ISLU: Indexing-Efficient Sparse LU Factorization for Circuit Simulation on GPUs (Invited Paper)

Dan Niu
School of Automation, Southeast University
Nanjing, 210096, China
danniu1@163.com

Yiyang Tao*
School of Integrated Circuits, Southeast University
Nanjing, 210096, China
220226351@seu.edu.cn

Zhou Jin*
SSSLab, Dept. of CST, China University of Petroleum-Beijing
Beijing, 102249, China
jinzhou@cup.edu.cn

Yichao Dong
School of Automation, Southeast University
Nanjing, 210096, China
230238503@seu.edu.cn

Chao Wang
School of Automation, Southeast University
Nanjing, 210096, China
220232025@seu.edu.cn

Changyin Sun
School of Artificial Intelligence, Anhui University
Hefei, 230601, China
cysun@seu.edu.cn

ABSTRACT

Sparse LU factorization is a vital technique in solving circuit linear equations, However, irregular data access patterns contribute to unsatisfactory computational efficiency and excessive memory usage. Conventional LU factorization methods generally involve two approaches: either they utilize space-intensive dense matrices for direct index-to-data mapping, or they inefficiently scour through indices to locate the positions of updated data elements. To resolve these challenges, we propose the **I**ndexing-Efficient **S**parse **LU** factorization (***ISLU***) in this work. A novel indexing-efficient member union is put forwarded to achieve efficient retrieval of indices within compressed formats, thereby significantly enhancing the LU decomposition efficiency. Furthermore, to expedite the establishment of indexing-efficient member union, we design, for the first time, parallel creating member union strategy for GPU platforms, which remarkably reduces the time overhead associated with constructing the proposed structures. Extensive experimental comparisons on 49 benchmark matrices and real SPICE transient simulations demonstrate that the performance enhancements by our proposed ISLU method are substantial, outperforming various excellent GPU and CPU solvers including commercial solvers.

ACM Reference Format:
Dan Niu, Yiyang Tao, Zhou Jin, Yichao Dong, Chao Wang, and Changyin Sun. 2024. ISLU: Indexing-Efficient Sparse LU Factorization for Circuit Simulation on GPUs (Invited Paper). In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '24), October 27–31, 2024, New York, NY, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3676536.3695410

This work was supported by National Key R & D Program of China (No. 2021YFB0300600), National Natural Science Foundation of China (No. 62374031, 62204265, 62234010), Natural Science Foundation of Jiangsu Province under Grant BK20240173, the State Key Laboratory of Computer Architecture (ICT, CAS) (Grant No. CARCHA202115). (*Corresponding author: Zhou Jin and Yiyang Tao).

ICCAD '24, October 27–31, 2024, New Jersey, NJ, USA

ACM ISBN 979-8-4007-1077-3/24/10
https://doi.org/10.1145/3676536.3695410

1 INTRODUCTION

Sparse LU factorization has a profound impact on the performance of circuit simulation. In SPICE simulations (DC, TRAN, AC), the solutions of numerous sparse linear equations in Newton-Raphson (NR) iterations [**1**] have become the primary performance bottleneck for large-scale circuit simulations. For instance, in a post-layout simulations, solving linear equations can consume more than three-quarters of the total simulation time [**2**]. These equations, representing the nonlinear characteristics of circuit devices, culminate in solving $Ax = b$ [**3**]. Sparse LU factorization is widely utilized in SPICE simulators for the solution of linear equations. The LU factorzation procedure consists of three steps: symbolic analysis, numerical factorization and substitution. Many solvers based on CPU or GPU platforms have been developed around numerical factorization [**4-9**].

Numerous sparse LU factorization methods have been developed for CPUs, including SuperLU [**10**], PARDISO [**11**], KLU [**12**], and NICSLU [**13**]. Despite the advancements made by many existing sparse LU decomposition methods, they still fall short of satisfaction due to the inefficiency of parallelism. For instance, *i)* In SuperLU, obtaining large supernodes in very sparse matrices is challenging, significantly reducing the parallelism in multiplication. *ii)* KLU does not achieve parallelization on CPUs. *iii)* Despite efforts to utilize CPU multi-thread, its limited capacity falls short in markedly increasing time for very large matrix decomposition, indicating the urgent need for improved parallel strategies.

GPU-enabled algorithms, including GLU and its variants [**14-16**], SFLU [**17**], GPU-enabled NICSLU [**18**] and etc [**19-26**], have been developed. However, there are still some issues: *i)* Increased resources entail greater storage needs. Effective GPU memory management is overlooked despite its criticality. *ii)* Performing factorization within compressed format often encounters difficulties in accessing the necessary indices, consequently leading to degradation in efficiency.

Unlike general-purpose solvers, the unique characteristics of circuit simulations can be exploited to optimize the performance of the factorization. Sparse matrices keep a consistent pattern of non-zero elements across iterations, even the numerical values change.

Previous circuit solvers, such as NICSLU and KLU, have taken advantage of this characteristic to offer two kinds of factorization: factorization with pivoting and factorization without pivoting. The former mode utilizes an Elimination Tree (Etree) to depict dependencies among columns and employs it to facilitate parallelization. However, the Etree evaluates any possible pivot order and tends to overestimate dependence. Factorization with pivoting exhibits subpar performance. The latter mode reutilizes the pivot order generated by the last factorization with pivoting. This reuse conserves the structure of the LU factors, thereby bypassing the symbolic operations. By constructing an elimination graph (Egraph) based on the LU structure analysis, the factorization without pivoting is guided to realize superior performance compared to factorization with pivoting. This insight inspires us to artificially construct reusable structures for further optimizing the performance of LU factorization.

To enhance the numerous LU factorization efficiency during iterative processes of various circuit simulations, an indexing-efficient LU factorization algorithm named ISLU is proposed. The key contributions are as follows:

- A parallel creating member union algorithm is proposed for the first time. It facilitates the pre-computation of required indices, achieving a substantial improvement in retrieval speed.
- A novel indexing-efficient member union is designed to enhance the efficiency of index retrieval. The efficiency of irregular access patterns in LU factorization is significantly enhanced by using our member union.
- Extensive comparison experiments with various excellent CPU and GPU parallel solvers on the public benchmark circuits and real SPICE TRAN simulations validate the high-efficiency of proposed ISLU method.

2 BACKGROUND

2.1 Sparse left-looking factorization on GPUs

The sparse left-looking factorization [**27**] has been widely adopted in solvers for circuit simulation. In the context of GPU implementations, the widely-used algorithm is pivot-free, as illustrated in Algorithm 1.

