A Fast Parallel Sparse Solver for SPICE-based Circuit Simulators

Xiaoming Chen, Yu Wang, Huazhong Yang
Department of Electronic Engineering, Tsinghua National Laboratory for Information Science and Technology,
Tsinghua University, Beijing 100084, China
Email: chenxm05@mails.tsinghua.edu.cn, yu-wang@tsinghua.edu.cn, yanghz@tsinghua.edu.cn

***Abstract*—The sparse solver is a serious bottleneck in SPICE-based circuit simulators. Although several existing researches have proposed some circuit simulation-oriented parallel solvers, there is still some room to improve the speed and scalability of these solvers. This paper proposes a fast parallel sparse solver based on a pivoting-reduction technique which takes full advantage of features of circuit simulation. Experimental results show that on average, the proposed solver is up to 50% faster than the state-of-the-art solver NICSLU, and up to 3.3× faster than KLU. Real DC simulation reveals that our solver is faster than NICSLU, PARDISO, and commercial solvers.**

I. Introduction

It is well known that the sparse solver and device evaluation are two most serious bottlenecks in Simulation Program with Integrated Circuit Emphasis (SPICE) [1]-based circuit simulators [2]. These two steps are executed for thousands of times in a transient simulation. For the simulation of small circuits, device evaluation consumes up to three quarters of the simulation time. Although it is very easy to parallelize device evaluation and the scalability can be almost linear, about one third of the simulation including the sparse solver still runs in serial. For large circuits, solving the sparse matrix can cost more than half of the simulation time. Due to the poor scalability of the sparse solver, actual simulators usually scale less than 3× on 8 cores [2]. The irregular structure and high sparsity of circuit matrices make it difficult to parallelize the solver efficiently.

In recent years, some sparse solvers are developed for accelerating SPICE-based circuit simulators, such as KLU [3], NICSLU [4], PARDISO [5], and ShyLU [6]. Among them, KLU is completely serial. KLU is used in the Xyce simulator [7], and it is reported that KLU is high-performance in circuit simulation [8]. NICSLU is based on KLU but it has its own parallel algorithm. NICSLU performs well on circuit matrices but it has not been tested in real circuit simulation. Although PARDISO is not specially for circuit simulation problems, it is also claimed that PARDISO is high-performance in semiconductor device simulation [9]. However, the test cases used in [9] are much denser than circuit matrices created by modified nodal analysis so the performance of PARDISO in general circuit simulation is still unclear. ShyLU which is also a general-purpose sparse solver, is also integrated into the Xyce simulator, the speedup over KLU is up to 20× when using 200 cores [10]. Although the simulation time is greatly reduced by ShyLU, the scalability is not good (ShyLU achieves about only 5× speedup using 50 cores compared with KLU [10]).

This work was supported by 973 project 2013CB329000, National Science and Technology Major Project (2013ZX03003013-003), National Natural Science Foundation of China (No. 61373026, 61261160501), the Importation and Development of High-Caliber Talents Project of Beijing Municipal Institutions, and Tsinghua University Initiative Scientific Research Program.

Some hardware-accelerated sparse solvers are also proposed recently, such as FPGA-based sparse solvers [11], [12] and GPU-based sparse solvers [13], [14]. They can be much faster than modern CPU-based solvers. However, these approaches are all implemented without pivoting due to the inflexible dynamic memory management of these platforms, leading to instable numerical solutions and potential misconvergence of Newton-Raphson iterations. In addition, the memory resources of these hardware platforms are too limited to handle large cases. Consequently, the mainstream of popular circuit simulators still focuses on accelerating simulation by multiply CPUs.

This paper tries to improve the speed and scalability of state-of-the-art circuit simulation-oriented parallel sparse solvers. In this paper, we propose a fast parallel sparse solver improved from the opened source of NICSLU. We make the following contributions in this paper.
(1) We develop a pivoting-reduction technique which takes full advantage of intrinsic features of the Newton-Raphson method, so the new technique is suitable for circuit simulation. The new solver is both fast and stable. Test results show that our solver improves NICSLU on both speed and scalability.
(2) Our solver is integrated into a commercial SPICE-based circuit simulator and evaluated by real-world test cases. Results show that our solver is faster than NICSLU, PARDISO, and commercial solvers.

