Sparsity-Oriented Sparse Solver Design for Circuit Simulation

Xiaoming Chen, Lixue Xia, Yu Wang, Huazhong Yang
Department of Electronic Engineering,
Tsinghua National Laboratory for Information Science and Technology,
Tsinghua University, Beijing 100084, China
Email: chenxm_1986@126.com, yu-wang@tsinghua.edu.cn, yanghz@tsinghua.edu.cn

***Abstract*—The sparse solver is a critical component in circuit simulators. The widely used solver KLU is based on a pure column-level algorithm. In this paper, we point out that KLU is not always the best algorithm for circuit matrices by experiments. We also demonstrate that the optimal algorithm strongly depends on the sparsity of the matrix. Two sparse LU factorization algorithms are proposed for extremely sparse matrices and dense matrices. A simple but effective strategy is proposed to select the optimal algorithm according to the sparsity. By combining the two new algorithms and the selection method together, the proposed solver achieves much higher performance than both KLU and PARDISO.**

I. Introduction

The rapid development of modern integrated circuits has brought great challenges to Simulation Program with Integrated Circuit Emphasis (SPICE) [1]-based circuit simulators, due to the increase in both the number of transistors and the dimension of the matrix. The sparse solver is a critical component in SPICE-accurate circuit simulators. A post-layout transient simulation of a typical analog/mixed-signal circuit can take several days to weeks, and the sparse solver may consume more than half of the total simulation time [2].

As a direct method, sparse LU factorization is widely used in circuit simulators because of its high robust and reliability. Typically, there are three steps in a sparse LU factorization: pre-processing, numerical factorization ($\mathbf{A} = \mathbf{LU}$), and substitution ($\mathbf{Ly} = \mathbf{b}$ and $\mathbf{Ux} = \mathbf{y}$). In circuit simulation, pre-processing is executed only once to re-order the matrix and obtain the symbolic patterns of the LU factors. Factorization and substitution are iterated in Newton-Raphson and/or transient simulation iterations. A substitution is usually much faster than a factorization, so LU factorization is the bottleneck in the sparse solver.

From the computational granularity point of view, there are two categories in sparse LU factorization algorithms. The popular approach is to gather nonzeros into dense sub-matrices and then use the basic linear algebra subproblems (BLAS) [3] to deal with the dense sub-matrices. The computational granularity is a dense sub-matrix in this category. There are dozens of supernode- or multifrontal-based packages which belong to this category [4]. Popular packages include PARDISO [5], SuperLU [6], [7], UMFPACK [8], MUMPS [9], etc. These packages are not specially targeted at circuit matrices. The other approach does not utilize any dense sub-matrices. Only a few packages belong to this category, including KLU [10], SPARSE [11], and NICSLU [12], [13]. These three packages are all targeted at circuit matrices. The computational granularity is a nonzero element in this category. The reason why circuit simulators do not utilize dense sub-matrices is that circuit matrices are considered too sparse to form big dense sub-matrices. SPARSE is an old package which performs poorly for big matrices [10], [14], because of the low cache efficiency of its linked list on modern processors. KLU adopts the sparse LU factorization algorithm proposed by Gilbert and Peierls (G/P algorithm) [15] as its kernel. KLU is faster than other BLAS-based sparse solvers for most circuit matrices [10]. KLU is also faster than SPARSE except for very small matrices [14]. NICSLU is a parallel sparse solver which is based on KLU so it also uses the G/P algorithm.

This work was supported by 973 project 2013CB329000, National Natural Science Foundation of China (No. 61373026, 61261160501), and Tsinghua University Initiative Scientific Research Program.

KLU or the G/P algorithm is widely used for circuit simulation problems. However, actually whether the G/P algorithm is really the best algorithm for circuit matrices is unclear. Till now, very few works have been published to comprehensively analyze the performance of different computational granularities for circuit matrices. In this paper, we will point out that the pure G/P algorithm is not always the best for circuit matrices by experiments. We also investigate that, carefully designing different algorithms and selecting the optimal algorithm according to the sparsity of the matrix can achieve better performance. We make the following main contributions in this paper.

- We point out that the pure G/P algorithm adopted by KLU is not always the best for circuit matrices. The best solver strongly depends on the sparsity.
- We develop two different sparse LU factorization algorithms for circuit matrices. One is for extremely sparse matrices and the other is for dense circuit matrices. The two algorithms can be both parallelized.
- We propose to use a simple metric which measures the sparsity to select the optimal algorithm.

