In the fourth step, the GPU computes the direction d_B and transfers to the CPU a flag variable. The CPU checks the flag variable to determine if the linear problem is optimal. If the linear problem is optimal, the algorithm terminates, while if it is not the GPU finds the index of the entering variable in the fifth step. In the sixth step, the GPU computed the next interior point. In the seventh step, the GPU finds the index of the leaving variable and transfers to the CPU another flag variable. The CPU checks the flag variable to determine if the linear problem is unbounded in order to terminate the algorithm. Otherwise, in the eighth step the GPU updates the basis and all the necessary variables. Then, the algorithm continues with the next iteration until a solution is found.

As described in section 4.2, the scaling is performed using the arithmetic mean scaling method followed by the equilibration scaling method and the basis update is performed using the MPFI updating scheme. Again, we reduced the

Table 9: GPU-based Primal-Dual Exterior Point Simplex Algorithm

```
1.  The CPU reads the linear problem and initialize all the necessary variables.
2.  The CPU transfers to the GPU the matrix A and the vectors b and c.
3.  The GPU scales the linear problem using arithmetic mean followed by
    equilibration.
4.  The GPU calculates a dual feasible partition and an interior point.
5.  The GPU computes the direction dB.
6.  The GPU transfers a flag variable to the CPU.
7.  while linear problem is not optimal
8.      The CPU checks the flag variable to determine if the linear problem
        is optimal. If the linear problem is optimal, the algorithm terminates.
9.      The GPU calculates the index of the entering variable.
10.     The GPU computes the next interior point.
11.     The GPU finds the index of the leaving variable.
12.     The GPU transfers a flag variable to the CPU.
13.     The CPU checks the flag variable to determine if the linear problem
        is unbounded. If the linear problem is unbounded, the algorithm
        terminates.
14.     The GPU updates the basis using the MPFI updating scheme.
15.     The GPU transfers a flag variable to the CPU.
16. end
```

communication time between the CPU and GPU in the same manner explained in section 4.2. Table 9 shows the pseudocode of the implementation of the PDEPSA on a GPU.

5. 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 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 in 448 total cores. The graphics card driver installed in our system is NVIDIA 64 kernel module 320.92. Both GPU-based algorithms have been implemented in MATLAB 2013b using MATLAB's Parallel Computing Toolbox.
In this computational study, we compare the proposed GPU-based algorithms with MATLAB's large-scale linprog built-in function. The large-scale algorithm

is based on Interior Point Solver [48], a primal-dual interior point algorithm. MATLAB's linprog function automatically executes on multiple computational threads, in order to take advantage of the multiple cores of the CPU. The execution time of this algorithm 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, MATLABs inherent multithreading is not as efficient as a pure parallel implementation. Execution times for all algorithms have been measured in seconds using tic and toc MATLAB's built-in functions. Finally, the results of the GPU-based implementations are very accurate, because NVIDIA Quadro 6000 is fully IEEE 764-2008 compliant 32- and 64-bit fast double-precision.
The test set used in the computational study was: (i) and a set of randomly generated sparse and dense optimal LPs (problem instances have the same number of constraints and variables and the largest problem tested has $6,000$ constraints and $6,000$ variables), and (ii) a set of benchmark problems (netlib, kennington, Mészáros) that do not have bounds and ranges sections in their mps files. Sparse LPs were generated with 10% and 20% density. For each instance we averaged times over 10 runs. All runs were executed as a batch job. The randomly generated LPs that have been solved are of the general form (LP.3):

$$\begin{array}{ll} min & c^T x \\ s.t. & Ax \oplus b \\ & x \geq 0 \end{array} \qquad (LP.3)$$

where $\oplus = \begin{cases} \leq \\ \geq \end{cases}$. The ranges of values that were used for the randomly generated LPs are $c \in [1...500]$, $A \in [10...400]$ and $b \in [10...100]$. MATLAB's random generators rand and sprand were used to generate uniformly distributed random numbers using the current timestamp as the seed.
Table 11 presents some useful information about the second 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 optimal objective value. The test bed includes 14 LPs from Netlib, 1 Kennington and 7 LPs from Mészáros collection.

