Machine Learning and GPU Accelerated Sparse Linear Solvers for Transistor-Level Circuit Simulation: A Perspective Survey (Invited Paper)

Zhou Jin, Wenhao Li, Yinuo Bai, Tengcheng Wang, Yicheng Lu and Weifeng Liu
Super Scientific Software Laboratory, China University of Petroleum-Beijing, China
Email: {jinzhou, weifeng.liu}@cup.edu.cn, {wenhao.li, tengcheng.wang}@student.cup.edu.cn,
yinuobai693@gmail.com, luyicheng_cup@163.com

Abstract— Sparse linear solvers play a crucial role in transistor-level circuit simulation, especially for large-scale post-layout circuit simulation when considering complex parasitic effects. As semiconductor technology advances rapidly, the increasing sizes of circuits result in sparse linear solvers that require extended execution times and additional memory resources. Consequently, high-performance sparse linear solvers emerge as pivotal tools to facilitate rapid circuit simulation and verification. However, circuit matrices frequently exhibit high sparsity and non-uniform distributions of nonzero elements, compounding the challenge of achieving efficient acceleration. Recently, the flourishing developments in machine learning technology and the continuous enhancement of hardware capabilities have presented new opportunities for accelerating sparse linear solvers. This paper provides a perspective review of these technological advancements, while also highlighting the challenges and future opportunities in this evolving landscape.

I. Introduction

One of the greatest challenges in integrated circuit (IC) design is repeated executions of computationally expensive SPICE (simulation program with IC emphasis) simulations, particularly when highly complex chip testing/verification is involved [30]. With increasing degrees of integration of modern integrated circuits, the reliability of a chip design is improved via a time-consuming verification process before tape-out. Such verification mainly ensures the circuit behavior and performance and its physical feasibility and robustness, where accurate transistor-level circuit simulation is involved. Moreover, many other stages of IC design and verification both in the front and back end, i.e., circuit optimization, reliability analysis, yield analysis, and signoff, also require repeated executions of an expensive SPICE simulation (due to the large scale of an IC design) [7].

Circuit simulation encompasses a variety of analysis types, such as DC analysis, transient analysis, small-signal AC analysis, and so on [17]. Although various analyses produce distinct sets of equations, these equations are ultimately transformed into a series of sparse linear systems to be solved through numerical discretization and linearization [22]. Consequently, solving the sparse linear systems occupies most of the simulation time. Especially in the back-end post-layout circuit simulation, where complex parasitic effects from the real world need to be considered, the linear solver further dominates the simulation time (often exceeding 90% of the total simulation time). Therefore, the acceleration of sparse linear solvers emerges as a paramount challenge in the field of circuit simulation.

Sparse direct solvers are the most commonly utilized approaches in circuit simulation, primarily due to their superiority in terms of precision, speed, and practical applicability. The direct method usually solves the linear system by triangular factorization and forward/backward substitution, and the most representative general software packages are MUMPS [2], UMFPACK [10], SuperLU_DIST [20, 31], PARDISO [27], etc. These packages are not specifically designed for circuit simulation problems. Circuit matrices are usually irregular and sparse, and it is not possible or quite difficult to form dense sub-matrices using any of these traditional methods to take advantage of the level-3 Basic Linear Algebra Subprograms (BLAS) to accelerate, and therefore the solution needs to be accelerated by incorporating circuit matrix features. Current solvers in the field of circuit simulation are KLU [13], NICSLU [4–6], FLU [32], GLU [16, 18, 23], etc. However, optimizing the performance of the sparse linear solver remains a challenge as the size of the circuit matrix continues to increase. Fortunately, the booming development of artificial intelligence (AI) and the increasing hardware computational power bring new possibilities to accelerate the linear solver. Various innovative algorithms have been proposed, including strategies that utilize machine learning techniques to address irregular sparsity distribution patterns (i.e., Density-aware LU) [3, 9, 28], as well as methods that embrace synchronization-free concepts to design GPU and heterogeneous distributed cluster acceleration mechanisms, thereby harnessing the substantial parallel computing capabilities (i.e., SFLU, PanguLU) [14, 33]. In addition, iterative solvers with better preconditioner are also investigated [8, 19, 29, 34].

In this paper, we present a perspective survey on recent progress of high-performance sparse linear solvers tailored for circuit simulation, especially enhanced with ML techniques and GPU acceleration. We also conduct several experiments on circuit matrices with various solvers and compare their performance on various platforms, including CPU, GPU and distributed clusters. We point out several challenges and opportunities to accelerate sparse linear solvers for circuit matrices. Finally, we demonstrate several future directions that worth to be explored.

