Efficient Implementation of the Simplex Method on a CPU-GPU System

Mohamed Esseghir Lalami, Vincent Boyer, Didier El-Baz
CNRS ; LAAS ; 7 avenue du colonel Roche, F-31077 Toulouse, France
Université de Toulouse ; UPS, INSA, INP, ISAE ; LAAS ; F-31077 Toulouse France
Email: mlalami@laas.fr vboyer@laas.fr elbaz@laas.fr

***Abstract*—The Simplex algorithm is a well known method to solve linear programming (LP) problems. In this paper, we propose a parallel implementation of the Simplex on a CPU-GPU systems via CUDA. Double precision implementation is used in order to improve the quality of solutions. Computational tests have been carried out on randomly generated instances for non-sparse LP problems. The tests show a maximum speedup of 12.5 on a GTX 260 board.**

***Keywords*-hybrid computing; GPU computing; parallel computing; CUDA; Simplex method; linear programming.**

I. Introduction

Initially developed for real time and high-definition 3D graphic applications, Graphics Processing Units (GPUs) have gained recently attention for High Performance Computing applications. Indeed, the peak computational capabilities of modern GPUs exceeds the one of top-of-the-line central processing units (CPUs). GPUs are highly parallel, multithreaded, manycore units.

In November 2006, NVIDIA introduced, Compute Unified Device Architecture (CUDA), a technology that enables users to solve many complex problems on their GPU cards (see for example [1] - [4]).

Some related works have been presented on the parallel implementation of algorithms on GPU for linear programming (LP) problems. O'Leary and Jung have proposed in [5] a combined CPU-GPU implementation of the Interior Point Method for LP; computational results carried out on NETLIB LP problems [6] for at most 516 variables and 758 constraints, show that some speedup can be obtained by using GPU for sufficiently large dense problems.

Spampinato and Elster have proposed in [7] a parallel implementation of the revised Simplex method for LP on GPU with NVIDIA CUBLAS [8] and NVIDIA LAPACK [9] libraries. Tests were carried out on randomly generated LP problems of at most 2000 variables and 2000 constraints. The implementation showed a maximum speedup of 2.5 on a NVIDIA GTX 280 GPU as compared with sequential implementation on CPU with Intel Core2 Quad 2.83 GHz. Bieling, Peschlow and Martini have proposed in [10] an other implementation of the revised Simplex method on GPU. This implementation permits one to speed up solution with a maximum factor of 18 in *single* precision on a NVIDIA GeForce 9600 GT GPU card as compared with GLPK solver run on Intel Core 2 Duo 3GHz CPU. To the best of our knowledge, these are the available references on parallel implementations on GPUs of algorithms for LP.

The revised Simplex method is generally more efficient than the standard Simplex method for large linear programming problems (see [11] and [12]), but for dense LP problems, the two approaches are equivalent (see [13] and [14]).

In this paper, we concentrate on the parallel implementation of the standard Simplex algorithm on CPU-GPU systems for dense LP problems. Dense linear programming problems occur in many important domains. In particular, some decompositions like Benders, Dantzig-Wolfe give rise to full dense LP problems. Reference is made to [15] and [16] for applications leading to dense LP problems.

The standard Simplex method is an iterative method that manipulates independently at each iteration the elements of a fixed size matrix. The main challenge was to implement this algorithm in double *precision* with CUDA C environment without using existing NVIDIA libraries like CUBLAS and LAPACK in order to obtain as best speedup as we can. By identifying the tasks that can be parallelized and good management of GPUs memories one can obtain good speedup with regards to sequential implementation.

We have been solving linear programming problems in the context of the solution of NP-complete combinatorial optimization problems (see [17]). For example, one has to solve frequently linear programming problems for bound computation purpose when one uses branch and bound algorithms and it may happen that some instances give rise to dense LP problems. The present work is part of a study on the parallelization of optimization methods (see also [1]).

The paper is structured as follows. Section II deals with the Simplex method. The parallel implementation of the Simplex algorithm on CPU-GPU systems is presented in Section III. The Section IV is devoted to presentation and analysis of computational results for randomly generated instances. Finally, in Section V, we give some conclusions and perspectives.