MATLABs GPU library does not support sparse matrices, so, all matrices are stored as dense (including zero elements). Hence, we altered MATLAB's IPM algorithm in order to use dense matrices and make the comparison with our algorithms fair.
In Tables 11 - 16 and Figures 4 - 6, the following abbreviations are used: (i) Primal-Dual Exterior Point Simplex Algorithm running on CPU - PDEPSA, (ii) MATLAB's large-scale linprog built-in function - IPM, (iii) Revised Sim-

Table 10: Statistics of the Netlib, Kennington and Mészáros LPs

Name	Constraints	Variables	Nonzeros A	Optimal Objective value
AGG	489	163	2,541	-3.60E+07
BEACONFD	174	262	3,476	3.36E+04
BNL2	2,325	3,489	16,124	1.81E+03
CARI	400	1,200	152,800	5.81E+02
OSA-07	1,119	23,949	167,643	5.35E+05
ROSEN1	520	1,544	23,794	-2.76E+04
ROSEN2	1,032	3,080	47,536	5.44e+04
ROSEN7	264	776	8,034	2.03e+04
ROSEN8	520	1,544	16,058	4.21e+04
ROSEN10	2,056	6,152	64,192	1.74e+05
SCORPION	389	358	1,708	1.88E+03
SCTAP2	1,091	1,880	8,124	1.72E+03
SCTAP3	1,481	2,480	10,734	1.42E+03
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
SLPTSK	2,861	3,347	72,465	2.34E+02
STOCFOR2	2,158	2,031	9,492	-3.90E+04
WOOD1P	245	2,594	70,216	1.44E+00

Table 11: Total execution time over the randomly generated optimal dense LPs

Problem	PDESPA	IPM	RSA-GPU	PDESPA-GPU
1,000x1,000	20.68	279.68	7.41 (1.66%)	3.02 (1.23%)
1,500x1,500	41.82	499.05	12.65 (1.41%)	5.06 (1.12%)
2,000x2,000	149.81	1,128.46	23.69 (0.93%)	8.94 (1.23%)
2,500x2,500	457.80	2,794.41	51.20 (0.72%)	17.32 (1.04%)
3,000x3,000	652.34	4,399.06	80.35 (0.63%)	26.46 (0.97%)
3,500x3,500	1,129.34	7,040.20	125.04 (0.54%)	40.26 (0.93%)
4,000x4,000	2,030.42	8,938.19	154.06 (0.46%)	49.34 (0.85%)
4,500x4,500	2,819.71	-	252.41 (0.42%)	70.60 (0.81%)
5,000x5,000	3,456.28	-	428.00 (0.41%)	95.41 (0.79%)
5,500x5,500	4,361.71	-	778.84 (0.34%)	146.23 (0.75%)
6,000x6,000	8,838.89	-	1,261.53 (0.30%)	205.42 (0.73%)
Average	2,178.07	3,582.72	288.65 (0.71%)	60.73 (0.90%)

plex Algorithm running on GPU - RSA-GPU, and (iv) Primal-Dual Exterior Point Simplex Algorithm running on GPU - PDEPSA-GPU. We also included execution times for PDEPSA running on CPU, because PDEPSA-GPU is faster than RSA and IPM; so, PDEPSA is utilized in order to make clear that the acceleration of PDEPSA-GPU is coming from the parallel implementation and not from the based algorithm. Tables 11 - 13 present the total execution time of the algorithms over the randomly generated optimal dense LPs, the randomly generated optimal sparse LPs with density 10% and the randomly generated optimal sparse LPs with density 20%, respectively. The total execution time of the GPU-based implementations of RSA-GPU and PDEPSA-GPU also include the communication time. The percentage of the communication time to the total execution time for the GPU-based algorithms is presented in the parentheses. Tables 14 - 16 present the iterations needed by each algorithm to solve the linear problem over the randomly generated optimal dense LPs, the randomly generated optimal sparse LPs with density 10% and the randomly generated optimal sparse LPs with density 20%, respectively. A time limit of 3 hours was imposed which explains why there are no measurements for IPM on randomly generated optimal dense LPs above $n = 4000$ and for PDEPSA on randomly generated optimal sparse LPs with density 10% above $n = 2500$ and on randomly generated optimal sparse LPs with density 20% above $n = 2000$.

Figures 4 - 6 present the speedup over the randomly generated optimal dense LPs, the randomly generated optimal sparse LPs with density 10% and the randomly generated optimal sparse LPs with density 20%, respectively. The following abbreviations are used: (i) PDESPA-GPU/PDEPSA is the speedup of PDEPSA-GPU over PDEPSA, (i) PDESPA-GPU/IPM is the speedup of PDEPSA-GPU over IPM, (ii) RSA-GPU/IPM is the speedup of RSA-GPU over IPM, and (iii) PDESPA-GPU/RSA-GPU is the speedup of PDEPSA-GPU over RSA-GPU.

