Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and
Electrical Engineering
Department of Computer and Information Science

Master Thesis

Linear programming on Cell/BE

by

Åsmund Eldhuset

Supervisor: Dr.Ing. Lasse Natvig
Co-supervisor: Dr. Anne C. Elster

Trondheim, June 1, 2009

Abstract

(TODO:)

Acknowledgements

(TODO:)

Contents

List of Figures

List of Tables

List of Symbols and Abbreviations

Abbreviation	Description	Definition
Cell BE	Cell Broadband Engine	page 13
ILP	Integer linear programming	page 4
ILP	Instruction-level parallelism	page 13
LP	Linear programming	page 3
LS	Local Store	page 14
MFC	Memory Flow Controller	page 14
PPE	PowerPC Processor Element	page 14
PPU	PowerPC Processor Unit	page 14
SPE	Synergistic Processor Element	page 14
SPU	Synergistic Processor Unit	page 14

Listings

Chapter 1

Introduction

(TODO:)

Task description (Natvig: This is the task description I entered in DAIM. Should probably be changed a bit...) The aim of the project is to implement a parallel linear solver for large sparse problems on the Cell BE using the Simplex method. Interior point methods may also be investigated.

(TODO: A paragraph about Miriam)

Chapter 2

Background

(TODO: Chapter introduction)

2.1 Linear programming

(ITP: Section introductions)

This section is primarily based on [8], [2](TODO: and [5] if we write about artificial variables).

2.1.1 Problem formulation. Standard and slack forms

The term *linear programming* (LP) refers to a type of optimisation problems in which one seeks to maximise or minimise the value of a linear function of a set of variables[1]. The values of the variables are constrained by a set of linear equations and/or inequalities. Linear programming is a fairly general problem type, and many important problems can be cast as LP problems — for instance, shortest path problems and maximum flow problems (see [2]). However, the true virtue of linear programming stems from its ability to model a vast range of optimisation problems for which specialised algorithms do not exist, including many situations from economics and industry processes.

The following framed text is an example of a simple linear programming problem. We will use this example throughout this section to illustrate how the linear programming algorithms work.

[1] Hence, LP is not (as the name would seem to suggest) a programming technique. The name originated in the 1940s, when "program" referred to military supply plans and schedules(TODO: citation).

— Example —

A company owns a factory that makes two kinds of products based on two different raw materials. (Natvig: This example just uses some random numbers; I will construct an example, probably using only integers, that can be solved neatly in a few iterations.) The profit the company makes per unit of product A is \$30, and the profit of product B is \$20. Producing one unit of A requires 1 unit of raw material R and 1 unit of raw material S; one unit of B requires 2 units of R and 1 unit of S. The company possesses 40 units of R and 50 units of S. We make the simplifying assumptions that all prices are constant and cannot be affected by the company, and that the company is capable of selling everything it produces. The company's goal is to maximise the profit, which can be described as $30x_1 + 20x_2$, where x_1 is the number of units of product A and x_2 is the number of units of product B. The following constraints are in effect:

- $x_1 + x_2 \leq 40$ (the production of A and B cannot consume more units of raw material R than the company possesses)
- $2x_1 + x_2 \leq 50$ (same for raw material S)
- $x_1, x_2 \geq 0$ (the company cannot produce negative amounts of its products)

Note that in regular LP problems, one cannot restrict the variables to be integers — in fact, adding this requirement produces a new kind of problem known as *integer linear programming* (ILP), which is NP-hard[2]. It is also, in general, a requirement that all variables are nonnegative. This is often the case in real-world problems that deal with physical quantities, but problems involving variables that may be negative as well as positive can still be modeled by rewriting each original variable as a difference of two nonnegative variables.

The function to be optimised is called the *objective function*. In the real world situation that gives rise to an optimisation problem, the function may contain a constant term, but it can be removed since that will affect all possible solutions in the same way. The objective function can then be written as $\zeta = c_1x_1 + c_2x_2 + \ldots + c_nx_n = \sum_{j=1}^{n} c_jx_j$, where the c_j are constants. The variables in the objective function are often called *decision variables*, since our task is not only to find the optimal value of the objective function, but also which variable values that yield

[2]NP-hardness is a term from complexity theory, which deals with the relative difficulties of solving different kinds of problems. The only known algorithms for solving NP-hard problems require an amount of time that is exponential in the size of the problem, which renders those algorithms useless for many real life problem sizes. For further reading on complexity theory, consult [4].

this function value. Throughout this report, we will consistently use n to refer to the number of decision variables and m to refer to the number of equations and/or inequalities. The variables will typically be labelled x_1 through x_n.

