An FPGA Implementation of the Simplex Algorithm

Samuel Bayliss [#1], Christos-S. Bouganis [#2], George A. Constantinides [#3], Wayne Luk [*4]

[#]*Department of Electrical and Electronic Engineering*
Imperial College London
University of London
South Kensington campus
London SW7 2AZ, UK
[1]samuel.bayliss@imperial.ac.uk
[2]ccb98@imperial.ac.uk
[3]gac1@imperial.ac.uk

[*]*Department of Computing*
Imperial College London
University of London
South Kensington campus
London SW7 2AZ, UK
[4]wl@doc.ic.ac.uk

***Abstract*— Linear programming is applied to a large variety of scientific computing applications and industrial optimization problems. The Simplex algorithm is widely used for solving linear programs due to its robustness and scalability properties. However, application of the current software implementations of the Simplex algorithm to real-life optimization problems are time consuming when used as the bounding engine within an integer linear programming framework. This work aims to accelerate the Simplex algorithm by proposing a novel parameterizable hardware implementation of the algorithm on an FPGA. Evaluation of the proposed design using real problems demonstrates a speed-up of up to 20 times over a highly optimized commercial software implementation running on a 3.4GHz Pentium 4 processor, which is itself 100 times faster than one of the main public domain solvers.**

I. Introduction

Linear programming is a scientific computing application which provides a general framework for describing optimization problems as a linear objective function and a set of linear constraints. A formulation for a maximization problem is shown in (1), where x is a vector with the variables, A is a matrix, and c and b are vectors of coefficients. The vector inequalities are interpreted as satisfied if and only if they are satisfied component-wise. A minimization problem can be formed by negating the objective function coefficients of the maximization problem.

$$\begin{aligned} \max \quad & c^T x \\ \text{subject to} \quad & Ax \leq b \\ & x \geq 0 \end{aligned} \tag{1}$$

Linear programs are characterized by the number of variables used to define the objective function and constraints, n, and the number of constraint equations used to define feasible solutions to the problem, m.

Linear programming is a useful tool for solving all manner of problems in which the objective function to be minimized or maximized is linear and the constraints can be modelled as a system of linear equations. The application of linear programming techniques is commonly associated with operational research problems. Real world applications of linear programming can be found in fields as varied as Aircraft and Crew Scheduling [1], Portfolio Optimization [2] and Staff Rostering [3]. The use of linear programming in graph optimization and set partition problems [4] makes it an important tool for use in many scientific computing applications. In hardware synthesis field, several applications of linear programming and integer linear programming have been reported. In [5], the authors give a scheduling formulation for high level synthesis, where in [6] the authors demonstrate the use of ILP in optimal wordlength allocation in digital hardware.

The Simplex algorithm [7], [4] provides a robust tool for solving problems modelled using a linear programming framework. Almost all the commercial and research tools available for linear programming use some variant of the Simplex algorithm [MINOS, CPLEX]. In 1972, Klee and Minty [8] demonstrated pathological examples where, in the worst-case, the number of iterations of the Simplex algorithm required to find an optimal solution is exponential in the number of constraints. However, Borgwardt [9] derives a probabilistic model which shows that under certain assumptions, the expected number of iterations required varies linearly with the number of constraints. This makes the Simplex algorithm a good candidate for fast practical solutions to linear programming problems.

A class of problems of particular interest are *integer* linear programming problems. These add the constraint that all the variables in any feasible solution take integer values. Real-world problems often require these constraints, since the entities modelled by variables can be indivisible. Moreover, the introduction of integer variables allows logical constraints, such as dichotomy, to be modelled within a linear programming

framework. The branch-and-bound methods used to solve integer linear programs and *mixed* integer linear programs, which contain both integer-constrained and unconstrained variables, typically proceed by a sequence of many non-integer relaxation problems which together are used to derive the final solution. An LP relaxation is the derivation of an LP problem from an ILP by using the same objective function and set of constraints, and replacing the integer variables by continuous constraints. The considerable time needed to solve these relaxation problems often makes integer linear programming an unattractive tool for large problems and sub-optimal heuristic methods are often used in place of ILP even when a suitable problem formulation is available. A faster implementation of the Simplex algorithm which took advantage of opportunities for parallel computation within each iteration, and pipelined relaxation problems to increase throughput would allow us to derive optimal solutions to many ILP problems previously considered too large to be tractable.