II. Mathematical background on Simplex method

Linear programming (LP) problems consist in maximizing (or minimizing) a linear objective function subject to a set of linear constraints. More formally, we consider the following problem :

$$\begin{aligned} \max x_0 = cx^{'}, \\ s.t: A^{'}x^{'} \leq b^{'}, \\ x^{'} \geq 0, \end{aligned} \tag{1}$$

with

$$c^{'} = (c_1, c_2, ..., c_n) \in \mathbf{R}^n,$$

$$A^{'} = \begin{pmatrix} a_{11} & a_{12} \cdots a_{1n} \\ a_{21} & a_{22} \cdots a_{2n} \\ \vdots & \vdots \ddots \vdots \\ a_{m1} & a_{m2} \cdots a_{mn} \end{pmatrix} \in \mathbf{R}^{m \times n},$$

and

$$x^{'} = (x_1, x_2, ..., x_n)^T,$$

n and m are the number of variables and constraints, respectively.
Inequality constraints can be written as *equality* constraints by introducing m new variables x_{n+l} named *slack* variables, so that:

$$a_{l1}x_1 + a_{l2}x_2 + ... + a_{ln}x_n + x_{n+l} = b_l, \ l \in \{1, 2, \ldots, m\},$$

with $x_{n+l} \geq 0$ and $c_{n+l} = 0$. Then, the standard form of linear programming problem can be written as follows:

$$\begin{aligned} \max x_0 = cx, \\ s.t: Ax = b, \\ x \geq 0, \end{aligned} \tag{2}$$

with

$$c = (c^{'}, 0, ..., 0) \in \mathbf{R}^{(n+m)},$$

$$A = \begin{pmatrix} A^{'}, I_m \end{pmatrix} \in \mathbf{R}^{m \times (n+m)},$$

I_m is the $m \times m$ identity matrix and $x = (x^{'}, x_{n+1}, x_{n+2}, \ldots, x_{n+m})^T$.
In 1947, George Dantzig proposed the *Simplex algorithm* for solving linear programming problems (see [11]). The Simplex algorithm is a pivoting method that proceeds from a first feasible extreme point solution of a LP problem to another feasible solution, by using matrix manipulations, the so-called *pivoting* operations, in such a way as to continually increase the objective value. Different versions of this method have been proposed. In this paper, we consider the method proposed by Garfinkel and Nemhauser in [19] which improves the algorithm of Dantzig by reducing the number of operations and the memory occupancy.
We suppose that the columns of A are permuted so that $A = (B, N)$, where B is an $m \times m$ *nonsingular* matrix. B is so-called *basic* matrix for the LP problem. We denote by x_B the sub-vector of x of dimension m of *basic* variables associated to matrix B and x_N the sub-vector of x of dimension n of *nonbasic* variables associated to N.
The problem can then be written as follows:

$$\begin{bmatrix} x_0 \\ x_B \end{bmatrix} = \begin{bmatrix} c_B B^{-1} b \\ B^{-1} b \end{bmatrix} - \begin{bmatrix} c_B B^{-1} N - c_N \\ B^{-1} N \end{bmatrix} x_N. \tag{3}$$

Remark: By setting $x_N = 0$, $x_B = B^{-1}b$ and $x_0 = c_B B^{-1} b$, a feasible basic solution is obtained if $x_B \geq 0$.

Simplex tableau

We introduce now the following notations:

- $\begin{bmatrix} s_{0,0} \\ s_{1,0} \\ \vdots \\ s_{m,0} \end{bmatrix} \equiv \begin{bmatrix} c_B B^{-1} b \\ B^{-1} b \end{bmatrix}$
- $\begin{bmatrix} s_{0,1} & s_{0,2} \cdots s_{0,n} \\ s_{1,1} & s_{1,2} \cdots s_{1,n} \\ \vdots & \vdots \ddots \vdots \\ s_{m,1} & s_{m,2} \cdots s_{m,n} \end{bmatrix} \equiv \begin{bmatrix} c_B B^{-1} N - c_N \\ B^{-1} N \end{bmatrix}$

Then (3) can be written as follows:

$$\begin{bmatrix} x_0 \\ x_{B_1} \\ \vdots \\ x_{B_m} \end{bmatrix} = \begin{bmatrix} s_{0,0} \\ s_{1,0} \\ \vdots \\ s_{m,0} \end{bmatrix} - \begin{bmatrix} s_{0,1} & s_{0,2} \cdots s_{0,n} \\ s_{1,1} & s_{1,2} \cdots s_{1,n} \\ \vdots & \vdots \ddots \vdots \\ s_{m,1} & s_{m,2} \cdots s_{m,n} \end{bmatrix} x_N. \tag{4}$$

From (4), we construct the so called *Simplex tableau* as shown in Table I.

