Linear Optimization on Modern GPUs

Daniele G. Spampinato* and Anne C. Elster†
Department of Computer and Information Science
Norwegian University of Science and Technology
Trondheim, Norway
*Email: *daniele.spampinato@gmail.com, †elster@idi.ntnu.no*

Abstract

Optimization algorithms are becoming increasingly more important in many areas, such as finance and engineering. Typically, real problems involve several hundreds of variables, and are subject to as many constraints. Several methods have been developed trying to reduce the theoretical time complexity. Nevertheless, when problems exceed reasonable sizes they end up being very computationally intensive. Heterogeneous systems composed by coupling commodity CPUs and GPUs are becoming relatively cheap, highly performing systems. Recent developments of GPGPU technologies give even more powerful control over them.
In this paper, we show how we use a revised simplex algorithm for solving linear programming problems originally described by Dantzig for both our CPU and GPU implementations. Previously, this approach has showed not to scale beyond around 200 variables. However, by taking advantage of modern libraries such as ATLAS for matrix-matrix multiplication, and the NVIDIA CUDA programming library on recent GPUs, we show that we can scale to problem sizes up to at least 2000 variables in our experiments for both architectures. On the GPU, we also achieve an appreciable precision on large problems with thousands of variables and constraints while achieving between 2X and 2.5X speed-ups over the serial ATLAS-based CPU version. With further tuning of both the algorithm and its implementations, even better results should be achievable for both the CPU and GPU versions.

1. Introduction

Today parallel systems appear as the key answer to leap over the brick wall of serial performance [1]. With the commercial sector's demands for video and gaming, it was foreseen by Elster [2] and others that graphics processor development would lead to devices suitable for High Performance Computing (HPC). As a matter of fact, programming general purpose applications on a GPU (also known as GPGPU) is now one of the most discussed topics [3].

1.1. Linear Optimization

Linear optimization is a topic that has been methodically studied by operational researchers during the last 70 years. It is becoming a more and more important task in many different areas, such as finance and engineering. Both Khachian [4] and Karmarkar [5], proved that Linear Programming (LP) is in $\mathcal{P}$, finding a polynomial time algorithm for it. As Greenlaw et al. [6] shows, we can push the complexity analysis even further, placing LP in the class of P-complete problems. From empirical experiences, it seems that the two classes $\mathcal{P}$ and $\mathcal{NC}$ are likely not coinciding [6]. P-complete problems appear to be inherently sequential, meaning that they are feasible problems without any highly parallel algorithm for their solution.
This paper presents some results obtained implementing a parallel, GPU-supported version of a revised simplex method algorithm for solving LP problems [7]. To our knowledge, recent studies are still based on the old GPGPU programming methodology [8], [9] where the graphics pipeline is used to perform general purpose computation. These implementations require applications to be designed taking graphics aspects into account, and programmed using graphics APIs.
The $\mathcal{P}$-completeness of linear programming might discourage one from looking for parallel solutions. However, we decide to continue with our study mainly because we are convinced that at least some parts of the algorithm are suitable for parallelism. Also, since GPUs offload the CPU(s), GPUs may help solve large LP problems, as well as other problems reducible to them, more quickly.

1.2. Graphics Processing Units (GPUs)

Graphics hardware is now about 40 years old. It was initially realized to support activities such as computer-aided design (CAD) and flight simulations. A good description of the evolution of graphics hardware with different references to the literature can be found in Blythe [3]. Graphics Processing Units (GPUs) are affordable computing solutions for speeding up computationally demanding applications, offering performance peaks in the TFLOPs range. For instance, the recent NVIDIA S1070 1U computing server has four GPUs and a total of 960 processor cores enabling it to perform up to 4 TFLOPs in single precision. This allows new GPUs to support with great success several scientific fields [3], [10]–[12].

NVIDIA GPUs. NVIDIA is currently one of the world's leading GPU manufacturers. Their NVIDIA Compute Unified Device Architecture (CUDA) [13] provides developers with a high-level programming model that allows developers to take full advantage of the GPU's powerful hardware, enabling a larger productivity of solutions. It is available for NVIDIA GPU families based on the NVIDIA Tesla architecture. The NVIDIA Tesla architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SMs). A Tesla multiprocessor consists of eight Scalar Processor cores (SPs), two special function units, a multithreaded instruction unit, and on-chip memory. The SM creates, manages, and executes concurrent threads in hardware with zero scheduling overhead [13]. This is an important factor to allow very fine-grained decomposition of problems by assigning, for instance, one thread to each data element.
On a GPU we can localize two distinct kinds of memory, on-chip and device. Shared memory is one of the on-chip memory spaces which is shared by all the SPs of a SM. Access times to the shared memory are comparable to those of a L1-cache on a traditional CPU. The device memory is high-speed DRAM memory with higher latency than on-chip memory (typically hundreds of times slower). The device memory is subdivided in read-write, noncached (global and local) and read-only, cached (texture) areas. NVIDIA CUDA allows the developers to enrich their serial programs with calls to parallel kernels. Kernels' code is executed by a set of threads mapped onto the GPU's SPs. CUDA expresses task and data parallelism through the threads hierarchy. Threads are grouped in 1D, 2D, or 3D blocks which are organized in 1D or 2D grids. This organization extends to multidimensional structures.