The rest of this paper is organized as follows. Section II reviews the basic algorithms of KLU and NICSLU, the motivation of this study is also explained. The proposed fast LU factorization algorithm is explained in Section III. Experimental results are presented in Section IV, followed by the conclusion in Section V.

II. Background and Motivation

This section first simply reviews the basic algorithms adopted by KLU [3] and NICSLU [4], which are the base of our proposed algorithm, and then we explain the motivation of this study.

A. KLU

The fundamental algorithm used in KLU is the sparse left-looking algorithm proposed by Gilbert et al. [15]. This algorithm factorizes the matrix column by column, one column at a time. Four sub-steps are executed to factorize a column: symbolic factorization, numerical update, partial pivoting, and pruning, as shown in Algorithm 1.

Symbolic factorization which is implemented by a depth-first-search method [3] computes the nonzero structure of each column, in which the structure of $\mathbf{U}$ indicates the columns that each column depends on. This also means that the column-level dependence information is contained in the nonzero pattern of the factors. Numerical update accumulates the numerical results of the ith column of $\mathbf{L}$ and $\mathbf{U}$ from the numerical results of dependent columns. Partial pivoting selects a sufficiently large element under a given threshold from the ith column of $\mathbf{L}$ and swap it to the diagonal to ensure the numerical stability. Pruning is used to reduce some time cost for later symbolic factorization [16]. Among the four sub-steps, symbolic factorization and numerical update cost most of the total factorization time. Here we only show the overall algorithm and highlight some critical points but ignore the details.

Because partial pivoting permutes the row order, the symbolic structure of $\mathbf{L}$ and $\mathbf{U}$ cannot be determined before factorization, which also means that symbolic factorization cannot be decoupled from numerical computation. This leads to that symbolic factorization is required for each column.

For convenience, the algorithm shown in Algorithm 1 is called a *factorization*. KLU also has another factorization method called *re-factorization*, which is performed without pivoting. Consequently, only numerical update is executed in re-factorization. The re-factorization algorithm uses fixed symbolic structure of the factors and fixed pivoting order which are both obtained from a previous factorization. Re-factorization is obviously faster than factorization but it can cause numerical instable problems.

Algorithm 1 Sparse left-looking LU factorization [15]

```
1: for $i = 1 : n$ do
2:     symbolic factorization: determine the structure of column $i$;
3:     numerical update: solve $\mathbf{Lx} = \mathbf{A}(:, i)$ using Algorithm 2;
4:     partial pivoting: partial pivoting on $\mathbf{x}$ and store $\mathbf{x}$ into $\mathbf{L}$ and $\mathbf{U}$;
5:     pruning: reduce symbolic factorization time for later columns;
6: end for
```

Algorithm 2 Solving $\mathbf{Lx} = \mathbf{A}(:, i)$

```
1: $\mathbf{x} = \mathbf{A}(:, i)$;
2: for $j < i$ where $\mathbf{U}(j, i)$ is a nonzero do
3:     $\mathbf{x}(j+1:n) = \mathbf{x}(j+1:n) - \mathbf{x}(j) \times \mathbf{L}(j+1:n, j)$;
4: end for
```

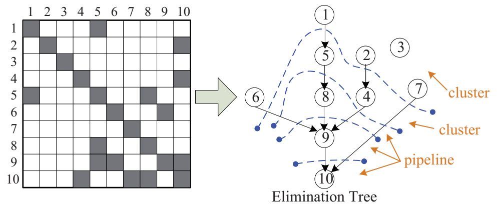


Fig. 1: Scheduling method in NICSLU [4], [17]

B. NICSLU

The fundamental algorithm adopted by NICSLU is generally similar to KLU, the main different is that NICSLU has parallel algorithms. NICSLU uses a column-level dependence graph to schedule the tasks. The dependence is extracted from the symbolic structure of the factors. Since the structure of the factors cannot be determined before factorization, the elimination tree (ET) [18] is used to represent the dependence. ET contains all possible column-level dependence regardless of how pivots are selected, thus it contains much redundant dependence. The ET is further partitioned into levels such that tasks in the same level are completely independent, as shown in Fig. 1. Two scheduling methods are proposed: the cluster method is used for levels which have many tasks (columns) and the pipeline method explores parallelism between dependent levels which have very few tasks [4], [17]. In the cluster method, tasks in one level are equally assigned to threads and all threads compute tasks in parallel. A barrier is required to synchronize each level. In the pipeline method, finer-grained waiting is required instead of the barrier. NICSLU also has the re-factorization algorithm but it is instable either.

