Exploiting Architecture Advances for Sparse Solvers in Circuit Simulation

Zhiyuan Yan[*†‡], Biwei Xie[*†‡], Xingquan Li[‡§], Yungang Bao[*†‡]
[*]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
[†]University of Chinese Academy of Sciences
[‡]Peng Cheng Laboratory
[§]Minnan Normal University
{yanzhiyuan, xiebiwei, baoyg}@ict.ac.cn, fzulxq@gmail.com

***Abstract*—Sparse direct solvers provide vital functionality for a wide variety of scientific applications. The dominated part of the sparse direct solver, LU factorization, suffers a lot from the irregularity of sparse matrices. Meanwhile, the specific characteristics of sparse solvers in circuit simulation and unique sparse pattern of circuit matrices provide more design spaces and also great challenges.**

In this paper, we propose a sparse solver named FLU and re-examine the performance of LU factorization from the perspectives of vectorization, parallelization, and data locality. To improve vectorization efficiency and data locality, FLU introduces a register-level supernode computation method by delicately manipulating data movement. With alternating multiple columns computation, FLU further reduces the off-chip memory accesses greatly. Furthermore, we implement a fine-grained elimination tree based parallelization scheme to fully exploit task-level parallelism. Compared with PARDISO and NICSLU, experimental results show that FLU achieves a speedup up to 19.51× (3.86× on average) and 2.56× (1.66× on average) on Intel Xeon respectively.

***Index Terms*—High Performance Computing, Circuit Simulation, Sparse LU Factorization**

I. Introduction

Integrated circuit simulation relies heavily on the SPICE-based simulators [1]. As the modern VLSI system scales up, simulation becomes extremely time-consuming and usually takes days and weeks. The kernel of the SPICE-based circuit simulator is a sparse solver [2] ($Ax = b$), in which Lower–Upper (LU) factorization based solution is proved to be efficient and widely adopted for better robustness and reliability. LU factorization factors a matrix into the product of a unit lower triangular matrix (L) and an upper triangular matrix (U). The runtime of the SPICE simulator is dominated by repeatedly invoking the sparse solver, in which the dimension of matrix A may be millions.

The inherent irregularity characteristics of sparse problems inhibit sparse LU factorization from fully utilizing the powerful capacity of modern architecture. The complex computation and memory access pattern make the situation even severe. Existing works, such as PARDISO [3], SuperLU [4], and MUMPS [5], concentrate on general-purpose sparse solvers and has already achieved good performance. Sparse solvers for SPICE simulators, such as KLU [6] and NICSLU [7] [8], are based on the Gilbert-Peierls algorithm and NICSLU implements the parallel version. SFLU [9] presents a synchronization-free parallel mechanism and further improves performance.

In this paper, we propose FLU, a novel LU-based sparse solver, which further improves the performance of LU factorization in circuit simulation. We consider the specific characteristics of SPICE-based circuit simulation and extend the design space from two perspectives: 1) the re-factorization phase of a SPICE-based solver iterates thousands of times with fixed symbolic structure and pivoting order; 2) the unique non-zero elements distribution of circuit matrices requires a more appropriate methodology for computation and data layout.

Our experiments show that on multi-core Xeon and many-core Xeon Phi processors, FLU can achieve a speedup up to 19.51× (3.86× on average) and 39.94× (6.94× on average) over PARDISO respectively; a speedup up to 2.56× (1.66× on average) and 3.74× (1.84× on average) over the best-existing approach NICSLU respectively.

Our primary contributions in this paper are listed as follows:

- By delicately manipulating data movement, we present a register level method to improve data locality and vectorization efficiency of supernode computation.
- We propose a multi-column method to alternatingly compute multiple columns, which can further reduce off-chip memory accesses of L.
- We present a fine-grained parallel scheme based on an elimination tree, which can fully exploit task-level parallelism and further reduce the thread waiting time.
- We evaluate our work with state-of-the-art sparse solvers on 37 circuit matrices and 73 general matrices. We analyze their performance and scalability.

The remainder of this paper is organized as follows. The basics of G/P algorithm, advanced architecture features, and motivation are introduced in Section II. We illustrate our optimized approaches in Section III. Section IV demonstrates and analyzes experimental results. We summarize related work in Section V. Finally, conclusions are represented in Section VI.

