NTNU

Norwegian University of
Science and Technology

TDT4590 - Complex Computer Systems, Specialization Project

Linear Optimization with CUDA

Daniele Giuseppe Spampinato
daniele.spampinato@gmail.com

Department of Computer and Information Science
Norwegian University of Science and Technology, Trondheim
(Norway)

January 2009

Supervisor
Dr. Anne C. Elster

Preface

This report is the result of a project assigned by the course *TDT4590 - Complex Computer Systems, Specialization Project*, during the fall semester 2008. The course gave the possibility to expand my knowledge and interest for the new trends in high performance computing.
I would like to thank Dr. Anne C. Elster for her high spirit and continuos support. A special thanks also to all the members of the HPC-Lab. For all the important experiences we shared, and for the friendly and valuable help they always gave me during the entire project development.

Trondheim, January 2009

Daniele Giuseppe Spampinato

Abstract

Optimization is an increasingly important task in many different areas, such as finance and engineering. Typical real problems involve several hundreds of variables, and are subject to as many constraints. Several methods have been developed trying to reduce the theoretical time complexity. Nevertheless, when problems exceed reasonable dimensions they end up in a huge computational power requirement.
Heterogeneous systems composed by coupling commodity CPUs and GPUs turn out to be relatively cheap, highly performing systems. Recent evolution of GPGPU technologies give even more powerful control over them.
This project aims at developing a parallel, GPU-supported version of an established algorithm for solving linear programming problems. The algorithm is selected among those suited for GPUs. A serial version based on the same solving model is also developed, and experimental results are compared in terms of performance, and precision.
Experimental outcomes confirm that the application written with CUDA leverages the GPU's huge number of cores, to solve with an appreciable preciseness, big problems with thousands of variables and constraints between 2 and 2.5 times faster than the serial version. This result is also important from a dimensional point of view. Previous attempts to solve linear programming problems with a GPU were constrained to a few hundreds of variables; a barrier definitively broken by the present solution.

Contents

Preface i

Abstract ii

Table of Contents iii

1 Introduction 1
- 1.1 Project Goals . . . 1
- 1.2 Report Outline . . . 2

2 Graphics Processing Unit 3
- 2.1 The NVIDIA Landscape . . . 4
- 2.2 The Tesla Architecture Processing Model . . . 6
  - 2.2.1 Streaming Multiprocessors . . . 7
  - 2.2.2 GPU Memories . . . 7
  - 2.2.3 The SIMT Paradigm . . . 8
- 2.3 The CUDA Programming Model . . . 9
  - 2.3.1 A Heterogeneous Programming Model . . . 10
  - 2.3.2 The CUDA Software Stack . . . 10
  - 2.3.3 Threads Organization . . . 12
  - 2.3.4 Memory Organization . . . 13
  - 2.3.5 Mapping to the Tesla Architecture . . . 14
  - 2.3.6 Compute Capability . . . 15

3 Linear Programming 16
- 3.1 Linear Programming Model . . . 17
  - 3.1.1 Geometric Interpretation . . . 20
  - 3.1.2 Duality Theory . . . 22
- 3.2 Solving Linear Programming Problems . . . 23
  - 3.2.1 Simplex-Based Methods . . . 23
  - 3.2.2 Interior Point Methods . . . 27
- 3.3 Complexity Aspects . . . 31

4 Linear Programming in CUDA 33
- 4.1 Method Selection . . . 33
- 4.2 Implementation Strategy . . . 36
  - 4.2.1 Data Structures . . . 36
  - 4.2.2 Kernels Configuration . . . 37

4.2.3 Non-Algebraic Routines: Computing Entering Variable 37
4.2.4 Non-Algebraic Routines: Computing Leaving Variable 38
4.2.5 Non-Algebraic Routines: Computing $\mathbf{B^{-1}}$ 39

5 Experimental Results **40**
5.1 The Experimental Environment 40
5.2 Methodology 40
5.2.1 Analysis Objectives 41
5.2.2 Tools 41
5.3 Results Elicitation and Analysis 42
5.3.1 Speedup Analysis 44
5.3.2 A Few Reflections 46

6 Conclusions and Future Work **48**
6.1 Conclusions 49
6.2 Future Work 50

Bibliography **51**

A Linear Programming Solvers **53**
A.1 Serial Version 53
A.1.1 Main Module: lpsolver.c 53
A.1.2 liblp.c 60
A.1.3 matman.c 62
A.2 CUDA Version 63
A.2.1 Main Module: culpsolver.cpp 63
A.2.2 culiblp.cu 68
A.2.3 cumatman.cu 84

B Tools **86**
B.1 popmat.c 86
B.2 matgen.py 87
B.3 clock.py 88

Chapter 1

Introduction

The past century has certainly been characterized by the computer's revolution. Computer systems evolve very quickly. Recently the ability to provide speed has gone incredibly beyond what one may have thought just ten years ago.
In addition, parallel systems now appear as the key answer to leap over the brick wall of serial performance [15]. Graphics Processing Units (GPUs) are considered today one of the most affordable computing solutions to speedup computationally demanding applications, offering performance peaks that introduced the teraflop era.
The present decade has been defined by Blythe [7] as the programmability and ubiquity decade for graphics devices, and, as a matter of fact, programming general purpose applications on a GPU (also known as GPGPU) is one of the most discussed topics today.
The GPGPU paradigm opens new frontiers especially in scientific computing, where there is a tremendous speed requirement. Having at disposal such a computational power together with an easier approach to control it, appears as a winning combination to get high performance with less effort at a cheaper price.
Many scientific fields have been supported with great success by GPUs [7, 24]. We selected linear optimization, a topic that has been methodically studied by operational researchers during the last 70 years. Several interesting theoretical models have been developed, and some of them fit quite well to GPUs.

1.1 Project Goals

This project aims at developing a parallel, GPU-supported version of an established algorithm for solving linear programming problems. We will implement and properly evaluate a serial and a parallel GPU-based version of the predefined application. We want to see how much it is possible to gain in performance writing code based on a solving approach still compatible with serial programming models. The algorithm will be selected so that it presents features that make it suitable for being executed on a GPU.
Our decision has been led mainly by the fact that optimization is an increasingly important task in many different areas, such as finance and engineering. Moreover, literature does not cover extensively such a topic in the light of the most recent tools

for GPGPU.
To our knowledge [10, 12], the latest studies are still based on the old GPGPU programming methodology, where the graphics pipeline is coopted to perform general purpose computation. Such a practice requires applications to be designed taking into account graphics aspects, and programmed using graphics API.
New GPGPU tools and techniques overcome such limitations. NVIDIA CUDA is a programming environment that provides developers with a new high-level programming model that allows to take full advantage of the GPUs powerful hardware, enabling a larger productivity of complex solutions.

1.2 Report Outline

Chapter 2 deals with the technological background, introducing the recent GPU technologies that will support our study. We will focus on NVIDIA's recent technologies, describing in particular the Tesla architecture processing model and the CUDA programming model.

Chapter 3 is about linear programming. It provides with models and methods to solve linear optimization problems. We will give the mathematical definition of a linear programming problem, underlining its geometrical interpretation. This will help the understanding of two important classes of solving methods: simplex-based and interior point methods. Consequences of linear programming P-completeness are also discussed.

Chapter 4 is about practical aspects of the implementation process. In this chapter we merge the knowledge coming from the two previous chapters in order to design the development of both a sequential and a parallel version of a LP solver, enlightening its most relevant features.

Chapter 5 presents the results of the comparison between the two different implementations given in Chapter 4.

Chapter 6 summarizes the project conclusions and suggests some future work.

Chapter 2

Graphics Processing Unit

For the last generation, terms like video games, 3D-acceleration and animation, video rendering and many others concerning image processing tasks, are getting more and more common. If we investigate what they have in common, we will hit on another very recent and quite interesting word: GPU. GPU is the acronym for Graphics Processing Unit. Today, not only young people refer often to such concepts, and ultimately to the one of GPU. Graphics processing is adopted in a wide range of different areas, from physics simulation to graphic arts. GPUs are used to play video games on a home PC as well as to run complex astrophysics simulations in a spatial laboratory. Market trends confirm the growing interest for GPUs. A recent study in graphics and multimedia[1][25], affirms that the percentage increase in shipments in the last third quarter of 2008 is the highest in the last six years. It reports that 111 million GPUs were shipped during the quarter just mentioned in comparison to the 94 million of the previous quarter. Table 2.1 summarizes the quarter-to-quarter growth rates from 2001 to 2008.

	8 year average	2001	2002	2003	2004	2005	2006	2007	2008
% growth from Q2 to Q3	12.30%	−0.48%	18.62%	16.07%	16.20%	11.59%	12.52%	11.58%	17.84%

Table 2.1: Growth rates from quarter 2 to quarter 3 from 2001 to 2008.
(Source: *Jon Peddie Research Press Release*)

Graphics hardware is now about 40 years old. It was initially developed to support computer-aided design (CAD) and flight simulation. A good description of the evolution of graphics hardware with different references to the literature may be found in Blythe [7]. The last generation of GPUs consists of highly parallel, multithreaded, many-core processors. Their huge number of streaming processors are perfectly suited for fine-grained, data-parallel workloads, consisting of thousands of

[1] Jon Peddie Research is a technically oriented multimedia and graphics research and consulting firm. It produces quarterly reports focused on the market activity of graphics hardware for desktop and notebook computing.

independent threads executing vertex, geometry, and pixel-shader program routines concurrently. It's worth highlighting, by the way, that GPU computing is not just employed for tasks that require graphics strictly speaking. We can find a number of applications that do not necessarily involve the use of graphics concepts where GPUs have successfully been utilized. Just as an example, Blythe [7] and Owens et al. [24] report about promising results in linear algebra, database management, and financial services. Normally, when referring to such non-graphics employments of the GPU, it is typical to use the expression *General-Purpose computation on GPUs*, or shortly GPGPU[2]. Notable is the deep interest and effort that the High-Productivity Computing (HPC[3]) community is putting into GPGPU. Of course we cannot think to efficiently exploit a GPU's capabilities to run every possible application. Even though we talked about general purpose programming, it does not mean that the GPUs are now designed as general purpose processors. In Owens et al. [24] we can find a very clear description of the characteristics that an application must feature to successfully map onto a GPU. In particular, it has to be an application with large computational requirements and high parallelizability, where the throughput is more important than latency.

The present chapter aims at describing what a GPU is and what allows us to use it for the purpose of our project. To enter into the specific context of our project, in Section 2.1 we show the current NVIDIA product portfolio. The last two sections are about the new NVIDIA Tesla architecture processing model (Section 2.2), and the CUDA programming model (Section 2.3).

2.1 The NVIDIA Landscape

NVIDIA Corporation[4] (Figure 2.1) is a multinational company specialized in the manufacture of graphics processors targeted at different use levels, such as servers, workstations, desktop computers, and mobile devices. NVIDIA is among the major vendors of graphics technologies together with Intel and AMD. Peddie Research [25] shows that in the third quarter of 2008, NVIDIA was the second major supplier of desktop graphics devices with a market share of 27.8%, second just to Intel who led with a market share of 49.4%.

Since November 2006, NVIDIA's GPUs are based on a new graphics and computing architecture, namely the Tesla architecture, and since February 2007 they are provided with the CUDA C programming environment to simplify many-core programming. The Tesla architecture and the CUDA environment may result in a winning combination. In the field of HPC a relevant part of the developers are scientists belonging to different branches, like mathematicians, physicists, biologists, and engineers. What most of them want are tools with a fast learning curve, easy and ready to be used efficiently. So the reason we consider the combination described above as a winning one, is that it tries to fill the gap between the programming model and the machine's processing model. This is in general an important require-

[2] http://www.gpgpu.org
[3] http://www.hpcwire.com
[4] http://www.nvidia.com

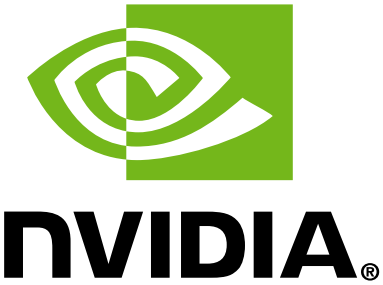

Figure 2.1: NVIDIA logo.

ment for high-performance computation, because it helps non-expert users avoid understanding nontrivial transformations between the model used to program the system (e.g. a CPU-GPU system), and the model that describes how the machine actually computes [17].
NVIDIA mainly releases three categories of GPUs, targeting the fields where typically a very high demand of intensive computation is present: gaming, professional graphics processing, and High-Performance Computing.

GeForce Family

The GeForce GPUs are designed for gaming applications. The last GeForce series is the GeForce GTX 200. The top-model card within the series is the GeForce GTX 280[5]. It presents 240 streaming processor cores each working at 1.296 GHz. The card has a dedicated memory of 1GB connected by a 512-bit GDDR3 interface with a bandwidth of 141.7 GB/s. Since the main target of these cards is the video games market, an interesting measure is the speed with which a particular card can perform texture mapping, called *texture fill rate*. It basically expresses the number of textured pixels that the GPU can render every second. For the GTX 280, for instance, the texture fill rate is 48.2 billion/s.

Quadro Family

The Quadro technology features an architecture delivering optimized CAD, DCC (Digital Content Creation), and visualization application performance. At a glance it may seem that the Quadro and GeForce families of GPUs are identical. In a way this is true. Many of the Quadro cards use the same chipset as the GeForce cards. Just to emphasize, it is even possible, in some special conditions, to *soft-mod* (i.e. modify in software) some GeForce cards to allow the system to emulate Quadro cards[6]. But there is a critical detail that makes the difference between the two families. Normally, video game-oriented systems provide a very high throughput, because of the relevant need for speed. Such an objective is achieved by texturing, shading or rendering just until an approximated, sub-optimal result. Quadro GPUs conversely, are designed to complete rendering operations with larger detail. This

[5] http://www.nvidia.com/object/geforce_gtx_280.html
[6] http://www.techarp.com/showarticle.aspx?artno=539

makes such cards useful in animation or audio/video editing for example. The latest Quadro FX 5600 is able to provide up to 76.8 GB/s [7].

Tesla Family

GPUs in the Tesla family of graphics processors are specifically intended for HPC applications. Cards belonging to this family offer high computational power and bigger dedicated memory. As an example, the recent C1060 [8] features 240 streaming processors with a core frequency of 1.296 GHz, similarly to the GeForce GTX 280, and it is able to perform 933 GFlops. Cards are equipped with 4 GB of dedicated memory with a bandwidth of 102 GB/s. All the Tesla GPUs lack a direct connection to display devices. The latter is a confirmation that the Tesla cards are not meant to be graphics-oriented, but rather a valid support for the HPC community. Normally, GPUs dedicate some of the global memory to the so-called *primary-surface*, which is used to refresh the display output. As an example, for a display with resolution 1600x1200 and 32-bit bit depth, the amount of memory allocated on the GPU's memory is 7.68 MB. If the resolution or the bit depth change increasing the memory requirements, the system may have to cannibalize memory allocated to other applications. In case of a CUDA application for instance, this may mean its crash. Tesla GPUs instead, removing direct interaction with graphics, avoid such side-effects that could disrupt the running of huge, critical computations.
As a last remark, we can say that with the Tesla family of GPUs, NVIDIA has broken the TFlops wall remaining within reasonable physical dimensions. The Tesla S1070 [9] is a system containing four Tesla processors, and as a consequence a total of 960 computing cores, performing about 4 TFlops. It is moreover interfaced to 16 GB of high-speed memory. The system is by the way relatively small, measuring 4x44x72 cm^3 with a weight of 15 kg. Good use of a system like this may even outperform a huge, cumbersome cluster for certain applications.

2.2 The Tesla Architecture Processing Model

We already introduced the Tesla architecture as the new ground for NVIDIA's recent graphics cards. It is now time to deepen some aspects related to its *processing model*. Before starting we think it is important to clarify a few very important concepts that may otherwise raise confusion. When we use the terms processing model and programming model, we refer to the definitions given in McCool [17]. The programming model is an abstract model used by programmers when implementing a piece of code, to reason out how to organize the computation. A processing model on the other hand describes how a physical machine computes. In a way we can say that the programming model is exposed by the programming language environment, while the processing model by the architecture vendor specification. The goal of the programmer is to map effectively and efficiently the application model to the programming model. The goal of the compiler is to effectively and efficiently translate

[7] http://www.nvidia.com/object/quadro_fx_5600_4600.html
[8] http://www.nvidia.com/object/tesla_c1060.html
[9] http://www.nvidia.com/object/tesla_s1070.html

the computation expressed by the programmer into the processing model used by the target hardware. Figure 2.2 summarizes the concepts developed so far. Indeed, in this section we use the terms architecture and processing model to refer to the same concept.
The Tesla architecture is built around some basic, important elements that, combined together, give rise to a model of the underlying processing unit. In the official CUDA Programming Guide[23] the architecture is briefly but completely described in the following way: a set of SIMT multiprocessors with on-chip shared memory. In fact, it is a very clear picture. We will now describe the architecture, focusing on each of its components and on the concept of SIMT.

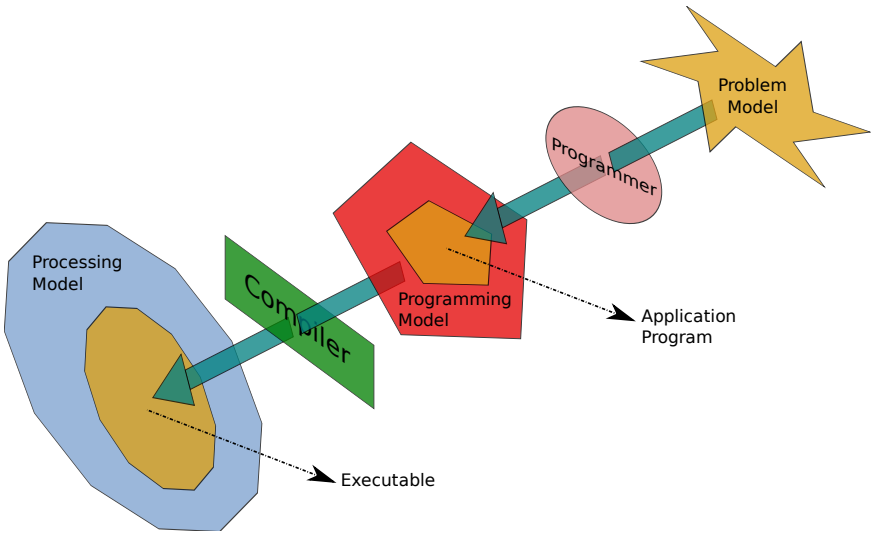

Figure 2.2: The Problem model at different levels.

2.2.1 Streaming Multiprocessors

The Tesla Architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SMs). A multiprocessor consists of eight Scalar Processor cores (SPs), two special function units for transcendentals, a Multithreaded Instruction Unit (M-IU), and on-chip memory. The SM creates, manages, and executes concurrent threads in hardware with zero scheduling overhead. This is an important factor to allow very fine-grained decomposition of problems by assigning one thread to each data element, for instance.

2.2.2 GPU Memories

On a GPU we can localize two distinct kind of memories: on-chip and device memory.
Each SM has on-chip memory of the following types:

- one set of local 32-bit *registers* per SP;
- a shared memory that is shared by all SPs with access times comparable to a L1-cache on a traditional CPU;

- a read-only *constant cache* that is shared by all SPs that speeds up reads from the constant read-only region of device memory;
- a read-only *texture cache* that is shared by all SPs and speeds up reads from the texture read-only region of device memory. The Texture cache is accessed via the *texture unit*, that implements several operations as described in the CUDA Programming Guide [23].

Device memory is a high-speed DRAM memory with higher latency than on-chip memory (typically hundreds of times slower). Device memory is subdivided in several regions. The most important memory spaces there allocated are global, local, constant and texture memory. Global and local memory spaces are read-write, not cached areas, while constant and texture memory spaces are read-only and cached. A couple of comments concerning the memory terminology. Device and global memory are at this point clearly not synonyms. Global is used to underscore the access pattern to the memory area. It is a part of the device memory of the GPU. Similarly, local and shared memory are not the same concept. First, they belong to different GPU components. Local memory is part of the device memory (slow) while shared memory is on-chip (fast). Local memory is used by the compiler to keep anything the developers consider local to the thread but does not fit in faster memory for some reason, such as no more registers available or arrays too big to fit in registers [9].