Table I
SIMPLEX TABLEAU

x_0	$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$\cdots$	$s_{0,n}$
x_{B_1}	$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$\cdots$	$s_{1,n}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
x_{B_m}	$s_{m,0}$	$s_{m,1}$	$s_{m,2}$	$\cdots$	$s_{m,n}$

By adding the slack variables in LP problem (see 2) and setting $N = A^{'}$, $B = I_m \Rightarrow B^{-1} = I_m$, a first basic feasible solution can be written as follows:
$x_N = x^{'} = (0, 0, ..., 0) \in \mathbf{R}^n$ and $x_B = B^{-1}b = b$.
At each iteration of the Simplex algorithm, we try to replace a basic variable, the so-called *leaving variable*, by a nonbasic variable, the so-called *entering variable*, so that the objective function is increased. Then, a better feasible solution is yielded by updating the Simplex tableau. More formally, the Simplex algorithm implements iteratively the following steps:

- **Step 1:** Compute the index k of the smallest negative value of the first line of the Simplex tableau, i.e.

$$k = \arg \min_{j=1,2,\ldots,n} \{s_{0,j} \mid s_{0,j} < 0\}.$$

The variable x_k is the *entering variable*. If no such index is found, then the current solution is optimal, else we go to the next step.

- **Step 2:** Compute the ratio $\theta_{i,k} = s_{i,0}/s_{i,k}, i = 1, 2, \cdots, m$ then compute index r as:

$$r = \arg \min_{i=1,2,\cdots,m} \{\theta_{i,k} \mid s_{i,k} > 0\}.$$

The variable x_{B_r} is the *leaving variable*. If no such index is found, then the algorithm stops and the problem is *unbounded*, else the algorithm continues to the last step.

- **Step 3:** Yield a new feasible solution by updating the previous basis. The variable x_{B_r} will leave the basis and variable x_k will enter into the basis. More formally, we start by saving the kth column which becomes the so-called 'old' kth column, then the Simplex tableau is updated as follows:
1 - Divide the rth row by the pivot element $s_{r,k}$:

$$s_{r,j} := \frac{s_{r,j}}{s_{r,k}}, \qquad j = 0, 1, \cdots, n.$$

2 - Multiply the new rth row by $s_{i,k}$ and subtract it from the ith row, $i = 0, 1, \cdots, m, i \neq r$:

$$s_{i,j} := s_{i,j} - s_{r,j} s_{i,k}, j = 0, 1, \cdots, n.$$

3 - Replace in the Simplex tableau, the old kth column by its negative divided by $s_{r,k}$ except for the pivot element $s_{r,k}$ which is replaced by $1/s_{r,k}$:

$$s_{i,k} := -\frac{s_{i,k}}{s_{r,k}}, \qquad i = 0, 1, \cdots, m \quad , \quad i \neq r,$$

and

$$s_{r,k} := \frac{1}{s_{r,k}}.$$

This step of the Simplex algorithm is the most costly in terms of processing time.
Then return to the step 1.

The Simplex algorithm finishes in 2 cases:

- when the optimal solution is reached (then the LP problem is solved).
- when the LP problem is unbounded (then no solution can be found).

The Simplex algorithm described by Garfinkel and Nemhauser is interesting in the case of dense LP problems since the size of the manipulated matrix (Simplex tableau) is $(m+1) \times (n+1)$. This decreases the memory occupancy and processing time, and permits one to test larger instances. In the sequel, we present the parallelization of this algorithm on GPU.

III. Simplex on CPU-GPU system

This section deals with the CPU- GPU implementation of the Simplex algorithm via CUDA. For that, a brief description of the GPU architecture is given in the following paragraph.

A. NVIDIA GPU architecture

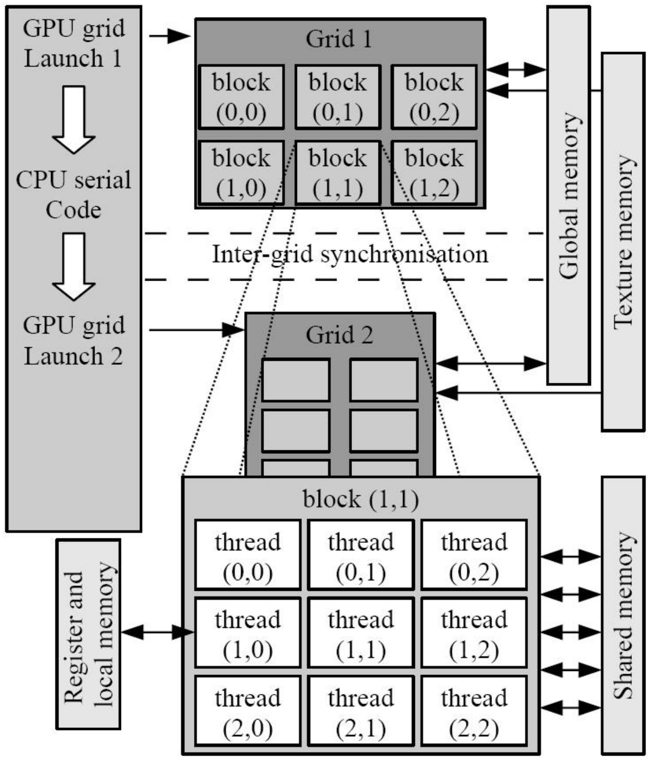

Figure 1. Thread and memory hierarchy in GPUs.

