KLU Sparse Direct Linear Solver Implementation into NGSPICE

Francesco Lannutti, Paolo Nenzi, Mauro Olivieri
DIET
Sapienza – University of Roma
Roma, Italy
Email: nicolati@fastwebnet.it, nenzi@diet.uniroma1.it, olivieri@diet.uniroma1.it

Abstract—**The simulation of large digital and mixed-signal integrated circuits is one of the challenges of the electronics design automation industry. In this work, a fast linear solver, KLU, is implemented into NGSPICE circuit simulator and its performances have been verified on standard netlists.**

Index Terms—**NGSPICE; KLU; circuit simulation; transient analysis.**

I. Introduction

The electrical-level simulation of large circuits, consisting millions of devices, is still an open issue that limits the possibility of full chip verification and one of the grand challenges in ITRS roadmap [1, 2]. The universal tool (still after 40 years from its conception) for time domain verification of ICs is SPICE (Simulation Program with Integrated Circuit Emphasis). In this context, SPICE refers to all the circuit simulators that are based on the same algorithms of the original UC Berkeley SPICE simulator.

The simulation time of transient analysis in SPICE grows super-linearly with the number of equations that describe the circuit. In [3], it has been reported that the run time per transient analysis iteration scales as $O(N^{1.2})$ (with "N", the number of equations), while the double precision FLOPS (Floating Point Operation per Second) of CPUs only as $O(N^{0.96})$ (with "N" the number of transistors in the CPU). This difference in the exponents implies that time spent for a single transient iteration will continuously increase with the technology scaling.

The efforts to reduce this widening gap can be classified in two categories: specialized algorithms and parallelization. The former approach gave birth to a class of simulators often called "fast-spice", presenting significant speedups against standard simulators for some class of circuits (e.g. RAM). The latter approach has been pursued since the early days of SPICE [4] and parallel implementations are still developed today, in particular for the newly available GPU architectures [5]. Speedups obtained by the parallelization of the transient analysis in SPICE are in the range 2x to 24x, lower than the ones of fast-spices, nevertheless the resulting simulators are not tailored to any specific circuit topology.

The parallelization of SPICE is not trivial because of the structure of the SPICE algorithm for transient analysis, shown in figure 1 (after [4]). The three highlighted operations in the inner loop are the most demanding in term of computing resources: linearization of non-linear devices around the trial operating point, conductances stamping into the modified nodal analysis (MNA) matrix and linear system solution.

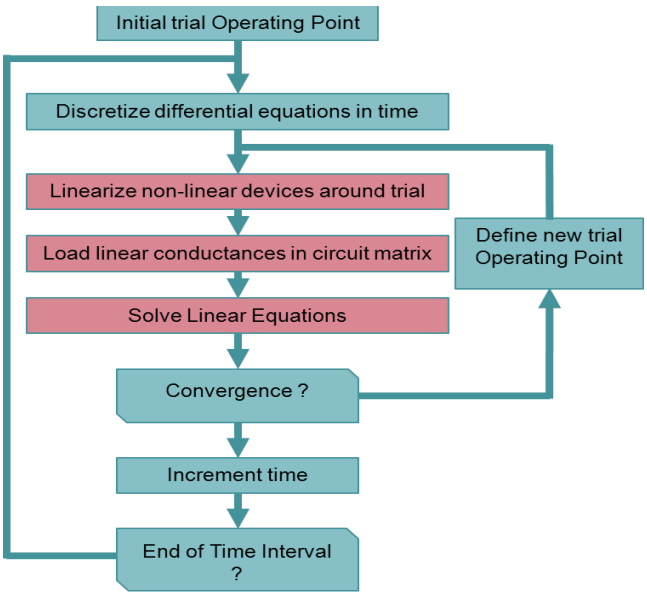

Figure 1. The SPICE algorithm for transient analysis, adapted from [4]. The three boxes in the inner loop are the most time consuming operations

The linearization of non-linear devices is the most easily parallelizable operation because it is intrinsically SIMD (Single Instruction Multiple Data). Furthermore, being the most time consuming operation, it's the most relevant contribution to speedup. Figure 2 shows the percent of the time spent in transient simulation for the netlists in the ISCAS 85 suite [6,7]. The *BSIM3load*, *CAPload* and *RESload* are the functions that compute the linear conductances and load them into the system matrix for BSIM3, capacitors and resistors devices, respectively. Their execution time sums up to 54% of the analysis time. Parallel implementations of linearization phase have been done both for CPUs using OpenMP [8] and GPUs [5]. The NGSPICE circuit simulator [9] implements, as option, the parallelization of device linearization as described in [8] for BSIM3 and BSIM4 devices (data in figure 2 have been computed with this option turned off). The speedup achieved on multicore CPUs is 2.