Standard form An LP problem is commonly called a *linear program*. The equations and inequalities that (together with the objective function) constitute an linear program may be represented in different forms. We shall first consider the *standard form*, in which only less-than-or-equal-to inequalities with all variables on the left hand side are allowed. (TODO: Why are not less-than allowed?) A problem containing equalities of the form $a_{i1}x_1 + \ldots + a_{in}x_n = b_i$ may be rewritten by splitting each equality into two inequalities: $a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i$ and $-a_{i1}x_1 - \ldots - a_{in}x_n \leq -b_i$. Also, the goal must be to maximise the objective function — if the original problem is to minimize ζ, we let our objective function be $-\zeta$. A linear program in standard form can be expressed as follows:

Maximise

$$\zeta = \sum_{j=1}^{n} c_j x_j \tag{2.1}$$

with respect to

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i, \quad \text{for } i = 1, \ldots, m. \tag{2.2}$$

$$x_1, \ldots, x_n \geq 0 \tag{2.3}$$

Slack form The other common representation is *slack form*, which only allows a set of equations (and a nonnegativity constraint for each variable). A slack form program should be produced by rewriting a standard form program. An inequality of the form $a_{i1}x_i + \ldots + a_{in}x_n \leq b_i$ is converted to an equation by adding a *slack variable* w_i. Together with the condition that $w_i \geq 0$, the equation $a_{i1}x_1 + \ldots + a_{in}x_n + w_i = b_i$ is equivalent to the original inequality (whose difference, or "slack", between the left and right hand sides is represented by w_i). When the program is constructed in this manner, each slack variable only appears in excactly one equation, which is an important property that will be utilised later. A linear program in slack form can be expressed as follows:

Maximise

$$\zeta = \sum_{j=1}^{n} c_j x_j \tag{2.4}$$

with respect to

$$w_i = b_i - \sum_{j=1}^{n} a_{ij} x_j, \quad \text{for } i = 1, \ldots, m. \tag{2.5}$$

$$x_1, \ldots, x_n, w_1, \ldots, w_m \geq 0 \tag{2.6}$$

— Example —

In standard form, our example is expressed as

Maximise

$$\zeta = 30x_1 + 20x_2$$

with respect to

$$x_1 + x_2 \leq 40$$
$$2x_1 + x_2 \leq 50$$
$$x_1, x_2 \geq 0$$

In slack form, it becomes

Maximise

$$\zeta = 30x_1 + 20x_2$$

with respect to

$$w_1 = 40 - x_1 - x_2$$
$$w_2 = 50 - 2x_1 - x_2$$
$$x_1, x_2, w_1, w_2 \geq 0$$

A proposed solution (that is, a specification of a value for each variable) of a linear program in slack form is called:

Feasible if it does not violate any of the constraints

Infeasible if it violates any constraint

Basic if it consists of setting all variables except the slack variables to zero

Optimal if it is feasible and no other feasible solutions yield a higher value for the objective function

(TODO: Move this paragraph to next section? Natvig says "Ta med etter behov".) The *linear programming theorem* (TODO: Is it actually called this? Find something to cite) states that the optimal solution of a linear program, if it exists, occurs when at least m variables are set to zero. (TODO: Combinatorics. Mention cycling here?)

2.1.2 The simplex method

The *simplex method*[3], developed by George Dantzig[3], was the first systematic approach for solving linear programs. It requires the linear program to be in slack form. The initial coefficients and constants are written down in a *tableau* that will change as the method progresses. The nonnegativity constraints are not represented anywhere, but are implicitly maintained by the method. Because the equations will undergo extensive rewriting, it will be convenient to not distinguish the slack variables from the other variables, so we will relabel w_i to x_{n+i-1} for $i = 1, \ldots, m$. Thus, the total number of variables is $n + m$. Furthermore, we will use overlines over the coefficients in the tableau to denote their *current* value (which will change in each iteration of the simplex method), and the indices of the coefficients will refer to the coefficients' position within the tableau — for instance, $-\overline{a}_{ij}$ is located in row i, column j. We also introduce a constant term $\overline{\zeta}$ (initially zero) in the objective function, which will help us keep track of the best function value we have found so far. The topmost row and leftmost column are not really a part of the tableau; they are simply headers — the topmost row shows which variables correspond to which columns, and the leftmost column shows the slack variables for each row. The first actual tableau row (below the double line) contains the objective function coefficients $\overline{c}_j$; the first actual tableau column (to the right of the double line) contains the $\overline{b}_i$ constants, and the rest of the tableau contains the negatives of the coefficients from the equations: $-\overline{a}_{ij}$. Initially, $\overline{c}_j = c_j$, $\overline{b}_i = b_i$, and $\overline{a}_{ij} = a_{ij}$. With $n = 3$ and $m = 3$, the initial tableau will look like this:

		x_1	x_2	x_3
ζ	$\overline{\zeta}$	$\overline{c}_1$	$\overline{c}_2$	$\overline{c}_3$
x_4	$\overline{b}_1$	$-\overline{a}_{11}$	$-\overline{a}_{12}$	$-\overline{a}_{13}$
x_5	$\overline{b}_2$	$-\overline{a}_{21}$	$-\overline{a}_{22}$	$-\overline{a}_{23}$
x_6	$\overline{b}_3$	$-\overline{a}_{31}$	$-\overline{a}_{32}$	$-\overline{a}_{33}$

Note that this is essentially just a tabular version of the standard form — for instance, the last row is interpreted as the equation $x_6 = \overline{b}_3 - \overline{a}_{31}x_1 - \overline{a}_{32}x_2 - \overline{a}_{33}x_3$.

[3]The reason for not calling it the "simplex *algorithm*" is that there exist several versions of the method, and that the general method formulation is somewhat underspecified because it does not say how to choose the pivot elements.

— *Example* —

In tableau form, our example becomes

		x_1	x_2
ζ	0	30	20
x_3	40	-1	-1
x_4	50	-2	-1

Note that w_1 and w_2 have been renamed to x_3 and x_4, respectively.

The variables are partitioned into two sets. The variables in the leftmost column (at the left side of the equations) are referred to as the *basic variables*, and the variables inside the tableau are called *nonbasic variables*. At any stage of the method, the set of the indices of the basic variables is denoted $\mathcal{B}$, and the set of nonbasic indices is denoted $\mathcal{N}$. Initially, $\mathcal{N} = \{1, \ldots, n\}$, and $\mathcal{B} = \{n+1, \ldots, n+m\}$. The sizes of the basic and nonbasic sets are constant, with $|\mathcal{B}| = m$ and $|\mathcal{N}| = n$.

For now, let us assume that the solution that is obtained by setting all nonbasic variables to zero is feasible (which is the case only if all of the b_i are nonnegative); we will remove this restriction later. This trivial solution will provide a lower bound for the value of the objective function (namely, the constant term). We will now select one nonbasic variable x_j and consider what happens if we increase its value (since all nonbasic variables are currently zero, we cannot decrease any of them). Since our goal is to maximise the objective function, we should select a variable whose coefficient c_j in the objective function is positive. If no such variables exist, we cannot increase the objective function value further, and the current solution (the one obtained by setting all nonbasic variables to zero, so that $\zeta = \overline{\zeta}$) is optimal — we can be certain of this since linear functions do not have local maxima.

(TODO: relabel w_i) It seems reasonable to select the variable with the greatest coefficient, say, x_e. How far can we increase this variable? Recall that each line in the tableau expresses one basic variable as a function of all the nonbasic variables; hence we can increase x_e until one of the basic variables becomes zero. Let us look at row i, which is now reduced to $w_i = b_i - a_{ie}x_e$ since all nonbasic variables except x_e are zero. If a_{ie} is positive, the value of w_i will decrease as x_e increases, so the largest allowable increase is limited by b_i. Thus, by setting $x_e = \frac{b_i}{a_{ie}}$, w_i becomes zero. However, other equations may impose stricter conditions. By looking at all rows where a_{ie} is positive, we can determine an l such that $\frac{b_l}{a_{le}}$ is minimal and set $x_e = \frac{b_l}{a_{le}}$. This will cause x_l to become zero. If all a_{ie} are nonnegative, we can increase x_e indefinitely without any w_i ever becoming negative, and in that case, we have determined the linear program to be *unbounded*; the method should report this to the user and terminate.