II. Background and Motivation

A. Gilbert-Peierls algorithm

Gilbert-Peierls algorithm (G/P algorithm) is widely used in the state-of-the-art solvers, including KLU, NICSLU, and SuperLU. It is a left-looking algorithm, i.e., the decomposed column is updated by the dependent left-hand columns. Algorithm 1 shows the pseudo-code of the G/P algorithm without

Algorithm 1 G/P algorithm without pivoting

```
1: L = I;
2: for $k = 1 : N$ do
3:     $tempvec$ = A(:,$k$);
4:     for $j = 1 : k - 1$ where U$(j, k)$ != 0 do
5:         $tempvec(j + 1 : N) -= tempvec(j) \times \mathbf{L}(j + 1 : N, j)$;
6:     end for
7:     $\mathbf{U}(1 : k, k) = tempvec(1 : k)$;
8:     $\mathbf{L}(k : N, k) = \frac{tempvec(k:N)}{tempvec(k)}$;
9: end for
```

pivoting. G/P algorithm is based on Gaussian elimination. It will traverse all the dependent columns on the left side and may generate fill-ins. This procedure is illustrated in line 4 and 5 of Algorithm 1. Column k depends on column j if and only if U(j, k) is a non-zero element (line 4).

Given that column k depends on column j ($j < k$), then the update operation occurs (line 5). Firstly, column k will be expanded to array *tempVec* with length N. N is the dimension of the matrix. Secondly, the non-zero elements of L will be read in sequence and multiply with *tempVec*. Finally, the results will be subtracted and written back to *tempVec*.

The dependency relationship has been known before the computation in Algorithm 1. This is decided by the specific feature of circuit simulation: the first factorization needs to be reordered and pivoted but the subsequent factorizations can be performed directly because the non-zero pattern of the matrices are fixed during Newton-Raphson iterations.

B. Advanced Architecture Features

Emerging architecture features bring both great chances and challenges for many performance-critical applications.

*a) **Vectorization**:* By enlarging SIMD width from AVX2(256 bit) to AVX512(512 bit) [10], Intel CPU can provide better hardware performance at data level parallelism (DLP). AVX512 offers 512-bit vector instructions and supports gather/scatter for non-continuous memory read and write.

*b) **Parallelization**:* Current vendors are trying to integrate many more cores than before in a single machine [11]. Machines with multi and many cores have become quite common in data centers, for example, an Intel Skylake machine may have up to 24 cores (48 HT) and an AMD Ryzen Threadripper has 64 cores (128 HT). Considering NUMA-based multi-socket architecture, there may be more than 100 cores in a single machine. Moreover, a single Intel Xeon Phi(Knights Landing) [12] CPU has up to 68 cores (272 HT).

C. Motivation

By combining the advanced features of emerging hardware and the sparsity patterns of sparse LU factorization, we analyze the performance bottleneck and design space from the following three perspectives:

1) Supernode Computation. We break the running time of LU factorization down to two parts on 17 representative sparse matrices in Fig.1. The results show that supernode-column related computation dominates the whole running time

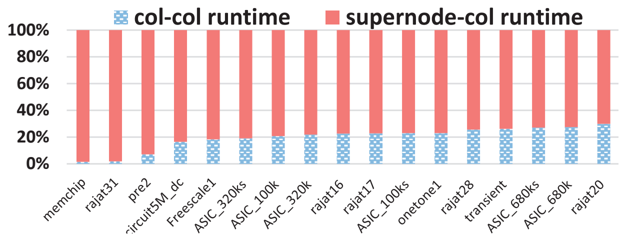

Fig. 1. Breakdown of LU decomposition running time.

(from 70% to 98%, and 80% on average), which makes it reasonable to fully pay more attention to supernode computation. Most state-of-the-art solvers may use kernels like GEMM from general BLAS libraries like OpenBLAS and Intel MKL for better performance. However, the general kernel lacks consideration of the unique characteristics of LU factorization, in which GEMM is performed on rectangular matrices with a large length-width ratio. It is necessary to customize a novel supernode-column computation scheme to further exploit the vectorization and memory access efficiency.