The Algorithm 1 reuses the LU structure generated by previous factorization with pivoting. It progresses columnn-by-column, transforming matrix data into its lower (L) and upper (U) triangular components. Here we introduce matrix A that mirrors the non-zero structure of LU factors but has been not factorized. $A(i, j)$ denotes the non-zero element located at the i-th row and j-th column of matrix A. The matrices A, L, and U are all stored in a compressed format. N denotes the dimension of the matrix. x is a vector of length N, and utilized to store the intermediate structural variables for the current column. L and U signify the upper and lower part of matrix A. Each column's factorization is handled by an individual parallel unit (line 1). In each column, the non-zero elements of the k-th column of matrix A are initially scattered into the vector x (line 2-4). Employing a dense matrix x for storing these values facilitates direct indexing via j during subsequent Multiply-Accumulate (MAC) operations. The algorithm sequentially traverses non-zero elements

Algorithm 1: Sparse left-looking factorization.

```
1  for k in 1:N in parallel do
2      for i in 1:n where A(i,k) ≠ 0 in parallel do
3          x(i) = A(i,k) ;
4      end
5      Synchronize threads
6      for i in 1:k-1 where U(i,k) ≠ 0 do
7          for j in i+1:N where L(j,i) ≠ 0 in parallel do
8              x(j) = x(j) − x(i) ∗ L(j,i)
9              // irregular selection of index j
10         end
11         Synchronize threads
12     end
13     for i in k:n where L(i,k) ≠ 0 in parallel do
14         L(i,k) = x(i)/x(k)
15     end
16     for i in 1:k where U(i,k) ≠ 0 in parallel do
17         U(i,k) = x(i)
18     end
19 end
   Output: Lower matrix (L) and upper matrix (U).
```

above the diagonal within the row, leveraging parallel thread execution within the column to perform a series of MAC operations (line 6-11). Given the sparsity of circuit matrices, **the selection of index j in the sequence of MAC operations is typically irregular. This non-contiguous indexing significantly impacts the overall efficiency of LU factorization**. The introduction of index i and the synchronization of threads serve to ensure computation safety. The lower triangular elements of the dense matrix x, divided by the diagonal elements, are gathered into the L factor (line 12-14). The upper triangular portion of x is gathered into the U factor (line 15-17).

The algorithm capitalizes on the ample parallel processing capabilities of GPU to reduce the computational expense of LU factorization. Nonetheless, it confronts serious challenges. Due to irregular access pattern, utilizing vector x as a compute-intensive intermediary diminishes both data storage efficiency and computational speed. On one hand, the introduction of x necessitates additional "scatter" and "gather" operations and incurs extra data loading time. On the other hand, the substantial spatial requirement of x, exacerbated by a growing number of allocated blocks, follows a dimensionally-dependent trend in GPU memory consumption. This may result in either decomposition failure or redundant data reloading.

2.2 Motivation

During the Newton-Raphson iterations, since the nonzero structure of circuit matrices remains constant, the symbolic analysis needs to be executed only once. the performance of circuit simulations is significantly shaped by the extensive numerical factorization and substitution. The repetitive nature of these computations presents an opportunity to exploit reusable matrix structural information.

Leveraging reusable structural information has already yielded significant improvements in circuit simulation solvers. Reusing the LU structure provided by pivoted factorization, without pivoting variant bypasses complex procedures such as partial pivoting and pruning, thereby achieving higher parallelism. Studies on 66 circuit matrices reveal that Etree are 77.4 times deeper and 10.5 times shorter than Egraph [**28**], suggesting a significantly stricter sequential requirement for factorization with pivoting. The adoption of reuse strategies markedly enhances LU factorization performance, underscoring the benefits of factorization without pivoting enabled by such optimizations.

To enhance the efficiency of factorization without pivoting, we introduce for the first time the concept of an index-efficient member union. Similar to Egraph, it serves as a container constructed by analyzing the non-zero pattern of LU factors to improve factorization throughput. Given that this container is solely tied to the non-zero structure of the LU matrix, it enables reuse across hundreds of iterations. **Through the indexing-efficient member union, the irregular access patterns are captured in a contiguous manner, permitting efficient execution of LU factorization for hundreds of iterations.**.

3 RELATED WORK

The academic landscape hosts numerous popular solvers, including SuperLU [**10**], PARDISO [**11**], MUMPS [**29**], UMFPACK [**30**], which primarily gear towards general-purpose applications rather than circuit simulation. Matrices encountered in circuit simulations tend to exhibit less regular structure, hindering substantial performance enhancements through conventional methods such as supernodal or multifrontal approaches. A few solvers, KLU [**12**], NICSLU [**13**], are tailored to leverage matrix sparsity and optimize for circuit simulation workflows.

Significant strides have been made in developing GPU solvers for circuit simulations, which particularly focus on parallel algorithms and optimization strategies for factorization without pivoting. Ren et al. [**31**] proposed a levelset approach for parallelization. References [**13-15**] used Right-looking algorithm in GLU to decompose circuit matrices. Lee et al. [**26**] proposed a series of detailed optimizations for factorization on GPUs. Zhao et al. [**17**] proposed a synchronization-free approach at SFLU to avoid the overhead of starting many GPU kernels. Nonetheless, the methodologies have not significantly mitigated the issue of inefficient factorization.

4 PROPOSED ALGORITHM

Our framework targets the enhancement of storage and computational performance for LU decomposition executed on GPUs. We introduce ***parallel creating member union algorithm***, tailored specifically for GPU. This algorithm harnesses the fine-grained parallelism inherent in creating member union framework, thereby dramatically reducing the time expenditure associated with proposed data structures. To address the additional spatial and temporal overheads induced by irregular access patterns, we introduce the ***indexing-efficient member union*** to assist in LU factorization. This method enables computations to proceed within a compact format, thereby enhancing the efficiency of the factorization process. The core of our framework resides in the indexing-efficient member union. Parallel creating member union algorithm is strategically devised to minimize the temporal complexities involved in establishing proposed structures. This work prioritizes the practicality and efficiency of LU decomposition over various excellent CPU and GPU solvers.

