2011 IEEE International Conference on High Performance Computing and Communications

Multi GPU Implementation of the Simplex Algorithm

Mohamed Esseghir Lalami, Didier El-Baz, Vincent Boyer
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, elbaz, vboyer@laas.fr

***Abstract*—The Simplex algorithm is a well known method to solve linear programming (LP) problems. In this paper, we propose an implementation via CUDA of the Simplex method on a multi GPU architecture. Computational tests have been carried out on randomly generated instances for non-sparse LP problems. The tests show a maximum speedup of 24.5 with two Tesla C2050 boards.**

***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. In [11], we have presented a parallel implementation via CUDA of the standard Simplex algorithm on CPU-GPU systems for dense LP problems. Experiments carried out on a CPU with 3 Ghz Xeon Quadro INTEL processor and a GTX 260 GPU card have shown substantial speedup of 12.5 in *double* precision. The authors have been recently aware of the paper [12] where the standard simplex method is implemented on a Tesla S1070 GPU card and CUBLAS library is used in the pivoting step. This approach differs from our implementation whereby we have tried to optimize, as much as possible, the pivoting step with CUDA on two Tesla C2050 GPU boards. 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 [13] and [14]), but for dense LP problems, the two approaches are equivalent (see [15] and [16]).

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 [17] and [18] for applications leading to dense LP problems.

In this paper, we propose an original solution based on multithreading in order to implement via CUDA the standard Simplex algorithm on multi GPU architectures. This solution is well suited to the case where CPUs are connected to several GPUs; it is also particularly efficient. We have been solving linear programming problems in the context of the solution of NP-complete combinatorial optimization problems (see [19]). 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],[2] and [11]).

The paper is structured as follows. Section 2 deals with the Simplex method. The multi GPU implementation of the Simplex algorithm is presented in Section 3. The Section 4 is devoted to presentation and analysis of computational results for randomly generated instances. Finally, in Section 5, we give some conclusions and perspectives.

978-0-7695-4538-7/11 $26.00 © 2011 IEEE
DOI 10.1109/HPCC.2011.32

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 :

$$\begin{array}{c} \max x_0 = cx^{'}, \\ s.t : A^{'}x^{'} \leq b^{'}, \\ x^{'} \geq 0, \end{array} \quad (1)$$

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:

$$\begin{array}{c} \max x_0 = cx, \\ s.t : Ax = b, \\ x \geq 0, \end{array} \quad (2)$$

with

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

$$A = \left(\ A^{'}, I_m \ \right) \in \mathbf{R}^{m \times (n+m)},$$

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 [13]). 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 [21] 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:

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

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:

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

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

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}	$s_{m,0}$	$s_{m,1}$	$s_{m,2}$	$\cdots$	$s_{m,n}$

Table I
Simplex tableau

By adding the slack variables in LP problem (see 2) and setting $N = A^{'}$, $B = I_m \Rightarrow B^{-1} = I_m$ a first solution can be written as follows:
$x_N = x^{'} = (0, 0, ..., 0) \in \mathbf{R}^n$ and $x_B = B^{-1}b = b$. Furthermore, if $x_B \geq 0$ this solution is named a first feasible basic solution.
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 $s_{r,k}$:

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

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 $1/s_{r,k}$:

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

and

$$s_{r,k} \text{ becomes } \frac{1}{s_{r,k}}.$$

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 $(m + 1) \times (n + 1)$ instead of $(m + 1) \times (n + m + 1)$ in the case of the standard Simplex method of Dantzig. 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 multi GPU architecture.

III. Simplex on Multi GPU system

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

A. NVIDIA GPU architecture

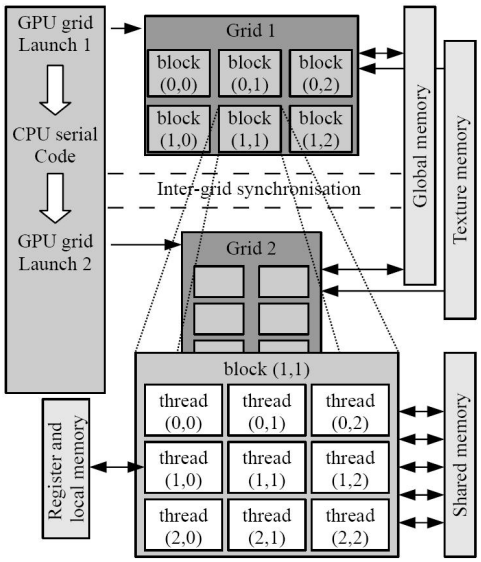

Figure 1. Thread and memory hierarchy in GPUs.