Table 12: Total execution time over the randomly generated optimal sparse LPs with density 10%

Problem	PDESPA	IPM	RSA-GPU	PDESPA-GPU
1,000x1,000	283.03	78.86	61.02 (0.12%)	9.84 (0.10%)
1,500x1,500	1,502.85	280.80	196.35 (0.08%)	28.58 (0.08%)
2,000x2,000	4,209.14	415.70	270.31 (0.06%)	37.46 (0.08%)
2,500x2,500	10,770.82	1,298.86	688.14 (0.06%)	88.66 (0.07%)
3,000x3,000	-	1,683.41	873.17 (0.05%)	104.01 (0.06%)
3,500x3,500	-	2,539.07	1,236.40 (0.05%)	148.83 (0.05%)
4,000x4,000	-	3,128.60	1,511.56 (0.03%)	182.00 (0.04%)
4,500x4,500	-	3,710.59	1,731.46 (0.03%)	213.49 (0.04%)
5,000x5,000	-	4,466.21	2,049.69 (0.02%)	255.90 (0.03%)
5,500x5,500	-	5,736.32	2,435.80 (0.02%)	315.66 (0.03%)
6,000x6,000	-	7,234.65	3,035.53 (0.02%)	385.42 (0.02%)
Average	4,191.46	2,779.37	1,280.86 (0.05%)	160.90 (0.05%)

Table 13: Total execution time over the randomly generated optimal sparse LPs with density 20%

Problem	PDESPA	IPM	RSA-GPU	PDESPA-GPU
1,000x1,000	247.57	88.50	34.26 (0.16%)	8.18 (0.12%)
1,500x1,500	1,260.59	162.82	59.39 (0.10%)	13.21 (0.09%)
2,000x2,000	6,015.30	355.79	127.77 (0.08%)	25.84 (0.09%)
2,500x2,500	-	758.21	232.88 (0.06%)	51.24 (0.07%)
3,000x3,000	-	1,077.84	314.60 (0.05%)	71.28 (0.06%)
3,500x3,500	-	2,487.42	649.23 (0.05%)	139.57 (0.05%)
4,000x4,000	-	3,072.17	776.63 (0.05%)	171.23 (0.05%)
4,500x4,500	-	3,853.41	953.61 (0.04%)	206.81 (0.04%)
5,000x5,000	-	5,024.56	1,237.60 (0.04%)	267.17 (0.04%)
5,500x5,500	-	6,539.02	1,535.80 (0.04%)	339.31 (0.04%)
6,000x6,000	-	8,104.58	1,835.53 (0.03%)	415.32 (0.03%)
Average	2,507.82	2,865.85	705.21 (0.06%)	155.38 (0.06%)

Table 14: Number of iterations over the randomly generated optimal dense LPs

Problem	PDESPA	IPM	RSA-GPU	PDESPA-GPU
1,000x1,000	134	34	118	134
1,500x1,500	103	32	68	103
2,000x2,000	163	29	148	163
2,500x2,500	257	42	249	257
3,000x3,000	170	42	163	170
3,500x3,500	258	45	180	258
4,000x4,000	242	38	220	242
4,500x4,500	323	-	216	323
5,000x5,000	149	-	131	149
5,500x5,500	299	-	280	299
6,000x6,000	415	-	330	415
Average	228.45	37.43	191.18	228.45

Table 15: Number of iterations over the randomly generated optimal sparse LPs with density 10%

Problem	PDESPA	IPM	RSA-GPU	PDESPA-GPU
1,000x1,000	1,256	22	1,275	1,256
1,500x1,500	2,069	26	1,836	2,069
2,000x2,000	2,581	28	2,450	2,581
2,500x2,500	3,503	29	2,845	3,503
3,000x3,000	-	24	3,285	4,688
3,500x3,500	-	25	4,675	5,332
4,000x4,000	-	26	5,035	6,801
4,500x4,500	-	27	5,508	7,517
5,000x5,000	-	28	6,516	8,669
5,500x5,500	-	30	7,945	10,314
6,000x6,000	-	31	9,287	12,456
Average	2,352.25	26.91	4,605.18	5,926.00

Table 16: Number of iterations over the randomly generated optimal sparse LPs with density 20%