4.1 Parallel Creating Member Union Algorithm

4.1.1 Troubles with Irregular Access. Sparse circuit matrices are typically stored in a compressed format. Storing circuit matrices in a compressed format, while space-efficient, presents challenges for computation due to the complexity in accessing nonzero. As illustrated in the lower half of Figure 1, the arrays ap, ai, and ax serve as an example of storing a sparse matrix in compressed column (CSC) format. Here, ap denotes the array holding pointers to the start of each column's non-zero elements, ai represents the array containing the row indices of the non-zero elements, and ax is the array storing the actual non-zero values. $ax[ap[k]+j]$ signifies the value of the j-th non-zero element located within the k-th column in the compressed format. When decomposing a column of data in compressed format, it is customary to first scatter the column into a vector x of size N. This allows for a direct correspondence between the data and their coordinates, thereby facilitating irregular access. This approach leads to increased GPU memory consumption as the dimension grows. Alternatively, it might employ a search algorithm to ascertain the requisite indices prior to conducting any MAC operations. This process is shown in Algorithm 2, which describes the process of performing LU decomposition within a matrix stored in CSC format. The relative position of index j within the compressed $A(:,j)$ is searched (line 5) and represented as $index$. Utilizing the $ap[k]$and $index$ facilitates the identification of the specific positions of the non-zero entries (line 6) within the compressed format. Due to the concurrent execution of search operations on index j and MAC operations, the overall efficiency of factorization is reduced.

4.1.2 Parallel Creating Member Union Strategy. The member union serves as a container optimized for managing irregular access pattern. Contrary to vector x that maps data values directly to their coordinates, it is to construct a mapping between index j and the locations of updated non-zero elements within compressed format. This process is described in Algorithm 3. The member union is comprised of three constituent members: $Vheader$, $Vlen$, and $Vdata$. The $Vheader$ functions as column pointer, giving the index of the pointer in the $Vlen$ that points to the first sequence of MAC operation in column k. $Vlen$ serves as an index pointer, giving the index of the element in the $Vdata$ that points to the i-th sequence of MAC operation in column k. $Vdata$ functions as the index container that stores each relative position of index j in $A(:,k)$ in column k. Algorithm 3 begins with the initialization of the $Vheader$, $Vlen$, $Vdata$ (line 1), and subsequently utilizes a pointer from the $Vheader$ to content within $Vlen$ (line 3), which in turn directs another pointer towards $Vdata$ (line 5). The relative position of index j in j-th column will be searched by navigating through non-zero elements within the j-th column (line 7). This enables $Vdata$ to employ pointer to deposit retrieved indices precisely at their intended destination (line 8). This algorithm has great parallelism on GPU. It can perform up to $32 \times (warp) \times (block)$ search algorithm simultaneously. The reason why so many index searches

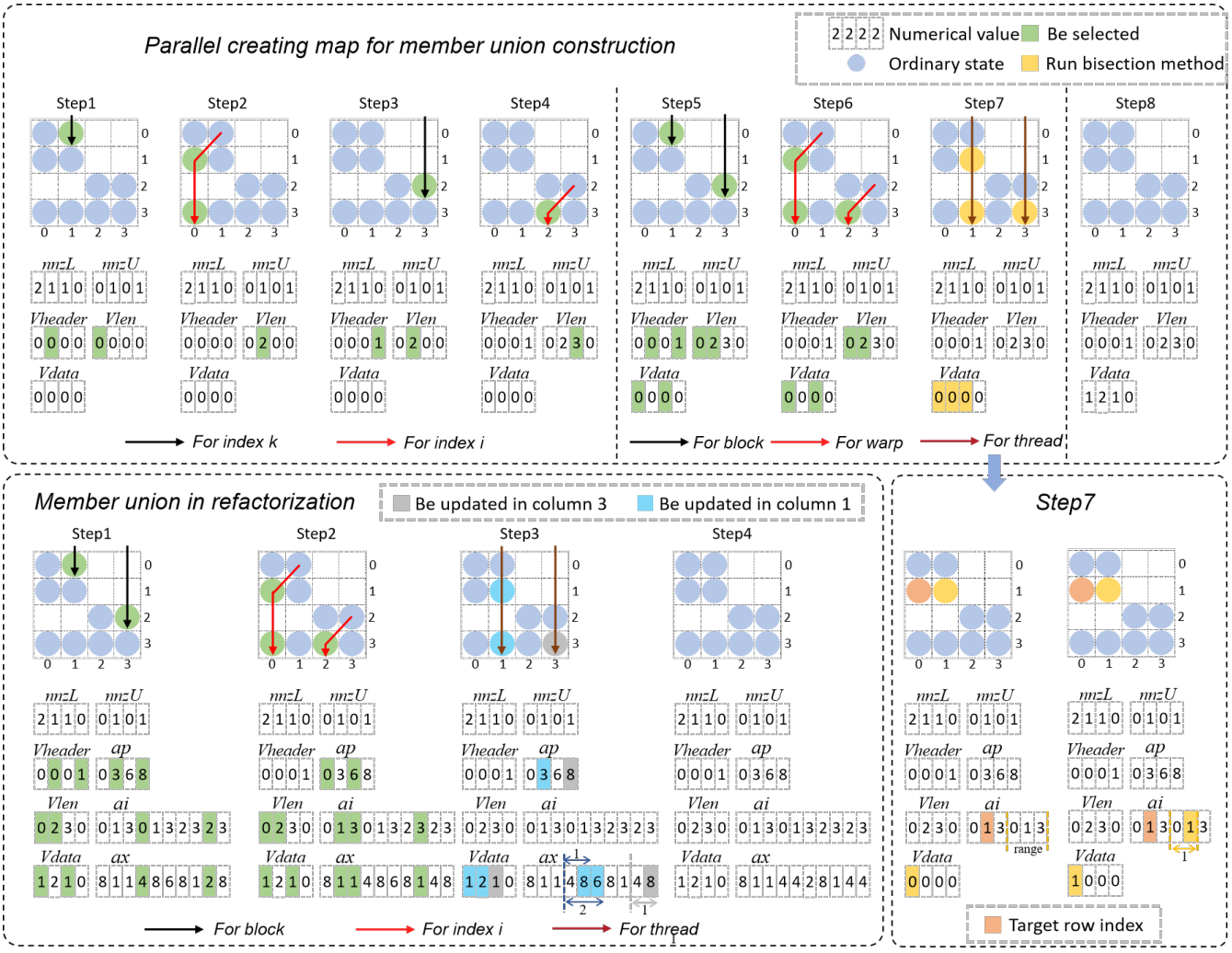

Figure 1: The diagram of establishing member union and utilizing member union for factorization, the eight steps of constructing member union and the four steps of using member union in MAC operation are respectively plotted.

can be performed at the same time is that, although in the process of numerical decomposition, the values of non-zero elements are updated in order but the index of each non-zero element needs not to be updated in order. In circuit simulations, characterized by unchanging non-zero entry patterns, member union akin to Egraph can be repeatedly utilized across iterations. The details can be shown in step 1 to step 8 in the upper part of Figure 1.