2.2.3 The SIMT Paradigm

SIMT (Single Instruction, Multiple Threads) is a new paradigm introduced to manage properly the big amount of threads executable on a Tesla-based GPU. Threads are grouped in *warps* and mapped to the SPs. One thread is mapped to one SP. Warps are created, managed, scheduled, and executed by the multiprocessor M-IU. The number of threads per warp is 32. It is very common to talk about half-warp since it is in facts the granularity unit for threads execution. An half-warp is either the first or the second half of a warp. Threads that belong to the same warp start together at the same program address, maintaining a total freedom to branch and execute independently. Every instruction issue time, the M-IU selects a warp that is ready to execute and issues the next instruction to the active threads of the warp. A warp executes one common instruction at a time, so full efficiency is realized when all threads in a warp agree on their execution path. However threads are free to execute differently, but when this happens performance risks to be seriously injured. The reason is that when there is disagreement among the threads, the threads' scheduler has to serialize their execution. When all the independent paths have been taken, the threads converge back to the same execution path. Of course, since warps are executed independently serialization is a problem that might occur within a warp. Branch divergences are not the only reason that may lead to threads serialization. If an instruction, regardless of its atomicity, executed by a warp writes to same location in global or shared memory for more than one thread in the warp, writes are serialized as well. Then, depending whether the instruction is an atomic or non-atomic one, different warranties are granted about which write will succeed.

2.3 The CUDA Programming Model

Recent programming models for GPUs are the results of the evolution of the well-known graphics pipeline. GPGPU developers were used to face with such a model when implementing their solutions. There are plenty of descriptions of the graphics pipeline in the literature, some recent attempts are in Blythe [7], Akenine-Möller and Ström [4], and Owens et al. [24]. Basically we can describe the graphics pipeline as a directed flow of data between its input and its output. The input is a set of triangles which vertices are processed in the first stage of the pipeline, the vertex processor, applying required transformation such as rotations and translations. Then the raster stage converts the results from the vertex processing to a collection of pixel fragments by sampling the triangles over a specified grid. Such fragments are then computed by the fragment processor which major task is to compute the color of the several fragments related to each pixel. Texturing, when required, is applied at this point. Finally the framebuffer is in charge to determine the final color of each pixel in the final image usually by keeping the closest fragment to the camera for each pixel location. At the beginning the pipeline as been conceived as a rigid structure, where the different stages were implemented as fixed functions. Very soon GPU designers realized that allowing more flexibility would have open the possibility to develop more complex effects. In this scenario the pipeline assumed a new connotation, and since the present decade it allows programmability at different level of the processing flow. Programs running on the GPU are often called *shaders*, and they are written with shader languages, generally an extension of traditional programming languages in order to support vertices and fragments and the interface to the pipeline stages (e.g. Cg [16]). Figure 2.3 depicts a typical graphics pipeline with shader programs for vertex and fragments processing.

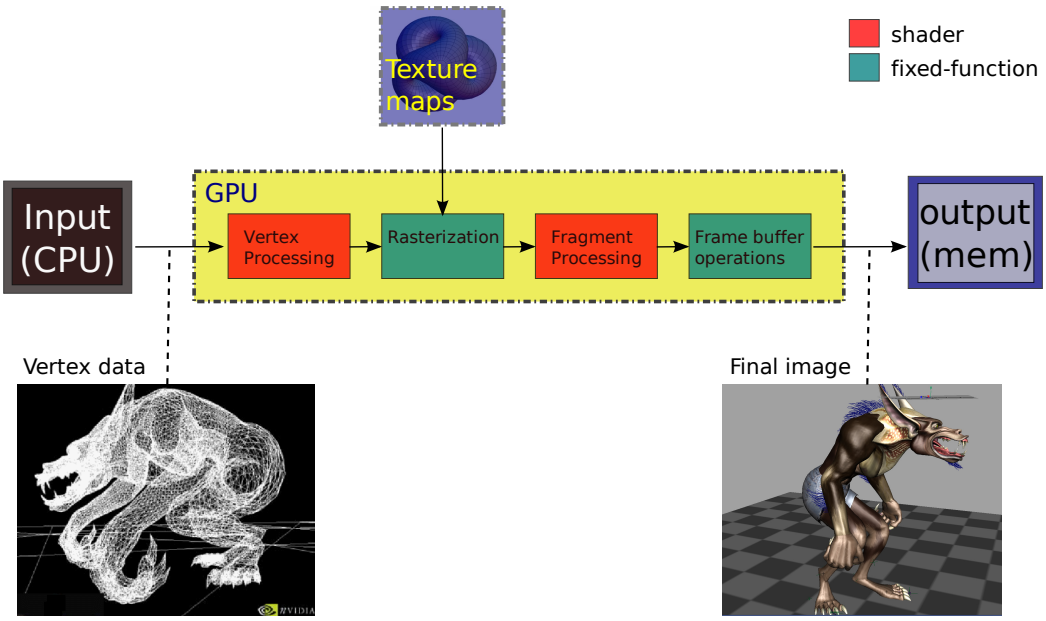

Figure 2.3: The graphics pipeline.

The next step towards a modern GPU approach was the advent of a *unified shader architecture*. With the programmable pipeline described above, developers

could take better advantage of the space repartition of the GPU resources, but still with an unpleasant disadvantage: load balancing. In that context, the slowest stages burden the whole performance. In a unified shader architecture we find several shader cores able to operate at every level of the old pipeline model. Each unified shader core can execute any type of shader and forward the result to another shader core (itself included), until the entire chain of shaders has been executed. The use of unified shader cores allows to allocate resources in a smarter way depending on the specific application, well managing load balancing. Akenine-Möller and Ström [4] well describe the benefit with a font rendering example, where an application that has to deal with detailed characters composed of tiny triangles with simple lighting, may devote more shader units for vertex processing and fewer per-pixel processing.
The CUDA (Compute Unified Device Architecture)[10] environment presents a cutting-edge programming model well-suited for modern GPUs architectures [23]. NVIDIA developed this programming environment to fit to the processing model of the Tesla Architecture, so to expose the parallel capabilities of GPUs to the developers.
Again is not our intention to explain everything about CUDA. The purpose here is to describe what is important to the project development. For a more comprehensive description refer to the CUDA programming Guide [23].

2.3.1 A Heterogeneous Programming Model

CUDA maintains a separated view of the two main actors involved in the computation, namely the *host* and the *device*. The host is the one that executes the main program, while the device is the coprocessor. A typical scenario sees a CPU as the host and a GPU as the coprocessor, but in general CUDA abstractions may be useful for programming other kinds of parallel systems [21]. Normally CUDA programs contain some pieces of code where intensive computation is required as shown in Figure 2.4. Such pieces of code are encapsulated in what is called a *kernel* and send to the device to be computed. From a memory viewpoint, CUDA assumes the existence of different memories, having host and device maintaining their own DRAM. Not accidentally the two memories are called host memory and device memory.

2.3.2 The CUDA Software Stack

Figure 2.5 illustrates the CUDA software stack showing the main layers it is composed of. A CUDA application basically lies on three main entities: the CUDA libraries, the CUDA Runtime API, and the CUDA Driver API.

The CUDA libraries contain efficient mathematical routines of common usage. One of them has to do with linear algebra, and so it will be further deepened. The other two layers consist of a low-level API, namely Driver API, and a high-level API, namely Runtime API, that is implemented on top of the first one.
Both the APIs provides programmers with the tools to proper manage threads, memory, and kernel invocation on the device. Just the Runtime API is an easier interface than the Driver API. The Runtime API provides a compact and intuitive

[10] http://www.nvidia.com/cuda

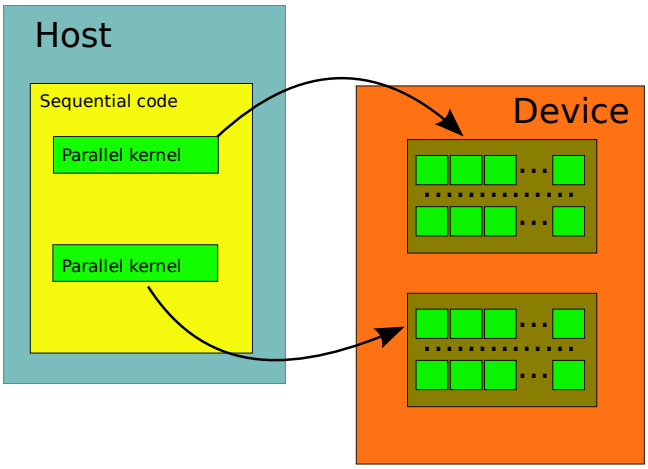

Figure 2.4: Heterogeneous programming paradigm of execution. The host executes the sequential code until a kernel invocation requires the computation to be moved onto the device to be run in parallel.

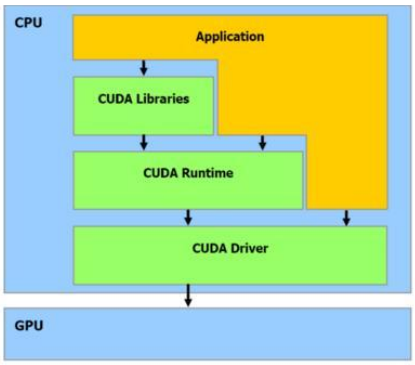

Figure 2.5: The CUDA software stack.(Source: *NVIDIA* [23])

way to deal with the concepts required to interface to the device. On the other hand the Driver API is harder to program and debug, but offers a better control, and is language-independent since it deals with cubin objects (i.e. object code of kernels written externally). Since these APIs do the same job just at different level, their use is mutually exclusive. However just looking at few examples of programs written using the Runtime API and the Driver API is enough to convince yourself that using the Driver API is not the best decision unless necessary.
In the rest of the report every reference to kernel, thread, and memory management as well as any other possible interface to the device is intended through the Runtime API.

The CUDA Linear Algebra Library

CUBLAS (Compute Unified Basic Linear Algebra Subprograms) [22] is an implementation of BLAS [1] on top of the CUDA Runtime API. It comes has a self-contained library that does not require any direct interaction at the driver or Runtime level. So it means that calling the routines the developer will exploit the GPU to manipulate matrices and vectors without caring about many of the details we

will discuss later on in this section like kernel configuration.
Even though it does not have to deal directly with the GPU programming, something about that transpires. Indeed, the model behind the library appears kind of wrapper for the heterogeneous paradigm we just described. That is, the developer must always go on to create and fill matrices and vectors in device memory, call the required sequence of CUBLAS routines, and finally download the results in host memory.
CUBLAS presents two main groups of functions: helper functions and BLAS routines. Helper functions assist in managing data in device memory. It provides functions for creating and destroying objects, as well as for moving data to or from GPU memory space.
CUBLAS BLAS routines are organized for maximum compatibility with the usual BLAS library calls. For this reason BLAS functions are organized in three levels (i.e. vector-vector, matrix-vector, and matrix-matrix operations), and the name convention totally recall the BLAS one.
Furthermore, always for compatibility purposes, CUBLAS uses column-major storage with 1-based indexing. This has to be carefully taken into account in order to not produce confusing, wrong results.

2.3.3 Threads Organization

Computational science applications have often to represent and compute real-world changes, such as weather and climate forecasting, galaxies evolutions, fluid flows, protein folding, and so on. Applications such as those just mentioned are typically based on numerical methods that may require to sample their domains in order to have a number of discrete data to elaborate.
The CUDA programming model that easily match such mathematical models. CUDA expresses task and data parallelism through the threads hierarchy. As we said a kernel is a portion of code executed in parallel by different threads. Threads are organized in one-, two-, or three-dimensional *blocks*. This organization provides a natural way to work in multi-dimensional matrices. Threads within the same block can share data and synchronize themselves through intrinsic synchronization functions.
However a kernel can be executed by multiple equally-shaped thread blocks. These multiple blocks are organized in one- or two-dimensional *grids*. Since threads blocks can be executed in any order, in parallel or in series, they are required to execute independently. This independence requirement is the key to write scalable code as it allows threads blocks to be scheduled in any order across any number of of cores. While the number of threads per block is limited by physical resources (Section 2.3.5), the number of thread blocks per grid is normally related to the size of the data to be processed.
The programmer has to know a priori the number of threads he wants to dedicate to a specific kernel, and it has to declare it in terms of grid and block dimensions in way that syntactically resembles the invocation

```
kernel <<< dimGrid, dimBlock >>> (...list of parameters...)
```

CUDA provides built-in variables that help identifying a number of useful information like the threads and blocks' indexes and the blocks and grids' dimensions. In this way every single thread can compute a unique value able to distinguish it from all the other threads.
The thread hierarchy together with such a fine control over the threads allows to define different levels of parallelism. The concurrent threads of a thread block permit a fine-grained data and thread parallelism. Independent thread blocks of a grid express coarse-grained data parallelism. Independent grids express coarse-grained task parallelism. Figure 2.6 enriches the execution model of a CUDA program with the threads organization described so far.

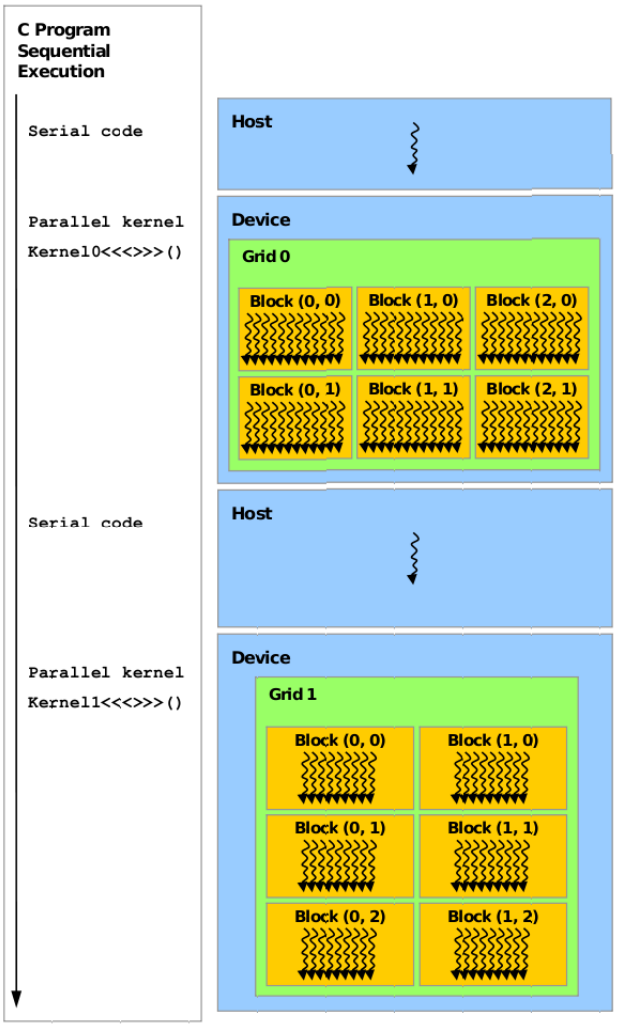

Figure 2.6: Heterogeneous programming with CUDA.(Source: *NVIDIA* [23])

2.3.4 Memory Organization

Every thread has its own local memory. All the variables within the scope of a kernel are allocated in such a memory. Aside that, threads can use also shared, global, texture and constant memory. Data allocated in shared memory is visible

to and accessible by all the threads within the same block. Such data has the same lifetime as the block itself. The global, texture, and constant memory are persistent across kernel invocations by the same application. Figure 2.7 exemplifies the concepts.

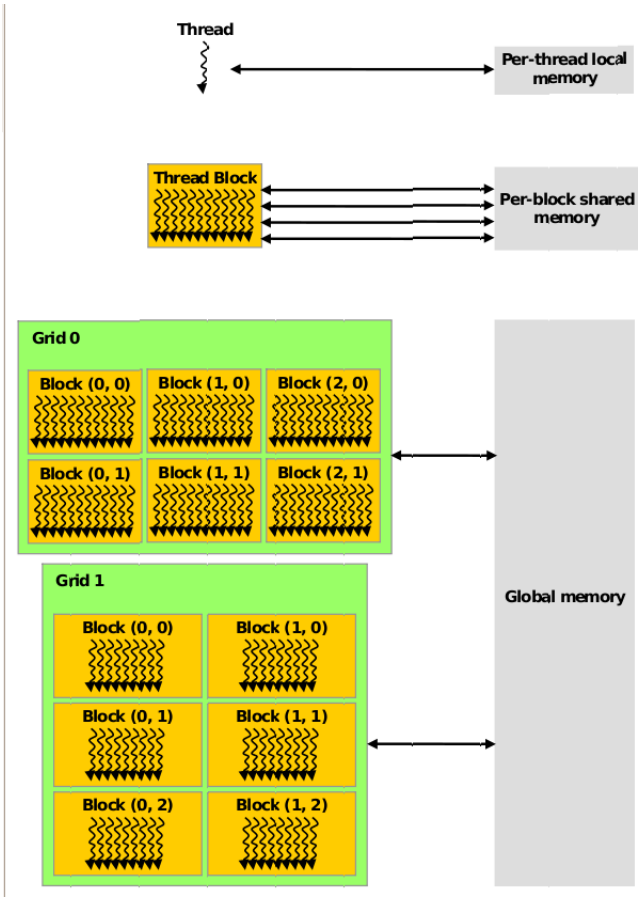

Figure 2.7: CUDA memory hierarchy.(Source: *NVIDIA* [23])

2.3.5 Mapping to the Tesla Architecture

At this point we want to describe how the main features of the CUDA programming model can map to the Tesla architecture.
When a kernel is is invoked the several thread blocks that compose the grid are enumerated and distributed to SMs with available execution capacity. Every SM can execute several thread blocks. As we said thread are grouped in warps to be executed. The way a block is split into warps is predefined and always the same: each warp contains threads of consecutive IDs with the first warp containing thread 0.
The number of blocks a SM can process at once is related to the configuration of the specific launched kernel, and in particular to how many registers per thread and shared memory per block have been declared. This because registers and shared memory are parted among the thread blocks associated to the SM. A critical point is that a kernel to be launched requires enough registers and shared memory per

SM to process at least one block, otherwise the kernel invocation will fail.
Memory concepts are easily mapped as well. CUDA local memory is mapped to registers until their are available. In case automatic variables exceed the register limit they are allocated on the device local memory space. CUDA shared, global, texture, and constant memory is naturally mapped on the homonym memory spaces on the device.

2.3.6 Compute Capability

Every device comes with a compute capability number that in a way describes the matching degree between the device and CUDA, or in other terms between the device processing model and the CUDA programming model. Simply speaking whether a feature offered by CUDA can be effectively implemented totally depends on the compute capability of the beneath device. To make an example, it is possible to define double-precision floating-point variables within a kernel, but support for those kind of variables is not provided on devices with compute capability lower than 1.3 (e.g. it would work on GPUs belonging to the GeForce 200 Series).
Compute capability numbers are defined by a major revision number and a minor revision number. The major revision number indicates the core architecture, while the minor one corresponds to incremental improvements of the core architecture with new features.

Chapter 3

Linear Programming

When we talk about *Linear Programming* (LP) we refer to mathematical models and techniques used to study and solve a specific family of problems. Such kind of problems requires to optimize a linear objective function fulfilling a specified set of constraints.
LP is object of study for an interdisciplinary branch of applied mathematics called Operational (or Operations) Research. The first models and methods have been developed during the Second World War with military aims. Some of the most famous mathematicians at that time contributed to put the foundations for a scientific approach to the problem. George Bernard Dantzig, for example, elaborated in the United States the Simplex method 3.2 in 1947, right while in Europe John Von Neumann was developing the duality theory, which we will allude to later on in the next section.
To give an idea of what a LP problem may look like consider the following imaginary scenario. The director of a famous HPC-Lab wants to build up a small but powerful cluster of GPUs. She has a precise target: performance. One day she receives an offer from one of the most important vendors of graphics processors. Two recent models in particular attracts her attention: model G, able to compute 700 GFLOPS, and model T, able to compute 900 GFLOPS. Of course, considering the target, she would like to buy several cards of model T. Unfortunately, she has to deal with some constraints coming from the departmental budget and the new environmental legislation which the lab is subject to. The former allows her to spend no more than 6000€, while the latter to not exceed the 500W limit of power requirement. Model G costs 500€ and requires 250W, while the more powerful model T costs 3400€ and consumes 200W.
The problem above exemplifies a LP problem: the lab director has to *maximize* the total performance of the cluster in terms of the number of GPUs of each model she will *decide* to buy. At the same time she has to take into account all the constraints of economical and legislative nature. The problem is presented schematically in Table 3.1.

