DEGREE PROJECT, IN SCIENTIFIC COMPUTING , SECOND LEVEL
STOCKHOLM, SWEDEN 2015

GPU Predictor-Corrector Interior Poin Method for Large-Scale Linear Programming

DAVID RYDBERG

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCI SCHOOL OF ENGINEERING SCIENCES

GPU Predictor-Corrector Interior Point Method for Large-Scale Linear Programming

D A V I D   R Y D B E R G

Master’s Thesis in Scientific Computing (30 ECTS credits)
Master's Programme in Applied and Computational Mathematics
Royal Institute of Technology year 2015
Supervisor at TriOptima AB: Carl Thornberg
Supervisor at KTH: Michael Hanke
Examiner: Michael Hanke

TRITA-MAT-E 2015:37
ISRN-KTH/MAT/E--15/37--SE

Royal Institute of Technology
School of Engineering Sciences

KTH SCI
SE-100 44 Stockholm, Sweden

URL: www.kth.se/sci

Abstract

This master's thesis concerns the implementation of a GPU-accelerated version of Mehrotra's predictor-corrector interior point algorithm for large-scale linear programming (LP). The implementations are tested on LP problems arising in the financial industry, where there is high demand for faster LP solvers. The algorithm was implemented in C++, MATLAB and CUDA, using double precision for numerical stability.

A performance comparison showed that the algorithm can be accelerated from 2x to 6x using an Nvidia GTX Titan Black GPU compared to using only an Intel Xeon E5-2630v2 CPU. The amount of memory on the GPU restricts the size of problems that can be solved, but all tested problems that are small enough to fit on the GPU could be accelerated.

Referat

GPU-accelererad inrepunktsmetod för storskalig linjärprogrammering

Detta masterexamensarbete behandlar implementeringen av en grafikkortsaccelererad inrepunktsmetod av predictor-corrector-typ för storskalig linjärprogrammering (LP). Implementeringarna testas på LP-problem som uppkommer i finansbranschen, där det finns ett stort behov av allt snabbare LP-lösare. Algoritmen implementeras i C++, MATLAB och CUDA, och dubbelprecision används för numerisk stabilitet.

En prestandajämförelse visade att algoritmen kan accelereras 2x till 6x genom att använda ett Nvidia GTX Titan Black jämfört med att bara använda en Intel Xeon E5-2630v2. Mängden minne på grafikkortet begränsar problemstorleken, men alla testade problem som får plats i grafikkortsminnet kunde accelereras.

Acknowledgements

First, I would like to thank Michael Hanke for assuming the role of both supervisor and examiner, as well as for providing excellent academic guidance and support.

I am also very grateful to my TriOptima supervisor, Carl Thornberg, for giving me the chance to do my thesis at TriOptima. His tireless and selfless effort to make my thesis as good as possible has had a huge influence on my work.

Lastly I want to thank everyone in the TriOptima office, and the financial development team in particular, for making me feel welcome and included.

Contents

List of Figures

Chapter 1

Introduction

Linear Programming (LP), has been described as “without doubt the most natural mechanism for formulating a vast array of problems with modest effort.”[23, p. 2] Linear programming is a certain type of optimization problem with a linear objective function and linear constraints. It has been an important concept since the 1940s when the simplex method was developed by George B. Dantzig. LP is used in a variety of fields, such as engineering, statistics, logistics and finance. The reason for its popularity is partly because it is a natural way to formulate many real-world problems. Even for problems that are not actually linear, it is often easier to transform it into a linear form than to try to formulate a nonlinear objective function and constraints.

One of the most important innovations in the field of LP was Karmakar’s 1984 paper[21] describing an Interior Point Method (IPM), which is a nonlinear approach to the linear programming problem. It has proven polynomial worst time complexity, a significant theoretical improvement to the exponential time complexity that the simplex method has been shown to have for certain problems.[17, p. 2]. The IPM has been studied in depth and many improvements have been suggested. In 1992, Mehrotra [29] described a predictor-corrector approach which is now the basis of most existing interior point implementations [43, p. 14]. In a comparison of 15 open-source and commercial LP solvers [4], the predictor-corrector approach was found to be used by 11 solvers.

In certain applications the computational time it takes to solve an LP problem is critical and in order to push the limits of lower computation times further, new strategies have to be considered.

Using a Graphics Processing Unit (GPU) to solve mathematical problems allow professionals in a number of fields to perform large and complex calculations very quickly with consumer grade hardware. Compared to a general purpose Central

Processing Unit (CPU), a GPU typically contains hundreds of times as many cores, allowing for massively parallel computations. It is becoming more common for supercomputers to use GPUs in addition to CPUs, since GPUs often offer more performance per Watt.

This thesis aims to join the field of predictor-corrector interior point methods (PCIPM) with that of GPU computing. By accelerating the PCIPM with a GPU, we can potentially arrive at solvers faster than those available today.

1.1 Problem Statement

In this thesis two versions of the predictor-corrector method are implemented, one with GPU-acceleration and one without. Comparing the performance of the two versions allow us to answer the primary problem statement:

> *For what sizes of linear programming problems is it possible to use a GPU to speed up the predictor-corrector interior point method, compared to using only a CPU?*

This problem statement is investigated for a certain class of problems arising in financial applications. The implementations are optimized for sparse constraint matrices, with number of constraints and variables in the order of 10 000. The constraint matrix is not assumed to be of any specific structure. The specific problems for which the problem statement is investigated are described in detail in section 3.2.

The IPM investigated is based on Mehrotra's predictor-corrector method, described by Lustig, Marsten and Shanno[24] and Zhang[44].

The hardware on which the algorithm performance is investigated is an Intel Xeon E5-2630v2 with an Nvidia GTX Titan Black, based on the Kepler architecture. CUDA v.7 is used as the GPU programming framework. The test system is described in detail in section 3.4.

1.2 Purpose

Determining the feasibility of accelerating the calculations of Mehrotra's predictor-corrector interior point method (PCIPM) on a GPU has a number of purposes. The PCIPM is one of the most commonly used methods for solving the LP problem. Accelerating the method with a GPU for problems with large and sparse constraint matrices not assumed to have any specific structure is a field where not much study has been done. Therefore the result of this master's thesis would be of interest to anyone who is looking to accelerate their implementation of the PCIPM, and possibly for researchers in the optimization field.

Another interested party is the project provider TriOptima. They are a world-leading provider of financial services in the post-trade Over-The-Counter (OTC) derivatives market with customers including the world's largest banks and financial institutions. They use optimization algorithms in a variety of financial applications, and are interested in this thesis for use in an internal research project.

Note that the purpose of this thesis is **not** to include all the best mathematical optimizations to create a linear programming solver that is as fast as possible, or to create a practically useful software package. Rather, it is hoped that the results of this thesis can serve as a proof of concept and a basis for implementing a GPU accelerated version of the PCIPM, as well as be used as a basis for future research.

1.3 Outline

This report is organized into the following chapters:

1. The **Introduction** chapter contains an introduction of the subjects of Linear Programming and GPU computing, the exact problem statement and purpose of this thesis, and what is new and interesting about it.

2. The **Background** chapter contains more in-depth background information on the subject of interior point methods, specifically the predictor-corrector interior point method and how it works. There is also an explanation of the architecture of a GPU, and how to leverage it for fast computations. Finally, previous research is presented with a focus on the results of GPU-accelerated interior point methods.

3. The **Method** chapter contains a description of the benchmark problems and of the implemented CPU and GPU versions of the algorithm. There is also information about the hardware set-up that was used when doing the performance tests, and how they were done.

4. The **Results** chapter contains the main results of this thesis, including various computation time measurements and the memory consumption of the implemented solvers.

5. The **Discussion** chapter contains a discussion of the implementation as well as the results and their validity. It also contains an answer to the problem statement and some ideas for further research.

6. The **Conclusion** chapter contains in broad strokes the conclusions that can be drawn from the results of this thesis.

Chapter 2

Background

In this chapter, the mathematical foundation needed to understand the report is explained. The Predictor-Corrector Interior Point Method (PCIPM) is then studied so that an algorithm can be developed. An overview of GPU computing and CUDA is provided, as well as a compilation of previous research related to GPU-accelerated interior point methods.

2.1 Mathematical Foundation

The mathematical foundation needed to understand the PCIPM is presented in this section. Newton's method is the method that the IPMs are based on. The performance of the Cholesky factorization is crucial for the performance of the PCIPM, which is the reason why it is explained here.

2.1.1 Computer Linear Algebra

The foundation of all implementations of numerical methods are computer libraries performing linear algebra operations. These can be divided into two approaches, using either dense or sparse matrix representations. The dense representations store all values in the matrix, often arranged contiguously in the memory. Sparse matrix representations are slightly more involved, and are used when the matrices considered contain many zeros. Sparse matrix operations are generally memory bandwidth bound, instead of being bound by the speed of which arithmetic operations can be performed.[3] The *density* of a matrix is defined as the ratio between the number of non-zero elements (or *nnz*) and the total number of elements.

In order to efficiently store sparse matrices on a computer, it is advantageous to not store the zeros explicitly but instead only store the non-zero elements and their position. There are a number of different formats in which these kinds of matrices can be stored, including the **COO** (coordinate) format. It is perhaps the most straight-forward way of storing sparse matrices. Every non-zero element is stored with one element in each of three vectors containing: column, row and element value.

The **CSR** (compressed sparse row) format is almost like the COO format, except it has compressed the row coordinate vector. It now contains $m+1$ elements (for an $m \times n$ matrix), where the element i describe at which matrix element the row i begins. The $m+1$ element contains the number of non-zero matrix elements. The **CSC** (compressed sparse column) format works like the CSR format, but instead of compressing the row indices, it compresses the column indices. An example of the COO, CSR and CSC format can be seen in Figure 2.1. For a symmetric matrix $A = A^T$ it is worth noting that the CSR format can easily be converted to the CSC format by swapping the row and column vector. The CSR and CSC formats do not enforce that the vectors containing the indices and values are sorted with respect to increasing indices. However, sorted vectors give better performance for most applications.

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 5 & 0 & 4 \\ 3 & 6 & 2 \end{bmatrix}$$

	row	0	1	1	2	2	2
COO format	column	1	0	2	0	1	2
	value	1	5	4	3	6	2
	row	0	1	3	6		
CSR format	column	1	0	2	0	1	2
	value	1	5	4	3	6	2
	row	1	2	0	2	1	2
CSC format	column	0	2	4	6		
	value	5	3	1	6	4	2

Figure 2.1. COO, CSR and CSC sparse matrix formats explained with an example.

Another sparse matrix format is **HYB**, the hybrid matrix storage. It uses a special format for storing relatively dense sub blocks of the matrix and the COO format for irregular data that lies outside the dense sub blocks.

There are a number of computer libraries for linear algebra operations.

OpenBLAS is an optimized open-source implementation of the BLAS (Basic Linear Algebra Subprograms) library for dense matrices. BLAS is a set of functions performing most basic linear algebra operations, such as matrix multiplication and dot products. OpenBLAS supports parallel computations of these operations. Many higher-level libraries use BLAS to perform the most basic operations, and by compiling them with OpenBLAS they can take advantage of multicore processors. OpenBLAS also includes the LAPACK library. LAPACK is a library containing some more advanced linear algebra functions, such as solving linear systems and eigenvalue problems.

SuiteSparse is a library created by Timothy A. Davis, containing the module CHOLMOD which specializes in fast Cholesky factorizations of sparse matrices.[7] It uses a supernodal method, which works by dividing the matrix into a number of smaller dense matrices, which are solved with dense BLAS matrix operations. The supernodal method is suited for sparse matrices with relatively high density. There is also support for a number of reordering methods, decreasing the number of nonzeros in the resulting Cholesky factor. SuiteSparse can be called from MATLAB or C++. By compiling the library with OpenBLAS, the dense matrix operations become multi-threaded. There is also support for GPU-acceleration of these dense matrix operations.[9] CHOLMOD is one of the fastest sparse Cholesky libraries, especially for problems where many factorizations with identical sparsity patterns are performed.[16] It is used in many applications, including the built-in sparse Cholesky factorization in MATLAB. CHOLMOD does not support single precision calculations, only double precision.

Eigen is an open-source C++ library for linear algebra started by Benoît Jacob.[10] It is widely used, for example in the Google's Ceres solver, the Space Trajectory Analysis Project at the ESA, and the Multiprecision Toolbox for MATLAB. Eigen supports both dense and sparse linear algebra and is considered one of the fastest linear algebra libraries. The Eigen library has support for using CHOLMOD for the Cholesky factorization.

Blaze is another open-source C++ library which focuses on high performance and parallel execution of its operations.[5] The Blaze library was started by Klaus Iglberger of the University of Erlangen-Nuremberg. He has published two related articles including [18], where the blaze library is benchmarked favourably against libraries such as Eigen and Intel MKL. Blaze exploits the AVX instruction set of modern Intel x86 CPUs, can be compiled with OpenBLAS for parallel BLAS calls and it also performs its own parallelizations on a higher level than BLAS.

2.1.2 The Cholesky Factorization

The *Cholesky factorization* of a Symmetric Positive Definite (SPD) matrix $A \in \mathbb{R}^{m \times m}$ is the decomposition into

$$U^T U = A \tag{2.1}$$

where U is an upper triangular matrix with positive diagonal. An SPD matrix is symmetric and it holds that for every vector $x \neq 0$, $x^T A x > 0$. All SPD matrices have a unique Cholesky factorization. The time complexity of the Cholesky factorization for dense matrices is $\mathcal{O}(m^3)$. Factorization of SPD matrices is numerically stable without pivoting.[41]

The Cholesky factorization is mainly used to solve the linear system $Ax = b$. This is done by solving two triangular systems:

$$U^T y = b \tag{2.2}$$

$$Ux = y \tag{2.3}$$

Each of these triangular solutions requires $\mathcal{O}(m^2)$ operations (for dense matrices).

In the context of interior point methods, the Cholesky factorization is sometimes modified to allow for semi-definite matrices. Described in [44], the Cholesky-Infinity method is basically the same as the standard Cholesky, but it can handle semi-definite matrices as well, where it holds that for every vector $x \neq 0$, $x^T A x \geq 0$. When the matrix is found to be semi-definite (which would require division by a zero pivot element in the standard Cholesky method) the pivot element is set to infinity. This modification has been shown to increase the numerical stability when used in the context of interior point methods.[44]

There are many computer libraries dedicated to performing the Cholesky factorization, using different methods. Some libraries that perform the Cholesky factorization on sparse matrices are compared in [16], among others the CHOLMOD library which was measured to be one of the fastest.

The Cholesky factorization is often parallelized using a blocked version of the algorithm.[1] Instead of traversing the diagonal one element at a time as in the unblocked factorization, the algorithm traverses the diagonal in larger blocks. The operations used to process these blocks include matrix multiplication and subtraction, for which parallel implementations exist.

2.1.3 Newton's Method

Newton's method is a numerical scheme with local quadratic convergence used to find the roots of nonlinear functions or nonlinear systems of equations. It is an iterative method which requires the derivative of the function.[11, p. 74] For a scalar function $f(x)$, the iterations look like

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{2.4}$$

For a system of equations $F(x) = 0$ we use the Jacobian $J_F(x)$ instead of the derivative, giving us

$$\begin{aligned} x_{n+1} &= x_n - J_F^{-1}(x_n)F(x_n) \\ \implies x_{n+1} - x_n &= -J_F^{-1}(x_n)F(x_n) \\ \implies \Delta x &= -J_F^{-1}(x_n)F(x_n) \end{aligned} \tag{2.5}$$

where Δx is the step from one iteration to the next, using $x_{n+1} = x_n + \Delta x$. Of course in practice the inverse of the Jacobian is never calculated, instead the system

$$J_F(x_n)\Delta x = -F(x_n) \tag{2.6}$$

is solved using a linear system solver.

2.1.4 Linear Programming

Linear Programming (LP) is a type of optimization problem which can be written on the standard form

$$\begin{aligned} \text{Minimize} \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & x \geq 0 \end{aligned} \tag{2.7}$$

where $x \in \mathbb{R}^n$ is a column vector of the variables to be determined, $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ are given column vectors and $A \in \mathbb{R}^{m \times n}$ is a given matrix. The function $c^T x$ is called the objective function, and the matrix A is called the constraint matrix. It consists of m constraints over n variables.

A linear programming problem can always be expressed in this standard form. For constraints with inequalities instead of equalities, it is transformed into standard form with the introduction of so called *slack variables.* An example of such a transformation is

$$\begin{aligned} & 4x_1 - 5x_2 \leq 10, \quad x_i \geq 0, i = 1, 2 \\ \implies & 4x_1 - 5x_2 + x_3 = 10, \quad x_i \geq 0, i = 1, 2, 3 \end{aligned}$$

where x_3 is introduced as a slack variable.

It is known that the optimal solution of the linear programming problem lies on a constraint boundary.[23, p. 33] The simplex method works by searching for the optimal solution in the set of boundary vertices.

The linear programming problem is in the class of $\mathcal{P}$-complete problems. These problems are inherently difficult to parallelize effectively (assuming that $\mathcal{NC} \neq \mathcal{P}$, which is very likely but not proven).[40]

2.1.5 The Dual Problem

Associated with any primal problem on the form of Equation (2.7) is a dual problem. In the dual problem, the variable coefficients of the primal problem become the constraint coefficients, and the constraint coefficients of the primal problem become variable coefficients.[30, p. 145] Every constraint in the primal problem corresponds to a variable in the dual problem, and every variable in the primal problem corresponds to a constraint in the dual problem. The dual problem is written on the form

$$\begin{aligned} \text{Maximize} \quad & b^T y \\ \text{subject to} \quad & A^T y + z = c \\ & z \geq 0 \end{aligned} \tag{2.8}$$

where $y \in \mathbb{R}^m$ are the dual variables and $z \in \mathbb{R}^n$ are the slack variables for the dual problem.

The relationship between the objective functions of the primal and dual problem is expressed in the Weak Duality Lemma:

$$b^T y \leq c^T x$$

for all feasible vectors (x, y).[23, p. 83] This means that the primal and dual objective functions bound each other; the dual objective function gives a lower bound on the primal objective function, and the primal objective function gives an upper bound on the dual objective function. When the optimal solution $(x*, y*)$ is found, it holds that $b^T y* = c^T x*$, which can be used for determining the optimality of the solution.