The rest of this paper is organized as follows. In Section II, we introduce the basis of the G/P algorithm and present our motivation. Section III presents the two proposed algorithms in detail. Experimental results are presented in Section IV. Finally, Section V concludes the paper.

Algorithm 1 G/P Algorithm (no pivoting) [15].

```
1: $\mathbf{L} = \mathbf{I}$; //$\mathbf{I}$ is the identity matrix
2: for $k = 1 : N$ do
3:     $\mathbf{x} = \mathbf{A}(:, k)$; //$\mathbf{x}$ is an uncompressed vector of length $N$
4:     for $j = 1 : k - 1$ where $\mathbf{U}(j, k) \neq 0$ do
5:         $\mathbf{x}(j + 1 : N) -= \mathbf{x}(j) \cdot \mathbf{L}(j + 1 : N, j)$;
6:     end for
7:     $\mathbf{U}(1 : k, k) = \mathbf{x}(1 : k)$;
8:     $\mathbf{L}(k : N, k) = \frac{\mathbf{x}(k : N)}{\mathbf{x}(k)}$;
9: end for
```

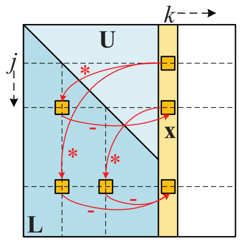


Fig. 1: Illustration of the G/P algorithm.

II. Background and Motivation

In this section, we first introduce some basis of the G/P algorithm, and then present our motivation of this work.

A. G/P Algorithm

Algorithm 1 lists the overall flow of the G/P algorithm without pivoting. Basically, it is a column-level algorithm, i.e., the matrix is factorized column by column (the `for` loop on line 2). Each column is numerically updated by some dependent columns on the left side (lines 4 to 6). Column k depends on column j if and only if $\mathbf{U}(j, k)$ is a nonzero (line 4) [12]. Fig. 1 illustrates the core numerical operation in the G/P algorithm. An assumption behind this algorithm is that the symbolic patterns of $\mathbf{L}$ and $\mathbf{U}$ are known in advance. The symbolic patterns can be obtained by a full G/P algorithm with partial pivoting proposed in [15]. In circuit simulation, the first factorization is performed with pivoting but subsequent factorizations can be performed without pivoting, because the symbolic pattern of the matrix created by modified nodal analysis is fixed during Newton-Raphson iterations.

Sparse matrices are stored in a compressed form, i.e., only the values and positions of nonzeros are stored. This leads to a problem when we want to visit a nonzero of the sparse matrix from the compressed storage, because we do not know its address in the compressed form in advance. The key idea to solve this problem in the G/P algorithm is to use an uncompressed vector $\mathbf{x}$, which temporarily holds the values of the column that is being updated. Except $\mathbf{x}$, matrices $\mathbf{A}$, $\mathbf{L}$ and $\mathbf{U}$ are all stored in compressed arrays. For each column, we need to transfer values between compressed matrices and the uncompressed vector $\mathbf{x}$ (lines 3, 5, 7 and 8). Fig. 2 shows an example of the transfer. For example, if we want to store the values of the uncompressed vector $\mathbf{x}$ back to the compressed array, we need to traverse the compressed array. For each nonzero, we first get the position from the compressed array, read the value from the corresponding position in $\mathbf{x}$, and then write it back to the compressed array.

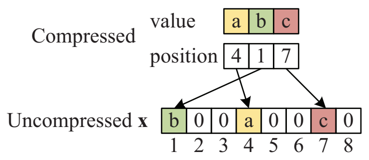


Fig. 2: Transferring data between compressed arrays and the uncompressed vector.

B. Motivation

Although the indexing problem is successfully solved by using an uncompressed vector $\mathbf{x}$ in the G/P algorithm, it can be easily understood that this method has the following two performance problems.

- When the matrix is extremely sparse, each column may have only a few nonzeros. In this case, visiting nonzeros in $\mathbf{x}$ will lead to a high cache miss rate because of the large address difference between nonzeros, especially when the matrix is large.
- Data transfers between compressed arrays and $\mathbf{x}$ are based on indirect memory accesses, which are slower than visiting an array of the same length continuously and directly. When the matrix is dense, there will be many nonzeros in each column. We may gather nonzeros in different columns into dense sub-matrices such that indirect memory access is avoided. Cache efficiency can also be improved by this method.