This example is indeed quite simple with respect to other possible, more complex applications, like for instance the optimization of some design factor in civil or maritime engineering.
In this chapter we will abstract from a specific real-life context, defining a general

	GPU Model G	GPU Model T	
Performance [GFLOPS]	700	900	
Cost constraint [€]	500	3400	6000
Power constraint [W]	250	200	500

Table 3.1: Schematic summary of the Lab setting problem.

mathematical model for a LP problem in Section 3.1, and providing in Section 3.2 methods for solving it. Based on the description of those methods, in the next chapter we will select the technique that better matches with the requirements of a GPU-suitable application. Finally, in Section 3.3 we will discuss some theoretical aspects of the LP complexity.

3.1 Linear Programming Model

As we said LP has become a classical topic embracing different scientific areas. As a consequence a lot has been written about it. Bertsimas and Tsitsiklis [5] is considered a very good reference by many in the O.R. field. We give here a more formal definition of a LP problem. Unfortunately, like in many other scientific fields, part of the terminology is not universally adopted in literature. Nonetheless, we will try to be precisely coherent with the terms here adopted throughout the whole report.
A linear programming problem is constituted by the following fundamental elements:

- a linear objective function or cost function $c(\cdot) : \mathbb{R}^n \to \mathbb{R}$, i.e. it must satisfy the relations $c(\mathbf{0}) = 0$ and $c(\alpha\mathbf{x}+\beta\mathbf{y}) = \alpha c(\mathbf{x})+\beta c(\mathbf{y})$, $\forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, $\forall \alpha, \beta \in \mathbb{R}$;

- a finite set, say m, of linear constraints, where every constraint is expressed like $a(\mathbf{x}) \bowtie b$, with $a(\cdot) : \mathbb{R}^n \to \mathbb{R}$, $\mathbf{x} \in \mathbb{R}^n$, $\bowtie \in \{\leq, =, \geq\}$, and $b \in \mathbb{R}$.

The main goal for a LP problem is to optimize - i.e. either maximize or minimize - the cost function. A possible representation of a LP problem is the following:

$$
\begin{aligned}
& max\ z = c_1x_1 + c_2x_2 + \cdots + c_nx_n \\
& \\
& a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1 \\
& a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2 \\
& \vdots \\
& a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m \\
& \\
& x_1, x_2, \cdots, x_n \geq 0
\end{aligned}
\tag{3.1}
$$

The last set of inequalities in (3.1) constraints the sign of the decision variables requiring them to be positive, a very reasonable requirements if we think that they

normally represent real quantities, such as the number of GPUs to buy for the lab. A LP problem is often expressed in terms of matricial relations and written in a more compact form. Let $\mathbf{c}$ and $\mathbf{x}$ be vectors in $\mathbb{R}^n$, $\mathbf{b}$ a vector in $\mathbb{R}^m$, and $\mathbf{A}$ a matrix in $\mathbb{R}^{mXn}$, such that

$$\mathbf{c} = \begin{bmatrix} c_1 & \cdots & c_n \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

$$\mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}$$

A matrix-based form for the (3.1) can be written as

$$\begin{aligned} max\ \mathbf{cx} \\ \mathbf{Ax} \leq \mathbf{b} \\ \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{3.2}$$

Considering the dimensions used in the previous definition of a LP problem's constituting elements, we adopt the following notation in this report:

- Vectors and matrices are written in bold using respectively small and capital letters. e.g. $\mathbf{x}$ is a vector while $\mathbf{A}$ is a matrix.
- All products are written putting the factors side by side.

So, going back to the (3.2), what we did is to express in matricial form all the linear relations over the real field described in our first definition. To put it another way, we may say that we performed a number of replacements. First we replaced the function cost with the multiplication between the vector of costs $\mathbf{c}$ and the vector of decision variables $\mathbf{x}$. Then we exchange the set of inequalities with just one inequality that involves the product between the matrix $\mathbf{A}$ of the constraints coefficients and the vector of decision variables $\mathbf{x}$, and the vector $\mathbf{b}$ of the constraints known terms.
Just to put in practice, we may try to define the problem of setting up the lab, that we presented at the beginning of this chapter, using the definitions introduced so far obtaining the following formulation:

$$\begin{aligned} max\ & \begin{bmatrix} 700 & 900 \end{bmatrix} \begin{bmatrix} x_G \\ x_T \end{bmatrix} \\ & \begin{bmatrix} 500 & 3400 \\ 250 & 200 \end{bmatrix} \begin{bmatrix} x_G \\ x_T \end{bmatrix} \leq \begin{bmatrix} 6000 \\ 500 \end{bmatrix} \\ & \begin{bmatrix} x_G \\ x_T \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix} \end{aligned} \tag{3.3}$$

For the sake of preciseness, we should remark that the representation of the Lab setting problem is actually not the best representation we may give of the problem. Indeed, since we are talking about how many GPUs we should buy, a better representation would model the decision variables over the set of integer values.
This kind of problems constitute the family of Integer Linear Programming Problems, a special category treated with ad-hoc techniques that differ from the ones used to deal with usual LP problems. We will not deal with this class of problems, but we address to the texts of Wolsey [27] and Bertsimas-Weismantel [6].
Looking again at the problem though, the shown representation is to be considered a relaxation of the discrete one, that will allow us to handle it in the continuous.
In some texts, the way we expressed our problem is often called *general* or *canonical* form. This is one of possibility to express a LP problem. Another typical form is the *standard* form, which involves the same terms simply organized in a different way:

$$\begin{aligned} min\ \mathbf{cx} \\ \mathbf{Ax} = \mathbf{b} \\ \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{3.4}$$

Of course they are not the only way to see and represent a problem. In general, passing from one representation to another is possible if we take into account the following set of equivalences:

$$\begin{aligned} max\ \mathbf{cx} &\equiv -min\ -\mathbf{cx} \\ \sum_j a_{ij}x_j = b_i &\equiv \begin{cases} \sum_j a_{ij}x_j \leq b_i \\ \sum_j a_{ij}x_j \geq b_i \end{cases} \\ \sum_j a_{ij}x_j \geq b_i &\equiv \sum_j -a_{ij}x_j \leq -b_i \\ \sum_j a_{ij}x_j \geq b_i &\equiv \sum_j a_{ij}x_j - s_i = b_i, \quad s_i \geq 0 \\ \sum_j a_{ij}x_j \leq b_i &\equiv \sum_j a_{ij}x_j + s_i = b_i, \quad s_i \geq 0 \end{aligned} \tag{3.5}$$

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

$$\begin{aligned} max\ \mathbf{cx} \\ \mathbf{Ax} = \mathbf{b} \\ \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{3.6}$$

This augmentation of the canonical form will turn out useful when we will try to formulate a numerical solution to the problem, since we will be allowed to manipulate linear transformations instead of having to deal directly with a set of inequalities.

3.1.1 Geometric Interpretation

It is useful for a better understanding of LP problems as well as the techniques used to solve them, to consider what could be a geometric interpretation for them.
As we started from the main element to define formally an LP problem, we may do the same now, trying to associate to each of them the right geometric meaning. In the space of the decision variables, constraints given in form of inequalities identify *half-spaces*, whereas given as a set of equations they detect hyperplanes. The latter interpretation can be associated also to level curves of the objective function.
The intersection of all the half-spaces/hyperplanes identified by the set of constraints is called *feasible region*. The feasible region is the set of all the points of the hyperspace that fulfill the constraints. In algebraic geometry, a region defined by the intersection of half-spaces is called *polyhedron*. In the case the polyhedron is bounded the term *polytope* is also adopted. Precisely because a polyhedron is the result of the intersection of half-spaces such set is said to be *convex*. This basically tells as that if we take two points within the set, and we take the union of all the possible convex combinations of them, we will end up with the segment joining the two points [1].
The geometry of a feasible region of a LP problem is often characterized by being a convex polyhedron. A convex polytope is know as *simplex*. Examples of simplex are the point (0-simplex), the line segment (1-simplex), and the triangle (2-simplex).
Normally when dealing with few dimensions, like 2 or 3, the graphics viewpoint gives an extremely intuitive perspective that can help grasping the concepts introduced so far.
Let us always consider the same example of the Lab setting. Figure 3.1 reports graphically the LP problem in (3.3). The area colored in orange is the convex polytope obtained from the intersection of the four half-spaces generated by the constraints. The green and the red straight lines are the sets of points satisfying respectively the costs and power constraints as narrow equalities. They generates two half-spaces directed towards the center of the Cartesian plane. The other two half-spaces are generated by the sign constraints on the two decision variables. This gives us the possibility to limit our study to the first quadrant.
The orange convex polytope is then the feasible region for the problem. Two other important elements are the vertices of the polytope and the segments linking them, which are called *faces*. An important result in LP says that if a solution exists as finite for a LP problem, than at least one optimal solution coincides with one of the vertices of the feasible region. Another close proof affirms that if one of the points within a face is optimal, then all the other (infinite) points in the face are optimal too.
Looking at the representation in Figure 3.1, we can assess that what stated is confirmed by the example. In our case, the vertices set V is composed by four points, that is V = {(0;0), (0;1.7), (0.67;1.67), (2;0)}. The plot reports four level curves of the objective function together with an arrow pointing the direction of maximum growth, which is the gradient of the function. We are looking for maximizing the objective function. A way to get an optimal solution may be to push the objective function curve along its growing direction, until we reach one of the boundaries of

[1] In general a set $C \subset \mathbb{R}^k$ is convex if $\forall \mathbf{x}, \mathbf{y} \in C$ and $\forall \lambda \in [0,1]$ the convex combination $\lambda \mathbf{x} + (1-\lambda)\,\mathbf{y} \in C$.

the feasible region. The optimal solutions are the points in the intersection between the curve and the region.
In our example if we push up the objective function curve following the direction of maximum growth, we reach the situation where the curve intersects the region in the only point $P \equiv (0.67; 1.67)$ at level 1966.7. This is the optimal solution for the "relaxed" Lab setting problem, and actually we have that $P \in V$.

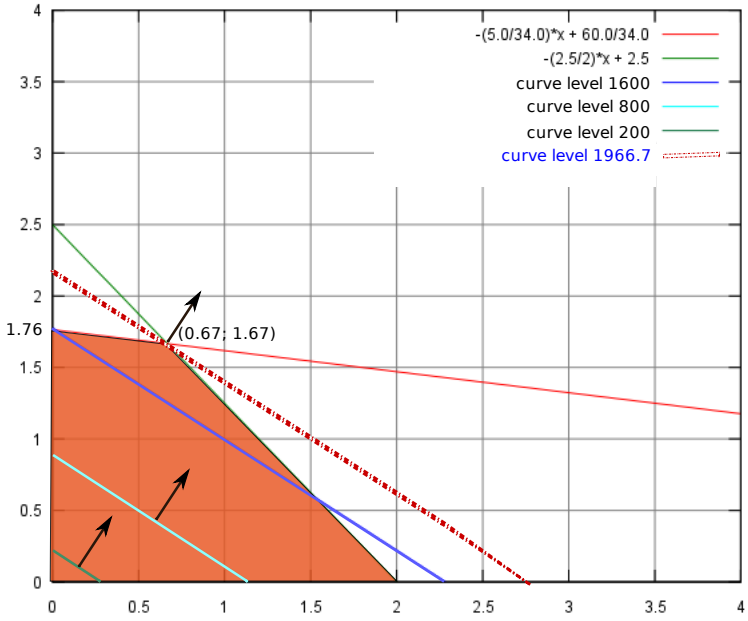

Figure 3.1: Geometric representation of the Lab setting problem.

Of course if we should consider the problem again in a real context, the solution found would be impractical, since it is not possible to buy a fraction of a GPU. For this reason we traced a curve level intersecting the point (1;1), giving as a result 1600 GFLOPS. That point is indeed the result we would found if we would study the Integer LP problem.
So far we have described what a feasible region is and we have given a simple but practical example. Of course this is not all about. In particular until now we have considered situations where the feasible region is bounded. This is not always the case. Sometimes problems in LP have constraints which generate an unbounded polyhedron as feasible region. Such problems are normally called *unbounded problems*. Note that the region is still called feasible, since the points in the region are actually feasible solutions. Nevertheless, the problem has the big weakness to not present any maximum limit for the objective function. Such problems are normally the result of an improper or incomplete analysis of the real investigated case. Figure 3.2 for instance, represents a LP problem with just two constraints producing an unbounded feasible region. Following the gradient direction, the objective level curve may grow without limit.
A third and last case to mention is the one where the feasible region is empty. It is possible to get such a region from a set of constraints totally incompatible with each other. In this case there are no solutions at all, and the problem is said to be an *unfeasible problem*.

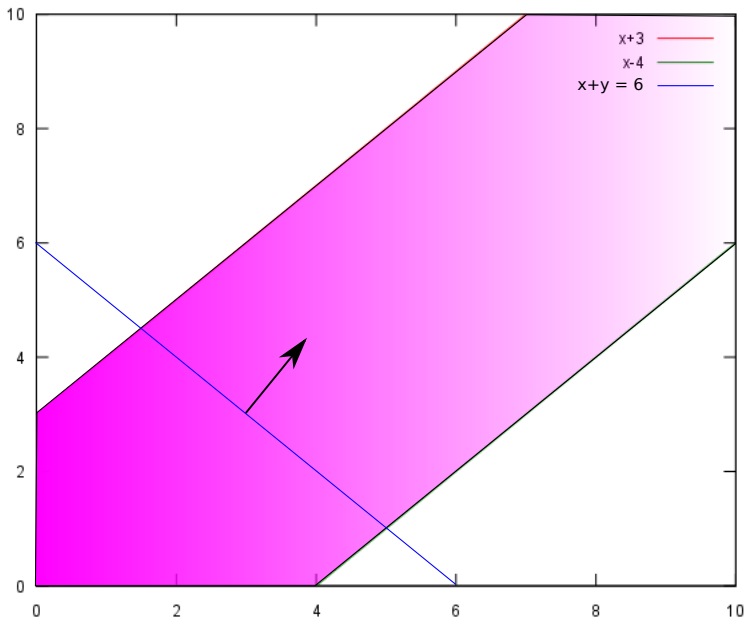

Figure 3.2: Geometric representation of an unbounded problem. The problem is the one of maximizing $z = x + y$ with the only two constraints $c1: \ x + y \leq 3$ and $c2: \ -x + y \leq 4$.

3.1.2 Duality Theory

Every LP problem expressed in the way we presented, can be associated to another formulation related to the same model the initial problem is based on. A problem described as in (3.2) is called *primal* and it can always be converted in what is called its *dual*.
Using the same mathematical entities we used for defining a primal, a dual problem can derived from the primal and written down as

$$\begin{aligned} min\, \mathbf{b^T y} \\ \mathbf{A^T y} \geq \mathbf{c^T} \\ \mathbf{y} \geq \mathbf{0} \end{aligned} \tag{3.7}$$

Moreover, it is not difficult to proof that the dual of the dual is the primal. As we started saying, duality gives a different perspective of the problem. Such a relationship between the two formulations is not just syntactical. Important theorems strictly relate them insomuch as certain methods for solving LP problems exploit both to get some results.
Practically We will not focus extensively on those relations, since we will rely on a technique that do not involve duality. However, we think it is convenient to mention at least the most important ones, since some methods described in the next section are built on top of them.
A first important theorem is the so called *weak duality theorem*. It states that given a primal and its dual, if they admit respectively feasible solutions $\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$, then $\mathbf{c}\bar{\mathbf{x}} \leq \mathbf{b^T}\bar{\mathbf{y}}$. This relation can be made stronger when the feasible solution is also the

optimal one. Indeed, the *strong duality theorem* affirms that if the primal admits optimal solution $\tilde{\mathbf{x}}$, then also the dual admits optimal solution $\tilde{\mathbf{y}}$, and $\mathbf{c}\tilde{\mathbf{x}} = \mathbf{b}^{\mathbf{T}}\tilde{\mathbf{y}}$.
Ultimately, from the previous theorems, it is possible to deduce as a corollary that the dual provides in a sense an upper bound for the primal. In fact, if the dual would contain just one solution, that value would limit the solution of the primal. Furthermore, we can also say that if the primal would present an unbounded optimality, the relative dual would turn out to be unfeasible.

3.2 Solving Linear Programming Problems

Now that we have introduced what a LP problem is, the next step is to find out how to solve it.
We introduce here two groups of methods, namely simplex-based and interior points methods. Methods within such groups are the ones commonly used to implement today's LP routines. Both the categories work with the representation provided in previous section, basically traversing the feasible region with two different approaches.

3.2.1 Simplex-Based Methods

The simplex method, is the first practically implemented method for solving LP problems. It has been developed by G. B. Dantzig, who is considered the father of LP, at the end of the 40s, and for years it has been considered "the way" to deal with optimization of linear constrained problems.
Has we previously said, if LP solutions exist they lay on vertices of the feasible region. The simplex method is an iterative method that traversing the faces of the feasible region, proceeds stepwise towards the optimal solution increasing the value of the objective function at each step.
Several methods have been then developed based on the same concepts as the simplex method. In this section we present the original simplex method and the revised matrix-based version, which is well-suited for implementations on a GPU.

The Simplex Method

The simplex method requires to organize the representation of the problem (i.e. the equations) in a way that ease the application of the method's operations at each step. Since understanding the way the basic version of the simplex method works is critical to understand how other revised versions do it, let us consider again the Lab setting example. Using slack variables we can rewrite the objective function and the set of constraint equation as a system of equalities

$$\begin{array}{rcrcrcrcrcl} z & - & 700x_g & - & 900x_t & & & & & = & 0 \\ & & 500x_g & + & 3400x_t & + & s_1 & & & = & 6000 \\ & & 250x_g & + & 200x_t & & & + & s_2 & = & 500 \end{array}$$

Now we can extract the initial *simplex tableau*

z	x_g	x_t	s_1	s_2	b
1	-700	-900	0	0	0
0	500	3400	1	0	6000
0	250	200	0	1	500

From the tableau it is possible to individuate two partitions of the set of variables. A first partition is the set of *basic variables* or simply *basis*.
A variable is basic if it has just one nonnegative value in its column. The basis provides at each step of the algorithm a feasible solution. A basic feasible solution is obtained by setting all the non-basic variables to zero. In the example, at the beginning the basis is populated by the slack variables.

$$x_g = 0 \quad x_t = 0 \quad s_1 = \frac{6000}{1} \quad s_2 = \frac{500}{1} \quad z = 6000 \cdot 0 + 500 \cdot 0 = 0$$

On the other hand, the rest of the variables with more than one nonnegative value in their columns are called *non-basic*. Note that the column subdivision in the above table is *not* intended to remark the basic/non-basic partition. It just distinguishes the original variables from the slack ones.
The simplex algorithm for a maximization LP problem is described in Algorithm 1.

Input: Matrix **A**, vectors **b** and **c**. Problem in canonical augmented form.
Output: Optimal solution or unbounded problem message

```
1  Tableau T ← CreateTableau(A, b, c);
2  Indexes basis ← Initialize(T.A);
3  while !exist(i) t.c. T.c_i > 0 do
4      /* Determine the pivot column */
5      Index p ← IndexMaximumValue(T.c);
6      Determine the pivot equation */
7      if !exist(i) t.c. T.A_ip > 0 then
8          exit("Problem unbounded");
9      Index r ← i t.c. min_i { T.b_i / T.A_ip , T.A_ip > 0 };
10     /* Elimination by row operation */
11     foreach i ∈ T.RowIndexes \ {r} do
12         EliminationByRowOperation(T, i, r);
13     /*Update the basis */
14     Index q ← GetLeavingVariableIndex(T);
15     UpdateBasis(basis, p, q);
16 exit(GetSolution(T, basis));
```

Algorithm 1: A simplex method algorithm.

Let us try to see how the algorithm would work on our tableau. During the execution of an exception-free step, we can say that the simplex algorithm focus on three main operations:

$\mathcal{O}1$ *Determine the pivot column*
The algorithm requires to select the *biggest* positive value from the vector of the objective function coefficients. In the specific case of our example, because of the way we organized the tableau, we will have to select the column with the *smallest* negative value in the first row. This because in (3.2.1) we moved all the terms from the right-side to the left-side.