C. Motivation

As can be seen, in both KLU and NICSLU, the re-factorization algorithm is faster but it is not stable, and the stable factorization algorithm runs slower. The goal of this paper is to propose a new LU factorization algorithm which will have the advantages of both factorization and re-factorization so it will be both fast and stable.

In circuit simulation, although matrix values change during Newton-Raphson iterations, the values change slow and the difference between two contiguous iterations is small, especially for when the Newton-Raphson method is converging. If the matrix values change very little in two contiguous iterations, the second factorization may use the same pivoting order as the previous one, such that the symbolic structure of the factors is unchanged during the two iterations. This gives us an opportunity that symbolic factorization can be skipped. This feature is intrinsic in the Newton-Raphson method. To our best, there is no similar idea proposed in sparse solver-related publications.

To show the potential of this idea for accelerating the sparse solver in circuit simulation, we show the the percentage of the symbolic factorization time and the numerical update time to the total factorization time in Fig. 2. These matrices are sorted by density. For extremely sparse matrices like circuit matrices, symbolic factorization and numerical update may have similar time cost. For denser matrices, numerical update costs more time than symbolic factorization. The average percentage of symbolic factorization time is 21.6%. For extremely sparse matrices, this percentage is about 30%∼50%. This percentage is also the maximum potential improvement ratio for the sparse solver if symbolic factorization is removed.

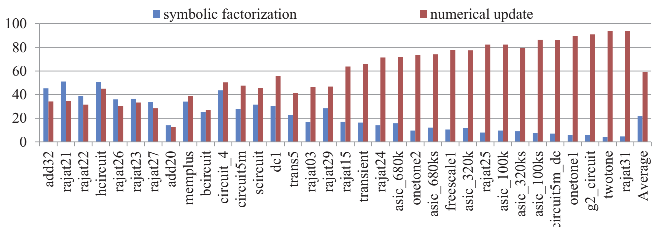


Fig. 2: Percentage of symbolic factorization time and numerical update time to the total factorization time

III. Pivoting-Reduction Based Fast LU Factorization For Circuit Simulation

In this section, we will explain the proposed circuit simulation-oriented fast LU factorization algorithm in detail. The proposed algorithm is an improved version of the sparse left-looking algorithm and we also use the similar scheduling scheme as NICSLU. The fast algorithm has both serial and parallel implementations.

A. Basic Idea

As mentioned in Section II-C, we can accelerate the sparse left-looking LU factorization algorithm by utilizing the intrinsic features of the Newton-Raphson method, because the symbolic and pivoting information can be re-used in subsequent factorizations in Newton-Raphson iterations. More specifically, when computing a column in a subsequent factorization, say column i, if we assume the structure of this column is the same as that in the previous factorization, the symbolic factorization for column i can be skipped. The numerical update can be done as usual. Then the previous pivot choice of column i is checked for whether it is still large enough to be the pivot. If so, the pivot choice is still unchanged for column i and the factorization procedure can be continued without changing the structure of the factors. Otherwise, a re-pivoting is required for column i, then the structure of the factors is changed from column i, so later columns which depend on column i require symbolic factorization to re-compute the structure. This technique is called **pivoting-reduction**. We have developed both serial and parallel implementations of this technique.

B. Serial Algorithm

Algorithm 3 Proposed fast LU factorization algorithm (serial)

```
1: $i = 1$;
2: /*do fast factorization*/
3: while $i \leq n$ do
4:     numerical update on column $i$ using Algorithm 2, using the symbolic
       structure of column $i$ obtained from the previous factorization;
5:     pivot check and re-pivoting using Algorithm 4;
6:     pruning on column $i$;
7:     $i = i + 1$;
8:     if re-pivoting has occurred then
9:         break;
10:    end if
11: end while
12: /*do normal factorization if re-pivoting has occurred*/
13: while $i \leq n$ do
14:     symbolic factorization on column $i$;
15:     numerical update on column $i$ using Algorithm 2;
16:     partial pivoting on column $i$;
17:     pruning on column $i$;
18:     $i = i + 1$;
19: end while
```

