

Sparse Matrix Methods for Circuit Simulation Problems

Timothy A. Davis and E. Palamadai Natarajan

Abstract Differential algebraic equations used for circuit simulation give rise to sequences of sparse linear systems. The matrices have very peculiar characteristics as compared to sparse matrices arising in other scientific applications. The matrices are extremely sparse and remain so when factorized. They are permutable to block triangular form, which breaks the sparse LU factorization problem into many smaller subproblems. Sparse methods based on operations on dense submatrices (supernodal and multifrontal methods) are not effective because of the extreme sparsity. KLU is a software package specifically written to exploit the properties of sparse circuit matrices. It relies on a permutation to block triangular form (BTF), several methods for finding a fill-reducing ordering (variants of approximate minimum degree and nested dissection), and Gilbert/Peierls' sparse left-looking LU factorization algorithm to factorize each block. The package is written in C and includes a MATLAB interface. Performance results comparing KLU with SuperLU, Sparse 1.3, and UMFPACK on circuit simulation matrices are presented. KLU is the default sparse direct solver in the XyceTM circuit simulation package developed by Sandia National Laboratories.

1 Overview

The KLU software package is specifically designed for solving sequences of unsymmetric sparse linear systems that arise from the differential-algebraic equations used to simulate electronic circuits. Two aspects of KLU are essential for these

T.A. Davis (✉)

Department of Computer and Information Science and Engineering, University of Florida,
FL, USA

e-mail: davis@cise.ufl.edu

E. Palamadai Natarajan

Ansys, Inc., USA

e-mail: ekanathan@gmail.com

problems: (1) a permutation to block upper triangular form [15, 17], and (2) an asymptotically efficient left looking LU factorization algorithm with partial pivoting [18]. KLU does not exploit supernodes, since the factors of circuit simulation matrices are far too sparse as compared to matrices arising in other applications (such as finite-element methods).

Circuit simulation involves many different tasks for which KLU is useful:

1. DC operating point analysis, where BTF ordering is often helpful. Convergence in DC analysis is critical in that it is typically the first step of a higher level analysis such as transient analysis.
2. Transient analysis, which requires a fast and accurate sparse LU factorization. The sparse linear factorization/solve stages typically dominate the run-time of transient analyses of post-layout circuits with a large number of parasitic devices.
3. Harmonic balance analysis, which is typically solved using Krylov based iterative methods, since the Jacobian representing all the harmonics is huge and cannot be solved with a direct method. KLU is useful in factor/solve stages involving the pre-conditioner.

Section 2 describes the characteristics of circuit matrices, which motivate the design of the KLU algorithm. Section 3 gives a brief description of the algorithm. A more detailed discussion may be found in [24]. Performance results of KLU in comparison with SuperLU [12], Sparse 1.3 [21, 22], and UMFPACK [4, 6, 7] are presented in Sect. 4. An extended version of this paper appears in [11].

In this paper, $|A|$ denotes the number of nonzeros in the matrix A .

2 Characteristics of Circuit Matrices

Circuit matrices arise from Newton's method applied to the differential-algebraic equations representing the underlying circuit [23]. A modified nodal analysis is typically used, resulting in a sequence of linear systems with unsymmetric sparse coefficient matrices with identical nonzero pattern (ignoring numerical cancellation). Circuit matrices exhibit certain unique characteristics for which KLU is designed, which are not generally true of matrices from other applications:

1. Circuit matrices are extremely sparse and remain so when factorized. The ratio of floating-point operation (flop) count over $|L + U|$ is much smaller than matrices from other applications (even for comparable values of $|L + U|$). A set of columns in L with identical or similar nonzero pattern is called a *supernode* [12]. Supernodal and multifrontal methods obtain high performance by exploiting supernodes via dense matrix kernels (the BLAS, [13]). Because their nodal interconnection is highly dissimilar and their fill-in is so low, the supernodes in circuit matrices typically have very few columns. Dense matrix kernels are not effective when used on very small matrices, and thus supernodal/multifrontal methods are not suitable for circuit matrices.