This paper will aim to solve these two issues by two proposed algorithms, to improve the performance of sparse solvers for circuit matrices. The goal of the two algorithms is to improve the cache efficiency and reduce indirect memory accesses.

III. Proposed Algorithms

In this section, we will propose two algorithms to improve the above two issues respectively. For the first issue, we will design a *map algorithm* to avoid the use of the uncompressed vector $\mathbf{x}$. More specifically, all the addresses corresponding to the positions that will be numerically updated during sparse LU factorization are recorded in advance, such that $\mathbf{x}$ is never required. For the second issue, we will borrow the concept of *supernode* used in SuperLU [6], [7] to enhance the cache efficiency when computing dense sub-matrices. Although circuit matrices cannot be too dense, by carefully designing a lightweight *supernode-column algorithm*, we can still achieve higher performance than the pure G/P algorithm for circuit matrices, as will be shown in Section IV. The G/P algorithm is also called a *column algorithm*.

A. Map Algorithm

Basically, the goal of the map algorithm is to avoid of using the uncompressed vector $\mathbf{x}$ in the column algorithm (i.e., the G/P algorithm). The necessity of using $\mathbf{x}$ in the G/P algorithm

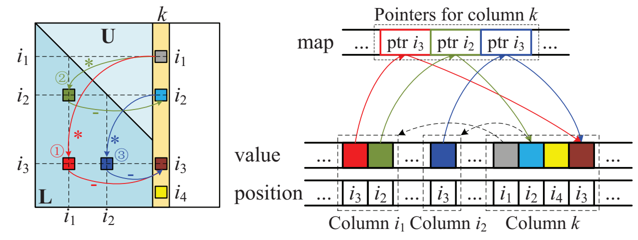
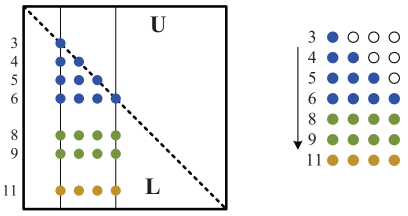


Fig. 3: Illustration of the map algorithm.



(a) Definition of supernode. (b) Storage of a supernode.

Fig. 4: Example of a supernode [6], [7].

is to solve the indexing problem for compressed arrays. For example, assume that we are using column j to update column k. We traverse the compressed array of $\mathbf{L}(:,j)$, and for each nonzero in $\mathbf{L}(:,j)$, say $\mathbf{L}(i,j)$, we need to find the address of $\mathbf{x}(i)$ to perform the numerical update. If $\mathbf{x}$ is compressed, finding the address of $\mathbf{x}(i)$ requires a traversal on $\mathbf{x}$. On the contrary, if $\mathbf{x}$ is uncompressed, the desired address is simply $\mathbf{x}[i]$. As mentioned in the previous section, the use of $\mathbf{x}$ leads to a high cache miss rate when $\mathbf{x}$ is too sparse. Circuit matrices can be very sparse. It can be obtained from [10] that the fill-in ratio $\frac{NNZ(\mathbf{L}+\mathbf{U}-\mathbf{I})}{NNZ(\mathbf{A})}$ of many circuit matrices is less than 5, but for non-circuit matrices, the fill-in ratio can be up to 1000. This means that there are very few nonzeros in each row/column of the LU factors of many circuit matrices.

The map algorithm does not use the uncompressed vector $\mathbf{x}$. Instead, the compressed storages of $\mathbf{L}$ and $\mathbf{U}$ are directly used in the numerical factorization. To solve the indexing problem, we use a pointer array which is called a map to record all the addresses corresponding to the positions that will be numerically updated during sparse LU factorization. The map records all such addresses in sequence. Consequently, when updating a column using dependent columns, we only need to directly update the values which are pointed by the corresponding pointers, instead of searching the update positions from compressed arrays. After each operation, the pointer is increased by one to point to the next update position.