NVIDIA's GPUs are SIMT (single-instruction, multiple-threads) architectures, i.e. the same instruction is executed simultaneously on many data elements by the different threads. They are especially well-suited to address problems that can be expressed as data-parallel computations.
As shown in Figure 1, a *grid* represents a set of blocks where each block contains up to 512 threads. A grid is launched via a single CUDA program, the so-called kernel. The execution starts with a host (CPU) execution. When a kernel function is invoked, the execution is moved to a device (GPU). When all threads of a kernel complete their execution, the corresponding grid terminates, the execution continues on the host until another kernel is invoked. When a kernel is launched, each multiprocessor processes one block by executing threads in group of 32 parallel threads named *warps*. Threads composing a warp start together at the same program address, they are nevertheless free to branch and execute independently. As thread blocks terminate, new blocks are launched on the

idle multiprocessors. Threads of different blocks cannot communicate with each other explicitly but can share their results by means of a global memory.

Remark: If threads of a warp diverge when executing a data-dependent *conditional* branch, then the warp serially executes each branch path. This leads to poor efficiency.

Threads have access to data from multiple memory spaces (see Figure 1). We can distinguish two principal types of memory spaces:

- *Read-only memories*: the *constant* memory for constant data used by the process and *texture* memory optimized for 2D spatial locality. These two memories are accessible by all threads.
- *Read and write memories*: the *global* memory space accessible by all threads, the *shared* memory spaces accessible only by threads in the same blocks with a high bandwidth, and finally each thread accesses to his own *registers* and *private local* memory space.

In order to have a maximum bandwidth for the global memory, memory accesses have to be coalesced. Indeed, the global memory access by all threads within a half-warp (a group of 16 threads) is done in one or two transactions if:

- the size of the words accessed by the threads is 4, 8, or 16 bytes,
- all 16 words lie:
  - in the same 64-byte segment, for words of 4 bytes,
  - in the same 128-byte segment, for words of 8 bytes,
  - in the same 128-byte segment for the first 8 words and in the following 128-byte segment for the last 8 words, for words of 16 bytes;
- threads access the words in sequence (the kth thread in the half-warp accesses the kth word).

Otherwise, a separate memory transaction is issued for each thread, which degrades significantly the overall processing time. For further details on the NVIDIA cards architecture and how to optimize the code, reference is made to [18].

When implementing the Simplex method, most of the time is spent in pivoting operations. This step involves $(m+1) \times (n+1)$ double precision multiplications and subtractions that can be parallelized on GPU.
The $SimplexTableau$ that is available first on the CPU must be allocated to the *Global Memory* of the GPU. This requires communications between the CPU and the GPU. The pivoting operations will be carried out by the GPU.
A Simplex tableau of size $(m+1) \times (n+1)$ is decomposed into $h \times w$ blocks with :

$$h = \lceil (m+1+16)/16 \rceil$$
$$w = \lceil (n+1+32)/32 \rceil$$

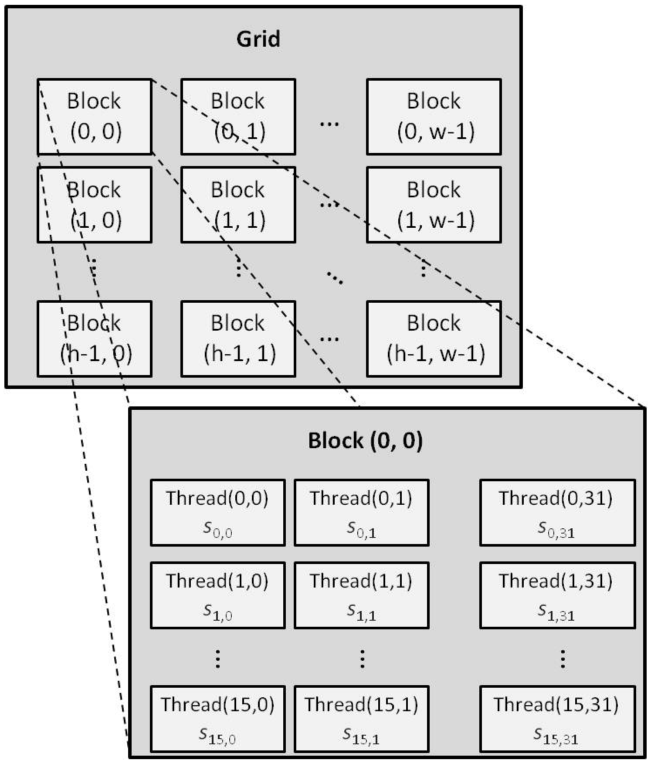

Figure 2. Allocation on GPU of the Simplex Tableau.