Algorithm 4 Pivot check and re-pivoting for column i

```
1: find the element $M$ with the maximum absolute value in $\mathbf{L}(i : n, i)$;
2: find the element $P$ at where the pivot of column $i$ in the previous
   factorization is (i.e. the previous pivot choice);
3: if $|P| < \varepsilon \times |M|$ then
4:     /*re-pivoting required*/
5:     find the diagonal element $D$ of column $i$;
6:     if $|D| < \varepsilon \times |M|$ then
7:         use $M$ as the pivot of column $i$;
8:     else
9:         use $D$ as the pivot of column $i$;
10:    end if
11: else
12:     /*use the previous pivot choice*/
13:     use $P$ as the pivot of column $i$;
14: end if
```

Algorithm 3 shows the pseudo code of the proposed fast serial LU factorization algorithm. Line 3 to line 11 of Algorithm 3 show how the fast sparse left-looking LU factorization is performed. For each column, the symbolic factorization is skipped and the numerical update is performed using the symbolic structure obtained in the previous factorization. Then we do pivot check and re-pivoting if necessary, as shown in Algorithm 4, where ε is a user-given threshold to control the pivot choice.

In Algorithm 4, we first check the previous pivot choice for whether it is now still large enough to be the pivot (line 3). If so, we still use the same pivot choice as the previous factorization (line 13) so the symbolic structure of column i will not be changed. However, if the previous pivot choice cannot be the pivot now, re-pivoting is required for column i (line 4 to line 10). In re-pivoting, the diagonal element is first checked. If it is large enough, the pivot of column i is the diagonal element (line 9); otherwise the maximum element of column i of $\mathbf{L}$ is selected as the pivot (line 7). This method makes the proposed fast LU factorization algorithm re-use previous information including the symbolic structure and the pivoting choice as much as possible in the Newton-Raphson method, such that the symbolic factorization can be skipped in as many columns as possible.

Once a re-pivoting occurs, the fast factorization is stopped and the rest of columns are computed by the normal factorization algorithm (line 13 to line 19). It should be noticed that when re-pivoting occurs at a column i, not all the subsequent columns (i.e. columns $i+1$, $i+2$, $\cdots$, n) are required to perform the normal factorization algorithm. Only the columns which depend on column i require normal factorization. However, searching for all dependent columns from the subsequent columns will traverse all the subsequent columns so it is time-consuming. Consequently, we use a simple method that once a re-pivoting occurs, all the subsequent columns are computed by the normal factorization algorithm.

C. Parallel Algorithm

We use a similar scheduling scheme including the cluster method and pipeline method [4], [17] as NICSLU to perform the parallel task scheduling. We call our new cluster method and new pipeline method *fast cluster* and *fast pipeline*. The two methods in NICSLU are called *normal cluster* and *normal pipeline*. Fig. 3 shows the scheduling framework of the fast parallel factorization algorithm. If no re-pivoting occurs, fast cluster and fast pipeline are executed. If re-pivoting occurs in fast cluster, the rest of the levels belonging to the cluster method are computed by normal cluster and other levels are computed by normal pipeline. If re-pivoting occurs in fast pipeline, the fast pipeline stops and a new normal pipeline is invoked for the rest of columns.

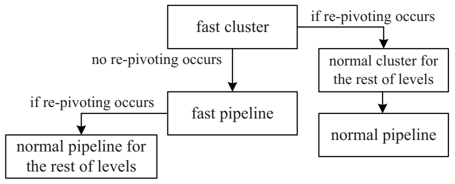


Fig. 3: Scheduling framework of the proposed fast LU factorization algorithm (parallel)

The fast cluster method computes columns using the code shown in line 4 to line 10 of Algorithm 3. Algorithm 5 shows the pseudo code of the fast pipeline method. When computing column i, a different numerical update method is performed, which is shown in Algorithm 6. For each column j that column i depends on, we wait for it to finish. If re-pivoting has occurred in column j, this indicates that the symbolic structure of column i is changed from that obtained from the previous factorization (because column i depends on column j and the symbolic structure of column j is changed), so the pipeline must be stopped and a new normal pipeline method is launched from column i. If no re-pivoting in dependent columns is detected, columns i continues to execute pivot check and re-pivoting using Algorithm 4. If re-pivoting has occurred in column i, the fast pipeline method is also stopped and a new normal pipeline is launched for the rest of columns.