2. Related Work

Two different attempts to map linear programming to a GPU before the advent of NVIDIA CUDA can be found in [8] and [9]. They both develop their solutions using Cg and OpenGL. Greeff [8] shows how GPU hardware can be used to solve linear programming problems using the revised simplex method presented in Sec. 3.1. The approach, which has partially inspired our work, is mainly constrained by limitations such as limited hardware capabilities and a hard-to-approach programming model. Jung and O'Leary [9] work on the same problem but with different tools, presenting a LP solver based on interior point methods [14], [15]. They use the GPU for some linear algebra intensive tasks like matrix assembly, Cholesky factorization, and forward and backward substitution. Like the solution proposed by Greeff, their solutions also suffer from old GPGPU restrictions. At the time of our development we were not aware of the work of Kipfer [16]. He describes a parallel implementation of the Lemke's algorithm for collision detection using NVIDIA CUDA. Lemke's algorithm is used to solve Linear Complementarity Problems (LCP) [17]. LCP is a special case of quadratic programming, which can be seen as a broader definition of the linear one where the objective function has quadratic order. Kipfer does not mention wether his solution can efficiently deal with linear problems, considering them special cases of LCP. However, given the singular importance of linear programming, we feel specific solutions must be designed to cope explicitly with LP. Kipfer's results can also be viewed as one more reason to believe that recent GPUs coupled with a high level programming model, are attractive tools for large optimization problems.

3. Linear Programming

Linear Programming optimizes a linear objective function fulfilling a specified set of constraints. Bertsimas and Tsitsiklis [18] is considered a good reference by many operational researchers. The terminology used in this paper is in accordance with [7].
A linear programming problem is composed by the following fundamental elements:

- an objective function $c(\cdot) : \mathbb{R}^n \to \mathbb{R}$, satisfying the relations $c(\mathbf{0}) = 0$ and $c(\alpha\mathbf{x}+\beta\mathbf{y}) = \alpha c(\mathbf{x})+\beta c(\mathbf{y})$, $\forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, $\forall \alpha, \beta \in \mathbb{R}$ (i.e. linear);
- a finite set of m linear constraints, where every constraint is expressed like $a(\mathbf{x}) \bowtie b$, with $a(\cdot) : \mathbb{R}^n \to \mathbb{R}$, $\mathbf{x} \in \mathbb{R}^n$, $\bowtie \in \{\leq, =, \geq\}$, and $b \in \mathbb{R}$.

The main goal for a LP problem is to either maximize or minimize (i.e. optimize) the objective function. A LP problem is expressed in canonical form in the following way. Let $\mathbf{c}$ and $\mathbf{x}$ be vectors in $\mathbb{R}^n$, $\mathbf{b}$ a vector in $\mathbb{R}^m$, and $\mathbf{A}$ a matrix in $\mathbb{R}^{m\times n}$, such that

$$max\, \mathbf{cx}, \quad \mathbf{Ax} \leq \mathbf{b}, \quad \mathbf{x} \geq \mathbf{0}. \tag{1}$$

Other representations are also used. In general, passing from one representation to another is possible if we take into account the following set of equivalences:

$$\begin{aligned}
max\, \mathbf{cx} &\equiv -min\ -\mathbf{cx} \\
\sum_j a_{ij}x_j = b_i &\equiv \begin{cases} \sum_j a_{ij}x_j \leq b_i \\ \sum_j a_{ij}x_j \geq b_i \end{cases} \\
\sum_j a_{ij}x_j \geq b_i &\equiv \sum_j -a_{ij}x_j \leq -b_i \\
\sum_j a_{ij}x_j \geq b_i &\equiv \sum_j a_{ij}x_j - s_i = b_i, \quad s_i \geq 0 \\
\sum_j a_{ij}x_j \leq b_i &\equiv \sum_j a_{ij}x_j + s_i = b_i, \quad s_i \geq 0.
\end{aligned}$$

The term s_i is called slack variable, since it provides the right value to fill in the gap between the left- and the right-side of a constraint.
For the purpose of our work, we focus on a second possible form, which can be easily derived from the canonical form using slack variables to augment the formulation:

$$max\, \mathbf{cx}, \quad \mathbf{Ax} = \mathbf{b}, \quad \mathbf{x} \geq \mathbf{0}. \tag{2}$$

This augmented canonical form turns out useful when formulating a numerical solution to the problem, since it allows to manipulate linear transformations instead of dealing with a set of inequalities.