II. Sparse Linear Solver in Circuit Simulation

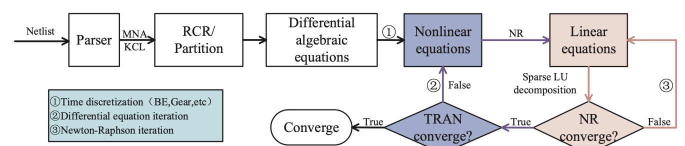


Fig. 1: Transient analysis flow in SPICE simulation [7].

Sparse linear solver dominates the transient analysis time. Transient analysis involves solving the time-domain response of a circuit, which represents the response of circuit components to a given excitation signal as it varies over time.

The task needs to solve a set of ordinary differential equations (ODE).

$$\boldsymbol{P}(\boldsymbol{x}(t), d\boldsymbol{x}(t)/dt, t) = 0 \tag{1}$$

where $\boldsymbol{x} = (\boldsymbol{v}, \boldsymbol{i})^T \in \mathbb{R}^m$, $m = N + M$, variable vector $\boldsymbol{v} \in \mathbb{R}^N$ denotes node voltage, and vector $\boldsymbol{i} \in \mathbb{R}^M$ represents internal branch current. Numerical integration algorithms, i.e., backward Euler as shown in equation (2), are used to discrete the ODE in the time domain.

$$\dot{\boldsymbol{x}}(t)|_{t=t_{n+1}} = (\boldsymbol{x}_{n+1} - \boldsymbol{x}_n)/h_{n+1} \tag{2}$$

It results in a set of nonlinear algebraic equations at each time point,

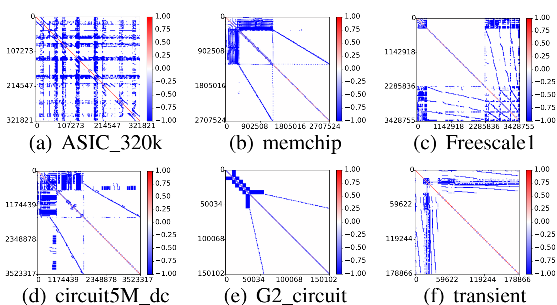
$$\boldsymbol{F}(\boldsymbol{x}) = 0 \tag{3}$$

where $\boldsymbol{F}(\cdot) : \mathbb{R}^m \to \mathbb{R}^m$. Ultimately, as shown in equation (4), the nonlinear system of equations is linearized by establishing the Newton-Raphson (NR) iterative method [24],

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k - \left[J(\boldsymbol{x}^k)\right]^{-1} F(\boldsymbol{x}^k) \tag{4}$$

where $\boldsymbol{J}(\boldsymbol{x}^k)$ is the Jacobian matrix and k is the NR iteration step. Therefore, the main task in transient analysis is to solve a series of sparse linear systems.

Circuit matrix is established by modified nodal analysis. The Jacobian matrix in Equation (4), which is also the coefficient matrix of the linear system, is typically derived from the circuit netlists using the Modified Nodal Analysis (MNA) method, and the number of nonzero elements (nnz) of the matrix then represent the number of components in the circuit. MNA is an approach that combines Kirchhoff's Current Law (KCL), Kirchhoff's Voltage Law (KVL), and component characteristic equations into matrix form, using node voltages and branch currents as variables. This method ensures a low matrix order while addressing independent source branches. Therefore, it is currently the main approach to construct circuit matrices for SPICE simulation.



(a) ASIC_320k (b) memchip (c) Freescale1 (d) circuit5M_dc (e) G2_circuit (f) transient

Fig. 2: The structure of the circuit sparse matrices [28].

Circuit matrix follows a non-uniform distribution of nnz. The matrix generated in circuit simulation is an extremely special class of sparse metrics with irregular nonzero elements distribution characteristics, as shown in Fig. 2. It can be further observed from TABLE I, the circuit matrices are typically highly sparse, with each row usually containing fewer than 10 nonzero elements, and do not guarantee symmetry. Moreover, a small number of nodes may be connected to a significant number of components, leading to a substantial number of nonzero elements in certain rows. These factors significantly affect the efficiency of the solving process.

TABLE I: Circuit sparse matrix properties [28].