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

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

Operation $\mathcal{O}1$ is motivated by the fact that we are looking for a value that can contribute in increasing the objective function.
Operation $\mathcal{O}2$ instead, selects the row that will allow us to move towards a new vertex of the feasible region with the smallest step-size. This is important insofar it minimizes the risk to step outside the region.
The last operation redraws the simplex tableau, in the light of the new pivot choice. Note that the algorithm updates the basis after $\mathcal{O}3$. This is correct. In fact, after the operation has been applied, the basis changes since the pivot column remains with just one nonnegative value. The row operation, in addition to elect a new *entering* variable, extracts a *leaving* variable from the basis increasing some of its column's values. To put it another way, the update function has to replace the leaving variable with the new entry.
Let us make it clearer by running one iteration of the algorithm. Since we have already set up the tableau and initialized the basis, we can start executing the three main operations we were talking about.

$\mathcal{O}1$ Select column 3 which has $c_T = 900$ as pivot column.

$\mathcal{O}2$ Evaluate ratios:

Row 2: $\frac{6000}{3400} \approx 1.76 \quad \longrightarrow$ Select row 2 as pivot row.

Row 3: $\frac{500}{200} = 2.5$

$\mathcal{O}3$ Apply the elimination by row operation

z	x_g	x_t	s_1	s_2	b	
1	-570	0	0.26	0	1560	$Row\ 1+ = \frac{900}{3400} Row\ 2$
0	500	3400	1	0	6000	
0	220	0	-0.06	1	140	$Row\ 3- = \frac{200}{3400} Row\ 2$

As we said after the core operations, also the basis has changed. In particular x_t entered the basis while s_1 left it. The new basic feasible solution is therefor

$$x_g = 0 \quad x_t = \frac{6000}{3400} \approx 1.76 \quad s_1 = 0 \quad s_2 = \frac{140}{1} \quad z = 1.76 \cdot 900 + 140 \cdot 0 = 1584$$

Moreover, we said that the simplex method moves along the boundaries of the feasible region. If we compare the previous result with the graphics representation in Figure 3.1, we may notice that after the first step, the algorithm moved from point (0;0) to point (0;1.76).

The Revised Simplex Method

The version of the simplex method just exposed requires specific data structures to be implemented. As we saw we need to explicitly keep track of indexes and to update the tableau at each iteration.
The revised version of the simplex algorithm we describe here, tries to formulate the same algorithm in terms of linear algebra computations. The revised method has been proposed by Dantzig et al. [8]. Basically the revised algorithm applies the same conceptual steps as the normal simplex method does, just translated into an algebraic perspective.
Similarly to Morgan [19], we present the algorithm defining a small dictionary that should help understanding the translation.
To work with the revised version, we need the problem to be expressed in terms of matrices as it is in Section 3.6. We consider the augmented canonical form, that provides us with a system of linear equations.
Furthermore, we need to define mathematical structures able to represent some fundamental concepts. We need a basis matrix, $\mathbf{B}$, consisting of the columns of $\mathbf{A}$ corresponding to the coefficients of the basic variables. Note that $\mathbf{B}$ is a m x m squared matrix, since we introduce a slack variable for each constraint. The m nonzero variables in a basic solution, can be represented as a vector $\mathbf{x_B}$. Similarly, we denote the coefficients of the objective function corresponding to the basic variables with the vector $\mathbf{c_B}$.
Now, we saw that the simplex algorithm chooses the pivoting or entering variable picking up the one that causes the greatest increment in the objective function. This is done by selecting the most negative entry in the objective row of the tableau. Even if the tableau, and in particular the objective row, is not explicitly represented, we can determine the entering variable based on the contribution of the non-basic variables. Such a contribution is estimated corresponding to each variable as $z_j - c_j = \mathbf{c_B B^{-1} A_j} - \mathbf{c_j}$. The variable corresponding to the smallest negative difference is the entering variable, say x_p. So, writing $\tilde{c} = \mathbf{c_B B^{-1} A_j} - \mathbf{c_j}$, we can say that

$$p = j \mid \tilde{c}_j = min_t \left\{\tilde{c}_t\right\}, \ \tilde{c}_j < 0$$

If we cannot find any negative $\tilde{c}$, means that we have reached the optimum (no contribution can cause improvement).

After having selected the entering variable the algorithm requires to find out the leaving variable. To do this we need to compute every time the basic solution $\mathbf{x_B}$. Since all the variables outside the basis are set to zero, at each iteration, the system

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

Table 3.2: Normal and revised versions of the main simplex method's steps.

of equation can be written as $\mathbf{Bx_B} = \mathbf{b}$. From the latter we can easily compute $\mathbf{x_B} = \mathbf{B^{-1}b}$. Defining $\alpha = \mathbf{B^{-1}A_p}$, the leaving variable, say x_q, is the one with minimum θ-ratio, where $\theta_j = \mathbf{x_B}_j/\alpha_j$. Precisely

$$q = j \mid \theta_j = min_t \{\theta_t\}\,,\ \alpha_j > 0$$

If $\alpha \leq \mathbf{0}$, the solution is unbounded.

Finally we have to update the basis, that in the revised context means we have to update the basis matrix $\mathbf{B}$. Table 3.2 summarizes how the main steps of the simplex method can be revised from an algebraic viewpoint, while Algorithm 2 presents the revised simplex method.

Several revised versions of the simplex method have been developed. They mainly differ from each other in their implementation, since the apply exactly the same concepts introduced so far. Morgan [19] presents and compare the most important ones, like the Bartel-Golub's method, the Forrest-Tomlin's method, and the Reid's method.

3.2.2 Interior Point Methods

An alternative approach to simplex-based methods is called *interior point method*. In general with this name we can point to a family of methods inspired by Narendra Karmarkar, that in 1984 developed a new polynomial time algorithm, then Karmarkar's algorithm, for solving linear programming problems [13].
The basic idea behind the this alternative optimization process can be given describing geometrically the difference between the simplex and the interior point methods. A typical interior point path goes through the feasible region traversing its interior as shown in Figure 3.3. The green points are the ones obtained by the interior point method, while the red ones by the simplex method.
The figure reports some relevant features important to recall that we can explain as follows:

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

```
/* Initialize data (Range assignements with Matlab-like notation). */
B[m][m] ← Initialize(A);
c_B[m] ← c[n − m : n − 1];
x_B[m] ← 0;
Optimum ← ⊥;
while !Optimum do
    /* Determine the entering variable */
    Index p ← {j|c̃_j == min_t (c_B B^{-1} A_t − c_t)};
    x_B ← B^{-1} b;
    if c̃_p ≥ 0 then
        Optimum ← ⊤;
        break;
    /* Determine the leaving variable */
    α ← B^{-1} A_p;
    Index q ← { j|θ_j == min_t ( x_{B_t} / α_t , α_t > 0 ) };
    if α ≤ 0 then
        exit("Problem unbounded");
        break;
    /*Update the basis */
    B ← UpdateBasis(A, p, q);
    /* Update basis cost */
    c_{B_q} ← c_p;
    /* Update basis solution */
    x_B ← B^{-1} b;
if Optimum then
    exit(x_B, z ← x_B c_B);
```

Algorithm 2: A revised simplex method algorithm.

- starting from an interior point, the method constructs a path that reaches the optimal solution after a few iterations.
- The method leads to a good estimate of the optimal solution after a few iterations.

While theoretically this class of methods have been considered from the beginning quite competitive with the classical simplex, practically there is no clear understanding of which algorithmic class gives the best performance.
Among the different algorithms, the class of primal-dual path-following interior point methods are widely considered the most efficient [26]. In the last part of this section we focus on Mehrotra's algorithm [18, 26] that can be considered the basis for a class of algorithms called *predictor-corrector* algorithms.

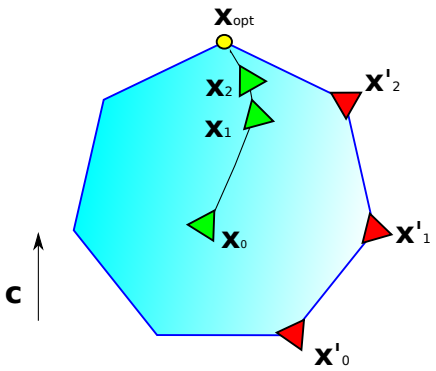

Figure 3.3: Polytope of a two-dimensional feasible region. The $\mathbf{x}_i$ points are determined by an interior point method, while the $\mathbf{x'}_i$ ones by a simplex method.

Mehrotra's Primal-Dual Predictor-Corrector Method

We consider a LP problem as given in (3.4) and its dual (3.7) in an augmented form. For the sake of readability we present again the two formulations:

$$\begin{aligned} (P) \quad & min\left\{\mathbf{cx} : \mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\right\}, \\ (D) \quad & max\left\{\mathbf{b^T y} : \mathbf{A^T y} + \mathbf{s} = \mathbf{c^T}, \mathbf{s} \geq \mathbf{0}\right\}. \end{aligned}$$

We assume, without loss of generality, that there exists a 3-tuple $(\mathbf{x}_0, \mathbf{y}_0, \mathbf{s}_0)$ such that

$$\mathbf{Ax}_0 = \mathbf{b}, \mathbf{x}_0 \geq \mathbf{0}, \quad \mathbf{A^T y}_0 + \mathbf{s}_0 = \mathbf{c^T}, \mathbf{s}_0 \geq \mathbf{0}.$$

This condition is called the *interior point condition*. The basic idea of primal-dual interior point methods is to find an optimal solution to (P) and (D) solving the following system:

$$\begin{aligned} \mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}, \\ \mathbf{A^T y} + \mathbf{s} = \mathbf{c^T}, \mathbf{s} \geq \mathbf{0}, \\ \mathbf{xs} = \mu\mathbf{e}. \end{aligned} \tag{3.8}$$

If the interior point condition holds, then for all $\mu > 0$ the (3.8) has a unique solution called μ-center of the primal-dual pair (P) and (D), denoted by $(\mathbf{x}(\mu), \mathbf{y}(\mu), \mathbf{s}(\mu))$. The set of μ-centers with $\mu > 0$ gives the central path of (P) and (D). Since it is demonstrated that the limit of the central path exists, and because the limit point satisfies the complementary condition, it yields optimal solution for the problem. Applying Newton's method to (3.8), we obtain the following linear system of equations

$$\begin{aligned} \mathbf{A}\Delta\mathbf{x} = \mathbf{0}, \\ \mathbf{A^T}\Delta\mathbf{y} + \Delta\mathbf{s} = \mathbf{0}, \\ \mathbf{x}\Delta\mathbf{s} + \mathbf{s}\Delta\mathbf{x} = \mu\mathbf{e} - \mathbf{xs}, \end{aligned} \tag{3.9}$$

where $\Delta\mathbf{x}, \Delta\mathbf{y}, \Delta\mathbf{s}$ give the Newton step.

Predictor-corrector algorithms deal with (3.9) using different values of μ in the predictor and corrector phases. In the predictor step, using $\mu = 0$, Mehrotra's algorithm operates computing the affine scaling search direction,

$$\begin{aligned} \mathbf{A}\Delta\mathbf{x}^{aff} &= \mathbf{0}, \\ \mathbf{A}^{\mathbf{T}}\Delta\mathbf{y}^{aff} + \Delta\mathbf{s}^{aff} &= \mathbf{0}, \\ \mathbf{x}\Delta\mathbf{s}^{aff} + \mathbf{s}\Delta\mathbf{x}^{aff} &= -\mathbf{x}\mathbf{s}, \end{aligned} \tag{3.10}$$

then it computes the maximum feasible step size α_{aff} that ensures the positivity constraint,

$$\alpha_{aff} = max\left\{\alpha \mid (\mathbf{x}(\alpha), \mathbf{y}(\alpha), \mathbf{s}(\alpha)) \in \mathcal{F}\right\}, \tag{3.11}$$

where $\mathcal{F} = \left\{(\mathbf{x}, \mathbf{y}, \mathbf{s}) \mid (\mathbf{x}, \mathbf{s}) > \mathbf{0},\ \mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{A}^{\mathbf{T}}\mathbf{y} + \mathbf{s} = \mathbf{c}^{\mathbf{T}}\right\}$. Afterwards, it uses the results coming from the previous step to get the centering direction from

$$\begin{aligned} \mathbf{A}\Delta\mathbf{x} &= \mathbf{0}, \\ \mathbf{A}^{\mathbf{T}}\Delta\mathbf{y} + \Delta\mathbf{s} &= \mathbf{0}, \\ \mathbf{x}\Delta\mathbf{s} + \mathbf{s}\Delta\mathbf{x} &= \mu_c\mathbf{e} - \mathbf{x}\mathbf{s} - \Delta\mathbf{x}^{aff}\Delta\mathbf{s}^{aff}. \end{aligned} \tag{3.12}$$

At this stage, μ is set to a specific value μ_c, which depends on the specific kind of algorithm in use. Finally the algorithm makes a step in the new direction just computed.
Algorithm 3 sums up the whole optimization strategy. The term $\mathcal{N}$, which indicates the specific neighborhood where the algorithm works in, depends also on the specific version of the Mehrotra's algorithm.

Input: Problem in the (P) and (D) form. $(\mathbf{x}_0, \mathbf{y}_0, \mathbf{s}_0) \in \mathcal{N}$. Accuracy parameter ϵ.
Output: Optimal solution.

```
1 while evaluateSolution(x,s,ε) do
2     /* Predictor step */
3     Solve (3.10);
4     α_aff = max{α | (x(α), y(α), s(α)) ∈ F};
5     /* Corrector step */
6     μ ← μ_c;
7     Solve (3.12);
8     α_c = max{α | (x(α), y(α), s(α)) ∈ N};
9     step(x + α_c Δx, y + α_c Δy, s + α_c Δs);
```

Algorithm 3: A revised simplex method algorithm.

3.3 Complexity Aspects

Theoretical computer science is an important field that deals with abstract aspects of computational models. When studying and designing algorithms it is always better to ground decisions on solid theoretical basis. This is an important principle not always taken into account when developing software. It may be extremely helpful to prevent energy profusion on problems that have already been proof even as unsolvable by a computer.
Before describing computational complexity issues that concern linear programming we want to recall the main concepts that will be involved in the discussion.
First the concept of decidability. When we say that a problem is decidable this means that is possible to write a program able to solve it in a finite time. Decidable problems can be classified depending on their complexity. The concept of complexity in itself is nonetheless quite arbitrary, so when considering algorithms the most used acceptations of complexity are those of temporal and spacial complexity. This means that an algorithm is classified based on the time or the space it requires to be computed. Moreover, when the analysis is done on parallel algorithms the processor complexity is also an important term.

For the sake of our discussion two complexity classes are noteworthy: $\mathcal{P}$ and $\mathcal{NC}$. $\mathcal{P}$ is the class of problems decidable in sequential time $n^{O(1)}$. Nick's Class, or shortly $\mathcal{NC}$, is the class of problems decidable in parallel time $(\log n)^{O(1)}$ and processors $n^{O(1)}$. Parallel computational analysis often refers to $\mathcal{P}$ as the class of feasible parallel problems, and to $\mathcal{NC}$ as the class of feasible highly parallel problems.
There is a theoretical results that states that a problem is decidable in sequential time $n^{O(1)}$ if and only if it is decidable in parallel time $n^{O(1)}$ with processors $n^{O(1)}$ (For more details consult Greenlaw et al. [11]). The lemma actually tells us that $\mathcal{NC} \subseteq \mathcal{P}$. The latter relation open to a problem that in its formulation resembles the much more famous $\mathcal{P} \subseteq \mathcal{NP}$, which is considered one of the six unsolved millennium problems[2]. Indeed, the important question for the parallel community is whether the inclusion is proper or not.
The problem is still open, but some conjectures are made by theoreticians based on experiences with two other important concepts, namely the one of reducibility and of $\mathcal{P}$-completeness. Intuitively, a problem P_1 is reducible to P_2 if there exists an algorithm able to compute the solution of every instance of P_1 in terms of the solution of an opportune instance of P_2. A problem is said to be $\mathcal{P}$-complete when it is a P-hard problem itself in P. A problem is P-hard if and only if every problem $P' \in \mathcal{P}$ is reducible in polynomial time to it.
From empirical experiences, it is common evidence that the two classes do not coincide. the reason is that it appears that P-complete problems are inherently sequential, meaning that they are feasible problems without any highly parallel algorithm for its solution.

In 1979 Khachian [14] proofed that linear programming is in $\mathcal{P}$, finding a polynomial time algorithm for it. A similar but improved result arrived a few years later due to the work of Karmarkar [13]. As Greenlaw et al. [11] shows, we can push the

[2] http://www.claymath.org/millennium/

complexity analysis even further, placing LP in the class of P-complete problems. In general such a conclusion should discourage from looking for a parallel solution. And this is probably the reason why little effort have been put in searching for a LP parallel definition. Nevertheless in the next chapter we will argue why we decided to continue with our study.

Chapter 4

Linear Programming in CUDA

Once introduced GPU technologies, the GPGPU approach, and the mathematical background behind linear programming, we start in this chapter concentrating on the CUDA implementation of a LP solver.
A first question we should address and answer is unquestionably the following: why to climb a mirror? We mentioned that LP is P-complete and so difficultly in $\mathcal{NC}$. Why then to try to find a parallel solution for it? Mainly for two reasons.
A first reason has "explorative" nature. Even though realistically we know that the problem is hardly parallelizable, we may always draw some conclusion about its quasi-parallelizability. In other words, we cannot exclude that with this study we will be able to find at least some parts of the algorithm that are suitable for parallelism. Previous experiences to support LP with heterogeneous resources appeared to us motivating enough to explore how the problem can be mapped to the new GPGPU paradigm presented by CUDA.
A second reason instead is more practical. We saw that GPUs are very powerful, affordable coprocessors. Whatever improvement the GPU would be able to produce, we think that it could be a step forward to quickly solve LP problems as well as many other problems reducible to them.
Aside this primary goal, there are some other issues we want to evaluate. First we want the gap between the algorithm and the implementation as narrow as possible, according to the new GPGPU approach. The code should easily give the idea of which algorithm has been implemented. We want also to check if the process of writing such an easy-to-read code can be modelled in a straightforward fashion. To put it another way, we want to see how much performance it is possible to gain applying a simple sequential-parallel programming concepts mapping.
We will therefor operate as follows. In Section 4.1 we will select the LP solving method to be implemented, defining the algorithm that we will use for both the sequential and the CUDA solution. Finally, in Section 4.2, we will explain the implementation strategy.

4.1 Method Selection

Designing LP solver based on GPUs with a better performance than existing, sequential ones is possible. Jung [12] and Greeff [10] report about good results obtained implementing respectively a revised version of the simplex method, and a primal-

dual interior point method.
As we already mentioned, even though interior point methods are theoretically faster, there are no practical outcomes that can state which method is the best one. Commercial software can be found today based on both the techniques.
What we want is an algorithm within the class of those suitable for GPUs. As a metric for the decision we base our choice on the characteristics elicited by Owens at al. [24] already mentioned in the introduction to Chapter 2.
Looking at the description we made in the previous chapter, we can observe that effectively both the revised simplex method and the Mehrotra's primal-dual predictor-corrector method are good candidates:

Computational requirements are large. We want to focus especially on huge-sized problems, with hundreds of variables and constraints.

Parallelism is substantial. The problem solving strategy involves mainly algebraic and reduction operations for which parallel solutions are well known.

Throughput is more important than latency. There are no specific requirements on the latency of single operations.

On the other side, the original simplex technique would not perform well, since it would require the management of ad-hoc data structures together with a number of pointers.
By the way, to use matrices instead of specific data structures is not totally free from problems. In particular we refer to physical factors that can become critically limiting. For instance round-off errors, significant digit loss, and widely differing orders of magnitude are quite common issues when manipulating numbers on a computer. The process of developing a good LP solver with matrix-based methods is actually a task in numerical stability apart from the algorithm implementation.
Discarded the basic simplex method, it remains to select a method within the matrix-based ones. Note that both the simplex-based and the interior point methods presented in Section 3.2 requires to deal with matrix inversion at a glance. Indeed the revised simplex method needs to work with the inverse matrix $\mathbf{B^{-1}}$, while the Mehrotra's algorithm has to solve linear systems of equations.
Such an expensive operation can be properly overcome with alternative operations depending on the specific applicative case. Let us consider the case studies previously mentioned. Jung [12] uses the Cholesky decomposition to efficiently solve linear systems of equation, while Greeff [10] cleverly notices that the algorithm only works with $\mathbf{B^{-1}}$. For this reason, it argues that is not required to pass every time from $\mathbf{B}$, suggesting to compute its inversion at each iteration from the old $\mathbf{B^{-1}}$ and values depending on entering and leaving variables. The precise technique is described later on in Section 4.2.
Taking into account what said so far, we conclude selecting the revised simplex method. It presents an algorithm that can be implemented using matrix operations without any additional non-matrix structures. Compared to the interior point method it requires less expensive operations to be performed.
In Section 3.2 Algorithm 2 describes a general revised algorithm of the simplex method. In this section we will try to expand the pseudo-code until a level where

all the important operations are listed.
Although the description will be done at an abstract level, it will emphasize the definition of the most important data elements and operations on them. This stage will support the development of both the sequential and the CUDA implementation. The new expanded version of the revised simplex method is reported in Algorithm 4.