This paper outlines research into a stream-based FPGA hardware implementation of the Simplex algorithm designed to perform much faster than traditional load-store processor-based implementations. By exploiting parallelism inherent in the algorithm and eliminating overheads associated with external memory latencies, the proposed hardware architecture significantly out-performs conventional software-based Simplex implementations even at modest clock-speeds. Key contributions include:

- a study of the opportunities for parallelism presented within the Simplex algorithm,
- to our knowledge, the first FPGA implementation of the Simplex algorithm,
- an implementation of the Simplex algorithm, using a 2D block partitioning which scales to useful problem sizes (up to 751 constraints in 751 variables for a Virtex XC4VFX140 device using 18 bits precision). Results demonstrate an up to 20 times speed up over commercial packages.

II. Simplex Algorithm

A. Primal Simplex Algorithm

If the unconstrained solution space is defined in n dimensions (each dimension assumed to be infinite), each inequality constraint in the linear programming formulation divides the solution space into two halves. The convex shape defined in n-dimensional space after m bisections represents the feasible area for the problem, and all points which lie inside this space are feasible solutions to the problem. Figure 1 shows the feasible region for a problem defined in two variables, $n = 2$, and three constraints, $m = 3$. Note that in linear programming, there is an implicit non-negativity constraints for the variables.

The linearity of the objective function implies that the the optimal solution cannot lie within the interior of the feasible region and must lie at the intersection of at least n constraint boundaries. These intersections are known as corner-point-feasible (CPF) solutions. In any linear programming problem

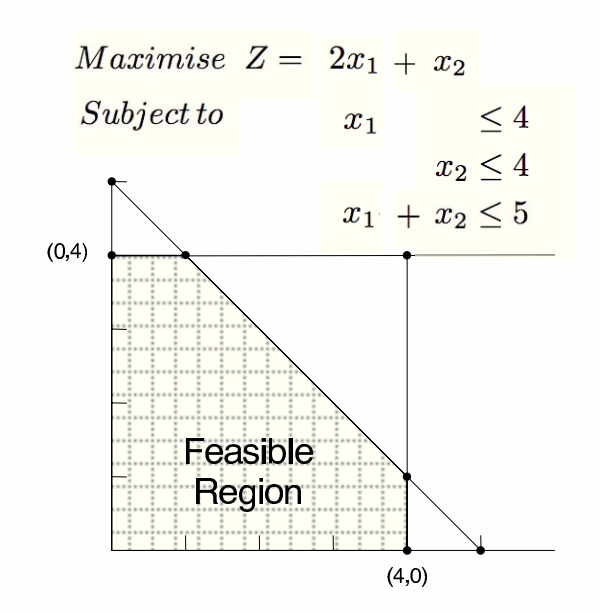

Fig. 1. Feasible Region for problem with $n = 2$ and $m = 3$

with n decision variables, two CPF solutions are said to be *adjacent* if they share $n - 1$ common constraint boundaries.

When interpreted geometrically, the Simplex algorithm moves from one corner-point feasible solution to a better corner-point-feasible solution along one of the constraint boundaries. There are only a finite number of CPF solutions, although this number is potentially exponential in n, however it is not necessary to visit all of them to determine the optimal solution to the problem. The convex nature of linear programming means that there are no local maxima present in the problem which are not also global maxima. Hence if at some CPF solution, no improvement is made by a move to another adjacent CPF then the algorithm terminates and we can be confident that the optimal solution has been found.

This geometric basis for the Simplex algorithm is expressed algebraically as a system of equations. The inequality constraints are converted to equality constraints by the introduction of slack variables. The result is a set of m equations in $n + m$ variables giving us n degrees of freedom in exploring possible solutions. At each Simplex iteration, variables are designated either as basic or non-basic, and the n non-basic variables are set equal to zero. The solution to the resulting system of equations defines a basic solution to the problem. Moving from one basic solution to another involves switching one variable from basic to non-basic and adjusting the values of the basic variables to continue satisfying the system of equations (a pivoting operation).