Problem	PDESPA	IPM	RSA-GPU	PDESPA-GPU
1,000x1,000	1,041	23	1,024	1,041
1,500x1,500	1,832	19	1,340	1,832
2,000x2,000	2,460	24	1,811	2,460
2,500x2,500	-	23	2,650	3,282
3,000x3,000	-	24	2,836	4,190
3,500x3,500	-	30	3,454	4,987
4,000x4,000	-	26	3,889	5,627
4,500x4,500	-	28	4,536	6,438
5,000x5,000	-	30	4,888	7,764
5,500x5,500	-	29	5,653	9,135
6,000x6,000	-	32	6,781	11,341
Average	1,777.67	26.18	3,532.91	5,281.55

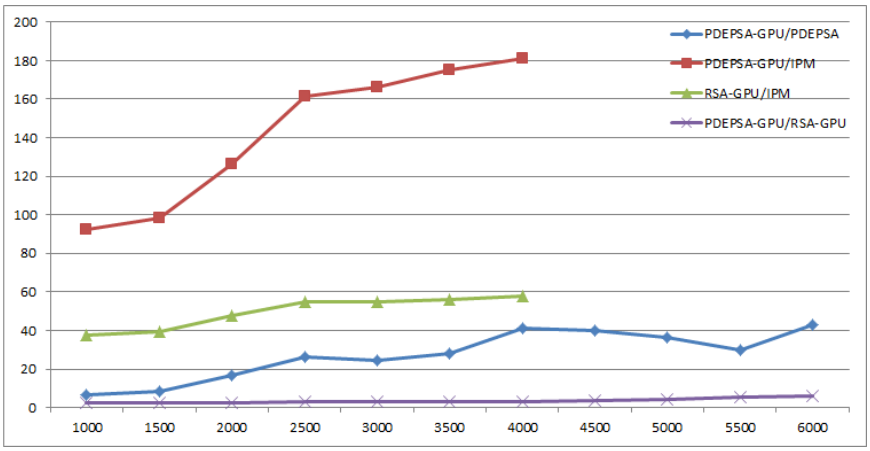

Figure 4: Speedup over the randomly generated optimal dense LPs

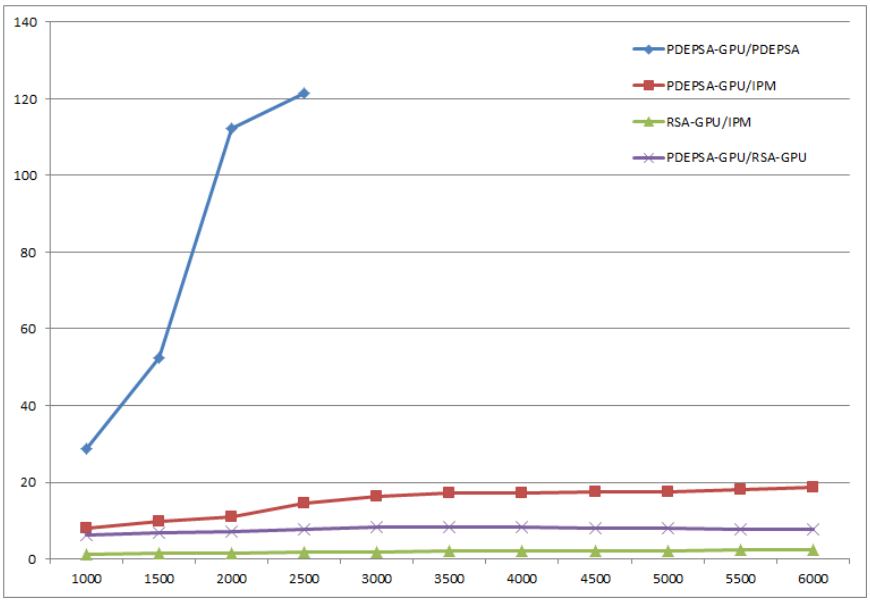

Figure 5: Speedup over the randomly generated optimal sparse LPs with density 10%

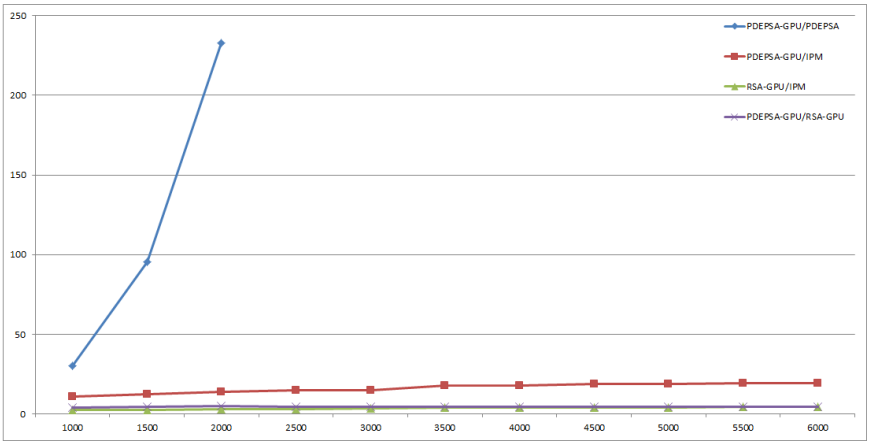

Figure 6: Speedup over the randomly generated optimal sparse LPs with density 20%

