Numerically-Stable and Highly-Scalable Parallel LU Factorization for Circuit Simulation

Xiaoming Chen
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
chenxiaoming@ict.ac.cn

ABSTRACT

A number of sparse linear systems are solved by sparse LU factorization in a circuit simulation process. The coefficient matrices of these linear systems have the identical structure but different values. Pivoting is usually needed in sparse LU factorization to ensure the numerical stability, which leads to the difficulty of predicting the exact dependencies for scheduling parallel LU factorization. However, the matrix values usually change smoothly in circuit simulation iterations, which provides the potential to "guess" the dependencies. This work proposes a novel parallel LU factorization algorithm with pivoting reduction, but the numerical stability is equivalent to LU factorization with pivoting. The basic idea is to reuse the previous structural and pivoting information as much as possible to perform highly-scalable parallel factorization without pivoting, which is scheduled by the "guessed" dependencies. Once a pivot is found to be too small, the remaining matrix is factorized with pivoting in a pipelined way. Comprehensive experiments including comparisons with state-of-the-art CPU- and GPU-based parallel sparse direct solvers on 66 circuit matrices and real SPICE DC simulations on 4 circuit netlists reveal the superior performance and scalability of the proposed algorithm. The proposed solver is available at https://github.com/chenxm1986/cktso.

KEYWORDS

Circuit simulation, parallel sparse LU factorization, pivoting reduction, numerical stability

1 INTRODUCTION

The famous circuit simulation kernel, SPICE, utilizes the theory of numerical computation to find the responses of integrated circuits. In a typical SPICE simulation process, a number of linear systems ($\mathbf{Ax} = \mathbf{b}$) need to be solved. The linear solver is usually the performance bottleneck of circuit simulators. For example, in a post-layout simulation, solving linear systems can take more than 3/4 of the total simulation time [8]. Sparse LU factorization [12] is widely adopted in SPICE-based simulators to solve linear systems. Sparse LU factorization involves three steps: symbolic analysis, numerical factorization and substitution. Fig. 1 shows the usage of a sparse solver in a time-domain simulation flow. The 1st step reorders the matrix to minimize fill-ins that will be generated during factorization. The 2nd step factorizes the matrix into *LU factors*: $\mathbf{A} = \mathbf{LU}$, where $\mathbf{L}$ is a lower-triangular matrix and $\mathbf{U}$ is an upper-triangular matrix. The last step computes $\mathbf{x}$ by two triangular solving phases ($\mathbf{Ly}=\mathbf{b}$ and $\mathbf{Ux}=\mathbf{y}$). Numerical factorization and substitution are executed in every Newton-Raphson (NR) iteration, where the former spends much longer time (typically >10-100×) than the latter.

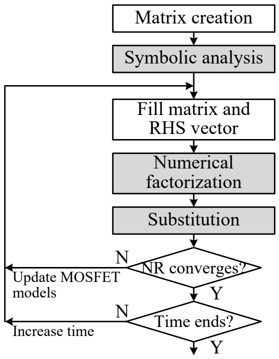


Figure 1: Sparse direct solver in a time-domain circuit simulation flow (RHS: right-hand-side).

Matrix $\mathbf{A}$ involved in the NR iterations of a circuit simulation process has some special features. The structure of $\mathbf{A}$ is fixed. Thus, symbolic analysis is performed only once. The values of $\mathbf{A}$ change in every NR iteration. As a result, *pivoting* (swapping large elements to the diagonal) is usually needed in LU factorization, due to the fact that small diagonal elements (i.e., *pivots*) will amplify the values of the LU factors and lead to round-off errors. While pivoting improves numerical stability, it brings row/column exchanges. This results in the need of dynamic determination of the structure of the LU factors, which cannot be decoupled from numerical computation. Instead, it has to be interleaved with numerical factorization.

KLU (sequential) [11] and NICSLU (parallel) [2, 5] are two state-of-the-art sparse solvers specially designed for circuit simulation. They both provide two factorization functions: **factorization with pivoting** and **re-factorization without pivoting**. The latter reuses the pivoting order generated in the last factorization with pivoting, so that the structure of the LU factors is reused and symbolic-related operations can be skipped, which improves the performance. Another essential difference is in the parallel scheduling. The task dependencies are determined by the structure of the LU factors. Re-factorization reuses the previous structure of the LU factors, so an exact dependency graph can be constructed before re-factorization, to schedule parallel re-factorization [5]. For factorization with pivoting, however, since the structure of the LU factors is only known after factorization is finished, determining the exact dependencies before factorization is impossible. People have proposed the concept of *elimination tree (ETree)* [18], which describes an **upper bound** of the dependencies when considering pivoting. In other words, the dependencies generated by **any** pivoting order are contained in the ETree. Thus, the ETree overestimates the dependencies. Overestimated dependencies imply poor scalability for parallel scheduling.

• This work was supported by National Natural Science Foundation of China under Grant 62122076 and by Innovative Project of Institute of Computing Technology, Chinese Academy of Sciences under Grant E261040.

• This paper was presented in IEEE/ACM International Conference on Computer-Aided Design (ICCAD'22) and will be published in ACM Digital Library and/or IEEE Xplore.

Though re-factorization without pivoting provided by KLU and NICSLU eliminates symbolic-related operations and provides better performance than factorization with pivoting, the changing matrix values during circuit simulation iterations may cause unstable re-factorization results, which may cause divergence of the NR method. Hence, to ensure the numerical stability, factorization and re-factorization must be carefully selected. Intuitively, when the matrix values change little, skipping pivoting tends to be numerically stable; when the matrix values change greatly, pivoting is needed, at the cost of lower performance and scalability. However, judging whether the matrix values change much is not an easy task, mainly due to the determination of the threshold which is problem dependent and needs much experience. In this work, we propose a novel parallel LU factorization algorithm which explores the advantages of both factorization with pivoting and re-factorization without pivoting. Its performance and scalability are similar to those of re-factorization but the numerical stability is equivalent to that of factorization. The contributions of this paper are as follows.

- A novel parallel LU factorization algorithm for circuit simulation is proposed, which takes full advantages of both factorization with pivoting and re-factorization without pivoting.
- The proposed algorithm combines a parallel re-factorization algorithm with pivot check and a pipelined factorization algorithm with pivoting. By taking into account the numerical features of circuit simulation and utilizing previous structural and pivoting information as much as possible, the proposed algorithm achieves high performance, scalability and numerical stability at the same time.
- Comprehensive experiments including comparisons with state-of-the-art CPU- and GPU-based parallel sparse direct solvers, as well as real SPICE DC simulations, reveal the superior performance and scalability of the proposed algorithm. Our solver is publicly available.