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

1 /* *Initialize data (Range assignements with Matlab-like notation).* */
2 $\mathbf{B}[m][m] \leftarrow \mathbf{I}_m$;
3 $\mathbf{c_B}[m] \leftarrow \mathbf{c}[n-m : n-1]$;
4 $\mathbf{x_B}[m] \leftarrow \mathbf{0}$;
5 *Optimum* $\leftarrow \perp$;
6 **while** *!Optimum* **do**
7 /* *Determine the entering variable* */
8 $\mathbf{y}[m] \leftarrow \mathbf{c_B B^{-1}}$;
9 $\mathbf{e}[n] \leftarrow [1\ \mathbf{y}] \cdot [-\mathbf{c}\ ;\ \mathbf{A}]$;
10 Index $p \leftarrow \{j | \mathbf{e}_j == min_t(\mathbf{e}_t)\}$;
11 $\mathbf{x_B} \leftarrow \mathbf{B^{-1}b}$;
12 **if** $\mathbf{e_p} \geq 0$ **then**
13 *Optimum* $\leftarrow \top$;
14 `break`;
15 /* *Determine the leaving variable* */
16 $\alpha[m] \leftarrow \mathbf{B^{-1}A_p}$;
17 **for** $t \leftarrow 0$ **to** $m-1$ **do**
18 $\theta_t \leftarrow \alpha_t > 0\ ?\ \frac{\mathbf{x_{B}}_t}{\alpha_t}\ :\ \infty$;
19 Index $q \leftarrow \{j | \theta_j == min_t(\theta_t)\}$;
20 **if** $\alpha \leq \mathbf{0}$ **then**
21 `exit("Problem unbounded")`;
22 `break`;
23 /* *Update the basis* */
24 $\mathbf{E}[m][m] \leftarrow$ `computeE`(α, q);
25 $\mathbf{B^{-1}} \leftarrow \mathbf{EB^{-1}}$;
26 /* *Update basis cost* */
27 $\mathbf{c_{B}}_q \leftarrow \mathbf{c}_p$;
28 /* *Update basis solution* */
29 $\mathbf{x_B} \leftarrow \mathbf{B^{-1}b}$;
30 **if** *Optimum* **then**
31 `exit(`$\mathbf{x_B}$, $z \leftarrow \mathbf{x_B c_B}$`)`;

Algorithm 4: A revised simplex method algorithm.

4.2 Implementation Strategy

We already mentioned that aside the final goal of writing a LP solver, we want to assess the methodology of passing from a sequential version of the code to a parallel one. We want to assess how much performance we can gain through a simple sequential-parallel programming concepts mapping.
Before starting describing the mapping let us analyze how Algorithm 4 is implemented in practice.
The sequential version of the LP solver is implemented in C and together with the CUDA parallel one is reported in Appendix A.
The main data structures are matrices and arrays. They are defined as 0-index, row-major arrays in central memory.
Routines can be categorized in two sub-classes: algebraic and not-algebraic routines. The first group of routines is implemented using BLAS, while the second one defining an ad-hoc set of functions.
For passing to the parallel version we adopt the mapping summarized in Table 4.1. Later on in the section we summarize notes and comments related to the main implementation features.

Sequential Version	**Parallel Version**
Array in central memory.	Array in device memory.
BLAS algebraic routine.	CUBLAS algebraic routine.
Non-algebraic routine.	Non-algebraic routine supported by or totally implemented as CUDA kernels.

Table 4.1: Mapping between sequential and parallel implementation concepts.

4.2.1 Data Structures

We gives a specular vision of how data is managed respectively in the sequential and parallel versions of the LP solver.
The problem is represented in the same way for both the versions. It consists of three main data structures: the constraints matrix, the costs array, and the known terms array.

Sequential Version

Matrices are managed as arrays. Arrays are accessed using the typical C mode, i.e. 0-index, row-major.
Where possible, loops have been replaced by algebraic operations, thereby introducing matrices. The most evident example of this is the computation of the contributions, which in Algorithm 4 is substituted with the multiplication $\mathbf{e}[n] \leftarrow [1\mathbf{y}] \cdot [-\mathbf{c}; \mathbf{A}]$.

Parallel Version

CUBLAS library manages matrices in Fortran style, making column-major accesses. In order to be CUBLAS-compliant and, at the same time, to maintain the C con-

vention, accesses to matrix's elements is done through the macro

```
#define R2C(i,j,s) (((j)*(s))+(i)),
```

where the stripe s has to be set to the number of rows of the matrix in use.

4.2.2 Kernels Configuration

Kernels work mainly on matrices, and each thread works with single values. Matrices are associated to a bidimensional grid and split in regular sub-matrices, associating each sub-matrix to a bidimensional block.
Blocks' dimensions are multiple of the warp size.

4.2.3 Non-Algebraic Routines: Computing Entering Variable

Sequential Version

Looking for the entering variable is implemented with two matrix-matrix multiplications, an array movement, and a minimum search. The portion of code in question is shown in Figure 4.1). There the minimum search function is wrapped in *entering_index()*.

```
...
// y = cb*Binv
cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
    1, m, m, 1, cb, m, Binv, m, 0, y, m);
memcpy(&yb[1], y, m*sizeof(float));

// e = [1 y]*[-c ; A]
cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
    1, n, m+1, 1, yb, n, D, n, 0, e, n);
ev = entering_index(e, n);
...
```

Figure 4.1: Portion of code for computing the entering variable.

Parallel Version

The parallel version is obtained applying the transformation previously described. Therefor the piece of code for computing the entering variable is composed by two CUBLAS calls, a data movement in global memory, and a minimum search.
The minimum search is implemented as a reduction kernel (Figure 4.2). The kernel is called over and over until a single value is reached. Figure 4.3 shows its logic with a small example.
Since the search space is passed through more than once, blocks load their portion of data in shared memory. Shared memory accesses are made with an odd stripe so to avoid bank conflicts.

```
...
//Each block loads its elements into shared mem,
//padding if not multiple of BS
__shared__ float sf[BS];
sf[tid] = (j<n) ? f[j] : FLT_MAX;
__syncthreads();
...
for(int s=blockDim.x/2; s>0; s>>=1)
{
  if(tid < s) sf[tid] = sf[tid] > sf[tid+s]
    ? sf[tid+s] : sf[tid];
  __syncthreads();
}

if(tid == 0) min[blockIdx.x] = sf[0];
...
```

Figure 4.2: Portion of array loaded in shared memory and reduction process for the search of the minimum value.

4.2.4 Non-Algebraic Routines: Computing Leaving Variable

Looking for the leaving variable requires a set of instructions in a way similar to the one required by the entering variable search task. It requires to extract a column from a matrix (data movement), multiply the resulting array by a different matrix, compute a element-wise division of two arrays, and operate another minimum search.

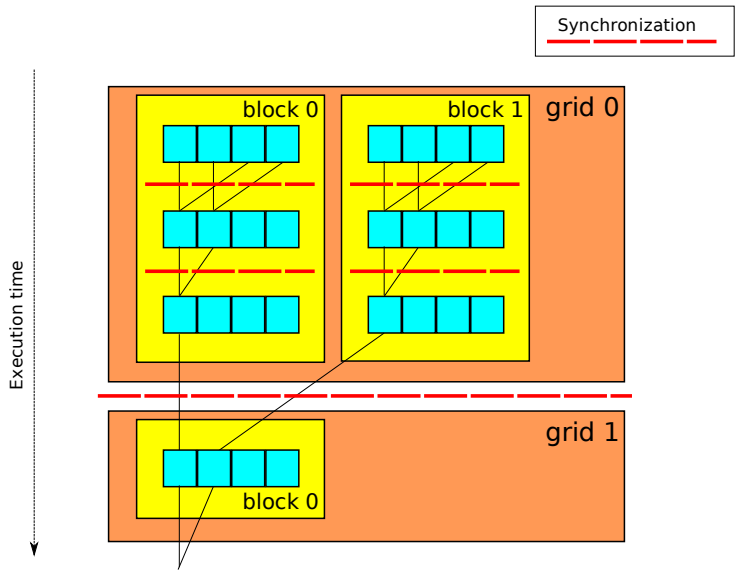

Figure 4.3: Reduction routine.

4.2.5 Non-Algebraic Routines: Computing $\mathbf{B^{-1}}$

We have already mentioned that the matrix inversion operation is avoidable. In fact the normal matrix **B** is not used at all during the computation. A method to compute its inverse matrix without dealing with inversion itself would be quite welcome.
Greeff [10] references a technique where $\mathbf{B^{-1}}$ is computed multiplying a matrix **E** by the actual value of $\mathbf{B^{-1}}$. This approach is the one effectively used in our implementation and it has already been taken into account when we formulated Algorithm 4.
E is a m x m matrix depending on entering and leaving variables. It is obtained starting from a m x m identity matrix. Let p and q be the entering and leaving index respectively. We apply the following substitution to the q^{th} column of $\mathbf{I}_m$:

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

Chapter 5

Experimental Results

With this chapter we have reached the final stage of the project development. Now we want to describe some results obtained running the LP solvers. We will describe the hardware and software environment where the experiment has been carried out. We will focus our attention on the methodology, defining which elements should be object of our analysis, and describing the tools involved in the test. Finally, observing the results, we will draw out some reflections.

5.1 The Experimental Environment

The experiment has been run on a CPU/GPU heterogeneous system. The CPU is a 64-bit Intel Core2 Quad (Q9550) 2.83 GHz, with 12 MB cache size, and bus working at 1333 MHz. The GPU is an NVIDIA GeForce GTX 280. The model has been described in Section 2.1.
The central system and the GPU communicate through PCI-Express 2.0. The GPU is connected to a 16-lanes slot (x16). Since each lane has a bandwidth of 0.5GB/s, the CPU-GPU system can transfer ub to 8GB/s.
The operating system is Ubuntu 8.04 with Linux kernel 2.6.24-22. The CPU application has been compiled using gcc version 4.2.4. The system has a BLAS library optimized by ATLAS version 3.6.0. The GPU software is developed with CUDA version 2.0.

5.2 Methodology

Both the serial version and the host code have been compiled with *-O3* optimization flag. The experiment has been run with 1000 different LP problems. Such problems are built upon random values. Problems size grows progressively. The biggest problem present a 2000 x 4000 constraints matrix. Problems are built with the same number of constraints and variables. Additional slack variables are added to generate a problem in augmented canonical form. For the biggest generated problem for instance, it means that it involves itself 2000 variables subjected to 2000 constraints, and other 2000 slack variables are added to generate the problem formulation required to compute the solution.
From now on, we will refer to the number of constraints to express the dimension

of a problem.
For each execution the main temporal parameters have been captured producing two different outputs.
A first output is a list of comma separated values (*csv* format) which summarizes the executions. Practically the file contains for each execution the number of constraints, the number of variables[1], the dimension of the constraints matrix, the elapsed time, and the optimum value found (if any).
The second output file, contains relevant times and speedups in a tabular format useful to support graphical plotting of the results.

5.2.1 Analysis Objectives

A very important dimension in performance analysis is certainly the time. Using milliseconds may not be sufficient to produce good estimates. We decided to enlarge the precision.
The OS provides developers with a library call able to retrieve the time of a system-wide realtime clock. The time is represented with seconds and nanoseconds since the Epoch[2].
Such a routine, namely *clock_gettime()*, returns the instant at the time of the call on the specified clock in the *timespec* format:

```
struct timespec {
   time_t tv_sec;        /* seconds */
   long   tv_nsec;       /* nanoseconds */
};
```

A brief comment on the structure and how to manage it is worth doing. The field *tv_nsec* are the nanoseconds from the last elapsed second. It is important to keep this in mind when trying to compute the difference between to points in time where the most recent instant counts less nanoseconds than the least recent one.
We managed to do it in the way reported in Figure 5.1.

Main target of our analysis are elapsed time and speedup of the two versions of the solver and some of their parts. Elapsed times are specially important because it is the time effectively perceived by a potential user of the solver.
The speedup will be computed for each timed part as t_s/t_p, where t_s is the sequential execution time and t_p the parallel one.

5.2.2 Tools

Aside the important tools that compose the software environment that we mentioned in the previous section, we developed a couple of applications to support the practical execution of the experiment.
A first application, called *popmat*, given the file name and the number of constraints

[1]Since the problem is given in input as an augmented canonical form, the number of variables is comprised of slack variables.

[2]Unix and POSIX systems count time since 00:00:00 UTC on January 1, 1970. Such a starting point is often called the Epoch.

```
...
clock_gettime(CLOCK_REALTIME, &start);
...
clock_gettime(CLOCK_REALTIME, &end);

nsec = end.tv_nsec-start.tv_nsec;
if(nsec < 0)
{
   nsec = 1E9+nsec;
   end.tv_sec-=1;
}

printf("Elapsed time: %.9f\n",
   (double)(end.tv_sec-start.tv_sec)+(double)(nsec*1E-9));
...
```

Figure 5.1: Timing technique.

and variables in canonical form, produces a file that contains the problem in augmented standard form.
There are also *matgen* and *clock*, two python scripts responsible to instantiate the problems, launch the tasks and capture the output, creating the files described at the beginning of this section.
Python and C code for the applications can be found in Appendix B. To plot the different diagrams we used Gnuplot[3].

5.3 Results Elicitation and Analysis

A first performance result is shown in Figure 5.2. It represents the elapsed time for the execution of both the versions of the LP solver. It presents a clear trend during the execution of matrices with less then 900 constraints. After that point it starts alternating good and very bad performance (even around 7s right after a few executions). Looking closer at the output files it is possible to see that it is an effect due to a lack of preciseness.
Significant deviation from expected results has been observed for values close to zero. At two points in Algorithm 4 it is necessary to compare variables to zero. Both the CPU and the CUDA implementations were too numerically inaccurate to allow a direct comparison with zero. For this purpose variables are compared with approximated values in the neighborhood of zero. Experimentally we found two different zero tolerances for the GPU and the CPU. With the CPU we used a tolerance $\epsilon = 10^{-4}$, while we used $\epsilon = 7\text{x}10^{-5}$ with the GPU.
So the CPU version turned out to be in some cases more precise than the GPU one. Optimal values retrieved by the GPU appeared to be identical to those found by the CPU until the fifth or the sixth decimal digit.
For problems where the optimal value is close to zero often the GPU cannot even

[3] http://gnuplot.info/

conclude the computation, protracting it (and creating huge delays) until a point where the result is *NaN*.

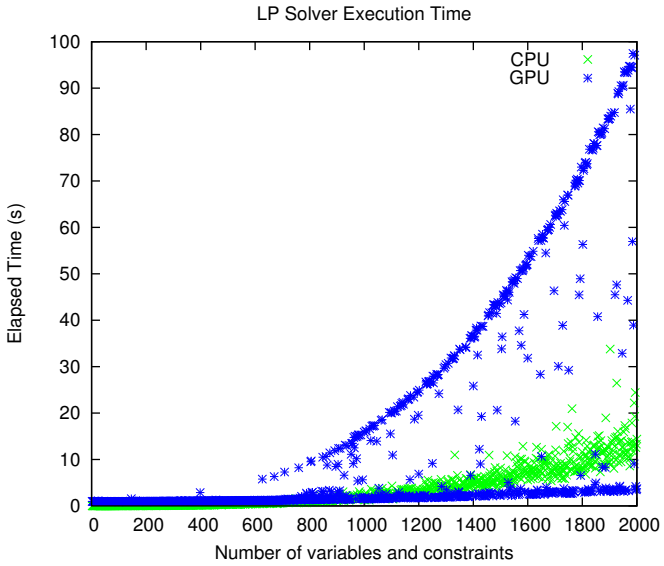

Figure 5.2: Elapsed time for the two versions of the LP solver.

To better investigate the reason why the serial solver performs better for problems with less than 900 constraints, we may look at Figure 5.3. It shows a gap of approximately 0.9s between the serial and the CUDA execution time. At this point could be relevant to verify the time required by smaller parts of the two programs. We acquired times for the three main task required by the revised algorithm, i.e. entering variable search (Figure 5.4), leaving variable search (Figure 5.5), and inverse basis updating (Figure 5.6).
Analysing the data we can note that the entering value task performs practically always worse on the GPU within a range of 0.1s.

The task is in turn composed by an algebraic (BLAS-implemented) and a not-algebraic subsets of instructions. Since the CUBLAS routines contribute just for the 0.004% over the total amount of time in the CUDA version, we can conclude that the biggest delay comes from the entering variable retrieval. The other two tasks give better results, in particular the basis updating one. It always requires less than 10^{-4}s.

Nevertheless, the entering variable retrieval task cannot be responsible for the 1s gap between serial and parallel version for small to medium-sized problems. Indeed, its delay contribution is still too small (0.5%). Actually, even summing up all the retrieved times for all the three program sub-parts we still have a huge gap ($\approx$ 0.9s). The only remaining part not yet assessed, is the one related to allocation, deallocation, and movement of data between central memory and device memory. Timing those parts we can realize that they exactly represent the reason of the 0.9s remaining gap.

Data is allocated in main memory using the *cudaMallocHost()* function in order to use pinned memory and set everything up for DMA transfers. With this expedient

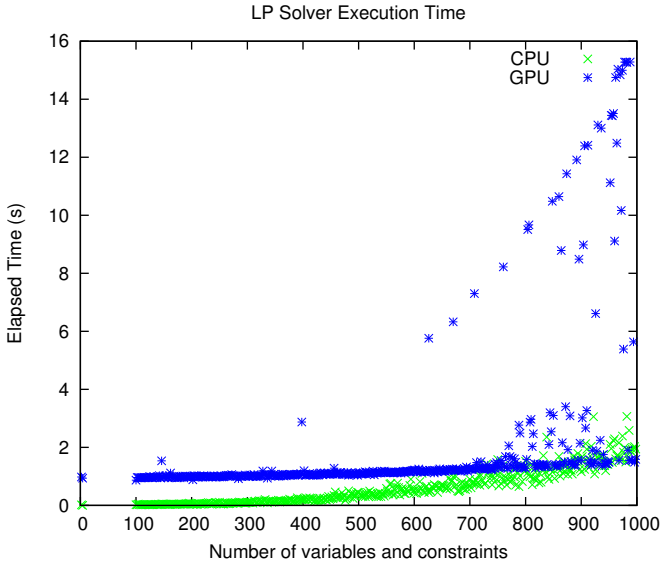

Figure 5.3: Elapsed time for the two versions of the LP solver until 1000 constraints.

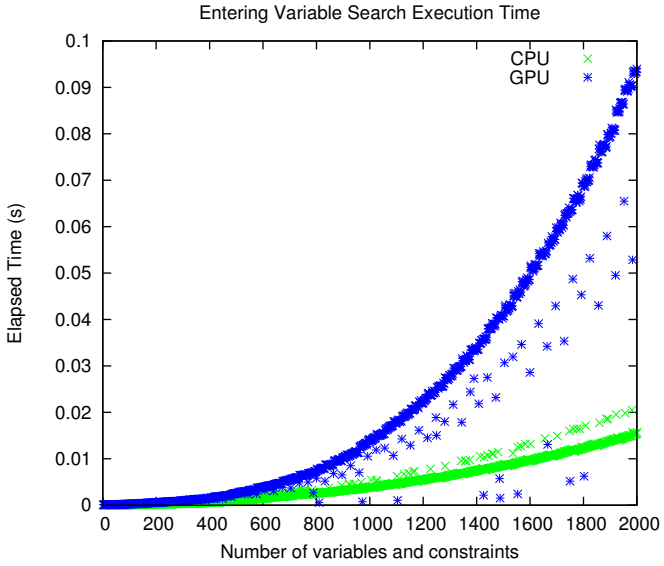

Figure 5.4: Time to search for the entering variable.

we save time in transferring data, that requires at most times on the order of 10^{-3}. Anyway the routine requires an almost constant latency of 0.9s independently from the problem size. That is exactly the amount of time that can be better perceived when working with smaller problems.