One way to express the solution of both the primal and dual linear programming problem is by a specialization of the Karush-Kuhn-Tucker (KKT) conditions (formulation taken from [43, p. 4]):

The vector $x^* \in \mathbb{R}^n$ is a solution of Equation 2.7 if and only if there exist vectors $z^* \in \mathbb{R}^n$ and $y^* \in \mathbb{R}^m$ for which the following conditions hold for $(x, y, z) = (x^*, y^*, z^*)$:

$$\begin{aligned} A^T y + z &= c \\ Ax &= b \\ x_i z_i &= 0, \quad i = 1, ..., n \\ x &\geq 0 \\ z &\geq 0 \end{aligned} \tag{2.9}$$

Here, the y and z variables can be regarded as either the variables of the dual problem, or Lagrange multipliers for the conditions $Ax = b$ and $x \geq 0$ in the primal problem.

2.2 Interior Point Methods

Interior point methods (IPM) is a class of methods which finds a solution by iterating in the set of feasible points, i.e. the points which fulfil the conditions $Ax = b$ and $x \geq 0$. This is a nonlinear approach to a linear problem, and the IPMs can be regarded as search algorithms.

One class of IPMs is the primal-dual IPMs, which simultaneously solves the primal and dual problem. They solve the LP problem by applying Newton's method to the KKT conditions to find the solution to this system of equations. The resulting system (which is an application of Equation 2.6 on the KKT system 2.9) is

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ Z & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -Xz \end{bmatrix} \tag{2.10}$$

where X is the matrix with the components of x on the diagonal, and Z is the matrix with the components of z on the diagonal. The direction of the solution to this pure Newton system is called the *affine-scaling* direction. An iteration in the affine-scaling direction looks like

$$(x, y, z)^{k+1} = (x, y, z)^k + (\Delta x, \Delta y, \Delta z) \tag{2.11}$$

Without modifications, the Newton iterations would lead us to the infeasible set of solutions where $x_i < 0$ or $z_i < 0$ for some i, which is not allowed. In order to counter this problem the *central path* is introduced, which is an important concept in the field of IPMs. The central path is described by a modified version of the KKT conditions:

$$\begin{aligned} A^T y + z &= c \\ Ax &= b \\ x_i z_i &= \mu, \quad i = 1, ..., n, \quad \mu > 0 \\ x &\geq 0 \\ z &\geq 0 \end{aligned} \tag{2.12}$$

The central path is a path in the feasible region, which converges to the optimal solution as $\mu \to 0$. The basic idea of the IPMs is to iterate towards the solution of this system with decreasing μ.[23, p. 122]

If μ is chosen to be small initially, the resulting system will be highly ill-conditioned. If μ is chosen to be $\mu = x^T z/n$ instead, μ will decrease as the optimal solution is approached while ensuring that the iterations stay close to the central path, stay clear of the boundaries of the infeasible set and avoid ill-conditioning. However, the iterations will mostly approach the central path, and not the optimal solution.[43, p. 8] By scaling down μ using a centering parameter $\sigma \in (0, 1)$, the iterations will move in a cone-like fashion towards the optimal solution which is *on* the boundary, without stepping *over* the boundary in any iteration.[43, p. 10] Choosing the centering parameter σ, so that a balance is struck between moving towards the optimal solution and staying clear of the boundaries, is usually an important part of formulating an IPM. A system of equations expressing a Newton iteration of this new approach can be written as

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ Z & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -Xz + \sigma\mu \end{bmatrix} \tag{2.13}$$

The system of equations (2.13) is usually solved by eliminating Δz and Δx, yielding the normal equations[39][24]

$$(ADA^T)\Delta y = g \tag{2.14}$$

where $D = XZ^{-1}$ is a diagonal matrix and g is an expression involving the vectors and matrices given in the problem. When Δy is calculated, we can use it to calculate Δx and Δz as well.

The matrix $H = ADA^T$ is a symmetric positive definite (SPD) $m \times m$ matrix. The positivity condition holds because we require $(x, z) > 0$. Unfortunately, it is usually much less sparse than A [39] which increases the computation time of solving systems involving this matrix. H is also ill-conditioned, which makes many solvers unsuitable.

A full step is rarely taken, since it may violate the conditions $x_i \geq 0, z_i \geq 0$. Instead the new position can be expressed as

$$(x, y, z)^{k+1} = (x, y, z)^k + \alpha(\Delta x, \Delta y, \Delta z) \tag{2.15}$$

where $\alpha \in [0, 1]$ is the step length. Choosing a good value of α is an important part of an IPM. The most important criteria is that the step cannot violate the positivity conditions of x and z.

The theoretical complexity of IPMs has been determined to be $\mathcal{O}(n^3)$ in average. However, the practical performance is actually much better.[30, p. 290]

2.3 Predictor-Corrector Interior Point Method

The *Predictor-Corrector Interior Point Method* (PCIPM) is a type of primal-dual method, originally described by Sanjay Mehrotra[29], and it has since then been researched and developed extensively. The PCIPM has two features that stand out from other primal-dual solvers[43, p. 14].

The first feature is that the algorithm works in two steps: first a predictor step in the affine-scaling direction is calculated, then a corrector step is calculated which brings the algorithm closer to the central path. The PCIPM uses the same Cholesky decomposition of the normal equation (2.14) twice, in both the Predictor step and the Corrector step. This ensures that the computation time only increases by one back-substitution compared to a method that only solves one linear system with a centering parameter $0 < \sigma < 1$. Usually a PCIPM requires fewer iterations, which makes up for the slightly longer computation time per iteration.[24]. Before computing the Cholesky decomposition, a reordering method is commonly used in order to reduce the fill-in of the matrix. Since the non-zero structure does not change from iteration to iteration, the reordering method only has to be calculated once, and used for all later iterations. A commonly used algorithm is the Approximate Minimum Degree algorithm (AMD), which is a good heuristic ordering method for reducing the number of non-zeros in the Cholesky decomposition.[14]

The second feature is the centering parameter σ used in the Correction step is dynamically chosen from the results of the Prediction step. This allows the algorithm to do more centering when the Predictor step is small and less centering if the Predictor step is large. The algorithm also uses different step lengths α_P, α_D for the Primal and Dual problems, making it move more efficiently towards the optimal solution.

The algorithm used in this thesis is based on the methods developed by Lustig et al.[24] and the LIPSOL package for MATLAB. The LIPSOL package solves the linear programming problem using a variant of the PCIPM, and was presented and explained in an oft-cited report from 1996 by Yin Zhang.[44] The LIPSOL method allows infeasible points in the iterations. This means that the algorithm can be started with initial values that lies outside the bounds, and the iterates will move towards the feasible set of points. The initial values of x and z only has to be positive[43, p. 11], but the algorithm works better if the values are relatively large.[24] For methods that do not allow bound infeasibility, finding a starting point is a difficult problem having the same complexity as solving the optimization problem itself.[23, p. 135]

In the LIPSOL algorithm, the primal and dual forms of the problems are slightly changed from the standard form to allow upper bounds for the variables[44]

$$
\begin{array}{llcll}
\multicolumn{2}{c}{\textbf{Primal:}} & & \multicolumn{2}{c}{\textbf{Dual:}} \\
\text{Minimize} & c^T x & & \text{Maximize} & b^T y - u^T w \\
\text{subject to} & Ax = b & & \text{subject to} & A^T y + z - w = c \\
& x_u + s = u & & & z \geq 0 \\
& x \geq 0, \quad s \geq 0 & & & w \geq 0
\end{array}
\tag{2.16}
$$

where $x_u \in \mathbb{R}^{n_u}$ is the vector containing x-variables with upper bounds, $u \in \mathbb{R}^{n_u}$ is the vector with the upper bounds, $s \in \mathbb{R}^{n_u}$ is the slack variables for the upper bounds, and $w \in \mathbb{R}^{n_u}$ is the corresponding slack variables for the dual problem.

The KKT conditions for this form of the LP problem looks like

$$
F(v) = \begin{bmatrix} Ax - b \\ x_u + s - u \\ A^T y + z - w - c \\ Xz \\ Sw \end{bmatrix} = 0 \tag{2.17}
$$

where v is a vector with x, y, z, s, w. Here, the diagonal matrices with vectors x, z, s, w on the diagonal is denoted X, Z, S, W. This system is solved iteratively with a variant of Newton's method, until a stop criteria consisting of feasibility measures and duality gap has reached a specified tolerance. Every iteration can be considered to consist of 4 phases:

Phase 1, predictor direction. We solve the system

$$
J(v)\Delta_P v = -F(v) \tag{2.18}
$$

where J is the Jacobian of the KKT conditions F in (2.17) and $\Delta_P v$ is a vector with the predictor steps $\Delta_P x, \Delta_P y, \Delta_P z, \Delta_P s, \Delta_P w$. This is a pure Newton step in the affine-scaling direction, which does not take into consideration the centrality term. We do not explicitly use the Jacobian J, but instead use the normal equations

$$
\begin{aligned}
(ADA^T)\Delta_P y &= -(Ax - b + AD(A^T y + z - w - c)) \\
\Delta_P x &= D(A^T \Delta_P y + A^T y + z - w - c) \\
X\Delta_P z &= -(Z\Delta_P x + Xz) \\
\Delta_P s &= -(\Delta_P x + x + s - u) \\
S\Delta_P w &= -(W\Delta_P s + Sw)
\end{aligned}
\tag{2.19}
$$

to solve this system, where the diagonal matrix $D = (X^{-1}Z + S^{-1}W)^{-1}$.

Phase 2, centering parameter. After the prediction step is computed we choose μ and σ in accordance to [44], with

$$\begin{aligned} \mu &= \frac{g}{n+n_u} \\ \sigma &= \left(\frac{\hat{g}}{g}\right)^2 \\ g &= x^T z + s^T w \\ \hat{g} &= (x+\hat{\alpha}_P \Delta x)^T (z+\hat{\alpha}_D \Delta z) + (s+\hat{\alpha}_P \Delta s)^T (w+\hat{\alpha}_D \Delta w) \end{aligned} \tag{2.20}$$

The step lengths $\hat{\alpha}_P, \hat{\alpha}_D$ are chosen by ratio test, ensuring that the positivity conditions for the variables x, s, z, w would hold if the step $v + \hat{\alpha}\Delta_p v$ were taken. This ratio test is performed as

$$\begin{aligned} \hat{\alpha}_P &= \min\left(1, \min_{j:\Delta x_j<0} \frac{x_j}{-\Delta x_j}, \min_{j:\Delta s_j<0} \frac{s_j}{-\Delta s_j}\right) \\ \hat{\alpha}_D &= \min\left(1, \min_{j:\Delta z_j<0} \frac{z_j}{-\Delta z_j}, \min_{j:\Delta w_j<0} \frac{w_j}{-\Delta w_j}\right) \end{aligned} \tag{2.21}$$

If both $\hat{\alpha}_P$ and $\hat{\alpha}_D$ are larger than 1, we skip the corrector step (Phase 3) and just take a predictor step of length 1. For most iterations, this is not the case.

Phase 3, corrector direction. When μ has been approximated, we compute the corrector step by solving the system

$$J(v)\Delta_C v = -\begin{bmatrix} 0 \\ 0 \\ 0 \\ \Delta X_P \Delta Z_P - \sigma\mu \\ \Delta S_P \Delta W_P - \sigma\mu \end{bmatrix} \tag{2.22}$$

making the total step $\Delta v = \Delta_P v + \Delta_C v$. This system is solved using the same Cholesky factorization of the normal equations as is used in the Predictor direction phase.

Phase 4, taking the step. The step lengths α_P, α_D for the dual and primal variables are found by ratio test in the same way as in Equation (2.21), ensuring that the variables x, s, z, w stay positive after taking the step. The steps

$$\begin{aligned} x &= x + \alpha_P \Delta x \\ y &= y + \alpha_D \Delta y \\ z &= z + \alpha_D \Delta z \\ s &= s + \alpha_P \Delta s \\ w &= w + \alpha_D \Delta w \end{aligned} \tag{2.23}$$

are taken, concluding this iteration.

2.4 General-Purpose Computing on a GPU

The graphics processing unit (GPU) is the processing unit of the computer's graphics card. The GPU was developed for quickly being able to render complex 3D geometry and texture on the computer's screen, primarily for the purposes of computer games and 3D modeling. However, in recent years the GPU has increasingly been used for general computations as a massively parallel processing unit. The term for this is *GPGPU*, or General-Purpose computing on a GPU. Unlike the CPU's multi-core architecture which commonly has 2-16 cores, the GPU's many-core architecture can have thousands of cores all executing simultaneously. Individually, the GPU cores are slower than the CPU cores but due to the amount of cores, the theoretical number of floating point operations per second (FLOP/s) is substantially higher on GPUs.[36] The memory bandwidth of GPUs is usually higher than that of CPUs as well, making GPUs suitable for problems that are memory bandwidth bound.[31] The amount of memory even for high-end hardware is usually around 4-6 GB, which often implies a restriction of the problem sizes. For certain parallelizable algorithms, the GPU can be used to accelerate computations significantly. However, the GPU cannot be expected to be faster for all algorithms and computations due to a number of reasons:

- Some algorithms or parts of algorithms are inherently sequential and cannot be parallelized. Practically, even parallelizable problems do not scale perfectly with the number of cores due to overhead, synchronization and communication costs.
- GPUs are faster for data-parallel problems. Data-parallel means that a number of threads work on their own independent subset of some data, performing the same operations. For many algorithms, data-parallelism is not feasible.
- Practical aspects such as memory transfer between CPU/GPU and using the global memory within the GPU limit the amount of time spent on computations. For performance reasons, it is desired is to have a large proportion of computations to memory operations, so called *high arithmetic intensity.* To some extent, the GPU will try to automatically reduce the memory latency by executing threads which are not waiting for memory transfers.
- There is not as much caching and flow control on a GPU as on a CPU. Memory access delay can sometimes be masked by doing computations on other threads while waiting for the memory operations,[31] but this is not always possible to do.
- GPUs in general have a floating point performance which is much higher for single precision than double precision.[35] Some algorithms require double precision arithmetic for numerical stability, and the higher accuracy is sometimes required for certain applications.

CUDA

Historically when programming the GPU, it was necessary to use APIs that was designed with only graphics processing in mind. Now, there are a number of APIs that abstract away the graphical nature of the GPU, and expose a framework for general computations.[36] Compute Unified Device Architecture (CUDA) is a popular proprietary computing platform developed by Nvidia to be used with their graphics cards. There are a number of reasons for choosing to work with CUDA in this thesis. The Nvidia CUDA framework is highly optimized for Nvidia graphics cards. Both the framework and graphics drivers are developed by Nvidia, and the CUDA framework can take advantage of the latest hardware features of the graphics cards. Other frameworks, such as OpenCL, support many different types of graphics cards but are not as optimized to specific hardware. The code does not need to work on many different architectures and computers. The idea of this thesis is to provide insight in whether it is feasible to accelerate the predictor-corrector method using GPUs, not to provide a software library that can be used on any computer. CUDA also includes some convenient tools such as a profiler that can be used to assess the performance of the code, and there are many CUDA libraries containing mathematical operations that can be used in the algorithm.

In CUDA, there are bindings for languages such as C, C++ and Fortran.[34] A CUDA C program consists of *host* methods executed on the CPU and *device* methods executed on the GPU. The device methods are called *kernels*. They are written similarly to C functions, and all C operators and some common math functions are available. Unlike C functions, calling a kernel will start a specified number of threads all executing the same code. Creating CUDA threads incurs much less overhead compared to creating CPU threads. One common way to write CUDA code is to replace loops in the algorithm with the creation of as many threads as there are loop iterations. A code example of a vector addition in ordinary C code can be seen in Figure 2.2, which can be compared with a corresponding operation performed on a GPU with CUDA C in Figure 2.3. Some common operations such as memory allocation, memory copy and a kernel call can be seen in this figure.

```
int main()
{
    /* generate vectors a_host, b_host, c_host of length N... */

        for (int i = 0; i < N; i++)
                c_host[i] = a_host[i] + b_host[i]


    /* free relevant vectors... */
}
```

Figure 2.2. C code for the example program performing vector addition.

```
__global__ void SumArrays(float* a, float* b, float* c)
{
        /* find the ID of the current thread */
    int id = threadIdx.x;
    /* sum the element corresponding to the thread ID */
    c[id] = a[id] + b[id];
}

__host__ int main()
{
    /* generate vectors a_host, b_host, c_host of length N... */

        /* allocate vectors on GPU global memory */
    cudaMalloc((void**) &a_device, N);
    cudaMalloc((void**) &b_device, N);
    cudaMalloc((void**) &c_device, N);

        /* copy vectors from CPU to GPU */
    cudaMemcpy(a_device, a_host, N, cudaMemcpyHostToDevice);
    cudaMemcpy(b_device, b_host, N, cudaMemcpyHostToDevice);

    /* run kernel with one thread for every
          element and perform sum on GPU */
    SumArrays<<<1, N>>>(a_device, b_device, c_device);

    /* copy the vector back to the CPU*/
    cudaMemcpy(c_host, c_device, N, cudaMemcpyDeviceToHost);

    /* free relevant vectors... */
}
```

Figure 2.3. CUDA code for the example program performing vector addition. The vector is assumed to be smaller than the maximum block size.

The threads of a CUDA program are grouped into sets of 32 threads called *warps*. A warp will serially execute any conditional branch of its threads, which may lead to many threads idling.[31] A number of warps are further grouped into *blocks*, which are executed on hardware units called Streaming Multiprocessors (SM). A number of blocks are automatically scheduled on a certain SM, decided at runtime. If the GPU contains a large number of SMs, more blocks will execute in parallel than if the number of SMs was lower. Nvidia calls this *automatic scalability.*[31] The size of the blocks, and how to group the blocks into a *grid* is decided by the programmer during a kernel call. The maximum number of threads in a block is 1024, but there is no limit to the number of blocks in the grid. Each kernel will have access to a unique ID in the grid/block structure which is commonly used to decide which subset of the data it performs its operations on.

Choosing the optimal number of threads in a block is simplified by Nvidia's Occupancy Calculator. It takes into account the number of registers per thread, the number of threads per block and the amount of shared memory per block. It tells you the *occupancy* of each multiprocessor, which is the ratio of active warps to the total number of warps. For example, if a thread uses too many registers, the memory in the SM will run out. This is compensated by limiting the amount of active threads, at the same time limiting performance.

It is possible to execute more than one kernel simultaneously, using *streams*. Streams are created and destroyed by the programmer and each time a kernel is called, it is assigned to a specific stream. The kernels assigned to a specific stream execute in order, but there is no guarantee whether instructions in separate streams execute simultaneously or in any particular order. If no stream is specified in a kernel call, it will be assigned to the default stream.