Circuit Matrix	N	Entries per row				Symmetry
		max	min	average	variation	
ASIC_320k	321,821	203,800	1	8.2	502.95	100.00%
memchip	2,707,524	27	2	5.5	2.06	0.32%
Freescale1	3,428,755	27	1	5.5	2.07	7.67%
circuit5M_dc	3,523,317	27	1	10.7	1356.61	55.99%
G2_circuit	150,102	4	1	2.9	0.52	0.0005%
transient	178,866	60423	1	5.4	147.2	68.99%

III. Sparse Direct Solver

The most representative and universal of direct methods is LU factorization, which decomposes the matrix A of a system of linear equations $Ax = b$ into the product of a lower triangular matrix L and an upper triangular matrix U. When the matrix A is sparse, the process can be divided into 3 steps: 1) Pre-processing: Reorder the matrices to minimize the number of fill-in elements. 2) Symbolic factorization: Identify the locations of these fill-in elements. 3) Numeric factorization: Calculate the values of nonzeros. Fig. 3 shows the left-looking and right-looking methods for sparse LU factorization, where the left-looking method decomposes the matrix from left to right, one column at a time, and the right-looking method decomposes the matrix row by row and column by column along the diagonal.

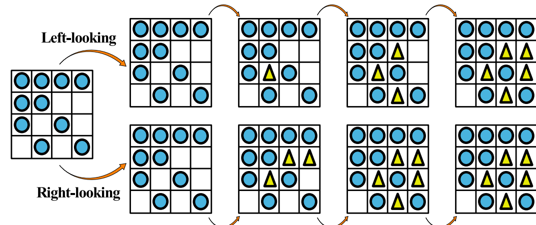


Fig. 3: The left-looking and right-looking methods for factorizing an example matrix of order four, where circles and triangles represent nonzeros and fill-ins, respectively.

For the circuit matrix, the distribution of nonzero elements established from the MNA is determined by the circuit topology. Since the circuit topology remains fixed throughout the entire simulation process, the pattern of nonzero elements in the matrix remains unchanged, with only the numerical values being updated in each iteration. Therefore, in circuit simulation, reordering in pre-processing and symbolic factorization typically need to be performed only once, while numeric factorization requires multiple times. This indicates that in circuit simulation, the most critical factor is not the time spent on pre-processing or symbolic factorization; rather, the primary concern lies in the speed of numeric factorization.

A. AI-based Acceleration

In the pre-processing phase, the key is to maintain numerical stability and reduce the padding of nonzero elements. The reordering method in the pre-processing stage greatly affects the amount of floating-point operations in the numeric factorization stage. However, finding the optimal reordering with minimum fill-ins is typically an NP-hard problem [11]. Existing methods are primarily heuristic-based, i.e., METIS, AMD and MMD for column permutation, MC64 and AWPM for row permutation, etc. It can be observed from Table II that different reordering methods produce completely different

TABLE II: Comparison between reordering methods for the matrix from circuit simulation. The test runs on AMD EPYC 7702 CPU 2.0 GHz with SuperLU_DIST 8.1.2.

Reordering method	METIS+ AWPM[1]	MMD+ AWPM[1]	MMD+ AWPM[2]	METIS+ MC64[1]	MMD+ MC64[1]	MMD+ MC64[2]
Reordered matrix						
Relative fill-ins	9.63	15.26	30.00	10.05	16.47	29.93
Reordering time (s)	52.06	51.04	70.15	3.44	3.14	19.71
Numeric time (s)	43.09	113.28	1694.32	49.18	153.66	1500.09

[1] Pre-processing for unsymmetric matrices: $A^T + A$
[2] Pre-processing for unsymmetric matrices: $A^T A$.

numbers of fill-ins and subsequent factorization time. Moreover, none of them is always optimal for all matrices, making it challenging to find a good alternative for different matrices.

To address this challenge, Cui et al. [9] first proposed an AI-based approach to choose the best reordering algorithm in power grid analysis, combined with support vector machine (SVM) and neural network (NN). Furthermore, to expand the application areas, Chen et al. [3] proposed a few-shot model based method for circuit simulation matrices. However, all existing AI-based strategies are generally based on supervised learning to choose the best alternative among existing methods and mainly considering the number of fill-ins as the criteria. On the one hand, the data in [3] show that the minimum number of fill-ins does not always imply the best factorization time. On the other hand, existing reordering methods do not necessarily always encompass the optimal reordering technique. Therefore, leveraging semi-supervised or unsupervised learning to generate specific optimal reordering methods for any matrix, rather than merely selecting from existing methods, is a highly promising direction in the future.