4.1.3 Member Union Construction Details. The top half of Figure 1 demonstrates an example of building a member union. Steps 1 to 4 involve initializing *Vheader* and *Vlen*. Steps 5 through 7 match line 2 to 11 in Algorithm 3. By Step 8, we have the member union loaded with the mapping relations. *nnzL* and *nnzU* reflect the number of non-zero elements in the lower and upper halves of each column in the matrix, excluding the diagonal elements. The arrows represent the pathways by respective parallel computing units. Different colored circles represent the varying states of non-zero elements, which are illustrated in the diagram. In the first step,

Algorithm 2: Traditional CSC MAC.

```
1  for k in 1:N in parallel do
2      for i in 1:k-1 where U(i,k) ≠ 0 do
3          for j in i+1:N where L(j,i) ≠ 0 in parallel do
4              index = the relative position of index j in A(:,k)
5              // compute the index for MAC operation.
                 ax[ap[k] + index] −= A(i,k) ∗ L(j,i)
6              // ax[ap[k] + index] denotes A(j,k).
7          end
8          Synchronize all threads
9      end
10 end
```

Algorithm 3: Parallel Creating Member Union.

```
1  Initialize and allocate memory for Vheader, Vlen and Vdata.
2  for k in 1:N in parallel do
3      fetch the pointer from Vheader.
4      for i in 1:k-1 where U(i,k) ≠ 0 in parallel do
5          fetch the pointer from Vlen.
6          for j in i+1:N where L(j,i) ≠ 0 in parallel do
7              index= the relative position of index j in A(:,k)
8              Use pointer to deposit index into Vdata.
9          end
10     end
11 end
```

Algorithm 4: Indexing-Efficient Refactorization.

```
1  for k in 1:N in parallel do
2      fetch the pointer from Vheader.
3      for i in 1:k-1 where U(i,k) ≠ 0 do
4          fetch the pointer from Vlen.
5          for j in i+1:N where L(j,i) ≠ 0 in parallel do
6              Use pointer to fetch index from Vdata.
                 ax[ap[k] + index] −= A(i,k) * L(j,i)
7              // ax[ap[k] + index] denotes A(j,k).
8          end
9      end
10     Compute column k for L.
11 end
```

$Vheader$ records the sequence number of the first non-zero element in each column that appears in the upper triangular matrix (U). This number represents the total number of non-zero elements in U being selected before executing that column, and it is placed in the corresponding column index in $Vheader$. In the second step, under the first step, $Vlen$ records the number of non-zero elements in the lower triangular matrix within the sub-column. This number is then accumulated with the previously recorded values in $Vlen$ and pushed into it. These two steps are repeated in subsequent processes to populate $Vheader$ and $Vlen$ (steps 3 and 4). In step 5, each block is assigned to a column, with $Vheader$ converting the column index pointing to $Vlen$, which in turn points to indices in $Vdata$. In step 6, each block's warp is allocated to the non-zero elements (nnz) of the upper triangle (U) of that column, directing them to the non-zero elements on the left that update this column. In step 7, the threads of each warp are assigned to the non-zero elements of the column that perform the MAC operations. Each thread searches for the relative position of the row index of the non-zero elment in column i with respect to the row index in current column k. Here, a binary search is employed to locate the target index. In step 8, we achieve a complete member union.

4.2 Member Union In Refactorization

4.2.1 Indexing-Efficient Refactorization algorithm. Following the member union, we proceed to outline the indexing-efficient refactorization algorithm as shown in Algorithm 4, which illustrates the incorporation of member union to assist in performing LU decomposition within compressed matrix. It first extracts the pointer from the $Vheader$ and $Vlen$ (line 2-4). Employing the $ap[k]$ and $index$ enables a series of MAC operations to be conducted within compressed format (line 7). Compared to Algorithm 2, the member union directly offers the indexes for MAC operation, eliminating the need to compute these indexes separately and enhancing computational efficiency. In comparison to Algorithm 1, this novel strategy presents three primary benefits: *i)* It diminishes surplus data transmission overhead. Since the LU decomposition operation is performed directly within the compressed format, the data transfer costs associated with gather and scatter operations are eliminated. *ii)* **It mitigates the irregularity of data selection by confining the scope of indexing value.** When the data of a column is processed within compressed format, the range over which data selection spans is reduced, thereby improving the regularity of data access. *iii)* It consumes reduced GPU memory. Member union records the pattern of accessing value rather than storing values, which results in diminished footprint on GPU memory.

4.2.2 Indexing-Efficient Refactorization Details. Here we introduce the method of using member union to assist LU decomposition, which is described in Figure 1. In the first step, each block is directed to each column. Here, we identify the locations of non-zero elements in the upper triangular of factorized column. In the second step, the selection of non-zero elements in the upper triangle of each column is executed in a top-down order. In the third step, all threads within a block perform MAC operations for that column. Selected non-zero elements from the lower triangle of the left sub-columns are used for the current column's decomposition. Since the content of $Vdata$ records the relative positions of non-zero elements in ai and ax of the column for MAC operations, these operations can be carried out in the CSC format.

5 EXPERIMENT

5.1 Experiment Setup

We tested 49 circuit matrices from sparse matrices collection [**32**], and several netlists with dimensions spanning from 10^2 to 10^8 to demonstrate the validity of our method. We implemented our Indexing-Efficient Sparse LU factorization (ISLU) as follows. ISLU decomposes data to double precision. ISLU implements the parallelization of LU decomposition by building an Egraph [**13**]. The ISLU component operates in a single-threaded manner. The index search, refactorization, and substitution processes are executed on the GPU. The tested matrix is decomposed only on a single GPU. If an instance occupies excessive space or its execution is overly prolonged, its outcome will not be depicted in the graph (Figure 3).

Our experiments conduct tests on ISLU, NVIDIA CuSolver [**33**], Intel MKL PARDISO [**11**], SFLU [**17**], and KLU [**12**], using a set of 49 circuit matrices. Commercial solver PARDISO is also utilized to test additional five netlists for SPICE simulation comparison, since it performs best among the popular sparse solvers [**34**]. The pivoting thresholds for our solver, as well as for KLU [**12**], CuSolver [**33**], and SFLU [**17**], are uniformly established at a value of 0.001.