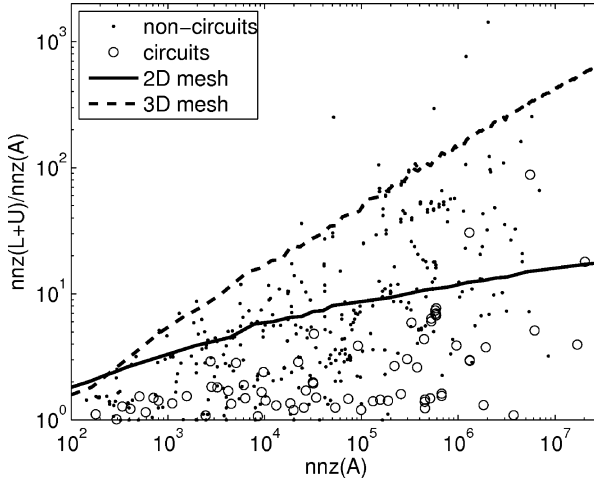


Fig. 1 Fill-in factor versus the number of nonzeros in the largest irreducible block

2. Nearly all circuit matrices are permutable to a block triangular form. In DC operating point analysis, capacitors are open and hence node connectivity is broken in the circuit. This helps in creating many small strongly connected components in the corresponding graph, and the resulting permuted matrix is block triangular with many small blocks. However in transient simulation, capacitors are not open and hence the nodes of the circuit are mostly reachable from each other. This often leads to one large diagonal block when permuted to BTF form, but still a large number of small blocks due to the presence of independent and controlled sources.

The following experiment illustrates the low fill-in properties of circuit matrices. As of March 2010, the University of Florida Sparse Matrix Collection [10] contains 491 matrices that are real, square, unsymmetric, and have full structural rank¹ (excluding matrices tagged as subsequent matrices in sequences of matrices with the same size and pattern). Of these 491 matrices, 81 are from circuit or power network simulation. Figure 1 plots the fill-in factor ($|L + U|/|A|$ versus $|A|$) for each matrix, using `lu` in MATLAB (R2010a). If the matrix is reducible to block triangular form, only the largest block is factorized for this experiment (found via `dmperm` [5]). For comparison, the two lines in Fig. 1 are 2D and 3D square meshes as ordered by METIS [20], which obtains the asymptotically optimal ordering for regular meshes.

The fill-in factor for circuit matrices stays remarkably low as compared to matrices from other applications. Very few circuit matrices experience as much fill-in as 2D or 3D meshes.

¹A matrix has full structural rank if a permutation exists so that the diagonal is zero-free.

The properties of circuit matrices demonstrated here indicate that they should be factorized via an asymptotically efficient non-supernodal sparse LU method, which motivates the KLU algorithm discussed in the next Section.

3 KLU Algorithm

KLU performs the following steps when solving the first linear system in a sequence.

1. The matrix is permuted into block triangular form (BTF). This consists of two steps: an unsymmetric permutation to ensure a zero free diagonal using maximum transversal [14, 15], followed by a symmetric permutation to block triangular form by finding the strongly connected components of the graph [16, 17, 26]. A matrix with full rank permuted to block triangular form looks as follows:

$$PAQ = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ & A_{22} & & \vdots \\ & & \ddots & \vdots \\ & & & A_{nn} \end{bmatrix}$$

2. Each block A_{kk} is ordered to reduce fill. The Approximate Minimum Degree (AMD) ordering [1, 2] on $A_{kk} + A_{kk}^T$ is used by default. The user can alternatively choose COLAMD [8, 9], an ordering provided by CHOLMOD (such as nested dissection based on METIS [20]), or any user-defined ordering algorithm that can be passed as a function pointer to KLU. Alternatively, the user can provide a permutation to order each block.
3. Each diagonal block is scaled and factorized using our implementation of Gilbert/Peierls' left looking algorithm with partial pivoting [18]. A simpler version of the same algorithm is used in the LU factorization method in the CSparse package, `cs_lu` [5] (but without the pre-scaling and without a BTF permutation). Pivoting is constrained to within each diagonal block, since the factorization method factors each block as an independent problem. No pivots can ever be selected from the off-diagonal blocks.
4. The system is solved using block back substitution.