The Simplex problem is typically presented in the form of a Simplex tableau. The reduced costs, the coefficients derived from pivoting operations on the objective function, are stored in row zero of the $(m + 1)$ x $(n + m + 1)$ tableau, and the right hand size of the constraint equations b is stored in the final column. Figure 2 shows pseudo code for the Simplex

```
SIMPLEX ALGORITHM(A, b, c)
    (N, B, A, b, c, v) ← INITIALIZE(A, b, c)
    # Optimality Test #
1   if c_j ≥ 0 for all index j ∈ N
      then
          # current solution is optimal #
2         return solution
      else
          # Pricing Test #
3         Select an index k ∈ N for which c_k < 0
4         Find the index i ∈ B that has the minimum
          b_i/a_ik and a_ik > 0
5         if such index exists
            then
                # Pivot Step #
6               (N, B, A, b, c, v) ← PIVOT(N, B, A,
                                        , b, c, v, k, i)
            else
7               return "unbounded "
8   goto step 1
```

Fig. 2. Pseudo-Code for Simplex Algorithm

algorithm. Fundamentally the steps taken in a single iteration of the Simplex algorithm are

Select Entering Variable: Choose a pivot column j such that the reduced cost (row 0) is negative. If no column exists such that the reduced cost element is negative then the optimal solution has been found and the algorithm terminates.

Ratio Test: For each positive element in the column indexed by j calculate the ratio $\delta_i = b_i/a_{ij}$. The index i which minimizes the ratio identifies the pivot row. If $\delta_i \leq 0$ for all $i = 1 \ldots m$ then the problem is unbounded, *i.e.* there is no optimal solution.

Pivot: Divide all the elements in the pivot row with index i by a scalar such that the coefficient which lies in the pivot row and pivot column becomes one. Subtract multiples of that row from all the other rows such that all the other elements in the pivot column become zero.

Several opportunities for parallelism can be exploited within each Simplex iteration.

- The pivoting operation used to transform the Simplex tableau on each iteration is performed by subtracting multiples of the pivot row from every other row in the tableau. This operation is typically expensive to perform sequentially on conventional computer hardware. However within a hardware implementation, each of these array operations can be performed in parallel.
- The selection of an entering variable can be performed in parallel using a tree of comparators.
- The ratio test used to select a pivot row can also be performed in parallel.

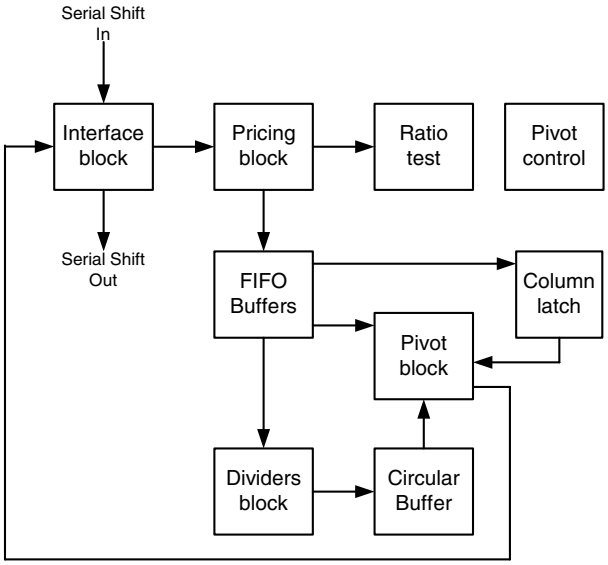

Fig. 3. Block diagram of Simplex implementation

Alongside these opportunities for intra-iteration parallelism, our implementation allows the streaming of several problems in a pipelined fashion through the hardware architecture. This inter-iteration parallelism adds a further performance edge over conventional sequential implementations of the Simplex algorithm.

III. Design Architecture

While [7] shows the expected number of iterations of the Simplex algorithm for certain problems varies linearly with the number of problem constraints, the exponential worst-case iteration count means it is infeasible to consider unrolling the algorithm to solve complete problems in a linear pipeline. Thus a circular pipeline structure is adopted, with each problem iteration feeding back in a circular fashion until an optimal solution is found.