2) Parallelization. Data dependency is the dominating obstacle to parallelizing sparse LU factorization. Threads consume some time on waiting and they will become idle threads in the worst case. Thus the advantage of multi-core cannot be fully exploited. In general, the degree of parallelism is decided by the elimination tree which is created in the symbolic stage. This tree will not be fat when the matrices are sparse. So the parallel mechanism level-by-level is not appropriate for circuit simulation solver. It is necessary to present a fine-grained parallelization scheme.

3) Data Locality. As shown in Algorithm 1, loading data from L and U and storing data to *tmpVec* always involve lots of memory accesses. Due to the data being stored in compressed format, a mass of indirect memory visits occur. The principle of improving memory access efficiency is not to convert irregularity into regularity, but to make it less irregular. Improving data re-utilization and keep it stay longer in register or cache is a promising direction to try.

III. FLU Methodology

A. Basic Idea

To address the challenges of mapping the irregular LU factorization algorithm to modern architecture, our proposed FLU re-examines the parallelization and vectorization approaches to better exploit the computational capacity of multi/many-core and large SIMD width from three perspectives:

- Register level vectorization for supernode computation;
- Multi-column scheme for better data locality;
- Fine-grained parallelization scheme.

B. Register Level Supernodal Computation

Supernode-related computation occupies a large part of the whole LU factorization procedure. We design a vectorization scheme that operates data mainly at the register level. Illustrated in Fig. 2, the whole scheme involves three steps:

Register Initialization. Here we use *register* to indicate the *SIMD register* in default. There are ω lanes in each register, and

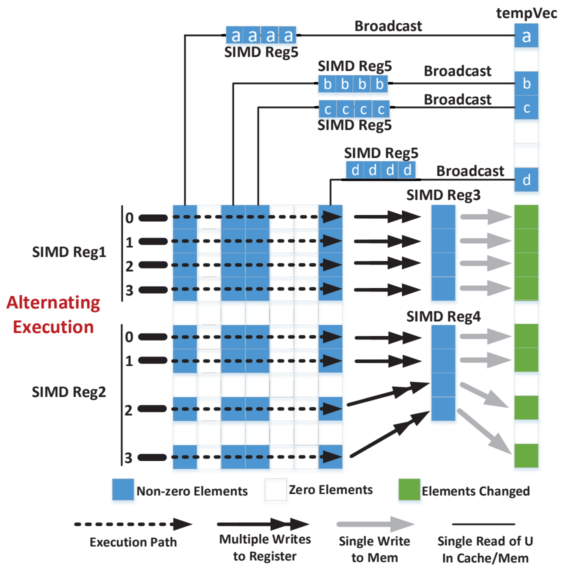

Fig. 2. Vectorized supernode-column computing

ω varies depending on the architecture. For AVX512, ω is 8 for double precision and 16 for single precision, and for AVX2, ω is 4 and 8 respectively. We set ω to 4 in Fig. 2 only for simplicity and better demonstration. For each computation task, we need three registers: Reg1 for data from L, Reg5 for data from U, and Reg3 for intermediate data. In Fig. 2, we conduct two computation tasks alternatingly, thus two additional registers (Reg2 and Reg4) are introduced. Note that Reg3 and Reg4 are set to zero in advance.

Register Level Computation. We load ω elements from the first column of *supernode* into Reg1 and broadcast the according elements from U into Reg5. Reg1 and Reg5 will be multiplied and accumulated to Reg3. The computation of supernode is thus converted to consecutive *fmadd* instructions carried out mostly at register level. This procedure terminates until all the columns in *supernode* and all the elements in U are loaded, computed, and accumulated to Reg3. To further improve the data locality, we carry out θ (two or more) computation tasks alternatingly. We set θ to two in Fig. 2 for simplicity. When the first ω elements are loaded to Reg1, we also load the second ω elements from the same column to Reg2. We compute Reg1 and Reg2 alternatingly and write their results to Reg3 and Reg4 respectively.

Eliminating and Writing back. Intermediate results stored in Reg3 and Reg4 need to be subtracted by according elements in *tempVec* and written back. As elements in *tempVec* are not always aligned and consecutive. We use *gather* instruction to load data from *tempVec* and *scatter* instruction to write data back to *tempVec*. This procedure moves to the next θ groups of ω elements and repeats the same steps above.