For subsequent factorizations for matrices with the same nonzero pattern, the first two steps above are skipped. The third step is replaced with a simpler left-looking method that does not perform partial pivoting (a *refactorization*). This allows the depth-first-search used in Gilbert/Peierls' method to be skipped, since the nonzero patterns of L and U are already known.

When the BTF form is exploited, entries outside the diagonal blocks do not need to be factorized, requiring no work and causing no fill-in. Only the diagonal blocks need to be factorized.

The final system of equations to be solved after ordering and factorization with partial pivoting can be represented as

$$(PRAQ)Q^T x = PRb \quad (1)$$

where P represents the row permutation due to the BTF and fill-reducing ordering and partial pivoting, and Q represents the column permutation due to just the BTF and fill-reducing ordering. The matrix R is a diagonal row scaling matrix (discussed below). Let $(PRAQ) = LU + F$ where LU represents the factors of all the blocks collectively and F represents the entire off diagonal region. Equation (1) can now be written as

$$x = Q(LU + F)^{-1}(PRb). \quad (2)$$

The block back substitution in (2) can be better visualized as follows. Consider a simple 3-by-3 block system

$$\begin{bmatrix} L_{11}U_{11} & F_{12} & F_{13} \\ 0 & L_{22}U_{22} & F_{23} \\ 0 & 0 & L_{33}U_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}. \quad (3)$$

The equations corresponding to the above system are

$$L_{11}U_{11}x_1 + F_{12}x_2 + F_{13}x_3 = b_1 \quad (4)$$

$$L_{22}U_{22}x_2 + F_{23}x_3 = b_2 \quad (5)$$

$$L_{33}U_{33}x_3 = b_3 \quad (6)$$

In block back substitution, we first solve (6) for x_3 , and then eliminate x_3 from (5) and (4) using the off-diagonal entries. Next, we solve (5) for x_2 and eliminate x_2 from (4). Finally we solve (4) for x_1 .

The core of the Gilbert/Peierls factorization algorithm used in KLU is solving a lower triangular system $Lx = b$ with partial pivoting where L , x and b are all sparse. It consists of a symbolic step to determine the non-zero pattern of x and a numerical step to compute the values of x . This lower triangular solution is repeated n times during the entire factorization (where n is the size of the matrix) and each solution step computes a column of the L and U factors. The importance of this factorization algorithm is that the time spent in factorization is proportional to the number of floating point operations performed. The entire left looking algorithm is described in the algorithm below.

The lower triangular solve is the most expensive step and includes a symbolic and a numeric factorization step. Let $b = A(:, k)$, the k th column of A . Let G_L be the directed graph of L with n nodes. The graph G_L has an edge $j \rightarrow i$ iff $l_{ij} \neq 0$. Let $\mathcal{B} = \{i | b_i \neq 0\}$ and $\mathcal{X} = \{i | x_i \neq 0\}$ represent the set of nonzero indices in b and x respectively. Now the nonzero pattern \mathcal{X} is given by

Algorithm 1 Left-looking LU factorization

```

 $L = I$ 
for  $k = 1$  to  $n$  do
  solve  $Lx = A(:, k)$  for  $x$ 
  do partial pivoting on  $x$ 
   $U(1 : k, k) = x(1 : k)$ 
   $L(k : n, k) = x(k : n) / U(k, k)$ 
end for

```

$$\mathcal{X} = \text{Reach}_{G_L}(\mathcal{B}) \quad (7)$$

$\text{Reach}_G(i)$ denotes all nodes in a graph G reachable via paths starting at node i . $\text{Reach}(S)$ applied to a set S is the union of $\text{Reach}(i)$ for all nodes $i \in S$. Equation (7) states that the nonzero pattern \mathcal{X} is computed by the determining the vertices in G_L that are reachable from the vertices of the set \mathcal{B} .