The Simplex algorithm is an iterative algorithm since iteration $n + 1$ is unable to begin execution before iteration n has completed. The inevitable latency of the design means hardware is left idle when working on a single problem. Pipelining several different problems through the hardware allows us to hide the design latency and achieve high throughput. In solving integer linear programming problems, we typically have to solve many different Simplex relaxation problems. The pipelined implementation presented here allows several different relaxations to be processed simultaneously at different stages within the hardware. Figure 3 shows a block diagram for the proposed pipelined architecture. The design uses a 2D block partitioning scheme, breaking the Simplex tableau into small regularly sized blocks, which enter the problem pipeline sequentially. Figure 4 shows the 2D block-partitioning of a Simplex tableau and indicates the position of the reduced costs row within the first p blocks to enter the system, and the right hand column of the linear programming formulation within the corresponding blocks.

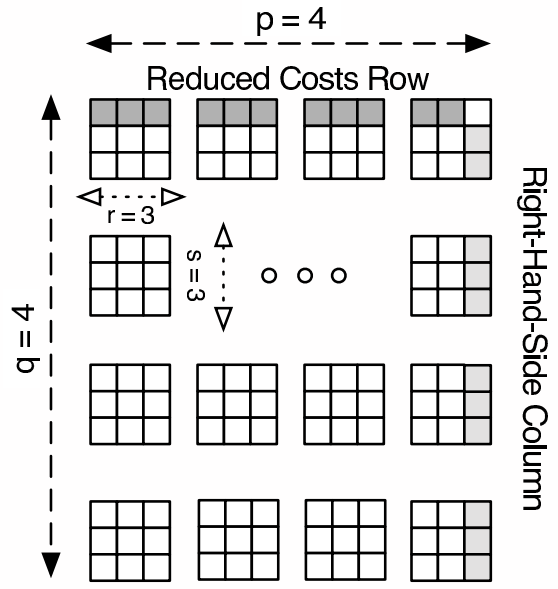

Fig. 4. 2D block partitioning of Simplex tableau when s is the vertical dimension of each block, r is the horizontal dimension of each block, p is the number of blocks in each tableau row and q is the number of blocks in each tableau column.

The 2D block partitioning scheme allows a great deal of flexibility in trading off FPGA area and overall performance. Larger block sizes means more computation can be performed in parallel and therefore reduces the time taken to perform each Simplex iteration. The aspect ratio of the blocks in the design is unconstrained and can therefore be parameterized to match the aspect ratio of a target problem. The 2D partitioning scheme also allows different problems with varying sizes, although sharing a fixed block size, to be streamed through the hardware simultaneously. The ability to interleave problems of different sizes allows many different problems to be solved simultaneously. That has a major impact in the solution of ILP problems where the different relaxations to the ILP have different problem dimensions. In addition, this is a desirable feature in embedded optimization algorithms where decisions required for adaptive behavior have to meet hard deadlines.

A. Interface block

Blocks enter the design in a row-major fashion beginning from the top left-most block of the Simplex tableau. As data leaves the pipeline, data is reintroduced to the pricing block through the interfacing block. This block allows the extraction of optimal problems from the pipeline and the introduction of new problems without disruption to other problems in the pipeline. When the pricing module finds no further negative reduced costs coefficients, the problem is flagged as optimal and passes through the hardware without pivoting. The bandwidth of blocks flowing around the iteration loop even using a modest 3×3 block partitioning far exceeds the bandwidth of regular off-chip interfaces. Therefore multiple iteration cycles through the hardware are necessary to read-out the optimal results and swap-in the next problem. The interfacing block contains a parallel-in serial-out shift register and logic to support the scheduling of new problems, stalling the pipeline if new problems must be entered.

B. Pricing block

A range of different pricing strategies is discussed in the literature for selecting an entering variable for the basis function. Our implementation uses the steepest edge criterion first suggested by Dantzig [7]. This selects the most negative coefficient in the reduced costs row to enter the basis. The first row of blocks to enter the system contains the reduced price coefficients used to select the entering variable column. A binary tree structure of comparators is used to select the most negative coefficient from each block. This coefficient is compared to the most negative coefficient from the preceding blocks in the current problem and if found to be more negative, the value is latched for comparison with subsequent blocks.

The fully pipelined pricing block generates horizontal and vertical sync control signals, selects the appropriate entering column, and latches the entering column and right hand side column from each row of blocks which flow through it. These column vectors are passed to the ratio test block.