Each block is relative to a sub-matrix with 16 lines and 32 columns; this corresponds to a block of 512 threads (the maximum number of threads per block). The grid of blocks covers all the Simplex tableau and each thread is associated to a given entry of the tableau (see Figure 2).
The main steps of the Simplex algorithm are described in Figure 3.

B. *Computing the entering and leaving variables:*

Finding the entering or leaving variables results in finding a minimum within a set of values. This can be done on GPU via reduction techniques. However, our experiments showed that we obtain better performance by doing this step sequentially on the CPU. Indeed, the size of the tested problems and the double precision operations lead to worst efficiency for the parallel approach. More explicitly, finding a minimum in a row of 10000 values, which corresponds to the maximal row size treated in our experiments, takes an average time of $0.27ms$ on CPU and $3.17ms$ on GPU at each step of the Simplex algorithm. Furthermore, the use of atomic functions of CUDA is not possible in this case since they do not support double precision operations.
Thus, this step is implemented in CPU and the minimum index is thereafter communicated to the GPU (see Figure 3).
In step 1 of the Simplex algorithm, the first line of the Simplex tableau is simply communicated to the CPU.

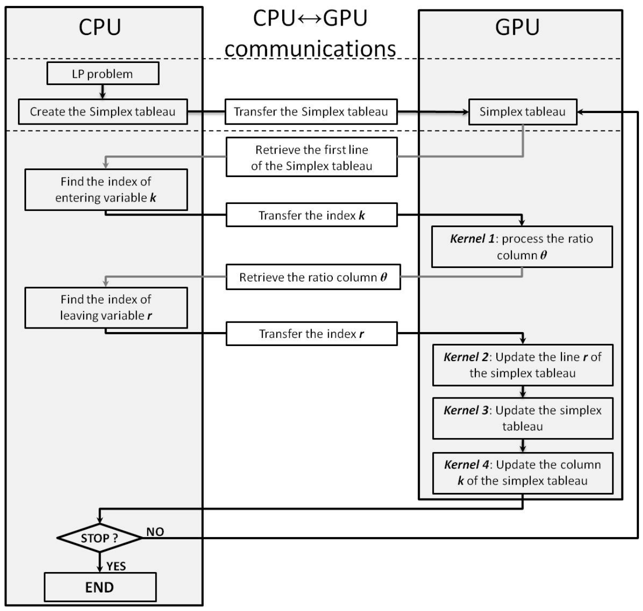
CPU
CPU↔GPU communications
GPU
LP problem
Create the Simplex tableau
Transfer the Simplex tableau
Simplex tableau
Retrieve the first line of the Simplex tableau
Find the index of entering variable *k*
Transfer the index *k*
Kernel 1: process the ratio column *θ*
Retrieve the ratio column *θ*
Find the index of leaving variable *r*
Transfer the index *r*
Kernel 2: Update the line *r* of the simplex tableau
Kernel 3: Update the simplex tableau
Kernel 4: Update the column *k* of the simplex tableau
STOP ?
NO
YES
END

Figure 3. Simplex algorithm on a CPU-GPU system.

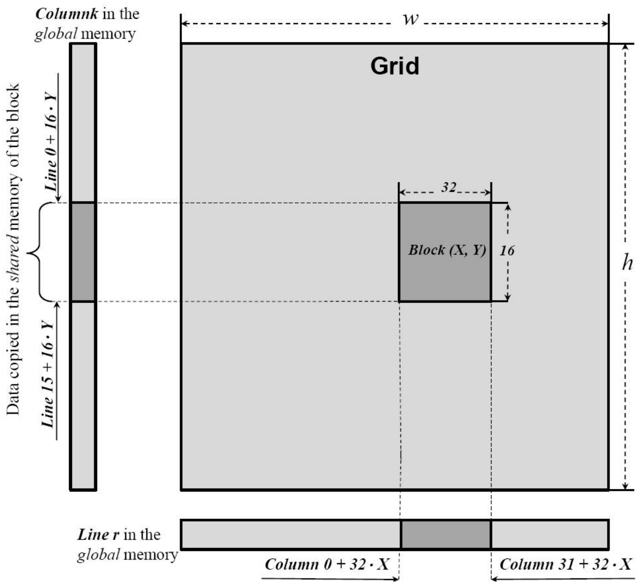
Columnk in the *global* memory
w
Grid
Data copied in the *shared* memory of the block
Line 0 + 16 · Y
Line 15 + 16 · Y
32
16
Block (X, Y)
h
Line r in the *global* memory
Column 0 + 32 · X
Column 31 + 32 · X

Figure 4. Matrix manipulation and memory management in kernel 3.

