A multi-GPU implementation and performance model for the standard simplex method

Xavier Meyer[1,2], Paul Albuquerque[1], Bastien Chopard[2]

Institute inIT, University of Applied Science of Western Switzerland[1]
Computer Science Dept., University of Geneva[2]

Abstract. The standard simplex method is a well-known optimization algorithm for solving linear programming models in the field of operational research. It is part of software that is often employed by businesses for solving scheduling or assignment problems. But their always increasing complexity and size drives the demand for more computational power. In the past few years, GPUs have gained a lot of popularity as they offer an opportunity to accelerate many algorithms. In this paper we present a mono and a multi-GPU implementation of the standard simplex method, which is based on CUDA. Measurements show that it outperforms the CLP solver provided the problem size is large enough. We also derive a performance model and establish its accurateness. To our knowledge, only the revised simplex method has so far been implemented on a GPU.

1 Introduction

The simplex method is a well-known optimization algorithm for solving linear programming (LP) models in the field of operational research. It is part of software that is often employed by businesses for finding solutions to problems such as the fleet assignment and crew scheduling in the airline industry. It was proposed by Dantzig in 1947 [1] and is often referred to as the standard simplex method. A more efficent method in terms of the number of arithmetic operations, named the revised simplex, was later developed. Nowadays its sequential implementation can be found in almost all commercial LP solvers. But the always increasing complexity and size of LP problems from the industry, drives the demand for more computational power. Indeed, current implementations of the revised simplex strive to produce the expected results, if any. In this context, parallelisation is the natural idea to investigate [2]. Already in 1996, Thomadakis et al. [3] implemented the standard method on a massively parallel computer and obtained an increase in performances when solving dense or large problems.

A few years back, in order to meet the demand for processing power, graphics card vendors made their graphical processing units (GPU) available for general-purpose computing. Since then GPUs have gained a lot of popularity as they offer an opportunity to accelerate algorithms having an architecture well-adapted to the GPU model. The simplex method falls into this category. Indeed, GPUs exhibit a massively parallel architecture optimized for matrix processing. To our

knowledge, only the revised simplex method has so far been implemented on a GPU [4], showing encouraging results on small to mid-size LP problems.

In this paper, we present a single and a multi-GPU implementation of the standard simplex method, based on the CUDA technology of NVIDIA. We also derive the associated performance model and establish its accurateness. Let us mention that there are many technicalities involved in CUDA programming, in particular regarding the management of tasks and memories on the GPU. Thus fine tuning is indispensable to avoid a breakdown on performance. With respect to the algorithm, special attention has to be given to numerical stability.

The paper is organized as follows. First, we begin with a short description of the standard simplex method and introduce the heuristics used in our implementations. The next section covers the basics of GPU architecture and its peculiarities. In the following one, we focus on the single GPU implementation and then explain how we split the matrix representing the LP model to obtain the multiple GPU version. In the fourth section, we describe the performance models related to these implementations. In the fifth section, comparisons are made with the CLP solver, which is known to be one of the best non-commercial software, to assess the quality of our implementations. Finally, we summarize the results obtained and consider new perspectives.

2 The standard simplex method

The simplex method [1] applies to linear programming models. For the sake of simplicity, let us consider the canonical form of such problems :

$$\begin{array}{ll} \text{maximize} & z = \mathbf{c^T} \cdot \mathbf{x} \\ \text{subject to} & \mathbf{A} \cdot \mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{array} \tag{1}$$

where $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{A}$ is the $m \times n$ constraints matrix. The objective function $z = \mathbf{c^T} \cdot \mathbf{x}$ is the inner product of the cost vector $\mathbf{c}$ and the unknown variables $\mathbf{x}$. An element $\mathbf{x}$ is called a solution, which is said to be a feasible if it satisfies the linear constraints imposed by $\mathbf{A}$ and the bound $\mathbf{b}$.

The first step of the simplex method is to reformulate the model. So called *slack variables* x_{n+i} are added to the canonical form in order to replace the inequalities by equalities :

$$x_{n+i} = b_i - \sum_{j=1}^{n} A_{ij} x_j \leq b_i \quad (i = 1, 2, ..., m) \tag{2}$$

The resulting problem is called the *augmented form*. The variables are divided into two disjoint sets: the basic and nonbasic variables. Basic variables, which form the basis of the problem, are on the right hand side of the equations; nonbasic variables, which form the core of the equations, appear on the left hand side.