Parameter Discussion. There are two parameters in our register level vectorization method. ω indicates the SIMD width and its value is fixed according to the underlying architecture. Alternatingly executing θ groups occupy at least (θ * 2 + 1) registers (Reg1, Reg3, and Reg5). Considering that δ (the number of SIMD registers of underlying architecture) is fixed, the value of θ is no larger than (δ -1)/2. θ also relies heavily on the memory bandwidth and cache hierarchy. Even if there are more SIMD registers available, too large θ may involve an unnecessary memory footprint and thus ruin the cache locality. For AVX512, we set θ = 2 in default.

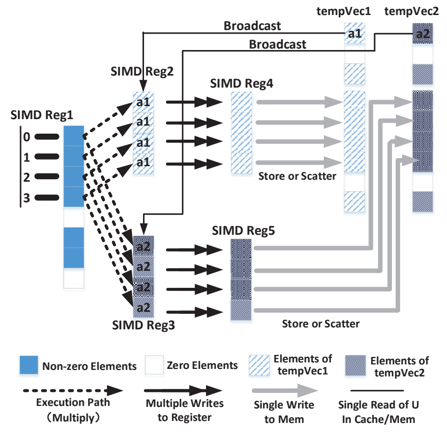

Fig. 3. Computing multiple columns alternatingly.

C. *Multi-column Scheme*

When computing consecutive columns with similar non-zero patterns, the same data of L may be loaded into cache multiple times, which results in cache thrashing and performance degradation. To relieve the memory pressure, FLU computes multiple columns(*tempVec1, tmepVec2* ...) alternatingly. Illustrated in Fig. 3 [1], the whole scheme involves three steps:

Multi-column Detection. Columns with similar non-zero element distribution access the same or adjacent columns in L. Suppose that τ is the number of columns. We compare the row index of non-zero elements of every τ consecutive columns to see how many of them are identical. If non-zero elements with identical row indices are larger than a specific value ϵ, they will be computed alternatingly. Otherwise, they would be computed one by one.

Multi-column Register Level Computation. In Fig.3, we set τ to two for simplicity. In total, we need five registers: Reg1 for data from L, Reg2 and Reg3 for data from U, Reg4 and Reg5 for intermediate data. Note that Reg4 and Reg5 are set to zero in advance. We firstly load ω elements into Reg1 from L. Then we load and broadcast the according elements of *tempVec1* and *tempVec2* into Reg2 and Reg3 respectively. Data in Reg1 will be used twice by both Reg2 and Reg3. Computation on τ columns is now converted to τ separate but alternating forward reduction operation with *fmadd* instructions. Reg4 and Reg5 store the final results after the accumulation.

Eliminating and Writing Back. Results in Reg4 and Reg5 need to be subtracted and written back to *tempVec1* and *tempVec2* respectively. Similar to the writing back procedure in

[1]Registers in Fig. 2 and Fig. 3 are independent, even if they may have the same name. We name them only for simplicity and better demonstration.

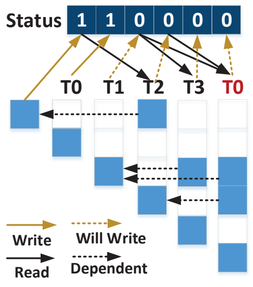

Fig. 4. Fine-grained parallelization scheme.

Section III-B, we suppose that the memory addresses in *tempVec1* and *tempVec2* are discontinuous and use *gather/scatter* instructions to *load/store* data. After this, the next ω elements of L would be fetched into the register and the same procedure repeats until τ columns are computed.

Parameter Discussion. τ columns computation requires τ temporary vectors like *tempVec1* and *tempVec2*. So, large τ consumes huge memory space. Manipulating multiple large vectors may severely ruin the cache locality. Besides, multi-column computation is also carried out at the register level. It occupies at least (τ * 2 + 1) registers (Reg1, Reg2, and Reg4). Considering that the number of SIMD registers of underlying architecture δ is fixed, the value of τ is no larger than (δ - 1)/2. Moreover, θ in Section III-B makes available registers less. For AVX512, we set τ = 2 in default. The threshold value ϵ is less than or equal to 1.0. $\epsilon = 0$ means we only detect identical columns, which are quite small in quantity. When $\epsilon = 0.5$ means we need to carry out 50% redundant computation, which will offset the performance improvement and even make it worse. We set ϵ = 0.75 in default.