Fig. 3 shows an example of the map algorithm. We are now updating column k, and column k depends on columns i_1 and i_2 because $\mathbf{U}(i_1,k)$ and $\mathbf{U}(i_2,k)$ are nonzeros. We first use column i_1 to update column k. Column i_1 has two nonzeros at rows i_3 and i_2 (note that nonzeros in $\mathbf{L}$ are not required to store in order). The first operation is $\mathbf{L}(i_3,k) -= \mathbf{U}(i_1,k)\cdot\mathbf{L}(i_3,i_1)$ (the red lines in Fig. 3). The address of $\mathbf{L}(i_3,k)$ is the first pointer of the pointers for column k in the map. The second operation is $\mathbf{U}(i_2,k) -= \mathbf{U}(i_1,k)\cdot\mathbf{L}(i_2,i_1)$ (the blackish green lines). The address of $\mathbf{U}(i_2,k)$ is the second pointer of the pointers for column k in the map. The third operation is to use the sole nonzero in column i_2 to update column k (the blue lines), and it can be done by a similar way.

Creating the map is trivial. We just need to go through the factorization process and record all the positions which are updated during sparse LU factorization in sequence. In circuit simulation, the map is created after the first factorization with pivoting. It is a one-time work so its time overhead can be ignored. Actually the time overhead of creating the map is generally less than the time of one factorization.

Algorithm 2 Supernode-column algorithm.

```
1: $\mathbf{L}=\mathbf{I}$; //$\mathbf{I}$ is the identity matrix
2: for $k=1:N$ do
3:     $\mathbf{x}=\mathbf{A}(:,k)$; //$\mathbf{x}$ is an uncompressed vector of length $N$
4:     for $j=1:k-1$ where $\mathbf{U}(j,k)\neq 0$ do
5:         if column $j$ has not been used for update column $k$ then
6:             if column $j$ belongs to a supernode that ends at column $s$ then
               //perform supernode-column update
7:                 $\mathbf{x}(j:s)=\mathbf{L}(j:s,j:s)^{-1}\cdot\mathbf{x}(j:s)$;
8:                 $\mathbf{x}(s+1:N) -= \mathbf{L}(s+1:N,j:s)\cdot\mathbf{x}(j:s)$;
9:             else //perform column-column update
10:                $\mathbf{x}(j+1:N) -= \mathbf{x}(j)\cdot\mathbf{L}(j+1:N,j)$;
11:            end if
12:        end if
13:    end for
14:    $\mathbf{U}(1:k,k)=\mathbf{x}(1:k)$;
15:    $\mathbf{L}(k:N,k)=\dfrac{\mathbf{x}(k:N)}{\mathbf{x}(k)}$;
16: end for
```

The map algorithm brings us two advantages for extremely sparse matrices. First, the cache efficiency is improved because the uncompressed vector $\mathbf{x}$ is avoided. Second, indirect memory accesses are also reduced, because indirect accesses to $\mathbf{x}$ are avoided.

B. Supernode-Column Algorithm

Although circuit matrices are typically very sparse, they can also be dense for some special circuits. For example, post-layout circuits will contain large power and ground meshes so matrices created by modified nodal analysis can be dense due to the mesh nature. To efficiently solve such matrices, we developed a supernode-column algorithm, where the concept of supernode is borrowed from SuperLU [6], [7]. A supernode is a set of successive columns of $\mathbf{L}$ with triangular diagonal block full and the same structure below the diagonal block, as shown in Fig. 4a. A supernode can be stored by a column-wise dense matrix (the triangular diagonal part of $\mathbf{U}$ is not stored in the supernode so these positions are left blank), as shown in Fig. 4b. We also need an array to store the row indexes of the supernode.

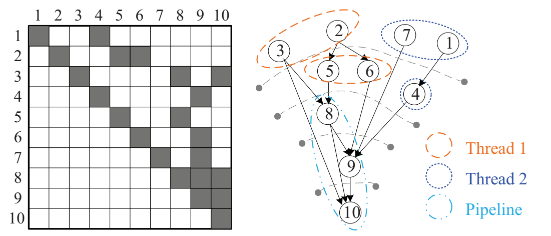
By applying supernodes in the column algorithm, we can use a supernode instead of a column to update another column at a time. Algorithm 2 shows the proposed supernode-column algorithm. When updating column k using column j, if column j belongs to a supernode which ends at column s, we will perform a supernode-column update. In other words, we use the supernode from column j to column s to update column k (lines 7 and 8). The supernode-column update does not introduce redundant computations, because according to the symbolic factorization theory [10], when $\mathbf{L}(j:s, j:s)$ is full and $\mathbf{U}(j,k)$ is a nonzero, $\mathbf{U}(j+1:s,k)$ are also nonzeros so column k also depends on columns from $j+1$ to s. The two operations of lines 7 and 8 can be effectively implemented by two BLAS routines `dtrsv` and `dgemv`, respectively, if supernodes are stored by the method shown in Fig. 4b. If column j does not belong to any supernode, we perform a column-column update (line 10) just like the G/P algorithm.