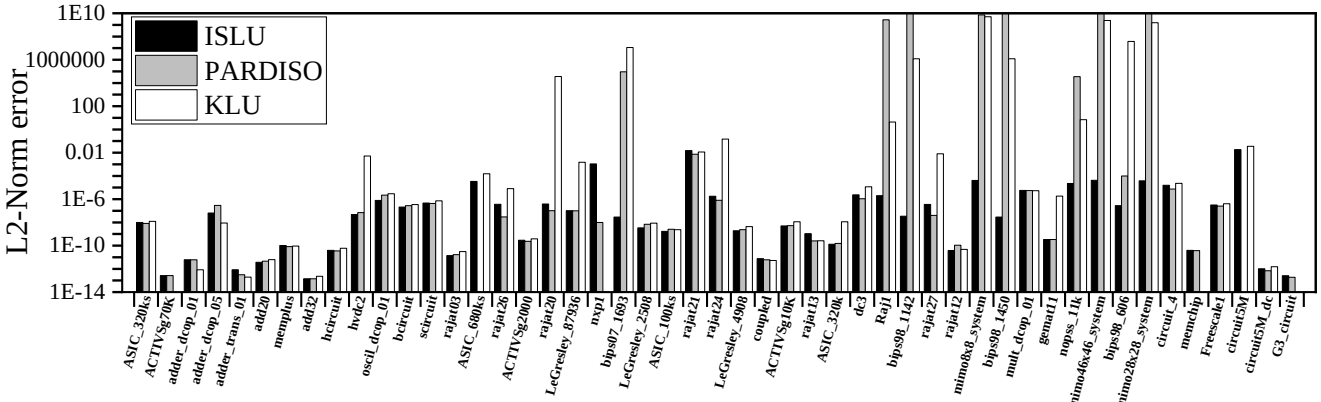

Figure 2: Residual of solutions over 49 public sparse circuit matrices with different methods.

The entire project is implemented using C++ and compiled with g++. It incorporates mixed compilation with CUDA 12.0 and nvcc to leverage the capabilities of GPU acceleration. And no BLAS libraries are utilized within our solver. Except for the SPICE TRAN simulation, all other experiments are conducted on an Intel 24-core server equipped with NVIDIA GeForce RTX 3090 GPU and Intel Xeon Gold 6336Y CPU.

5.2 Main Result

Our work entails a comprehensive performance comparison of ISLU against various GPU and CPU solvers, including commercial solvers, such as Intel MKL PARDISO [**11**] and NVIDIA CuSolver [**33**], specialized circuit simulation solvers, such as NICSLU (GPU) [**18**] and KLU [**12**], LLA [**26**], RLA [**35**], and SFLU [**17**], GLU 3.0 [**16**]. The speedups compared to open source PARDISO, KLU, CuSolver, and SFLU are conducted on a set of 49 public circuit matrices. Factorization time for NISCLU(GPU), GLU 3.0, LLA, and RLA are obtained from the respective references.

In terms of factorization speed, ISLU outperforms PARDISO (16 threads) by arithmetic mean **9.38x** and geometric mean **4.75x** speedups, outperforms KLU (sequential) by arithmetic mean 82.48x and geometric mean 5.37x, outperforms CuSolver (GPU) by arithmetic mean 31.82x and geometric mean 9.50x, outperforms SFLU (GPU) by arithmetic mean 7.79x and geometric mean 7.43x, outperforms LLA by arithmetic mean 2.35x and geometric mean 2.08x, outperforms RLA by arithmetic mean 7.65x and geometric mean 6.70x, and outperforms NICSLU (GPU) by arithmetic mean 5.15x and geometric mean 3.78x, and outperforms GLU3.0 by arithmetic mean 9.59x and geometric mean 8.47x speedups.

In the SPICE TRAN simulation experiments, we contrasted the total runtime of our solver with that of PARDISO. As shown in Table 3, aside from the factorization and substitution stages, the solver incurs minimal time expenditure on other processes (symbolic analysis, the creation of indexing-efficient member union, parallelism analysis and others). Across five circuit netlists, our solver exhibits a **4.73x** geometric mean speedup over PARDISO (12 threads).

5.3 Accuracy of Solutions

Figure 2 illustrates the solution accuracy of our solver compared to PARDISO and KLU. Here error is quantified via the L2 norm ($||\mathbf{Ax}\text{-}\mathbf{b}||_2$). Our solver demonstrates comparable performance to PARDISO and KLU on a multitude of circuit matrices (ASIC_320ks, add20, bcircuit and others), but with the added advantage of ensuring more consistent and reliable precision. As shown in examples such as LeGresley_1693, solvers like KLU and PARDISO cannot ensure that their solutions will have an error approximately equivalent to that of our solver. In the 49 benchmark matrices, PARDISO fails to solve 2 circuit matrices (ASIC_680ks, circuit5M), and KLU does not deal with 4 circuit matrices (ACTIVSg70K, nxp1, memchip, G3_circuit), while our solver passes all the cases. This signifies that our solver is capable of accurately and reliably solving a larger range of matrices compared to PARDISO and KLU. The reason of retaining accuracy is that our pivoting approach selects pivots within a column and adopts the pivot checking to improve accuracy. However, PARDISO opts for pivots within dense diagonal subblocks, which can confine the pivoting scope and potentially lead to erroneous pivot selections. KLU fails to adequately manage very small pivots.

5.4 ISLU vs CPU Solvers

5.4.1 Performance comparisons under different SPRs. Here we evaluate the performance of our proposed ISLU against two excellent CPU solvers, PARDISO (using 16 threads) [**11**] and KLU (using 1 thread) [**12**]. The speedup of our solver's factorization over KLU (sequential) and PARDISO (using 16 threads) on 49 benchmark circuit matrices is depicted in Figure 3. We will evaluate these circuit matrices with their respective sparsity ratio (*SPR*). *SPR* [**36**] is given by

$$SPR = \frac{FLOPs}{NNZ(U+L-I)} \tag{1}$$

When matrix exhibits greater sparsity, *SPR* tends to decrease.

In Figure 3, among **the 49 benchmark circuit matrices ranked from low SPR to high SPR**, PARDISO (using 16 threads) fails to solve 2 matrices, while KLU (sequential) does not complete the computation for 4 matrices. KLU is a serial solver, and it performs poorly on both large circuit matrices and slightly dense matrices. For slightly denser and larger instances such as ASIC_320ks and ASIC_100ks, KLU runs slower compared to both our solver and PARDISO (using 16 threads).