The reachability problem is solved using a depth-first search. During the depth-first search, the Gilbert/ Peierls algorithm computes the *topological* order of \mathcal{X} . If the nodes of a directed acyclic graph are written out in topological order from left to right, then all edges in the graph would point to the right. If $Lx = b$ is solved in topological order, all numerical dependencies are satisfied. The natural order $1, 2, \dots, n$ is one such ordering (since the matrix L is lower triangular), but any topological ordering will suffice. That is, x_j must be computed before x_i if there is a path from j to i in G_L . Since the depth-first graph traversal produces \mathcal{X} in topological order as an intrinsic by-product, the solution of $Lx = b$ can be computed using the algorithm below. Sorting the nodes in \mathcal{X} to obtain the natural ordering could take more time than the number of floating-point operations, so this is skipped. The computation of \mathcal{X} and x both take time proportional to the floating-point operation count.

Algorithm 2 Solve $Lx = b$ where L , x and b are sparse

```

 $\mathcal{X} = \text{Reach}_{G_L}(\mathcal{B})$ 
 $x = b$ 
for  $j \in \mathcal{X}$  in any topological order do
   $x(j + 1 : n) = x(j + 1 : n) - L(j + 1 : n, j)x(j)$ 
end for

```

4 Performance Comparisons with Other Solvers

Five different sparse LU factorization techniques are compared:

1. KLU with default parameter settings: BTF enabled, the AMD fill-reducing ordering applied to $A + A^T$, and a strong preference for pivots selected from the diagonal.

Table 1 The thirteen test matrices with the highest run times

Matrix	Entire matrix		Largest block		Rows in 2nd largest block	Singletons $\times 10^3$
	Rows $\times 10^3$	Nonzeros $\times 10^3$	Rows $\times 10^3$	Nonzeros $\times 10^3$		
Raj1	263.7	1,300.3	263.6	1,299.6	5	0.2
ASIC_680k	682.9	2,639.0	98.8	526.3	2	583.8
rajat24	358.2	1,947.0	354.3	1,923.9	172	3.4
TSOPF_RS_b2383_c1	38.1	16,171.2	4.8	31.8	654	0.0
TSOPF_RS_b2383	38.1	16,171.2	4.8	31.8	654	0.0
rajat25	87.2	606.5	83.5	589.8	57	3.4
rajat28	87.2	606.5	83.5	589.8	57	3.4
rajat20	86.9	604.3	83.0	587.5	57	3.6
ASIC_320k	321.8	1,931.8	320.9	1,314.3	6	0.3
ASIC_320ks	321.7	1,316.1	320.9	1,314.3	6	0.1
rajat30	644.0	6,175.2	632.2	6,148.3	7	11.7
Freescale1	3,428.8	17,052.6	3,408.8	16,976.1	19	0.0

2. KLU with default parameters, except that BTF is disabled. For most matrices, using BTF is preferred, but in a few cases the BTF pre-ordering can dramatically increase the fill-in in the LU factors.
3. SuperLU 3.1 [12], using non-default diagonal pivoting preference and ordering options identical to KLU (but without BTF).² These options typically give the best results for circuit matrices. SuperLU is a supernodal variant of the Gilbert/Peierls' left-looking algorithm used in KLU.
4. UMFPACK [4, 6, 7] with default parameters. In this mode, UMFPACK evaluates the symmetry of the nonzero pattern and selects either the AMD ordering on $A + A^T$ and a strong diagonal preference, or it uses the COLAMD ordering with no preference for the diagonal. For most circuit simulation matrices, the AMD ordering is used. UMFPACK is a right-looking multifrontal algorithm that makes extensive use of BLAS kernels.
5. Sparse 1.3 [21, 22], the sparse solver used in SPICE3f5, the latest version of SPICE.³