D. Fine-grained Parallelization Scheme

The parallelization performance of LU factorization relies heavily on the elimination tree. We introduce a *poll-wait* mechanism into this scenario (Fig. 4). There are two steps:

Data Preparation. We construct a bitmap-based state vector *stateVec*, in which *stateVec[i]* indicates whether the computation of column i completes. Note that all elements in *stateVec* are set to zero in default. 0 indicates the column is unscheduled or in progress, while 1 indicates the factorization of the column has been completed. We take four threads namely T0, T1, T2, and T3 for demonstration in Fig. 4.

Polling and Waiting. Before we compute a specific column, we first check the state of all the columns it depends on. If it is zero (means not ready), the current thread/task keeps waiting and polling. Otherwise, the current thread/task starts to work. Each thread/task updates according state in *stateVec* when it completes, while other threads/tasks which are waiting for it would be notified and continue to work.

E. Discussion

Though that FLU is designed for sparse solver in circuit simulation, its methodology is also applicable to general-purpose sparse solvers. The selection of parameters above, like θ, τ, and ϵ, is heavily affected by underlying architecture and input matrices. However, the given default value always show modest performance on most of the input matrices. We enlarge τ if the fill-in ratio($(nnz(L)+nnz(U))/nnz(A)$) is large. Note that the default values are selected for AVX512 platforms, but the same method of parameter selection is also applicable to other architectures.

TABLE I
THE CONFIGURATION OF EXPERIMENTAL PLATFORMS.

	Intel Xeon	Intel Xeon Phi
Processor type	Gold 6130 (Skylake)	7250 (Knights Landing)
Number of cores	32	68
Number of threads/core	2	4
Basic frequency (GHz)	2.10	1.40
Vector instruction set	AVX512	AVX512
L1 cache size	32KB (I)+32KB (D)	32KB (I)+32KB (D)
L2 cache size	1MB	34MB
L3 cache size	22MB	N/A
Memory size	126GB	64GB
OS	Ubuntu 18.04.2	Debian 9.9
Compiler	Intel®ICC 19.0	

IV. Experiments

A. Experiment Setup

Platforms. We conduct experiments on two representative platforms with totally different architecture design decisions: an Intel Xeon Phi based platform(many-core system) and an Intel Xeon based platform(multi-core system). The detailed hardware/software information of these two systems can be found in Table I.

Benchmark Suite. We use 110 sparse matrices collected from the University of Florida Sparse Matrix Collection [13]. 37 of them are selected from circuit simulation scenarios by taking all the input matrices of previous research [6] [14] and removing the redundant ones. The other 73 general matrices cover various domains: web graphs, fluid mechanics, etc.

Comparison. We evaluate the performance of FLU with two state-of-the-art solvers: NICSLU and PARDISO. NICSLU is well suited for extremely sparse matrices, especially for circuit matrices. PARDISO is a well-known sparse solver from Intel MKL and is widely used in various fields. We run each experiment 100 times and report the average factorization time. NICSLU allocates one and only one thread on each physical core. For a fair comparison, we run all experiments with 32 threads on Xeon and 68 threads on Xeon Phi.

B. Performance Results and Analysis

We compare the factorization time of FLU, NICSLU and PARDISO with 37 circuit simulation matrices on two platforms. Fig. 5 shows the execution time of FLU and NICSLU, normalized to PARDISO. Fig. 5(a) depicts the experimental results on Xeon platform with 32 threads. Similarly, Fig. 5(b) depicts the experimental results on Xeon Phi platform with 68 threads. Compared with NICSLU and PARDISO, FLU achieves an average speedup of 1.66× and 3.86× respectively on Xeon platform. Accordingly, the average speedup is 1.84× and 6.94× on Xeon Phi platform. It can be seen from Fig. 5 that FLU outperforms other two solvers for almost all matrices except