PARDISO (using 16 threads) is slower than our solver across nearly all matrices. Simultaneously, our solver outperforms PARDISO (using 16 threads) even on slightly dense matrices (ASIC_320ks

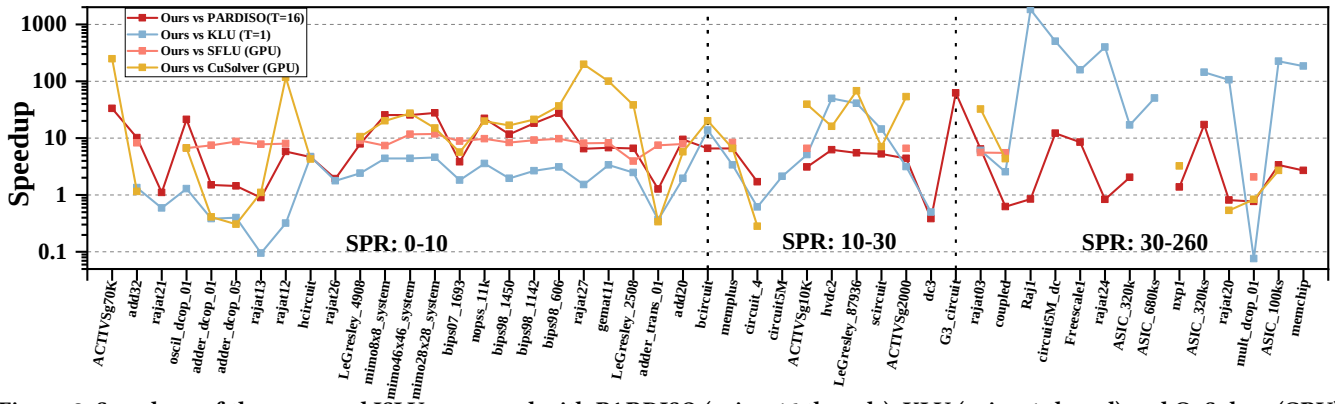

Figure 3: Speedups of the proposed ISLU compared with PARDISO (using 16 threads), KLU (using 1 thread) and CuSolver (GPU), SFLU (GPU) for the runtime of factorization.

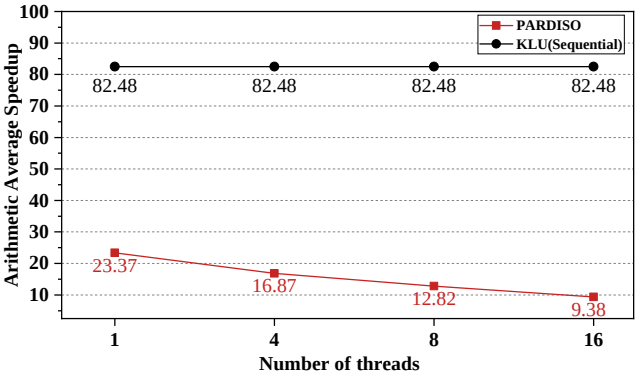

Figure 4: Arithmetic speedups over PARDISO (using 1 thread, 4 threads, 8 threads, 16 threads) and KLU (sequential).

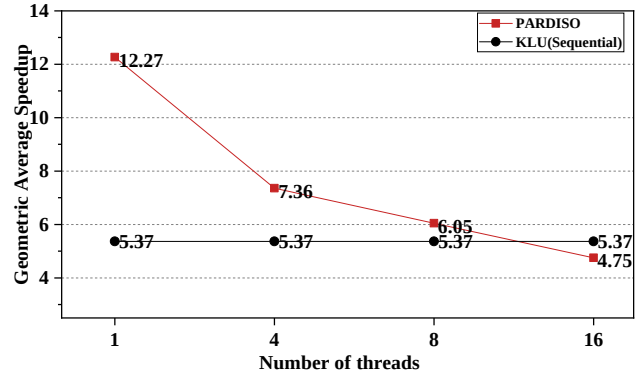

Figure 5: Geometric speedups over PARDISO (using 1 thread, 4 threads, 8 threads, 16 threads) and KLU (sequential).

and memchip). This superiority is largely attributed to the proposed indexing-efficient member union, which mitigates the irregularity inherent in sparse matrix operations and enhances data locality. This means that our solver is capable of accommodating a broader spectrum of matrix densities than PARDISO (using 16 threads).

Table 1: Performace comparisons on popular GPU solvers.

Matrix	ISLU (ms)	RLA [35]	LLA [26]	GLU 3.0 [16]
rajat12	0.91	3.00	1.37	2.23
circuit_2	0.88	4.00	1.43	4.14
memplus	0.59	6.00	1.77	6.67
rajat27	1.40	8.00	4.26	10.53
onetone2	6.04	62.00	11.61	60.96
rajat15	8.10	58.00	17.78	71.13
rajat26	3.42	13.00	6.17	32.36
circuit_4	18.29	35.00	13.82	68.94
rajat20	35.04	208.00	46.47	241.82
ASIC_100ks	14.24	187.00	49.83	215.49
hcircuit	2.71	12.00	8.13	46.99
scircuit	4.47	46.00	-	-
transient	62.84	390.00	60.01	-
Raj1	90.58	902.00	-	845.18
ASIC_320ks	17.62	216.00	61.18	216.51
ASIC_680ks	13.77	184.00	66.46	210.69
Ari Mean Speedup		**7.65**	**2.35**	**9.59**
Geo Mean Speedup		**6.70**	**2.08**	**8.47**

Table 2: Performace comparisons between our ISLU and NICSLU.

Matrix	N (x10^3)	ISLU (ms)	NICSLU (ms) [18]	Speedup
add32	5.0	0.20	0.40	2.00
hcircuit	105.7	2.71	6.50	2.39
add20	2.4	0.14	0.40	2.85
bcircuit	68.9	1.47	5.70	3.87
circuit_4	80.2	18.29	18.50	1.01
scircuit	171.0	4.47	58.00	12.97
rajat03	7.6	0.68	3.00	4.41
coupled	11.3	5.91	14.00	2.36
rajat15	37.3	8.10	41.70	5.14
rajat18	94.3	30.36	50.60	1.66
raj1	263.7	90.21	227.40	2.52
transient	178.9	62.84	114.40	1.82
rajat24	358.2	115.15	223.10	1.93
ASIC_680k	682.9	352.61	593.60	1.68
onetone2	36.1	6.11	44.70	7.31
ASIC_680ks	682.7	13.98	150.60	10.77
ASIC_320k	321.8	186.95	404.80	2.16
ASIC_100k	99.3	95.81	273.50	2.85
ASIC_320ks	321.7	17.77	168.10	9.45
ASIC_100ks	99.2	14.24	166.70	11.70
onetone1	36.1	10.31	161.00	15.61
twotone	120.8	145.35	983.8	6.76
Arithmetic Mean				**5.15**
Geometric Mean				**3.78**

Table 3: Performance comparisons between our ISLU and PARDISO (12 threads) on SPICE TRAN simulation.