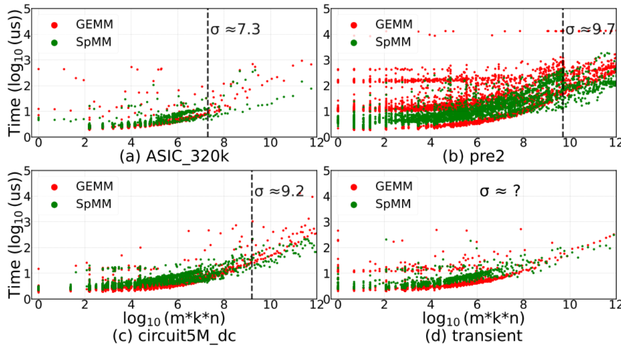


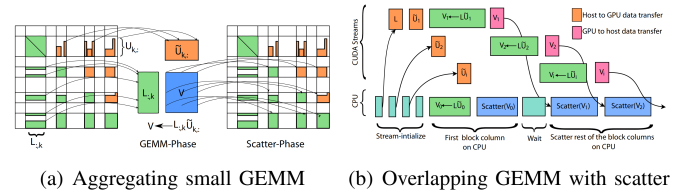
Fig. 4: A comparison of GEMM and SpMM on circuit matrices with different orders [28].

In the numeric factorization phase, the most crucial and time-consuming part is often matrix multiplication [28]. The main concepts of conventional methods, i.e., multifrontal or supernodal, is to form dense submatrix blocks, enabling the invocation of General Matrix Multiplication (GEMM) in Level-3 BLAS for enhanced computational performance. However, due to the specific property of the circuit matrix, it has been found that introducing sparse matrix multiplication may bring substantial acceleration potential. As shown in Fig. 4, for different matrices, dense (GEMM) and sparse (SpMM) multiplication show superior performance in some cases, respectively, and selecting the appropriate kernel for each matrix multiplication becomes a crucial issue. Therefore, in [28] a density-aware adaptive matrix multiplication equipped with random forest to optimize performance of the most time-consuming matrix multiplication kernel is proposed to accelerate the sparse LU factorization.

B. GPU Acceleration

In contrast to multi-core CPUs, GPUs possess greater acceleration potential due to features such as a large number of cores, large register files, and high memory bandwidth. However, efficiently accelerating sparse LU factorization on GPUs poses two major challenges: 1) designing specific algorithms to accelerate the computational kernel with the circuit matrix structure, and 2) utilizing the GPU to accelerate task scheduling when there are strong dependencies between tasks (often columns).

Accelerating computational kernels on GPUs. Sao et al. [26] developed a strategy based on SuperLU_DIST [21] to aggregate small dense BLAS operations into larger ones to fully utilize the computational power of GPUs (as shown in Fig. 5), and they also used the CUDA stream to hide the memory copy latency. Li's team and others [1] also proposed a systematic way of addressing matrix computations on GPUs for batches containing a matrix of different sizes, and addressed a performance-critical component in a multifrontal sparse LU solver STRUMPACK [15]. To allow the sparse irregular matrix to be better accelerated on GPUs, Fu et al. [14] used a regular 2D blocking strategy in PanguLU and combined it with the decision tree to put part of the kernels on GPUs to achieve relative acceleration. However, it is still challenging to combine matrix features to achieve further speedup.



(a) Aggregating small GEMM (b) Overlapping GEMM with scatter

Fig. 5: GPU-accelerated methods in SuperLU_DIST[26].

Tasks scheduling on GPU. Some work choose to divide and schedule tasks based on level-set on GPUs. Chen et al. [5] based on G-P algorithms and combined the features of task-level and data-level parallelism on GPUs to accelerate sparse LU factorization, and developed the NICSLU software package, which is optimized for the work partition, the number of active thread groups and the memory access pattern. To further improve the parallelism of G-P algorithms on GPU platforms, Peng et al. [23] developed GLU 3.0 based on GLU 1.0 and 2.0, using a relaxed principle to find all required dependencies, plus some redundant one and developed three different modes of the GPU kernels that adapt to different stages in computing tasks. changes. However, all of the above methods require a high synchronization cost, and if GPUs can be reasonably utilized for resource scheduling, it will be possible to greatly release the arithmetic power and accelerate the sparse LU factorization. Therefore, Zhao et al. [33] proposed a synchronization-free sparse direct method solver called SFLU (shown in Fig. 6) on GPUs, where each thread block eliminate one column and running all thread blocks simultaneously to

TABLE III: An overview of surveyed sparse direct solver.