3.1. Simplex-Based Methods

The simplex method, originally developed by G. B. Dantzig, was the first practically implemented method for solving LP problems.
The simplex method is an iterative method that traversing the faces of the feasible region, proceeds stepwise towards the optimal solution increasing the value of the objective function at each step. If LP solutions exist, they lie on vertices of the feasible region. The simplex algorithm focuses on three main operations applied on a tableau that summarizes the whole problem:

$\mathcal{O}1$ ***Determine the pivot column***
The algorithm requires to select the *biggest* positive value from the vector of the objective function coefficients.

$\mathcal{O}2$ ***Determine the pivot equation***
Divide the right side (b column) by the corresponding entries in the pivot column. Take as the pivot equation the one that provides the *smallest* ratio.

$\mathcal{O}3$ ***Elimination by row operation***
Determine a new tableau with zeros above and below the pivot. For example, we may use the Gauss-Jordan elimination technique.

The implementation of the original simplex method requires specific data structures to keep track of indexes and to update the tableau at each iteration.
Several methods were developed based on the same concepts as the simplex method. A revised matrix-based version of the original simplex method was developed by Dantzig and Orchard-Hays [19]. Based on the latter, other revised versions were studied. Morgan [20] presents and compare some important ones, like the Bartel-Golub's method, the Forrest-Tomlin's method, and the Reid's method.

3.1.1. The Revised Simplex Method. We present the algorithm defining a small dictionary in Tab. 1 that should help understanding the similarities with the classical simplex method. We define the problem in augmented canonical form, which provide us with a system of linear equations. A basis matrix, $\mathbf{B}$, consists of the columns of $\mathbf{A}$ corresponding to the basic variables, i.e. those variables considered part of a feasible solution. Note that $\mathbf{B}$ is a $m \times m$ squared matrix, since we introduce a slack variable for each constraint. The m nonzero variables in a basic solution, can be represented as a vector $\mathbf{x_B}$. Similarly, we denote the coefficients of the objective function corresponding to the basic variables with the vector $\mathbf{c_B}$.
Now, we saw that the simplex algorithm chooses the pivoting or entering variable by picking up the one that causes the greatest increase in the objective function. This is done by selecting the negative entry with the greatest magnitude in the objective row of the tableau. Even if the tableau, and in particular the objective row, is not explicitly represented, we can determine the entering variable based on the contribution of the non-basic variables. Such a contribution is estimated corresponding to each variable as $z_j - c_j = \mathbf{c_B B^{-1} A_j} - \mathbf{c_j}$. The variable corresponding to the negative difference with the greatest magnitude is the entering variable, say x_p. So, writing $\tilde{c}_j = \mathbf{c_B B^{-1} A_j} - \mathbf{c_j}$, we can say that $p = j \mid \tilde{c}_j = min_t \{\tilde{c}_t\}$, $\tilde{c}_j < 0$. If we cannot find any negative $\tilde{c}$, means that we have reached the optimum (no contribution can cause improvement). After having selected the entering variable, the simplex algorithm determines the leaving variable. To do this, the up-to-

Table 1. Normal and revised versions of the main simplex method's steps

Simplex Method	Revised Simplex Method
$\mathcal{O}1$: Determine the entering variable x_p based on the greatest contribution. If no better improvement is achievable, optimum found.	$\mathcal{O}1$: Select $x_p \,\lvert \tilde{c}_p = min_t \{\tilde{c}_t\}$, $\tilde{c}_p < 0$ If $\tilde{c}_p \geq 0$ optimum found.
$\mathcal{O}2$: determine the leaving variable x_q that provides smallest ratio between known terms and pivotal elements. If not possible, the problem is unbounded.	$\mathcal{O}2$: Compute $\mathbf{x_B} = \mathbf{B^{-1}b}$ Compute $\alpha = \mathbf{B^{-1}A_p}$ Select $x_q \mid \theta_q = min_t \{\theta_t\}$, $\alpha_q > 0$ If $\alpha \leq \mathbf{0}$ the problem is unbounded.
$\mathcal{O}3$: update the tableau.	$\mathcal{O}3$: Update $\mathbf{B}$.

date basic solution $\mathbf{x_B}$ is required at each step. Since all the variables outside the basis are set to zero, the system of equations can be written as $\mathbf{Bx_B} = \mathbf{b}$. From the latter we can easily compute $\mathbf{x_B} = \mathbf{B^{-1}b}$. Defining $\alpha = \mathbf{B^{-1}A_p}$, the leaving variable, say x_q, is the one with minimum θ-ratio, where $\theta_j = \mathbf{x_B}_j/\alpha_j$ and $q = j \mid \theta_j = min_t \{\theta_t\}$, $\alpha_j > 0$. If $\alpha \leq \mathbf{0}$, the solution is unbounded.
Finally, we have to update the basis represented by $\mathbf{B}$. In Sec. 4, the algorithm in Fig. 1 presents the revised simplex method.