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 1024 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. With CUDA 3.0, 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 [20].
In [11], we have studied the parallel implementation via CUDA of the simplex method on a CPU/GPU system with a 3 GHz Xeon Quadro Intel processor and a GTX 260 GPU card. In this paper, we study the parallel implementation of the simplex method on a multi GPU architecture, i.e. a DELL Precision T7500 Westmere based on Quad-Core Intel Xeon E5640 2.66 GHz with 12 GB of main memory and two NVIDIA Tesla C2050 GPUs. The Tesla C2050 GPU, wich is based on the new-generation CUDA architecture codenamed $Fermi$, has 3 GB DDR5 of memory and 448 streaming processor cores (1.15 GHz) what delivers a peak performance of 515 Gigaflops in double precision floating point. The interconnection between the host and the two GPUs is done via a PCI-Express Gen2 interface.

B. Parallel algorithm

Principle

We denote by I the number of available GPUs. When

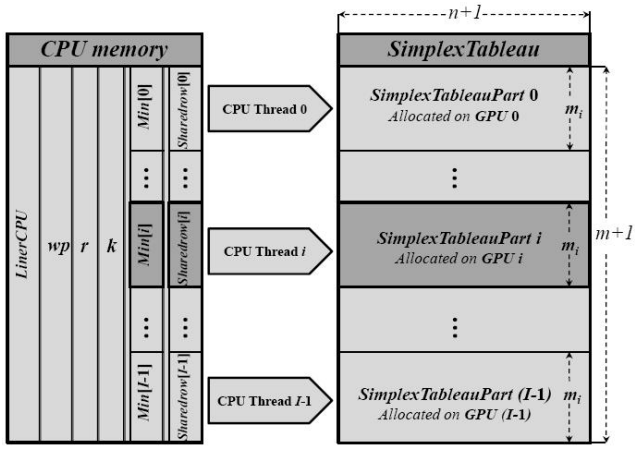

Figure 2. $SimplexTableau$ decomposition and memory access of CPU threads.

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 $(m+1) \times (n+1)$ double precision subtractions that can be parallelized on I GPUs.

For that purpose, the $SimplexTableau$ is decomposed

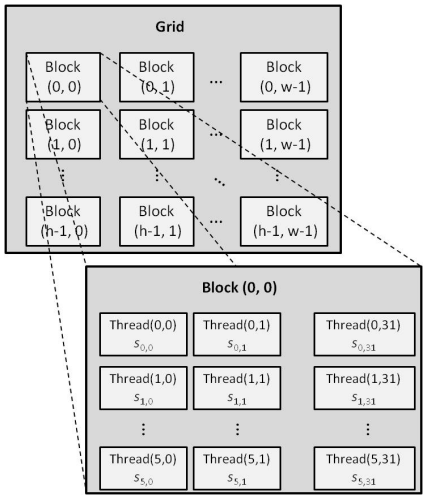

Figure 3. Allocation on GPU of a $SimplexTableauPart$.

into I parts (see Figure 2) so that each GPU updates one part at every iteration. The proposed implementation is based on the concurrent implementation of I identical CPU threads so that each GPU i is managed by its own CPU thread i, $i = 1, ..., I$. Hence, the CPU thread l is composed of 4 kernels required by the Simplex algorithm and these 4 kernels are executed on the part i of the $SimplexTableau$ by the $i-th$ GPU. This approach permits one to maintain the context of each CPU thread all along the application, i.e. CPU threads are not killed at the end of each Simplex

algorithm iteration. As a consequence, communications are minimized.
We chose to split *horizontally* the $SimplexTableau$ since this decomposition permits one to process the *ratio column* θ and the *entering variable column* in parallel between all GPUs. The *leaving variable line* is computed by only one GPU at each iteration.