5.3.1 Speedup Analysis

Figure 5.7 shows the overall speedup, while Figures 5.8, 5.9, 5.10 the local speedups for the three main sub-tasks analyzed so far. The first figure, that shows the overall speedup, puts in evidence a speedup curve growing faster and faster for bigger problems. It begins with a slow step up until a 0.5X factor around 1400 constraints. From that point the speedup factor grows quicker, reaching values between 2X and

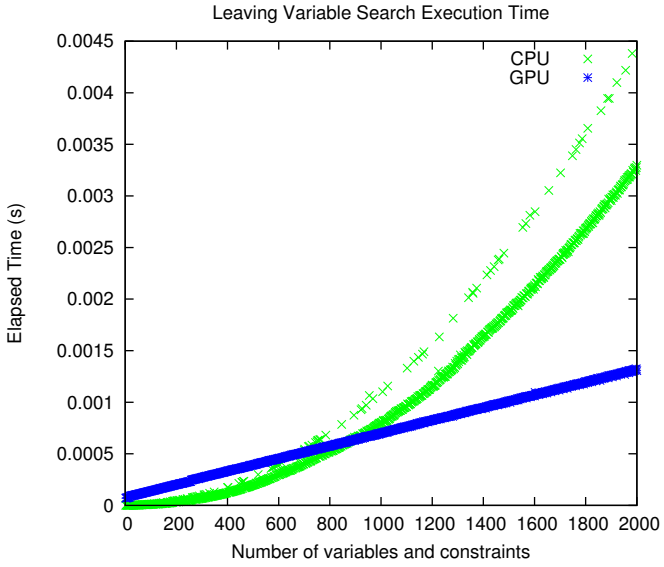

Figure 5.5: Time to search for the leaving variable.

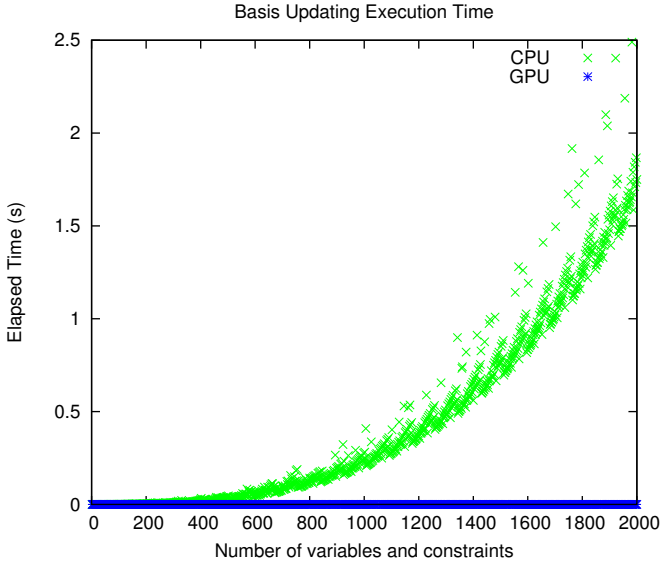

Figure 5.6: Time to update and inverse the basis.

2.5X around 2000 constraints. The CUDA version starts outperforming the serial one from problems with 1600-1800 constraints.
Looking carefully at the revised algorithm, we may notice that in some way the GPU cannot be fully exploited if we completely support the algorithm control flow. Many steps present data-dependency relations that can kill GPU occupancy for small problems. Moreover, there at least two main closure points where there is to transfer just a couple of values, i.e. when entering and leaving variable are found so to test their signs.

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

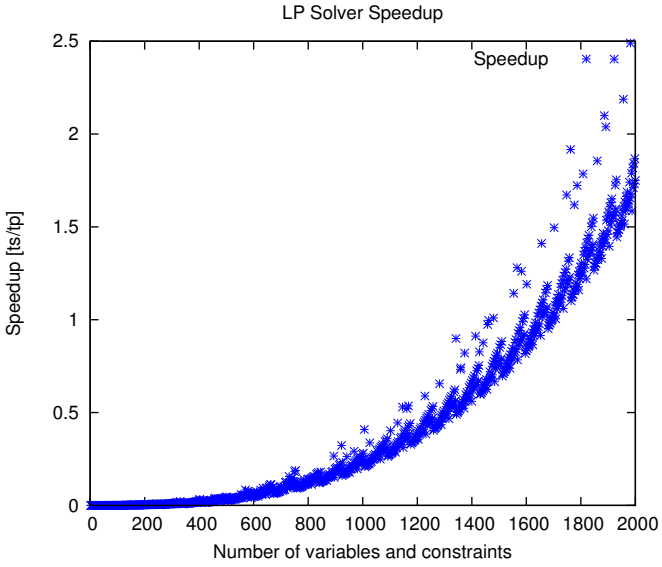

Figure 5.7: Overall speedup.

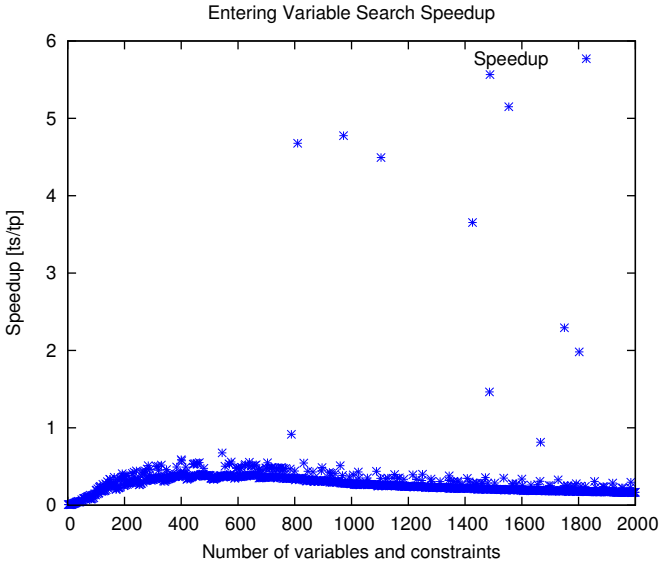

Figure 5.8: Local speedup for the entering variable search task.

BLAS and by CUBLAS. On the other hand, The embarassingly parallel task of building **E** finds rich soil in the GPU execution.

5.3.2 A Few Reflections

From the preciseness analysis we found that in some kind of computations a lack in preciseness apart from giving wrong or less precise results, may also decrease the whole application performance.
Even though some procedures like the entering variable retrieval have been found improvable, they are not the main bottleneck. A possible improvement may be obtained designing a bigger, more compact data structure that may better exploit the transfer bandwidth.

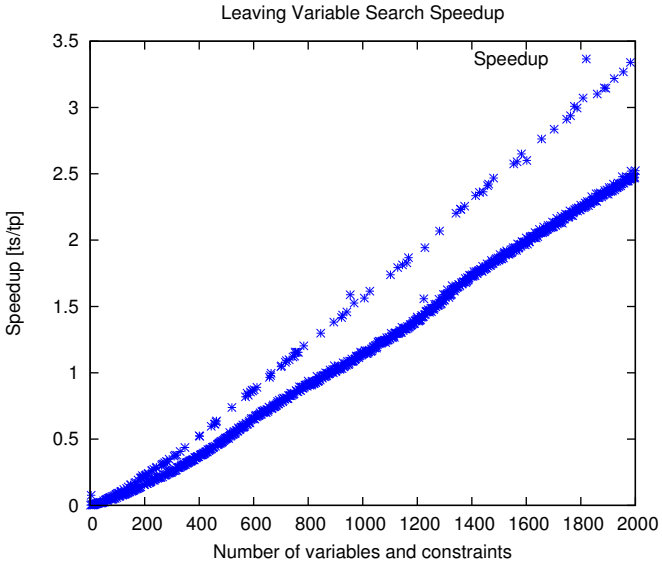

Figure 5.9: Local speedup for the leaving variable search task.

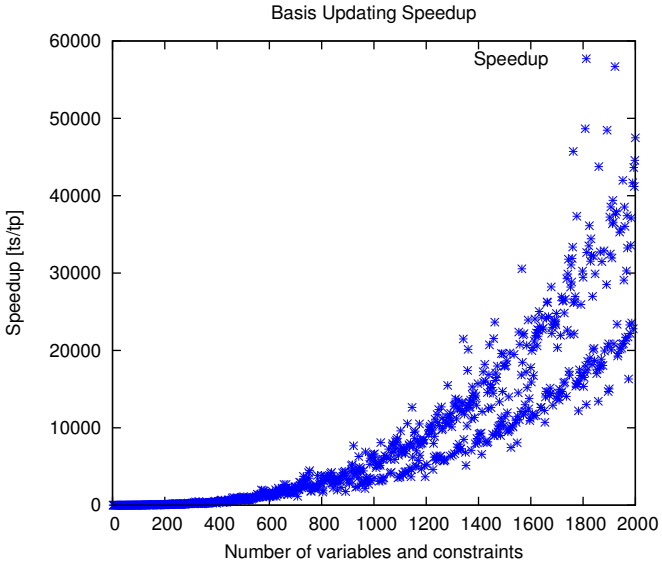

Figure 5.10: Local speedup for the basis updating task.

Finally, observing the whole experiment, we can say that the starting assumption of doing GPGPU using solutions shareable with the serial programming paradigm, does not produce great performance. This approach can sometimes dangerously strangle the powerful computing capability of a GPU. From our experiment, we noticed that this happens specially in presence of strong data dependencies in the control flow. Redefining the solution so to remove single value comparisons for example, may perform much better.

Chapter 6

Conclusions and Future Work

In this chapter we recap the main concepts exposed throughout the report, draw our conclusions and list some potential future work.

We started focusing on recent technologies in the GPGPU area, namely the Tesla architecture and the CUDA programming model. CUDA exposes some important key abstractions, such as a hierarchy of synchronizable threads and different levels of memory, that bring to several advantages. Two of them we think are quite relevant since they represent the main difference between the old and the new concept of GPGPU.
A first advantage given by the CUDA programming model is that it allows programmers to directly concentrate on the problem decomposition. Concentrating on details such as graphics aspects, was the main stumbling rock for developers not experts in graphics programming. A second positive aspect comes from the fact that the CUDA programming model fits quite well to the Tesla processing model. This is an important aspect that can help programmers to better exploit hardware capabilities for producing efficient code. Those are two properties that make the CUDA programming environment a good framework for high performance software development.

Afterwards we introduced the linear programming topics. We gave a formal definition for a linear programming problem, together with its geometrical interpretation. We deepen the two main classes of methods used today to face linear programming problem resolution: simplex-based methods, and interior point methods.
We considered both the classic and the revised definition of the simplex method, and the Mehrotra's primal-dual predictor-corrector method. Among all the presented techniques both the revised simplex method and the Mehrotra's method present a solving algorithm suited for GPUs. Some simplifications in the numerical computation of the revised simplex method are possible, and make it, in our opinion, preferable to the Mehrotra's approach.

We then implemented both a sequential and a CUDA versions based on a common, execution-context independent model of the solving technique. With this we mean that we did not rewrite the algorithm so to pander to specific characteristics of

one or the other execution-context. We simply defined the algorithm in a traditional sequential fashion, we implemented the serial Atlas-based version of the application, and finally we wrote the CUDA version based on a straightforward mapping between sequential and parallel programming concepts.
Although it may appear a naïve approach, it was a first attempt to evaluate how much it was possible to gain by simply pairing CUDA with a potentially parallelizable algorithm.

6.1 Conclusions

The project aimed at developing a parallel, GPU-supported version of an established algorithm for solving linear programming problems. A first conclusion is about the time devoted to the development process. Using the new GPGPU approach provided by CUDA, implementing the application took almost as much as the process of writing the serial code. More expensive is instead the debugging, which is still poorly supported. Discovering bugs and errors in a CUDA kernel may require to create extra and/or ad-hoc data structures to move partial computations back and forth to the device. Code readability and comprehensibility are quite improved with respect to past graphics-based examples.
Compared to the sequential application and to previous solutions developed with the older GPGPU methodology, experimental outcomes confirmed that the application written with CUDA scales reasonably well, solving large problems with thousands of variables and constraints. Greeff [10] reports about the impossibility to solve problems with more than 200 variables with its solution. The CUDA version had a performance increase between 2 and 2.5 times over the serial version. However, for smaller problems (less then 900 variables) the serial Atlas-based version still proved to be the best choice. In the light of linear programming P-completeness such a result looks quite appreciable. Local speedup of specific part of the application showed the extreme versatility of GPUs to compute embarassingly parallel tasks, turning out in speedup factors on the order of $1\text{-}4 \cdot 10^4$.
Great attention has to be paid when designing the communication between the central system and the GPU device, which is still probably the main bottleneck in heterogeneous software development. Performance is strongly related to a good data transfer design.
Another important conclusion regards the preciseness factor. It turned out to still be quite critical, specially for performance in those parts where the workload strictly depends on precision, such as comparisons that may anticipate the end of the computation. We used different tolerances to approximate zero with near-zero floating point values. The CPU defeated the GPU from this point of view, being an order of magnitude more precise than the graphics unit.
Finally, it is worth remarking that even though the GPU-supported version is slower than the serial one for smaller problems, the use of the GPU helps offloading the CPU. This is undoubtedly a positive effect, even more if the application is used in a multithreaded environment.

6.2 Future Work

As we mentioned, we applied a simple mapping approach to pass from the serial to the CUDA version of the application. The fact that the GPU only shows speedups above certain problem sizes, should motivate a better redefinition of the solving approach, considering the heterogeneous execution context directly from the algorithm design stage. Furthermore, a redesign of the data structures in view of a more efficient data transfer has also to be taken into account.
It would be interesting also to keep going with the development of optimization software based on GPU in the light of the most recent GPGPU novelties, such as those presented in this report or new ones likely to become effective standards, like OpenCL [3]. It should include the implementation of alternative solving methods like interior point methods, trying to verify which method is less influenced by the precision factor.
We finally think that standardization is also an important goal. The tool should be able to manage problems stored in standard formats accepted by the most used commercial and academic solvers, like the MPS [20] or CPLEX [2] formats. This may allow to compare the application with other efficient software and eventually promote a wider adoption.

Bibliography

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

[2] "ILOG CPLEX," http://www.ilog.com/products/cplex/, last seen Jan. 2008.

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

[4] T. Akenine-Möller and J. Ström, "Graphics processing units for handhelds," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 779–789, May 2008.

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

[6] D. Bertsimas and R. Weismantel, *Optimization Over Integers*. Dynamic Ideas, 2005.

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

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

[9] R. Farber, "CUDA, supercomputing for the masses," http://www.ddj.com/architect/207200659, Apr. 2008, last seen Jan. 2008.

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

[11] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo, *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995.

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

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

[14] L. G. Khachian, "A polynomial time algorithm for linear programming," *Doklady Akademii Nauk SSSR*, vol. 244, no. 5, pp. 1093–1096, 1979, english translation in *Soviet Mathematics Doklady*, vol. 20, pp. 191-194.

[15] J. L. Manferdelli, N. K. Govindaraju, and C. Crall, "Challenges and opportunities in many-core computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 808–815, May 2008.

[16] W. R. Mark, R. S. Glanville, K. Akeley, and M. Kilgard, "Cg: a system for programming graphics hardware in a C-like language," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 896–907, 2003.

[17] M. D. McCool, "Scalable programming models for massively multicore processors," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 816–831, May 2008.

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

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

[20] B. Murtagh, *Advanced Linear Programming: Computation and Practice*. McGraw-Hill, 1981.

[21] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, March/April 2008.

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

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

[24] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[25] J. Peddie, "GPU market defies gravity - so far," http://www.jonpeddie.com/press-releases/gpu_market_defies_gravity_so_far/, Oct. 2008, last seen Jan. 2008.

[26] M. Salahi, J. Peng, and T. Terlaky, "On mehrotra-type predictor-corrector algorithms," *SIAM Journal on Optimization*, vol. 18, no. 4, pp. 1377–1397, Oct. 2007.

[27] L. A. Wolsey, *Integer Programming*. Wiley, 1998.