However, the step 2 requires the processing of a column of ratios. This is done in parallel by Kernel 1 and the ratio column θ is communicated to the CPU. Since step 3 requires the column k, the column of entering variable of the Simplex tableau, Kernel 1 is also used to get the "old" $Columnk$ and to store it in the device memory in order to avoid the case of memory conflict.

Remark: Communication with the CPU use page-locked host memory in order to have a higher bandwidth between host memory and device memory.

Kernel 1: The GPU Kernel for processing ratio column θ and getting the "old" column $Columnk$ of entering index k.

```
global_void Kernel1($double\ SimplexTableau[m+1][n+1]$
                    $double\ *\theta$,
                    $double\ *Columnk, int\ k$)
{
$int\ idx = blockDim.x * blockIdx.x + threadIdx.x$;
$double\ w = SimplexTableau[idx][k]$;
/*Copy the weights of entering index k*/
$Columnk[idx] = w$;
$\theta[idx] = SimplexTableau[idx][1]/w$;
}
```

C. Updating the basis:

In the sequel, we use the standard CUDA notation whereby x, y denote the column and the row, respectively. Step 3 is entirely carried out on the GPU. The line of the Simplex tableau relative to the index of the leaving variable r is updated as follows by Kernel 2: The thread x of block X processes the element $SimplexTableau[r][x+32\times X]$ with $x = 0,\cdots,31$ and $X = 0,\cdots,w-1$. The pivot element $SimplexTableau[r][k]$ obtained from the old $Columnk[r]$, is shared between all threads. It is more beneficial to use the *shared memory* which is expected to be much faster than *global* memory (see [18]).

According to Figure 2, the remaining part of the Simplex tableau is updated by the Kernel 3. Indeed, for each block of dimension 16×32, a column of 16 element of leaving index k is loaded in a shared memory. Then blocks process the corresponding part of Simplex tableau independently (in parallel) such as thread (x, y) of the block (X, Y) processes the element $SimplexTableau[y+16\times Y][x+32\times X]$ with $x = 0,\cdots,31$, $y = 0,\cdots,15$ and $X = 0,\cdots,w-1$, $Y = 1,\cdots,h-1$ (see Figure 4).

Updating the column k of Simplex tableau requires the old $Columnk$ and in order to avoid the addition of branching condition like $if(idx == k)\ \ return$; in Kernel 3, which results in a divergent branch, we use kernel 4 to update separately the column k.

Thus, step 3 requires the three following kernels:

Kernel 2: The GPU Kernel for processing the line relative to the index of the leaving variable r.

```
global_void Kernel2($double\ SimplexTableau[m+1][n+1]$
                    $double\ *Columnk, int\ k, int\ r$)
{
$int\ idx = blockDim.x * blockIdx.x + threadIdx.x;$
$\_\_shared\_\_\ double\ w;$
/*Get the pivot element : $SimplexTableau[r][k]$ in the shared memory */
$if(threadIdx.x == 0)\quad w = Columnk[r];$
$\_\_syncthreads();$
/*Update the line of leaving index $r$*/
$SimplexTableau[r][idx] = SimplexTableau[r][idx]/w;$
}
```

Kernel 3: The GPU Kernel for Updating the basis.

```
global_void Kernel3($double\ SimplexTableau[m+1][n+1]$
                    $double\ *Columnk, int\ k, int\ r$)
{
$int\ idx = blockDim.x * blockIdx.x + threadIdx.x;$
$int\ jdx = blockIdx.y * blockDim.y + threadIdx.y;$
$\_\_shared\_\_\ double\ w[16];$
/*Get the column of entering index $k$ in shared memory */
$if(threadIdx.y == 0\ \ \&\&\ \ threadIdx.x < 16)$
{
$w[threadIdx.x] = Columnk[blockIdx.y * blockDim.y +$
                             $threadIdx.x];$
}
$\_\_syncthreads();$
/*Update the basis except the line $r$*/
$if(jdx == r)\quad return;$
$SimplexTableau[jdx][idx]$=$SimplexTableau[jdx][idx] -$
              $w[threadIdx.y] * SimplexTableau[r][idx];$
}
```