CUDA Memory Types

The *global memory* is usually 1 GB or larger with high access times. The memory performance of the global memory largely depends on coalesced reads of data. If consecutive threads in a half-warp (16 threads) perform reads from consecutive memory addresses, it can be performed with a single transaction. Since the global memory is very slow compared to other memory types, access to it should be minimized. The global memory is the memory to which most data from the CPU is transferred. These transfers are relatively slow and can thus affect the performance of the program. Newer GPUs have the ability to transfer data to and from the CPU simultaneously as computations are performed. Also accessible from the CPU are the *constant memory* and the *texture memory* which are read-only from kernels. The texture memory has some special functions, such as the ability to linearly interpolate values when accessing a fractional memory address.

The *shared memory*, which is common to all threads within a block, has about 2 orders of magnitude faster access times than global memory. The shared memory resides inside the SM. Instead of reading the same data from the global memory multiple times for different threads in a block, shared memory should be used for caching. In fact, the shared memory and the L1 cache uses the same memory and it can be specified whether to allocate a larger part of the memory as shared memory or as L1 cache.

When declaring variables inside a kernel function, they are stored in the *per thread memory*. The physical storage of the per thread memory is in registers in the SM. If there is not enough free memory in the registers, a certain region of the global memory called the *local memory* will be used instead (this is called register spilling). The registers are approximately as fast as the shared memory.

In conclusion, there are a number of things to consider for optimal performance when writing a CUDA program. Since the global memory is comparatively slow access to it should be minimized. When accessed, it should be performed in a coalesced manner. This can sometimes be achieved using the shared memory as cache. Furthermore, data transfer between the host and the GPU memory is slow and should also be minimized. It can be accelerated up to 2.4x by copying to and from *pinned* host memory allocated with the function `cudaMallocHost`.[6] It is often a good idea to use the asynchronous memory copy function and perform computations simultaneously. When using local variables in the CUDA kernels, care should be taken so that register spilling does not occur since this will slow down access to the local variables considerably. Also, using too many registers in a kernel may reduce the number of warps that can execute on the SM.

CUDA Libraries

There exists a number of libraries built upon Nvidia CUDA. Some of them will be used in this thesis:

cuSPARSE is an Nvidia library[33] that performs sparse BLAS functions, such as sparse matrix-vector (SpMV) and sparse matrix-matrix (SpMM) multiplications. It is included in the CUDA Toolkit. Its functions include `csrmv` which solves the SpMV problem for matrices in CSR format and `csrgemm` which performs the SpMM multiplication for matrices in CSR format.

cuSOLVER is another Nvidia library[32] that is included in CUDA Toolkit, starting with version 7.0. This library contains, among other direct linear system solvers, a dense Cholesky factorizer.

SuiteSparse is a library that was described in section 2.1.1. It can be compiled with the Nvidia CUDA C compiler, which enables GPU-acceleration.

MAGMA is a library for dense linear algebra with a LAPACK like syntax. It aims to utilize both the CPU and the GPU for computations, which include the Cholesky factorization.[25] The MAGMA library is developed at the University of Tennessee. The library has been successfully used in GPU-acceleration of the PCIPM, see [13] and section 2.5.

2.5 Previous Research

There has been significant previous research in the field of accelerating linear programming with GPUs, specifically Interior Point Methods.

Smith, Gondzio and Hall at the University of Edinburgh showed in a report[39] from 2011 that the matrix-free IPM[15] could be accelerated using a GPU. They achieved a speedup for several classes of linear programming problems, including some problems with sparse constraint matrices of size 16000×16000 to 256000×256000. The speedup was approximately the same as the one that could be achieved using parallel algorithms on an 8-core CPU.

Jung and O'Leary describe in an article from 2008[20] an implementation of a variant of the PCIPM on a CPU-GPU system. They used the GPU for tasks such as Cholesky factorization. However, on tasks such as solving the linear system, they achieved better results using only the CPU. For small (sizes up to 516×758) and dense problems they found that the GPU did not provide any advantage to a CPU solver, the reason of which was attributed to the high cost of transferring data between the CPU and GPU. For dense random problems with $m > 640, n = 4m$, they did however see a speedup with the GPU solver. The article ends on an optimistic note, saying that they "expect that GPUs will be an increasingly attractive tool for matrix computation in the future".

Gade-Nielsen, Jørgensen and Dammann [13] studied GPU-acceleration of the PCIPM for Model Predictive Control problems. They implemented a MATLAB CPU, a MATLAB GPU and a CUDA C version of the algorithm. All matrices were stored in a dense format for all versions. The MATLAB GPU version used the GPU for the Cholesky factorization and the matrix-matrix multiplication. In the CUDA version the GPU was used for all operations. For large and dense enough problems, they managed to achieve a speedup of approximately 2x with the MATLAB GPU version, and a speedup of approximately 6x with CUDA C (still compared to the MATLAB CPU version). MATLAB's built-in `linprog` function performed approximately as good as the MATLAB CPU version of their algorithm for dense problems, but when the problems were sparse enough `linprog` was as much as 3x faster than their CUDA GPU version.

Some more results are available in Gade-Nielsen's PhD thesis [12]. Here, in one of the MATLAB GPU versions, the constraint matrix is stored in a sparse format, and only the Cholesky factorization is performed on the GPU. This version achieved a speedup of up to 2x compared to a MATLAB CPU version also using sparse matrices. Compared to this MATLAB CPU version, the CUDA version with dense matrices achieved a speedup of up to 4x. This version imposes lower problem size restrictions than the other versions, due to the memory limits of the GPU and the requirements of storing the constraint matrix as dense.

In [42], Vuduc et al. discuss the viability of using GPU-acceleration for some problems, including sparse Cholesky factorization and iterative sparse linear system solvers. They argue that for these problems, well-tuned GPGPU code is roughly equivalent to well-tuned multi-core CPU code regarding performance. This is relevant to this thesis since both of these operations are major parts of the computational effort of the PCIPM.

In a presentation on a GPU Technology Conference, Rennich, Davis and Vandermersch[37] present their work on a sparse Cholesky factorization performed on the GPU. It was found that CHOLMOD using a combination of CPU and GPU was successful in accelerating the factorization for matrices with more than 1.5 million non-zero elements, compared to using only a CPU.

In the report [3] by N. Bell and M. Garland, it was found that the SpMV operation could be accelerated on the GPU, but that it largely depends on the storage format of the matrix. For unstructured sparse matrices, it was found that HYB generally was the fastest format. The CSR format compared favourably to other formats on matrices with large row sizes. They also note that CSR is commonly used for the SpMM multiplication as well.

S. Rose[38] has examined the SpMM multiplication with CSR format. He found that his GPU implementation is faster than the MATLAB CPU version of the algorithm, as long as the density of the matrix is high enough. For large matrices with a density of 10^{-2} the GPU code is faster, while for large matrices with a density of 10^{-4} MATLAB is faster.

Chapter 3

Method

In this chapter, the method with which the problem statement is investigated is presented. First, the background behind the benchmark problems is explained along with relevant numbers describing the problems. The implementations are described in detail for scientific repeatability and so that the reader is convinced that the various choices that have been made are well motivated and investigated, lending credibility to the subsequent performance comparison. This chapter also contains a description of the testing environment in which the performance comparison was made.

3.1 Choice of Method

The method chosen to be used for answering the problem statement is to implement two versions of the PCIPM: one using only a CPU and one that is accelerated by a GPU. These versions can then be compared and the result can be seen as an indication of the feasibility of using a GPU to accelerate an arbitrary implementation of the PCIPM. This method of investigating the problem statement was chosen for a number of reasons:

- By implementing the method from start to finish, an understanding is gained of what makes it work and how to best use a GPU to accelerate it.
- Timing the two versions can be done using the same libraries and methods.
- By implementing a minimal version of the algorithm without preprocessing and only the most basic optimizations, the result may be seen as more general and applicable to a larger set of existing LP solvers.

In order for this method to give reliable results, some criteria have to be fulfilled:

- The benchmark problems should be realistic LP problems that arise in some application where someone is interested in the solution.
- The optimization level of the CPU and GPU version should be as similar as possible. If one version is optimized to a larger degree than the other, the performance comparison will not give a fair result.
- All benchmarks should be performed in a way that gives consistent results and represents some real-world scenario in which the solvers could be used.

Whether these points are fulfilled in this thesis is discussed in the rest of the report, and in particular the rest of this chapter.

3.2 Benchmark Problems

The benchmark problems come from the project provider TriOptima. They are real problems which are solved in the company's internal work process. The structure of these problems are typical for linear programming in a financial context, where bilateral trades from a set of parties need to be constrained on several time-bucketed risk figures. The risk figures are derivatives calculated numerically by looking at the change in some financial attribute resulting from a small change in some risk factor. This risk figure calculation is performed on several time-buckets, i.e. future time intervals, which makes up the constraint matrix A.

The sizes of the benchmark problems can be seen in Table 3.1.

Name	Constraints	Variables	density A	nnz AA^T	density AA^T
`lp6x7`	5 825	6 540	0.013	12 000 063	0.35
`lp7x7`	6 821	7 296	0.005	11 395 721	0.24
`lp10x10`	9 500	9 974	0.004	28 588 426	0.32
`lp13x19`	12 669	19 351	0.016	45 639 661	0.28
`lp23x29`	23 003	29 391	0.008	103 409 439	0.20
`lp31x39`	31 334	39 079	0.004	128 429 628	0.13
`lp46x51`	46 458	51 149	0.002	267 669 484	0.12

Table 3.1. The benchmark problems used in this thesis. *nnz* refer to the number of non-zero elements in a matrix and the density is defined as nnz divided by the matrix size, which is relevant for choosing between using sparse and dense linear algebra.

For all the problems, the original constraints consist of inequalities which are converted into equalities by the introduction of slack variables. The slack variables are included in the number of variables in Table 3.1, but due to the way that they are introduced in the matrix A they do not contribute to the number of non-zeros in AA^T.

The problems are stored in a data format described in Appendix A. They are parsed and converted to standard form in the same way for the CPU and GPU versions of the code.

3.3 Implementations

The algorithm which was implemented is listed in pseudocode in Algorithm 1. It follows the LIPSOL implementation, adhering to the description of the algorithm in section 2.3. One reason for choosing LIPSOL is that it is described in detail in [44]. Another reason is that it is the same code which the MATLAB `linprog` interior point method is based upon, which increases its credibility. By basing the algorithm of this thesis on a standard implementation, the performance investigation will presumably be more relevant to current projects.

The initial point is found in the same way as in [24] and LIPSOL. The last step in each iteration consists of calculating a stop criteria. The value of the stop criteria is a combination of the primal, dual and upper feasibility, as well as the duality gap. This value can be thought of as the accuracy of the current iteration, with a smaller value indicating a higher accuracy.

Some implementations of the PCIPM uses a method to increase the sparsity of the ADA^T matrix by separating the most dense columns of A into a matrix U and performing the Cholesky factorization on a matrix P without the most dense columns. After doing this, the Sherman-Morrison formula [2] can be used to solve the system. This method of increasing the sparsity of the Cholesky decomposition has not been implemented in this thesis. The reason for this is that for these benchmark problems, the relatively even density of the columns prevented any large gain in sparsity. For example, by removing the 500 most dense columns in the `lp13x19` problem, the density of ADA^T and P respectively was reduced from 0.28 to 0.20. This was not deemed a large enough reduction to motivate the increase in code complexity in the implemented algorithm.

Most commercial implementations of the PCIPM include some kind of preprocessing or presolve, which can have a number of effects on the actual computations performed. For example, MATLAB's `linprog` preprocessing[27] make sure that fixed variables are removed and that the constraint matrix has full structural rank, among other things. CPLEX is an example of a solver which goes further and can in many cases leverage redundancies to decrease the number of constraints and variables.[19] The goal of this thesis is to investigate whether the PCIPM can be accelerated by a GPU. Decreasing the size of the problem before running the algorithm does not

Algorithm 1 Predictor-Corrector Interior Point Method: detailed description

```
1:  $\tau_0 = 0.9995$
2:  Find initial values for $x, y, z, s, w$.
3:  while $stop > tol$ do
4:      $D = (X^{-1}Z + S^{-1}W)^{-1}$
5:      $H = ADA^T$
6:      Compute $U$ such that $U^TU = H$ by Cholesky factorization
7:      $r_b = Ax - b$                              ▷ Primal feasibility residual.
8:      $r_u = x + s - u$                           ▷ Upper bounds feasibility residual.
9:      $r_c = A^Ty + z - w - c$                    ▷ Dual feasibility residual.
10:     $r_{xz} = Xz$
11:     $r_{sw} = Sw$
12:     $r_{cp} = r_c - X^{-1}r_{xz} + S^{-1}(r_{sw} - W^{-1}r_u)$
13:     $\Delta y = U^{-1}(U^{-T}(-(r_b + ADr_{cp})))$
14:     $\Delta x = D(A^T\Delta y + r_{cp}$
15:     $\Delta z = -X^{-1}(Z\Delta x + r_{xz})$
16:     $\Delta s = -(\Delta x + r_u)$
17:     $\Delta w = -S^{-1}(W\Delta s + r_{sw})$
18:     Compute $\alpha_P$ and $\alpha_D$ such that $x+\alpha_P\Delta x, s+\alpha_P\Delta s, z+\alpha_D\Delta z, w+\alpha_D\Delta w > 0$
19:     if $\tau_0\alpha_P < 1$ or $\tau_0\alpha_D < 1$ then
20:         $g = x^Tz + s^Tw$
21:         $\hat{g} = (x+\min(1,\alpha_P)\Delta x)^T(z+\min(1,\alpha_D)\Delta z)+(s+\min(1,\alpha_P)\Delta s)^T(w+\min(1,\alpha_D)\Delta w)$
22:         $\mu = \hat{g}^2/g/(n + n_u)$
23:         $r_{xz} = \Delta X\Delta z - \mu$
24:         $r_{sw} = \Delta S\Delta w - \mu$
25:         $r_{cc} = -X^{-1}r_{xz} + S^{-1}r_{sw}$
26:         $\Delta_C y = U^{-1}(U^{-T}(-ADr_{cc}))$
27:         $\Delta_C x = D(A^T\Delta_C y + r_{cc})$
28:         $\Delta_C z = -X^{-1}(Z\Delta_C x + r_{xz})$
29:         $\Delta_C s = -\Delta_C x$
30:         $\Delta_C w = -S^{-1}(W\Delta_C s + r_{sw})$
31:         $\Delta v = \Delta v + \Delta_C v$          ▷ Containing variables $\Delta x, \Delta y, \Delta z, \Delta s, \Delta w$.
32:         Compute $\alpha_P$ and $\alpha_D$ such that $x + \alpha_P\Delta x, s + \alpha_P\Delta s, z + \alpha_D\Delta z, w + \alpha_D\Delta w > 0$
33:     end if
34:     $x = x + \min(1, \tau_0\alpha_P)\Delta x$
35:     $y = y + \min(1, \tau_0\alpha_D)\Delta y$
36:     $z = z + \min(1, \tau_0\alpha_D)\Delta z$
37:     $s = s + \min(1, \tau_0\alpha_P)\Delta s$
38:     $w = w + \min(1, \tau_0\alpha_D)\Delta w$
39:     $stop = \frac{||r_b||}{\max(1,||b||)} + \frac{||r_u||}{\max(1,||u||)} + \frac{||r_c||}{\max(1,||c||)} + \frac{|c^Tx-b^Ty+u^Tw|}{\max(1,|c^Tx|,|b^Ty-u^Tw|)}$
40: end while
```

help answering this question, but may actually make the results more difficult to interpret. Because of this, preprocessing is not implemented in the program of this thesis.

It is worth noting that generally the fastest way of solving linear systems of equations on the GPU is to use iterative methods, such as the PCG method. In the PCIPM this is not feasible, since one of the main points of the algorithm is to use the same Cholesky factorization twice. This means that two solves can be performed for practically the same cost as one solve. The Cholesky factorization is also more stable than iterative methods. Iterative methods in the context of IPMs require a regularization of the mathematical formulation as well as special preconditioning.[15] Leveraging the GPU's capabilities of using iterative solvers to solve linear programming has already been studied, most notably by Smith et al.[39] This thesis does not aim to replicate these results, but rather to investigate another method which is based on using a direct solver.

The specific structure of the benchmark problems will **not** be exploited in this thesis. The motivation for this is that the results of this thesis should be applicable to PCIPMs in general. Certainly, some things such as the density and structure of the matrices differ from one problem to another and it may be the case that the structure of these benchmark problems gives different performance results than other problems would.

A compressed description of the main algorithm (seen in Algorithm 1) is listed in Algorithm 2. This version will be used to explain the GPU implementations in section 3.3.3.

3.3.1 Parallelization

Parallelizing the PCIPM is a difficult problem. Since all IPMs are iterative and the iterations must be taken one at a time, these methods are inherently sequential.

It is possible to parallelize some of the separate operations of each iteration. There are two operations that make up most of the computation time: the sparse matrix-matrix (SpMM) multiplication used to compute ADA^T, and the Cholesky factorization of the same matrix, corresponding to line 5 and 6 in Algorithm 2. For dense matrices, the Cholesky factorization use $\mathcal{O}(m^3)$ operations and the matrix-matrix multiplication of ADA^T use $\mathcal{O}(m^2 n)$ operations, where m is the number of constraints and n is the number of variables. These operations are hence candidates for parallelization. All other operations, apart from for Cholesky factorization and SpMM, have a time complexity of $\mathcal{O}(n^2)$ or less making them practically irrelevant for the overall time consumption of the solver.