— Example —

Recall the tableau:

		x_1	x_2
ζ	0	30	20
x_3	40	-1	-1
x_4	50	-2	-1

Since 30 is the greatest objective function coefficient, we select x_1 to be increased. x_3 becomes zero if $x_1 = \frac{b_1}{a_{11}} = \frac{40}{1}$, and x_4 becomes zero if $x_1 = \frac{b_2}{a_{12}} = \frac{50}{2}$. The latter is the most restrictive constraint, so x_4 will become zero when we increase x_1.

The next step, called *pivoting*, is an operation that exchanges a nonbasic variable and a basic variable. The purpose of pivoting is to produce a new situation in which no b_i is negative, so that we can repeat the previous steps all over again. The nonbasic variable that was selected to be increased, x_e, is called the *entering variable*, since it is about to enter the collection of basic variables. x_l, which becomes zero when x_e is increased appropriately, is called the *leaving variable*, since it is to be removed from said collection. Keep in mind that since x_l is a basic variable, it only occurs in one equation, namely

$$x_l = b_l - \sum_{j \in \mathcal{N}} a_{lj} x_j. \tag{2.7}$$

We can eliminate the entering variable from (and introduce the leaving variable into) the set of *nonbasic* variables (the "main" part of the tableau) by rewriting Equation 2.7:

$$x_e = \frac{1}{a_{le}} \left(b_l - x_l - \sum_{j \in \mathcal{N} - \{e\}} a_{lj} x_j \right). \tag{2.8}$$

Now that we have an expression for x_e, we can substitute it into all of the other equations — this will eliminate x_e and introduce x_l into the rest of the tableau.

For all $i \in \mathcal{B} - \{l\}$, we have:

$$x_i = b_i - \sum_{j \in \mathcal{N}} a_{ij} x_j \tag{2.9}$$

$$= b_i - a_{ie} x_e - \sum_{j \in \mathcal{N} - \{e\}} a_{ij} x_j \tag{2.10}$$

$$= b_i - \frac{a_{ie}}{a_{le}} \left(b_l - x_l - \sum_{j \in \mathcal{N} - \{e\}} a_{lj} x_j \right) - \sum_{j \in \mathcal{N} - \{e\}} a_{ij} x_j \tag{2.11}$$

$$= \left(b_i - \frac{a_{ie}}{a_{le}} b_l \right) - \frac{a_{ie}}{a_{le}} x_l - \sum_{j \in \mathcal{N} - \{e\}} \left(a_{ij} - \frac{a_{ie}}{a_{le}} a_{lj} \right) x_j. \tag{2.12}$$

Although this might look complicated, it amounts to subtracting $\frac{a_{ie}}{a_{le}}$ times the tableau row for x_l from all other tableau rows (including the objective function row), and then (TODO:)

Equation 2.8 is the new form of the tableau row that originally corresponded to the basic variable x_l. The new row, which corresponds to x_e, can be easily obtained from the old one by dividing the row by a_{le} and setting the coefficient of what is now x_l to $\frac{1}{a_{le}}$.

Finally, we remove l from $\mathcal{B}$ and add it to $\mathcal{N}$, and remove e from $\mathcal{N}$ and add it to $\mathcal{B}$.

and adding appropriate multiples of it to each of the other equations: (TODO: Complete this) This step is called a *pivot*. After pivoting, we again have a tableau in which all b_i are nonnegative, and the entire process may be repeated.

A 3×3 tableau will look like this after one pivot with x_2 as the entering variable and x_5 as the entering variable:(TODO:)

		x_1	x_5	x_3
ζ	$\overline{\zeta}$	c_1	c_2	c_3
x_4	$b_1 - \frac{b_2}{a_{22}}$	$\frac{-a_{11}}{a_{22}}$	$\frac{-a_{12}}{a_{22}}$	$\frac{-a_{13}}{a_{22}}$
x_2	$\frac{b_2}{a_{22}}$	$\frac{-a_{21}}{a_{22}}$	$\frac{-a_{22}}{a_{22}}$	$\frac{-a_{23}}{a_{22}}$
x_6	$\frac{b_3}{a_{22}}$	$\frac{-a_{31}}{a_{22}}$	$\frac{-a_{32}}{a_{22}}$	$\frac{-a_{33}}{a_{22}}$