4. Implementation Strategy

The algorithm in Fig. 1 shows a refined pseudocode with the program structure of both our CPU and GPU implementations. Our main data structures are matrices and arrays. The LP problem is represented in the same way for both versions. It consists of three main data structures: the constraints matrix, the costs array, and the known terms array. Where possible, loops have been replaced by algebraic operations on matrices. An example of this is the computation of the contributions, which in Algorithm 1 is substituted with the multiplication $\mathbf{e}[n] \leftarrow [1\ \mathbf{y}] \cdot [-\mathbf{c}; \mathbf{A}]$.
The routines can be categorized as algebraic and nonalgebraic. The first group of routines is implemented using BLAS [21] and NVIDIA CUBLAS [22], while the second one requires an ad-hoc set of functions. Tab. 2 summarizes how data and routines are implemented in both the ATLAS-based and the GPU-based versions. Our kernels mainly work on single elements of the matrices. Matrices are associated to 2D grids and split into regular submatrices, associating each submatrix to a 2D block. Blocks dimensions are multiples of the warp size.

```
**Require:** Matrix $\mathbf{A}$, vectors $\mathbf{b}$ and $\mathbf{c}$. Problem in canonical
  augmented form.
**Ensure:** Optimal solution or unbounded problem message

  procedure RSM($\mathbf{A}, \mathbf{b}, \mathbf{c}$)
      /* Initialize data (Range assignements with Matlab-like
  notation) */
      $\mathbf{B}[m][m] \leftarrow \mathbf{I}_m$
      $\mathbf{c_B}[m] \leftarrow \mathbf{c}[n-m : n-1]$
      $\mathbf{x_B}[m] \leftarrow \mathbf{0}$; $Optimum \leftarrow \bot$
      while $!Optimum$ do
          /* Determine the entering variable */
          $\mathbf{y}[m] \leftarrow \mathbf{c_B B^{-1}}$
          $\mathbf{e}[n] \leftarrow [1\ \mathbf{y}] \cdot [-\mathbf{c}\ ;\ \mathbf{A}]$
          Index $p \leftarrow \{j | \mathbf{e}_j == min_t\,(\mathbf{e}_t)\}$
          $\mathbf{x_B} \leftarrow \mathbf{B^{-1}b}$
          if $\mathbf{e_p} \geq 0$ then
              $Optimum \leftarrow \top$; BREAK
          end if
          /* Determine the leaving variable */
          $\alpha[m] \leftarrow \mathbf{B^{-1}A_p}$
          for $t \leftarrow 0,\ m-1$ do
              $\theta_t \leftarrow \alpha_t > 0\ ?\ \frac{\mathbf{x_B}_t}{\alpha_t}\ :\ \infty$
          end for
          Index $q \leftarrow \{j | \theta_j == min_t\,(\theta_t)\}$
          if $\alpha \leq \mathbf{0}$ then
              EXIT($"Problemunbounded"$); BREAK
          end if
          /* Update the basis */
          $\mathbf{E}[m][m] \leftarrow$ COMPUTEE($\alpha$, $q$)
          $\mathbf{B^{-1}} \leftarrow \mathbf{EB^{-1}}$
          /* Update basis cost */
          $\mathbf{c_B}_q \leftarrow \mathbf{c}_p$
          /* Update basis solution */
          $\mathbf{x_B} \leftarrow \mathbf{B^{-1}b}$
      end while
      if $Optimum$ then
          EXIT($\mathbf{x_B}$, $z \leftarrow \mathbf{x_B c_B}$)
      end if
  end procedure
```

Figure 1. The revised simplex method algorithm.

Table 2. Data and routines implementation in both the ATLAS-based and the GPU-based versions

Feature	ATLAS-based version	GPU-based version
Matrix/Vector	Array in central memory	Array in device memory
Algebraic routine	BLAS algebraic routine	CUBLAS algebraic routine
Nonalgebraic routine	Ad-hoc routine	Ad-hoc routine supported by or totally implemented as CUDA kernels

4.1. Non-Algebraic Routines

4.1.1. Computing Entering and Leaving Variable.

The entering variable is computed using two matrix-

matrix multiplications, an array movement to obtain non-basic variables contributions, and a minimum search to look for the minimum contribution. The leaving variable requires an array movement and a matrix-matrix multiplication to compute α, an array traversal to compute θ, and a minimum search to get the minimum θ element.
Our GPU-based versions are composed by NVIDIA CUBLAS calls, data movements in global memory, and minimum searches. The minimum search is implemented as a reduction kernel. Since the search space is passed through more than once, blocks load their portion of data in shared memory. Shared memory accesses are made with an odd stripe to avoid bank conflicts.