Unfortunately, these two most costly operations are dependent on one another:

Algorithm 2 Predictor-Corrector Interior Point Method: compressed description

```
 1: $\tau_0 = 0.9995$
 2: Find initial values for $x, y, z, s, w$.
 3: while $stop > tol$ do
 4:     $D = (X^{-1}Z + S^{-1}W)^{-1}$
 5:     $H = ADA^T$
 6:     Compute $U$ such that $U^TU = H$ by Cholesky factorization
 7:     Compute residuals $r_b$, $r_u$, $r_c$, $r_{xz}$, $r_{sw}$, $r_{cp}$.
 8:     Compute predictor direction $\Delta y$ using $U^TU\Delta y = rhs$.
 9:     Compute predictor direction $\Delta x, \Delta z, \Delta s, \Delta w$.
10:     Compute $\alpha_P$ and $\alpha_D$ such that $x+\alpha_P\Delta x, s+\alpha_P\Delta s, z+\alpha_D\Delta z, w+\alpha_D\Delta w > 0$
11:     if $\tau_0\alpha_P < 1$ or $\tau_0\alpha_D < 1$ then
12:         Compute centering parameter $\mu$.
13:         Compute residuals $r_{xz}$, $r_{sw}$, $r_{cc}$.
14:         Compute corrector direction $\Delta_C y$ using $U^TU\Delta_C y = rhs$.
15:         Compute corrector direction $\Delta_C x, \Delta_C z, \Delta_C s, \Delta_C w$.
16:         $\Delta v = \Delta v + \Delta_C v$
17:         Compute $\alpha_P$ and $\alpha_D$ such that $x + \alpha_P\Delta x, s + \alpha_P\Delta s, z + \alpha_D\Delta z, w + \alpha_D\Delta w > 0$
18:     end if
19:     Take steps in the $\Delta v$ direction.
20:     $stop = \frac{||r_b||}{\max(1,||b||)} + \frac{||r_u||}{\max(1,||u||)} + \frac{||r_c||}{\max(1,||c||)} + \frac{|c^Tx-b^Ty+u^Tw|}{\max(1,|c^Tx|,|b^Ty-u^Tw|)}$
21: end while
```

the Cholesky factorization uses the result of the SpMM multiplication, and the SpMM multiplication uses data that are computed using the previous Cholesky factorization. This means that these two operations cannot be run simultaneously.

Much research has gone into developing fast methods for Cholesky factorization and sparse matrix-matrix multiplication, since these are two very common operations. This includes versions that take advantage of GPUs. Because of this, developing faster methods for performing these operations is deemed out of the scope of this thesis. Instead, existing research and software libraries are leveraged when the PCIPM is implemented for two different architectures: the multi-core architecture of the CPU and the many-core architecture of the GPU. Details about these versions are described in the following two subsections.

3.3.2 CPU Implementation

In order to have something to compare the GPU-accelerated PCIPM to, it is important to implement a well-optimized CPU version which is mathematically equivalent to the GPU version. The CPU versions also serve as a basis for the GPU implementations, since the same mathematical steps are followed.

Two CPU versions of the algorithm were implemented. They both perform the same operations, but make use of different algebra libraries and data representations. Both versions use the multi-threading capabilities of the CPU. Both versions store the constraint matrix A in some compressed sparse way (CSR or CSC), and all operations on this matrix are hence operations optimized for sparse matrices. All vectors are stored in a dense way, since they are comparatively full of non-zeros. Double precision is used since it is required for the benchmark problems to prevent the ADA^T matrix from becoming too ill-conditioned and only positive semi-definite (instead of the required positive definite).

CPUDense

The version `CPUDense` is a MATLAB implementation of the algorithm. MATLAB uses the Intel MKL library for the BLAS functions, which is one of the fastest BLAS libraries. In this version, the matrix ADA^T is converted into a dense representation before the Cholesky factorization is performed, since that matrix is comparatively dense (approximately 25% non-zeros, more details in Table 3.1). This strategy was implemented and tested in MATLAB, the Armadillo C++ library and the Eigen C++ library. The MATLAB version uses multi-threading and was found to be the fastest and was selected to be used in the benchmarks. Unfortunately the SpMM multiplication does not use multi-threading, but the Cholesky factorization does.

The densities of the constraint matrices A are approximately 1%. Because of this, they are stored in a sparse way. A version where the constraint matrix is stored in a dense way was tested as well, the computation time result of which can be seen in Figure 3.1. The version with a dense A matrix performed much worse than the version where A is stored as a sparse matrix. The speedup achieved by using sparse matrices compared to dense matrices is 2x - 8x.

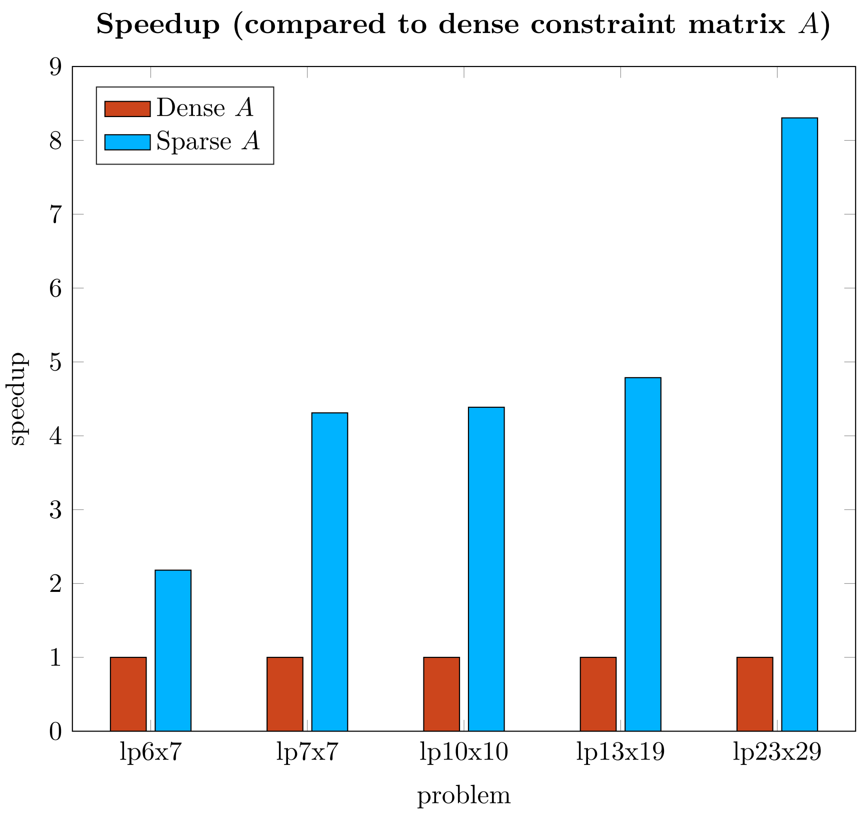


Figure 3.1. Speedup achieved by using sparse linear algebra compared to dense linear algebra for the constraint matrix A using solver `CPUDense`. A higher bar implies a faster solution time.

CPUSparse

The version `CPUSparse` is implemented in C++ and uses the library Eigen, described in section 2.1.1. This library is single-threaded, and the SpMV and SpMM multiplications are performed on only one core. However, it contains bindings for computing the Cholesky factorization using CHOLMOD which is multi-threaded. In the `CPUSparse` version the matrix ADA^T is kept in its sparse representation when performing the Cholesky factorization with CHOLMOD. Another way of implementing a CPU version with a sparse Cholesky factorization would have been to modify the MATLAB `CPUDense` version so that it uses a sparse Cholesky factorization. This was tested and found to be slightly slower than the version with Eigen and CHOLMOD.

A C++ library was not found in which the SpMM multiplication utilizes more than one core. A number of different libraries were tested for the computation of the SpMM multiplication, to find the fastest library and therefore achieve a fair comparison in the performance analysis. Three libraries were tested: Eigen, CHOLMOD and blaze. The Eigen library has a built-in SpMM multiplication, requiring no matrix format conversion and allowing the code to be as simple as possible. The CHOLMOD library can also be used for the SpMM multiplication and can be integrated into the Eigen library quite easily, since the data structures are similar in the two libraries. Lastly, the blaze library was tested. Unfortunately, this library uses another type of storage for the matrices which means that the matrix data has to be copied before and after the SpMM multiplication.

The result of the performance comparison of the SpMM multiplication can be seen in Figure 3.2. The three libraries perform similarly, with blaze being slightly quicker for some matrices and CHOLMOD being slightly quicker for other matrices. Since there is no large performance difference and the matrix format of CHOLMOD better matches that of Eigen, CHOLMOD was chosen to be used for the performance comparisons in section 4.

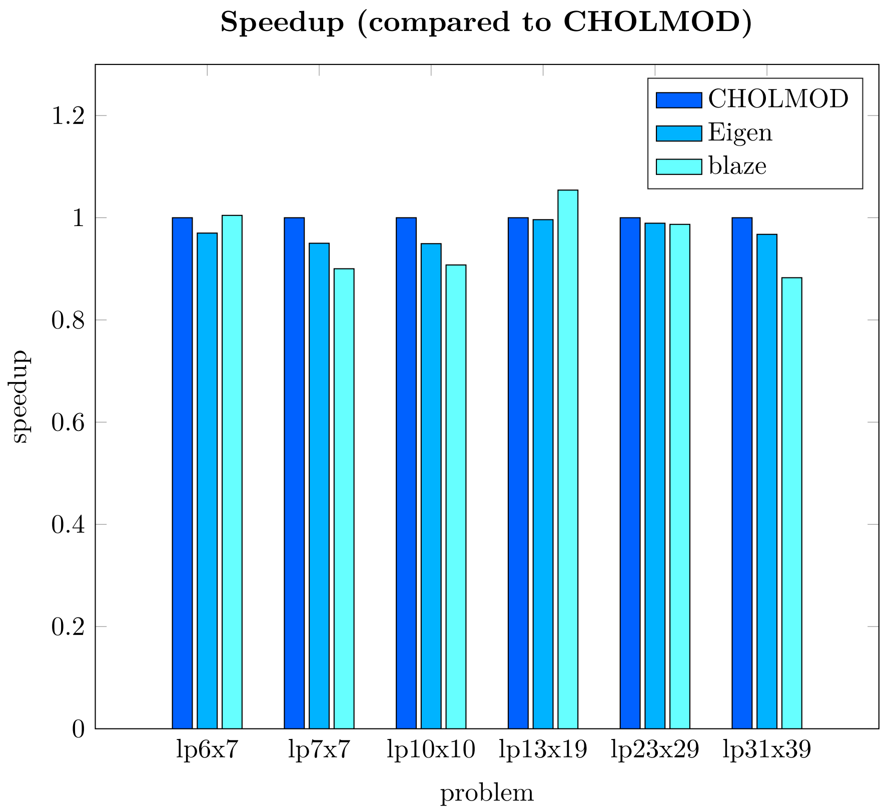


Figure 3.2. Comparing different SpMM libraries for the `CPUSparse` solver. Plot shows speedups of full solutions of the specified LP problems.

In order to reduce the number of non-zeros in the Cholesky factor, a few reordering algorithms are tested during the first factorization of every solution. The algorithms tested include the Approximate Minimum Degree algorithm. Finding a good reordering only has to be performed once in each solution, since the non-zero pattern of the matrix ADA^T does not change between iterations. The same reordering can thence be used for all subsequent factorizations.

3.3.3 GPU Implementation

The main result of the report is dependent on having a well-thought-out and optimized GPU version of the PCIPM. The GPU versions of the algorithm should perform the same mathematical calculations as the CPU versions, so that the performance comparisons become as fair as possible. But, while the mathematical optimizations should be equivalent, optimizing the computations becomes slightly more involved in the GPU versions, since you can leverage the capabilities of both the CPU and GPU. As described in the section 2.4 there are also a number of performance strategies to consider, such as minimizing memory transfers. The GPU implementations are based on the framework CUDA from Nvidia, described in section 2.4. Specifically, version 7.0 of the CUDA Toolkit was downloaded from Nvidia's web page. This version is the latest at the time of writing this report, and was the first version to include a function for Cholesky factorization, which is interesting to study for this thesis.

The fact that double precision has to be used for stability in these benchmark problems is problematic for performance reasons, since GPUs are much slower with double precision arithmetic than single precision. If the algorithm could be constructed such that single precision could be used instead, the GPU would be better utilized and the performance would presumably increase significantly.

Two GPU versions were implemented, having some things in common. The constraint matrix A, which has a density of approximately 1%, is stored in a sparse matrix format in order to save memory and reduce computation time. For sparse matrix operations, both GPU versions use the cuSPARSE library described in section 2.4. These operations include the SpMV and the SpMM multiplications. Since cuSPARSE only performs the SpMM multiplication on matrices stored in the CSR format, this format was chosen to be used in this thesis. As noted in [3], CSR is not the best format for the SpMV operation. However, in the implementations of this thesis the SpMV operation only uses a small fraction of the computation time, so the CSR format was considered good enough. The CPU was in both versions responsible for orchestrating the algorithm (which is inherently sequential), delegating computations to the GPU.

The SpMM multiplication is performed in the same way on both GPU versions of the algorithm. In cuSPARSE, this operation is performed with two function calls: `cusparseXcsrgemmNnz` which returns the number of non-zero elements and the vector with row indices in the CSR matrix format (see section 2.1.1 for more details). `cusparseDcsrgemm` which returns the vector with column coordinates and the vector with values. The result of the SpMM multiplication in each iteration has the same non-zero pattern. Because of this, the `cusparseXcsrgemmNnz` function is only called once per solution and the result is reused in each iteration.

GPUDense

The version `GPUDense` uses the MAGMA library for the Cholesky factorization. The MAGMA library accepts matrices that reside on GPU memory, allowing this version of the algorithm to be completely free of any explicit memory transfers. The MAGMA library uses both the CPU and GPU for its computations however, to ensure that the CPU does not idle while waiting for the GPU computations to complete. The result of performing the SpMM multiplication on ADA^T is a sparse matrix. This matrix has to be converted to dense format before performing the factorization with the MAGMA library. The cuSPARSE library includes a function for performing this conversion, which is fast and does not affect performance significantly. The `GPUDense` version of the algorithm is listed in Algorithm 3.

In order for the MAGMA library's Cholesky factorizer to work with the highly ill-conditioned matrices which occur in the context of the PCIPM, it has to be modified. The MAGMA Cholesky factorizer uses a blocking algorithm which calls the LAPACK function `dpotrf` for individual blocks in the factorization. This LAPACK function was replaced by a Cholesky-Infinity function described in section 2.1.2 and implemented in accordance to [44]. Since LAPACK functions are generally highly optimized, a performance comparison was done to ensure that the modified Cholesky-Infinity function did not perform much worse than the original one. This is important so that the performance comparison in section 4 does not misleadingly indicate that the MAGMA solver is slower than it actually is. As can be seen in Figure 3.3, the modified algorithm is slightly slower but does not reduce performance more than 5% for any test case.

Algorithm 3 Version `GPUDense`. Light green color indicates conversion between sparse and dense matrix format. Orange color indicates operations performed by the MAGMA library. No color indicates computations performed on the GPU, using libraries that is included in the CUDA Toolkit or kernels developed for this thesis.

```
 1: $\tau_0 = 0.9995$
 2: Find initial values for $x, y, z, s, w$.
 3: while $stop > tol$ do
 4:     $D = (X^{-1}Z + S^{-1}W)^{-1}$
 5:     $H = ADA^T$
 6:     Convert $H$ to dense format.
 7:     Compute $U$ such that $U^TU = H$.
 8:     Compute residuals $r_b$, $r_u$, $r_c$, $r_{xz}$, $r_{sw}$, $r_{cp}$.
 9:     Compute predictor direction $\Delta y$ using $U^TU\Delta y = rhs$.
10:     Compute predictor direction $\Delta x, \Delta z, \Delta s, \Delta w$.
11:     Compute $\alpha_P$ and $\alpha_D$ such that $x+\alpha_P\Delta x, s+\alpha_P\Delta s, z+\alpha_D\Delta z, w+\alpha_D\Delta w > 0$
12:     if $\tau_0\alpha_P < 1$ or $\tau_0\alpha_D < 1$ then
13:         Compute centering parameter $\mu$.
14:         Compute residuals $r_{xz}$, $r_{sw}$, $r_{cc}$.
15:         Compute corrector direction $\Delta_C y$ using $U^TU\Delta_C y = rhs$.
16:         Compute corrector direction $\Delta_C x, \Delta_C z, \Delta_C s, \Delta_C w$.
17:         $\Delta v = \Delta v + \Delta_C v$
18:         Compute $\alpha_P$ and $\alpha_D$ such that $x + \alpha_P\Delta x, s + \alpha_P\Delta s, z + \alpha_D\Delta z, w + \alpha_D\Delta w > 0$
19:     end if
20:     Take steps in the $\Delta v$ direction.
21:     $stop = \frac{||r_b||}{\max(1,||b||)} + \frac{||r_u||}{\max(1,||u||)} + \frac{||r_c||}{\max(1,||c||)} + \frac{|c^Tx-b^Ty+u^Tw|}{\max(1,|c^Tx|,|b^Ty-u^Tw|)}$
22: end while
```

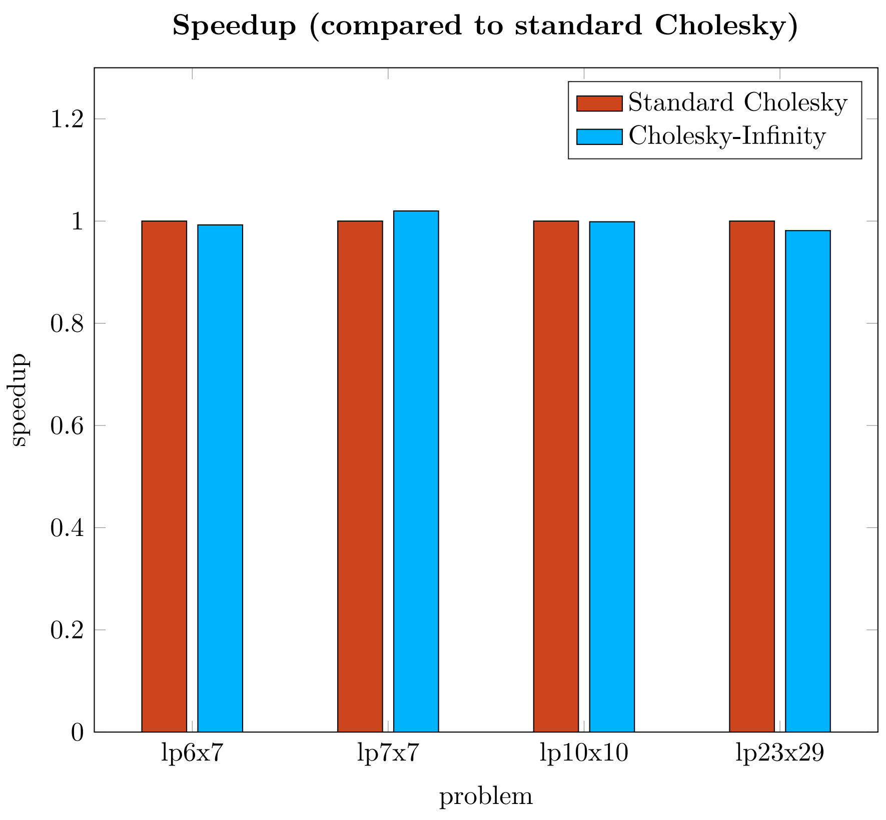


Figure 3.3. Comparing the standard MAGMA Cholesky factorizer to the modified Cholesky-Infinity version. Plot shows speedups of full solutions of the specified LP problems.