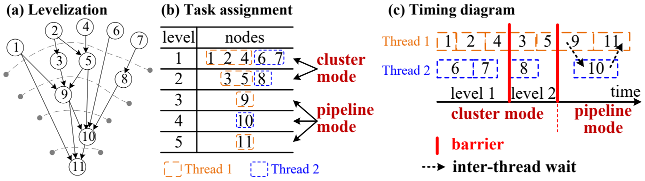
2 BACKGROUNDS

2.1 Sparse Up-Looking LU Factorization

Sparse left-looking LU factorization [14] is widely adopted by sparse solvers for circuit simulation (e.g., KLU [11] and NICSLU [2, 5]). Left-looking computes the LU factors column by column. This work adopts the transposed version, which can be named as sparse up-looking LU factorization, as listed in Algorithm 1. It computes **L** and **U** row by row (line 1). In the algorithm, **x** is a vector of length N for holding intermediate results of the current row. For each row, symbolic prediction (line 2), numerical update (lines 3-5) and pivoting (lines 6-7) are executed. Symbolic prediction (line 2) is to determine the structure of the current row, through a depth-first search algorithm on already computed rows [11, 14]. As pivoting changes the structure of the factors, symbolic prediction must be interleaved with numerical update in every row. Numerical update uses dependent rows on the upper side to update the values of the current row. Sparse direct solvers typically use a threshold-based partial pivoting method. Line 6 shows that only when the diagonal element is smaller than the maximum element in the current row of **U** times a threshold (ε), we exchange the diagonal element and the largest element in the current row of **U** (line 7).

```
1 for $i = 1 : N$ do
2     Symbolic prediction: find structures of $\mathbf{L}(i, 1{:}i)$ and $\mathbf{U}(i, i{:}N)$;
3     $\mathbf{x} = \mathbf{A}(i, :)$;
4     for $j = 1 : i - 1$ where $\mathbf{L}(i, j)$ is nonzero do
5         $\mathbf{x}(j+1:N) = \mathbf{x}(j+1:N) - \mathbf{x}(j) \times \mathbf{U}(j, j+1:N)$;
6     if $|\mathbf{x}(i)| < \varepsilon \times \max_{i+1 \le k \le N} \{|\mathbf{x}(k)|\}$ then
7         Exchange $\mathbf{x}(i)$ with $\mathbf{x}(\arg\max_{i+1 \le k \le N} \{|\mathbf{x}(k)|\})$;
8     $\mathbf{L}(i, 1:i) = \mathbf{x}(1:i)$;
9     $\mathbf{U}(i, i:N) = \mathbf{x}(i:N)/\mathbf{x}(i)$;
```

Algorithm 1: Sparse up-looking LU factorization [14].



level	nodes
1	1 2 4 6 7
2	3 5 8
3	9
4	10
5	11

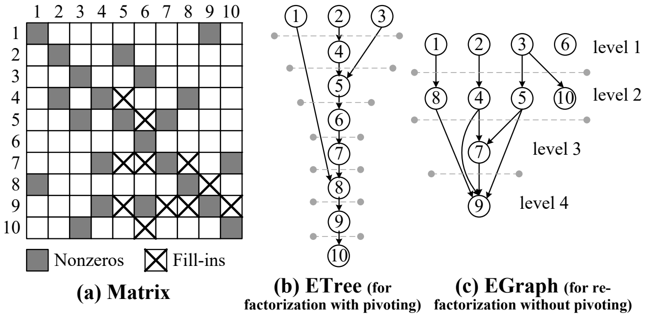
Figure 2: Levelization-based parallel scheduling [2, 5].

If symbolic prediction (line 2) and pivoting (lines 6-7) are removed, the algorithm becomes re-factorization without pivoting. Re-factorization reuses the structure of the LU factors and the pivoting order found in the previous factorization with pivoting, and only numerical update is performed. If one or more small pivots that do not meet the pivoting rule appear in re-factorization, they cannot be handled and may lead to large errors in the solution.

2.2 Dependency Description and Parallelization

The sparsity implies the parallelism. From lines 4 and 5 of Algorithm 1 we can find that **row i depends on row j (i.e., row j updates row i) if and only if $\mathrm{L}(i, j)$ is a nonzero element and $\mathrm{U}(j, j+1:N)$ has at least one nonzero element**. Based on this principle, a row-level dependency graph can be constructed based on the structure of the LU factors, which is named as an *elimination graph (EGraph)*. The EGraph describes exact inter-row dependencies corresponding to a specific symbolic structure of the LU factors, and can be used for scheduling parallel re-factorization without pivoting [5]. For parallel factorization with pivoting, however, since the structure of the LU factors is known only after factorization is completed, it is impossible to use the EGraph (i.e., the exact dependencies) for parallel scheduling. As mentioned earlier, the ETree [18] which describes an upper bound of the inter-row dependencies can be employed. The ETree is constructed based on the structure of matrix **A**, so it is independent with the LU factors.

The ETree and EGraph are both directed acyclic graphs. A levelization-based method [2, 5] may explore the parallelism to schedule parallel factorization and re-factorization. As illustrated in Fig. 2, the EGraph or ETree can be levelized so that nodes (a node corresponds to a row) in the same level have no dependency. The *level* of a node is the maximum path length from any source node to itself. NICSLU [2, 5] uses a duel-mode scheduling method for nodes with different dependencies. For front levels that have many nodes, they are processed level by level. Nodes in a level are computed by threads in parallel without dependencies. This is the so-called **cluster mode**. A barrier synchronization is needed for each level. For the remaining levels that have very few nodes in each level,



(a) Matrix (b) ETree (for factorization with pivoting) (c) EGraph (for re-factorization without pivoting)

Figure 3: Matrix example and its corresponding ETree and EGraph (assuming pivots are original diagonal elements).

a **pipeline mode** is proposed which explores finer-grained parallelism between dependent rows. The levelization-based method has been used for scheduling parallel sparse solvers not only on CPUs, but also on GPUs (e.g., [1, 16, 20]).