Table 17: Total execution time over the Netlib, Kennington and Mészáros LPs

Name	PDESPA	IPM	PDESPA-GPU
AGG	1.12	**0.59**	0.73
BEACONFD	0.15	**0.08**	0.11
BNL2	1,386.43	46.89	**17.10**
CARI	7.12	**2.26**	2.31
OSA-07	412.18	115.34	**12.62**
ROSEN1	15.81	**1.54**	2.22
ROSEN2	149.06	8.12	**6.76**
ROSEN7	1.43	**0.41**	1.02
ROSEN8	17.78	**1.45**	2.76
ROSEN10	1,907.00	45.31	**24.75**
SCORPION	0.56	**0.21**	0.43
SCTAP2	215.14	7.01	**2.70**
SCTAP3	215.14	17.76	**4.18**
SHIP04L	2.35	**0.56**	1.12
SHIP04S	0.86	**0.28**	0.59
SHIP08L	17.81	**2.09**	2.34
SHIP08S	2.56	**0.48**	1.11
SHIP12L	65.64	**5.01**	5.05
SHIP12S	8.50	**0.93**	2.07
SLPTSK	2,889.50	245.51	**15.54**
STOCFOR2	1,120.04	37.43	**9.44**
WOOD1P	8.96	**0.84**	1.12
Average	**383.87**	**24.55**	**5.28**

Table 17 presents the total execution time of the algorithms over the Netlib, Kennington and Mészáros LPs. We excluded RSA from this computational study, because PDEPSA-GPU is clearly superior to it. Table 18 presents the iterations needed by each algorithm to solve the linear problem over the Netlib, Kennington and Mészáros LPs. Execution times with bold emphasize indicate the best execution time for the specific linear problem. Figure 7 presents the speedup over the Netlib, Kennington and Mészáros LPs. The following abbreviations are used: (i) PDESPA-GPU/PDEPSA is the speedup of PDEPSA-GPU over PDEPSA, and (ii) PDEPSA-GPU/IPM is the speedup of PDEPSA-GPU over IPM.

Before the discussion of the results, we should note again that MATLAB's large-scale linprog built-in function is not completely parallelized, but utilizes the inherent multithreading in MATLAB. We have selected this algorithm because it is a sophisticated commercial LP solver and it is implemented in MATLAB as our algorithms. So, we believe that it is fair to use MATLAB's large-scale linprog built-in function as a reference point for the comparison with our GPU-based algorithms. We also present the execution time of our CPU-based implementation for PDEPSA in order to make clear that the acceleration is

Table 18: Number of iterations over the Netlib, Kennington and Mészáros LPs

Name	PDESPA	IPM	PDESPA-GPU
AGG	146	20	146
BEACONFD	21	10	21
BNL2	2,146	32	2,146
CARI	456	15	456
OSA-07	918	31	918
ROSEN1	525	14	525
ROSEN2	1,113	15	1,113
ROSEN7	239	12	239
ROSEN8	582	14	582
ROSEN10	2,450	15	2,450
SCORPION	92	14	92
SCTAP2	377	17	377
SCTAP3	631	20	631
SHIP04L	217	12	217
SHIP04S	157	12	157
SHIP08L	428	14	428
SHIP08S	231	13	231
SHIP12L	864	16	864
SHIP12S	576	15	576
SLPTSK	1,438	29	1,438
STOCFOR2	1,205	26	1,205
WOOD1P	186	25	186
Average	681.73	17.77	681.73