The University of Florida Sparse Matrix Collection [10] includes 81 real square unsymmetric matrices or matrix sequences (only the first matrix in each sequence is considered here) arising from the differential algebraic equations used in SPICE-like circuit simulation problems, or from power network simulation. All five methods were tested on all 81 matrices, except for two matrices too large for any method on the computer used for these tests (a single-core 3.2 GHz Pentium 4 with 4 GB of RAM). The thirteen matrices requiring the most amount of time to analyze, factorize, and solve (as determined by the fastest method for each matrix) are shown in Table 1. All of the matrices come from a transient analysis, since the run time

² Threshold partial pivoting tolerance of 0.001 to give preference to the diagonal, the SuperLU symmetric mode, and the AMD ordering on $A + A^T$.

³<http://bwrc.eecs.berkeley.edu/Classes/icbook/SPICE/>

for KLU is very low for matrices arising from a DC analysis. The table lists the matrix name followed by the size of the whole matrix and the largest block in the BTF form (the dimension and the number of nonzeros). The last two columns list the dimension of the second-largest block, and the number of 1-by-1 blocks, respectively.

A performance profile compares the relative run times of multiple methods on a set of test problems. Let the relative run time of a method on a particular problem be equal to its run time for that problem divided by the fastest run time of any method for that problem. A relative run time of 1.0 means that the method is the fastest for that problem among the methods being compared; 2.0 means that it took twice the time as the fastest method. The x axis of a performance profile is this relative run time. The y axis of a performance profile is the number of problems. A point (x, y) is plotted if a method has a relative run time of x (or less) for y problems in the test set.

The performance profiles of the four methods are shown in Fig. 2. It excludes the symbolic ordering and analysis, since this step is done just once for a whole sequence of matrices. Note that the x axis of Fig. 2 is a log scale. For most matrices, KLU (with BTF) is the fastest method. In the worst case (the Raj1 matrix) it is 26 times slower than SuperLU, but this is because the permutation to BTF used by KLU causes fill-in to dramatically increase.

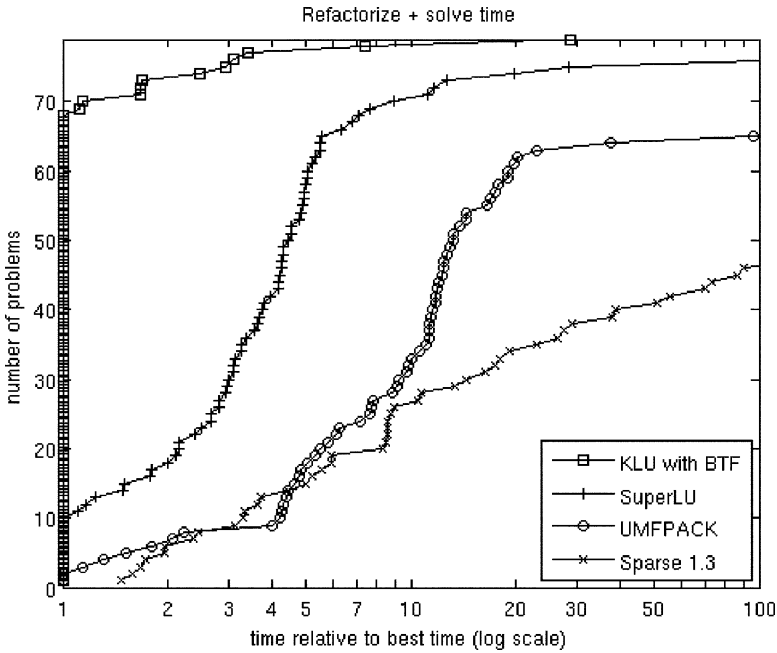


Fig. 2 Performance profile of refactorize+solve time

Table 2 Analyze+factorize+solve time in seconds, and relative fill-in ($|L + U|/|A|$) for KLU. Run times within 25% of the fastest are shown in bold. A dash is shown if the method ran out of memory