The simplex algorithm searches for the optimal solution through an iterative process. A typical iteration consists of three operations : selecting the entering

variable, choosing the leaving variable and pivoting. The entering variable is a nonbasic variable whose increase will lead to an augmentation of the objective value z. This variable must be selected with care in order to yield a substantial leap towards the optimal solution. The leaving variable is the basic variable which first violates its constraint as the entering variable increases. The choice of the leaving variable must guarantee that the solution remains feasible. Once both variables are defined, the pivoting operation makes these variables switch sets: the entering variable enters the basis, taking the place of the leaving variable which becomes nonbasic.

However, specific heuristics or methods are needed in order to improve the performance and stability of this algorithm. Our implementations use the *two-phase simplex* [1] to get hold of an initial feasible solution. Various methods for choosing the entering variable offer a compromise between computation cost and the number of iterations needed to solve the problem. Among these, the *steepest edge* [5] greatly reduces the number of iterations required. Moreover, its implementation fully benefits from the special architecture of a GPU. The stability and robustness of the algorithm depend considerably on the choice of the leaving variable. With respect to this, the *expand* [6] method proves to be very useful, particularly in the sense that it helps to avoid cycles.

3 GPU Architecture : CUDA

Since our programs are designed to run on a given type of GPU, we will focus on the parallel computing architecture developed by NVIDIA: Compute Unified Device Architecture (CUDA [7]). This architecture falls into the *Single Instruction Multiple Data* (SIMD) category. However, it is classified by NVIDIA under the concept of *Single Instruction Multiple Threads* (SIMT). The key feature of this architecture is to exploit data level parallelism and to offer fast context switching for threads. When used to process large matrices, the last generation of GPUs displays performances up to 1 TFLOPS on single precision operations.

A GPU appears as a device, usually connected to the motherboard via the PCI bus, which is controlled by the CPU. The global memory of the GPU is used to share data between both processing units. Moreover, it is possible to have several GPUs in a computer. In this case, each GPU must be managed by its own CPU thread. In a standard application, the CPU manages the data and the global algorithm, while the GPU does the computing, as in a master-slave paradigm. Due to the nature of the GPU architecture, conditional branching instructions are usually delegated to the CPU to avoid sequential execution of the GPU threads.

The architecture of a GPU is organized around two important notions : work decomposition and memory levels. The first one, work decomposition, describes the way a computation can be decomposed using multiple threads. The parallel processing is related to the architecture of the computing units on a GPU. The latter contains several *streaming multiprocessors* (SM). Each of them has a fixed number of processing units that execute in parallel instructions from a single

instruction unit. Thus, each thread running on a multiprocessor must execute the same instruction to guarantee true parallelism.

This hierarchy of processing units leads to decomposing a computation over a multitude of threads, each of them doing mainly the same work. Let us, for example, take the case of a matrix addition: **A=B+C**. Each thread could load a single element B_{ij} and C_{ij} from the matrices. The thread would then sum both values and store the result into A_{ij}. However, this set of threads has to be organized in order to be distributed on the SMs. This is why threads are grouped into blocks which will be dispatched to the SMs. Blocks are further organized into a grid of blocks, so as to provide a unique ID for each thread. The ID of a thread consists of the position of the thread in its block and the position of the latter on the grid. This ID is of crucial importance to access individual data during the computations.

The second notion concerns the different memory levels, each one offering specific advantages. A thread has its own registers on which the instructions are applied. Data accesses on registers are the fastest in this architecture. However, registers are shared inside a block, thus leading to a limited number of registers per thread. At another level, each block has its own shared memory accessible only by its own threads. This memory is nearly as fast as the registers when correctly accessed. More importantly, shared memory is the safest and fastest way to communicate between threads. Since SMs are not synchronized instruction-wise, threads on different blocks can not communicate. Finally, the main memory of the GPU (GRAM) is a global memory. This memory is within the scope of all threads as well as the CPU. But access to this memory is costly since, in addition to the read/write instruction, it takes 400 to 600 cycles before the data is available from the global memory.