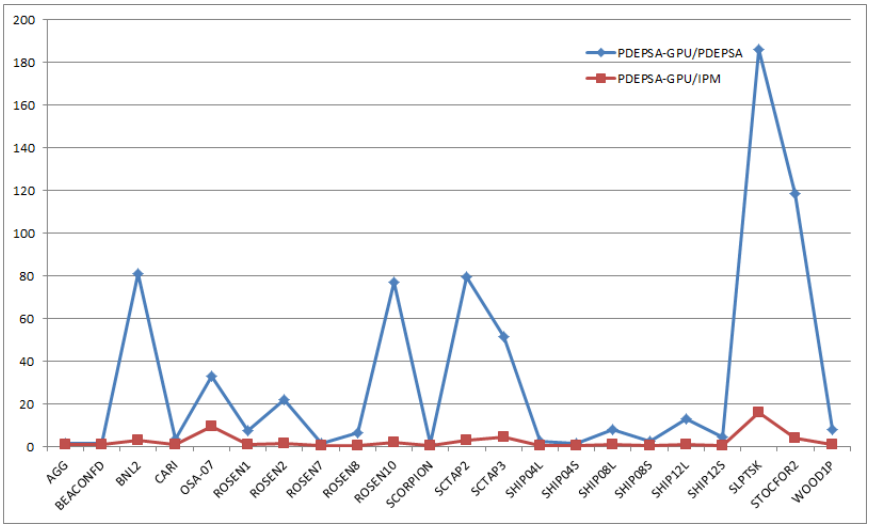

Figure 7: Speedup over the Netlib, Kennington and Mészáros LPs

coming from the parallel implementation and not from the based algorithm.
The results show a large speedup of both proposed GPU-based algorithms over MATLAB's large-scale linprog built-in function on randomly generated sparse and dense LPs. From the results over the randomly generated LPs, we observe: (i) GPU-based PDEPSA achieves a maximum speedup over IPM of about 181 (on average 143) on dense LPs, about 19 (on average 15) on sparse LPs with density 10%, and about 20 (on average 16) on sparse LPs with density 20%, (ii) GPU-based RSA achieves a maximum speedup over IPM of about 58 (on average 50) on dense LPs, about 2 (on average 2) on sparse LPs with density 10%, and about 4 (on average 4) on sparse LPs with density 20%, (iii) GPU-based PDEPSA is much faster than GPU-based RSA on all LPs, (iv) our CPU-based PDEPSA algorithm is much faster than IPM over the randomly generated dense LPs, but much slower over the randomly generated sparse LPs, (v) GPU-based PDEPSA outperforms CPU-based PDEPSA on all instances, and (iv) the percentage of the communication time to the total execution time on both GPU-based algorithms is very small. For the sake of completeness, we also present the number of iterations needed by each algorithm to solve the linear problem in Tables 13 - 15, but only the total execution time of the algorithms is used to compare the algorithms. This is because interior point methods converge after a few iterations, but with a large computational cost per iteration.
The computational study over the benchmark LPs shows that our GPU-based PDEPSA is on average 2.30 times faster than IPM and 32.25 times faster than the CPU-based PDEPSA. Furthermore, the CPU-based PDEPSA is slower than IPM on all instances. This observation shows that the speedup gained over the IPM algorithm stems from the parallelization and not the base algorithm. Moreover, the speedup gained over the benchmark LPs is much lower than the speedup over the randomly generated LPs, because MATLABs GPU library does not support sparse matrices and all matrices are stored as dense, as stated before.
These findings are significant because they show that primal-dual exterior point simplex algorithms are more efficient for GPU-based implementations than the revised simplex algorithm. To the best of our knowledge this is the only paper presenting a parallelization of an exterior point simplex algorithm on GPUs. Furthermore, our GPU-based implementations presented great speedup not only on randomly generated sparse and dense LPs, but also on benchmark LPs.

6. Conclusions

GPUs have been already applied for the solution of linear optimization algorithms, but GPU-based implementations of an exterior point simplex algorithm have not yet been studied. In this paper, we proposed two efficient GPU-based implementations of the revised simplex algorithm and a primal-dual exterior point simplex algorithm. We performed a computational study on large-scale randomly generated optimal sparse and dense LPs and found that both GPU-based algorithms outperform MATLAB's interior point method. The primal-

dual exterior point simplex algorithm was the fastest GPU-based implementation and the maximum speedup gained over MATLAB's interior point method was about 181 on dense LPs and about 20 on sparse LPs. Furthermore, the GPU-based primal-dual exterior point simplex algorithm shows a great speedup (on average 2.3) over MATLAB's interior point method on a set of benchmark LPs.
In future work, we plan to port our implementation using CUDA C/C++ in order to take advantage of a high performance computing language and compare it with other state-of-the-art solvers, like CPLEX and GUROBI. Finally, we plan to experiment with other primal-dual exterior point simplex algorithm variants.

Acknowledgements

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Quadro 6000 GPU used for this research.