Algorithm 5 Fast pipeline method of the proposed fast LU factorization algorithm for computing column i

```
 1: numerical update on column $i$ using Algorithm 6, using the symbolic
    structure of column $i$ obtained from the previous factorization;
 2: if numerical update returns false then
 3:     return false;
 4: end if
 5: pivot check and re-pivoting using Algorithm 4;
 6: pruning on column $i$;
 7: $i = i + 1$;
 8: if re-pivoting has occurred then
 9:     return false;
10: end if
11: return true;
```

Algorithm 6 Numerical update in the fast pipeline method

```
1: $\mathbf{x} = \mathbf{A}(:, i)$;
2: for $j < i$ where $\mathbf{U}(j, i)$ is a nonzero do
3:     wait for column $j$ to finish;
4:     if re-pivoting has occurred in column $j$ then
5:         return false;
6:     end if
7:     $\mathbf{x}(j+1:n) = \mathbf{x}(j+1:n) - \mathbf{x}(j) \times \mathbf{L}(j+1:n, j)$;
8: end for
9: return true;
```

D. Remarks

From the algorithms presented above, it can be concluded that the performance of the proposed fast algorithms strongly depend on the differences of matrices during Newton-Raphson iterations. If the matrices change little during iterations, each fast LU factorization can always use the previous pivoting order such that no re-pivoting happens. This is the best case. On the contrary, if matrices change dramatically, re-pivoting will always happen. The worst case is that re-pivoting always happens at the first column of each fast LU factorization such that the proposed solver degenerates to NICSLU. Consequently, the performance of our solver should be always faster than or the same as that of NICSLU.

IV. Evaluation Results

A. Experimental Setup

Our proposed solver is evaluated by two schemes: benchmark test and real case test. For benchmark test, we compare the proposed solver with NICSLU [4] and KLU [3] on public matrix benchmarks. Benchmarks (all benchmarks used in this paper are circuit matrices) are obtained from University of Florida Sparse Matrix Collection [19]. NICSLU and KLU both use default configurations. For real case test, our solver, NICSLU, and PARDISO [5] (version 10.2.7, from Intel Math Kernel Library) are all integrated into a commercial SPICE-based circuit simulator and we compare their performance in DC simulation. The two test schemes are executed on two different machines, as shown in Table I.

TABLE I: Experimental setup

	benchmark test	real case test
CPU	2×Intel Xeon E5-2690	Intel Xeon X5650
memory	64GB	48GB
operating system	CentOS 6.5 X64	RedHat 5.7 X64
compiler (optimization)	icc 14.0.2 (-O3)	gcc 4.7.2 (-O3)

B. Experimental Methodology

As mentioned in Section III-D, the performance of the proposed solver strongly depends on the differences of matrices during Newton-Raphson iterations. To show the potential of the proposed solver for accelerating circuit simulation, we show results under the best case in benchmark test. We run LU factorization for twice, the first factorization is a normal left-looking factorization which is the same as that in NICSLU, and the second factorization will be the proposed algorithm. Since the matrix is identical in the two factorizations, no re-pivoting will happen in the second factorization. This produces the best case. Real case test will show the real results of the proposed solver in DC simulation.

Two types of speedups are defined: pure *speedup* means the acceleration ratio of some solver compared with another solver; *relative speedup* means the acceleration ratio of some parallel solver compared with itself of single thread, which also means the scalability.

C. Benchmark Test

1) Scalability and Comparison with NICSLU: Our proposed algorithm is directly improved from NICSLU, so comparing with NICSLU will straightforwardly show the efficiency of our algorithm. Table II shows the sum of factorization time and substitution time of NICSLU and the proposed solver. These benchmarks are sorted by their $flops\_per\_nonzero = \frac{flops}{NNZ(\mathbf{L}+\mathbf{U}-\mathbf{I})}$ values ($flops$: number of floating-point operations; NNZ: number of nonzeros). $flops\_per\_nonzero$ can be regarded as the density of the factors. Note that NICSLU has an important feature that it can automatically choose serial or parallel factorization according to $flops\_per\_nonzero$ [4]. Our solver inherits from NICSLU so our solver also has this feature. Our solver and NICSLU both use serial factorization for the first 17 matrices shown in Table II, so the computational time under different number of threads is almost the same for these 17 matrices.