2.3 Motivation

The height and width of the ETree/EGraph can be an intuitive estimation of the scalability. The height (maximum level) is the critical path length and the width (maximum number of nodes in each level) is the maximum parallelism. As illustrated in Fig. 3, the ETree is tall and narrow, and the EGraph is short and wide. To be more specific, we have tested 66 circuit matrices from the SuiteSparse Matrix Collection [9]. On geometric mean, the ETree is 77.4× taller than the EGraph, while the EGraph is 10.5× wider. Therefore, it is expected that the scalability of parallel factorization with pivoting (scheduled by the ETree) is much poorer than that of re-factorization without pivoting (scheduled by the EGraph). According to the results in [4], NICSLU only achieves about 2× speedup on average in factorization with pivoting when using 16 threads. The primary purpose of this work is to explore the advantages of parallel re-factorization without pivoting to improve the scalability of parallel factorization with pivoting.

In circuit simulation, considering that the NR iterations will eventually converge in DC simulation or at each time node in transient simulation, the matrix values usually change smoothly, and successive iterations tend to reuse most of the previous pivoting order. This intuition provides an opportunity to "guess" the exact inter-row dependencies before factorization (except the first-time factorization). We aggressively assume that the structure of the LU factors and the pivoting order are not changed from the previous factorization, and tentatively use the "guessed" EGraph to schedule parallel re-factorization, where pivots are checked. The original re-factorization algorithm provided by KLU [11] and NICSLU [2, 5] does not perform pivot check. Even if pivot check is added, they can only exit when an unsatisfactory pivot is found. This work provides a novel mechanism to restart factorization with pivoting from the interrupted row (i.e., the row with an unsatisfactory pivot).

Since the EGraph is "guessed" based on the structure of the LU factors obtained in the previous factorization, when an unsatisfactory pivot is found, the EGraph can no longer be used because re-pivoting is needed and the structure of the remaining LU factors will change. Instead, the ETree that contains all possible inter-row dependencies can be used for scheduling the remaining factorization with pivoting. A key question must be answered: which nodes will be computed with pivoting. For instance, in Fig. 3(c), if the first two levels of the EGraph are finished and an unsatisfactory pivot is found in node 7, nodes 7, 8, 9 and 10 need to be computed with pivoting according to the ETree in Fig. 3(b), though nodes 8 and 10 have already been computed in re-factorization with pivot check.

In circuit simulation iterations, since the matrix values usually change smoothly, we have a high possibility that re-pivoting does not happen so that the "guessed" EGraph can be used for the entire matrix. Even if re-pivoting happens in some row, the factorization of part matrix is scheduled by the EGraph, so the overall performance and scalability may still be improved. By using our strategy, the advantages of both factorization with pivoting and re-factorization without pivoting are given full play.

3 RELATED WORKS

There are many popular sparse solvers (e.g., KLU [11], SuperLU [13], PARDISO [22], UMFPACK [10], et al.). Most of them are targeted at general applications rather than circuit simulation. As circuit matrices are generally much sparser than matrices from other applications [11], general-purpose sparse solvers typically do not perform well on circuit matrices. Only a few sparse solvers specially designed for circuit simulation: KLU [11] and NICSLU [2, 5], where the former is sequential and the latter is parallel.

NICSLU [2, 5] and multi-threaded SuperLU [13] both use the ETree [18], which describes an upper bound of the dependencies, to schedule parallel factorization with pivoting. For re-factorization of NICSLU, it uses the EGraph which is an exact representation of the dependencies, to schedule parallel tasks [5]. NICSLU achieves better performance than other popular solvers (KLU, PARDISO, SuperLU, UMFPACK, etc.) in real simulation problems [6, 7, 21]. PARDISO [22] is generally the best sparse solver for a wide range of applications except circuit simulation [15]. It selects pivots within dense diagonal sub-blocks so that the dependency graph is not changed with pivoting. This enables to build an exact dependency graph before factorization. However, the shrunken pivoting range reduces numerical accuracy of solutions, and sometimes an iterative refinement [19] is needed due to inaccurate solutions.

NICSLU integrates a fast parallel factorization algorithm by eliminating symbolic-related operations [3]. It uses the ETree to schedule parallel factorization level by level. Symbolic prediction and pivoting are skipped. Once an unsatisfactory pivot is found at some level, all the remaining levels are computed with symbolic prediction and pivoting. The purpose of NICSLU's fast factorization is just to eliminate symbolic-related operations, which improves the factorization performance by 30-50% at most. However, the scalability of NICSLU's fast factorization is almost not improved compared with the original factorization without skipping symbolic-related operations, due to the poor scalability implied in the ETree. Differently, the primary purpose of this work is to improve the scalability of parallel factorization using the dependencies of re-factorization without pivoting, which implies much better scalability, and the elimination of symbolic-related operations is just a by-product.

There are also some GPU accelerations [1, 16, 20, 23] for sparse solvers for circuit simulation. GPUs provide much higher parallelism and performance than CPUs. However, the performance of sparse direct solvers on GPUs is highly constrained by the memory bandwidth, so it is not easy to exploit the high performance and parallelism for sparse direct solvers on GPUs. Recent results show that a right-looking algorithm (RLA) based GPU-based solver for circuit simulation is slower than NICSLU for most circuit matrices [16]. Though SFLU [23] is found to be faster than SuperLU_DIST [17] for a large number of matrices, the majority of the benchmarks are not circuit matrices and SuperLU_DIST is not designed for circuit simulation. Due to the difficulty of implementing runtime memory allocation on GPUs, all of these GPU-based sparse solvers for circuit simulation only accelerate re-factorization without pivoting.

4 PROPOSED ALGORITHM

Considering that parallel factorization with pivoting has to be scheduled by the ETree which implies strong dependencies, the primary purpose of this work is to build a parallel factorization algorithm with improved scalability, by exploring advantages of parallel re-factorization without pivoting, while the numerical stability is not affected. As explained in Section 2.3, the basic idea is to exploit the numerical features of circuit simulation and aggressively use the "guessed" EGraph that is constructed based on the structure of the LU factors of the previous factorization to schedule parallel re-factorization with pivot check. Once a pivot is found to be too small, re-pivoting is needed and the algorithm switches to parallel factorization with pivoting for the remaining matrix. The restart point should be correctly determined.