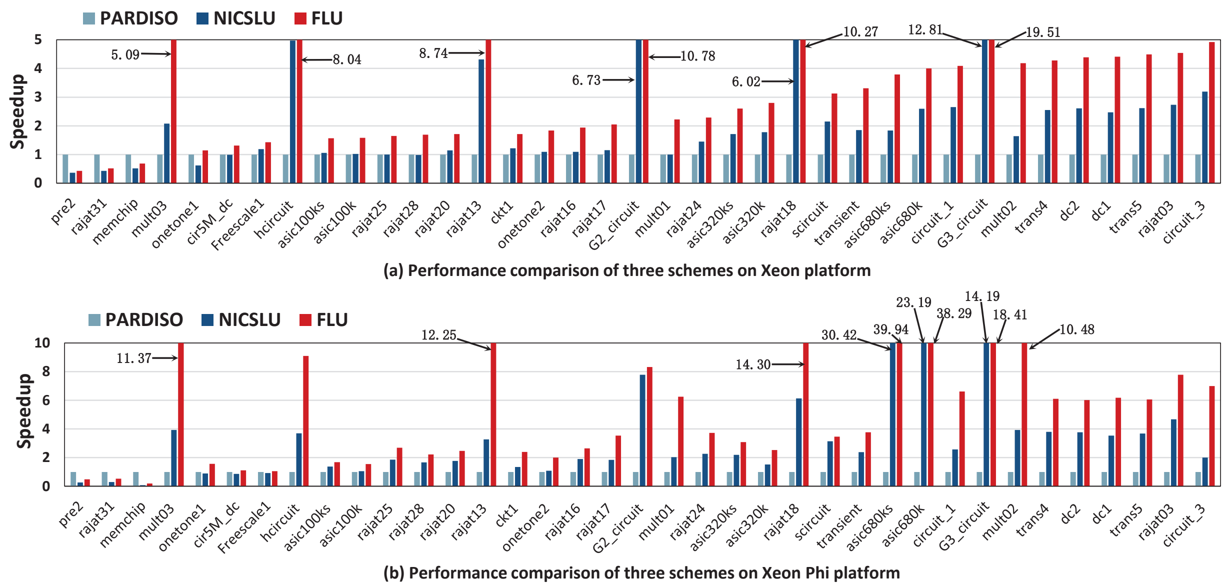
Fig. 5. Performance comparison of FLU, NICSLU and PARDISO. We normalize the execution time and show the speedup of FLU and NICSLU over PARDISO for better demonstration.

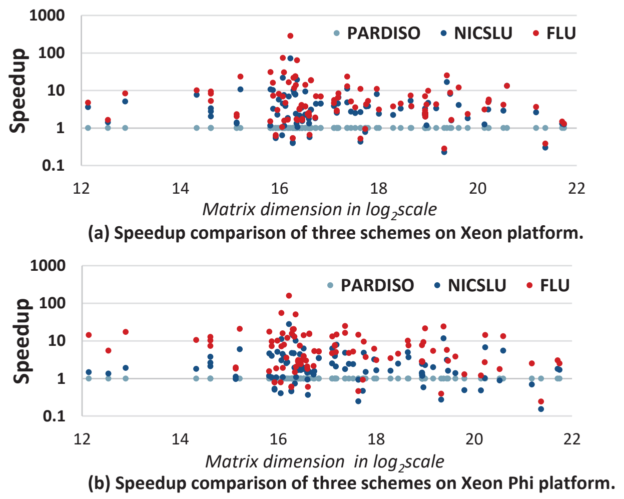
Fig. 6. Performance comparison of FLU, NICSLU and PARDISO on 110 matrices(37 circuit simulation matrices and 73 general matrices). We show the speedup of FLU and NICSLU over PARDISO for better demonstration.

three relatively dense matrices: *pre2, rajat31 and memchip*. Thus we can conclude that the proposed schemes of FLU improve the vectorization and parallelization efficiency of LU factorization. Although the supernode-supernode and right-looking mechanism of PARDISO make it much slower than FLU and NICSLU on most input matrices, it performs well on dense matrices. Fig. 5 also indicates that the performance improvement of FLU on Xeon Phi platform is better than that on Xeon. It demonstrates that FLU obtains a good and strong scalability when up to 68 threads.

Table II shows the speedups of NICSLU and FLU over their single-thread execution time, respectively. We analyze only FLU and NICSLU because their performance is significantly better than PARDISO, and omit the results on Xeon platform due to space limitation. FLU achieves average speedups of 2.74×, 7.15×, and 12.29× when the number of threads is 4, 16, and 64, respectively. Table II indicates that the speedups of FLU are higher than that of NICSLU. So FLU performs better on multi/many-core systems.