The next important contribution, from figure 2, is the linear system solution that accounts for 36% of total analysis time (the sum of *spFactor*, *spSolve* and *spClear*). The parallelization of a direct sparse solver (SPICE based circuit simulators use a direct solver) is not trivial because of the irregular structure of the sparse matrix and its sparsity. Most of the elements of a matrix resulting from MNA are zero and the non-zero pattern has no regularity. Thus the solution of a sparse linear system is an I/O intensive problem. Nevertheless the parallelization of sparse direct linear solvers has been attempted since the early days of SPICE [4] or, recently in [10].

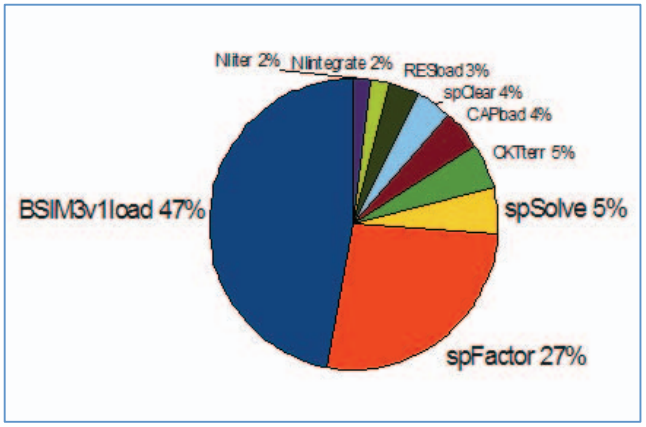

Figure 2. Time spent in different functions of the ngspice code during transient analysis on the ISCAS 85 [6,7] suite. The names correspond to the function names in the code and the time is computed as percent of the time spent for all the netlists in the suite

In order to complete the analysis of time spent in the transient analysis from figure 2, 5% of the time is spent in *CKTterr*, the truncation error calculation, 2% in *NIiter*, the code for Newton-Raphson iterations and 2% in *NIintegrate*, the integrator routine. The remaining 1% (not shown in the figure) is spent in the rest of the simulator.

In this work the implementation of a recently developed serial direct sparse linear solver, providing speedups comparable to parallel implementations is presented. The NGSPICE circuit simulator is used as driver for testing the implementation. In the modified version of NGSPICE, the original solver Sparse (version 1.3) [11] has been replaced with KLU [12] developed by Timothy Davis (University of Florida). KLU is direct solver optimized for the structure of linear systems resulting from the nodal analysis of electronic circuits that has been demonstrated [12] to be up 1.000 times faster than Sparse 1.3 in solving an MNA linear system.

II. Algorithms Analysis

The matrices generated by the modified nodal analysis of electronic circuits (MNA matrices) posses some properties that can be exploited to enhance the performances of direct solvers [12]. The main diagonal of an MNA matrix is almost free of zero-valued elements. The MNA matrix has un-symmetric values with a roughly symmetric non-zero pattern, it is very sparse and if ordered properly leads to sparse LU factors. Moreover, it can be permuted to a block triangular form (BTF) with a single diagonal block that contains most of the non-zero elements, surrounded by other, much smaller, diagonal blocks. The smallest circuit in the ISCAS 85 suite is the c432 ("c" stands for "combinational", and 432 is the number of lines describing the circuit). The circuit is a 27-channel interrupt controller [6] with 36 input and 7 outputs, consisting of 160 logic gates. The transient analysis MNA matrix for c432 has 5007 rows and columns with 27521 non-zero elements (corresponding to 0.11%). The largest block of the matrix consists of 27445 non-zero elements spanning 4930 rows and columns. The remaining elements 76 are singletons (a singleton is a single non-zero element on the main diagonal).