Initialisation:

The $SimplexTableau$ of size $(m+1) \times (n+1)$ that is available first on the CPU, is split into I parts , called $SimplexTableauPart$. Each $SimplexTableauPart$ of size $m_i \times (n+1)$ where $m_i = \lceil (m+1)/I \rceil$, is allocated to the *Global Memory* of one GPU. This requires communications between the CPU and the GPUs. The pivoting operations will be carried out by the GPUs.
The $SimplexTableauPart$ is decomposed into $h \times w$ blocks in the GPU with :

$$h = \lceil m_i/6 \rceil$$
$$w = \lceil (n+1)/32 \rceil$$

Each block is relative to a submatrix with 6 lines and 32 columns; this corresponds to a block of 192 threads (the optimal number of GPU threads per block that minimize processing time). The grid of blocks covers all the $SimplexTableauPart$ and each GPU thread is associated to a given entry of the tableau (see Figure 3).

Thread processing:

The procedure carried out by the CPU in one iteration is described in the *Simplex Thread* Algorithm .

Algorithm *Simplex Thread (ith processing thread on CPU):*

/* Shared data between CPU Threads */
$Min[I], k, r, wp, LinerCPU[n+1], Sharedrow[I][m_i]$,
/* Local CPU Thread data */
$SimplexTableauPart[m_i][n+1]$,
$Columnk[m_i]$, $Liner[n+1]$,
begin *Procedure*
/* Computing entering variable index */
if $i = 0$ do
GPU to CPU com($SimplexTableauPart[0], Sharedrow$),
end if
Synchronize()
$Min[i]$:= Find_min($Sharedrow[i]$),
Synchronize()
k :=Find_min(Min),
/* Computing leaving variable index */
Kernel1(),
GPU_to_CPU_com($\theta, Sharedrow[i]$),
$Min[i]$:= Find_min($Sharedrow[i]$),
Synchronize()
r :=Find_min(Min),
/* Updating basis parts */
id := Find_GPUid_pivot(),
if $i = id$
  wp := Get_pivot_GPU_to_CPU(),
  Kernel2(),
  GPU_to_CPU_com($Liner, LinerCPU$),
  Set_pivot_line_to_0(),
end if
CPU_to_GPU_com($LinerCPU, Liner$),
Kernel3(),
Kernel4(),
end *Procedure*.

We can distinguish in the Algorithm *Simplex Thread* two types of data: *shared* and *local* data.

- *Shared data:*
  k, r and wp are, respectively, the entering index, the leaving index and pivot element.
  $Sharedrow[I][m_i]$, a row of size $I \times m_i$, is used to receive the first line of the $SimplexTableau$ and the *ratio column* θ in order to process, respectively, k and r.
  $LinerCPU[n+1]$ receive the line r of the $SimplexTableau$ from the GPU which hosts the *pivot line*.
  These datas are stored in the *CPU memory* see (Figure 2), exactly in a page-locked host memory.
- *Local data:*
  Stored in the *global memory* of the GPUs, these datas are used by the GPUs kernels of the *Simplex Thread* Algorithm.

The function *Synchronize()* performs a global synchronization of all CPU threads in order to insure data consistency. GPU data exchanges are made via the following two functions:

- *GPU_to_CPU_com*($source, destination$): whereby each GPU writes in the CPU (destination) values contained in source.
- *CPU_to_GPU_com*($source, destination$): whereby each GPU reads values in the CPU (source).

Computing the entering and leaving variables:

Finding the entering or leaving variables results in finding a minimum within a set of values. In reference [11], we explained that it is better to do this step in CPU. Thus, the CPU function *Find_min()* is used by the CPU thread i to find the local minimum $Min[i]$ index in $Sharedrow[i]$. Thereafter a global minimum index is computed in the row Min of I values and communicated to the GPUs.
In step 1 of the Simplex algorithm, the first line of the Simplex tableau is simply communicated to the CPU in

$Sharedrow$ by the GPU 0. 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.