Kernel 4: The GPU Kernel for processing the column of entering index k.

```
global_void Kernel4($double\ SimplexTableau[m+1][n+1]$
                    $double\ *Columnk, int\ k, int\ r$)
{
$int\ jdx = blockDim.x * blockIdx.x + threadIdx.x;$
$\_\_shared\_\_\ double\ w;$
/*Get the pivot element : $SimplexTableau[r][k]$ in the shared memory */
$if(threadIdx.x == 0)\quad w = Columnk[r];$
$\_\_syncthreads();$
/*Update the column of the entering index $k$*/
$SimplexTableau[jdx][k] = -Columnk[jdx]/w;$
/*Update the pivot element $SimplexTableau[r][k]$*/
$if(jdx == r)\quad SimplexTableau[jdx][k]$=$1/w;$
}
```

IV. Computational experiments

We present now computational results for CPU-GPU and CPU implementations of the Simplex algorithm. Experiments have been carried out on a CPU with Intel Xeon 3.0 GHz and a NVIDIA GTX 260 GPU. The GTX 260 has 192 cores and a 1.4 GHz clock frequency.

We have considered randomly generated LP problems where $a_{ij}, b_i, c_j, i \in \{1, ..., m\}$ and $j \in \{1, ..., n\}$, are integer variables that are uniformly distributed over the integer$[1,\ 1000]$. We can note that the generated matrix A is a *non-sparse* matrix. We have been using *double precision* in order to insure a good precision of the solution. Processing times are given for 10 instances and the resulting speedups have been computed as follows:

$$speedup = \frac{processing\ time\ on\ CPU(s.)}{processing\ time\ on\ CPU\text{-}GPU(s.)}.$$

We note that *processing time on CPU* corresponds to the time obtained with the sequential version of the same Simplex algorithm implemented on the CPU.

Figure 5 displays processing times for the different sizes of LP problems and for both sequential and parallel algorithms.

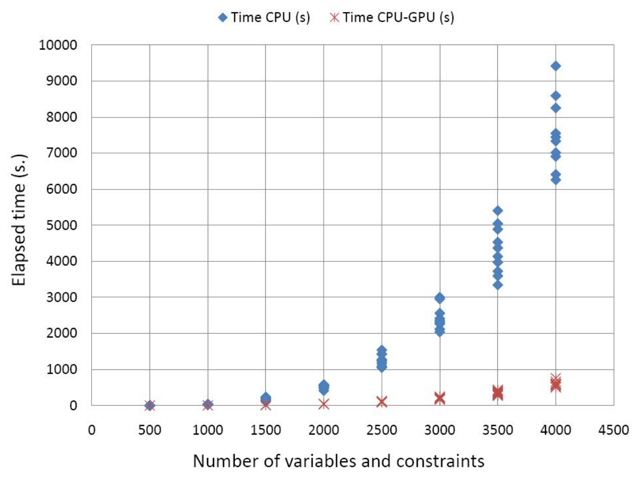

Figure 5. Elapsed time (simplex on CPU and CPU-GPU).

We can see that for each size of LP problem, processing time dispersion is low. Although processing time increases with the size of problems (see Table II), the parallel algorithm is always faster than the sequential algorithm.

Figure 6 shows that speedup increases with the size of problems. An average speedup of 12.61 has been obtained for large instances ($\geq$ 3000 $\times$ 3000). Large instances, e.g. 6000$\times$6000, 7000$\times$7000 and 8000$\times$8000, leading to speed

Table II
PROCESSING TIME OF THE SIMPLEX ALGORITHM (S).

$m \times n$	CPU	CPU-GPU system
500×500	1.16	0.44
1000×1000	29.28	3.37
1500×1500	184.21	16.21
2000×2000	524.61	44.30
2500×2500	1250.51	106.27
3000×3000	2432.42	196.92
3500×3500	4301.11	350.00
4000×4000	7517.75	596.34

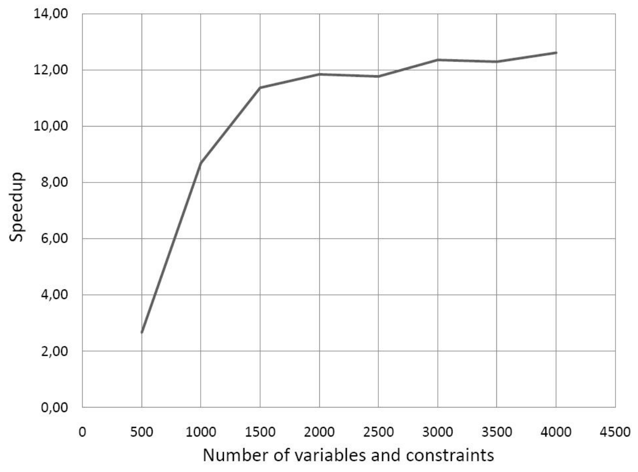

Figure 6. Average speedup.

up 12.5 and processing time $> 11hours$ have been tested without exceeding the memory capacity of the GPU card. For small size problems e.g. 500×500, the speedup is relatively small ($average\ speedup = 2.66$) since the real power of the GPU is slightly exploited in this case. We note that parallel implementation of Simplex algorithm on GPU permits one to solve efficiently larger problems within small processing time. Finally we note that our experimental results can hardly be compared with the one in [10] since this reference deals with the solution of sparse LP problems via the revised Simplex method and computations are performed in single precision in [10].