A. Analysis of the Sparse linear solver

Sparse is the default solver in the NGSPICE circuit simulator, developed by Kenneth Kundert [11]. It solves the linear system

$$Ax = b \qquad (1)$$

with "*A*" sparse and "*b*" dense or sparse, and is optimized to solve repeatedly systems with the same non-zero pattern. This is the case of the transient analysis, where the circuit structure (that defines the non-zero of the matrix) does not change during the analysis, and only the numerical values of elements change. Sparse considers the sparse matrix as linked list of nodes each one pointing to the next element in column and to the next element in row. This structure allows the circuit simulator to build the overall system matrix starting from the stamps of the single instances in the circuit. It is not necessary to know a priori the final size of the matrix. The structure of the matrix in sparse is optimized for O(1) access operations, achieved by storing the pointer to each element in the calling routine. Sparse uses the Markowitz preordering to reduce the fill-ins [12] and factor the entire matrix in LU form using partial pivoting. In addition, Sparse can exploit the sparsity of the RHS vector. The solution of a linear system is divided into two steps: factorization and solve.

Sparse has two factorization functions, one for computing the LU factors, called only at the start of the analysis, and the other one for updating the factors with new numerical values, called at each iteration (the *spFactor* in figure 2).

The solve step (the *spSolve* function in figure 2), is the forward and backward substitution of the RHS in the lower and upper triangular matrices L and U to compute the "x" vector.

B. Analysis of the KLU algorithm

KLU is a direct sparse matrix solver, designed by Timothy Davis at the University of Florida [12] tailored to efficiently solve the linear systems resulting from the MNA analysis. In KLU the system matrix is stored in compressed storage column format (CSC). All the KLU operations are performed in a "workspace" memory area that is allocated with the matrix, to reduce the costly memory allocation and copy operations. The algorithm solves the linear system in five steps:

1. Maximum transversal,
2. Block triangular form permutation,
3. Column approximate minimum degree reordering,

4. LU factorization using Gilbert-Peierls algorithm,

5. Forward/backward substitution.

The Maximum Transversal [14] uses a DFS (Depth First Search) on the system matrix to find a permutation that removes all the zeroes on the main diagonal. If the matrix isn't singular, it's always possible to find an un-symmetric column permutation Q that removes all the zeroes on the main diagonal.

After this step the following system is solved:

$$AQy = b \qquad (2)$$

where $Qx = y$. In the next step, the BTF algorithm, based on the Tarjan algorithm [15] is used to find the strongly connected components of the graph representing the circuit. In this step, a symmetric row/column permutation P that converts the matrix into an upper block triangular form is found. This form is particularly advantageous because all the elements below the diagonal blocks are zero and do not contribute to fill-ins and the LU factorization is needed only for the diagonal blocks. The off-diagonal blocks are only linear combination of the unknowns of corresponding diagonal blocks.

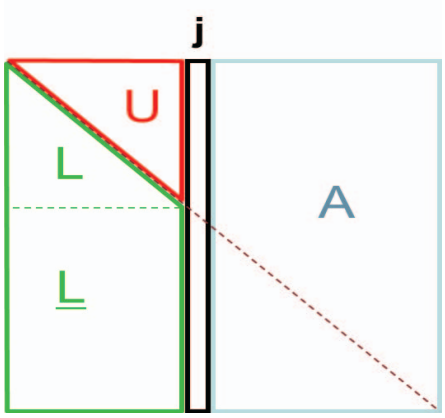

Figure 3. Gilbert Peierls LU factorization process, the picture shows the jth step of the factorization. The LU factorization is built one column at time. The U and L are the LU factors at the j_{th} step, where L represents the part of the matrix not factored yet. The j vector corresponds to the jth column of the generic block A.

The linear system to be solved becomes:

$$PAQP^T y = Pb \qquad (3)$$

The COLAMD (COLumn Approximate Minimum Degree) algorithm [16] is then applied to each diagonal block to reduce the fill-ins in the following factorization step.

KLU uses the Gilbert-Peierls LU factorization (GPLU) [16], to compute the LU factors for each diagonal block. GPLU algorithm factors each block by repeated solution of linear systems. It produces a column of L and a column of U at every step, using the j_{th} column of the generic diagonal block A as right hand side vector (RHS) of the j_{th} triangular system, composed by the $(j-1)_{th}$ columns of L (see Figure 3). The solution vector, at each step, becomes the j_{th} column of L and U (the upper part pertains to U and lower parts to L). The pseudocode in figure 4 describes the algorithm.

The version of GPLU implemented in KLU exploits the sparsity of the block to reduce the number of operations required for factorization to O(flops(LU)) [17].

The system solve step compute the dense solution vector x, through a chain of forward and backward substitutions applied to each LU factored block and block-back substitutions in the off-diagonal blocks.