To improve performances, a GPU architecture makes use of pipelines. There is a fixed depth (D) pipeline which depends on the number of processors per SM. Within a block, threads are grouped into *warps*. A *warp* consists of D batches whose threads can be executed concurrently by the SM. The main purpose of another pipeline is to hide the latency due to global memory accesses. The scheduler swaps *warps* that are waiting for data, with *warps* that are ready to be executed. The more warps, and thus threads, there are per block, the more efficiently this latency can be hidden. SIMT fast context switches becomes particularly handy to improve pipeline performances.

4 Implementations

4.1 Single GPU

We are going to consider a simplified version of our implementation to explain the main concepts. The first step performed by our implementation is done sequentially by the CPU. It consists in reading a linear programming model from a standard MPS file and transforming it to its augmented form. This modified problem is then entirely loaded in the global memory of the GPU. The next steps are described by algorithm 1.

Algorithm 1 Simplex method

Input: $prob_{dev}$ {Data structure containing the problem on the GPU}
Output: $feasible, infeasible$ {Feasibility of the problem}

while exists(x_{in}) **do**
  $x_{in} \leftarrow$ find_entering(p_{dev}) {GPU}
  $x_{out} \leftarrow$ find_leavingVar(x_{in}, p_{dev}) {GPU}
  if not exists(x_{out}) **then**
    return $infeasible$
  end if
  pivoting(x_{in}, x_{out}, p_{dev}) {GPU}
end while
return $feasible$

At each iteration, the CPU launches the *kernels*[1] and checks the results returned. The first *kernel* searches for the entering variable using the *steepest edge* method. The complexity of this method is $\mathcal{O}(mn)$ and benefits thoroughly from the massively parallel architecture of the GPU. This *kernel* uses the problem loaded in the global memory and returns only the selected entering variable. For the *expand* method used for choosing the leaving variable, it is required to use 2 or 3 *kernels* because of the different phases of this heuristic. Each phase has complexity $\mathcal{O}(m)$ and returns only one result. Finally, the pivoting operation requires a small constant amount of global memory accesses done by the CPU in a preliminary stage. Then the pivoting is done by a *kernel* available in CUBLAS, an optimized CUDA library for basic linear algebra. Once the final optimal solution is found, the CPU has to retrieve the problem data.

This algorithm ensures that the communications between the CPU and the GPU are minimized. Besides loading the $m \times n$ matrix representing the problem into the GPU memory and retrieving it into the CPU memory at the end of the computation, all other communications take a small and constant amount of time and can therefore be neglected when dealing with large problems.

The problem is stored in the global memory of the GPU in such a way as to optimize data access. GPUs are able to retrieve more than a single data in one instruction whenever the addresses are coalesced[2]. For example, a *streaming multiprocessor* is able to read single precision variables for up to 16 threads in a single instruction, if the addresses are coalesced. Such accesses substantially increase the global memory throughput.

4.2 Multi-GPU

The multi-GPU implementation uses the same algorithm as described previously in section 4.1. However, some communications have to be added in order to synchronize the GPUs. The pattern of the required communications is dependent on the way our LP problem is scattered across the GPUs. The constraint matrix

[1] CUDA functions are called *kernels*.
[2] CUDA concept for successive data addresses.

could be split: horizontally, which amounts to distributing the equations; vertically (see figure 1), which corresponds to distributing the variables; or by considering tiles. Each of these methods has its own advantages and disadvantages. The tile partitioning offers a good scalability but requires a lot of communications when the number of GPUs is *small* (up to a few tens). The vertical cut minimizes communications but is less scalable. However, since only enormous problems would take advantage of more than a few tens of GPUs, we selected the vertical splitting. The horizontal splitting involves duplicating more data and more communications. It was therefore discarded.