4.1 Overall Flow

Fig. 4 shows the overall flow of the proposed parallel LU factorization algorithm. In each NR iteration of circuit simulation, a linear system needs to be solved. Based on the structure of the LU factors obtained in the previous iteration, we construct a "guessed" EGraph using the dependency principle mentioned in Section 2.2. If the structure of the LU factors does not change in successive iterations, the "guessed" EGraph is constructed only once. The "guessed" EGraph is tentatively used for scheduling parallel re-factorization with pivot check. If one or more unsatisfactory pivots are found in parallel re-factorization with pivot check, re-factorization is interrupted and pipelined tail factorization with pivoting, which is scheduled by the ETree, will be called to compute the remaining matrix. Before that, we need to determine the row from which tail factorization with pivoting starts. On the contrary, if re-pivoting is not needed, it means that the structure of the LU factors does not change and the "guessed" EGraph is correct. In this case, the entire matrix is computed by parallel re-factorization with pivot check, which is the best case. The best case is expected to have similar performance and scalability to re-factorization without pivoting. The worst case happens when the first row needs re-pivoting so that the entire matrix is computed with pivoting. The worst case has similar performance and scalability to factorization with pivoting.

4.2 Parallel Re-Factorization with Pivot Check

To schedule parallel re-factorization with pivot check, the "guessed" EGraph is levelized [2, 5] by calculating the levels of all nodes, as illustrated in Fig. 2(a). Nodes in the same level are independent and can be computed in parallel. Due to the definition of level (the maximum path length from any source node to each node), the levelization generally has a common feature that the number of nodes in a level becomes fewer with level increasing. In fact, the levelization results of both the EGraph and ETree look like a funnel, as shown in Figs. 3(b) and 3(c).

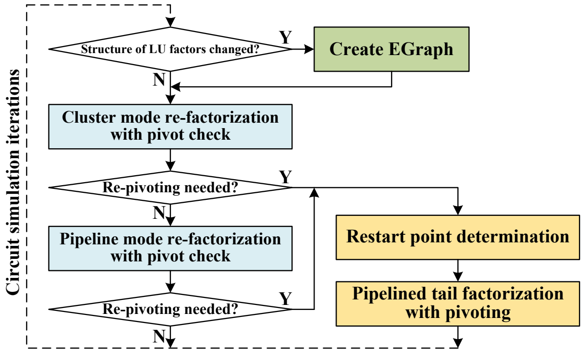

Circuit simulation iterations
Structure of LU factors changed?
Y
Create EGraph
N
Cluster mode re-factorization with pivot check
Re-pivoting needed?
Y
N
Pipeline mode re-factorization with pivot check
Restart point determination
Re-pivoting needed?
Y
Pipelined tail factorization with pivoting
N

Figure 4: Overall flow of proposed parallel LU factorization algorithm.

Like NICSLU [2, 5], we also use a cluster mode and a pipeline mode to parallelize rows with different dependencies, as illustrated in Figs. 2(b) and 2(c). A threshold can be defined to distinguish levels with many nodes and levels with few nodes in the "guessed" EGraph. In this work, we use #threads×α (α is an empirical parameter and we use 2.0) as the threshold. We find the first level that has less than #threads×α nodes as the dividing level. Levels before and after the dividing level are parallelized using the cluster mode and pipeline mode, respectively. Algorithms 2 and 3 show the cluster mode and pipeline mode re-factorization algorithms (with pivot check), respectively.

4.2.1 *Cluster Mode.*

In the cluster mode re-factorization with pivot check (Algorithm 2), the EGraph is processed level by level (line 2). For each level, nodes are evenly assigned to threads. Since nodes in the same level are independent, all threads can compute the assigned rows in parallel (line 3). For a specific row, numerical update (lines 5-7) is performed based on the structure of the LU factors obtained in the previous factorization. After that, pivot check (lines 8-10) is performed. If the diagonal element is smaller than the maximum element in the current row of **U** times the pivoting threshold ε, the pivot violates the pivoting rule and the corresponding thread exits. To tell the other threads, a shared variable *interrupted* is set to 1 (line 9). Those threads which do not find unsatisfactory pivots will complete the assigned rows in the current level. A barrier (line 14) is needed to synchronize all threads in each level. After the barrier, if *interrupted* is 1, it means that at least one unsatisfactory pivot has been found in the current level and all threads have to exit. If the cluster mode finishes without any unsatisfactory pivot found, the pipeline mode re-factorization algorithm with pivot check will be invoked to compute the remaining rows.

4.2.2 *Pipeline Mode.*

The pipeline mode re-factorization with pivot check (Algorithm 3) explores finer-grained parallelism between dependent rows, as the levels which belong to the pipeline mode have much fewer nodes and they do not fit the cluster mode well. First, all nodes which belong to the pipeline mode are put into a sequence level by level. They are assigned to threads on-the-fly in a dynamic

```
interrupted = 0; // shared variable
for $L = 1 : L_{Cluster}$ do
    for threads in parallel do
        for row $i$ assigned to this thread do
            $\mathbf{x} = \mathbf{A}(i, :)$;
            for $j = 1 : i - 1$ where $\mathbf{L}(i, j)$ is nonzero do
                $\mathbf{x}(j+1:N) = \mathbf{x}(j+1:N) - \mathbf{x}(j) \times \mathbf{U}(j, j+1:N)$;
            if $|\mathbf{x}(i)| < \varepsilon \times \max_{i+1 \le k \le N} \{|\mathbf{x}(k)|\}$ then
                interrupted = 1;
                exit thread ;
            $\mathbf{L}(i, 1:i) = \mathbf{x}(1:i)$;
            $\mathbf{U}(i, i:N) = \mathbf{x}(i:N)/\mathbf{x}(i)$;
            $finish[i] = 1$;
    barrier;
    if interrupted then
        exit thread;
```

Algorithm 2: Cluster mode re-factorization with pivot check.

```
1  interrupted = 0; // shared variable
2  for threads in parallel do
3      for row $i$ assigned to this thread do
4          $\mathbf{x} = \mathbf{A}(i, :)$;
5          for $j = 1 : i - 1$ where $\mathbf{L}(i, j)$ is nonzero do
6              while $!finish[j]$ do
7                  if interrupted then
8                      exit thread;
9              $\mathbf{x}(j+1:N) = \mathbf{x}(j+1:N) - \mathbf{x}(j) \times \mathbf{U}(j, j+1:N)$;
10         if $|\mathbf{x}(i)| < \varepsilon \times \max_{i+1 \le k \le N} \{|\mathbf{x}(k)|\}$ then
11             interrupted = 1;
12             exit thread ;
13         $\mathbf{L}(i, 1:i) = \mathbf{x}(1:i)$;
14         $\mathbf{U}(i, i:N) = \mathbf{x}(i:N)/\mathbf{x}(i)$;
15         $finish[i] = 1$;
```

Algorithm 3: Pipeline mode re-factorization with pivot check.

way. An integer `max_busy` always points to the maximum row index that is being computed in the sequence. Once a thread needs to get a new row to compute, it atomically increases `max_busy` by 1 and fetches the result. The pointed value of `max_busy` in the sequence is the fetched row index. This task assignment method is completely dynamic with negligible overhead of atomic operations. All threads compute fetched rows in parallel in an asynchronous way. For a row, numerical update (lines 4-9) is performed. Before using a dependent row to update the current row, we need to wait for it to finish (line 6). Pivot check (lines 10-12) is performed after numerical update. Here we also set *interrupted* to 1 if an unsatisfactory is found. It should be emphasized that when waiting for a dependent row to finish (lines 6-8), we must also check *interrupted*. Without checking *interrupted*, deadlock will occur if a thread has exited and another thread is waiting for it. As can be seen, the computations of dependent rows can overlap with point-to-point waiting operations.