Gilbert Peierls LU factorization algorithm

```
L = I
for each column j
    b = A(:,j)
    x = L \ b       % see below
    (do partial pivoting)
    U(1:j,j) = x(1:j)
    L(j+1:n, j) = x(j+1:n) / U(j,j)
    end

% Solution of x = L \ b

x = b
for columns I in (1:j-1)
    x(i+1:n) = x(i+1:n) – L(i+1:n, i) * x(i)
end
```

Figure 4. Pseudocode for the basic GP algorithm used by KLU to compute the LU factors of each diagonal block.

C. *Comparison of the performances of the two linear solvers*

The first analysis we did was to compare the performances of both solvers, Sparse and KLU outside of the circuit simulator, using the test tools included in each solver. We dumped the system matrix corresponding to the netlists in the ISCAS 85 suite, with the RHS vector. The netlists characteristics are reported in table 1. For each circuit, the number of equations for the non back-annotated version (std) and for the back-annotated one (ann) is reported.

TABLE I. CHARACTERISTICS OF THE ISCAS85 NETLISTS

ISCAS 85	Netlists characteristics		
	Logic gates	*Functional Blocks*	*Equations (std/ann)*
C432	160	5	5007/9789
C499	202	2	10221/19243
C880	383	7	7998/14975
C1355	546	2	10509/19485
C1908	880	6	15252/23818
C2670	1193	7	23433/40452
C3540	1669	11	33229/49218
C5315	2307	10	50294/78941
C6288	2406	240	45571/67045
C7552	3512	8	67695/101564

Back-annotation has been done using the standard parameters described in [7] and only RC parasites have been considered. The results of this initial analysis are presented in figure 5. For every netlist, KLU executes faster than Sparse 1.3 and, the time difference increases with the dimension of the matrix, as shown in the figure 5. It is interesting to notice that both curves present the same features as the peaking corresponding to the c499_ann and the c5315_ann netlists, due to the fact that the annotated netlist has more elements than the next (not annotated) one. This behavior suggests that both solvers execute in a time proportional to the number of non-zero elements in the matrix, with KLU being more efficient

and asymptotically more efficient (as the curves diverges). The obtained results are consistent with the ones reported by Davis [12].

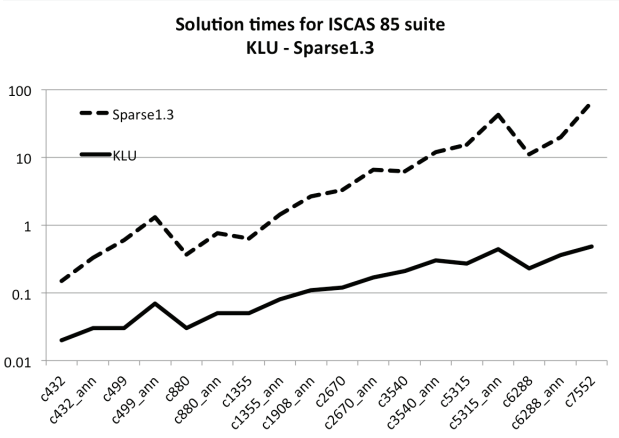

Figure 5. The execution times for the solution of linear systems obtained from the ISCAS85 suite. The "_ann" netlists contain RC back-annotation.

III. NGSPICE IMPLEMENTATION RESULTS

Afterwards KLU solver has been implemented into the NGSPICE circuit simulator to test its performances on the solution of non-linear systems in the transient analysis loop.

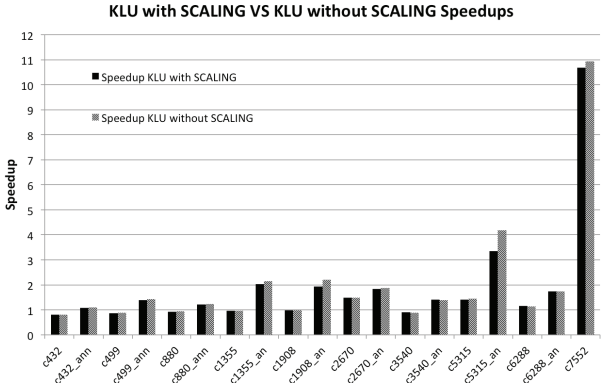

Figure 6. Speedup obtained by KLU (with and without scaling) over Sparse 1.3 solver during the transient analysis on the netlists of ISCAS85.

The results are shown in figure 6, where the speedup for the single transient analysis iteration is reported. KLU results slower than Sparse on the six smaller (of the ten) non-annotated netlists and always faster for the annotated ones. The maximum speedup obtained is 11 for the c7552 netlist with a mean value of 1.8. This behavior on small netlists is imputable to the higher cost of re-factorization in KLU. Sparse function *spFactor* is more performant than it's KLU counterpart *klu_refactor*.