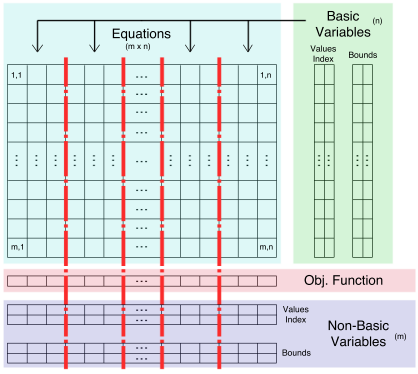

Fig. 1. Vertical splitting of the LP model

After reading and preparing the problem, the CPU has to create CPU threads that will manage the GPUs. Then, each of these CPU threads initializes its GPU and loads a sub-matrix in its GPU global memory. We will now describe what operations are done during an iteration of the multi-GPU implementation.

Concurrently, each CPU thread launches the *steepest edge kernel* so as to obtain a local entering variable. The first communication intervenes here: each GPU must share with the others its local result. They then reduce the local result so as to elect the same best entering variable. Due to the selected splitting, only the CPU thread that holds the entering variable, launches the *expand kernels* to select the leaving variable. The column of this variable, called *pivot column*, is required by all the other GPUs in order to execute the pivoting step. Thus this column, along with the leaving variable, is broadcasted. Finally, each GPU prepares and launches the pivoting *kernel* without any further communication.

It is important to point out that GPUs can not communicate directly. Communications take place through the CPU threads. This can either be done by using POSIX or MPI (Message Passing Interface). In our implementation, we chose POSIX since we used at most 4 GPUs on the same machine. The CPU threads communicate using a shared data structure stored in the CPU memory, and POSIX barriers permit to remain synchronized at each stage of an itera-

tion. However, if the implementation had to be run on a GPU cluster, the usage of MPI would be required. MPI proposes optimized schemes for reducing or scattering data that we did not use in this implementation.

5 Performance models

Performance models are useful to predict the behaviour of implementations as a function of various parameters. We are mainly interested to see how our single GPU implementation behaves when the problem size varies. In the multi-GPU case, the model will show what speedup can be obtained with several GPUs.

In order to model our implementations, we first have to explain how *kernels* are modelled. We can then focus on the single GPU implementation model. The multi-GPU is derived from the single GPU model.

5.1 Modelling a *kernel*

CUDA *kernels* require a different kind of modelling than usually encountered in parallelism. Indeed, we have to take into account the GPU architecture described in section 3. The key idea is to capture in the model the decomposition of computations into threads, warps and blocks. One must also pay a particular attention to global memory accesses and to how the pipelines reduce the associated latency. Our *kernel* model is inspired by [8].

The first step is to estimate the work done by a single thread of a *kernel* in terms of cycles. Since the latency due to global memory accesses, is a non-blocking operation, we will count these accesses separately from arithmetic instructions. In CUDA, each arithmetic instruction requires a different number of cycles. For example, a single precision add costs 4 cycles while a single precision division costs 36 cycles. The work of a thread in terms of arithmetic instructions is then obtained by summing the costs of every arithmetic instruction.

The number of cycles involved in a global memory access, amounts to a 4 cycle instruction (read/write) followed by a non-blocking latency of 400-600 cycles. To correctly estimate the work due to such accesses, one needs to sum only the latency that is not hidden. Two consecutive read instructions executed by a thread, would cost twice the 4 cycles, but only once the latency due to its non-blocking behaviour. Once the amount of cycles involved in these memory accesses has been determined, it is then necessary to take into account the latency hidden by the scheduler (*warp swapping*).

The total work C_T done by a thread can be defined either as the sum or as the maximum of the memory access cycles and the arithmetic instructions cycles. Summing both types of cycles means we consider that latency can not be used to hide arithmetic instructions. This could occur for example when the latency is entirely hidden by the pipeline. The maximum variant represents the opposite situation where arithmetic instructions and memory accesses are concurrent. Then only the biggest of the two represents the total work of a thread.

Now that we know how much work is performed by a thread, we have to find out how many threads are run by a processor. A *kernel* decomposes into blocks of threads. Thus each *streaming multiprocessor* (SM) has to execute N_B blocks. Each of these blocks has itself N_W warps consisting of N_T threads. An SM has N_P processors which execute batches of threads in a pipeline of depth D. Thus the total work C_{Proc} done by a processor is given by

$$C_{Proc} = N_B \cdot N_W \cdot N_T \cdot C_T \cdot \frac{1}{N_P \cdot D} \tag{3}$$

Finally, the execution time T_K of a *kernel* is simply obtained by dividing the total work of a processor C_{Proc} by its frequency.