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

---

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

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 GPUs.
The thread function *Find_GPUid_pivot()* is used to find the index of the GPU which host the line of the Simplex tableau relative to the index of the leaving variable r. This line $Liner$ is processed by the Kernel 2 as follows: The GPU thread x of block X processes the element $Liner[x+32\times X]$ with $x = 0, \cdots, 31$ and $X = 0, \cdots, w-1$. The pivot element $wp := SimplexTableau[r][k]$ obtained from the old $Columnk[r]$, is shared between all GPU threads.
The $Liner$ and wp are sent to the CPU memory and thereafter stored in device memory of all GPUs. In order to avoid the addition of branching condition like $if(jdx == r) \ return$; in Kernel 3, the $SimplexTableau[r]$ line and $Columnk[r]$ value are set to , respectively, 0 and -1.
The remaining part of the Simplex tableau is updated by the Kernel 3. Indeed, for each block of dimension 6×32, a column of 6 element of leaving index k is loaded in a shared memory. Then blocks process the corresponding part of $SimplexTableauPart$ independently (in parallel) such as GPU thread (x, y) of the block (X, Y) processes the element $SimplexTableauPart[y + 6 \times Y][x + 32 \times X]$ with $x = 0, \cdots, 31$, $y = 0, \cdots, 5$ 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($double \ wp, int \ k, int \ r$,
$double \ * Columnk, double \ * Liner$,
$double \ SimplexTableauPart[m_l][n+1]$)
{
$int \ idx = blockDim.x * blockIdx.x + threadIdx.x$;
/*Set the rth element of the Columnk to -1 */
$if(threadIdx.x == 0) \ Columnk[r] = -1$;
$\_\_syncthreads()$;
/*Update the line of leaving index r*/
$Liner[idx] = SimplexTableauPart[r][idx]/wp$;
}

Kernel 3: The GPU Kernel for Updating the basis.

---

global_void Kernel3($int \ r, int \ k, int \ r$,
$double \ * Columnk, double \ * Liner$,
$double \ SimplexTableauPart[m_l][n+1]$)
{
$int \ idx = blockDim.x * blockIdx.x + threadIdx.x$;
$int \ jdx = blockIdx.y * blockDim.y + threadIdx.y$;
$double \ s = SimplexTableauPart[jdx][idx]$;
$\_\_shared\_\_ \ double \ w[6]$;
/*Get the column of entering index k in shared memory */
$if(threadIdx.y == 0 \ \&\& \ threadIdx.x < 6)$
{
$w[threadIdx.x] = Columnk[blockIdx.y * blockDim.y + threadIdx.x]$;
}
$\_\_syncthreads()$;
/*Update the basis part */
$SimplexTableauPart[jdx][idx] = s - w[threadIdx.y] * Liner[idx]$;
}

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

---

global_void Kernel4($double \ * Columnk, double \ wp$,
$double \ SimplexTableauPart[m_l][n+1]$)
{
$int \ jdx = blockDim.x * blockIdx.x + threadIdx.x$;
/*Update the column of the entering index k*/
$SimplexTableauPart[jdx][k] = -Columnk[jdx]/wp$;

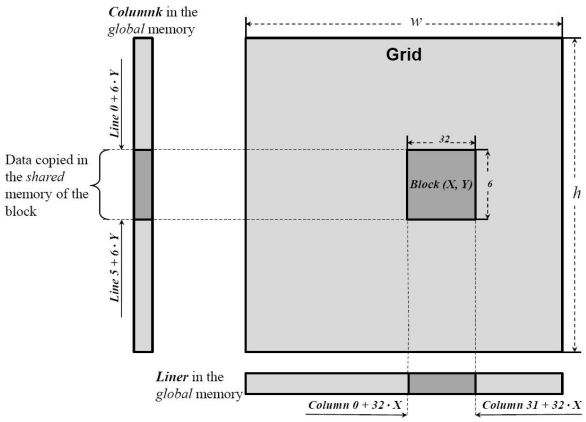

Figure 4. Matrix manipulation and memory management in kernel 3.

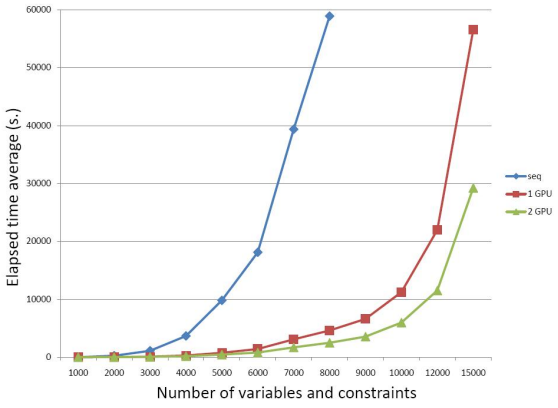

Figure 5. Elapsed time (simplex on CPU, one GPU and two GPUs).