Solver	Left/Right looking	Blocking method	Kernel	Parallelism				
				Level	Method	Distributed	Multi-thread	GPU
MUMPS[1] [2]	Left	Multifrontal	Level-3 BLAS	Tree/Node	Level-set	✓	✓	-
UMFPACK[1] [10]	Left	Multifrontal	Level-3 BLAS	-	-	-	-	-
SuperLU_DIST[1] [20, 31]	Right	Supernode	Level-3 BLAS	Supernode	Level-set	✓	✓	✓
PARDISO[1] [27]	Left&Right	Supernode	Level-3 BLAS	Supernode	Pipeline	✓	✓	-
PanguLU[1] [14]	Right	Regular 2D block	Adapative sparse kernel	2D block	Synchronization-free	✓	✓	✓
KLU[2] [13]	Left	Block diagonal	-	-	-	-	-	-
NICSLU[2][4–6]	Left	Supernode	Level-3 BLAS	Supernode	Level-set	-	✓	✓
FLU[2] [32]	Left	Supernode	Level-3 BLAS	Supernode	Register Level	-	✓	-
GLU[2] [16, 18, 23]	Left&Right	-	-	Element	Level-set	-	-	✓
SFLU[2] [33]	Left&Right	-	-	Element	Synchronization-free	-	-	✓
Density-aware LU[2] [28]	Right	Supernode	Adaptive kernel	Supernode	Level-set	✓	✓	-

[1] Sparse direct solver for general matrices.
[2] Sparse direct solver for circuit matrices.

fully utilize the GPU resources by exchanging the dependency information stored in global memory with all thread blocks in the computation state or in the busy waiting state. The new method avoided global synchronizations between levels in the existing methods and increased the amount of parallelizable work for GPUs of a large mount of compute units. However, when the matrix size is small or there are strong dependencies between columns, existing GPU acceleration methods still face challenges to fully utilize the computational power of GPUs. Therefore, how to effectively combine computation and scheduling strategies to fully utilize the arithmetic power of GPUs remains a challenging problem.

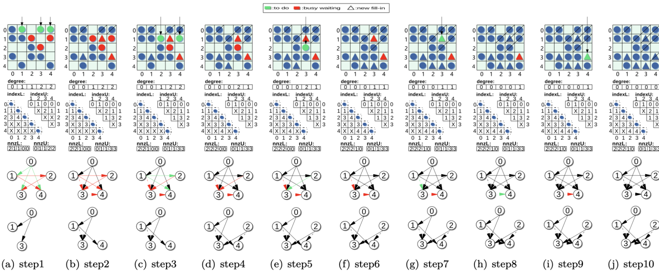


Fig. 6: The SFLU method used for factorizing an example matrix of order five [33].

TABLE III shows an overview of the direct solvers of the sparse surveyed. We conduct experiments with SuperLU_DIST [25], PARDISO [27], KLU [13], GLU [23], SFLU [33] and PanguLU [14] with gcc-9.3.0, OpenMPI-4.1.2 and cmake-3.23.1. Details of matrices [12] are shown in TABLE IV. The experiment platform is 4*NVIDIA A100 GPUs using CUDA 11.3.0 and driver 510.85.02, with 40 GB, B/W 1555GB/s, 2 * Intel Xeon 8180 CPUs (28 cores) and 512GB DDR4.

TABLE IV: The matrices tested.

matrix	n(A)	nnz(A)	matrix	n(A)	nnz(A)
add32	4.96e3	1.98e4	meg4	5.86e3	2.53e4
circuit_3	1.21e4	4.81e4	memplus	1.78e4	9.91e4
rajat27	1.06e4	9.74e4	ckt11752_dc_1	4.97e4	3.33e5
twotone	1.21e5	1.21e6	pre2	6.59e5	5.83e6
ASIC_680k	6.82e5	2.64e6	G3_circuit	1.59e6	7.66e6

We evaluate the performance and Fig. 7 shows the experimental results, where the performance of each solver is taken to be the optimal performance on this experimental platform. It can be seen that no solver is optimal for all matrices, and the superiority of GPU acceleration over CPU is not always guaranteed. Specifically, for smaller matrices (add32, meg4,

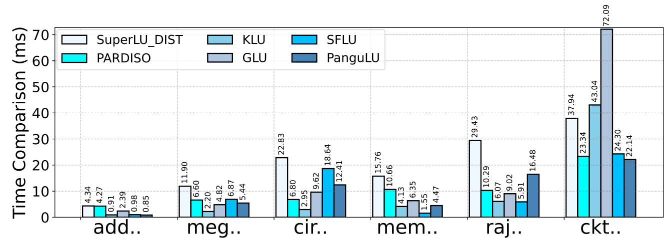


Fig. 7: A comparison of numeric factorization.