4.3 Restart Point Determination

Once an unsatisfactory pivot is found in either cluster mode or pipeline mode of parallel re-factorization with pivot check, re-pivoting is needed for that row, and the remaining rows must also be factorized with pivoting, as the structure of them will be changed. More importantly, some already finished rows may also need to be recomputed with pivoting. The reason is explained as follows. The EGraph for scheduling parallel re-factorization with pivot check is "guessed" based on the structure of the LU factors of the previous factorization. Due to re-pivoting and the change of the LU factors, the dependencies described by the "guessed" EGraph become partially incorrect. Hence, we must determine which rows need to be recomputed.

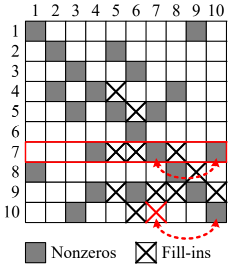


Figure 5: Example of re-pivoting on row 7: columns 7 and 10 are exchanged.

```
$\mathcal{P} = \emptyset$;
for unfinished node $i$ in EGraph do
    $Q = \emptyset$;
    Perform breadth-first search in ETree from $i$, in which visited nodes are
      skipped, and put visiting sequence into $Q$;
    Mark nodes in $Q$ visited;
    $\mathcal{P} = \mathcal{P} \cup Q$;
```

Algorithm 4: Determining the nodes that need to be computed with pivoting.

We still use the matrix example of Fig. 3(a) to illustrate why this could happen. Based on the "guessed" EGraph of Fig. 3(c), we assume that the first two levels are finished and row 7 needs to be re-pivoted. Assume that columns 7 and 10 are exchanged due to the re-pivoting of row 7, as illustrated in Fig. 5. Such a row exchange leads to an additional fill-in at $\mathbf{L}(10, 7)$, which leads to an additional dependency of $7 \longrightarrow 10$, which does not exist in the "guessed" EGraph. When scheduled by the "guessed" EGraph, row 10 is finished before row 7. However, once re-pivoting happens on row 7 like this example, row 10 becomes dependent with row 7 and row 10 must be recomputed.

Since the dependencies implied in the ETree are always correct for any pivoting order, the ETree can be used to determine which rows need to be recomputed. For the above case, it can easily be derived that rows 7, 8, 9 and 10 will be computed with pivoting, according to the ETree shown in Fig. 3(b). Algorithm 4 shows the method for determining the nodes that need to be computed with pivoting. We traverse unfinished nodes in the "guessed" EGraph, and for each unfinished node, all of its successors in the ETree need to be computed with pivoting. These nodes are put into set Q. The union ($\mathcal{P}$) of Qs of all unfinished nodes in the EGraph contains the nodes that will be computed with pivoting.

4.4 Pipelined Tail Factorization with Pivoting

Considering that the ETree is tall and narrow, we just compute the remaining matrix (including the rows which need to be recomputed

```
1  for threads in parallel do
2      for row $i$ assigned to this thread do
3          $\mathcal{U} = \emptyset$;
4          $\mathbf{x} = \mathbf{A}(i, :)$;
           // pre-factorization
5          while predecessors of $i$ not all finished do
6              $\mathcal{F} = \emptyset$;
7              Symbolic prediction, in which unfinished predecessors are
                 skipped, and put detected dependent rows that are not in $\mathcal{U}$
                 into $\mathcal{F}$;
8              for $j \in \mathcal{F}$ do
9                  $\mathbf{x}(j+1:N) = \mathbf{x}(j+1:N) - \mathbf{x}(j) \times \mathbf{U}(j, j+1:N)$;
10             $\mathcal{U} = \mathcal{U} \cup \mathcal{F}$;
           // post-factorization
11         Symbolic prediction: find structures of $\mathbf{L}(i, 1:i)$ and $\mathbf{U}(i, i:N)$;
12         for $j = 1 : i-1$ where $\mathbf{L}(i, j)$ is nonzero and $j \notin \mathcal{U}$ do
13             $\mathbf{x}(j+1:N) = \mathbf{x}(j+1:N) - \mathbf{x}(j) \times \mathbf{U}(j, j+1:N)$;
14         if $|\mathbf{x}(i)| < \varepsilon \times \max_{i+1 \le k \le N} \{|\mathbf{x}(k)|\}$ then
15             Exchange $\mathbf{x}(i)$ with $\mathbf{x}(\arg\max_{i+1 \le k \le N} \{|\mathbf{x}(k)|\})$;
16         $\mathbf{L}(i, 1:i) = \mathbf{x}(1:i)$;
17         $\mathbf{U}(i, i:N) = \mathbf{x}(i:N)/\mathbf{x}(i)$;
18         $finish[i] = 1$;
```

Algorithm 5: Pipelined tail factorization with pivoting.

with pivoting) using the pipeline mode without employing the cluster mode, as there may be very few rows that can be computed by the cluster mode. The pipeline mode factorization with pivoting is more complicated. Like the pipeline mode re-factorization with pivot check, we first put all remaining rows into a sequence. As introduced in Section 4.2.2, the same dynamic scheduling method by utilizing atomic operations is used for assigning rows to threads on-the-fly. Algorithm 5 shows the pipeline mode factorization with pivoting. The pipelined factorization flow of each thread can be divided into two parts, where lines 3-10 are pre-factorization and lines 11-18 are post-factorization.

In pre-factorization, the dependent rows of the current row (i) are not all finished, but we can use finished dependent rows to update row i first, while those rows which are needed by row i but are unfinished are skipped. Based on this principle, in line 7, symbolic prediction for row i is performed by skipping unfinished predecessors, and the detected finished predecessors which are needed by row i are put into $\mathcal{F}$. Then we use the rows in $\mathcal{F}$ to partially update row i (lines 8-9). This is the key to explore parallelism between dependent rows. The set $\mathcal{U}$ holds all rows which have already been used by row i. In post-factorization, all dependent rows of i are finished, so a complete symbolic prediction (line 11) is performed to determine the structure of row i as well as the complete dependencies of row i. After that, those skipped rows in pre-factorization (i.e., the dependent rows which are not in $\mathcal{U}$) can now be used to update row i (lines 12-13). Pivoting (lines 14-15) is performed after numerical update is finished.