In addition to the main `GPUDense` version, a version was implemented where the dense MAGMA Cholesky solver is replaced by the new cuSOLVER library introduced in CUDA Toolkit v.7.0. A performance comparison was made, and the result can be seen in Figure 3.4. Since MAGMA was found to be slightly faster for all test cases but the smallest one, it was chosen to be used for the performance comparisons in section 4.

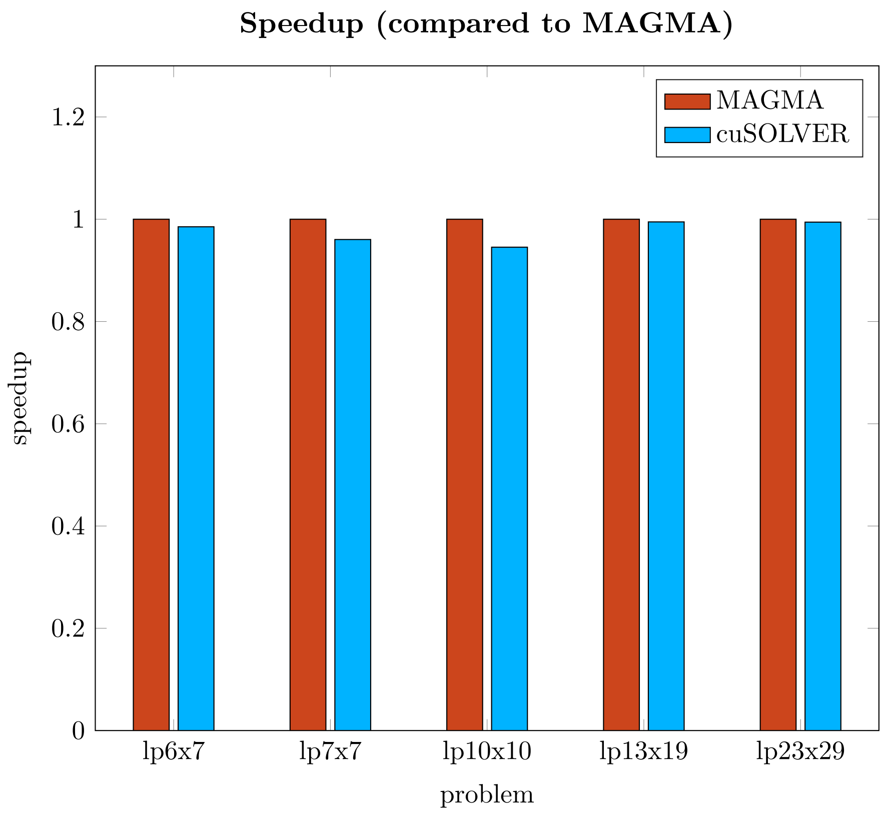


Figure 3.4. Comparing the MAGMA Cholesky factorizer to the corresponding function in cuSOLVER. Plot shows speedups of full solutions of the specified LP problems.

For completeness, a version of the `GPUDense` solver was implemented which uses a dense matrix representation for the matrix A. This version is slower, as was the case for the corresponding CPU version, but the performance difference is not as large as for the CPU version. The speedup can be seen to be no more than 2.5x in Figure 3.5 and for the smallest problem which is relatively dense, storing A as dense gives a slightly faster solver. Because of memory restrictions, the largest problem tested is `lp13x19`.

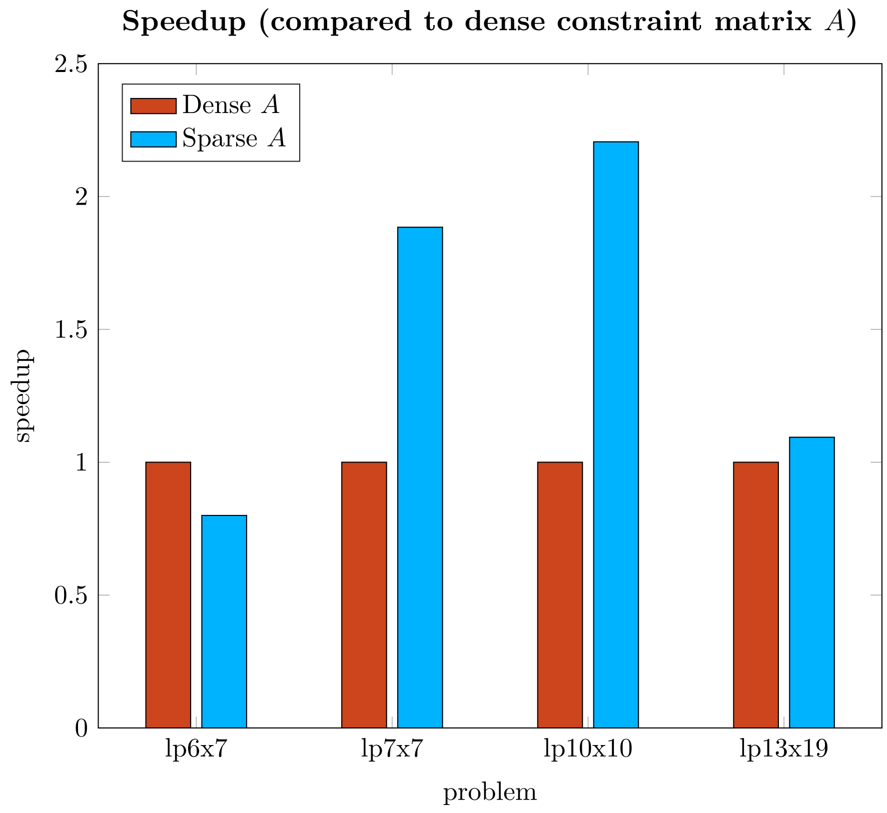


Figure 3.5. Speedup achieved using sparse linear algebra compared to dense linear algebra for the constraint matrix A using solver `GPUDense`. Plot shows speedups of full solutions of the specified LP problems.

GPUSparse

The version `GPUSparse` uses the CHOLMOD library for the Cholesky factorization. Since this library requires the input matrix to be located on the CPU memory, a memory transfer has to be performed before doing the factorization. The CHOLMOD library requires the matrix to be on the CSC sparse matrix format, but the matrix is stored as CSR on the GPU. Since the matrix ADA^T is symmetric, the row indices of the CSR format can simply be copied into the column indices of the CSC format, eliminating the need for a costly conversion. Apart from copying the large ADA^T matrix some other copies between CPU and GPU is required as well, but only small data in the form of vectors of length m. The host memory which is used for the GPU-CPU copies is allocated as pinned, using the `cudaMallocHost` function. The pinned host memory is allocated only once per solution and reused in all iterations.

As in the `GPUDense` version, the Cholesky factorizer was modified into a Cholesky-Infinity algorithm. It was done using the same strategy, replacing a LAPACK function call in the CHOLMOD library with a version implemented in accordance to [44]. This modification increased the performance slightly, as can be seen in Figure 3.6. Using the standard Cholesky factorization, it was able to achieve an accuracy of 10^{-4} for all problems except `lp46x51`. This seems to indicate that it is less sensitive to numerical errors than the MAGMA Cholesky factorizer in the `GPUDense` version.

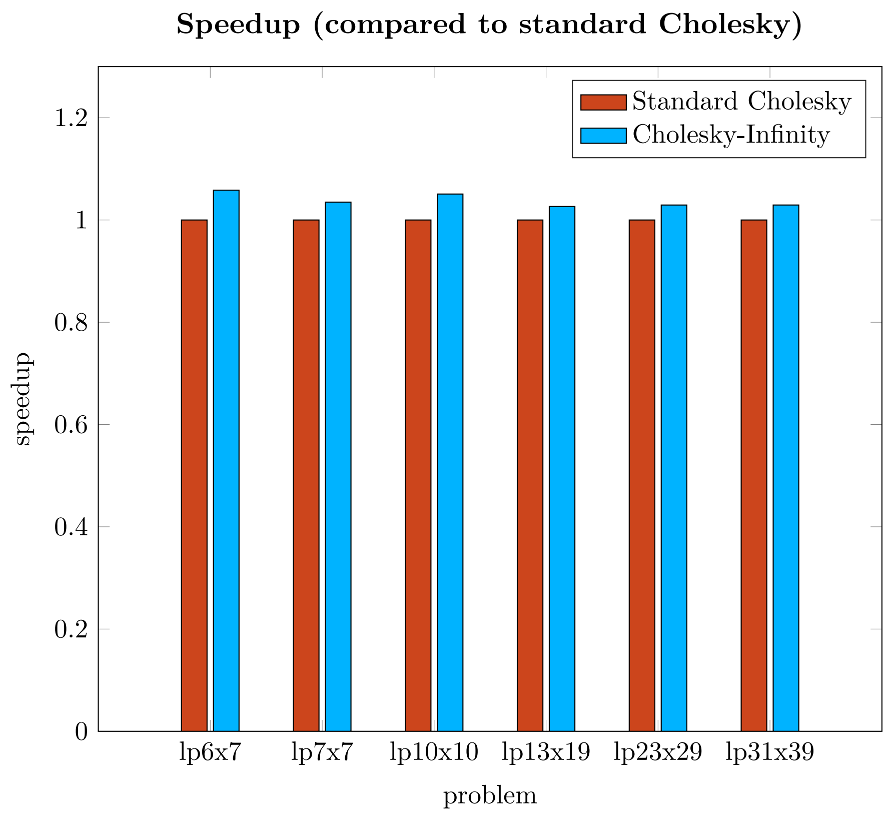


Figure 3.6. Comparing the standard CHOLMOD Cholesky factorizer to the modified Cholesky-Infinity version. Plot shows speedups of full solutions of the specified LP problems.

As in the `CPUSparse` version, a reordering algorithm is used in order to reduce the number of non-zero elements in the Cholesky factor U. The `GPUSparse` version of the algorithm is listed in Algorithm 4.

Algorithm 4 Version `GPUSparse`. Light green color indicates memory transfer between CPU and GPU. Light blue color indicates operations performed by the CHOLMOD library. No color indicates computations performed on the GPU, using libraries that is included in the CUDA Toolkit or kernels developed for this thesis.

```
 1: $\tau_0 = 0.9995$
 2: Find initial values for $x, y, z, s, w$.
 3: while $stop > tol$ do
 4:     $D = (X^{-1}Z + S^{-1}W)^{-1}$
 5:     $H = ADA^T$
 6:     Copy $H$ to CPU memory.
 7:     Compute $U$ such that $U^T U = H$.
 8:     Compute residuals $r_b$, $r_u$, $r_c$, $r_{xz}$, $r_{sw}$, $r_{cp}$.
 9:     Copy $rhs$ to CPU memory.
10:     Compute predictor direction $\Delta y$ using $U^T U \Delta y = rhs$.
11:     Copy $\Delta y$ to GPU memory.
12:     Compute predictor direction $\Delta x, \Delta z, \Delta s, \Delta w$.
13:     Compute $\alpha_P$ and $\alpha_D$ such that $x+\alpha_P\Delta x, s+\alpha_P\Delta s, z+\alpha_D\Delta z, w+\alpha_D\Delta w > 0$
14:     if $\tau_0\alpha_P < 1$ or $\tau_0\alpha_D < 1$ then
15:         Compute centering parameter $\mu$.
16:         Compute residuals $r_{xz}$, $r_{sw}$, $r_{cc}$.
17:         Copy $rhs$ to CPU memory.
18:         Compute corrector direction $\Delta_C y$ using $U^T U \Delta_C y = rhs$.
19:         Copy $\Delta_C y$ to GPU memory.
20:         Compute corrector direction $\Delta_C x, \Delta_C z, \Delta_C s, \Delta_C w$.
21:         $\Delta v = \Delta v + \Delta_C v$
22:         Compute $\alpha_P$ and $\alpha_D$ such that $x + \alpha_P\Delta x, s + \alpha_P\Delta s, z + \alpha_D\Delta z, w + \alpha_D\Delta w > 0$
23:     end if
24:     Take steps in the $\Delta v$ direction.
25:     $stop = \frac{||r_b||}{\max(1,||b||)} + \frac{||r_u||}{\max(1,||u||)} + \frac{||r_c||}{\max(1,||c||)} + \frac{|c^T x - b^T y + u^T w|}{\max(1,|c^T x|,|b^T y - u^T w|)}$
26: end while
```

Kernels

Both GPU versions use some kernels which were implemented specifically for use in this thesis. One example of such a kernel can be seen in Figure 3.7, which is used for updating the matrix $D = (X^{-1}Z + S^{-1}W)^{-1}$. Other examples are coefficient-wise multiplication and division and the step length calculation. The kernels implemented specifically for this thesis only perform $\mathcal{O}(m)$ and $\mathcal{O}(n)$ operations. The Nvidia Nsight Profiler was run on the program in order to ensure that the kernels did not use a disproportionate amount of time. The profiler ranks the kernels based on how much the performance of the program will be affected by optimizing that kernel. A rank of 100 means that it is an important kernel which should be optimized, and a rank of 1 means that optimization probably will not have a significant effect. All kernels implemented specifically for this thesis received a rank of 1. Because of this, not much time has been spent optimizing these kernels. All kernels that received a higher rating than 1 belong to some CUDA library.

```
template <typename T>
__global__ void buildD_kernel(T* __restrict__ valD,
                    const T* __restrict__ x, const T* __restrict__ z,
                    const T* __restrict__ s, const T* __restrict__ w,
                    int n, int nu)
{
        int id = blockIdx.x*blockDim.x + threadIdx.x;
        if (id < n)
                valD[id] = z[id]/x[id];
        if (id < nu)
                valD[id] += w[id]/s[id];
        if (id < n)
                valD[id] = 1/valD[id];
}
```

Figure 3.7. Kernel for calculating the diagonal matrix $D = (X^{-1}Z+S^{-1}W)^{-1}$. The array `valD` contains the diagonal of D.

3.4 Performance Analysis

The hardware specifications for the system on which all performance tests are performed can be seen in Table 3.2.

CPU	Intel Xeon E5-2630 v2
CPU Cores	6
CPU Clock Speed	2.60 GHz (3.10 GHz turbo)
CPU Memory Bandwidth	51 GB/s
RAM	32 GB
GPU	Nvidia GTX Titan Black
GPU Memory	6 GB
GPU Memory Bandwidth	288 GB/s

Table 3.2. Test system hardware.

The Intel Xeon E5-2630 v2 is based on the Ivy Bridge architecture and was released in Q3 2013. It currently retails for around 6 000 - 7 000 SEK at the time of writing this report. The Nvidia GTX Titan Black is based on the Kepler architecture (GK110) and was released in Q1 2014. It currently retails for around 10 000 SEK. What sets the GTX Titan apart form other graphics cards in the GTX series is that it has a double precision mode which increases double precision performance to 1/3 of the single precision performance, instead of the default 1/24. This makes the card suitable for this thesis, since double precision is required for numerical stability. However, by activating the double precision mode the memory bandwidth of the GPU is reduced from 336 GB/s to 288 GB/s.

All implementations of the algorithm are compiled with nvcc on a Ubuntu 14.04 machine, using the `-O3` compiler optimization flag. For the C++ parts of the code, nvcc uses the gcc compiler set to the C++11 standard. CUDA Toolkit v.7.0, gcc 4.8.2, OpenBLAS 0.2.8, and Eigen 3.2.3 were used. Some of the libraries have to be compiled before use. SuiteSparse 4.4.3 was compiled using gcc with support for multithreading using OpenBLAS, and with support for GPU-acceleration with compute capability 3.5 using nvcc. Other than these settings, the default settings in SuiteSparse were used. Magma 1.6.1 was compiled for the Kepler architecture using nvcc, gcc and gfortran, and using the BLAS package OpenBLAS.

The `CPUDense` version is run by a 64-bit version of MATLAB R2014b with Intel MKL 11.1.1.

Two different ways of measuring the computation times were used for each solver. One version measures the total solution time including finding the initial point and all iterations until the point where a tolerance of 10^{-4} was achieved. The other version measures the time the program spends doing various operations, such as the Cholesky factorization, SpMM and SpMV. The time it takes to read and parse the file containing the problem is not included in the measurements for any solver. All times measured and presented in this report are wall clock times. Most times are measured using the `std::chrono` time library introduced in the standard library of C++11.[8] All times that are not measured using `std::chrono` instead use the `cudaEventElapsedTime`, which is used for the per-operation time measurements of the GPU versions.

The memory used is measured for the GPU versions of the algorithm. The number presented signify the most memory that is occupied on the GPU during the entire solution run of a problem. The measurement is updated at some specific times during the algorithm, usually preceding a freeing of some temporary variables. This number is supposed to signify the memory that is required on the graphics card in order to solve the given problem.

The implemented program takes a number of parameters: the input file describing the LP problem; whether to print debug outputs; whether to use per-operation time measurements; and a list of solvers that will be executed sequentially. These parameters combined with bash scripts make it possible to batch the performance measurements which in total takes a few hours to run through. The timings are written to a file which is then parsed with special MATLAB functions in which the plots are generated.

The performance measurements were all done with the CPU governor set to `Performance` instead of the default `Ondemand`, to ensure that the CPU clock speed is not throttled and the turbo mode is activated. The graphics card was set to its CUDA Double Precision Mode, which increases the theoretical double precision performance from 213 GFLOP/s to 1700 GFLOP/s. The graphics card governor was also changed from the default Adaptive to Prefer Maximum Performance. The graphics card used for the performance measurements does not have any monitor plugged in, and is hence used as dedicated computation hardware. The graphics card is plugged in to a PCIe 3.0 x16 bus.

The problems are all solved on the same Ubuntu machine as was used for the compilation. A problem is considered solved when the stop criteria is smaller than 10^{-4}.

Chapter 4

Results

In this chapter, the results of the performance measurements are presented. The results are split into three sections: first a comparison of achieved accuracy of the solvers, then various computation time measurements and lastly the memory requirements.

4.1 Accuracy

The implemented solvers are able to achieve different accuracies for different problems. The accuracy refers to the value of the stop criteria, i.e. a smaller value indicates a higher degree of accuracy. In Table 4.1 the highest achieved accuracy is presented for each solver and problem. The solvers with Inf in the Cholesky column use the Cholesky-Infinity factorization and the solvers with Std use the standard Cholesky factorization. For the rest of the report the `GPUDense` and `GPUSparse` versions with the Cholesky-Infinity method are the ones that are being used in all benchmarks.