5.2 A *kernel* example : *Steepest Edge*

The selection of the entering variable is done by the *steepest edge* method. This method requires the computation of the euclidean norm of each column η_j of the constraint matrix. The coefficients c_j of the variables in the objective function, are then divided by the norm $\|\eta_j\|$. The selected variable is the one having the smallest *steepest edge* ratio:

$$se_l = \frac{c_l}{\|\eta_l\|} = \min_{j=1..n} \left(\frac{c_j}{\sqrt{1 + \sum_{i=1}^{m} x_{ij}^2}} \right) \tag{4}$$

The processing of a column is done in a single block. Each thread of a block has to compute $N_{el} = \frac{m}{N_W \cdot N_T}$ coefficients of the column. This first computation consists of N_{el} multiplications and additions. The resulting partial sum of squared variables must then be reduced on a single thread of the block. This requires $N_{red} = \log_2(N_W \cdot N_T)$ additions. Since the shared memory is used optimally, there are no added communications. Finally, the coefficient c_j must be divided by the result of the reduction.

Each block of the *kernel* computes $N_{col} = \frac{n}{N_B \cdot N_{SM}}$ columns, where N_{SM} is the number of SM per GPU. After processing a column, a block keeps only the minimum of its previously computed se_j ratios. Thus equation (5) below represents the sum of all arithmetic instruction cycles for a given thread, where c_{ins} denotes the number of cycles taken by instruction ins.

$$C_{Arithm} = N_{col} \cdot (N_{el} \cdot (C_{add} + C_{mul}) + N_{red} \cdot C_{add} + C_{div} + C_{cmp}) \tag{5}$$

Each thread has to load N_{el} variables to compute its partial sum of squared variables. The thread computing the division also loads the coefficient c_j. This must be done for the N_{col} columns that a block has to deal with. Thus the sum of cycles relative to memory accesses is defined by equation 6, where $C_{latency}$ denoting the cost in cycles of the latency. We have to remember that the scheduler hides some latency by swapping the warps and so the total latency must be divided by the number of warps N_W.

$$C_{Accesses} = \frac{N_{col} \cdot (N_{el} + 1) \cdot C_{latency}}{N_W} \tag{6}$$

At the end of the execution of this *kernel*, each block stores in global memory its local minimum se_l. The CPU will then have to retrieve the $N_B \times N_{SM}$ local minimums and reduce them. It is then profitable to minimize the number N_B of blocks per SM. With a maximum of two blocks per SM, the cost of this final operation done by the CPU can be neglected when m and n are big.

It now remains to either maximize or sum C_{Arithm} and $C_{Accesses}$ to obtain C_T. Equation (3) can then be applied and its result divided by the frequency of a processor to get the time $T_{KSteepestEdge}$ of the *steepest edge kernel*.

5.3 Single GPU implementation model

As previously seen in section 4.1, the single GPU implementation requires only a few communications between the CPU and the GPU. Since each of these communications are constant and small, they will be neglected in the model. For the sake of simplicity, we will consider the second phase of the *two-phase simplex* where we apply iteratively the three main operations: selecting the entering variable, choosing the leaving variable and pivoting. Each of these operations is computed as a *kernel*. The time for an iteration $T_{iteration}$ then amounts to the sum of all three *kernel* times:

$$T_{iteration} = T_{KSteepestEdge} + T_{KExpand} + T_{KPivoting} \tag{7}$$

The times $T_{KExpand}$ and $T_{KPivoting}$ for the *expand* and pivoting *kernels* are obtained in a similar fashion as for the *steepest edge kernel* described in the previous section.

With the estimated time per iteration $T_{iteration}$, we can express the total time T_{prob} required to solve a problem

$$T_{prob} = T_{init} + r \cdot T_{iteration} \tag{8}$$

where r is the number of iterations necessary to solve a problem Previous research by Dantzig [9] asserts that r is proportional to $m \cdot log(n)$.

5.4 Multi-GPUs implementation model

The model for the multi-GPUs implementation is quite the same as the one for our single GPU implementation. Each GPU basically executes the same *kernels* than the ones defined in the previous section. An important difference is the size of the local matrix processed by a GPU. Local matrices are of size $\frac{m \cdot n}{k}$, where k is the number of GPU used. Another difference is that we have to take into account the communications between the GPUs.