C. Ratio test

A number of different strategies can be considered for finding the elements with the smallest ratio from two vectors of numbers. Clearly a division operation can be used to find the ratio of each pair of numbers and the resulting scalar numbers are compared using a tree of comparators. The results published in this paper refer to an implementation using cross-multiplication of the candidate vectors using a sequential multiplier implemented in LUTs within the FPGA.

D. FIFO Buffer Implementation

The full Simplex tableau must be read into the design before the algorithm selects the appropriate pivot row. Hence the pivot operation cannot begin until all the blocks in a given problem have been read into the system. The block elements are stored in FIFO buffers implemented using dual-port Block-RAM. These embedded memories are driven by a two times faster clock derived from an embedded delay-locked-loop.

Alternate cycles are used for sequential access to the data (*i.e.* FIFO mode), using one memory port to read and another to write, and random access to elements held within the FIFO buffer using both memory ports as two independent random access channels. This allows pivot row elements and entering column elements to be pre-loaded from memory whilst data is streamed out through the pivoting block ensuring maximum throughput in the design. The dual-port memory blocks in a modern FPGA claim operation at up to 550MHz and so few implementation problems were encountered in running the memories at double clock speed within our design.

The loading of data from the FIFOs into the circular buffer and the column-latch block used for pivoting is coordinated

by the pivot-control block. This uses an address offset from the beginning of the problem data to load the entering column data and an address offset from the end of the problem data to load pivot row data into the dividers used for pivot operation. The pivot control block also keeps track of parameters for problems within the pipeline such as problem ID, horizontal and vertical block counts, optimality, degeneracy and overflow flags.

E. Circular Buffer and Pivot Blocks

The pivoting operation generates a new basic feasible solution by elementary row operations. The pivot *value* refers to the number which lies in both the entering column and pivot row. The chosen pivot *row* is divided by a scalar value to force the pivot value to one. Multiples of this row are subtracted from each of the other rows in the matrix to force all the other elements in the pivot column to zero.

In the hardware implementation, elements from the pivot row are loaded into a fully pipelined divider from the FIFO buffer using one of the two random access channels. Results from the divider are stored in a circular buffer implemented in distributed RAM.

At the beginning of each row of blocks, the appropriate elements from the pivot *column* are loaded from the FIFO buffers using the second random access channel and latched. The hardware is designed such that as each block $block_{i,j}$ is loaded sequentially from the FIFO, the appropriate portion of the divided row i is retrieved from the head of the circular buffer and multiplied by the value held in the latched column. The data are multiplied together and subtracted from all the elements in the tableau except from the elements belonging in the pivot row.

A block of data are pivoted using a chain of single processing elements. Each processing element contains a fully pipelined multiplier, embedded multipliers are used, two shift registers to match the delay through the divider, and a subtractor, which is implemented using LUT fast-carry chains.

IV. Scalability and numerical stability

The size of problems solvable using the demonstrated Simplex implementation is bounded by the FIFO buffer-size. The implementation uses a single Xilinx Block-RAM per pivoting-element. With an 18 bit wordlength, each FIFO is capable of buffering up to 1024 entries. For the 4×4 block size, this places an upper limit on the problem size, assuming problems have a 1:1 aspect ratio of variables to constraints, of 751×751. This limit constraints the size of the LP problems that can be addressed by the proposed system. However, this limit of the maximum problem size is large enough to fit real-life LP problems that are produced by relaxations of ILP problems. Using more than one Block-RAM per pivoting-element would allow scaling to larger problems.

A study of the numerical stability of the Simplex algorithm in its various forms can be found in [10]. Accumulation of rounding error, both in hardware and in software implementations, is a major cause of instability. This essentially results from the pivoting operation, a problem well understood in numerical analysis in the context of Gaussian Elimination [11]. The proposed implementation is fully parameterizable by wordlength, so that numerical accuracy can be guaranteed; results have been collected for both 12-bit and 18-bit datapaths, although the wordlength required for convergence to the optimal CPF solution will be problem dependent in general.