and circuit_3), serial solver KLU has the best performance, followed by SFLU, and the performance of GLU is generally balanced. The performance advantage of SFLU is gradually manifested when the size of the matrix is gradually enlarged (memplus and rajat27), indicating that after the matrix reaches a certain size, the use of GPU-based synchronization-free scheduling strategy can play a great advantage. However, the heterogeneous distributed solvers SuperLU_DIST and PanguLU do not play a better advantage until the matrix ckt11752_dc_1, where the advantage of SuperLU_DIST and PanguLU are gradually revealed, but the performance of KLU and GLU starts to decline instead, which is related to both the matrix size and the inter-column dependency. Additionally, there are always GPU solvers (GLU or SFLU or PanguLU) that outperform the PARDISO performance for matrices of different sizes, reflecting the greater potential of GPU acceleration compared to CPU multicore acceleration. The diversity of computing platforms and matrix features all affect performance, therefore, combining matrix features and taking full advantage of the computing platform to further optimize the LU factorization remains a great challenge.

C. *Heterogeneous Distributed Acceleration*

When confronted with large-scale matrices, the utilization of distributed heterogeneous acceleration emerges as a highly effective approach. In this regard, Amestoy et al. developed MUMPS [2], which uses asynchronous communication and dynamic task scheduling for acceleration in multifrontal methods, but MUMPS is currently only available in CPU. SuperLU performs heterogeneous distributed optimization; Sao et al. [25] combined GPU optimization with the 2D [31] mesh to further improve the algorithm by proposing a 3D mesh algorithm. These methods are generalized but not suitable for irregular circuit matrices, once the matrix has been reordered to determine the structure, the strong coupling relationship between columns also affects the scalability of these level-set based methods. Therefore, Fu et al. [14] proposed PanguLU, a

regular scalable block-cyclic sparse direct solver on distributed heterogeneous platforms. In PanguLU, a mapping approach was designed for load balancing, a variety of block-wise sparse BLAS methods were selected for higher GPU efficiency, and a synchronization-free communication strategy was developed to reduce the overall latency cost.

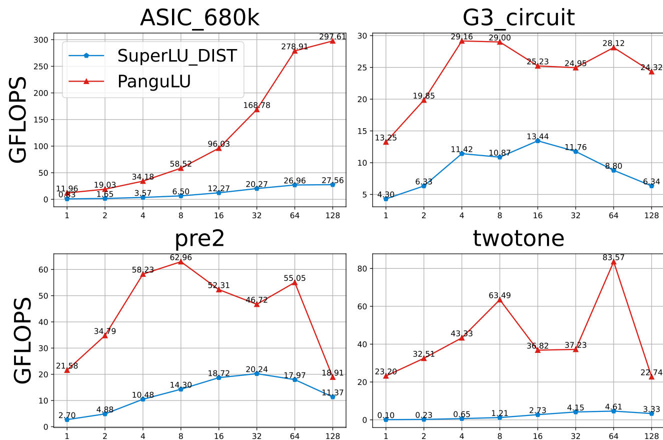


Fig. 8: Performance comparison between SuperLU_DIST and PanguLU on 128 A100 GPUs, where the GFLOPS is Giga Floating-point Operations Per Second.

We further evaluate the performance of SuperLU_DIST and PanguLU on a 32-node 128-GPU distributed clusters in numeric factorization using 1, 2, 4, 8, 16, 32, 64 and 128 A100 GPUs, the configuration information for each node are shown in Section III.*B*. Fig.8 shows the experimental results. Compared to SuperLU_DIST, PanguLU is superior in terms of GFLOPS and scalability, which also shows that the synchronization-free communication scheduling strategy and sparse kernels possess significant advantages when faced with circuit matrices. However, as the number of processes increases, the GFLOPS of SuperLU_DIST and PanguLU continue to increase on the lagest matrix ASIC_680k, but gradually decrease on G3_circuit, pre2, and twotone. This reduction is mainly due to the increase in communication costs, despite the faster computation by using more GPUs. It indicates that, while distributed methods have the potential for parallel acceleration, the resulting overhead is not negligible and is not suitable for all matrices. Therefore, to exploit large-scale supercomputers with heterogeneous processors, there is still a great challenge to improve the scalability as well as to reduce the synchronization and communication costs between processes with irregularly sparse structured dependencies.

IV. Sparse Iterative Solver

In contrast to direct methods, iterative methods start from an initial conjectural solution and use the information in the system of equations to iteratively update it, gradually approximating the true solution. With a gradual increase in the size of the circuit, direct solvers could be very inefficient for these large sparse circuit matrices due to the memory limitation. Even after the permutation, the number of fill-ins generated during factorization may be huge. Therefore, iterative algorithms are gradually gaining attention to solve such problems [19]. In addition, for some circuit matrices with special structure, i.e., RF circuits with block format, iterative solver may also demonstrate effectiveness. Since the convergence speed of iterative methods depends too much on precondition techniques, research mainly focuses on constructing a good preconditioner.