Solver								
Name	Cholesky	lp6x7	lp7x7	lp10x10	lp13x19	lp23x29	lp31x39	lp46x51
`CPUSparse`	Std	3.2e-7	7.9e-9	3.3e-6	1.1e-6	1.1e-7	7.5e-8	*
`CPUDense`	Std	3.0e-7	7.9e-9	3.3e-6	5.4e-7	6.3e-7	5.7e-8	**
`GPUSparse`	Std	2.6e-7	7.9e-9	3.3e-6	3.0e-7	6.3e-7	3.0e-7	7.2e-1
`GPUSparse`	Inf	3.1e-8	6.1e-11	2.9e-8	6.0e-8	2.4e-7	4.4e-9	3.1e-4
`GPUDense`	Std	2.6e-7	7.9e-9	3.3e-6	*	6.3e-7	**	**
`GPUDense`	Inf	6.1e-8	8.7e-10	1.5e-9	2.4e-7	1.4e-7	**	**

Table 4.1. Achievable accuracy of different solvers. A smaller value indicates a higher degree of accuracy.
* numerical error
** out of memory

As can be seen in Table 4.1, the versions using the Cholesky-Infinity algorithm are consistently able to achieve a higher degree of accuracy than their standard Cholesky counterparts.

The reason why there are differences between the solvers in regard to accuracy for a given problem may have to do with which Cholesky factorization implementation is used. For all solvers, the achievable accuracy is limited by the matrix ADA^T. When this matrix becomes semi-definite or indefinite, the algorithm aborts. Implementation details may include different pivot element limits for when the matrix is considered non-definite. Different factorization strategies may also cause slight numerical differences between the algorithms. Mostly however, the solvers using the standard Cholesky factorization achieve very similar accuracies.

It is not clear why the `lp13x19` problem does not work with the unmodified `GPUDense` solver. It is suspected that the structure and density of the problems are important factors in regards to the ill-conditioning of ADA^T, but this has not been studied further.

Solver								
Name	Cholesky	lp6x7	lp7x7	lp10x10	lp13x19	lp23x29	lp31x39	lp46x51
`CPUSparse`	Std	20	18	20	22	21	23	*
`CPUDense`	Std	20	18	20	22	21	23	**
`GPUSparse`	Std	20	18	20	22	21	23	*
`GPUSparse`	Inf	20	18	20	22	21	23	*
`GPUDense`	Std	20	18	20	*	21	**	**
`GPUDense`	Inf	20	18	20	22	21	**	**

Table 4.2. Number of iterations to reach an accuracy of 10^{-4} for different solvers.
* numerical error
** out of memory

Table 4.2 shows that for each problem all successful solvers require the same number of iterations to reach an accuracy of 10^{-4}. Since all solvers are based on the same mathematical operations, this is expected and can be seen as an indication that the solvers are working as intended. For all problems, there are almost no differences in the convergence of the four solvers. To illustrate this, Figure 4.1 shows convergence of the four solvers for the problem `lp10x10`. The solvers converge so similarly that they appear to be identical in the plot. The only visible difference between the solvers can be seen on the last iteration where `GPUSparse` achieves a slightly better accuracy than the other solvers.

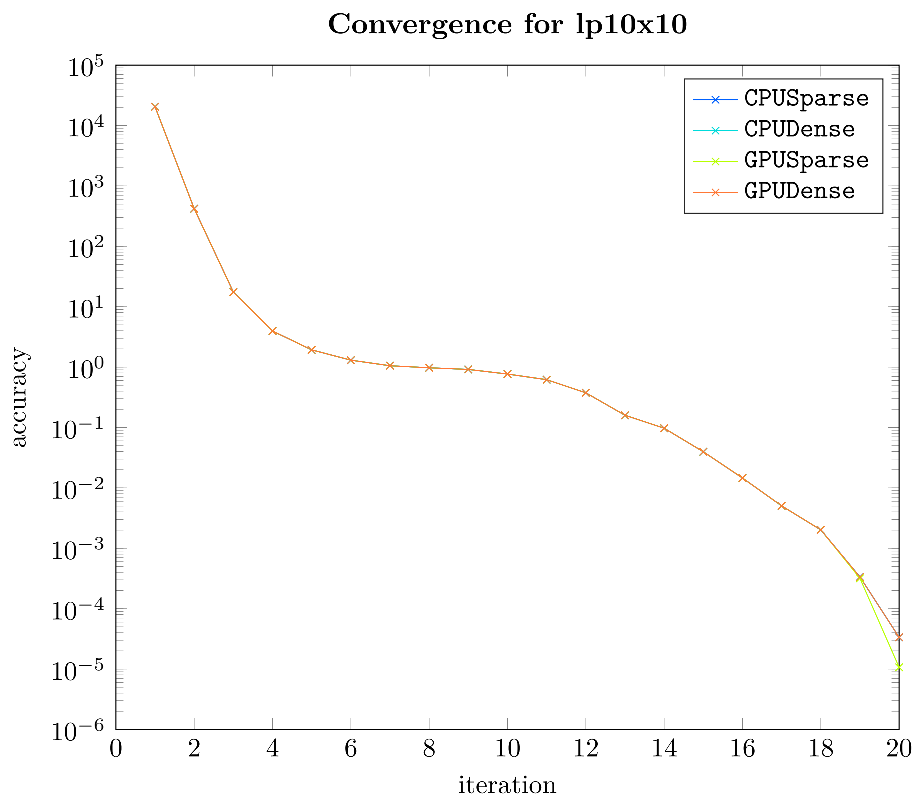


Figure 4.1. Convergence for problem `lp10x10`. The convergence of the four solvers are almost identical.

4.2 Computation Time Analysis

In this section, the main results of this thesis are presented. This thesis aims to answer for which sizes of linear programming problems it is possible to speed up the predictor-corrector interior point method. The results of the performance measurements are shown in the following subsections, and using these results the problem statement can be answered.

The speedups presented in this section follow the formula

$$S = \frac{T_{ref}}{T_{new}}$$

where S is the speedup, T_{ref} is the time of the reference solver and T_{new} is the time of the solver that is being considered.

4.2.1 Benchmark Consistency

All benchmarks were run with the same hardware settings, as described in section 3.4. As many applications as possible were closed before performing the benchmark tests. There is no randomness in the algorithm, so every run should in theory perform the exact same instructions. In Table 4.3, the relative difference of the computation time of two test runs is presented. The relative difference is calculated as $d = \left|\frac{t_1 - t_2}{t_2}\right|$.

Solver	lp6x7	lp7x7	lp10x10	lp13x19	lp23x29	lp31x39
`CPUSparse`	0.1%	0.2%	0.1%	0.1%	0.1%	0.1%
`CPUDense`	0.1%	1.1%	2.5%	2.4%	0.4%	2.2%
`GPUSparse`	0.1%	0.8%	0.4%	0.4%	0.4%	0.4%
`GPUDense`	0.2%	0.1%	0.1%	0.0%	0.0%	

Table 4.3. Benchmark consistency. Relative difference in measured computation time between two runs of the benchmark suite. All numbers rounded to a tenth of a percent.

As can be seen in Table 4.3, the benchmarks times are quite consistent. The consistency was worst for the `CPUDense` solver, having a relative difference in computation time of less than 3%. All others solvers have a relative difference in computation time of less than 1%. The results presented in the rest of the report are means of three different test runs. Because of the small relative difference in computation time between runs, it is believed that this manner of constructing the final results is accurate enough for the purposes of this thesis.

4.2.2 Total Solution Time

The total solution times for each solver and problem are shown in Figure 4.2. The corresponding speedups are shown in Figure 4.3. These times have been measured as described in section 3.4.

The GPU solvers are faster than the CPU solvers for every tested problem. The figures shows that the `GPUDense` solver achieves a speedup of 4x - 6x compared to the best CPU solver. The `GPUSparse` solver achieves a speedup of 2x - 3x compared to the best CPU solver. Comparing the best CPU solver and the best GPU solver for each individual problem, a speedup of at least 2x can be achieved by the GPU solver.

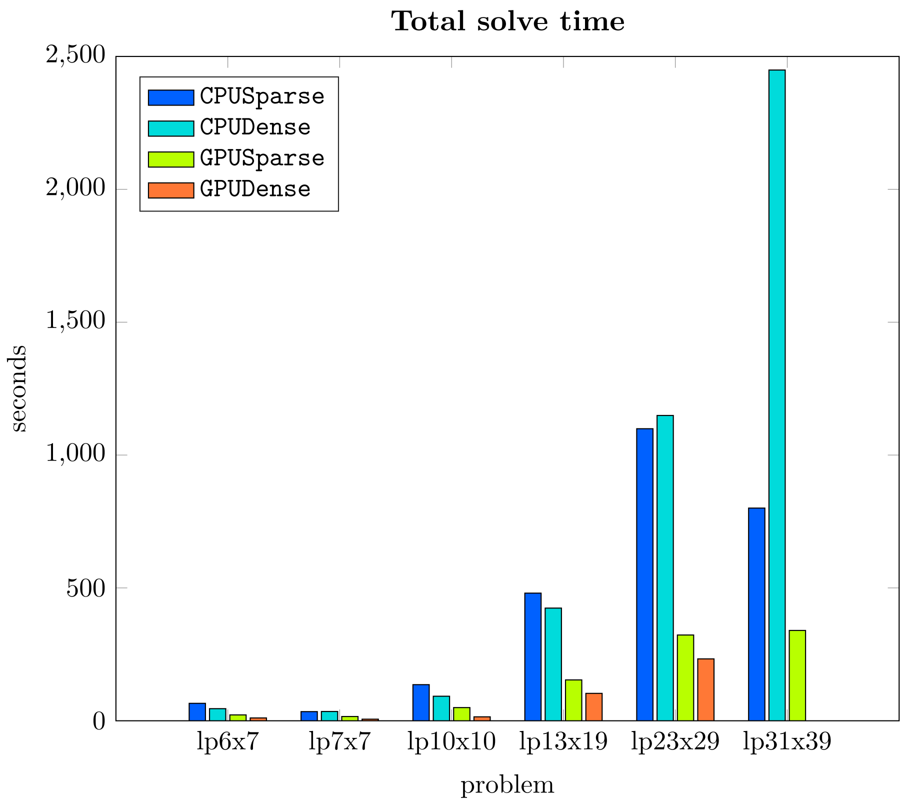


Figure 4.2. Total solution time for each solver and problem. Note that `GPUDense` cannot solve `lp31x39` since the memory requirements are too high.

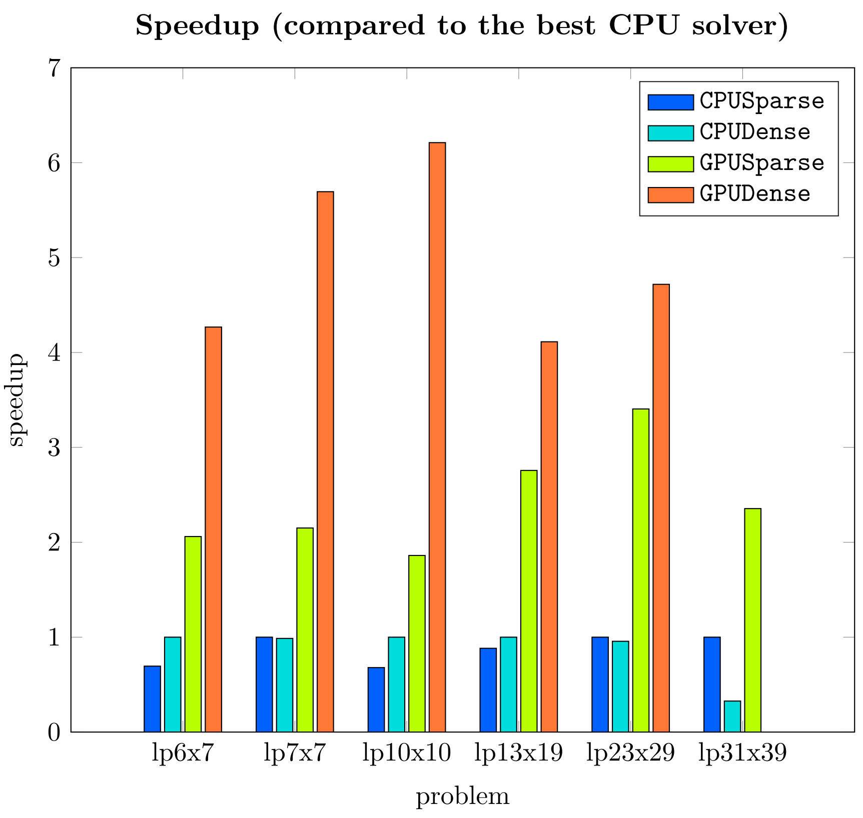


Figure 4.3. Speedup of all four solvers compared to the fastest CPU solver for each problem. Note that `GPUDense` cannot solve `lp31x39` since the memory requirements are too high.

4.2.3 Per-Operation Time

In Figure 4.4, the time for individual operations in the four solvers are shown. These timings were first measured to be able to see which operations are worth trying to optimize, and later also to compare how well certain operations can be accelerated using the GPU.

It is clear from Figure 4.4 that the computation time of all solvers consists almost exclusively of the SpMM multiplication and the Cholesky factorization. All other operations consist of less than 3% of the total time for any combination of solver and problem. Both the Cholesky factorization and the SpMM multiplication are accelerated significantly using a GPU.

For the `GPUDense` solver, the SpMM multiplication dominates the other operations as the problem size grows. The dense Cholesky factorization scales well on the GPU. For the `GPUSparse` solver, the relation is reversed: the Cholesky operation uses a larger ratio of the computation time as the problem size grows. The SpMM multiplication is performed in the same way in both GPU solvers and uses a similar amount of time.

Since both the `CPUSparse` and `GPUSparse` versions use CHOLMOD to factorize the sparse constraint matrix, this data can be used to find the speedup of CHOLMOD when it utilizes the GPU to factor these matrices. The speedup of CHOLMOD was found to be 1.4x - 3.9x compared to using only a CPU.

The `GPUSparse` version is the only version where memory operations are not completely negligible. This is expected since in this version the large ADA^T matrix is copied from device memory to host memory every iteration. Despite this large copy, memory operations use less than 1% of the total computation time for any problem.

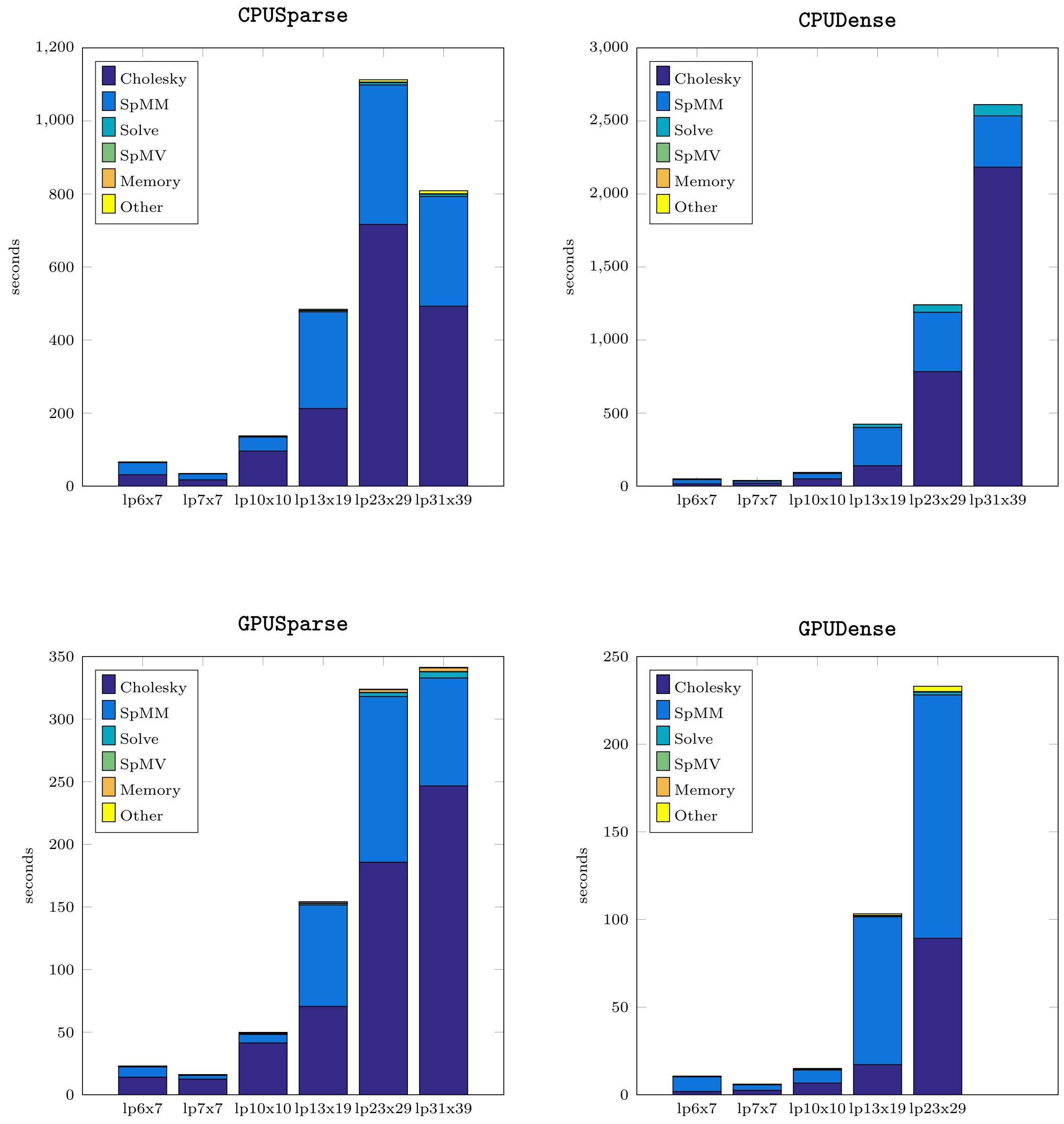


Figure 4.4. Execution time for individual operations in the four solvers. Note that the scale differs between the solvers. Note that `GPUDense` cannot solve `lp31x39` since the memory requirements are too high.

4.2.4 Comparison to the MATLAB `linprog` Function

In order to convince the reader that the solvers that have been developed during this thesis are comparable in computation time to existing solvers, the GPU versions of the algorithm are compared to MATLAB's built-in linear programming solver `linprog` set to use the default solver `interior-point`.[28] The result of this comparison is shown in Figure 4.5.