Appendix A

Linear Programming Solvers

A.1 Serial Version

A.1.1 Main Module: lpsolver.c

```
#include "matman.h"
#include "liblp.h"

#include <sys/types.h>
#include <time.h>
#include <cblas.h>

#define MAX_ITER 1000

/**
 * Arrays' indexes follow the C convention (0 <= i < N)
 **/

// Main problem arrays: costs and constrains
float *c, *A, *b;

// Binv: Basis matrix inverse
// newBinv: temporary matrix inverse for swap purposes
// E: used to compute the inversion using just one mm multiplication
// newBinv = E * Binv
float *Binv, *newBinv, *E;

// e: cost contributions vector used to determine the entering variable
// D, y, yb: arrays used to compute the cost contributions vector
// D = [-c ; A]  y = cb * Binv  yb = [1 y]  e = yb * D
float *D, *y, *yb, *e;

float *I; // Identity matrix Im

// xb: current basis
```

```
// cb: basis costs
// xb = Binv * b
float *cb, *xb;

// A_e: entering variable column of constraint factors
// alpha: the pivotal vector used to determine the leaving variable
// theta: Increases vector
// alpha = Binv * A_e
float *A_e, *alpha, *theta;

// Vector of flags indicating valid increases
// (valid alpha[i]) <==> (theta_flag[i] == 1)
int *theta_flag;

// Vector containing basis variables' indexes
int *bi;

// Constraints matrix dimensions m and n.
// Indexes of the entering and leaving variables.
int m, n, ev, lv;

// Cost to optimize
// z = c * x
float z;

void help();

/****************** MAIN *********************/

int main(int argc, char **argv)
{
  int i, opt, ret;
  FILE *sourcefile;
struct timespec start, end, ev_start, ev_end, lv_start, lv_end,
    b_start, b_end;
struct timespec blas_end;
long nsec;

switch(argc)
{
case 2:
if(strcmp(argv[1],"-h")==0 || strcmp(argv[1],"--help")==0)
   {
help();
   } else
   if((sourcefile = fopen(argv[1], "r")) == NULL)
{
```

```
printf("Error opening %s\n", argv[2]);
return 1;
}
break;
default:
printf("Wrong parameter sequence.\n");
return 1;
}

clock_gettime(CLOCK_REALTIME, &start);

// read m and n
fscanf(sourcefile, "%d%d", &m, &n);
if(m>n)
{
printf("Error: it should be n>=m\n");
return 1;
}
printf("m=%d n=%d\n", m, n);
  printf("Size: %d\n", m*n);

//Initialize all arrays

// c
allocate_array(&c, 1, n);
read_array(sourcefile, c, 1, n);

// b
allocate_array(&b, m, 1);
read_array(sourcefile, b, m, 1);

// A
allocate_array(&A, m, n);
read_array(sourcefile, A, m, n);

//Close source file
fclose(sourcefile);

// Im
  create_identity_matrix(&I, m);

// Binv, newBinv, E
allocate_array(&Binv, m, m);
allocate_array(&newBinv, m, m);
allocate_array(&E, m, m);
// Initialize Binv = Im
memcpy(Binv, I, m*m*sizeof(float));
```

```
// D, y, yb, e
allocate_array(&D, m+1, n);
allocate_array(&y, 1, m);
allocate_array(&yb, 1, m+1);
allocate_array(&e, 1, n);
// Set first element of yb = 1
yb[0] = 1;
// Initialize D = [-c ; A]
memcpy(D, c, n*sizeof(float));
cblas_sscal(n, -1, D, 1);
memcpy(&D[n], A, m*n*sizeof(float));

// cb, xb
allocate_array(&cb, 1, m);
allocate_array(&xb, m, 1);
// Initialize with the last m elements of c
memcpy(cb, &c[n-m], m*sizeof(float));
memcpy(xb, b, m*sizeof(float));

// A_e, alpha, theta
allocate_array(&A_e, m, 1);
allocate_array(&alpha, m, 1);
allocate_array(&theta, 1, m);

// theta_flag & bi
allocate_int_array(&theta_flag, 1, m);
allocate_int_array(&bi, 1, m);
// Initialize with the basis indexes
for(i=0; i < m; i++)
bi[i] = (n-m)+i;

// Optimization loop

i=0;

do {
    /* Timing */
clock_gettime(CLOCK_REALTIME, &ev_start);
    /* Timing */
    // y = cb*Binv
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
        1, m, m, 1, cb, m, Binv, m, 0, y, m);

    memcpy(&yb[1], y, m*sizeof(float));

    // e = [1 y]*[-c ; A]
```

```
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
        1, n, m+1, 1, yb, n, D, n, 0, e, n);
    /* Timing */
clock_gettime(CLOCK_REALTIME, &blas_end);
    /* Timing */
    ev = entering_index(e, n);
    /* Timing */
clock_gettime(CLOCK_REALTIME, &ev_end);
    /* Timing */
    if(e[ev] >= -EPS)
    {
opt = 1;
      break;
    }

// alpha = Binv*A_e
    /* Timing */
clock_gettime(CLOCK_REALTIME, &lv_start);
    /* Timing */
extract_column(A, A_e, ev, n, m);
cblas_sgemv(CblasRowMajor, CblasNoTrans, m, m, 1,
    Binv, m, A_e, 1, 0, alpha, 1);

    compute_theta(xb, alpha, theta, theta_flag, m);
lv = leaving_index(theta, theta_flag, m);
    /* Timing */
clock_gettime(CLOCK_REALTIME, &lv_end);
    /* Timing */
if(lv < 0)
{
opt = 2;
break;
}

    /* Timing */
clock_gettime(CLOCK_REALTIME, &b_start);
    /* Timing */
if(compute_E(E, alpha, I, m, lv))
{
opt = 3;
break;
}
  // Binv = E*Binv
cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
    m, m, m, 1, E, m, Binv, m, 0, newBinv, m);
memcpy(Binv, newBinv, m*m*sizeof(float));
    /* Timing */
```

```
clock_gettime(CLOCK_REALTIME, &b_end);
    /* Timing */
// Update cb
bi[lv] = ev;
cb[lv] = c[ev];

// xb=Binv*b
cblas_sgemv(CblasRowMajor, CblasNoTrans, m, m, 1,
    Binv, m, b, 1, 0, xb, 1);

    i++;
  } while(i<MAX_ITER);

if(opt == 1)
{
cblas_sgemv(CblasRowMajor,CblasNoTrans,
    1, m, 1, cb, m, xb, 1, 0, &z, 1);
printf("Optimum found: %f\n", z);
for(i=0; i<m; i++)
printf("x_%d = %f\n", bi[i], xb[i]);
} else if(opt == 2)
printf("Problem unbounded.\n");
else printf("Problem unsolvable: either qth==0 or loop too long.\n");

// Deallocate arrays
free_array(A);
free_array(b);
free_array(c);
free_array(D);
free_array(E);
free_array(I);
free_array(A_e);
free_array(Binv);
free_array(newBinv);
free_array(cb);
free_array(xb);
free_array(y);
free_array(yb);
free_array(e);
free_array(alpha);
free_array(theta);
free_array(theta_flag);
free_array(bi);

clock_gettime(CLOCK_REALTIME, &end);
//Overall time
nsec = end.tv_nsec-start.tv_nsec;
```

```
if(nsec < 0)
{
nsec = 1E9+nsec;
end.tv_sec-=1;
}

printf("Elapsed time: %.9f\n",
     (double)(end.tv_sec-start.tv_sec)+(double)(nsec*1E-9));

//BLAS entering variable computation time
nsec = blas_end.tv_nsec-ev_start.tv_nsec;
if(nsec < 0)
{
nsec = 1E9+nsec;
blas_end.tv_sec-=1;
}

printf("BLAS entering variable computation time: %.9f\n",
     (double)(blas_end.tv_sec-ev_start.tv_sec)+(double)(nsec*1E-9));

//Entering variable computation time
nsec = ev_end.tv_nsec-ev_start.tv_nsec;
if(nsec < 0)
{
nsec = 1E9+nsec;
ev_end.tv_sec-=1;
}

printf("Entering variable computation time: %.9f\n",
     (double)(ev_end.tv_sec-ev_start.tv_sec)+(double)(nsec*1E-9));

//Leaving variable computation time
nsec = lv_end.tv_nsec-lv_start.tv_nsec;
if(nsec < 0)
{
nsec = 1E9+nsec;
lv_end.tv_sec-=1;
}

printf("Leaving variable computation time: %.9f\n",
     (double)(lv_end.tv_sec-lv_start.tv_sec)+(double)(nsec*1E-9));

//Binv updating time
nsec = b_end.tv_nsec-b_start.tv_nsec;
if(nsec < 0)
{
nsec = 1E9+nsec;
```

```
b_end.tv_sec-=1;
}

printf("Binv updating time: %.9f\n",
     (double)(b_end.tv_sec-b_start.tv_sec)+(double)(nsec*1E-9));

  return 0;
}

void help()
{
    printf("Input format:\n");
    printf("M N <vector c> <vector b> <matrix A>\n");
}
```

A.1.2 liblp.c

```
#include "liblp.h"
#include "matman.h"

int entering_index(float *v, int size)
{
  int i;
  int min_i = 0;
  for(i = 1; i < size; i++)
    if(v[i] < v[min_i])
      min_i = i;
  return min_i;
}

int leaving_index(float *t, int *flag, int size)
{
int i;
int minpos_i = -1;
for(i=0; i< size; i++)
{
if(minpos_i < 0)
{
if((flag[i] > 0) && (t[i] >= -EPS))
minpos_i=i;
} else
{
if((flag[i] > 0) && (t[i] >= -EPS) && (t[i] < t[minpos_i]))
minpos_i=i;
}
}
```

```
return minpos_i;
}

void compute_theta(float *x, float *a, float *t, int *flag, int size)
{
  int i;
  for(i = 0; i < size; i++)
    if(a[i] > 0)
    {
     flag[i]=1;
      t[i]=x[i]/a[i];
    } else flag[i]=0;
}

int compute_E(float *E, float *a, float *I, int size, int li)
{
int i;
float qth = a[li];

if((qth >= -EPS) && (qth <= EPS))
{
printf("qth == 0....exit...\n");
return 1;
}

memcpy(E, I, size*size*sizeof(float));

for(i = 0; i < size; i++)
a[i] = -a[i]/qth;
a[li]=1/qth;

  for(i = 0; i < size; i++)
E[(i*size)+li] = a[i];

return 0;
}

void extract_column(float *M, float *v, int start_i, int stride, int size)
{
int i;
for(i = 0; i<size; i++)
v[i] = M[start_i+(i*stride)];
}

void create_identity_matrix(float **m, int size)
{
```

```
int i;

allocate_array(m, size, size);
for(i=0; i<size; i++)
(*m)[i*size+i] = 1;
}
```

A.1.3 matman.c

```
#include "matman.h"

/**
 *  Allocate array of float initialized to all bits 0.
 *  Returns 1 if there is an error, 0 otherwise
 */
int allocate_array(float **a, int m, int n)
{
if( (*a = (float *)calloc(m*n, sizeof(float))) == NULL )
   return 1;

return 0;
}

/**
 *  Allocate array of int initialized to all bits 0.
 *  Returns 1 if there is an error, 0 otherwise
 */
int allocate_int_array(int **a, int m, int n)
{
    if( (*a = (int *) calloc(m * n, sizeof( int ))) == NULL )
     return 1;

    return 0;
}

// Print an array of float in the proper format
void display_array(char *name, float *a, int m, int n)
{
int i, j;
printf("Array %s:\n", name);
for(i=0;i<m;i++)
{
for(j=0; j<n;j++)
printf("%f ", a[(i*n)+j]);
printf("\n");
}
}
```

```
//Print an array of integer in the proper format
void display_int_array(char *name, int *a, int m, int n)
{
int i, j;
printf("Int array %s:\n", name);
for(i=0;i<m;i++)
{
for(j=0; j<n;j++)
printf("%d ", a[(i*n)+j]);
printf("\n");
}
}
/**
 * Read array from standard input.
 */
int read_array(FILE *file, float *a, int m, int n)
{
    int i, dim;
dim = m*n;
//Data from the standard input.
    for( i = 0; i < dim; i++ )
{
     fscanf(file, "%f", &a[i]); //Get the ith-element of the matrix from
} //the command line, converting it
//from text to float
    return 0;
}

// Release allocated memory
void free_array(void *a)
{
free(a);
}
```

A.2 CUDA Version

A.2.1 Main Module: culpsolver.cpp

```
#include "cumatman.h"
#include "culiblp.h"

#include <sys/types.h>
#include <time.h>
```

```
/**
 * Arrays' indexes follow the C convention (0 <= i < N)
 **/

// Main problem arrays: costs and constrains
float *c, *A, *b, *xb;
int *bi;

float z;

struct timespec ev_start, ev_end, lv_start, lv_end, b_start,
   b_end, alloc_start, alloc_end, dealloc_start, dealloc_end, init_start, init_end;
struct timespec blas_end;
void help();

/****************** MAIN *********************/

int main(int argc, char **argv)
{
  int i, m, n;
  FILE *sourcefile;
  struct timespec start, end, read_start, read_end, hostall_start, hostall_end;
  long nsec;

switch(argc)
{
case 2:
if(strcmp(argv[1],"-h")==0 || strcmp(argv[1],"--help")==0)
   {
help();
   } else
   if((sourcefile = fopen(argv[1], "r")) == NULL)
{
printf("Error opening %s\n", argv[2]);
return 1;
}
break;
default:
printf("Wrong parameter sequence.\n");
return 1;
}

clock_gettime(CLOCK_REALTIME, &start);

// read m and n
fscanf(sourcefile, "%d%d", &m, &n);
```

```
if(m>n)
{
printf("Error: it should be n>=m\n");
return 1;
}
printf("m=%d n=%d\n", m, n);
printf("Size: %d\n", m*n);

//Initialize all arrays


clock_gettime(CLOCK_REALTIME, &hostall_start);

allocate_array(&c, 1, n);
allocate_array(&b, m, 1);
allocate_array(&A, m, n);

clock_gettime(CLOCK_REALTIME, &hostall_end);

clock_gettime(CLOCK_REALTIME, &read_start);
// c
read_array(sourcefile, c, 1, n);
// b
read_array(sourcefile, b, m, 1);
// A
read_array(sourcefile, A, m, n);
clock_gettime(CLOCK_REALTIME, &read_end);


//Close source file
fclose(sourcefile);

// xb
allocate_array(&xb, 1, m);

// bi
allocate_int_array(&bi, 1, m);

z = lpsolve(A, b, c, xb, bi, m, n);

if(isnan(z))
printf("Problem unsolvable: either qth==0 or loop too long.\n");
else if(isinf(z))
printf("Problem unbounded.\n");
else {
printf("Optimum found: %f\n", z);
for(i=0; i<m; i++)
```

```
printf("x_%d = %f\n", bi[i], xb[i]);
}

// Deallocate arrays
free_array(A);
free_array(b);
free_array(c);
free_array(xb);
free_array(bi);

clock_gettime(CLOCK_REALTIME, &end);
nsec = end.tv_nsec-start.tv_nsec;
if(nsec < 0)
{
nsec = 1E9+nsec;
end.tv_sec-=1;
}

printf("Elapsed time: %.9f\n",
     (double)(end.tv_sec-start.tv_sec)+(double)(nsec*1E-9));

//Read computation time
nsec = read_end.tv_nsec-read_start.tv_nsec;
if(nsec < 0)
{
nsec = 1E9+nsec;
read_end.tv_sec-=1;
}

printf("Read time: %.9f\n",
     (double)(read_end.tv_sec-read_start.tv_sec)+(double)(nsec*1E-9));

//Host alloc computation time
nsec = hostall_end.tv_nsec-hostall_start.tv_nsec;
if(nsec < 0)
{
nsec = 1E9+nsec;
hostall_end.tv_sec-=1;
}

printf("Host allocation time: %.9f\n",
     (double)(hostall_end.tv_sec-hostall_start.tv_sec)+(double)(nsec*1E-9));

//BLAS entering variable computation time
nsec = blas_end.tv_nsec-ev_start.tv_nsec;
if(nsec < 0)
{
```

```
nsec = 1E9+nsec;
blas_end.tv_sec-=1;
}

printf("BLAS entering variable computation time: %.9f\n",
     (double)(blas_end.tv_sec-ev_start.tv_sec)+(double)(nsec*1E-9));

//Entering variable computation time
nsec = ev_end.tv_nsec-ev_start.tv_nsec;
if(nsec < 0)
{
nsec = 1E9+nsec;
ev_end.tv_sec-=1;
}

printf("Entering variable computation time: %.9f\n",
     (double)(ev_end.tv_sec-ev_start.tv_sec)+(double)(nsec*1E-9));

//Alloc computation time
nsec = alloc_end.tv_nsec-alloc_start.tv_nsec;
if(nsec < 0)
{
nsec = 1E9+nsec;
alloc_end.tv_sec-=1;
}

printf("Alloc time: %.9f\n",
     (double)(alloc_end.tv_sec-alloc_start.tv_sec)+(double)(nsec*1E-9));

//Dealloc computation time
nsec = dealloc_end.tv_nsec-dealloc_start.tv_nsec;
if(nsec < 0)
{
nsec = 1E9+nsec;
dealloc_end.tv_sec-=1;
}

printf("Dealloc time: %.9f\n",
     (double)(dealloc_end.tv_sec-dealloc_start.tv_sec)+(double)(nsec*1E-9));

//Init computation time
nsec = init_end.tv_nsec-init_start.tv_nsec;
if(nsec < 0)
{
nsec = 1E9+nsec;
init_end.tv_sec-=1;
}
```

```
printf("Init time: %.9f\n",
     (double)(init_end.tv_sec-init_start.tv_sec)+(double)(nsec*1E-9));

//Leaving variable computation time
nsec = lv_end.tv_nsec-lv_start.tv_nsec;
if(nsec < 0)
{
nsec = 1E9+nsec;
lv_end.tv_sec-=1;
}

printf("Leaving variable computation time: %.9f\n",
     (double)(lv_end.tv_sec-lv_start.tv_sec)+(double)(nsec*1E-9));

//Binv updating time
nsec = b_end.tv_nsec-b_start.tv_nsec;
if(nsec < 0)
{
nsec = 1E9+nsec;
b_end.tv_sec-=1;
}

printf("Binv updating time: %.9f\n",
     (double)(b_end.tv_sec-b_start.tv_sec)+(double)(nsec*1E-9));

  return 0;
}

void help()
{
    printf("Input format:\n");
    printf("M N <vector c> <vector b> <matrix A>\n");
}
```