References

[1] Badr, E.S., Moussa, M., Papparrizos, K., Samaras, N., Sifaleras, A.: Some computational results on MPI parallel implementations of dense simplex method. Trans Eng Comput Technol, 17, 228-231 (2006)

[2] Benhamadou, M.: On the simplex algorithm 'revised form'. Advances in Engineering Software, 33, 769-777 (2002)

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

[4] Bixby, R.E., Martin, A.: Parallelizing the dual simplex method. INFORMS Journal on Computing, 12(1), 45-56 (2000)

[5] Borgwardt, H.K.: The average number of pivot steps required by the simplex method is polynomial. Zeitschrift fur Operational Research, 26(1), 157-177 (1982)

[6] Cvetanovic, Z., Freedman, E., Nofsinger, C.: Efficient decomposition and performance of parallel PDE, FFT, Monte Carlo simulations, simplex, and sparse solvers. The Journal of Supercomputing, 5, 219-238 (1991)

[7] Dantzig, G.B., Orchard-Hays, W.: The product form of the inverse in the simplex method. Math. Comp., 8, 64-67 (1954)

[8] Dantzig, G.B.: Linear programming and extensions. Princeton, NJ: Princeton University Press (1963)

[9] Eckstein, J., Boduroglu, I.I., Polymenakos, L.C., Goldfarb, D.: Data-parallel implementations of dense simplex methods on the Connection Matching CM-2. ORSA Journal on Computing, 7(4), 402-416 (1995)

[10] Gade-Nielsen, N.F., Jorgensen, J.B., Dammann, B.: MPC Toolbox with GPU Accelerated Optimization Algorithms. In: Proceedings of the 10th European Workshop on Advanced Control and Diagnosis (ACD 2012), Copenhagen, Denmark (2012)

[11] Gay, D.M.: Electronic mail distribution of linear programming test problems. Mathematical Programming Society COAL Newsletter, 13, 10-12 (1985)

[12] Goldfarb, D., Reid, J.K.: A Practicable Steepest-Edge Simplex Algorithm. Mathematical Programming, 12(3), 361-371 (1977)

[13] Hall, J.A.J., McKinnon, K.I.M.: PARSMI, a parallel revised simplex algorithm incorporating minor iterations and Devex pricing. In: Wasniewski, J., Dongarra, J., Madsen, K., Olesen, D., Eds., Applied Parallel Computing, Lecture Notes in Computer Science, Springer, 1184, 67-76 (1996)

[14] Hall, J.A.J., McKinnon, K.I.M.: ASYNPLEX, an asynchronous parallel revised simplex algorithm. Annals of Operations Research, 81, 27-49 (1998)

[14] Hall, J.A.J.: Towards a practical parallelisation of the simplex method. Computational Management Science, 7(2), 139-170 (2010)

[15] Ho, J., Sundarraj, R.: On the efficacy of distributed simplex algorithms for linear programming. Computational Optimization and Applications, 3, 349-363 (1994)

[16] Homm, F., Kaempchen, N., Ota, J., Burschka, D.: Efficient occupancy grid computation on the GPU with lidar and radar for road boundary detection. In: Proceedings of the Intelligent Vehicles Symposium (IV), 1006-1013, Michigan, USA (2010)

[17] Jung, J.H., O'Leary, D.P.: Implementing an interior point method for linear programs on a CPU-GPU system. Electronic Transaction on Numerical Analysis, 28, 174189 (2008)

[18] Klee, V., Minty, G.J.: How good is the simplex algorithm?. Inequalities III, New York, Academic Press, 159-175 (1972)

[19] 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), 1999-2006, Washington, USA (2011)

[20] Lalami, M.E., El-Baz, D., Boyer, V. Multi gpu implementation of the simplex algorithm. In: Proceedings of the 2011 IEEE 13th International Conference on High Performance Computing and Communications (HPCC), 179-186, Banff, Canada (2011)

[21] Lentini, M., Reinoza, A., Teruel, A., Guillen, A.: SIMPAR: a parallel sparse simplex. Computational and Applied Mathematics, 14(1), 49-58 (1995)

[22] Li, J., Lv, R., Hu, X., Jiang, Z.: A GPU-Based Parallel Algorithm for Large Scale Linear Programming Problem. In: Watada, J., Phillips-Wren, G., Jai, L., Howlett, R.J., Eds., Intelligent Decision Technologies, SIST 10, Springer Berlin Heidelberg, Springer-Verlag, Berlin, 37-46 (2011)

[23] Maros, I., Khaliq, M.H.: Advances in design and implementation of optimization software. European Journal of Operational Research, 140(2), 322-337 (1999)