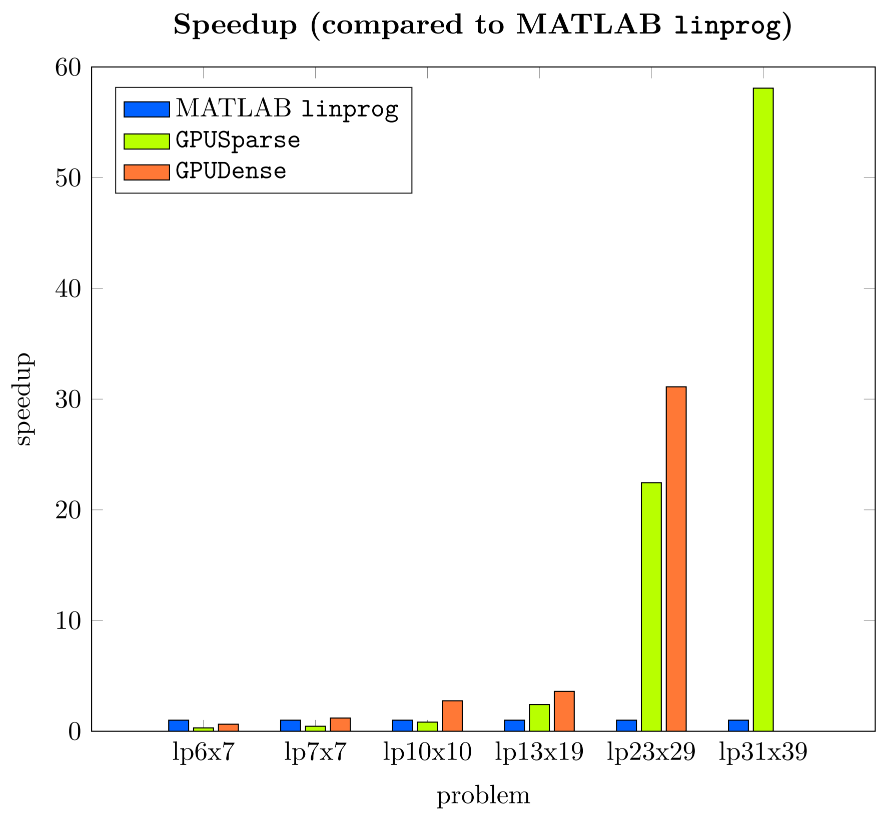


Figure 4.5. Speedup compared to MATLAB's `linprog` linear programming solver. Note that `GPUDense` cannot solve `lp31x39` since the memory requirements are too high.

For the smallest studied problem, `linprog` is slightly faster than the GPU implementations. This is probably related to the various optimizations implemented in `linprog`, as discussed in section 3.3. For all problems but the smallest, `linprog` is slower than the `GPUDense` solver.

For large problems, `linprog` is very slow compared to the GPU versions. `linprog` solves the `lp31x39` problem in 5.5 hours. This corresponds to a speedup for `GPUSparse` of 58. The `interior-point` solver of `linprog` does not seem to be developed for such large problems as these, though MathWorks recommends it for solving large-scale sparse LP problems.[26]

4.2.5 Graphics Card Comparison

The graphics card used for all comparisons, the Nvidia GTX Titan Black, is an expensive card with very high double precision speed. It may be of interest to see how a slower GPU would perform in these tests, in order to understand what is required in terms of hardware to speed up the PCIPM. Because of this, a comparison has been made with the double precision mode of the GTX Titan Black switched off (213 GFLOP/s) and the standard configuration with double precision mode switched on (1700 GFLOP/s). The result of this comparison can be seen in Figure 4.6.

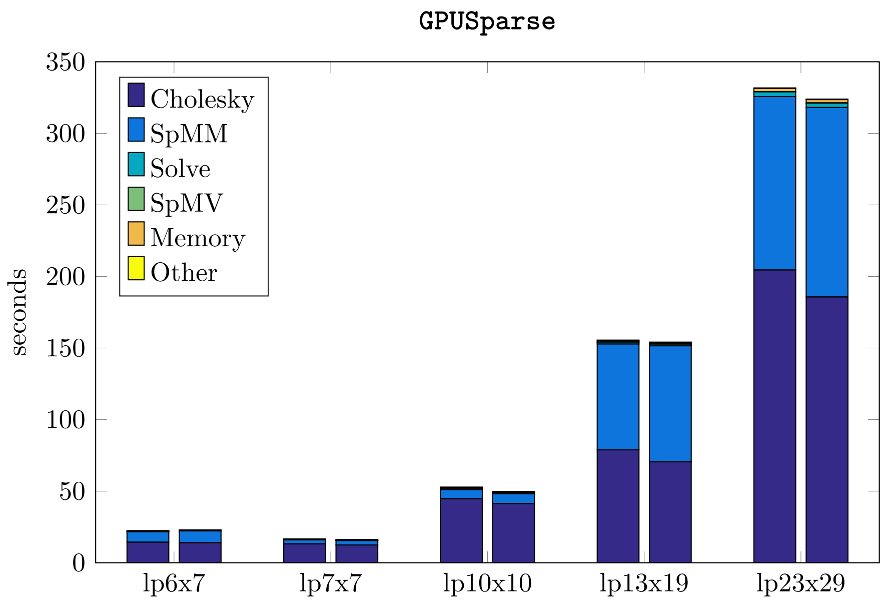
Note that the SpMM multiplication is not accelerated by activating the double precision mode, instead it actually seems to be slightly slower. This is presumably caused by the fact that the memory bandwidth is slightly higher (up from 288 GB/s to 336 GB/s) when the double precision mode is switched off. It is mentioned in [3] that sparse matrix operations are usually bound by memory bandwidth, which may explain these results.

The sparse Cholesky factorization is accelerated slightly. This effect is offset by the slight increase in SpMM time, which means that the `GPUSparse` solver is approximately equally fast whether the GPU is set to work with 213 GFLOP/s or 1700 GFLOP/s.

These results suggest that double precision mode should be activated for the sparse Cholesky factorization and then deactivated for the SpMM multiplication in the `GPUSparse` solver. Unfortunately, the double precision mode setting cannot be changed in the program code. It is actually changed in the Nvidia X Server Setting on an operating system level.

From the results regarding `GPUDense` it is clear that the dense Cholesky factorization is accelerated significantly by activating the double precision mode on the GPU.

These results suggest that compared to the CPU versions of the algorithm a speedup of 2x - 3x can be expected using the `GPUSparse` solver, even for GPUs with low floating-point performance. The memory bandwidth may be an important factor, but more research is required before anything conclusive can be said about this.



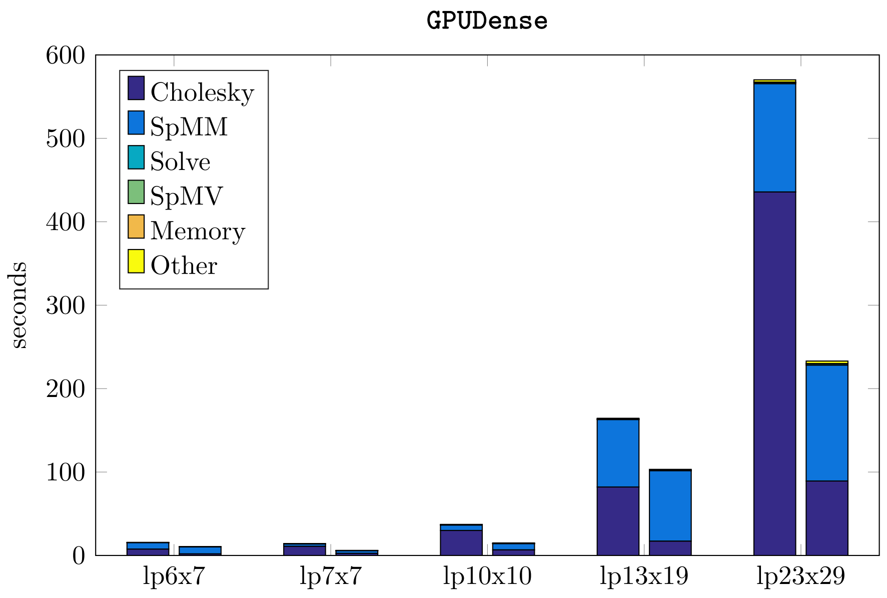


Figure 4.6. Comparison of two GPU speeds for the two GPU solvers. The left bar in each group corresponds to GTX Titan Black 213 GFLOP/s and the right bar in each group correspond to GTX Titan Black 1700 GFLOP/s. Note that the scale differs between the solvers.

4.3 Memory Requirements Analysis

Since the graphics card only has 6 GB of memory, the maximum problem size is limited. Figure 4.7 shows the observed memory consumption of each GPU solver and problem.

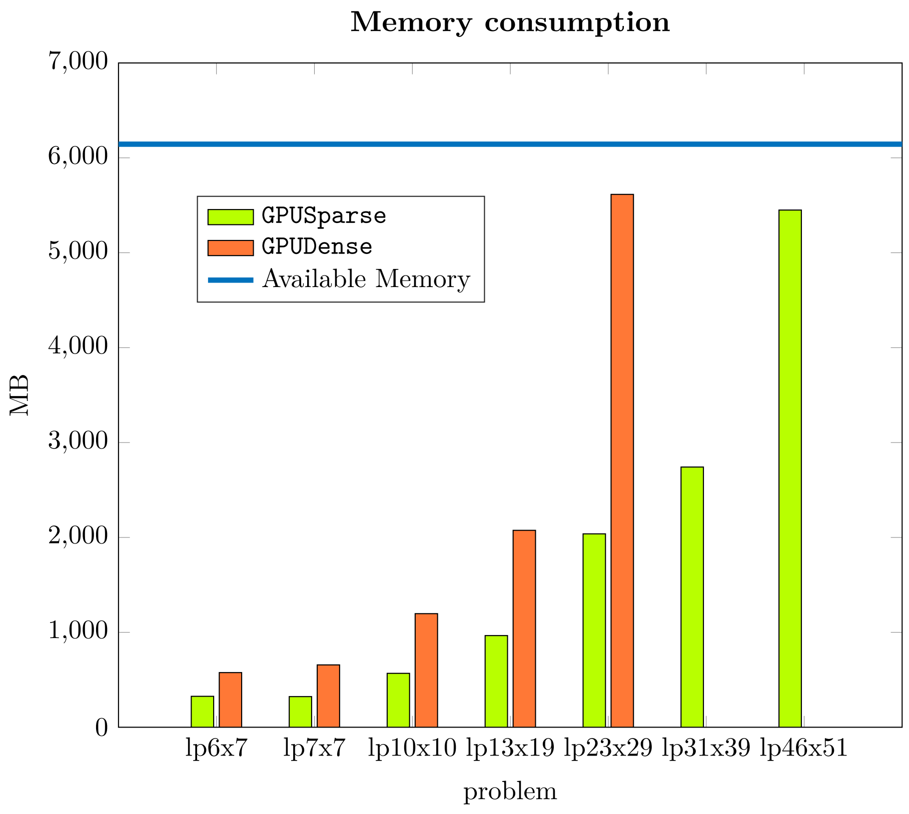


Figure 4.7. Memory consumption of each solver and problem.

As can be expected, the dense solver uses more memory than the sparse one even though the density of ADA^T is quite high. The results of this study suggest that the maximum problem size for the `GPUDense` solver is approximately the size of `lp23x29` and the maximum problem size for the `GPUSparse` solver is approximately the size of `lp46x51`. In the sparse solver, CHOLMOD needs to allocate a large amount of GPU memory in order to perform its computations quickly. In [37], they use 3 GB for every problem, but the memory allocations are not taken into account in the performance comparisons. Since the memory allocations are taken into account in this project and they constitute a non-negligible part of the computation time, the memory allocated to CHOLMOD was set to m^2 bytes which was deduced by experiment to be faster than both $m^2/2$ and $2m^2$.

For LP problems arising in other contexts the density may be much lower, which will make a large difference in the memory consumption of the `GPUSparse` solver, but not much difference for the `GPUDense` solver. The reverse is true as well: if the density is higher, the memory consumption of the `GPUSparse` solver will increase but the memory consumption of the `GPUDense` solver will stay approximately the same. A rule of thumb may be that the memory requirements of the `GPUDense` solver depend mostly on the number of constraints (i.e. the dimension of ADA^T) and the memory requirements of the `GPUSparse` solver depend mostly on the number of non-zeros in ADA^T.

Chapter 5

Discussion

In this chapter, we discuss the results and whether they can be trusted or not. Some ideas on potential future research is also presented.

5.1 Threats to Validity

Doing a completely fair performance comparison of an algorithm between a CPU and a GPU is impossible. First, the comparison will be highly dependent on which hardware is being used. In this thesis we have been comparing a high-end CPU to a high-end GPU, which has been selected to be roughly on par in terms of performance. Second, the respective architectures of the CPU and the GPU are different, where the CPU is based on a multi-core (2-16 cores) architecture and the GPU is based on a many-core (thousands of cores) architecture. Because of this, the implementations must be made using different strategies. Since the performance comparison is highly dependent on the implementation, this is likely a source of unfairness. In [22], it is argued that many CPU vs GPU comparisons are flawed and that a GPU speedup of no more than 2.5x is reasonable. They argue that the reason why performance comparisons often show a higher speedup is because the CPU version is not as well optimized as the corresponding GPU version. It can be argued that this flaw is present in this thesis, due to the fact that the SpMM multiplication on the CPU is single-threaded. The reason why it is single-threaded is that no C++ library was found which performed this operation using more than one core. It was investigated thoroughly and the fastest library was chosen. It is unclear how important multi-threading is for sparse matrix operations since these operations commonly are bound by memory bandwidth.[3] We have shown in this thesis (see Figure 4.6) that arithmetic speed does not significantly affect the speed of the SpMM multiplication on the GPU, and it may be reasonable to assume that the same would be true for the CPU as well.

In addition to the problem with the hardware comparison and the SpMM multiplication, the CHOLMOD library may not have been used entirely correctly in the `GPUSparse` version. Using only the CPU, the symbolic factorization and the allocated workspace of this library can be reused every iteration. Unfortunately, activating the GPU support lead to errors using the same strategy. This problem was solved using costly reallocations (of pinned host memory) every iteration. This may be a limitation of the CHOLMOD library, but if it is an implementation error in this project the comparisons in chapter 4 may underestimate the potential performance of the `GPUSparse` version.

Another thing that may be criticized is that the initial data copy of the problem onto the GPU is not included in the performance comparisons, even though that operation is non-present in the CPU version. The reason is that the code is structured in such a way that the data is set once, and then the solvers can be run multiple times without having to read the problem from disk every time. The time it takes to copy the data is negligible compared to the solution time, in the order of 0.1 seconds even for the `lp31x39` problem, so this unfairness cannot be considered severe. The reason why the data copy cost is not higher is that the largest data being copied is the constraint matrix A, which is a relatively sparse matrix (around 0.01 density). The ADA^T matrix, which is the one using the most memory, is generated on GPU.

The preceding arguments tell us that the performance comparison is not necessarily fair, which is a fact that must be accepted. However, we have made an effort to increase the fairness as much as possible:

- Although the optimizations for the respective architecture are different, the same mathematical operations are performed in the CPU versions of the algorithm and the GPU versions. This claim is backed up by Figure 4.1 and Table 4.2.
- The CPU and the GPU hardware are comparable in retail price, and represent feasible choices for heavy computations.
- All timings are wall-clock times. Most measurements are of the total time for a converged solution, which should be relatively fair and unambiguous. A solution is considered converged when the stop criteria has reached a size of less than 10^{-4}.
- The reported timings are averages of three different test runs.

5.2 GPU-Acceleration of the Predictor-Corrector Method

The problem statement of this thesis was formulated as

> *For what sizes of linear programming problems is it possible to use a GPU to speed up the predictor-corrector interior point method, compared to using only a CPU?*

The results of this thesis indicate that it is indeed possible to use the GPU to accelerate the predictor-corrector interior point method for all evaluated problem sizes between 6 000 and 31 000 constraints. The problem size is restricted by the available GPU memory, but the results suggest that until that problem size limit is reached we can expect a speedup of at least 2x by using the GPU compared to using only the CPU.

The speedup achieved is 4x - 6x for all problems except `lp31x39` where a speedup of 2x was achieved. This result seem to correspond quite well to other efforts to accelerate the PCIPM on the GPU, notably by Gade-Nielsen et al.[13] where a speedup of 2x - 6x was achieved for smaller and denser problems. A large difference between Gade-Nielsen's solver and ours is that our implementations store the constraint matrix A sparsely. While `linprog` is as much as 3x faster than Gade-Nielsen's GPU solver for large sparse problems, our fastest GPU solver is shown to be as much as 58x faster than `linprog` without exploiting any specific structure of our benchmark problems. However, our implementation perform worse than `linprog` for small problems where Gade-Nielsen's solver seem to excel. Since Gade-Nielsen's report and ours use different benchmark problems and different hardware nothing conclusive can be said of our relative performance.

The speedup of CHOLMOD when using a GPU was investigated as well, in the context of the interior point method for the first time (to the best of our knowledge). This was mentioned in [12] as something interesting to consider for future research. The achieved speedup was found to be 1.4x - 3.9x. These numbers can be compared to [37], where the speedup was found to be 0.6x - 3.5x when the memory allocations were not taken into account. The reason why the GPU-acceleration works well in the context of this thesis is probably related to the high density and large size of the matrices considered. We are also using a later version 4.4.3 of SuiteSparse in this thesis, compared to version 4.3.0 in [37].

5.3 Implementation Comments

It is worth to note that the implemented solvers using the standard Cholesky factorization was not able to achieve an accuracy of 10^{-8} on most problems before the numerical instabilities made the solution break down. This is part of the reason why 10^{-4} was selected as the required accuracy. As discussed in section 3.3.3, the Cholesky factorizer in the MAGMA library was more sensitive to numerical errors than the other tested libraries. Before it was modified to use Cholesky-Infinity, it was not even able to do one iteration of the `lp13x19` problem. After the modification, the same problem was solved to an accuracy of 10^{-7}. The CHOLMOD library was found to be less sensitive, but for the `lp46x51` problem, it was not able to converge to a reasonable accuracy before the matrix became semi-definite. This problem was relieved by implementing the Cholesky-Infinity method in the CHOLMOD library as well. These experiments highlight the need to modify the Cholesky factorizers before using them in a large-scale interior point method context, and that it is possible to do so in a GPU context.