— Example —

After one pivot with x_1 as the entering variable and x_4 as the leaving variable, we get the following tableau:

		x_4	x_2
ζ	750	-15	5
x_3	15	0.5	-0.5
x_1	25	-0.5	-0.5

For the next pivot operation, only x_2 can be selected as the entering variable, which causes x_3 to be selected as the leaving variable. After the pivot, the tableau looks like this:

		x_4	x_3
ζ	900	-10	-10
x_2	30	1	-2
x_1	10	-1	1

Since all objective function coefficients are now negative, we have reached an optimal solution with the value $\zeta = \overline{\zeta} = 900$. This solution value is obtained by setting the nonbasic variables (x_3 and x_4) to 0, in which case $x_1 = 10$ and $x_2 = 30$. We can easily verify that these variable values do not violate any constraints, and by substituting the values into the original objective function, we can verify that the optimal value is indeed $\zeta = 30x_1 + 20x_2 = 30 \cdot 10 + 20 \cdot 30 = 900$.

Degeneracy and cycling (TODO: Briefly discuss degenerate pivots.) A tableau is *degenerate* if (TODO:). Degeneracy may cause trouble because a pivot on a degenerate row will not cause the objective function value to change. With severely bad luck, the algorithm may end up cycling through a number of degenerate states. This, however, rarely happens — according to [8], (TODO:)

Initialisation

The method presented so far is capable of solving linear programs whose initial basic solution (the one obtained by setting all nonbasic variables to 0) is feasible. (TODO: Phase I and Phase II) This may not always be the case. We get around this by introducing an *auxiliary problem* which is based on the initial problem and is guaranteed to have a basic feasible solution, and whose solution will provide us with a starting point for solving the original problem. (TODO: Complete this)

Formal algorithm statement

(TODO: Use the `algorithm` package to give a compact description of the simplex method) (TODO: Should ideally be recognisable in the real code; maybe reference the real code here (or the other way around?))

Complexity and numerical instability

(TODO:)

(ITP: Other stuff that should perhaps be added: geometric interpretation; duality)

2.1.3 Interior point algorithms

2.1.4 Use of LP to solve advanced flow problems

A *flow network* is a graph where a *flow* of some substance (expressed in e.g. (TODO: spell "litres") per second) is associated with each edge. In addition, each edge may have upper and lower bounds (known as *capacities*) on the flow value, and possibly a *cost* that will be incurred per unit of flow that is sent through the edge. The goal may, for instance, be to send as much flow as possible from a designated *source* node to a designated *sink* (destination) node, or to send a certain flow as cheaply as possible. Other variations are also possible. If there are no lower bounds and no costs, there exist efficient algorithms for the flow problem, such as the Edmonds-Karp algorithm[2]. In more complex situations, no specialised algorithms exist, but LP comes to the rescue. Cormen et al.[2] give a good overview of how to express a flow problem as an LP problem, which we (TODO: spell "summarise") here:

- There is one variable for each edge, expressing the amount of flow through that edge. (TODO: Cormen has two?)
- (TODO: Finish)

(TODO: Consult Miriam on this)

2.1.5 Existing LP solvers

ILOG CPLEX

CPLEX, developed by the company ILOG, is the industry standard LP solver(Natvig: Who/what can I cite here?). Being proprietary closed-source software, we cannot examine its inner workings (but they are probably too complex for this project). While our department does not have a CPLEX license, we can still to some extent compare the answers from our solvers to those that CPLEX gives — sites such as (TODO: citation) provide CPLEX' answers to the `netlib` problem

sets, and Miriam has a licence that they can use to find the answers to their own data sets. (TODO: Something on why Miriam doesn't just use CPLEX rather than bothering with PS3?)

GLPK

Gnu Linear Programming Kit

Unfortunately, the code base is extremely large, comprising more than (TODO:) lines of C code distributed across nearly 100 files. While only a handful of these files contain functionality that is directly related to the simplex method, reverse engineering it still would be a daunting task — especially given that their coding conventions apparently calls for very short variable names.

GLPK is released by its authors under version 3 of the GNU General Public License.

OOPS

retroLP

As opposed to virtually all other LP solvers, retroLP[9] implements the original simplex method, not the revised method. The former is advantageous for dense problems, which occur in some special applications such as "wavelet decomposition, digital filter design, text categorization, image processing and relaxations of scheduling problems."[10] As compared to GLPK, the code is fairly short and readable — but it still consists of (TODO:) lines.

retroLP is released by its authors under version 2 of the GNU General Public License.