4.1.2. Computing $\mathbf{B^{-1}}$. Dantzig and Orchard-Hays [19] introduced a technique where $\mathbf{B^{-1}}$ is computed multiplying a matrix $\mathbf{E}$ by the actual value of $\mathbf{B^{-1}}$. $\mathbf{E}$ is a $m \times m$ matrix depending on entering and leaving variables. It is obtained starting from a $m \times m$ identity matrix. Let p and q be the entering and leaving index respectively. We apply the following substitution to the q^{th} column of $\mathbf{I}_m$:

$$\mathbf{E}_{iq} = \begin{cases} -\frac{\alpha_i}{\alpha_q} & i!=q, \\ \frac{1}{\alpha_q} & i = q. \end{cases}$$

5. Experimental Results

In this section, some of our results obtained running the LP solvers are presented. Our experiments are run on a CPU/GPU heterogeneous system. The CPU is a 64-bit Intel Core2 Quad (Q9550) 2.83 GHz, with 12 MB cache, and 1333 MHz bus. The GPU is an NVIDIA GeForce GTX 280. It has 240 streaming processor cores each working at 1.296 GHz. The card has a dedicated memory of 1GB connected by a 512-bit GDDR3 interface with a bandwidth of 141.7 GB/s. The central system and the GPU communicate through PCI-Express 2.0. The GPU is connected to a 16-lanes slot (x16) able to transfer up to 8GB/s. The operating system used is Ubuntu 8.04 with Linux kernel 2.6.24-22. Our CPU code is compiled and O3-optimized using gcc version 4.2.4. The system has a BLAS library optimized by ATLAS version 3.6.0. ATLAS is not configured to exploit the multiple cores of the CPU. The GPU software is developed with NVIDIA CUDA version 2.0.

5.1. Methodology

Our experiments involve 1000 different LP problems built upon random values. Problems size grows progressively. The largest problem tested has a 2000×4000 constraints matrix. Problems are built with the same number of constraints and variables. Hence, we will refer to the number of constraints to express the dimension of a problem. Additional slack variables are added to generate a problem in augmented canonical form. Time was measured with nanosecond precision using the *clock_gettime()* library call. Both the architectures are too numerically inaccurate to allow a direct comparison with zero. For this purpose, variables are compared with approximated values in the neighborhood of zero. With the CPU we used a tolerance with magnitude $\epsilon = 10^{-6}$, while with the GPU $\epsilon = 10^{-5}$. This difference is due to the incompatibility of the GPU used with the IEEE-754 standard for single precision values.

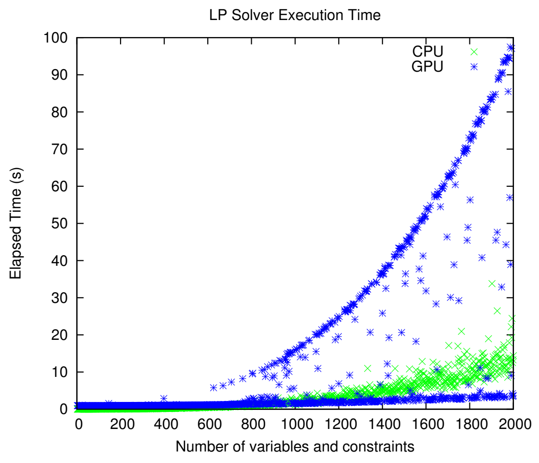

Figure 2. Elapsed time for the two versions of the LP solver.

5.2. Results and Analysis

Fig. 2 shows the elapsed time for the execution of both versions of the LP solver. It shows a clear trend during the execution of matrices with less then 900 constraints. After that point it starts alternating between good and very bad performance. Looking closer at the execution output, we found that the effect was due to a lack of precision. For some LP problems with a near-zero optimal value, the GPU could not conclude the computation, protracting it (and creating huge delays) until a *NaN* result.
In general, optimal values retrieved by the GPU appeared to be identical to those found by the CPU until the fifth or the sixth decimal digit. The ATLAS-based solver performed better for problems with less than 900 constraints, producing a gap of approximately 0.9s between the serial and the NVIDIA CUDA execution

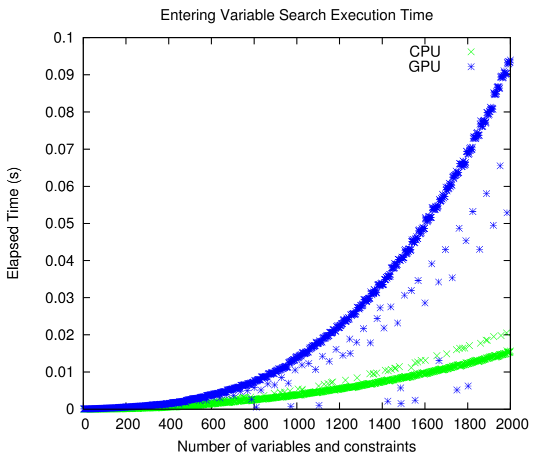

Figure 3. Time to search for the entering variable.