Benchmark	ISLU (s)		Overall runtime (s)	PARDISO (12T) (s) [11]		Overall runtime (s)	Overall speedup
	fact+sub	other		fact+sub	other		
OSC	26.79	0.37	27.16	105.48	0.54	106.02	3.90
ACT	120.09	0.25	120.34	857.28	0.36	857.64	7.12
POW	35.21	0.41	35.62	117.62	3.91	121.60	3.41
DLL	55.72	0.44	56.16	444.45	0.97	445.52	7.39
TES	125.21	5.57	130.78	408.49	2.97	411.46	3.14
Arithmetic Mean							**5.10**
Geometric Mean							**4.73**

5.4.2 Performance comparisons under different threads. Moreover, the PARDISO using different threads (1, 4, 8, 16) are also tested and compared. The arithmetic mean and geometric mean speedups of our solver over KLU (sequential) and PARDISO with different threads on 49 circuit matrices are represented in Figure 4 and Figure 5, respectively. Our solver outperforms PARDISO with threat 1, 4, 8, 16 by the arithmetic mean speedups of 23.37x, 16.87x, 12.82x, and 9.38x, respectively. The high geometric mean speedups are also presented in Figure 5. It is clear that higher arithmetic mean and geometric mean speedups can be obtained if the PARDISO with less thread is selected and compared.

5.5 ISLU vs GPU Solvers

First, we conduct the performance comparisons with two excellent open-source GPU solvers CuSolver [**33**] and SFLU [**17**] on the 49 public circuit matrices. The test results are also presented in Figure 3 to save space. It can be seen that the proposed ISLU surpasses CuSolver and SFLU for nearly all the circuit cases. ISLU outperforms CuSolver by arithmetic mean 31.82x and geometric mean 9.50x speedups, also achieves arithmetic mean 7.79x and geometric mean 7.43x speedups for SFLU. In addition, to give more comprehensive evaluations, some preeminent but closed-source methods including LLA [**26**], and RLA [**35**], NICSLU (GPU) [**18**] are also compared. The comparison results are shown in Tables 1 and 2. The factorization time of those four methods are obtained from their respective publications and we test the proposed ISLU using the same matrices (different scales and different SPRs) in those papers. From the two tables, the performance enhancements offered by our proposed ISLU method are also substantial. ISLU outperforms the four excellent methods by up to 9.59x arithmetic mean speedup and 8.47x geometric mean speedups.

Table 4: SPICE TRAN in test benchmark Netlists.

Netlists	Dim.[a]	SPR	#R[b]	#C[b]	#B[b]	#M[b]	#Ours. Iter.[c]	#PAR. Iter.[c]
OSC	1.4E4	1.4E4	7.3E4	3.0E4	22	4.2E3	819	865
ACT	4.3E3	2.0E2	1.3E3	1.7E3	1.2E2	4.9E3	33097	32417
POW	1.3E5	9.2E4	1.8E4	9.1E2	15	4.7E5	99	99
DLL	3.7E4	2.7E4	1.1E5	2.6E5	0	6.4E3	950	982
TES	6.9E4	5.6E4	3.6E4	1.5E5	0	1.9E3	679	686

[a]Dimension of created matrix.
[b]R="Resistor", C="Capacitor", B="Bioplar", M="Mosfet".
[c]Number of iterations under our ISLU and PARDISO solvers.

5.6 Performance Comparisons on SPICE TRAN

To evaluate the performance of our solver in a real SPICE simulator environment, we integrate the proposed solver into a commercial SPICE simulator [**37**] and perform TRAN simulation. Both the proposed ISLU and PARDISO (12 threads) are conducted on the 12-core server with Intel Xeon Gold 6226 CPU and NVIDIA A800 GPU. Here we also attempt to incorporate the above-mentioned GPU solvers into SPICE simulator for comparisons, but either they are not open-source or lack the assurance of stable solution accuracy. To more effectively illustrate the disparities between our solver and PARDISO (12 threads), Table 3 records the factorization, back-substitution time and the runtime of other procedures (symbolic analysis, the creation of indexing-efficient member union, parallelism analysis and others) of each matrix generated from netlists. The netlist information is presented in Table 4. From Table 3, our solver also achieves faster TRAN simulation in SPICE, which outperforms PARDISO (12 threads) by **5.10x** arithmetic mean speedup and **4.73x** geometric mean speedup.

6 CONCLUSION

In this work, we investigate the potential for accelerating decomposition of circuit matrices on GPU platform through the proposed resource management strategies. The indexing-efficient member union is proposed, which enables the efficient extraction of indices. To mitigate the temporal overhead in the formation of the member Union, we design the parallel creating member union algorithm. The superiority of our proposed ISLU is validated through extensive comparisons with various excellent GPU and CPU solvers on benchmark circuits and SPICE transient simulation.

REFERENCES

[1] C. Ho, E. Ruehli and A. Brennan. 1975. The modified nodal approach to network analysis. *IEEE Transactions on circuits and systems* 22, 6, (1975), 504–509.

[2] R. Daniels, V. Sosen and H. Elhak. 2010. Accelerating analog simulation with HSPICE precision parallel technology. *Technical Report.* Synopsys Corporation, (2010), 1-4.

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

[4] Z. Jin, W. Li, Y. Bai, T. Wang, Y. Lu, W. Liu. 2024. Machine Learning and GPU Accelerated Sparse Linear Solvers for Transistor-Level Circuit Simulation: A Perspective Survey (Invited Paper). *29th ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, (2024), 1-6.

[5] X. Fu, B. Zhang, T. Wang, W. Li, Y. Lu, E. Yi, J. Zhao, X. Geng, F. Li, J. Zhang, Z. Jin, W. Liu. 2023. PanguLU: A Scalable Regular Two-Dimensional Block-Cyclic Sparse Direct Solver on Distributed Heterogeneous Systems. *36th International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, (2023), 1-15.

[6] T. Wang, W. Li, H. Pei, Y. Sun, Z. Jin, W. Liu. 2023. Accelerating Sparse LU Factorization with Density-Aware Adaptive Matrix Multiplication for Circuit Simulation. *60th ACM/IEEE Design Automation Conference (DAC)*, (2023), 1-6.