As we have seen in section 4.2, GPUs have to share the entering variable computed locally in order to elect the same entering variable. Similarly the GPU

choosing the leaving variable has to share the *pivoting column* with the others. Thus we have communications depending on the number of equations n (also the size of a column), and the number of GPUs k. The time for an iteration is then defined by the equation (9) below where the sum of *kernels* time, T_{Kernel}, are added to the sum of communications time, T_{Share}.

$$T_{iteration} = \sum T_{Kernel} + \sum T_{Share} \tag{9}$$

It is possible to define the total time required to solve a problem by applying equation (8) as seen in the previous section.

6 Measurements

Here we show how our implementations compete against an existing solver. First, we discuss the tests that were performed and on what datasets. Then we take a look at the measurements and analyze them.

6.1 Tests

Existing solvers can classified into two categories : open source and proprietary. The solvers offering the best performances are proprietary : *gurubi* and *IBM CPLEX*. Among the well-known open source solvers, we can quote *lpsolve*, *GLPK* and *CLP*. The last one, *CLP*, has been selected for our tests because of its excellent performances and its affiliation to the COIN-OR project. This project is a large scale project aiming to provide open source operational research tools. *CLP* is a sequential implementation of the revised simplex method.

Prior to comparing implementations, we had to ensure that our implementations were as functional as *CLP*. For that matter, we used a set of problems available on the *NETLIB* [10] repository. This dataset is usually used as a benchmark for LP solvers. It consists of a vast variety of real and specific problems for testing the stability and robustness of an implementation. Our implementations are able to solve all of these problems.

In order to test the performances of these implementations, a large range of problem size and density (percentage of non-null coefficients) is needed. As none of the existing datasets were offering the needed diversity of problems, we used a problem generator. It is then possible to create problems of specific size and density. Unfortunately, all the problems generated fall into the same category. Hence, generated problems of the same size and density will be solved within approximately the same number of iterations and the optimum of objective function will only slightly fluctuate.

Our test environment is composed of a CPU server (2 Intel Xeon X5570, 2.93GHz, with 24GB DDR3 RAM) and a GPU computing system (*NVIDIA tesla S1070*). This system connects 4 GPUs to the server. Each GPU has 4GB GDDR3 graphics memory, 30 streaming multiprocessors, each holding 8 processors (1.4GHz). Such a GPU offers at peak performance up to 1TFLOPS for single precision floating point, 80GFLOPS for double precision floating point and 100GB/s memory bandwidth between threads (registers) and global memory.

6.2 Results

We measured the execution time for each implementation on problems of different size with density below 5%. As shown in figure 2, for large enough problems ($m > 750$ and $n > 3750$) our implementations outperform substantially *CLP*. A time limit of 30 minutes was imposed which explains why there are no measurements for *CLP* above $m = 1250$. However on small problems *CLP* remains more efficient (see inset in fig. 2) This is mainly due to the initialisation time of a GPU which includes the time required to setup the *kernel* context.