Fig. 4 shows the speedup of the proposed solver compared with NICSLU. Our solver achieves an average speedup of 1.32×, 1.46× and 1.50× when the number of threads is 1, 4 and 8, respectively. The speedup trends to be higher for sparser matrices because the ratio of symbolic factorization time to the total factorization time is higher for sparser matrices. The speedup is close to the maximum potential improvement ratio presented in Section II-C.

Fig. 5 compares the relative speedup of NICSLU and our solver for the last 17 matrices on which NICSLU and our solver both use parallel factorization. NICSLU achieves an average relative speedup of 1.85× and 2.48×, when the number

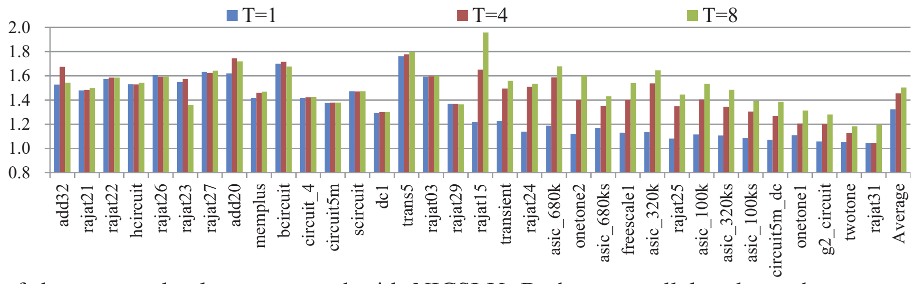


Fig. 4: Speedup of the proposed solver compared with NICSLU. Both are parallel and use the same number of threads

TABLE II: Factorization time+substitution time (second) of NICSLU and our proposed solver

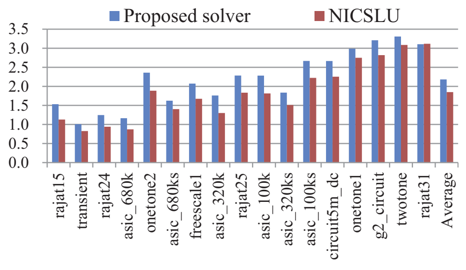
	NICSLU			Proposed solver		
	T=1	T=4	T=8	T=1	T=4	T=8
add32	0.001	0.001	0.001	0.000	0.000	0.000
rajat21	0.094	0.094	0.095	0.063	0.063	0.064
rajat22	0.009	0.009	0.009	0.006	0.006	0.006
hcircuit	0.016	0.015	0.015	0.010	0.010	0.010
rajat26	0.012	0.012	0.012	0.008	0.008	0.008
rajat23	0.029	0.029	0.025	0.019	0.018	0.018
rajat27	0.006	0.006	0.006	0.003	0.003	0.003
add20	0.001	0.001	0.001	0.001	0.001	0.001
memplus	0.004	0.004	0.004	0.003	0.003	0.003
bcircuit	0.041	0.041	0.040	0.024	0.024	0.024
circuit_4	0.013	0.013	0.013	0.009	0.009	0.009
circuit5m	1.976	1.977	1.977	1.436	1.433	1.434
scircuit	0.087	0.087	0.087	0.059	0.059	0.059
dc1	0.045	0.044	0.044	0.035	0.034	0.034
trans5	0.058	0.058	0.058	0.033	0.033	0.033
rajat03	0.008	0.008	0.008	0.005	0.005	0.005
rajat29	0.530	0.531	0.530	0.387	0.388	0.388
rajat15	0.105	0.093	0.095	0.086	0.056	0.049
transient	0.168	0.203	0.192	0.137	0.136	0.123
rajat24	0.413	0.439	0.409	0.363	0.291	0.266
asic_680k	0.693	0.796	0.787	0.583	0.502	0.469
onetone2	0.133	0.071	0.058	0.119	0.050	0.036
asic_680ks	0.563	0.401	0.350	0.482	0.297	0.244
freescale1	7.945	4.749	4.012	7.028	3.393	2.605
asic_320k	0.742	0.571	0.515	0.653	0.371	0.313
rajat25	0.371	0.203	0.161	0.343	0.150	0.112
asic_100k	0.606	0.334	0.273	0.543	0.238	0.178
asic_320ks	0.729	0.482	0.411	0.658	0.359	0.277
asic_100ks	0.700	0.315	0.235	0.644	0.242	0.169
circuit5m_dc	18.519	8.217	6.077	17.267	6.477	4.386
onetone1	0.604	0.220	0.148	0.545	0.182	0.112
g2_circuit	5.751	2.040	1.327	5.439	1.695	1.037
twotone	5.315	1.722	0.976	5.047	1.526	0.825
rajat31	262.371	84.167	54.465	250.555	80.686	45.698