To further explore the performance of FLU, we conduct experiments on more sparse matrices from various domains. Fig. 6 shows the speedup of 110 matrices (consisting of 37 circuit simulation matrices and 73 general matrices) on Xeon and Xeon Phi platforms. Similar to Fig. 5, we normalize the execution time of FLU and NICSLU to PARDISO for better demonstration. On Xeon platform, FLU achieves average speedups of 1.73× and 10.85× over NICSLU and PARDISO respectively. Accordingly, the average speedup is 3.08× and 9.39× on Xeon Phi platform. The experimental results show that the proposed schemes of FLU is suitable for not only SPICE based circuit simulation, but also could be applied to more extensive scenarios.

V. RELATED WORK

Sparse solvers are essential building blocks of many scientific applications and have been studied for decades on various matrices from diverse prospects [2] [15]. Many solvers have been proposed to improve computational efficiency. However, the sparsity and irregularity of the sparse matrices make it difficult for the optimization of LU based sparse solver.

KLU [6] aims at circuit simulation and is widely used in SPICE simulator. However, it has only serial implementation. SuperLU [4] and MUMPS [16] are general solvers and can run

TABLE II
SPEEDUPS OF NICSLU AND FLU OVER SINGLE THREAD EXECUTION ON XEON PHI PLATFORM.

Dataset	NICSLU			FLU		
	T=4	T=16	T=64	T=4	T=16	T=64
pre2	2.95	5.12	6.40	2.83	5.22	9.06
rajat31	2.78	5.55	5.72	2.87	6.55	11.47
memchip	2.78	5.50	6.86	2.93	6.13	11.05
mult03	2.08	4.08	4.22	2.88	8.39	17.40
onetone1	2.91	9.79	13.24	2.95	8.71	20.26
cir5M_dc	2.53	4.42	5.39	3.08	5.57	8.53
Freescale1	2.56	3.90	4.80	3.15	6.21	9.51
hcircuit	1.41	2.76	2.64	2.12	5.08	10.22
asic100ks	2.74	8.38	11.09	2.83	8.49	14.96
asic100k	2.64	6.59	8.51	2.85	7.96	13.84
rajat25	2.74	7.45	10.44	2.84	8.62	18.26
rajat28	2.90	8.30	10.35	2.94	8.16	18.89
rajat20	2.73	7.60	10.77	2.84	8.48	18.19
rajat13	1.15	1.64	0.60	1.75	3.03	2.71
ckt1	2.39	6.51	9.20	2.89	9.10	16.87
onetone2	3.08	9.55	8.81	3.16	10.40	16.77
rajat16	2.81	8.03	9.97	2.98	9.48	20.37
rajat17	2.82	8.01	9.52	2.96	7.94	16.41
G2_circuit	2.13	5.06	6.97	2.70	7.47	11.03
mult01	2.32	4.13	5.13	3.17	9.50	18.00
rajat24	2.48	5.20	6.36	2.63	7.36	10.45
asic320ks	2.59	6.52	6.74	2.80	8.16	13.66
asic320k	2.49	5.12	5.80	2.82	7.24	12.46
rajat18	1.45	2.82	2.86	2.17	5.13	9.03
scircuit	1.78	4.28	5.09	2.76	6.61	10.53
transient	2.45	5.35	5.38	2.73	7.58	11.90
asic680ks	2.45	5.36	5.43	2.63	7.41	10.36
asic680k	2.38	4.59	4.18	2.65	6.39	8.58
circuit_1	2.55	2.86	1.66	2.12	3.95	3.18
G3_circuit	2.59	6.36	7.87	3.10	8.47	11.77
mult02	2.34	4.62	4.17	3.11	9.81	19.47
trans4	2.07	3.69	3.84	2.55	6.01	8.88
dc2	2.15	4.14	4.57	2.53	6.09	8.80
dc1	2.04	4.12	4.37	2.49	5.97	8.88
trans5	2.08	4.05	4.39	2.58	5.98	8.94
rajat03	3.13	6.02	3.60	3.14	8.33	9.95
circuit_3	1.58	2.07	0.67	2.03	3.47	3.96
Arithmetic mean	**2.41**	**5.39**	**6.15**	**2.74**	**7.15**	**12.29**

on a distributed system. These two solvers have one thing in common that they exploit the dense sub-matrices to accelerate the process of LU factorization by calling the functions of dense matrices, which are referred to as supernode and multifrontal technologies. But the state-of-the-art solvers don't optimize supernodal computation properly. They invoke the functions of CBLAS library directly. NICSLU [8] and PARDISO [3] aim at multi-core shared-memory system. The parallelization of these two solvers is based on a level-set and divides the execution into two modes according to the distribution of data dependent columns. Besides, many studies also focus on GPU [17] and FPGA [14] platforms. SFLU [9] presents a synchronization-free method to further improve the parallel performance on GPU platform.