There are two reasons why the `lp46x51` problem was not used for the performance comparisons. First, an accuracy of 3.1×10^{-4} was the best accuracy achieved, which is slightly worse than the accuracy of 10^{-4} used for the other performance comparisons. Second, the Cholesky-Infinity algorithm was tested for the `CPUSparse` version of the algorithm, but was found to be very slow compared to the standard Cholesky algorithm. This is because our implemented version of the LAPACK function `dpotrf` is not blocking and hence single-threaded. This is not important for the GPU versions, since in that algorithm the `dpotrf` calls are for small individual blocks of a larger algorithm which is already multi-threaded. There are no reason why a multi-threaded CPU version of the Cholesky-Infinity algorithm could not be implemented, but it has not been done in this thesis. The above stated arguments imply that the performance comparison would be biased in the GPU version's favour and it was therefore excluded.

It may seem strange that a sparse Cholesky solver was chosen to be part of the performance tests when the matrices that are being factored have a density of around 0.25. There is no universal density limit for when a matrix should be treated as sparse, but for most purposes a matrix with a density of 0.25 would be treated as dense. One reason for why a sparse representation was tested in this thesis is because it allowed us to use the CHOLMOD library, which is highly optimized and reputable. Since it uses a supernodal factorization strategy, it is suited for relatively dense matrices. As can be seen from the results, this approach worked quite well for large matrices where the sparse solver was almost as fast as the dense solver. Another reason for using a sparse matrix representation is that despite the relatively high density, the memory requirements of the sparse version of the algorithm is significantly lower than that of the dense version, allowing us to solve larger problems on the GPU. It is worth noting however, that the densities of the largest studied problems were lower (around 0.13) than the smaller studied problems. The total solve time in Figure 4.2 seem to indicate that the performance of the sparse solver is a function of the number of non-zeros in ADA^T, rather than the number of constraints or variables. This is what we expect for a well-implemented sparse algorithm.

Another approach to this thesis would have been to try to use a GPU to accelerate an existing open-source software package implementing the predictor-corrector interior point method. Possible candidates could have been the BPMPD or CLP package.[4] This approach would probably have lead to a more practical solver, since various optimizations, file-type support and pre-solving would have been included from the beginning. However, by implementing our own solver, we gain more understanding of the code and what makes the IPM work. We are also more free to implement the GPU versions using any strategy that we can come up with instead of having to adapt the strategy to the existing code base.

5.4 Further Research

Currently, the implemented GPU solvers have only been compared to CPU counterparts and the MATLAB `linprog` solver. Both of these comparisons have been favourable for the GPU solvers. It would be interesting to compare the solvers to other GPU LP solvers, such as the matrix-free IPM by Smith et al.[39]. Since iterative solvers are often faster on the GPU and do not require as much memory, it would be an interesting comparison presumably in favour of the matrix-free IPM. Comparing it to the PCIPM solver by Gade-Nielsen et al.[13] would also be interesting.

Developing this solver into a more usable software package for solving general LP problems would require a number of improvements to be able to compete in performance with existing software. Generalizing the allowed problem structure would be necessary, as well as implementing a presolver and other optimizations such as dense column separation. The `GPUDense` version could be used for all problems that is below a certain size, with `GPUSparse` taking over when the memory requirements become too high.

As always when doing performance comparisons, the chosen benchmark problems will affect the result. Inevitably, some problems are better suited for some solution strategies. In this thesis we have used relatively homogeneous benchmark problems arising from the same context. It may be the case that these problems are better or worse suited for GPU-acceleration than other problems would have been. It would be interesting to test the implemented solvers on other unrelated benchmark problems as well.

The GPU solvers in this thesis have been optimized to such an extent that the SpMM multiplication and the Cholesky factorization consist of at least 97% of the computation time for any given problem. In order to accelerate the GPU solvers further, we believe that it is these two operations that must be optimized. CHOLMOD is being developed continuously at the time of writing this report. An improved version of CHOLMOD has potential to accelerate the `GPUSparse` solver. The library cuSOLVER from Nvidia was recently released and includes a Cholesky factorizer. The cuSOLVER Cholesky factorizer was found to perform almost as good as the more mature MAGMA solver. There may also be strategies overlooked in this thesis and different from the ones realized in `GPUSparse` and `GPUDense`, that can be used to implement a GPU-accelerated PCIPM. Apart from optimizing the implementations, we know that mathematical optimizations such as dense column separation can make a large differences in some types of LP problems. Further research is required to see if these optimizations are easily implemented in GPU versions of the algorithm.

Chapter 6

Conclusion

This thesis concerns the possibility of accelerating the predictor-corrector interior point method by using a GPU.

Studies of interior point methods lead us to the LIPSOL algorithm, presented in [44], the theoretical foundation of which is what our algorithm is based upon. CUDA was selected as the GPU programming framework, since it allows for optimized implementations tailored for specific hardware and the fact that many useful libraries are based upon it.

The CPU versions of the algorithm were developed in C++ to be used for performance comparisons, and based upon these the GPU versions were developed. Two versions were developed that only used the CPU and two corresponding versions that were accelerated with the GPU: one that that performs the Cholesky factorization where the matrix ADA^T is represented as a sparse matrix and one where it is represented as a dense matrix.

In order to answer the problem statement, a set of large and relatively sparse benchmark problems arising in financial applications was collected. The performance of the CPU and GPU versions of the algorithm was compared for these benchmark problems. We have shown that a speedup of 2x - 6x can be achieved using the GPU, despite the inherently sequential nature of the interior point method and the fact that double precision arithmetic had to be used for numerical stability. The GPU solver with the dense matrix representation was the fastest for all problems where it could be used. We have shown that the relatively small memory of the GPU imposed a limit on the problem size for which the GPU could be utilized, at least with the methods used in this thesis. Problems with between 6 000 and 23 000 constraints could be accelerated at least 4x using the dense GPU solver. The problem with 31 000 constraints required too much memory to be solved by the dense GPU solver, but could be accelerated 2x using the sparse GPU solver.

In order to achieve high accuracy in the context of the interior point methods, standard Cholesky factorizers have to be modified to compensate for numerical instabilities. One way to do this is to use the Cholesky-Infinity algorithm, described in [44]. We have shown that this modification works well with the GPU solvers, increasing the achievable accuracy without incurring a significant performance loss.

The implemented GPU solvers were also compared to the built-in `linprog` function in MATLAB. This comparison showed that `linprog` is faster for small problems, but that the implemented GPU solvers is significantly faster for large problems.

We have also shown that a speedup of 2x - 3x was possible even when the arithmetic performance of the graphics card was throttled from 1700 GFLOP/s to 213 GFLOP/s. The results from this study indicate that the arithmetic performance is not the most important factor for the sparse matrix operations.

In conclusion, we have shown that a GPU can indeed be used to accelerate the predictor-corrector interior point method for all problem sizes considered in this thesis. Our GPU implementations compare favourably to MATLAB's built-in solver `linprog`. We have also shown that the algorithm can be accelerated modestly using a GPU even if the arithmetic performance of the GPU is throttled to one eighth of its maximum speed.

References

[1] G. Baker et al. “PLAPACK: high performance through high-level abstraction”. In: *1998 International Conference on Parallel Processing, 1998. Proceedings.* 1998 International Conference on Parallel Processing, 1998. Proceedings. Aug. 1998, pp. 414–422. DOI: 10.1109/ICPP.1998.708513.

[2] M. S. Bartlett. “An Inverse Matrix Adjustment Arising in Discriminant Analysis”. In: *The Annals of Mathematical Statistics* 22.1 (Mar. 1951), pp. 107–111. ISSN: 0003-4851, 2168-8990. DOI: 10.1214/aoms/1177729698. URL: http://projecteuclid.org/euclid.aoms/1177729698 (visited on 02/18/2015).

[3] Nathan Bell and Michael Garland. *Efficient sparse matrix-vector multiplication on CUDA*. Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008. URL: http://sbel.wisc.edu/Courses/ME964/Literature/techReportGarlandBell.pdf (visited on 01/28/2015).

[4] Hande Y. Benson. “Interior-Point Linear Programming Solvers”. In: *Wiley Encyclopedia of Operations Research and Management Science* (2009). URL: http://onlinelibrary.wiley.com/doi/10.1002/9780470400531.eorms0416/full (visited on 04/16/2015).

[5] *blaze-lib - A high performance C++ math library - Google Project Hosting.* URL: https://code.google.com/p/blaze-lib/ (visited on 03/18/2015).

[6] Michael Boyer. *CUDA Memory Transfer Overhead.* URL: http://www.cs.virginia.edu/~mwb7w/cuda_support/memory_transfer_overhead.html (visited on 04/24/2015).

[7] Yanqing Chen et al. “Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate”. In: *ACM Trans. Math. Softw.* 35.3 (Oct. 2008), 22:1–22:14. ISSN: 0098-3500. DOI: 10.1145/1391989.1391995. URL: http://doi.acm.org.focus.lib.kth.se/10.1145/1391989.1391995 (visited on 03/02/2015).

[8] *<chrono> - C++ Reference.* URL: http://www.cplusplus.com/reference/chrono/ (visited on 04/28/2015).

[9] Timothy A. Davis. *User Guide for CHOLMOD: a sparse Cholesky factorization and modification package.* Mar. 26, 2014. URL: `https://www.cise.ufl.edu/research/sparse/cholmod/CHOLMOD/Doc/UserGuide.pdf` (visited on 03/02/2015).

[10] *Eigen.* URL: `http://eigen.tuxfamily.org/index.php?title=Main_Page` (visited on 02/06/2015).

[11] Gerd Eriksson. *Numeriska Algoritmer med MATLAB.* June 2008.

[12] Nicolai Fog Gade-Nielsen. *Interior Point Methods on GPU with application to Model Predictive Control.* Ed. by Bernd Dammann and John Bagterp Jørgensen. Red. by Bernd Dammann and John Bagterp Jørgensen. DTU Compute PHD-2014. Technical University of Denmark, 2014.

[13] Nicolai Fog Gade-Nielsen, John Bagterp Jørgensen, and Bernd Dammann. "MPC Toolbox with GPU Accelerated Optimization Algorithms". In: *10th European Workshop on Advanced Control and Diagnosis.* 2012. URL: `http://orbit.dtu.dk/fedora/objects/orbit:115906/datastreams/file_d0b0f644-7f92-4e8e-971f-8702a2962a6e/content` (visited on 01/23/2015).

[14] A. George and J. Liu. "The Evolution of the Minimum Degree Ordering Algorithm". In: *SIAM Review* 31.1 (Mar. 1, 1989), pp. 1–19. ISSN: 0036-1445. DOI: `10.1137/1031001`. URL: `http://epubs.siam.org/doi/abs/10.1137/1031001` (visited on 02/17/2015).

[15] Jacek Gondzio. "Matrix-free interior point method". In: *Computational Optimization and Applications* 51.2 (Oct. 14, 2010), pp. 457–480. ISSN: 0926-6003, 1573-2894. DOI: `10.1007/s10589-010-9361-3`. URL: `http://link.springer.com/article/10.1007/s10589-010-9361-3` (visited on 01/26/2015).

[16] Nicholas I. M. Gould, Jennifer A. Scott, and Yifan Hu. "A Numerical Evaluation of Sparse Direct Solvers for the Solution of Large Sparse Symmetric Linear Systems of Equations". In: *ACM Trans. Math. Softw.* 33.2 (June 2007). ISSN: 0098-3500. DOI: `10.1145/1236463.1236465`. URL: `http://doi.acm.org.focus.lib.kth.se/10.1145/1236463.1236465` (visited on 03/18/2015).

[17] D. den Hertog. *Interior Point Approach to Linear, Quadratic and Convex Programming.* Vol. 277. Dordrecht: Kluwer Academic Publishers, 1994.

[18] K. Iglberger et al. "High performance smart expression template math libraries". In: *2012 International Conference on High Performance Computing and Simulation (HPCS).* 2012 International Conference on High Performance Computing and Simulation (HPCS). July 2012, pp. 367–373. DOI: `10.1109/HPCSim.2012.6266939`.

[19] ILOG. *Preprocessing: Presolver and Aggregator.* URL: `http://www-eio.upc.es/lceio/manuals/cplex75/doc/usermanccpp/html/solveLPS9.html` (visited on 04/27/2015).

[20] Jin Hyuk Jung and Dianne P. O'Leary. "Implementing an interior point method for linear programs on a CPU-GPU system". In: *Electronic Transactions on Numerical Analysis* 28 (2008), pp. 174–189. URL: `http://www.emis.ams.org/journals/ETNA/vol.28.2007-2008/pp174-189.dir/pp174-189.pdf` (visited on 01/23/2015).

[21] N. Karmarkar. "A new polynomial-time algorithm for linear programming". In: *Combinatorica* 4.4 (Dec. 1, 1984), pp. 373–395. ISSN: 0209-9683, 1439-6912. DOI: `10.1007/BF02579150`. URL: `http://link.springer.com/article/10.1007/BF02579150` (visited on 01/23/2015).

[22] Victor W. Lee et al. "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU". In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA '10. New York, NY, USA: ACM, 2010, pp. 451–460. ISBN: 978-1-4503-0053-7. DOI: `10.1145/1815961.1816021`. URL: `http://doi.acm.org.focus.lib.kth.se/10.1145/1815961.1816021` (visited on 04/08/2015).

[23] David G. Luenberger and Yinju Ye. *Linear and Nonlinear Programming*. 3rd ed. New York: Springer, 2008.

[24] Irvin J. Lustig, Roy E. Marsten, and David F. Shanno. "On implementing Mehrotra's predictor-corrector interior-point method for linear programming". In: *SIAM Journal on Optimization* 2.3 (1992), pp. 435–449. URL: `http://epubs.siam.org/doi/abs/10.1137/0802022` (visited on 01/23/2015).

[25] *MAGMA*. URL: `http://icl.utk.edu/magma/` (visited on 03/02/2015).

[26] MathWorks. *Choosing a Solver*. URL: `http://se.mathworks.com/help/optim/ug/choosing-a-solver.html` (visited on 04/24/2015).

[27] MathWorks. *Linear Programming Algorithms*. URL: `http://se.mathworks.com/help/optim/ug/linear-programming-algorithms.html#brnpenw` (visited on 02/03/2015).

[28] MathWorks. *Solve linear programming problems - MATLAB linprog*. URL: `http://se.mathworks.com/help/optim/ug/linprog.html` (visited on 04/09/2015).

[29] S. Mehrotra. "On the Implementation of a Primal-Dual Interior Point Method". In: *SIAM Journal on Optimization* 2.4 (Nov. 1, 1992), pp. 575–601. ISSN: 1052-6234. DOI: `10.1137/0802028`. URL: `http://epubs.siam.org/doi/abs/10.1137/0802028` (visited on 01/23/2015).

[30] Stephen G. Nash and Ariela Sofer. *Linear and Nonlinear Programming*. Singapore: McGraw-Hill, 1996.

[31] Nvidia. *CUDA C Programming Guide*. URL: `http://docs.nvidia.com/cuda/cuda-c-programming-guide/` (visited on 01/26/2015).

[32] Nvidia. *cuSOLVER*. Jan. 12, 2015. URL: `https://developer.nvidia.com/cusolver` (visited on 04/02/2015).

[33] Nvidia. *cuSPARSE Reference Guide*. URL: http://docs.nvidia.com/cuda/cusparse/ (visited on 01/29/2015).

[34] Nvidia. *Developing with CUDA*. URL: http://www.nvidia.com/object/cuda-parallel-computing-platform.html (visited on 01/26/2015).

[35] Nvidia. *Nvidia Professional Graphics Solutions*. 2013. URL: http://www.nvidia.com/content/PDF/line_card/6660-nv-prographicssolutions-linecard-july13-final-lr.pdf (visited on 01/28/2015).

[36] J.D. Owens et al. "GPU Computing". In: *Proceedings of the IEEE* 96.5 (May 2008), pp. 879–899. ISSN: 0018-9219. DOI: 10.1109/JPROC.2008.917757.

[37] Steve Rennich, Tim Davis, and Philippe Vandermersch. *GPU Acceleration of Spare Matrix Factorization in CHOLMOD*. URL: http://on-demand.gputechconf.com/gtc/2014/presentations/S4201-gpu-acceleration-sparse-matrix-factorization-cholmod.pdf (visited on 01/27/2015).

[38] Sean Rose. "GPU Sparse Matrix Multiplication with CUDA". In: (2013). URL: https://sait.fsu.edu/research/projects/rose_report.pdf (visited on 01/28/2015).

[39] Edmund Smith, Jacek Gondzio, and Julian Hall. "GPU acceleration of the matrix-free interior point method". In: *Parallel Processing and Applied Mathematics*. Springer, 2012, pp. 681–689. URL: http://link.springer.com/chapter/10.1007/978-3-642-31464-3_69 (visited on 01/23/2015).

[40] D.G. Spampinato and A.C. Elster. "Linear optimization on modern GPUs". In: *IEEE International Symposium on Parallel Distributed Processing, 2009. IPDPS 2009*. IEEE International Symposium on Parallel Distributed Processing, 2009. IPDPS 2009. May 2009, pp. 1–8. DOI: 10.1109/IPDPS.2009.5161106.

[41] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra*. Philadelphia: SIAM, June 1, 1997. 373 pp. ISBN: 9780898713619.

[42] Richard Vuduc et al. "On the Limits of GPU Acceleration". In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*. HotPar'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 13–13. URL: http://dl.acm.org/citation.cfm?id=1863086.1863099 (visited on 01/26/2015).

[43] Stephen J. Wright. *Primal-Dual Interior-Point Methods*. SIAM, Jan. 1997. 316 pp. ISBN: 9780898713824.

[44] Yin Zhang. *Solving Large-Scale Linear Programs by Interior-Point Methods Under the MATLAB Environment*. Optimization Methods and Software, 1996.

Appendix A

Data Format

An example of the file format used in this thesis for describing an LP problem looks like

```
2 3
Maximize
 obj: 2 x#0 + 1.5 x#1
Subject To
 c#0: 12 x#0 + 24 x#1 >= 120
 c#1: 16 x#0 + 16 x#1 >= 90
 c#2: 30 x#0 + 12 x#1 >= 120
Bounds
 0 <= x#0 <= 15
 0 <= x#1 <= 20
End
```

The file begins with the number of variables and number of constraints followed by either `Minimize` or `Maximize`, after which the objective function is entered. Then, after the line `Subject To`, the constraints are entered in the form shown above. The constraints may be either of $<=, >=, =$. In the section `Bounds`, the upper and lower variable bounds are entered. The file ends with the keyword `End`.

TRITA -MAT-E 2015:37
ISRN -KTH/MAT/E--15/37--SE