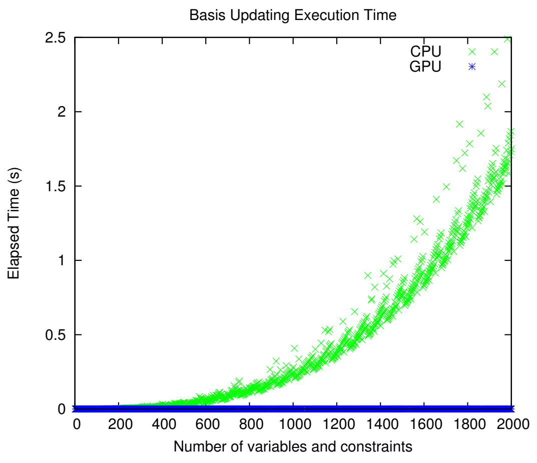

Figure 5. Time to update and inverse the basis.

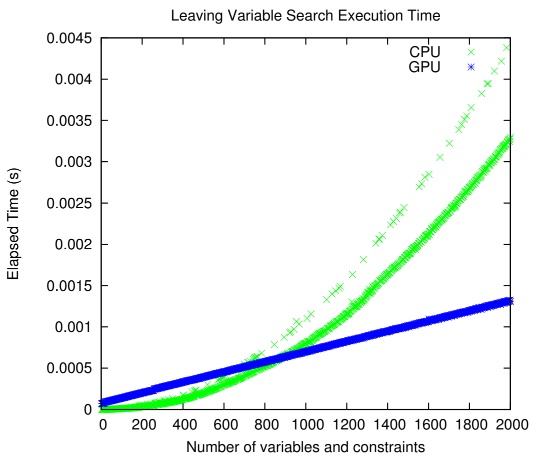

Figure 4. Time to search for the leaving variable.

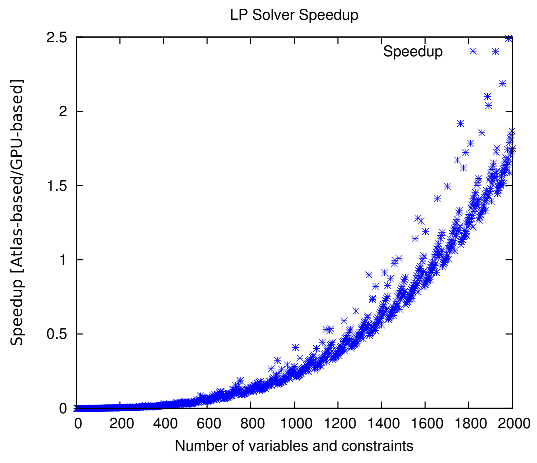

Figure 6. Overall speedup.

time. We acquired times for the three main task required by the revised algorithm, i.e. entering variable search (Fig. 3), leaving variable search (Fig. 4), and inverse basis updating (Fig. 5). Analysing the data, we noted that the entering value task performed almost always worse on the GPU within a range of 0.1s. The task is composed by an algebraic and a nonalgebraic subsets of instructions. Since the NVIDIA CUBLAS routines contribute just 0.004% of the total amount of time in the NVIDIA CUDA version, we conclude that the largest delay comes from the entering variable retrieval. The other two tasks gave better results, in particular basis updating which always required less than 10^{-4}s.

Nevertheless, the entering variable retrieval task cannot be responsible for the 1s gap between ATLAS and GPU version for small to medium-sized problems. Even summing up all the retrieved times for all the three program sub-parts we still end up with a 0.9s gap. Timing allocation, deallocation, and movement of data between central memory and device memory, we realized that those parts exactly represent the reason of the 0.9s remaining gap. Data was allocated in main memory using the *cudaMallocHost()* function in order to use pinned memory and set everything up for DMA transfers. With this optimization we saved time in transferring the data, but to set up the efficient transfer required a constant latency of 0.9s independently of the problem size. This latency is hence dominating for smaller problems.