TABLE I

Synthesis results using different design parameters

Block size	Word-length	Slices	RAMs	MULs	Freq.
4×4	18 bit	10,067	16	16	117MHz
4×4	12 bit	6,139	16	16	130MHz
8×8	18 bit	27,411	64	64	108MHz
12×12	18 bit	48,036	144	144	105MHz

V. Evaluation

A. Synthesis Results

After verification of the behavior of the design using synthetic problems, the design was synthesized using Xilinx XST 7.1 and implemented in a Virtex 4VFX140 device. Table I shows the area and clock-speed achieved varying the block size and word-length used in the design. A 4×4 block size refers to the block partitioning size, where larger blocks imply more parallelism and hence reduced problem latency and increased throughput. The proposed design is pipelined at the block level, able to process one $r \times r$ block per clock cycle. These designs consume between 11% and 84% of the 4VFX140. Using the results in Table I we can interpolate/extrapolate the requirements in area given a block size $r \times r$ and the wordlength w of the system. The main area in a block is allocated to the dividers that are needed for pivoting, which is a quadratic function of the wordlength used. Thus, the area of the design in slices, A_{slices}, can be approximated using (2).

$$A_{slices} = c_1 r w^2 + c_2 r^2 w \tag{2}$$

Using the data from Table I and linear regression, we obtain the values of the coefficients: $c_1 = 6.73$ and $c_2 = 8.02$. Figure 5 shows the predicted area using (2) for different values of the wordlength and the block size. The current synthesized designs are also plotted in the graph.

B. Benchmark Performance

Table II shows five benchmarks selected from the netlib library [12] to test the performance of the design. The selected benchmarks were timed running on a 3.4GHz Pentium 4 with 1GB of RAM, using a public available LP solver, Lp-Solve, and a commercial software CPLEX. The time taken to load the problems into memory and initialize the basic solution were stripped from the times that are reported. Table III illustrates the obtained results from the two software packages and alongside we present the time taken using our smallest hardware implementation which is a 4×4 block partitioning running at 117MHz. It should be noted that the Block-RAM is clocked at double the clock frequency. The results demonstrate that a considerable speed-up is achieved using the proposed

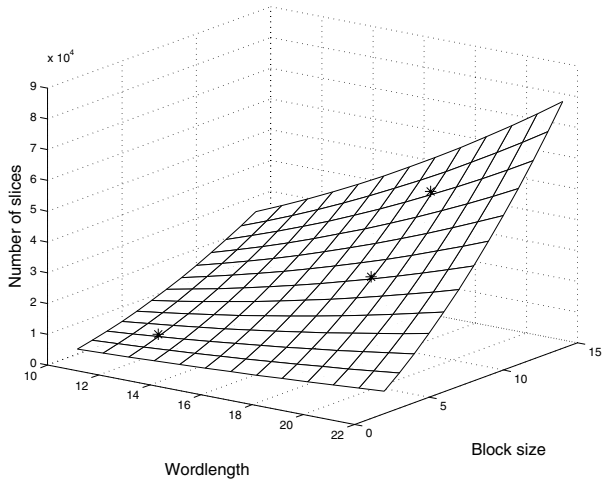

Fig. 5. Prediction of the required area for different values of the design's wordlength and block size. The actual synthesized designs from Table I are also superimposed.

TABLE II

A SELECTION OF BENCHMARK PROBLEMS SELECTED FROM THE NETLIB ONLINE REPOSITORY [12].

Benchmark	Constraints (m)	Variables (n)
Adlittle	57	97
Afiro	28	32
Blend	75	83
Recipe	92	180
Share2b	97	79

architecture even when its performance is compared against a commercial program like CPLEX. The median speedup over the CPLEX software package is 8.9 times.

Although the software time shown in Table III can be measured in microseconds, it should be remembered that this is the time for a *single* Simplex iteration. The number of such iterations when solving an ILP is typically exponential in the problem size. For example, the online repository MIPLIB [[13], arki001] cites an example problem with 1048 constraints and 1388 variables taking one month of CPU time and involving 100 million complete relaxations. Problems that would fit within our existing FPGA design are also reported as taking 14 hours [[13], noswot]. Thus a 10x speedup in solution for the inner-loop of such procedures is a critically important factor in solving large scale integer linear programming problems.