2.2 Cell Broadband Engine

The *Cell Broadband Engine* (Cell BE) is a single chip multiprocessor architecture jointly developed by IBM, Sony and Toshiba. The initial design goals was to create an architecture that would be suitable for the demands of future gaming and multimedia applications (meaning not only high computational power, but also high responsiveness to user interaction and network communications), with a performance of 100 times that of Sony PlayStation 2[6]. Several obstacles to such goals exist; in particular the infamous *brick walls*[1]:

Memory wall (TODO:)

Power wall (TODO:)

ILP wall *Instruction-level parallelism* (ILP) techniques such as pipelines and (TODO:)

2.2.1 Architecture

Overview

The Cell BE consists of one *PowerPC Processor Element* (PPE) and eight *Synergistic Processing Elements* (SPE)

PPE

PowerPC Processor Unit (PPU) Separate register files for fixed-point, floating-point, and vector. 32 SIMD registers.

SPE

Unified register file with 128 128-bit registers

Synergistic Processor Unit (SPU)

Memory bus and DMA controller

Base addresses (both in local storage and in system memory (TODO: correct?)) for all DMA transfers must be aligned on a 16-byte (quadword) border(TODO: term?), and the data to be transferred must be a multiple of 16 bytes. Performance is improved if aligned, whole cache lines (128 bytes(TODO: verify)) are transferred at a time.

Local Store (LS) *Memory Flow Controller* (MFC)

Another method that is available for communication between the cores is

2.2.2 Programming methods

The `vector` data type

Compiler intrinsics

Compiler directives

`__attribute__((aligned(16)))`, `spu_sel`, `__builtin_expect`, `_align_hint`, `mallo`
loop unrolling, function inlining (watch for code size!)

2.2.3 Tools and libraries

(Natvig's comment: Good: which libs are used in the project? Better: Which libs are relevant for the project?)

BlockLib

[11]

CellSS

[7]

Chapter 3

Design

(TODO: Chapter introduction)

3.1 Overall approach

(TODO: Gradual, step by step approach)

3.2 Initial experiments

3.2.1 Arithmetic performance

(Natvig: Should we do this ourselves, or find someone who has already done it?)

All data in registers

Single precision

Double precision

All data in LS

Single precision

Double precision

Double buffering of data from main storage

(Natvig: Maybe we can find out that data transfer takes so much time that the DP performance loss doesn't have too much of an impact?)

Single precision

Double precision

3.3 Dense simplex

In order to become familiar with programming the Cell BE, we initially implemented a few versions of the simplex method for dense problems. (Natvig's comment: This can be justified when we have a task description and "angrepsmåte")

3.3.1 PPE version

(TODO: Far from finished) As described in Section 2.2.1, the PPE supports SIMD instructions (also referred to as vector instructions) capable of operating on four single precision floating point values simultaneously. Since the simplex method primarily consists of row operations on the tableau, it is an excellent target for such vectorisation — the only problem is the low arithmetic intensity, which may reduce performance because a lot of data needs to be loaded into the registers, and only a very simple and fast operation is being performed on each element before it is thrown out again.

(TODO: Something on why we chose C++?)

(Natvig's comment: Avoid the "not invented here" syndrome. Look into reusing existing code/libraries) We wrote a class called `Matrix`, which represents a dense matrix and supports standard matrix operations. The initial version was nonvectorised (SISD) and used just standard C++. Rewriting this to utilise the vector operations involved only a few fairly trivial steps:

- Use `malloc_align` instead of **`new`** to get memory blocks that are aligned on proper boundaries (TODO: Not sure if this was even necessary)
- Pad the rows with zeroes such that the number of elements in each row is a multiple of four, so that the vector operations will not "fall of the end" of each row
- Rewrite the loops in the matrix operation functions (such as `addRows` and `multiplyRow` to use vector operations. (Natvig's comment: Preferrably pseudocode here) `data` is a pointer to the array that contains the entire matrix. `physicalCols` is the number of columns rounded up to the nearest multiple of `VECTOR_WIDTH`, which is set to 4.

```
void Matrix::addRows(int sourceRow, int destinationRow,
    float factor) {
  vector float factor_v = (vector float){factor, factor,
      factor, factor};
```

```
  vector float * source_v = (vector float *)(data +
      sourceRow * physicalCols);
  vector float * destination_v = (vector float *)(data +
      destinationRow * physicalCols);
  for (int j = 0; j < physicalCols / VECTOR_WIDTH; ++j) {
    destination_v[j] = vec_madd(source_v[j], factor_v,
        destination_v[j]);
  }
}
```

(TODO: Do loop unrolling as well?)