Matrix	KLU+BTF		KLU no BTF		SuperLU	Sparse 1.3
	Fill	Time	Fill	Time	Time	Time
Raj1	40.3	111.0	5.5	4.6	4.2	3,038.9
ASIC_680ks	2.6	5.0	2.7	7.2	4.6	818.1
ASIC_680k	2.1	5.8	2.1	7.4	5.8	8,835.1
rajat24	28.7	119.0	3.3	6.0	13.9	–
TSOPF_RS_b2383_c1	1.3	6.5	2.1	71.8	34.9	–
TSOPF_RS_b2383	1.3	6.5	2.1	72.0	34.2	–
rajat25	6.7	8.5	35.2	31.7	37.2	2,675.4
rajat28	6.9	9.1	28.4	25.4	50.0	3,503.0
rajat20	7.0	9.1	35.2	31.3	40.5	4,314.1
ASIC_320k	2.5	30.4	42.9	447.5	18.1	7,908.2
ASIC_320ks	3.2	36.6	3.2	36.4	21.5	684.9
rajat30	5.1	73.0	3.2	23.8	22.5	–
Freescape1	3.9	86.8	3.9	85.6	–	–

Table 3 Refactorize+solve time in seconds

Matrix	KLU+BTF	KLU no BTF	SuperLU	Sparse 1.3
	Time	Time	Time	Time
Raj1	94.4	3.0	3.3	127.4
ASIC_680ks	3.9	5.4	3.5	256.7
ASIC_680k	4.6	5.1	4.6	835.8
rajat24	91.2	3.7	12.4	–
TSOPF_RS_b2383_c1	5.2	40.8	10.9	–
TSOPF_RS_b2383	5.1	41.0	10.9	–
rajat25	6.7	27.0	36.8	374.4
rajat28	7.3	21.8	49.6	512.7
rajat20	7.3	26.8	40.2	657.1
ASIC_320k	28.7	429.0	17.1	870.1
ASIC_320ks	35.0	35.0	20.7	182.0
rajat30	60.5	18.6	19.6	–
Freescape1	70.5	70.6	–	–

The time for the thirteen largest matrices is shown in Tables 2 and 3. The fastest run times and run times within 25% of the fastest are shown in bold. A dash is shown if the method ran out of memory.

For sparse Cholesky factorization, the flops per $|L|$ ratio is an accurate predictor of the relative performance of a BLAS-based supernodal method versus a non-supernodal method. If this ratio is 40 or higher, chol in MATLAB (and $x=A\b b$ for sparse symmetric positive definite matrices) automatically selects a supernodal