TABLE III

COMPARISON OF PER-ITERATION PERFORMANCE OF SOFTWARE AND HARDWARE IMPLEMENTATIONS OF SIMPLEX (HARDWARE IMPLEMENTATION IS 4×4 BLOCK PARTITIONED DESIGN AT 117MHZ)

Benchmark	Iteration Time			Speed-up
	Software		Hardware	
	Lp-Solve	CPLEX		Lp-Solve (CPLEX)
Adlittle	3.43ms	0.06333ms	0.0030ms	1143 (21.1)
Afiro	6.35ms	0.07571ms	0.0070ms	907 (10.8)
Blend	3.33ms	0.03500ms	0.0035ms	951 (10)
Recipe	6.06ms	0.06333ms	0.0096ms	631 (6.6)
Share2b	3.72ms	0.03082ms	0.0044ms	845 (7)

It should be noted that the degree of speedup over software will be a function of the sparsity of the A matrix in (1), as CPLEX includes sophisticated procedures to take advantage of patterns of sparseness. However, despite this, the proposed architecture achieves up to 20x speedup on these real-world problems, which are not fully dense in nature; even greater speedups are likely on dense problems.

VI. CONCLUSIONS

This paper presents a novel, scalable, and parameterizable architecture for FPGA implementation of the Simplex algorithm. The scalability of the proposed architecture makes it applicable to real-life problems, especially when a pipeline can be filled with many relaxations of an initial large integer linear program. A Xilinx Virtex 4 implementation of the proposed architecture has been achieved, with a datapath clock-rate of 105 to 117MHz and a double-rate memory subsystem, running at 210 to 234MHz. The partitioning of the problem can be varied at design time to match the required device size, and an area model is presented allowing this to be done: the throughput of an $r \times r$ block partition increases quadratically with r, as does the area requirement. The Virtex 4 implementation demonstrates that speedups of up to 20 times (median 8.9 times) are achievable over a state-of-the art commercial solver running on a 3.4GHz PC with 1GB of RAM, and 100 times more compared to a commonly-used public domain solver.

Future work is likely to involve the integration of this design into a larger framework for hardware-based branch-and-bound solution of integer linear programming problems. Moreover, specialisation of the design to sparsity patterns known at design-time appears to be a promising direction for future speed-up on particular classes of problem; robust software solvers often contain such a "toolbox" of special case approaches.

ACKNOWLEDGMENT

The authors wish to acknowledge the financial support of the EPSRC under the platform grant EP/C549481/1 and the UK Research Council (Basic Technology Research Programme "Reverse Engineering Human Visual Processes" GR/R87642/02).

REFERENCES

[1] C. Martin, D. Jones, and P. Keskinocak, "Optimizing on-demand aircraft schedules for fractional aircraft operators," *Inferfaces*, vol. 33, no. 5, 2003.

[2] E. I. Ronn, "A new linear programming approach to bond portfolio management," *Journal of Financial and Quantitative Analysis*, vol. 22, no. 4, 1987.

[3] B. Gendron, "Scheduling employees in quebec's liquor stores with integer programming," *Interfaces*, vol. 35, 2005.

[4] F. S. Hillier and G. J. Lieberman, *Introduction to Mathematical Programming*, 2nd ed. McGraw-Hill Inc, 1995.

[5] L.-Y. Wang and Y.-T. Lai, "Graph-theory-based simplex algorithm for VLSI layout spacing problems with multiple variable constraints," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 8, pp. 967–979, 2001.

[6] G. A. Constantinides, P. Y. K. Cheung, and W. Luk, "Optimum and heuristic synthesis of multiple word-length architectures," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 13, no. 1, pp. 39–57, 2005.

[7] A. Schrijver, *Theory of Linear and Integer Programming*. Wiley and Sons, 1972.
[8] V. Klee and G. J. Minty, *Inequalities, III*. Academic Press, 1972, ch. How good is the simplex algorithm?, pp. 159–175.
[9] K. H. Borgwardt, *The Simplex Method, A Probabilistic Analysis*. Springer-Verlag, 1987.
[10] S. S. Morgan, "A comparison of simplex method algorithms," Master's thesis, University of Florida, 1997.
[11] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. SIAM, 2002.
[12] "http://www.netlib.org/."
[13] T. Achterberga, T. Koch, and A. Martin, "The mixed integer programming library: Miplib 2003," http://miplib.zib.de/, 2003.