V. CONCLUSION AND FUTURE WORK

In this article we have proposed a parallel implementation of the Simplex method for solving linear programming problems on CPU-GPU system with CUDA. The parallel implementation has been performed by optimizing the different steps of the Simplex algorithm. Computational results show that our implementation in double precision on CPU-GPU system is efficient since for large non-sparse linear programming problems, we have obtained stable speedups around 12.5. Our approach permits one also to solve problems of size 8000×8000 without exceeding the memory capacity of the GPU.

In future work, we plan to test larger LP problems on multi GPU architecture. We plan also to implement the revised Simplex algorithm on GPU without using CUBLAS or LAPACK libraries in order to go on further in the optimization of the parallel implementation.

ACKNOWLEDGMENT

Dr Didier El Baz thanks NVIDIA for support through Academic Partnership.

REFERENCES

[1] V. Boyer, D. El Baz, M. Elkihel, "Dense dynamic programming on multi GPU," in *Proc. of the 19th International Conference on Parallel Distributed and networked-based Processing, PDP 2011, Ayia Napa, Cyprus*, 545–551, February 2011.

[2] E. B. Ford, "Parallel algorithm for solving Keplers equation on graphics processing units: application to analysis of Doppler exoplanet searches," *New Astronomy*, 14:406-412, 2009.

[3] Y. Zhang, J. Cohen, J. D. Owens, "Fast tridiagonal solvers on the GPU," in *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (PPoPP 2010):127–136, Bangalore, India, January 2010.

[4] V. Vineet, P. J. Narayanan, "CUDA cuts: fast graph cuts on the GPU," in *Workshop on Visual Computer Vision on GPU's*, 2008.

[5] D. P. O'Leary, J. H. Jung, "Implementing an interior point method for linear programs on a CPU-GPU system," *Electronic Transactions on Numerical Analysis*, 28:879–899, May 2008.

[6] NETLIB, http://www.netlib.org/

[7] D. G. Spampinato, A. C. Elster, "Linear optimization on modern GPUs," in *Proc. of the 23rd IEEE International Parallel and Distributed Processing Symposium, (IPDPS 2009)*, Rome, Italy, May 2009.

[8] CUDA - CUBLAS Library 2.0, NVIDIA Corporation,

[9] LAPACK Library, http://www.culatools.com/

[10] J. Bieling, P. Peschlow, P. Martini, "An efficient GPU implementation of the revised Simplex method," in *Proc. of the 24th IEEE International Parallel and Distributed Processing Symposium, (IPDPS 2010)*, Atlanta, USA, April 2010.

[11] G. B. Dantzig, *Linear Programming and Extensions*, Princeton University Press and the RAND Corporation, 1963.

[12] G. B. Dantzig, M. N. Thapa, *Linear Programming 2: Theory and Extensions*, Springer-Verlag, 2003.

[13] S. S. Morgan, *A Comparison of Simplex Method Algorithms*, Master's thesis, Univ. of Florida, Jan. 1997.

[14] G. Yarmish, "The simplex method applied to wavelet decomposition," in *Proc. of the International Conference on Applied Mathematics, Dallas, USA*, 226–228, November 2006.

[15] J. Eckstein, I. Bodurglu, L. Polymenakos, and D. Goldfarb, “Data-Parallel Implementations of Dense Simplex Methods on the Connection Machine CM-2,” *ORSA Journal on Computing*,vol. 7,4:434–449, 2010.

[16] S. P. Bradley, U. M. Fayyad, and O. L. Mangasarian, “Mathematical Programming for Data Mining: Formulations and Challenges,” *INFORMS Journal on Computing*, vol. 11,3:217–238, 1999.

[17] V. Boyer, D. El Baz, M. Elkihel, “Solution of multidimensional knapsack problems via cooperation of dynamic programming and branch and bound,” *European J. Industrial Engineering*, 4,4:434–449, 2010.

[18] NVIDIA, Cuda 2.0 programming guide, http:// developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf (2009)

[19] R. S. Garfinkel, D. L. Nemhauser, *Integer Programming*, Wiley-Interscience, 1972.