5.2.1. Speedup Analysis. Fig. 6 shows the overall speedup, while Fig. 7, 8, 9 the local speedups for the three main sub-tasks analyzed so far. Fig. 6 shows a speedup curve growing faster and faster for bigger problems. It begins with a slow step up until a 0.5 factor around 1400 constraints. From that point the

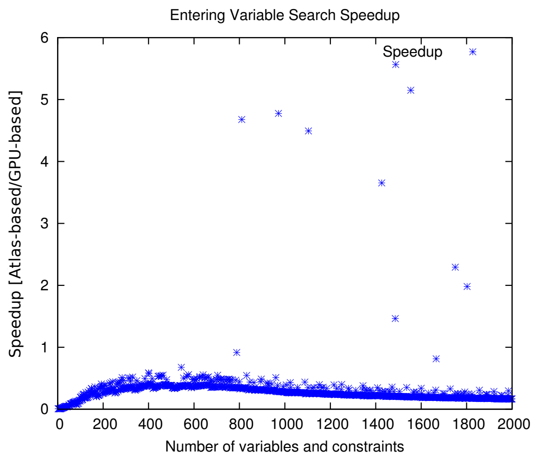

Figure 7. Local speedup for the entering variable search task.

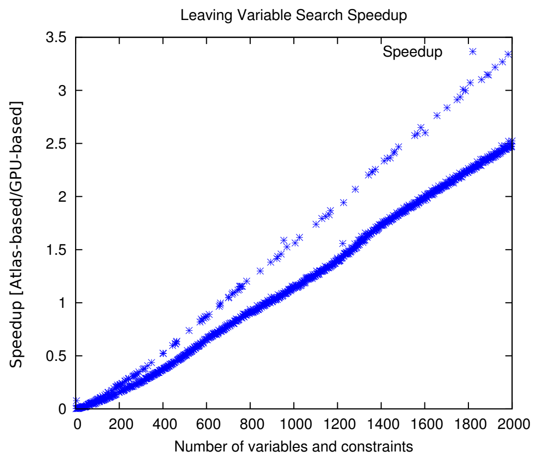

Figure 8. Local speedup for the leaving variable search task.

speedup factor grows quicker, reaching values between 2X and 2.5X around 2000 constraints. The NVIDIA CUDA version starts outperforming the serial one for problems with 1600-1800 constraints.

Notice that the GPU cannot be fully exploited due to the control flow. Many steps present data-dependency relations that can affect GPU performance for small problems. Moreover, there at least two main convergence steps where only a pair of values are transferred, i.e. where entering and leaving variables are found and their signs tested.

The basis updating task, in particular, exhibits a speedup curve that grows for increasing problem dimensions. It has to build the $\mathbf{E}$ matrix and to multiply it by $\mathbf{B^{-1}}$. The matrix-matrix multiplication can be done efficiently both by the serial BLAS and by CUBLAS. On the other hand, the embarassingly parallel task of

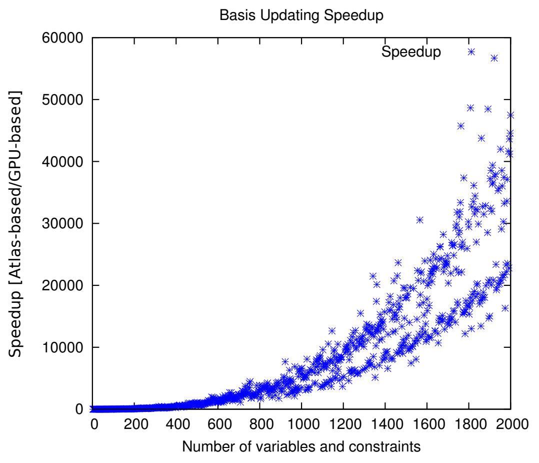

Figure 9. Local speedup for the basis updating task.

building $\mathbf{E}$ suits the GPU perfectly. We think that the super-linear speedup showed in Fig. 9 comes from caching difficulties on the CPU for huge matrices, together with the high-speed, non-cached memory contribution on the GPU. We are currently looking further into this issue.

6. Conclusions and Future Work

With LP's great demand for computing resources, it is an attractive application area targeting modern architectures. Previously, Greeff [8] reported that they were not able to solve problems with more than 200 variables with their GPU-based solution. In this paper, we showed that modern GPUs and CPUs with BLAS libraries facilitate linear programming problems with 2000 or more variables, covering the size of a lot of real world LP problems. Both our ATLAS-based and NVIDIA CUDA implementations were based on a common, execution-context independent model of the solving technique, i.e. we did not rewrite the algorithm so to favour a specific execution context.