IV. Computational experiments

We present now computational results obtained with one CPU, a system with one CPU and one GPU and a system with one CPU and two GPUs, respectively. We have used a CPU with Intel Xeon E5640 2.66 GHz and a two NVIDIA Tesla C2050 GPUs.We recall that we have used CUDA 3.2 for the parallel code and gcc for the serial one.

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 note that the generated matrix A is a *non-sparse* matrix. We have been using *double precision* in order to ensure a good precision of the solution. Processing times are given for 5 instances and the resulting speedups have been computed as follows:

$$\text{speedup1} = \frac{\text{processing time on CPU (s.)}}{\text{processing time on system with one GPU (s.)}}$$

$$\text{speedup2} = \frac{\text{processing time on system with one GPU (s.)}}{\text{processing time on system with two GPUs (s.)}}$$

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. $speedup1$ is the obtained speedup between CPU and system with one GPU. $speedup2$ is the obtained speedup between one GPU and system with two GPUs.

Figure 5 displays processing times for the different sizes of LP problems and for both sequential and parallel algorithms. We can see that the parallel algorithms (with one and two GPUs) are always faster than the sequential algorithm. In the sequential case and for problems of size 8000×8000, we note that the processing time exceeds the time limit of $16\ hours$ that we have imposed.

Table II shows that both $speedup1$ and $speedup2$ increase with the size of problems. For small size problems e.g. 1000×1000, the $speedup1$ is relatively small ($average\ speedup = 2.93$) since the real power of the GPU is slightly exploited. We note that an average speedup of 12.7 is obtained with one GPU. For large instances, $speedup2$ increases and meets a level around 1.93. This shows that the use of two GPUs leads to a very small lost of efficiency for large problems. This gives a speedup of 24.5 between CPU and two GPU implementations. We note also that, large instances, i.e 19000×19000 on system with one GPU and 27000×27000 on system with two GPUs, have been tested without exceeding the memory occupancy of GPU cards. This confirms the interest of the proposed approach since the use of several GPUs permits one to solve efficiently larger problems within reasonable processing time.

Finally we note that our experimental results can hardly be compared with the one in [11] where we have used a CPU twice as slow as the CPU used here.

$m \times n$	$speedup1$	$speedup2$
1000×1000	2.93	0.43
2000×2000	10.62	1.02
3000×3000	12.24	1.38
4000×4000	12.73	1.59
5000×5000	12.71	1.71
6000×6000	12.74	1.76
7000×7000	12.72	1.82
8000×8000	12.74	1.85
9000×9000	/	1.86
10000×10000	/	1.88
12000×12000	/	1.91
15000×15000	/	1.93

Table II
Average speedups: $speedup1$ (one CPU/ one GPU) and $speedup2$ (one GPU/ two GPUs).

V. Conclusion and future work

In this paper we have proposed a multi GPU parallel implementation in double precision of the Simplex method for solving linear programming problems with CUDA. The

parallel implementation has been performed by optimizing the different steps of the Simplex algorithm. Computational results show that our multi GPU implementation in double precision is efficient since for large non-sparse linear programming problems, we have obtained stable speedups around 24.5. Our approach permits one also to solve problems of size 15000×15000 without exceeding the memory occupancy of the GPUs.
In future work, we plan to test larger LP problems with more GPUs.

VI. Acknowledgment

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

References

[1] V. Boyer, D. El Baz, M. Elkihel, "Solving knapsack problems on GPU," in *Computers & Operations Research*.

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

[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] M. E. Lalami, V. Boyer, D. El Baz, "Efficient Implementation of the Simplex Method on a CPU-GPU System," in *Proceedings of the Symposium IEEE IPDPS 2011, Anchorage USA*, May 2011.

[12] X. Meyer, P. Albuquerque, B. Chopard "A multi-GPU implementation and performance model for the standard simplex method" submitted to *the 17th International European Conference on Parallel and Distributed Computing*, 2011.

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

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

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

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

[17] 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.

[18] 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.

[19] 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.

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

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