[7] Y. Chen, H. Pei, X. Dong, Z. Jin, C. Zhuo. 2022. Application of Deep Learning in Back-End Simulation: Challenges and Opportunities. *27th ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, (2022), 641-646.

[8] Y. Zhao, X. Yang, Y. Bai, L. Zeng, D. Niu, W. Liu, Z. Jin. 2024. CSP: Comprehensively-Sparsified Preconditioner for Efficient nonlinear Circuit Simulation. *43rd ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, (2024), 1-9.

[9] G. Feng, H. Wang, Z. Guo, M. Li, T. Zhao, Z. Jin, W. Jia, G. Tan. N. Sun. 2024. Accelerating Large-scale Sparse LU Factorization for RF Circuit Simulation. *30th International European Conferenee en Parallel and Distributed Computing (Euro-Par)*, (2024), 1-9.

[10] S. Li. 2005. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software (TOMS)* 31, 3, (2005), 302–325.

[11] O. Schenk, K. Gärtner, W. Fichtner and A.D. Stricker. 2001. PARDISO: a high-performance serial and parallel sparse linear solver in semiconductor device simulation. *Future Generation Computer Systems* 18, 1, (2001), 69–78.

[12] T. Davis, E. Palamadai Natarajan. 2010. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Transactions on Mathematical Software (TOMS)* 37, 3, (2010), 1–17.

[13] X. Chen, Y. Wang, H. Yang. 2013. NICSLU: An adaptive sparse matrix solver for parallel circuit simulation. *IEEE transactions on computer-aided design of integrated circuits and systems (TCAD)* 32, 2, (2013), 261–274.

[14] K. He, X.-D. Tan, H. Wang and G. Shi. 2015. GPU-accelerated parallel sparse LU factorization method for fast circuit analysis. *IEEE Transactions on Very Large Scale Integration Systems (VLSI)* 24, 3, (2015), 1140–1150.

[15] W. Lee, R. Achar, and S. Nakhla. 2018. Dynamic GPU parallel sparse LU factorization for fast circuit simulation. *IEEE Transactions on Very Large Scale Integration Systems (VLSI)* 26, 11, (2018), 2518-2529.

[16] S. Peng, and X-D. Tan. 2020. GLU3. 0: Fast GPU-based parallel sparse LU factorization for circuit simulation. *IEEE Design & Test* 37, 3, (2020), 78-90.

[17] J. Zhao, Y. Wen, Y. Luo, Z. Jin, W. Liu, and Z. Zhou. 2021. SFLU: Synchronization-Free Sparse LU Factorization for Fast Circuit Simulationon GPUs. In *58th ACM/EEE Design Automation Conference (DAC)*, (2021), 37-42.

[18] X. Chen, L. Ren, Y. Wang and H. Yang. 2014. GPU-accelerated sparse LU factorization for circuit simulation with performance modeling. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 26, 3, (2014), 786-795.

[19] S. Olaf, K. Gärtner, and W. Fichtner. 2000. Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors. *BIT Numerical Mathematics* 40, (2000), 158-176.

[20] R. Gnanavignesh , and U. Shenoy. Parallel sparse LU factorization of power flow Jacobian using GPU. *In TENCON 2019-2019 IEEE Region 10 Conference (TENCON)*, (2019), 1857-1862.

[21] X. Chen, D. Wang and H. Yang. 2013. Nonzero pattern analysis and memory access optimization in GPU-based sparse LU factorization for circuit simulation. *In Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*, (2013), 1-8.

[22] N. Galoppo, N. K. Govindaraju, M. Henson and D. Manocha. 2005. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. *In Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, (2005), 3-3.

[23] A. Gaihre, X. Li, H. Liu. 2021. Gsofa: Scalable sparse symbolic lu factorization on gpus. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 33, 4, (2021), 1015–1026.

[24] R. Gnanavignesh, U. Shenoy. 2019. Gpu-accelerated sparse lu factorization for power system simulation. In *IEEE PES Innovative Smart Grid Technologies Europe (ISGT-Europe)*, (2019), 1–5.

[25] Y. Xia, P.Jiang, G. Agrawal, R. Ramnath. 2023. End-to-End LU Factorization of Large Matrices on GPUs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, (2023), 288–300.

[26] W. Lee, R. Achar. 2022. Algorithmic Advancements and a Comparative Investigation of Left and Right Looking Sparse LU Factorization on GPU Platform for Circuit Simulation. In IEEE Access 10, (2022), 78993-79003.

[27] J. Gilbert, T. Peierls. 1988. Sparse partial pivoting in time proportional to arithmetic operations. In SIAM Journal on Scientific and Statistical Computing 9, 5, (1988), 862–874.

[28] X. Chen. Numerically-Stable and Highly-Scalable Parallel LU Factorization for Circuit Simulation. In *In Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, (2022), 1–9.

[29] P. Amestoy, I. Duff. 2000. MUMPS: a general purpose distributed memory sparse solver. In *International Workshop on Applied Parallel Computing*, (2000), 121–130.

[30] T. Davis. 2004. Algorithm 832: UMFPACK V4. 3—an unsymmetric-pattern multifrontal method. In ACM Transactions on Mathematical Software (TOMS) 30, 2, (2004), 196–199.

[31] L. Ren, X. Chen, Y. Wang, C. Zhang and H. Yang. Sparse LU factorization for parallel circuit simulation on GPU. 2012. *Proceedings of the 49th Annual Design Automation Conference*, (2012), 1125–1130.

[32] S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. A. Davis, M. Henderson, Y. Hu and R. Sandstrom. 2019. The suitesparse matrix collection website interface. *Journal of Open Source Software* 4, 35, (2019), 1244.

[33] CuSolver Library Documentation. Accessed: April. 2024. [Online]. Available: https://docs.nvidia.com/cuda/cusolver/index.html.

[34] 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 Transactions on Mathematical Software (TOMS)* 33, 2, (2007), 10-es.

[35] W. Lee and R. Achar. 2020. 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 (TCAD)* 40, 11, (2020), 2320-2330.

[36] X. Chen, L. Xia, Y. Wang and H. Yang. 2016. Sparsity-oriented sparse solver design for circuit simulation. *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, (2016), 1580-1585.

[37] C. Zhao, Z. Zhou and D. Wu. 2020. Empyrean ALPS-GT: GPU-accelerated Analog Circuit Simulation (Invited Talk). *In Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD)*, (2020), 1-3.