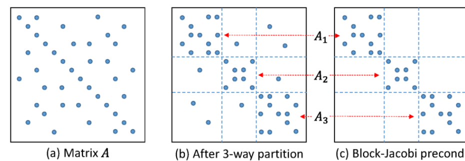


Fig. 9: An example of building a block-Jacobi preconditioner from a matrix A based on 3-way partitioning [29].

Zhao et al. [34] proposed a preconditioner using spectral sparsification for nonlinear circuit simulation. Li et al. [19] proposed a parallel incomplete LU (ILU) preconditioned generalized minimal residual (GMRES) solver in transient analysis for solving linear and nonlinear circuits. In [29], a block Jacobi preconditioner was proposed as shown in Fig. 9. Chow et al. [8] developed a fine-grained parallel algorithm for ILU. However, iterative solvers have not yet been widely adopted as a practical alternative in circuit simulation.

V. Conclusions and Analysis

In this paper, we present a perspective survey on recent progress of high-performance sparse linear solvers tailored for circuit simulation. Several innovative algorithms have been presented, including strategies for harnessing machine learning techniques to address irregular sparsity distribution patterns, as well as methods that embrace synchronization-free concepts to design GPU and heterogeneous distributed cluster acceleration mechanisms, thereby harnessing the substantial parallel computing capabilities. Additionally, we provide a brief introduction to several iterative solvers recently developed in circuit simulation. We also highlight the challenges and future opportunities in this evolving landscape.

- For circuit matrices, the traditional acceleration methods for finding dense sub-matrices are no longer applicable for the irregular nature of the nonzero element distribution. How to introduce sparse computations and strike a trade-off between different computational kernels remains a pivotal challenge.
- In GPU acceleration, how to utilize massive computational units more efficiently is the key to improve performance. This usually requires sufficiently large matrix to be solved with good parallelism. Moreover, better task scheduling strategies and computational kernels need to be designed on GPUs to improve the parallel computing performance.
- It is observed that no single solver can achieve optimal performance for all matrices, mainly due to the difficulty of taking into account various factors such as the matrix and hardware platform characteristics in algorithm design. Additionally, the data distribution of the matrix and its inherent data dependencies may significantly influence the results.

In the future, we believe that the following research directions are worth exploring, 1) AI-based unsupervised or semi-supervised optimization, such as producing brand new matrix reordering method or computation kernel algorithm. 2) Design optimization algorithms that are better suited for GPUs based on various circuit matrix structures. 3) Distributed synchro-

nization and communication optimization. 4) Obtaining oracle preconditioners in iterative solver.

VI. ACKNOWLEDGEMENT

This work was supported by National Key R&D Program of China (Grant No. 2021YFB0300600), the NSFC Key Program (Grant No. 61972415, 62204265, 62234010, 62372467), State Key Laboratory of Computer Architecture (ICT, CAS) (Grant No. CARCHA202115), GHfund A (Grant No. 202302017546). Weifeng Liu is the corresponding author.

REFERENCES

[1] A. Abdelfattah, P. Ghysels, W. Boukaram, S. Tomov, X. S. Li, and J. Dongarra. Addressing irregular patterns of matrix computations on gpus and their impact on applications powered by sparse direct solvers. In *SC '22*, 2022.

[2] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. Mumps: a general purpose distributed memory sparse solver. In *PARA '00*, 2000.

[3] Q. Chen, Y. Ye, M. Li, H. Yan, and L. Shi. Optimized matrix ordering of sparse linear solver using a few-shot model for circuit simulation. *Integration, the VLSI Journal*, 2023.

[4] X. Chen. Numerically-stable and highly-scalable parallel lu factorization for circuit simulation. In *ICCAD '22*, 2022.

[5] X. Chen, L. Ren, Y. Wang, and H. Yang. Gpu-accelerated sparse lu factorization for circuit simulation with performance modeling. *IEEE TPDS*, 2014.

[6] X. Chen, Y. Wang, and H. Yang. Nicslu: An adaptive sparse matrix solver for parallel circuit simulation. *IEEE TCAD*, 2013.

[7] Y. Chen, H. Pei, X. Dong, Z. Jin, and C. Zhuo. Application of deep learning in back-end simulation: challenges and opportunities. In *ASP-DAC '22*, 2022.

[8] E. Chow and A. Patel. Fine-grained parallel incomplete lu factorization. *SIAM SISC*, 2015.