VI. CONCLUSION

To further explore the design space of LU-based sparse solver in circuit simulation, we propose FLU, a novel sparse solver with full consideration of vectorization, parallelization, and data locality. FLU introduces a register-level supernodal computation method that could both improve vectorization efficiency and data locality. By alternatingly computing multiple columns, FLU further reduces the off-chip memory accesses greatly. Evaluation on 37 circuit matrices shows that FLU performs much better than NICSLU and PARDISO. Though that FLU is designed for circuit simulation, its methodology is also applicable to general sparse solvers. We extend the benchmarks to 73 general matrices and the experimental results indicate a speedup up to 39.94× and 2.56 over Intel MKL PARDISO and best-existing approach NICSLU respectively.

ACKNOWLEDGMENT

This work is supported in part by Key-Area Research and Development Program of Guangdong Province (NO.2020B010164003), the Major Key Project of PCL (PCL2021A08), the National Natural Science Foundation of China (Grant No. 62090020), Youth Innovation Promotion Association of Chinese Academy of Sciences (2013073), the Strategic Priority Research Program of Chinese Academy of Sciences (Grant No. XDC05030400), and Beijing Academy of Artificial Intelligence (BAAI).

REFERENCES

[1] L. W. Nagel, "Spice2: A computer program to simulate semiconductor circuits," *Ph. D. dissertation, University of California at Berkeley*, 1975.
[2] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar, "A survey of direct methods for sparse linear systems." *Acta Numer.*, vol. 25, pp. 383–566, 2016.
[3] O. Schenk and K. Gärtner, "Solving unsymmetric sparse systems of linear equations with pardiso," *Future Generation Computer Systems*, vol. 20, no. 3, pp. 475–487, 2004.
[4] X. S. Li, "An overview of superlu: Algorithms, implementation, and user interface," *ACM Transactions on Mathematical Software (TOMS)*, vol. 31, no. 3, pp. 302–325, 2005.
[5] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster, "A fully asynchronous multifrontal solver using distributed dynamic scheduling," *SIAM Journal on Matrix Analysis and Applications*, vol. 23, no. 1, pp. 15–41, 2001.
[6] T. A. Davis and E. Palamadai Natarajan, "Algorithm 907: Klu, a direct sparse solver for circuit simulation problems," *ACM Transactions on Mathematical Software (TOMS)*, vol. 37, no. 3, pp. 1–17, 2010.
[7] X. Chen, Y. Wang, and H. Yang, "A fast parallel sparse solver for spice-based circuit simulators," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 205–210.
[8] X. Chen, L. Xia, Y. Wang, and H. Yang, "Sparsity-oriented sparse solver design for circuit simulation," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 1580–1585.
[9] 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 *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 37–42.
[10] M. Cornea, "Intel avx-512 instructions and their use in the implementation of math functions," *Intel Corporation*, 2015.
[11] J. Diaz, C. Munoz-Caro, and A. Nino, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE Transactions on parallel and distributed systems*, vol. 23, no. 8, pp. 1369–1386, 2012.
[12] G. Chrysos, "Intel® xeon phi™ coprocessor-the architecture," *Intel Whitepaper*, vol. 176, p. 43, 2014.
[13] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.
[14] W. Wu, Y. Shan, X. Chen, Y. Wang, and H. Yang, "Fpga accelerated parallel sparse matrix factorization for circuit simulations," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2011, pp. 302–315.
[15] A. M. Bruaset, *A survey of preconditioned iterative methods*. CRC Press, 1995, vol. 328.
[16] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster, "Mumps: a general purpose distributed memory sparse solver," in *International Workshop on Applied Parallel Computing*. Springer, 2000, pp. 121–130.
[17] S. Peng and S. X.-D. Tan, "Glu3. 0: Fast gpu-based parallel sparse lu factorization for circuit simulation," *IEEE Design & Test*, vol. 37, no. 3, pp. 78–90, 2020.