Please note that the proposed supernode-column algorithm is different from SuperLU or PARDISO, although they also utilize supernodes to enhance the cache performance for dense sub-matrices. SuperLU and PARDISO both use a supernode-supernode algorithm, where each supernode is updated by dependent supernodes. The reason why they use such a method is that, when multiple columns depend on a same supernode, the supernode will be read for multiple times to update these columns separately. Consequently, gathering these columns into a destination supernode and updating them together will make the source supernode be read only once. However, considering the fact that modern processors always have a large cache and supernodes in circuit matrices cannot be too large, many supernodes can reside in the cache simultaneously. Reading a supernode more than once cannot affect the performance significantly. In addition, the supernode-supernode algorithm will introduce some additional computations. Consequently, we develop a supernode-column algorithm which is more lightweight than the supernode-supernode algorithm adopted by SuperLU and PARDISO.

The supernode-column algorithm brings us three advantages for dense circuit matrices. First, indirect accesses in supernodes are avoided. Second, we can utilize vendor-optimized BLAS library to deal with dense sub-matrices such that the performance can be significantly improved. Finally, cache efficiency can also be improved because supernodes are stored by continuous arrays.

C. Parallelization

Both the map algorithm and the supernode-column algorithm can be parallelized using the same method as the parallel column algorithm proposed in [12]. Since we directly adopt



(a) Symbolic pattern of $\mathbf{U}$. (b) Dependence graph and the scheduling method.

Fig. 5: Dependence graph-guided parallel scheduling method [12].

the existing parallelization strategy, we only provide a simple framework here. In Section II-A, we have mentioned that column k depends on column j if and only if $\mathbf{U}(j,k)$ is a nonzero. Based on this conclusion, a column-level dependence graph which is a directed acyclic graph (DAG) can be extracted from the symbolic pattern of $\mathbf{U}$. The DAG is then levelized into levels by an as-soon-as-possible scheduling method such that nodes in the same level are completely independent. For levels which have many nodes, the nodes in each level are equally assigned to threads so they can be computed in parallel. Such levels are processed level by level. For levels with very few nodes, a pipeline scheduling method is proposed in [12] to explore parallelism between dependent levels. Fig. 5 shows an example of the scheduling method. This parallelization strategy can be directly applied to the two proposed algorithms. The only difference is that in the parallel supernode-column algorithm, we need to wait for dependent supernodes to finish, instead of dependent columns.

IV. Experimental Results

In this section, we first investigate how to select the optimal algorithm, and then evaluate the performance of the proposed solver by two ways: a benchmark test and a real simulation test based on NGSPICE [16].

A. Optimal Algorithm Selection

In this test and the benchmark test shown in the next sub-section, we evaluate the two proposed algorithms for 44 benchmarks obtained from the University of Florida sparse matrix collection [17]. All these benchmarks are nonsymmetric circuit matrices from DC, transient or frequency-domain simulations. The two tests are both implemented on a Linux server equipped with two Intel Xeon E5-2690 CPUs and 64GB memory. The proposed supernode-column algorithm uses the BLAS library provided by the Intel Math Kernel Library (MKL) to compute supernodes. All codes are compiled by the Intel C++ compiler (icc) with `-O3` optimization.

The two proposed algorithms are both compared with the G/P algorithm (i.e., the column algorithm) to investigate how to select the optimal algorithm. The G/P algorithm is implemented by ourselves instead of using KLU, because the data structure defined in KLU is not suitable for implementing our proposed algorithms. To make a fair comparison, we implement them using the same data structure and the same pre-processing method, such that the only difference is the algorithm itself.