5 EXPERIMENTS

Based on the proposed parallel LU factorization algorithm, we build a complete sparse direct solver. All CPU-based solvers are run on a Linux workstation equipped with an Intel Xeon Gold 6130 CPU (16 cores, 2.1GHz). Sixty-six circuit matrices (dimensions from 1,020 to 5,558,326) from SuiteSparse Matrix Collection [9] are used as the benchmarks. We compare the factorization performance among our solver, NICSLU [2, 5], KLU [11], and Intel MKL PARDISO [22]. KLU and NICSLU are two sparse solvers specially designed for circuit simulation. The reordering results of NICSLU are also used for our solver and KLU (producing the same structure of the LU factors), and the three solvers all use a pivoting threshold of 0.001. PARDISO is a general-purpose sparse direct solver and generally performs the best among a series of popular sparse solvers in non-SPICE applications [15].

We also compare our solver with two most recent GPU-based sparse solvers for circuit simulation, SFLU [23] and RLA [16], though our algorithm is purely CPU based. They are both not publicly available, so we compare the factorization performance of our solver with the performance numbers extracted from their publications (results of 8 and 17 circuit matrices are available for SFLU and RLA, respectively).

5.1 Accuracy of Solutions

Fig. 6 shows the accuracy of the solutions of our solver, KLU, and PARDISO. Here the accuracy is evaluated by the L2-norm of the residual, i.e., $||\mathbf{Ax} - \mathbf{b}||_2$. NICSLU almost obtains the same accuracy as KLU so the accuracy of NICSLU is not shown in Fig. 6 to make the figure clear. For 7 matrices (ASIC_320k, ASIC_320ks, ASIC_680k, ASIC_680ks, ckt11752_dc_1, Hamrle2, and rajat18), our solver produces much more accurate solutions than PARDISO, while for only 1 matrix (fpga_dcop_01), PARDISO produces much more accurate solutions than our solver. The geometric mean of the accuracy improvement ratios of the 66 benchmarks is 1.19× and 3.56×, respectively, compared with KLU and PARDISO. This implies that our solver generally achieves similar (slightly better) accuracy as KLU, and higher accuracy than PARDISO. The proposed pivoting-reuse technique does not affect the solution accuracy. The reason of obtaining higher accuracy than PARDISO is mainly due to the pivoting strategy of PARDISO — it only selects pivots within dense diagonal sub-blocks which shrinks the pivoting range.

5.2 Best-Case Performance

Here we evaluate the best-case performance and scalability of the proposed factorization algorithm. The base case means that the previous pivot choices can all meet the threshold-based pivoting rule. In this case, no re-pivoting is needed, so that the entire matrix is completed by re-factorization with pivot check which is scheduled by the “guessed” EGraph.

Fig. 7 compares the factorization time when using 16 threads except for KLU which is a sequential solver. In the 66 benchmarks, our solver is the fastest for 61 cases. KLU is generally the slowest as it is purely sequential. NICSLU is generally faster than PARDISO for small to medium matrices but slower than PARDISO for large matrices. On the contrary, our solver keeps a consistent trend that it is faster than PARDISO for almost all matrices.

Fig. 8 shows the average speedups of our solver compared with KLU (sequential), NICSLU, and PARDISO. As KLU is sequential, our solver achieves higher speedup when using more threads. When compared with parallel NICSLU, our solver shows much better scalability, as the speedup becomes higher with more parallelism. For sequential factorization, our solver is on average 1.21× faster than NICSLU. When our solver and NICSLU both use 16 threads,

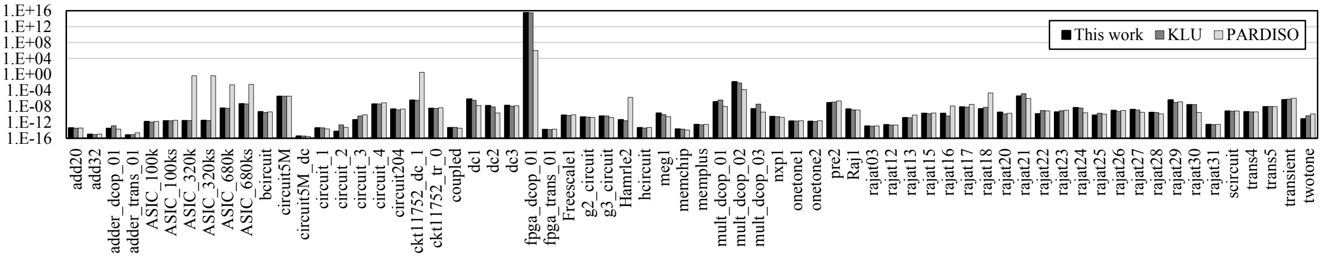


Figure 6: Residual of solutions.

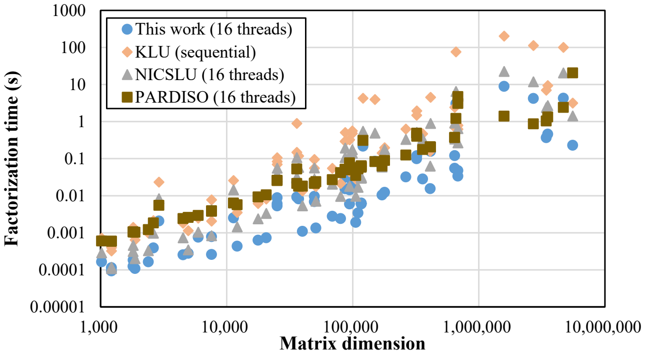


Figure 7: Performance comparison.