of threads is 4 and 8, respectively, while the average relative speedup of our solver is 2.18× and 3.15×. This comparison indicates that our solver scales better than NICSLU.

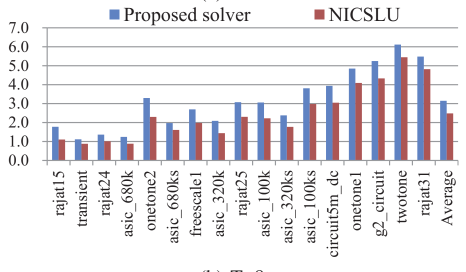
2) Comparison with KLU: Fig. 6 shows the speedup of our solver compared with KLU. Our solver is on average (geometric mean) 1.94×, 2.81× and 3.27× faster than KLU, when the number of threads of our solver is 1, 4 and 8, respectively. For several matrices, our solver is much faster than KLU, this is because of the number of fill-ins. For these matrices, out solver generates much fewer fill-ins than KLU.

D. Real Case Test

Our proposed solver, NICSLU, and PARDISO are all integrated into a commercial SPICE-based circuit simulator. In this section, we compare the DC performance of the three solvers on real cases of different sizes. Tests results are shown in Table III. All the three solvers use 4 threads to performance the DC simulation. The proposed solver is always faster than NICSLU and achieves an average improvement of



(a) T=4



(b) T=8

Fig. 5: Scalability (relative speedup) of the proposed solver and NICSLU

16% (geometric mean) compared with NICSLU in real DC simulation. This improvement is smaller than that reported in benchmark test because of two main reasons: first, the circuit simulator has some serial work (device evaluation and other sundries) which affects the parallel performance and reduces the efficiency of parallel code according to the Amdahl's law [20]; second, the improvement reported in benchmark test is under the base case. If device evaluation is also parallelized, the speedup will be higher. NICSLU is on average 38% faster (geometric mean) than PARDISO in real DC simulation.

We also compare our solver with 2 default solvers of the commercial simulator on the same cases. The average speedup is about 2× to 3×. For the reason of trade secret, detailed results are not shown.

V. Conclusion

Sparse solver is an important component in SPICE-based simulators. This paper has proposed a fast parallel sparse solver for circuit simulation problems. The new solver is based on a pivoting-reduction technique which takes full advantage of intrinsic features of the Newton-Raphson method. During Newton-Raphson iterations, matrix values change slow so the symbolic structure and pivoting information obtained from a previous factorization can be re-used in subsequent factorizations, such that the subsequent factorizations can be accelerated by skipping the symbolic factorization for some columns. Benchmark test shows that our new solver is faster

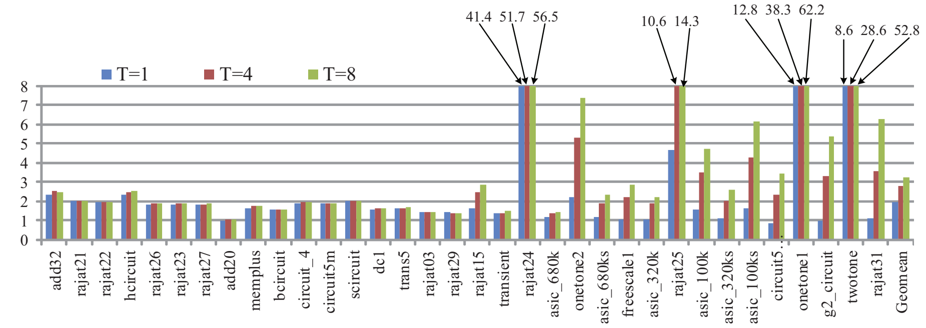


Fig. 6: Speedup of the proposed solver compared with KLU. KLU is serial

TABLE III: Wall time of DC simulation and the speedups of the proposed solver over NICSLU and PARDISO, all the three solvers use 4 threads