Fig. 6 shows the comparison for the factorization time. We have found that the number of floating point operations per nonzero (i.e., $\overline{flops} = \frac{flops}{NNZ(\mathbf{L}+\mathbf{U}-\mathbf{I})}$) is a good metric to measure the sparsity of matrices. All benchmarks shown in Fig. 6 are sorted by this metric. The map algorithm is generally faster than the column algorithm for extremely sparse matrices which are on the left side of dc1. Since $\overline{flops}$ of dc1 is 31, we propose to use the map algorithm when $\overline{flops} < 30$. The parallel map algorithm has higher speedups than the sequential map algorithm, compared with the corresponding column

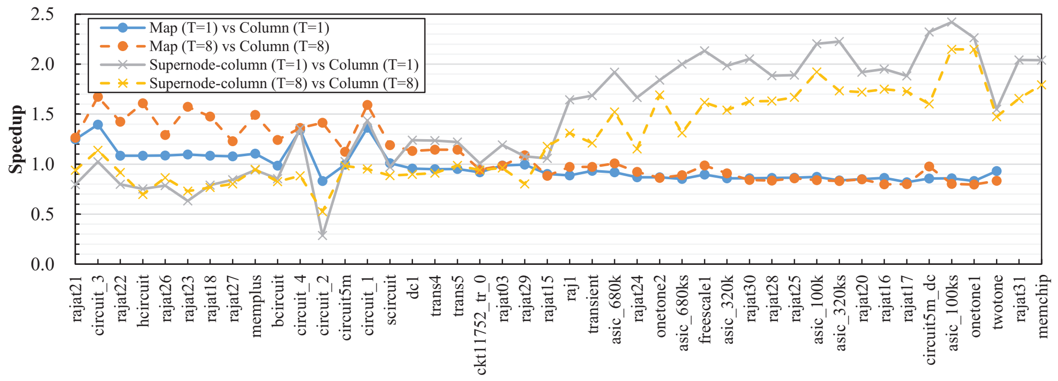


Fig. 6: Speedups of the map algorithm and the supernode-column algorithm compared with the column algorithm (i.e., the G/P algorithm), for the factorization time. The map algorithm fails on the last two matrices due to insufficient memory.

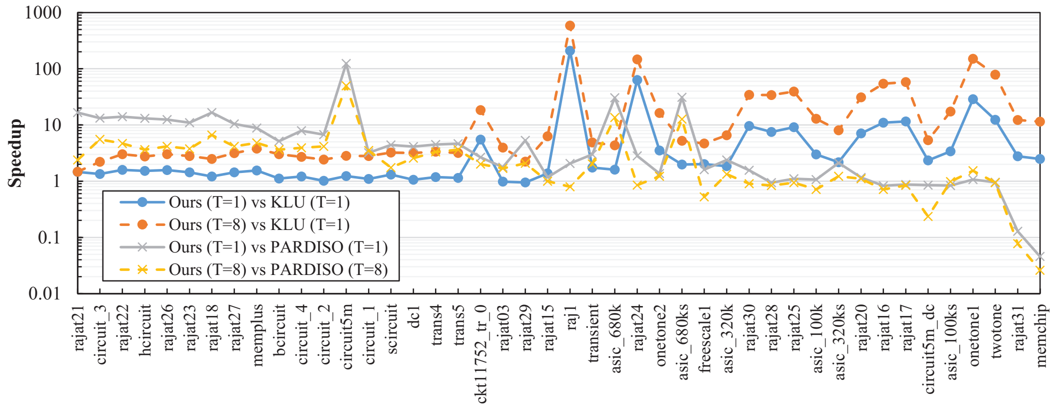


Fig. 7: Speedups of the proposed solver compared with KLU and PARDISO, for the total runtime of factorization and substitution.

algorithm. This is because that the cache miss rate of the parallel column algorithm is higher than the sequential column algorithm, as multiple uncompressed vectors $\mathbf{x}$ share the same cache in the parallel column algorithm. The supernode-column algorithm is faster than the column algorithm for matrices which are on the right side of rajat15. Since $\overline{flops}$ of rajat15 is 80, we propose to use the supernode-column algorithm when $\overline{flops} > 80$. By applying this selection strategy, our solver achieves about 1.5× average speedup compared with the G/P algorithm, for all the 44 benchmarks. The metric $\overline{flops}$ can be easily predicted in the pre-processing step by a fill-in reduction algorithm [18].