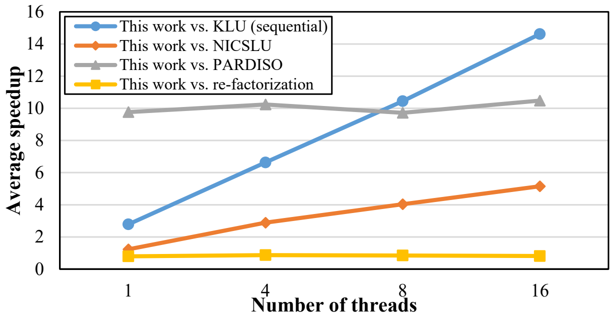
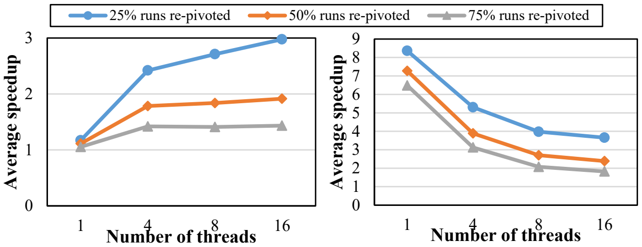


Figure 8: Average speedups versus KLU, NICSLU, PARDISO, and re-factorization, in the best case.

our solver achieves on average 5.15× faster than NICSLU. This is mainly because NICSLU uses the ETree to schedule fast parallel factorization. On the contrary, we use the "guessed" EGraph before re-pivoting is needed. When compared with PARDISO, the speedup keeps almost consistent, indicating that our solver has similar scalability as PARDISO. Our solver is consistently about 10× faster than PARDISO when both use 1 to 16 threads.

In Fig. 8, we also compare the proposed parallel factorization algorithm with parallel re-factorization without pivoting, where the latter is obtained by removing all symbolic- and pivoting-related operations from the proposed factorization algorithm. The proposed factorization algorithm is slightly slower than re-factorization due to the symbolic- and pivoting-related operations, but the speedups under different numbers of threads stabilizes at about 0.8, implying that the proposed algorithm in the best case has similar scalability to re-factorization without pivoting.

If parallel factorization is directly compared with sequential factorization, the average speedups are 2.44×, 3.87× and 5.46×, respectively, for 4-, 8- and 16-threaded parallelism.



(a) This work vs. NICSLU

(b) This work vs. PARDISO

Figure 9: Average speedups versus NICSLU and PARDISO in general case.

5.3 General-Case Performance

To evaluate the general case in benchmark tests, for each matrix, we run 100 times and randomly select 25%, 50% and 75% runs to interrupt re-factorization with pivot check. If a run is selected to be interrupted, the interruption row is randomly selected from $[1, N]$. The average factorization time of 100 runs is evaluated as the general-case performance and compared with the performance of NICSLU and PARDISO, as shown in Fig. 9.

A common trend is the decrease of the speedups with the increase of the number of runs that incur re-pivoting. This is easy to understand since once re-pivoting happens, the factorization of the remaining matrix is scheduled by the ETree which contains much stronger dependencies. When compared with NICSLU, the scalability of our factorization algorithm still becomes better with parallelism increasing. This is because NICSLU always uses the ETree to schedule parallel factorization but our solver uses the EGraph (without re-pivoting) or a combination of the EGraph and ETree (with re-pivoting needed). Even if 75% runs need re-pivoting, our solver is about 1.4× faster than NICSLU, when both use 4, 8 and 16 threads. When compared with PARDISO, the scalability becomes lower with parallelism increasing. PARDISO has better scalability than our solver with re-pivoting needed because PARDISO uses a static dependency graph to schedule parallel factorization, which is similar to the EGraph. However, the penalties of PARDISO are the shrunken pivoting range and the reduced solution accuracy, as compared in Fig. 6. Even if 75% runs need re-pivoting, our solver is 3.13×, 2.05× and 1.83× faster than PARDISO, for 4-, 8- and 16-threaded parallelism, respectively.

5.4 Comparisons with GPU-based Solvers

Tables 1 and 2 show the factorization performance comparisons with SFLU [23] and RLA [16], respectively, which are two most recent GPU-based sparse solvers for circuit simulation in literature. Note that the GPU-based solvers only implement re-factorization

Table 1: Performance comparison between our solver (using 16 threads) and SFLU [23].

Benchmark	This work (ms)	SFLU (ms)	Speedup
adder_dcop_42	0.124	2.28	18.39×
circuit_2	0.254	2.06	8.11×
fpga_dcop_13	0.114	0.35	3.07×
Hamrle2	0.756	2.42	3.20×
memplus	0.631	1.45	2.30×
rajat22	1.093	8.49	7.77×
rajat27	0.731	3.91	5.35×
mult_dcop_03	5.294	41.66	7.87×
Average			**7.01×**

Table 2: Performance comparison between our solver (using 16 threads) and RLA [16].

Benchmark	This work (ms)	RLA (ms)	Speedup
rajat12	0.108	3	27.78×
circuit_2	0.254	4	15.75×
memplus	0.631	6	9.51×
rajat27	0.731	8	10.94×
onetone2	9.148	62	6.78×
rajat15	8.196	58	7.08×
rajat26	1.341	13	9.69×
circuit_4	2.41	35	14.52×
rajat20	15.462	208	13.45×
ASIC_100ks	17.139	187	10.91×
hcircuit	1.902	12	6.31×
scircuit	10.43	46	4.41×
transient	12.29	390	31.73×
Raj1	31.97	902	28.21×
ASIC_320ks	99.041	216	2.18×
ASIC_680ks	33.728	184	5.46×
G3_circuit	8874.01	41385	4.66×
Average			**12.32×**

without pivoting. The results of our solver correspond to the best case, which has similar performance with re-factorization without pivoting. The SFLU and RLA results are directly extracted from their publications, which were evaluated on NVIDIA Titan RTX and NVIDIA GeForce RTX 2060, respectively. Our solver with 16-threaded parallelism is stably faster than both SFLU and RLA. The average speedups over SFLU and RLA are 7.01× and 12.32×, respectively, and the maximum speedups are up to 18.39× and 31.73×. Even for G3_circuit which is a fairly large benchmark (dimension 1,585,478), our solver is still 4.66× faster than GPU-based RLA.

5.5 Performance of SPICE DC Simulations

To evaluate the performance of our solver in real SPICE simulations, we integrate the proposed solver into an in-house parallel SPICE simulator and perform DC simulations using 16 threads. The same operation is also conducted for NICSLU and PARDISO (also using 16 threads) and we compare their DC simulation performance. Four real netlists based on the 45nm BSIM4 MOSFET model are simulated. Table 3 compares the total time of the entire DC simulation. The simulator with out solver runs the fastest. Note that besides the sparse solver, there are other steps (e.g., MOSFET model evaluation) in a SPICE simulation process, so speedups of the DC simulation time are lower than those of pure sparse solvers.

Table 3: Comparison on SPICE DC simulation performance (solvers use 16 threads).