ckt	dimension*	DC simulation wall time (second)			speedup	
		NICSLU	PARDISO	Proposed	vs. NICSLU	vs. PARDISO
1	1,163,448	1368	14140	1281	1.07	11.04
2	496,540	109	128	95	1.15	1.35
3	445,592	82	111	75	1.09	1.48
4	298,270	194	279	180	1.08	1.55
5	296,612	169	201	158	1.07	1.27
6	206,996	147	123	142	1.04	0.87
7	120,921	69	470	65	1.06	7.23
8	95,314	1791	625	1577	1.14	0.40
9	60,068	394	271	334	1.18	0.81
10	82,257	204	156	187	1.09	0.83
11	81,437	2510	2231	2118	1.19	1.05
12	59,313	5379	1422	2526	2.13	0.56
13	21,235	199	296	189	1.05	1.57
arithmean					**1.18**	**2.31**
geomean					**1.16**	**1.38**

* Dimension means the size of the matrix extracted from the circuit simulator.

than existing NICSLU and KLU. Real case test reveals that the new solver is on average 16% and 38% faster (geometric mean) than NICSLU and PARDISO respectively in real DC simulation. Our solver is also much faster than commercial solvers in real DC simulation.

The proposed idea can be used not only in circuit simulation, but also in any other sparse solver which is used in the Newton-Raphson method, because our idea is based on the intrinsic features of the Newton-Raphson method. In addition, due to the inflexibility of dynamic memory management and the poor performance on irregular operations (e.g. branch, loop), many existing GPU-based implementations of sparse solvers are without pivoting so they are instable. Our idea increases the possibility of porting the fast and stable factorization code to GPUs because the symbolic factorization which has many branch/loop operations and dynamic memory allocations is avoided.

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] 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.

[4] 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.

[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] S. Rajamanickam, E. Boman, and M. Heroux, "ShyLU: A Hybrid-Hybrid Solver for Multicore Platforms," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, 2012, pp. 631–643.

[7] R. L. Schiek and E. E. May, "Xyce Parallel Electronic Simulator: Biological Pathway Modeling and Simulation," Sandia National Laboratory, Tech. Rep., 2006.

[8] M. Sipics, "Sparse matrix algorithm drives SPICE performance gains," *SIAM News*, vol. 40, no. 4, 2007.

[9] O. Schenk, K. Gärtner, W. Fichtner, and A. Stricker, "PARDISO: a high-performance serial and parallel sparse linear solver in semiconductor device simulation," *Future Generation Computer Systems*, vol. 18, no. 1, pp. 69 – 78, 2001.

[10] H. K. Thornquist and S. Rajamanickam, "A Hybrid Approach for Parallel Transistor-Level Full-Chip Circuit Simulation," Sandia National Laboratory, Tech. Rep., 2014.

[11] N. Kapre and A. DeHon, "Parallelizing sparse Matrix Solve for SPICE circuit simulation using FPGAs," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, Dec 2009, pp. 190–198.

[12] T. Nechma and M. Zwolinski, "Parallel Sparse Matrix Solution for Circuit Simulation on FPGAs," *Computers, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2014.

[13] X. Chen, L. Ren, Y. Wang, and H. Yang, "GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling," *Parallel and Distributed Systems, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2014.

[14] L. Ren, X. Chen, Y. Wang, C. Zhang, and H. Yang, "Sparse LU factorization for parallel circuit simulation on GPU," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, june 2012, pp. 1125–1130.

[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] S. C. Eisenstat and J. W. H. Liu, "Exploiting structural symmetry in a sparse partial pivoting code," *SIAM J. Sci. Comput.*, vol. 14, no. 1, pp. 253–257, Jan. 1993.

[17] 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.

[18] J. W. H. Liu, "The role of elimination trees in sparse factorization," *SIAM J. Matrix Anal. Appl.*, vol. 11, no. 1, pp. 134–172, Jan. 1990.

[19] 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.

[20] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, 1967, pp. 483–485.