3.3.2 SPE version

Our approach is fairly obvious[1]:

1. The PPE, which initially holds the entire tableau, distributes the tableau rows evenly among the SPEs (TODO: SPE or SPU?), giving each SPE a batch of consecutive rows.

2. The first SPE analyses the objective function to determine the leaving variable and sends the column number to the PPE, which distributes this number to the other SPEs. If no leaving variable was found, the optimal solution has been found, and the SPEs are asked to send their basic variable values to the PPE and terminate. (TODO: which pivot rule?)

3. Each SPE determines the strictest bound (that is imposed by its subset of the rows) on the value of the leaving variable and sends the bound (TODO: and the corresponding row number) to the PPE.

4. The PPE determines which SPE that "wins" and requests this SPE to transfer the pivot row to main memory; afterwards, all the other SPEs are requested to receive this row(TODO: wording). If no SPEs found a finite bound, the problem is unbounded, and the SPEs are asked to terminate.

5. Each SPE performs row operations on its part of the tableau, using the pivot row, and notify the PPE upon completion. Go to step 2.

[1] After having written the application, we found that [9] essentially uses the same approach, albeit for cluster computers with MPI.(Natvig: I'm trying to express that although it's not difficult to come up with this approach, I *did* do it myself, before finding that paper. Is that something I should do?)

3.4 Sparse simplex

3.5 Dense interior point

3.6 Sparse interior point

Chapter 4

Implementation and testing

(TODO: Chapter introduction)

4.1 Simplex algorithm

4.2 Test plan

4.2.1 Unit testing

(TODO:)

4.2.2 Large data sets

(TODO: Something on the `netlib` LP problem set)

4.2.3 (TODO: Other implementations)

Chapter 5

Evaluation

(TODO: Chapter introduction)

5.1 Performance measurements

(TODO: Describe system specifications and how timing was performed)

5.1.1 (TODO: What to measure)

5.1.2 (TODO: How to measure)

5.2 Results

5.2.1 Dense simplex

5.2.2 Sparse simplex

5.2.3 Dense interior point

5.2.4 Sparse interior point

5.3 Discussion

Chapter 6

Conclusion

(TODO:)

6.1 Experiences

6.2 Future work

Bibliography

[1] K. Asanovíc, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick, *The Landscape of Parallel Computing Research: A View from Berkeley*, Tech. Rep. UCB/EECS-2006-183, Electrical Engineering and Computer Sciences — University of California at Berkeley, December 2006. [cited at p. 13]

[2] T. H. Cormen, C. R. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, McGraw-Hill Science/Engineering/Math, 2nd ed., 2003. [cited at p. 3, 12]

[3] G. Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963. [cited at p. 7]

[4] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979. [cited at p. 4]

[5] F. S. Hillier and G. J. Lieberman, *Introduction to Operations Research*, McGraw-Hill Science/Engineering/Math, July 2004. [cited at p. 3]

[6] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, *Introduction to the cell multiprocessor*, IBM J. Res. Dev., 49 (2005), pp. 589–604. [cited at p. 13]

[7] TODO, *CellSS*, (TODO:), ((TODO:)). [cited at p. 15]

[8] R. J. Vanderbei, *Linear Programming: Foundations and Extensions*, Springer, 2nd ed., 2001. [cited at p. 3, 11]

[9] G. Yarmish, *A Distributed Implementation of the Simplex Method*, PhD thesis, Polytechnic University, March 2001. [cited at p. 13, 19]

[10] G. Yarmish and R. V. Slyke, *retroLP, An Implementation of the Standard Simplex Method*, Tech. Rep. TR-CIS-2001-05, Polytechnic University, August 2001. [cited at p. 13]

[11] M. Ålind, M. V. Eriksson, and C. W. Kessler, *BlockLib: A Skeleton Library for Cell Broadband Engine*, in (TODO:), 2008, p. (TODO:). [cited at p. 14]