[9] G. Cui, W. Yu, X. Li, Z. Zeng, and B. Gu. Machine-learning-driven matrix ordering for power grid analysis. In *DATE '19*, 2019.

[10] T. A. Davis. Algorithm 832: Umfpack v4.3-an unsymmetric-pattern multifrontal method. *ACM TOMS*, 2004.

[11] T. A. Davis. *Direct methods for sparse linear systems*. SIAM, 2006.

[12] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM TOMS*, 2011.

[13] T. A. Davis and E. Palamadai Natarajan. Algorithm 907: KLU, A direct sparse solver for circuit simulation problems. *ACM TOMS*, 2010.

[14] X. Fu, B. Zhang, T. Wang, W. Li, Y. Lu, E. Yi, J. Zhao, X. Geng, F. Li, J. Zhang, Z. Jin, and W. Liu. Pangulu: A scalable regular two-dimensional block-cyclic sparse direct solver on distributed heterogeneous systems. In *SC '23*, 2023.

[15] P. Ghysels, X. S. Li, C. Gorman, and F.-H. Rouet. Strumpack: Scalable preconditioning using low-rank approximations and random sampling. In *SC '16*, 2016.

[16] K. He, S. X.-D. Tan, H. Wang, and G. Shi. Gpu-accelerated parallel sparse lu factorization method for fast circuit analysis. *IEEE TVLSI*, 2015.

[17] Z. Jin, H. Pei, Y. Dong, X. Jin, X. Wu, W. W. Xing, and D. Niu. Accelerating nonlinear dc circuit simulation with reinforcement learning. In *DAC'22*, 2022.

[18] W. K. Lee, R. Achar, and M. S. Nakhla. Dynamic gpu parallel sparse lu factorization for fast circuit simulation. *IEEE TVLSI*, 2018.

[19] L. Li, Z. Liu, K. Liu, S. Shen, and W. Yu. Parallel incomplete lu factorization based iterative solver for fixed-structure linear equations in circuit simulation. In *ASP-DAC '23*, 2023.

[20] X. S. Li. An overview of superlu: Algorithms, implementation, and user interface. *ACM TOMS*, 2005.

[21] X. S. Li and J. W. Demmel. Superlu_dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM TOMS*, 2003.

[22] F. N. Najm. *Circuit simulation*. John Wiley & Sons, 2010.

[23] S. Peng and S. X.-D. Tan. Glu3.0: Fast gpu-based parallel sparse lu factorization for circuit simulation. *IEEE Design & Test*, 2020.

[24] P. Sadayappan and V. Visvanathan. Circuit simulation on shared-memory multiprocessors. *IEEE TC*, 1988.

[25] P. Sao, X. S. Li, and R. Vuduc. A communication-avoiding 3d algorithm for sparse lu factorization on heterogeneous systems. *JPDC*, 2019.

[26] P. Sao, R. Vuduc, and X. S. Li. A distributed cpu-gpu sparse direct solver. In *Euro-Par '14*, 2014.

[27] O. Schenk, K. Gärtner, W. Fichtner, and A. Stricker. Pardiso: a high-performance serial and parallel sparse linear solver in semiconductor device simulation. *FGCS*, 2001.

[28] T. Wang, W. Li, H. Pei, Y. Sun, Z. Jin, and W. Liu. Accelerating sparse lu factorization with density-aware adaptive matrix multiplication for circuit simulation. In *DAC '23*, 2023.

[29] Y. Wang, W. Zhang, P. Li, and J. Gong. Convergence-boosted graph partitioning using maximum spanning trees for iterative solution of large linear circuits. In *DAC '17*, 2017.

[30] W. W. Xing, X. Jin, T. Feng, D. Niu, W. Zhao, and Z. Jin. Boa-pta: A bayesian optimization accelerated pta solver for spice simulation. *ACM TODAES*, 2022.

[31] I. Yamazaki and X. S. Li. New scheduling strategies and hybrid programming for a parallel right-looking sparse lu factorization algorithm on multicore cluster systems. In *IPDPS '12*, 2012.

[32] Z. Yan, B. Xie, X. Li, and Y. Bao. Exploiting architecture advances for sparse solvers in circuit simulation. In *DATE '22*, 2022.

[33] J. Zhao, Y. Wen, Y. Luo, Z. Jin, W. Liu, and Z. Zhou. Sflu: Synchronization-free sparse lu factorization for fast circuit simulation on gpus. In *DAC '21*, 2021.

[34] X. Zhao and Z. Feng. Gpscp: A general-purpose support-circuit preconditioning approach to large-scale spice-accurate nonlinear circuit simulations. In *ICCAD '12*, 2012.