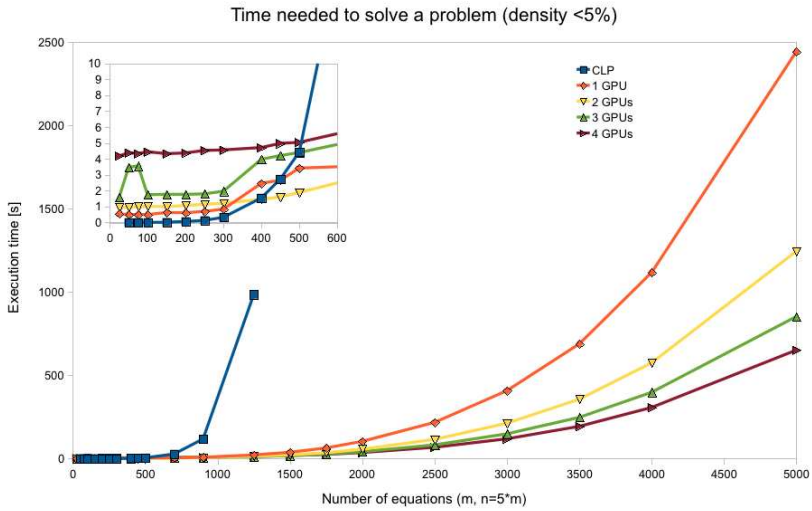

Fig. 2. Implementations execution time as a function of the problem size

Implementations of the revised simplex method, like *CLP*, are highly influenced by the problem density. The denser the problem, the slower the algorithm runs. Thus we mainly compared the performances of *CLP* against our implementations on problems of density lower than 5%. However, to take a deeper look into how these implementations behave under various densities, we measured the execution time on fixed size problems (500×2000) but with increasing density. The outcome was that the execution time of our standard simplex implementations remained constant, while the *CLP* execution time grew continuously. In this experiment *CLP* was outperformed as soon as the problem density rose above 5% (see fig. 3). Moreover, observing how the problem density evolved throughout a run, we noticed that in many cases the density increased significantly. Finally, we validated our performance models by showing that they accurately fit our measurements. The correlation between measurements and models for 1 to 4 GPUs was above 0.999. For big problems ($m > 5000$ and $n > 25000$), the speedup predicted by the model is nearly optimal in the range up to 40 GPUs.

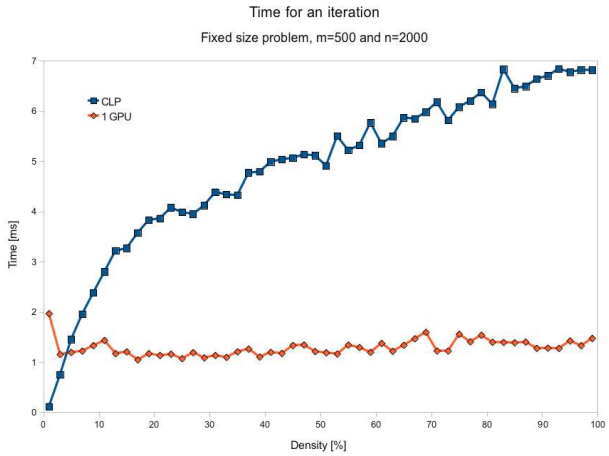

Fig. 3. Time of an iteration as a function of the problem density

7 Conclusion

In this paper, we presented robust and efficient single and multi GPU implementations of the standard simplex method. These implementations outperformed a well-known open source linear programming solver, *CLP*, on mid-size to big and/or dense problems. Moreover, we derived accurate performance models for these implementations. These models actually predict a nearly optimal speedup while using up to 40 GPUs to solve large problems.

As GPUs are constantly evolving, we can expect for such implementations an improvement of performances in the near future. The latest GPU generation of NVIDIA is up to 5 times faster on double precision instructions than the one used in this work. Finally, the interior point method could probably also benefit from the massively parallel architecture of GPUs.

References

1. V. Chvatal. *Linear Programming*. W.H. Freeman, 1983.
2. J.A.J. Hall. Towards a practical parallelisation of the simplex method, 2007.
3. M.E. Thomadakis and J.-C. Liu. An efficient steepest-edge simplex algorithm for SIMD computers, 1996.
4. J. Bieling et al. An efficient GPU implementation of the revised simplex method. In *Proc. of the 24th Int'l Parallel and Distributed Processing Symp.* IEEE, 2010.
5. A. Swietanowski. A new steepest edge approximation for the simplex method for linear programming. *Computat. Optimization and Appl.*, 10:271–281, 1998.
6. P.E. Gill et al. A pratical anti-cycling procedure for linearly constrained optimization. *Mathematical Programming*, 45:437–474, 1989.
7. NVIDIA. *CUDA Compute Unified Device Architecture : Programming Guide*, 2008.
8. K. Kothapalli et al. A performance prediction model for the CUDA GPGPU platform. Technical report, Int'l Inst. of Information Technology, Hyderabad, 2009.

9. G.B. Dantzig. Expected number of steps of the simplex method for a linear program with a convexity constraint. Technical report, Stanford University, 1980.
10. J. Dongarra and E. Grosse. The NETLIB repository. http://www.netlib.org, 2010.