#Nodes[a]	Dim.[b]	NICSLU		PARDISO		This work		
		Time (s)	#Iter.	Time (s)	#Iter.	Time (s)	#Iter.	#Re-piv.[c]
7719	37810	5.46	252	9.098	252	4.48	252	4
5090	25346	3.467	236	6.518	241	3.011	236	2
26564	46820	6.872	236	11.24	245	5.986	236	5
11705	12088	2.448	89	1.136	89	0.8961	89	3

[a] Number of nodes in the circuit, which is reported by the simulator.

[b] Dimension of created matrix.

[c] Number of iterations in which re-pivoting happens.

The number of NR iterations is also shown in Table 3. Our solver and NICSLU generate the identical number of NR iterations, which implies that they generate the same solution accuracy. However, PARDISO needs more NR iterations to get convergence for 2 cases, which explains that the pivoting strategy of PARDISO may lead to higher errors. As expected, for most iterations, our solver computes the entire matrix without re-pivoting needed, which is the primary starting point of this work. Hence, the overall DC simulation performance is very close to the best case, and the general cases evaluated in Section 5.3 are actually conservative. There are indeed some iterations in which re-pivoting happens. This explains the necessity of using the proposed algorithm to replace pure re-factorization without pivoting, as the latter will generate inaccurate solutions in those iterations that need re-pivoting.

6 CONCLUSION

Conventionally, in many practical applications, sparse solvers are usually made as general black boxes. However, if the special features of the matrices involved in the targeted application are taken into account, the solver design may be different and can be more dedicated. In circuit simulation, by utilizing the slow change feature of the matrix's values, one may "guess" the exact dependency graph based on the structural and pivoting information from the previous iteration, to schedule highly-scalable LU re-factorization with pivot check. Once the factorization is interrupted due to small pivots, pivoting is performed for the remaining matrix in a pipelined way. Both benchmark tests and real SPICE DC simulations have proven the superior performance and scalability of the proposed algorithm, compared with state-of-the-art CPU- and GPU-based solvers for circuit simulation.

REFERENCES

[1] X. Chen, L. Ren, Y. Wang, and H. Yang. 2015. GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling. *IEEE Transactions on Parallel and Distributed Systems* 26, 3 (2015), 786–795.

[2] X. Chen, Y. Wang, and H. Yang. 2013. NICSLU: An Adaptive Sparse Matrix Solver for Parallel Circuit Simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 2 (2013), 261–274.

[3] X. Chen, Y. Wang, and H. Yang. 2015. A fast parallel sparse solver for SPICE-based circuit simulators. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 205–210.

[4] X. Chen, Y. Wang, and H. Yang. 2017. *Parallel sparse direct solver for integrated circuit simulation*. Springer.

[5] X. Chen, W. Wu, Y. Wang, H. Yu, and H. Yang. 2011. An EScheduler-Based Data Dependence Analysis and Task Scheduling for Parallel Circuit Simulation. *IEEE Transactions on Circuits and Systems II: Express Briefs* 58, 10 (2011), 702–706.

[6] Yousu Chen, Kurt Glaesemann, Mark Rice, and Zhenyu Huang. 2015. Integrated State Estimation and Contingency Analysis Software Implementation using High Performance Computing Techniques. *IFAC-PapersOnLine* 48, 30 (2015), 227–232.

[7] Yousu Chen, Mark Rice, Kurt Glaesemann, and Zhenyu Huang. 2015. Sub-second state estimation implementation and its evaluation with real data. In *2015 IEEE Power Energy Society General Meeting*. 1–5.

[8] R. Daniels, H. Von Sosen, and H. Elhak. 2010. *Accelerating Analog Simulation with HSPICE Precision Parallel Technology*. Technical Report. Synopsys Corporation.

[9] T. A. Davis. [n. d.]. SuiteSparse Matrix Collection. https://sparse.tamu.edu/

[10] T. A. Davis. 2004. Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.* 30, 2 (June 2004), 196–199.

[11] T. A. Davis and E. P. Natarajan. 2010. Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems. *ACM Trans. Math. Softw.* 37, 3 (Sept. 2010), 36:1–36:17.

[12] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar. 2016. A survey of direct methods for sparse linear systems. *Acta Numerica* 25 (2016), 383–566.

[13] J. W. Demmel, J. R. Gilbert, and X. S. Li. 1999. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination. *SIAM J. Matrix Anal. Appl.* 20, 4 (July 1999), 915–952.

[14] J. R. Gilbert and T. Peierls. 1988. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Statist. Comput.* 9, 5 (1988), 862–874.

[15] Nicholas I. M. Gould, Jennifer A. Scott, and Yifan Hu. 2007. A Numerical Evaluation of Sparse Direct Solvers for the Solution of Large Sparse Symmetric Linear Systems of Equations. *ACM Trans. Math. Softw.* 33, 2 (jun 2007), 10–es.

[16] Wai-Kong Lee and Ramachandra Achar. 2021. GPU-Accelerated Adaptive PCBSO Mode-Based Hybrid RLA for Sparse LU Factorization in Circuit Simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40, 11 (2021), 2320–2330.

[17] Xiaoye S. Li and James W. Demmel. 2003. SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Math. Softw.* 29, 2 (jun 2003), 110–140.

[18] J. W. H. Liu. 1990. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.* 11, 1 (1990), 134–172.

[19] Cleve B. Moler. 1967. Iterative Refinement in Floating Point. *J. ACM* 14, 2 (apr 1967), 316–321.

[20] Shaoyi Peng and Sheldon X.-D. Tan. 2020. GLU3.0: Fast GPU-based Parallel Sparse LU Factorization for Circuit Simulation. *IEEE Design Test* 37, 3 (2020), 78–90.

[21] Lukas Razik, Lennart Schumacher, Antonello Monti, Adrien Guironnet, and Gautier Bureau. 2019. A comparative analysis of LU decomposition methods for power system simulations. In *2019 IEEE Milan PowerTech*. 1–6.

[22] O. Schenk, K. Gärtner, W. Fichtner, and A. Stricker. 2001. PARDISO: A High-Performance Serial and Parallel Sparse Linear Solver in Semiconductor Device Simulation. *Future Gener. Comput. Syst.* 18, 1 (Sept. 2001), 69–78.

[23] Jianqi Zhao, Yao Wen, Yuchen Luo, Zhou Jin, Weifeng Liu, and Zhenya Zhou. 2021. SFLU: Synchronization-Free Sparse LU Factorization for Fast Circuit Simulation on GPUs. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 37–42.