B. Benchmark Results

The proposed solver with optimal algorithm selection is compared with KLU and PARDISO. PARDISO is from MKL and also uses the BLAS library provided by MKL. Since factorization and substitution are both iterated in circuit simulations, here we compare the total runtime of factorization and substitution.

Fig. 7 shows the speedups. Our solver is faster or at least not slower than KLU for all the 44 benchmarks. As shown in Table I, the average (geometric mean) speedups over KLU are 2.8× and 8.36×, when our solver uses 1 thread and 8 threads respectively. The large difference in the speedups is mainly

TABLE I: Average speedups of the proposed solver compared with KLU and PARDISO.

	Geometric mean
Ours (T=1) vs KLU (T=1)	2.80
Ours (T=8) vs KLU (T=1)	8.36
Ours (T=1) vs PARDISO (T=1)	3.17
Ours (T=8) vs PARDISO (T=8)	1.78

caused by different pre-processing methods between KLU and our solver. Our solver is faster than PARDISO for most of the benchmarks and slower for only a few dense matrices. The average (geometric mean) speedups over PARDISO are 3.17× and 1.78×, when our solver and PARDISO both use 1 thread and 8 threads respectively. Fig. 7 indicates that for a few dense circuit matrices like rajat31 and memchip, the supernode-column algorithm is not the best but the supernode-supernode algorithm adopted by PARDISO is the best. However, it is rare to see such dense matrices in circuit simulation. Our solver achieves the highest performance for most of the benchmarks.

The speedups shown in Fig. 7 clearly indicate that KLU is not always the best algorithm for circuit simulation. By carefully designing algorithms for extremely sparse matrices and dense circuit matrices, the proposed solver achieves higher average performance than both KLU and PARDISO. Although

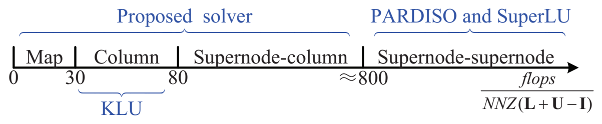


Fig. 8: Best algorithm for matrices of different sparsity.

TABLE II: Transient simulation time (in seconds) of NGSPICE when using KLU and the proposed solver, respectively.

Benchmark	Dimen.	Transis.	KLU	Ours (T=1)	Ours (T=4)
ckt1	21110	200	144.6	94.2 (1.53×)	47.4 (3.05×)
ckt2	82210	400	1722.4	633.1 (2.72×)	297.5 (5.79×)
ckt3	183309	600	5862.6	2377.6 (2.47×)	1366.1 (4.29×)
ibmpg1t mod.	55356	242	46.2	45.4 (1.02 ×)	36.6 (1.26×)
ibmpg2t mod.	167779	768	963.9	516.3 (1.87 ×)	308.2 (3.13 ×)
Average				**1.92×**	**3.50×**

PARDISO is faster than KLU for some dense circuit matrices, PARDISO is not the best for most circuit matrices since its heavyweight supernode-supernode algorithm is only suitable for very dense matrices. Combining this conclusion and the previous sub-section, we summarize the best algorithm for matrices of different sparsity in Fig. 8.

C. Simulation Results

Thanks to the work that integrates KLU into NGSPICE [14], we are able to integrate our solver into NGSPICE easily. In the NGSPICE-based simulation test, we compare the transient simulation time of NGSPICE when using the proposed solver and KLU respectively. This test is implemented on a Windows 7 desktop computer equipped with an Intel i7-4790 CPU and 16GB memory. All codes are compiled by Visual Studio 2015 with `/O2` optimization. In this test, we use three self-generated circuits and two circuits obtained from IBM power grid benchmarks [19]. Our benchmarks are post-layout-like, i.e., there are a big power network and a big ground network. Since IBM power grid benchmarks are pure linear circuits, some inverter chains are inserted between the power network and the ground network to make the benchmarks nonlinear.

Table II shows the transient simulation time of NGSPICE, along with the matrix dimension, the number of transistors, and the speedups. When the proposed solver is used in NGSPICE, the transient simulation time is reduced by 1.92× and 3.5× on average when using 1 thread and 4 threads respectively, compared with the case where KLU is used. Among the five benchmarks, except for that modified ibmpg1t uses the map algorithm, other benchmarks all use the supernode-column algorithm. The reason why the speedup of modified ibmpg1t is low is that the solver consumes about only 18s (for KLU) in this case. In other words, model evaluation dominates the transient simulation time in modified ibmpg1t. For other benchmarks, the sparse solver dominates the transient simulation time so the speedups are remarkable.