The main advantage given by the NVIDIA CUDA programming model over previous ones like Cg, is that it allows programmers to directly concentrate on the problem decomposition, at the same time giving high-level control of the hardware capabilities. It is therefore worth noting that by using NVIDIA CUDA's environment, it took almost as much as time to write the CPU code implementing the ATLAS-based application, as it took us to write the GPU version. Discovering bugs and errors in a NVIDIA CUDA kernel is, however, challenging and required creating extra code to move partial computations back and forth to the device.

Our GPU CUDA based implementation performed between 2X and 2.5X better than our ATLAS-based CPU version for large problems. This is already significant considering LP $\mathcal{P}$-completeness. Another important finding is regarding precision. A lack of suffient precision when, for instance computing the comparisons that may anticipate the end of the computation, adversely affects performance.

6.1. Current and Future Work

We used different tolerances to approximate zero with near-zero floating point values. The CPU defeated the GPU from this point of view, being an order of magnitude more precise than the graphics unit. This was because the GPU used was not IEEE 754-compliant for single-precision values. We are currently also doing double precision tests. Note that our GPU implementation is based on the same algorithm used for the ATLAS-based CPU version. The fact that the use of the GPU speeds up the solution starting from certain problem sizes, should motivate a better redefinition of the approach, considering the heterogeneous execution context directly from the algorithm design stage. A redesign of the data structures for more efficient data transfer should also be taken into account. It would also be interesting to test implementations based on other recent GPGPU innovations, such as OpenCL [23]. Alternative solution methods such as interior point methods, should also be investigated.

Acknowledgment

The authors wishes to thank NVIDIA for their donations through their Professor Partnership Program.

References

[1] J. L. Manferdelli *et al.*, "Challenges and opportunities in many-core computing," *Proc. of the IEEE*, vol. 96, no. 5, pp. 808–815, May 2008.

[2] A. C. Elster, "High-performance computing: Past, present and future," in *PARA 2002, LNCS 2367*, J. Fagerholm *et al.*, Ed. Springer-Verlag, 2002, pp. 433–444.

[3] D. Blythe, "Rise of the graphics processor," *Proc. of the IEEE*, vol. 96, no. 5, pp. 761–778, May 2008.

[4] L. G. Khachian, "A polynomial time algorithm for linear programming," *Soviet Mathematics Doklady*, vol. 20, pp. 191–194, 1979, original version in *Doklady Akademii Nauk SSSR*, vol. 244, 5, pp. 1093-1096.

[5] N. Karmarkar, "A new polynomial-time algorithm for linear programming," *Combinatorica*, vol. 4, no. 4, pp. 373–395, 1984.

[6] R. Greenlaw *et al.*, *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995.

[7] D. G. Spampinato, "Linear programming with CUDA," Norwegian Univ. of Science and Technology, Tech. Rep., Jan. 2009.

[8] G. Greeff, "The revised simplex algorithm on a GPU," Univ. of Stellenbosch, Tech. Rep., Feb. 2005.

[9] J. H. Jung and D. P. O'Leary, "Implementing an interior point method for linear programs on a CPU-GPU system," *Electronic Transaction on Numerical Analysis*, vol. 28, pp. 174–189, 2008.

[10] J. D. Owens *et al.*, "GPU computing," *Proc. of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[11] R. Eidissen, "Comparing Cg and CUDA implementations of selected transform algorithms," Norwegian Univ. of Science and Technology, Tech. Rep., Jun. 2008.

[12] L. C. Larsen, "Utilizing GPUs on cluster computers," Norwegian Univ. of Science and Technology, Tech. Rep., 2006.

[13] *NVIDIA CUDA Programming Guide Version 2.0*, NVIDIA Corporation.

[14] S. Mehrotra, "On the implementation of a primal-dual interior point method," *SIAM Journal on Optimization*, vol. 2, pp. 575–601, 1992.

[15] S. J. Wright, *Primal-Dual Interior Point Methods*. Society for Industrial and Applied Mathematics, 1997.

[16] P. Kipfer, "LCP algorithms for collision detection using CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley, 2007, pp. 723–740.

[17] K. J. Murty, *Linear Complementarity, Linear and Non-linear Programming*. Heldermann Verlag, 1988.

[18] D. Bertsimas and J. Tsitsiklis, *Introduction to Linear Optimization*. Athena Scientific, 1997.

[19] G. B. Dantzig and W. Orchard-Hays, "Alternate algorithm for the revised simplex method: Using a product form of the inverse," *RAND*, Nov. 1953.

[20] S. S. Morgan, "A comparison of simplex method algorithms," Master's thesis, Univ. of Florida, Jan. 1997.

[21] "BLAS - basic linear algebra subprograms," http://www.netlib.org/blas/, last seen Jan. 2008.

[22] *CUDA - CUBLAS Library 2.0*, NVIDIA Corporation.

[23] "OpenCL," http://www.khronos.org/opencl/, last seen Jan. 2008.