Appendices

Appendix A

Code

A.1 Simplex algorithm, first version

matrix.h

```
#include <iostream>

class Matrix;

class RowIndexDescriptor {
public:
  RowIndexDescriptor(const Matrix *, int);
  float & operator [] (int) const;
private:
  const Matrix * matrix;
  int row;
};


class Matrix {
  friend class RowIndexDescriptor;
  friend std::ostream & operator << (std::ostream &, const
     Matrix &);
public:
  Matrix(int, int);
  Matrix(int, int, bool);
  Matrix(const Matrix &);
  ~Matrix();
  int getRows() { return rows; }
  int getCols() { return cols; }
  RowIndexDescriptor operator [] (int) const;
  Matrix operator * (const Matrix &) const;
  Matrix invert() const;
```

```
  void multiplyRow(int row, float factor);
  void addRows(int sourceRow, int destinationRow, float factor)
     ;
  void swapRows(int firstRow, int secondRow);
private:
  int rows;
  int cols;
  float * data;
};

std::ostream & operator << (std::ostream &, const Matrix &);
```

matrix.cpp

```
#include "matrix.h"

using namespace std;

RowIndexDescriptor::RowIndexDescriptor(const Matrix * matrix,
    int row) {
  this->matrix = matrix;
  this->row = row;
}

float & RowIndexDescriptor::operator [] (int col) const {
  return matrix->data[row * matrix->cols + col];
}

Matrix::Matrix(int rows, int cols) {
//  if (rows <= 0 || cols <= 0)
//    throw std::exception();
  this->rows = rows;
  this->cols = cols;
  this->data = new float[rows * cols];
  for (int i = 0;  i < rows * cols; ++i)
    this->data[i] = 0;
}

Matrix::Matrix(int rows, int cols, bool identity) {
//  if (rows <= 0 || cols <= 0)
//    throw std::exception();
  this->rows = rows;
  this->cols = cols;
  this->data = new float[rows * cols];
  for (int i = 0;  i < rows * cols; ++i)
    this->data[i] = 0;
```

```
  if (identity && rows == cols) {
    for (int i = 0; i < rows; ++i) {
      (*this)[i][i] = 1;
    }
  }
}

Matrix::Matrix(const Matrix & source) {
  this->rows = source.rows;
  this->cols = source.cols;
  this->data = new float[source.rows * source.cols];
  for (int i = 0;  i < source.rows * source.cols; ++i)
    this->data[i] = source.data[i];
}

Matrix::~Matrix() {
  delete data;
}

RowIndexDescriptor Matrix::operator [] (int row) const {
  return RowIndexDescriptor(this, row);
}

ostream & operator << (ostream & out, const Matrix & matrix) {
  out << "=== " << matrix.rows << " x " << matrix.cols << " ===
      " << endl;
  for (int r = 0; r < matrix.rows; ++r) {
    out << matrix.data[r * matrix.cols];
    for (int c = 1; c < matrix.cols; ++c)
      out << '\t' << matrix.data[r * matrix.cols + c];
    out << endl;
  }
  out << "======" << endl;
  return out;
}

Matrix Matrix::operator * (const Matrix & other) const {
  //if (cols != other->rows)
    //throw;
  Matrix result(rows, other.cols);
  for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < other.cols; ++j) {
      float sum = 0;
      for (int k = 0; k < cols; ++k) {
        sum += (*this)[i][k] * other[k][j];
      }
      result[i][j] = sum;
```

```
    }
  }
  return result;
}

void Matrix::multiplyRow(int row, float factor) {
  if (factor == 0) return;
  for (int j = 0; j < cols; ++j) {
    (*this)[row][j] *= factor;
  }
}

void Matrix::addRows(int sourceRow, int destinationRow, float
   factor) {
  if (factor == 0) return;
  for (int j = 0; j < cols; ++j) {
    (*this)[destinationRow][j] += (*this)[sourceRow][j] *
       factor;
  }
}

void Matrix::swapRows(int firstRow, int secondRow) {
  if (firstRow == secondRow) return;
  for (int j = 0; j < cols; ++j) {
    float tmp = (*this)[firstRow][j];
    (*this)[firstRow][j] = (*this)[secondRow][j];
    (*this)[secondRow][j] = tmp;
  }
}

Matrix Matrix::invert() const {
  if (rows != cols) throw "Non-square matrices cannot be
     inverted";
  Matrix self(*this);
  Matrix inverse(rows, cols, true);
  for (int rc = 0; rc < cols; ++rc) {
    // Locate row with nonzero in this column
    int searchRow = rc;
    while (searchRow < rows && self[searchRow][rc] == 0)
      ++searchRow;
    if (searchRow == rows)
      throw rc;
    // Swap with current row; now the current row has nonzero
       in this column
    self.swapRows(rc, searchRow);
    inverse.swapRows(rc, searchRow);
    float factor = 1 / self[rc][rc];
```