V. Conclusion

This paper demonstrates that the widely used KLU is not always the best algorithm for circuit matrices. The best algorithm strongly depends on the sparsity of the matrix. By carefully designing two algorithms optimized for extremely sparse matrices and dense matrices, and selecting the optimal algorithm according to the sparsity, we can achieve on average 2.8× to 8.36× speedups than KLU, and 1.78× to 3.17× speedups than PARDISO. The NGSPICE transient simulation time can be reduced by up to 5.79× compared with KLU. We hope that this work can promote the development of sparse solvers in commercial circuit simulators.

References

[1] L. W. Nagel, "SPICE 2: A computer program to stimulate semiconductor circuits," Ph.D. dissertation, University of California, Berkeley, California, US, 1975.
[2] R. Daniels, H. V. Sosen, and H. Elhak, "Accelerating Analog Simulation with HSPICE Precision Parallel Technology," Synopsys Corporation, Tech. Rep., 2010.
[3] J. J. Dongarra, J. D. Cruz, S. Hammerling, and I. S. Duff, "Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 18–28, Mar. 1990.
[4] X. S. Li, "Direct Solvers for Sparse Matrices." [Online]. Available: http://crd-legacy.lbl.gov/ xiaoye/SuperLU/SparseDirectSurvey.pdf
[5] O. Schenk and K. Gärtner, "Solving unsymmetric sparse systems of linear equations with PARDISO," *Future Generation Computer Systems*, vol. 20, no. 3, pp. 475 – 487, 2004.
[6] X. S. Li, "An overview of SuperLU: Algorithms, implementation, and user interface," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 302–325, Sep. 2005.
[7] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu, "A Supernodal Approach to Sparse Partial Pivoting," *SIAM J. Matrix Anal. Appl.*, vol. 20, no. 3, pp. 720–755, May 1999.
[8] T. A. Davis, "Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method," *ACM Trans. Math. Softw.*, vol. 30, no. 2, pp. 196–199, Jun. 2004.
[9] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster, "A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling," *SIAM J. Matrix Anal. Appl.*, vol. 23, no. 1, pp. 15–41, Jan. 2001.
[10] T. A. Davis and E. Palamadai Natarajan, "Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems," *ACM Trans. Math. Softw.*, vol. 37, no. 3, pp. 36:1–36:17, Sep. 2010.
[11] K. Kundert and A. Sangiovanni-Vincentelli, "SPARSE 1.3." [Online]. Available: http://www.netlib.org/sparse/
[12] X. Chen, W. Wu, Y. Wang, H. Yu, and H. Yang, "An EScheduler-Based Data Dependence Analysis and Task Scheduling for Parallel Circuit Simulation," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 58, no. 10, pp. 702 –706, oct. 2011.
[13] X. Chen, Y. Wang, and H. Yang, "NICSLU: An Adaptive Sparse Matrix Solver for Parallel Circuit Simulation," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 32, no. 2, pp. 261 –274, feb. 2013.
[14] F. Lannutti, P. Nenzi, and N. Olivieri, "KLU sparse direct linear solver implementation into NGSPICE," in *Mixed Design of Integrated Circuits and Systems (MIXDES) 2012*, May 2012, pp. 69–73.
[15] J. R. Gilbert and T. Peierls, "Sparse partial pivoting in time proportional to arithmetic operations," *SIAM J. Sci. Statist. Comput.*, vol. 9, no. 5, pp. 862–874, 1988.
[16] "NGSPICE." [Online]. Available: http://ngspice.sourceforge.net/
[17] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, dec 2011.
[18] P. R. Amestoy, T. A. Davis, and I. S. Duff, "Algorithm 837: AMD, an approximate minimum degree ordering algorithm," *ACM Trans. Math. Softw.*, vol. 30, no. 3, pp. 381–388, Sep. 2004.
[19] "IBM Power Grid Benchmarks." [Online]. Available: http://dropzone.tamu.edu/∼pli/PGBench/