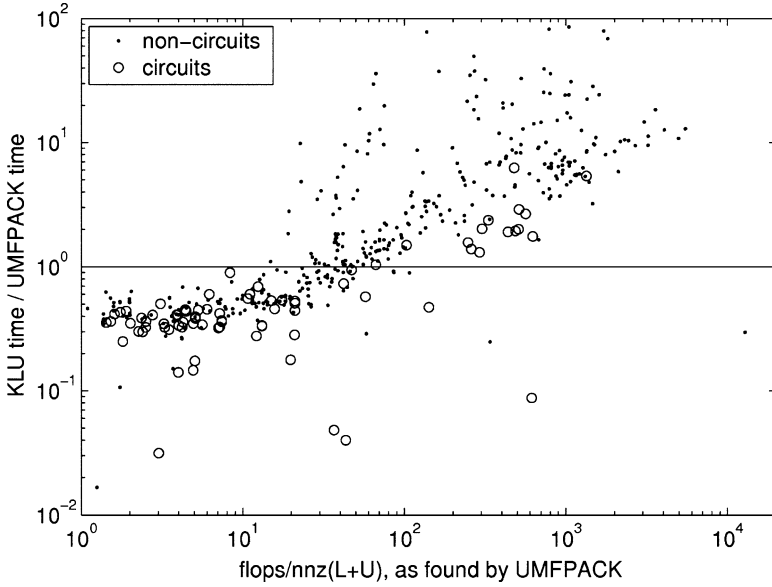
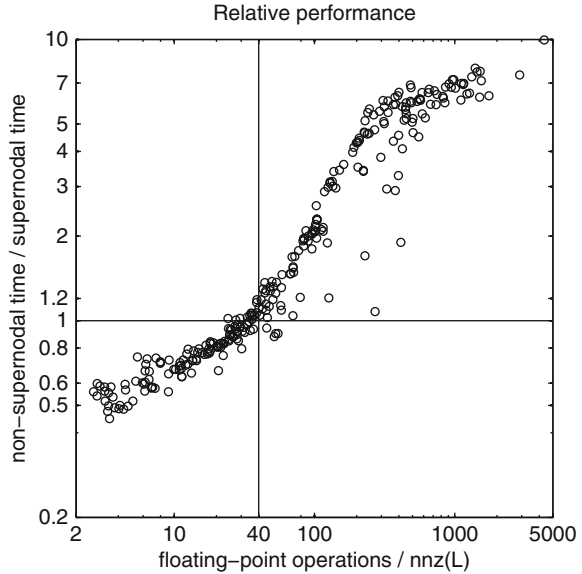


Fig. 3 Relative performance of KLU versus UMFPACK as a function of $\text{flops}/|L + U|$

Fig. 4 Relative supernodal (BLAS-based) and non-supernodal (not BLAS-based) performance for sparse Cholesky



solver. Otherwise, a non-supernodal solver is used [3]. A similar comparison is shown in Fig. 3 between KLU and UMFPACK. If the matrix is reducible, only the largest block is factorized. Figure 4 shows the results for sparse Cholesky factorization from [3].

These results are remarkable for three reasons:

1. Circuit matrices tend to have a low $\text{flop}/|L + U|$ ratio as compared to other matrices.
2. Even when the $\text{flop}/|L + U|$ ratio is high enough (200 or more) to justify using the BLAS, the relative performance of a BLAS-based method (UMFPACK) versus KLU is much less than what would be expected if only non-circuit matrices were considered. Thus, circuits not only remain sparse when factorized, even large circuit matrices with higher $\text{flops}/|L + U|$ ratios hardly justify the use of the BLAS.
3. The $\text{flops}/|L + U|$ ratio for LU factorization (Fig. 3) is not a very accurate predictor of the relative performance of BLAS-based sparse methods as compared to non-BLAS-based methods, as it is for sparse Cholesky factorization (Fig. 4).

5 Summary

KLU has been shown to be an effective solver for the sequences of sparse matrices that arise when solving differential algebraic equations for circuit simulation problems. It is the default sparse solver in Xyce, a circuit simulation package developed by Sandia National Laboratories [19], for which it has been proven to be a robust and reliable solver [25].

Acknowledgements We would like to thank Mike Heroux for coining the name “KLU” and suggesting that we tackle this project in support of the Xyce circuit simulation package developed at Sandia National Laboratories [19, 25]. Portions of this work were supported by the Department of Energy, and by National Science Foundation grants 0203270, 0620286, and 0619080.

References

1. Amestoy, P.R., Davis, T.A., Duff, I.S.: An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.* **17**(4), 886–905 (1996)
2. Amestoy, P.R., Davis, T.A., Duff, I.S.: Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.* **30**(3), 381–388 (2004)
3. Chen, Y., Davis, T.A., Hager, W.W., Rajamanickam, S.: Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Trans. Math. Softw.* **35**(3), 1–14 (2008). DOI <http://doi.acm.org/10.1145/1391989.1391995>
4. Davis, T.A.: Algorithm 832: UMFPACK V4.3, an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.* **30**(2), 196–199 (2002)
5. Davis, T.A.: *Direct Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA (2006)
6. Davis, T.A., Duff, I.S.: An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM J. Matrix Anal. Appl.* **18**(1), 140–158 (1997)
7. Davis, T.A., Duff, I.S.: A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Trans. Math. Softw.* **25**(1), 1–19 (1999)