A.2.2 culiblp.cu

```
#include <stdio.h>

#include "culiblp.h"
#include "cumatman.h"

int kn, km, km1;

int *devidx;
float *devred, *devtemp;
```

```
// test stuff
float *tmm, *tmn, *tm1n, *t1m, *t1n, *t1m1;
int *it1m;
// test stuff


float lpsolve(float *A, float *b, float *c, float *xb, int *bi, int m, int n)
{
int i, opt;
cublasStatus stat;

float *devc, *devA, *devb;

// Binv: Basis matrix inverse
// newBinv: temporary matrix inverse for swap purposes
// E: used to compute the inversion using just one mm multiplication
// newBinv = E * Binv
float *devBinv, *devnewBinv, *devE;

// e: cost contributions vector used to determine the entering variable
// D, y, yb: arrays used to compute the cost contributions vector
// D = [-c ; A]  y = cb * Binv  yb = [1 y]  e = yb * D
float *devD, *devy, *devyb, *deve;

// xb: current basis
// cb: basis costs
// xb = Binv * b
float *devcb, *devxb;

// A_e: entering variable column of constraint factors
// alpha: the pivotal vector used to determine the leaving variable
// theta: Increases vector
// alpha = Binv * A_e
float *devA_e, *devalpha, *devtheta;

// Vector of flags indicating valid increases
// (valid alpha[i]) <==> (theta_flag[i] == 1)
int *devtheta_flag;

// Vector containing basis variables' indexes
int *devbi;

//Counter for unbounded solution checking
int *devnum_max;

// Indexes of the entering and leaving variables.
```

```
int ei, li;

// Cost to optimize
// z = c * x
float z;

//Proper dimensions for kernel grids
kn = (int)ceil((float)n/BS);
km = (int)ceil((float)m/BS);
km1 = (int)ceil((float)(m+1)/BS);

//CUBLAS initialization
/* Timing */
clock_gettime(CLOCK_REALTIME, &alloc_start);
/* Timing */
stat = cublasInit();
if(stat != CUBLAS_STATUS_SUCCESS)
{
printf("Device memory allocation failed.\n");
return 1;
}

// c
stat = cublasAlloc(n, sizeof(*c), (void **)&devc);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}

// b
stat = cublasAlloc(m, sizeof(*b), (void **)&devb);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}

// A
stat = cublasAlloc(m*n, sizeof(*A), (void **)&devA);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
```

```
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}

// Binv, newBinv, E
stat = cublasAlloc(m*m, sizeof(*devBinv), (void **)&devBinv);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}

stat = cublasAlloc(m*m, sizeof(*devnewBinv), (void **)&devnewBinv);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}

stat = cublasAlloc(m*m, sizeof(*devE), (void **)&devE);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}

// D, y, yb, e
stat = cublasAlloc((m+1)*n, sizeof(*devD), (void **)&devD);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}

stat = cublasAlloc(m, sizeof(*devy), (void **)&devy);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
```

```
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}

stat = cublasAlloc(m+1, sizeof(*devyb), (void **)&devyb);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}

stat = cublasAlloc(n, sizeof(*deve), (void **)&deve);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}

// cb, xb
stat = cublasAlloc(m, sizeof(*devcb), (void **)&devcb);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}

stat = cublasAlloc(n, sizeof(*devxb), (void **)&devxb);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}

// A_e, alpha, theta
stat = cublasAlloc(m, sizeof(*devA_e), (void **)&devA_e);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
```

```
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}

stat = cublasAlloc(m, sizeof(*devalpha), (void **)&devalpha);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}

stat = cublasAlloc(m, sizeof(*devtheta), (void **)&devtheta);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}

// red, temp, idx
stat = cublasAlloc(km, sizeof(*devred), (void **)&devred);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}

stat = cublasAlloc(km, sizeof(*devtemp), (void **)&devtemp);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}

stat = cublasAlloc(1, sizeof(*devidx), (void **)&devidx);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
printf("Memory allocation failed: lack of resources.\n");
```

```
else printf("Error in allocation.\n");
return 1;
}

// num_max
stat = cublasAlloc(1, sizeof(*devnum_max), (void **)&devnum_max);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}

// theta_flag & bi
stat = cublasAlloc(m, sizeof(*devtheta_flag), (void **)&devtheta_flag);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}

stat = cublasAlloc(m, sizeof(*devbi), (void **)&devbi);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_ALLOC_FAILED)
printf("Memory allocation failed: lack of resources.\n");
else printf("Error in allocation.\n");
return 1;
}
/* Timing */
clock_gettime(CLOCK_REALTIME, &alloc_end);
/* Timing */

/* Timing */
clock_gettime(CLOCK_REALTIME, &init_start);
/* Timing */
//Move A,b,c(,yb,D) on device
stat = cublasSetMatrix(m, n, sizeof(*A), A, m, devA, m);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_MAPPING_ERROR)
printf("Error accessing device memory.\n");
else printf("Setting error.\n");
return 1;
```

```
}

stat = cublasSetVector(m, sizeof(*b), b, 1, devb, 1);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_MAPPING_ERROR)
printf("Error accessing device memory.\n");
else printf("Setting error.\n");
return 1;
}

stat = cublasSetVector(n, sizeof(*c), c, 1, devc, 1);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_MAPPING_ERROR)
printf("Error accessing device memory.\n");
else printf("Setting error.\n");
return 1;
}

//Initialize yb
zeros<<<km1, BS>>>(devyb, 1, m);
init_yb<<<1, BS>>>(devyb);

//Initialize D
init_cInD<<<kn, BS>>>(devc, devD, m+1, n);
init_AInD<<<dim3(kn, km1), dim3(BS, BS)>>>(devA, devD, m, n);

//Initialize devBinv <- Im
init_I<<<dim3(km, km), dim3(BS, BS)>>>(devBinv, m);

//devcb <- devc[n-m] to devc[n]
cublasScopy(m, &devc[n-m], 1, devcb, 1);

//devxb <- devb
cublasScopy(m, devb, 1, devxb, 1);

//devbi[i] = (n-m)+i
init_bi<<<km, BS>>>(devbi, m, n);
/* Timing */
clock_gettime(CLOCK_REALTIME, &init_end);
/* Timing */

i=0;

do {
/* Timing */
```

```
clock_gettime(CLOCK_REALTIME, &ev_start);
/* Timing */
    // y = cb*Binv
cublasSgemm('N', 'N', 1, m, m, 1.0f, devcb, 1, devBinv, m, 0.0f, devy, 1);
cublasScopy(m, devy, 1, &devyb[1], 1);
    // e = [1 y]*[-c ; A]
cublasSgemm('N', 'N', 1, n, m+1, 1.0f, devyb, 1, devD, m+1, 0.0f, deve, 1);
/* Timing */
clock_gettime(CLOCK_REALTIME, &blas_end);
/* Timing */
    ei = entering_index(deve, n);
/* Timing */
clock_gettime(CLOCK_REALTIME, &ev_end);
/* Timing */
    if(ei < 0)
    {
opt = 1;
      break;
    }

// alpha = Binv*A_e
/* Timing */
clock_gettime(CLOCK_REALTIME, &lv_start);
/* Timing */
extract_column(devA, devA_e, ei, n, m);
cublasSgemv('N', m, m, 1.0f, devBinv, m, devA_e, 1, 0.0f, devalpha, 1);

int num_max;

cudaMemset(devnum_max, 0, 1);
compute_theta<<<km, BS>>>(devxb, devalpha, devtheta,
      devtheta_flag, m, devnum_max);
    cudaMemcpy(&num_max, devnum_max, sizeof(int), cudaMemcpyDeviceToHost);

if(num_max == m)
{
opt = 2;
break;
}
li = leaving_index(devtheta, devtheta_flag, m);
/* Timing */
clock_gettime(CLOCK_REALTIME, &lv_end);
/* Timing */

/* Timing */
clock_gettime(CLOCK_REALTIME, &b_start);
/* Timing */
```

```
if(compute_E(devE, devalpha, m, li))
{
opt = 3;
break;
}
  // Binv = E*Binv
cublasSgemm('N', 'N', m, m, m, 1.0f, devE, m, devBinv, m, 0.0f,
     devnewBinv, m);
cublasScopy(m*m, devnewBinv, 1, devBinv, 1);
/* Timing */
clock_gettime(CLOCK_REALTIME, &b_end);
/* Timing */

//bi[lv] = ev;
//cb[lv] = c[ev];
update_bi_cb<<<km, BS>>>(devbi, devcb, devc, li, ei);

// xb=Binv*b
cublasSgemv('N', m, m, 1.0f, devBinv, m, devb, 1, 0.0f, devxb, 1);

    i++;
  } while(i<MAX_ITER);

if(opt == 1)
{
z = cublasSdot(m, devcb, 1, devxb, 1);

stat = cublasGetVector(m, sizeof(*devxb), devxb, 1, xb, 1);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_MAPPING_ERROR)
printf("Error accessing device memory.\n");
else printf("Setting error.\n");
return 1;
}
stat = cublasGetVector(m, sizeof(*devbi), devbi, 1, bi, 1);
if(stat != CUBLAS_STATUS_SUCCESS)
{
if(stat == CUBLAS_STATUS_MAPPING_ERROR)
printf("Error accessing device memory.\n");
else printf("Setting error.\n");
return 1;
}
} else if(opt == 2)
z = INFINITY;
else z = NAN;
```

```
/* Timing */
clock_gettime(CLOCK_REALTIME, &dealloc_start);
/* Timing */
cublasFree(devc);
cublasFree(devb);
cublasFree(devA);
cublasFree(devBinv);
cublasFree(devnewBinv);
cublasFree(devE);
cublasFree(devD);
cublasFree(devy);
cublasFree(devyb);
cublasFree(deve);
cublasFree(devcb);
cublasFree(devxb);
cublasFree(devA_e);
cublasFree(devalpha);
cublasFree(devtheta);
cublasFree(devnum_max);
cublasFree(devtheta_flag);
cublasFree(devbi);
cublasFree(devidx);
cublasFree(devtemp);
cublasFree(devred);

cublasShutdown();
/* Timing */
clock_gettime(CLOCK_REALTIME, &dealloc_end);
/* Timing */

return z;

}

/************** WRAPPERS ***********************/

int entering_index(float *e, int n)
{
  float val_min;
  int min_i = get_min_idx(e, n, &val_min);
return (val_min >= -EPS) ? -1 : min_i;
}

void extract_column(float *M, float *v, int start_i, int stride, int size)
{
cublasScopy(size, &M[R2C(0,start_i,size)], 1, v, 1);
}
```

```
int leaving_index(float *t, int *flag, int size)
{
return get_min_idx(t, size, NULL);
}

int compute_E(float *E, float *alpha, int m, int li)
{
float qth, *devqth; // = a[li];

cudaMalloc((void **)&devqth, sizeof(float));

get_val<<<km, BS>>>(alpha, li, devqth);

cudaMemcpy(&qth, devqth, sizeof(float), cudaMemcpyDeviceToHost);

if((qth >= -EPS) && (qth <= EPS))
{
printf("qth == 0....exit...\n");
return 1;
}

init_I<<<dim3(km, km), dim3(BS, BS)>>>(E, m);

compute_new_E<<<km, BS>>>(E, alpha, m, li, qth);

return 0;
}

int get_min_idx(float *a, int n, float *val)
{
int numBlocks = (int)ceil((float)n/BS);
int size = n;
int min_idx = -1;

cublasScopy(size, a, 1, devtemp, 1);

do
{
reduce_min<<<numBlocks, BS>>>(devtemp, size, devred);

size = numBlocks;
if(numBlocks > 1)
{
cublasScopy(size, devred, 1, devtemp, 1);
numBlocks = (int)ceil((float)numBlocks/BS);
}
```

```
} while(size > 1);

numBlocks = (int)ceil((float)n/BS);
get_idx<<<numBlocks, BS>>>(a, devidx, devred, n);
cudaMemcpy(&min_idx, devidx, sizeof(int), cudaMemcpyDeviceToHost);
if(val != NULL)
cudaMemcpy(val, devred, sizeof(float), cudaMemcpyDeviceToHost);

return min_idx;
}

/************* KERNELS ***********************/

__global__ void zeros(float *a, int m, int n)
{
int i = blockIdx.y*blockDim.y + threadIdx.y;
int j = blockIdx.x*blockDim.x + threadIdx.x;
int s = gridDim.x*blockDim.x;

int id = AT(i,j,s);

if(id<m*n)
{
i = id/n;
j = id%n;
a[R2C(i,j,m)] = 0;
}
}

__global__ void reduce_min(float *f, int n, float *min)
{
int tid = threadIdx.x;
int j = blockIdx.x*blockDim.x + tid;

//Each block loads its elements into shared mem,
//padding if not multiple of BS
__shared__ float sf[BS];
sf[tid] = (j<n) ? f[j] : FLT_MAX;
__syncthreads();

//Apply reduction
for(int s=blockDim.x/2; s>0; s>>=1)
{
if(tid < s) sf[tid] = sf[tid] > sf[tid+s] ? sf[tid+s] : sf[tid];
__syncthreads();
}
```

```
if(tid == 0) min[blockIdx.x] = sf[0];

}

__global__ void get_val(float *f, int index, float *val)
{
int j = blockIdx.x*blockDim.x + threadIdx.x;

if(j == index) *val = f[j];

}

__global__ void get_idx(float *f, int *index, float *val, int n)
{
int j = blockIdx.x*blockDim.x + threadIdx.x;

if(j == 0)
index[0] = -1;
__syncthreads();

if(j < n)
{
float diff = f[j]-val[0];
if(diff>=-EPS && diff<=EPS) atomicCAS(index, -1, j);
}

}

__global__ void init_yb(float *yb)
{

int i = blockIdx.x*blockDim.x + threadIdx.x;

if(i == 0) yb[0] = 1;

}

__global__ void init_cInD(float *c, float *D, int m, int n)
{
int i = blockIdx.y*blockDim.y + threadIdx.y;
int j = blockIdx.x*blockDim.x + threadIdx.x;
int s = gridDim.x*blockDim.x;

int id = AT(i,j,s);

if(id<n)
{
```

```
i = id/n;
j = id%n;
D[R2C(i,j,m)] = -c[id];
}

}

__global__ void init_AInD(float *A, float *D, int m, int n)
{
int i = blockIdx.y*blockDim.y + threadIdx.y;
int j = blockIdx.x*blockDim.x + threadIdx.x;
int s = gridDim.x*blockDim.x;

int id = AT(i,j,s);

if(id<m*n)
{
i = id/n;
j = id%n;
D[R2C(i+1,j,m+1)] = A[R2C(i,j,m)];
}

}

__global__ void init_I(float *I, int m)
{

int i = blockIdx.y*blockDim.y + threadIdx.y;
int j = blockIdx.x*blockDim.x + threadIdx.x;
int s = gridDim.x*blockDim.x;

int id = AT(i,j,s);

if(id<m*m)
{
i = id/m;
j = id%m;
I[R2C(i,j,m)] = (float)(i==j);
}

}

__global__ void init_bi(int *bi, int m, int n)
{

int i = blockIdx.y*blockDim.y + threadIdx.y;
int j = blockIdx.x*blockDim.x + threadIdx.x;
```

```
int s = gridDim.x*blockDim.x;

int id = AT(i,j,s);

if(id<m)
bi[id] = (n-m)+id;

}

//num_max counts how many alpha[i] are <= 0
__global__ void compute_theta(float *xb, float *alpha, float *theta,
       int *theta_flag, int m, int *num_max)
{
int i = blockIdx.y*blockDim.y + threadIdx.y;
int j = blockIdx.x*blockDim.x + threadIdx.x;
int s = gridDim.x*blockDim.x;

int id = AT(i,j,s);

if(id<m)
{
int cond = (alpha[id]>0);
  theta_flag[id]= cond;
    theta[id]=xb[id]/alpha[id]*cond + FLT_MAX*(1-cond);
atomicAdd(num_max, 1-cond);
}
}

__global__ void compute_new_E(float *E, float *alpha, int m,
       int li, float qth)
{
int i = blockIdx.y*blockDim.y + threadIdx.y;
int j = blockIdx.x*blockDim.x + threadIdx.x;
int s = gridDim.x*blockDim.x;

int id = AT(i,j,s);

if(id<m)
{
alpha[id] = -alpha[id]/qth;
if(id==li) alpha[id]=1/qth;

E[R2C(id, li, m)] = alpha[id];
}
}

__global__ void update_bi_cb(int *bi, float *cb, float *c,
```

```
       int li, int ei)
{
int j = blockIdx.x*blockDim.x + threadIdx.x;
//bi[lv] = ev;
//cb[lv] = c[ev];
if(j == li)
{
bi[j] = ei;
cb[j] = c[ei];
}
}
```

A.2.3 cumatman.cu

```
#include "cumatman.h"

/**
 *  Allocate array of float initialized to all bits 0.
 *  Returns 1 if there is an error, 0 otherwise
 */
int allocate_array(float **a, int m, int n)
{
cudaMallocHost((void **)a, m*n*sizeof(float));
// if( (*a = (float *)calloc(m*n, sizeof(float))) == NULL )
//   return 1;

return 0;
}

/**
 *  Allocate array of int initialized to all bits 0.
 *  Returns 1 if there is an error, 0 otherwise
 */
int allocate_int_array(int **a, int m, int n)
{
    cudaMallocHost((void **)a, m*n*sizeof(int));
//    if( (*a = (int *) calloc(m * n, sizeof( int ))) == NULL )
//     return 1;

    return 0;
}

// Print an array of float in the proper format
void display_array(const char *name, float *a, int m, int n)
{
int i, j;
printf("Array %s:\n", name);
```

```
for(i=0;i<m;i++)
{
for(j=0; j<n;j++)
printf("%f ", a[R2C(i,j,m)]);
printf("\n");
}
}


//Print an array of integer in the proper format
void display_int_array(const char *name, int *a, int m, int n)
{
int i, j;
printf("Int array %s:\n", name);
for(i=0;i<m;i++)
{
for(j=0; j<n;j++)
printf("%d ", a[R2C(i,j,m)]);
printf("\n");
}
}
/**
 * Read array from standard input.
 */
int read_array(FILE *file, float *a, int m, int n)
{
    int i,j;

//Data from the standard input.

for(i=0; i<m; i++)
for(j=0; j<n; j++)
{
     fscanf(file, "%f", &a[R2C(i,j,m)]); //Get the ith-element of the matrix from
} //the command line, converting it
//from text to float
    return 0;
}

// Release allocated memory
void free_array(void *a)
{
cudaFreeHost(a);
}
```

Appendix B

Tools

B.1 popmat.c

The application accept as an input a file name and the number of constraints and variables. It generates a LP problem in canonical augmented form, containing random values between 0 and MAX.

```
/**
* popmat.c
* Program for creating LP input files.
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX 100

int main(int argc, char* argv[]) {

FILE *file;
int i, j, m, n;

if(argc < 3) {
fprintf(stderr, "usage: popmat filename m n\n");
exit(1);
}

file = fopen(argv[1], "w");

m = atoi(argv[2]);
n = atoi(argv[3]);
fprintf(file, "%d %d ", m, m+n);
fprintf(stderr, "m n written.\n");
srand(clock());
```

```
for(i = 0; i < n; i++ ) {
fprintf(file, "%f ", (float)(rand()%MAX));
}
for(i = 0; i < m; i++ ) {
fprintf(file, "%f ", (float)0);
}
fprintf(stderr, "c written.\n");
for(i = 0; i < m; i++ ) {
fprintf(file, "%f ", (float)(rand()%MAX));
}
fprintf(stderr, "b written.\n");
for(i = 0; i < m; i++ ) {
for(j = 0; j < n; j++ ) {
fprintf(file, "%f ", (float)(rand()%MAX));
}
for(j = 0; j < m; j++ ) {
fprintf(file, "%f ", (float)(i==j));
}
}
fprintf(stderr, "A written.\n");

fclose(file);

return 0;
}
```

B.2 matgen.py

Given a file name *matgen.py* produces a LP problem set of progressively bigger dimensions.

```
#!/usr/bin/python
#----------------------------
# Config section
#----------------------------
#Set the list of programs to be run
program="popmat"
#Define the directory containing the programs to be run (relative to this script)
programsDir="./"
#Define the number of executions for each program with each matrix
iterations= 1000
#Name of the output file
outFileDir="./matrices/"

#--------------------------------------------
# Do not edit beyond this line
#--------------------------------------------
```

```
from popen2 import popen4
import sys

def main():
    m = 2
    n = 2
    for iteration in range(iterations):
        cmdLine=programsDir + program + " "+ outFileDir
        + sys.argv[1] + str(iteration) + ".in " + str(m) + " " + str(n)
        print "Execution of program: "+ cmdLine
        execOut,execIn = popen4(cmdLine)
        output = execOut.read()
        m = m+2
        n = n+2
    print "Done"

#-------------------------------------------------
#Execution starts here
#-------------------------------------------------
if __name__=="__main__":
    main()
```

B.3 clock.py

The script, given a set of LP problems, is responsible to launch them, expect for their output, and save the results in different formats for further analysis.

```
#!/usr/bin/python
#---------------------------
# Config section
#---------------------------
#Set the list of programs to be run
programs=["lpsolver","culpsolver"]
#Define the directory containing the programs to be run
   (relative to this script)
programsDir="./"
#Define the number of executions for each program with each matrix
iterations= 1

#Name of the output file
outFileName="results.csv"
dataFileName="results.dat"
innerFileName="inner_results.dat"
#-----------------------------------------------
# Do not edit beyond this line
#-----------------------------------------------
from popen2 import popen4
```

```
import re
import sys

def getFileList(wildcardedFileNames):
    """Gets the list of files from command line. If some
       filenames contain
       wildcards, they are divided in single file names"""

    import glob
    fileList=[]

    """Each fileName from the command line can contain
       wildcards, therefore it may be a list of files.
       This two "for" extract the list of files in every
       group and adds the files, one by one, to the list
       of file names."""
    for fileName in wildcardedFileNames:
        tmpList=glob.glob(fileName)
        for oneFile in tmpList:
            fileList.append(oneFile)

    return fileList



def main():
    #Get the file list
    files=sys.argv[1:]
    fileList=getFileList(files)
    #print fileList
    #Open the output file and prepare its heading
    print "Preparing the output file..."
    OUTFILE=file(outFileName,"w")
    DATAFILE=file(dataFileName, "w")
    INNERFILE=file(innerFileName, "w")
    OUTFILE.write("Program;Matrix file name;
       # constraints;
       # variables;Matrix size;Elapsed time[ns];Optimum\n")
    DATAFILE.write("# constraints;# variables;Matrix size;
        lpsolver time[ns];culpsolver time[ns];Speedup\n")
    INNERFILE.write("# constraints;# variables;Matrix size;
        lpsolver ev_time[ns];lpsolver lv_time[ns];
        lpsolver b_time[ns];
        culpsolver ev_time[ns];culpsolver lv_time[ns];
        culpsolver b_time[ns];ev_speedup;lv_speedup;
        b_speedup\n")
    print "Done"
```

```
#Prepare the extraction regexp
varRE=re.compile("m=(\d+) n=(\d+)")
sizeRE=re.compile("Size: (\d+)")
elapsedRE=re.compile("Elapsed time: (\d+.\d+)")
optRE=re.compile("Optimum found: (\d+.\d+)")
evRE=re.compile("Entering variable computation time: (\d+.\d+)")
lvRE=re.compile("Leaving variable computation time: (\d+.\d+)")
bRE=re.compile("Binv updating time: (\d+.\d+)")
noptRE=re.compile("^Problem")

for fileName in fileList:
    for iteration in range(iterations):
        opt = []
        times = []
        evTimes = []
        lvTimes = []
        bTimes = []
        for p in [0,1]:
            cmdLine=programsDir + programs[p] + " " + fileName
            print "Execution #"+str(iteration+1)+" of program: "
                + programs[p] + " with matrix in " + fileName
            execOUT,execIN= popen4(cmdLine)
            print "Waiting for the results of the determinant
                calculation"
            output= execOUT.read()
            print "Extracting informations from the output"
            #Extract
            numCons=varRE.search(output).group(1)
            numVar=varRE.search(output).group(2)
            size=sizeRE.search(output).group(1)
            times.append(elapsedRE.search(output).group(1))
            evTimes.append(evRE.search(output).group(1))
            lvTimes.append(lvRE.search(output).group(1))
            bres = bRE.search(output)
            if type(bres) == type(noptRE.search("Problem")):
                bTimes.append(bres.group(1))
            else:
                bTimes.append("NaN")
            res = optRE.search(output)
            if type(res) == type(noptRE.search("Problem")):
                opt.append(res.group(1))
            else:
                opt.append("NaN")
            if p==0:
                DATAFILE.write(str(numCons)+"\t"+str(numVar)
                   +"\t"+str(size)+"\t"+str(times[0])+"\t")
```

```
                    INNERFILE.write(str(numCons)+"\t"+str(numVar)
                       +"\t"+str(size)+"\t"+str(evTimes[0])+"\t"
                       +str(lvTimes[0])+"\t"+str(bTimes[0])+"\t")
                else:
                    DATAFILE.write(str(times[1])+"\t"
                        +str(float(times[0])/float(times[1]))+"\n")
                    INNERFILE.write(str(evTimes[1])+"\t"+str(lvTimes[1])
                        +"\t"+str(bTimes[1])+"\t"
                        +str(float(evTimes[0])/float(evTimes[1]))
                        +"\t"+str(float(lvTimes[0])/float(lvTimes[1]))+"\t"
                        +str(float(bTimes[0])/float(bTimes[1]))+"\n")
                OUTFILE.write (programs[p]+";"+fileName+";"+str(numCons)
                   +";"+str(numVar)+";"+str(size)+";"+str(times[p])
                   +";"+str(opt[p])+"\n")
            if opt[0] == opt[1]:
                OUTFILE.write("Fitting: OK\n");
            else:
                OUTFILE.write("Fitting: KO\n");
print "Done"
    print "Data saved in "+ dataFileName
    print "Summary saved in "+ outFileName
    print "Inner data saved in "+ innerFileName

    OUTFILE.close()
    DATAFILE.close()
    INNERFILE.close()

#-------------------------------------------------
#Execution starts here
#-------------------------------------------------
if __name__=="__main__":
    main()
```