[24] MATLAB'S Parallel Computing Toolbox. `http://www.mathworks.com/products/parallel-computing/` (Last accessed on 10/11/2013)

[25] Meyer, X., Albuquerque, P., Chopard, B.: A multi-GPU implementation and performance model for the standard simplex method. In: Proceedings of the 1st International Symposium and 10th Balkan Conference on Operational Research, 312-319, Thessaloniki, Greece (2011)

[26] NVIDIA CUDA. `https://developer.nvidia.com/category/zone/cuda-zone` (Last accessed on 10/11/2013)

[27] NVIDIA CUDA-CUBLAS Library 2.0, NVIDIA Corp., `http://developer.nvidia.com/object/cuda.html` (Last accessed on 10/11/2013)

[28] LAPACK Library, `http://www.cudatools.com` (Last accessed on 10/11/2013)

[29] Paparrizos, K.: An infeasible exterior point simplex algorithm for assignment problems. Mathematical Programming, 51(1-3), 45-54 (1991)

[30] Paparrizos, K., Samaras, N., Stephanides, G.: An efficient simplex type algorithm for sparse and dense linear programs. European Journal of Operational Research, 148(2), 323-334 (2003)

[31] Paparrizos, K., Samaras, N., Stephanides, G.: A new efficient primal dual simplex algorithm. Computers & Operations Research, 30(9), 1383-1399 (2003)

[32] Ploskas, N., Samaras, N.: Basis Update on Simplex Type Algorithms. In: Proceedings of the EWG-DSS Thessaloniki 2013, Thessaloniki, Greece, 11 (2013)

[33] Ploskas, N., Samaras, N., Margaritis, K.: A parallel implementation of the revised simplex algorithm using OpenMP: some preliminary results. Optimization Theory, Decision Making, and Operational Research Applications, Springer Proceedings in Mathematics & Statistics, 31, 163-175 (2013)

[34] Ploskas, N., Samaras, N.: The impact of scaling on simplex type algorithms. In: Proceedings of the 6th Balkan Conference in Informatics, Thessaloniki, Greece, 17-22 (2013)

[35] Ploskas, N., Samaras, N.: A Computational Comparison of Scaling Techniques for Linear Optimization Problems on a GPU. International Journal of Computer Mathematics, accepted for publication.

[36] Ploskas, N., Samaras, N.: A Computational Comparison of Basis Updating Schemes for the Simplex Algorithm on a CPU-GPU System. American Journal of Operations Research, 3, 497-505 (2013)

[37] Ploskas, N., Samaras, N.: GPU Accelerated Pivoting Rules for the Simplex Algorithm. Journal of Systems and Software, submitted for publication.

[38] Samaras, N.: Computational improvements and efficient implementation of two path pivoting algorithms. Ph.D. dissertation, Department of Applied Informatics, University of Macedonia (2001)

[39] Shu, W.: Parallel implementation of a sparse simplex algorithm on MIMD distributed memory computers. Journal of Parallel and Distributed Computing, 31(1), 25-40, 1995.

[40] Smith, E., Gondzio, J., Hall, J.: GPU acceleration of the matrix-free interior point method. In: Parallel Processing and Applied Mathematics, 681-689, Springer Berlin Heidelberg (2012)

[41] Spampinato, D.G., Elster, A.C.: Linear optimization on modern GPUs. In: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009), Rome, Italy (2009)

[42] Stunkel, C.B., Reed, D.A.: Hypercube implementation of the simplex algorithm. In: Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications, 2, 1473-1482, New York, USA, (1988)

[43] The Khronos OpenCL Working Group: OpenCL - The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/` (Last accessed on 5/8/2013)

[44] Thomadakis, M.E., Liu, J.C.: An efficient steepest-edge simplex algorithm for SIMD computers. In: Proceedings od the 10th International Conference on Supercomputing (ICS 1996), 286-293, Philadelphia, Pennsylvania, USA (1996)

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

[46] Yarmish, G., Slyke, R.: A distributed, scaleable simplex method. The Journal of Supercomputing, 49, 373-381 (2009)

[47] Zhang, E. Z., Jiang, Y., Guo, Z., Shen, X.: Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In: Proceedings of the 24th ACM International Conference on Supercomputing, 115-126, Tsukuba, Japan (2010)

[48] Zhang, Y.: Solving Large-Scale Linear Programs by Interior-Point Methods under the MATLAB Environment. Optimization Methods and Software, 10(1), 1-31 (1998)