The impact matrix scaling, to improve pivoting, has been verified. Matrix scaling solves numerical stability at the cost of additional computation at each re-factorization. This is clearly visible in figure 6 where, the execution times are higher in fifteen of the nineteen netlists and, almost equal in the remaining cases.

IV. CONCLUSIONS

The implementation of the KLU linear solver into NGSPICE has been presented as a possible solution to the ITRS requirements of reducing the simulation time of large digital and mixed-signal ICs. The KLU solver has been verified to be up to 100 times faster than Sparse in the solution of a linear system coming from MNA analysis, and only 11 times faster in the execution of a transient analysis, when implemented into the circuit simulator. This behavior has been attributed to the higher complexity of KLU refactoring function, compared to the Sparse one, making KLU an effective replacement for very large circuits only. In figure 6 it is evident that considerable speedup is obtained for the c7552 netlist only.

Additional speedup can be obtained by optimizing the implementation of KLU into NGSPICE. The actual research is focused on implementing other linear solvers (UMFPACK and SuperLU) to compare performances on serial and, where available, parallel implementations.

ACKNOWLEDGMENT

The authors would acknowledge Zia Abbas, Antonio Mastrandrea and Francesco Menichelli, for the fruitful discussions, their invaluable support in the implementation of KLU code in NGSPICE and in the automation of simulations and reports that greatly reduced the data analysis time.

REFERENCES

[1] ITRS, "International Technology Roadmap for Semiconductors (Design)," ITRS Design Roadmap, www.itrs.net, p. 19, 2009.

[2] ITRS, "International Technology Roadmap for Semiconductors (Modeling and Simulation)," ITRS Modeling and Simulation Roadmap, www.itrs.net, p. 19, 2009.

[3] N. Kapre, "SPICE^2 – A Spatial Parallel Architecture for Accelerating the SPICE Circuit Simulator", Ph.D. Thesis, California Institute of Technology, 2010.

[4] A. Vladimirescu, "LSI Circuit Simulation on Vector Computers", Memorandum No. UCB/ERL M82/75, 1982.

[5] R. Poore, "GPU-accelerated time-domain circuit simulation," *Custom Integrated Circuits Conference, 2009. CICC*, no. Cicc, pp. 629-632, 2009.

[6] M. C. Hansen, H. Yalcin, J. P. Hayes, "Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering,", *IEEE Design & Test*, *16*(3), 1999.

[7] J. Xu, "SPICE Simulation of ISCAS85 Benchmark Circuit for research" available online at:http://www.ece.uic.edu/~masud/iscas2spice.htm.

[8] T. H. Weng, R.K. Perng, and B. Chapman, "OpenMP Implementation of SPICE3 Circuit Simulator," *International Journal of Parallel Programming*, vol. 35, no. 5, pp. 493-505, Jul. 2007.

[9] Ngspice Circuit Simulator, avalable online at: http://www.ngspice.org.

[10] S. Hutchinson, et al., "The Xyce Parallel Electronic Simulator – An Overview," *Proceedings of the International Conference ParCo2001*, Naples, Italy, September 2001.

[11] K. S. Kundert. Sparse matrix techniques. In *Circuit Analysis, Simulation and Design*, part 1, A. E. Ruehli (editor), North-Holland 1986.

[12] T. Davis and E. P. Natarajan, "Algorithm 907: KLU, a direct sparse solver for circuit simulation problems," *ACM Transactions on Mathematical Software*, vol. 37, no. 3, pp. 36:2 - 36:17, 2010.

[13] H. M. Markowitz, “The elimination form of the inverse and its application to linear programming”, *Management Sci.*, 3 (1957), pp. 255–269.

[14] I. S. Duff, "On algorithms for obtaining a Maximum Transversal," *ACM Transaction,* vol. 7, no. 3, pp. 315-330, 1981.

[15] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal,* 1972.

[16] T. A. Davis, J. R. Gilbert, S. Larimore and E. Ng, "A column approximate minimum degree ordering algorithm,", *ACM Trans. Math. Softw.*, 30, pp. 35-376, 2004.

[17] J. R. Gilbert, and T. Peierls, “Sparse partial pivoting in time proportional to arithmetic operations,”, *SIAM J. Sci. Statist. Comput.* 9, pp. 862–874, 1988.