8. Davis, T.A., Gilbert, J.R., Larimore, S.I., Ng, E.G.: Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.* **30**(3), 377–380 (2004)
9. Davis, T.A., Gilbert, J.R., Larimore, S.I., Ng, E.G.: A column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.* **30**(3), 353–376 (2004)
10. Davis, T.A., Hu, Y.: University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, **38**(1), (2011). URL <http://www.cise.ufl.edu/sparse/matrices>
11. Davis, T.A., Palamadai Natarajan, E.: Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Trans. Math. Softw.* **37**(3), 36:1–36:17 (2010). DOI <http://doi.acm.org/10.1145/1824801.1824814>
12. Demmel, J.W., Eisenstat, S.C., Gilbert, J.R., Li, X.S., Liu, J.W.H.: A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Anal. Appl.* **20**(3), 720–755 (1999)
13. Dongarra, J.J., Du Croz, J., Duff, I.S., Hammarling, S.: A set of level-3 basic linear algebra subprograms. *ACM Trans. Math. Softw.* **16**(1), 1–17 (1990)
14. Duff, I.S.: Algorithm 575: Permutations for a zero-free diagonal. *ACM Trans. Math. Softw.* **7**(1), 387–390 (1981)
15. Duff, I.S.: On algorithms for obtaining a maximum transversal. *ACM Trans. Math. Softw.* **7**(1), 315–330 (1981)
16. Duff, I.S., Reid, J.K.: Algorithm 529: Permutations to block triangular form. *ACM Trans. Math. Softw.* **4**(2), 189–192 (1978)
17. Duff, I.S., Reid, J.K.: An implementation of Tarjan’s algorithm for the block triangularization of a matrix. *ACM Trans. Math. Softw.* **4**(2), 137–147 (1978)
18. Gilbert, J.R., Peierls, T.: Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Statist. Comput.* **9**, 862–874 (1988)
19. Hutchinson, S.A., Keiter, E.R., Hoekstra, R.J., Waters, L.J., Russo, T., Rankin, E., Wix, S.D., Bogdan, C.: The XyceTM parallel electronic simulator – an overview. In: Joubert, G.R., Murli, A., Peters, F.J., Vanneschi, M. (eds.) *Parallel Computing: Advances and Current Issues*, Proc. ParCo 2001, pp. 165–172. Imperial College Press, London (2002)
20. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.* **20**, 359–392 (1998)
21. Kundert, K.S.: Sparse matrix techniques and their applications to circuit simulation. In: Ruehli, A.E. (ed.) *Circuit Analysis, Simulation and Design*. North-Holland, New York (1986)
22. Kundert, K.S., Sangiovanni-Vincentelli, A.: User’s guide: Sparse 1.3. Tech. rep., Dept. of EE and CS, UC Berkeley (1988)
23. Nichols, K., Kazmierski, T., Zwolinski, M., Brown, A.: Overview of SPICE-like circuit simulation algorithms. *IEE Proc. Circuits, Devices & Sys.* **141**(4), 242–250 (1994)
24. Palamadai Natarajan, E.: KLU - a high performance sparse linear system solver for circuit simulation problems. M.S. Thesis, CISE Department, Univ. of Florida (2005)
25. Sipics, M.: Sparse matrix algorithm drives SPICE performance gains. *SIAM News* **40**(4) (2007)
26. Tarjan, R.E.: Depth first search and linear graph algorithms. *SIAM J. Comput.* **1**, 146–160 (1972)

Scientific Computing in Electrical Engineering SCEE
2010

Michielsen, B.; Poirier, J.-R. (Eds.)

2012, XVI, 460 p., Hardcover

ISBN: 978-3-642-22452-2