SFLU: Synchronization-Free Sparse LU Factorization for Fast Circuit Simulation on GPUs

Jianqi Zhao[1], Yao Wen[1], Yuchen Luo[1], Zhou Jin[1], Weifeng Liu[1] and Zhenya Zhou[2]
1. Super Scientific Software Laboratory,
Department of Computer Science and Technology, China University of Petroleum-Beijing, Beijing, China
2. Huada Empyrean Software Co. Ltd Beijing, China
Email: 13022291538@163.com, wy044399@163.com, 546780156@qq.com, jinzhou@cup.edu.cn,
weifeng.liu@cup.edu.cn, zhouzhy@mail.empyrean.com.cn

2021 58th ACM/IEEE Design Automation Conference (DAC) | 978-1-6654-3274-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/DAC18074.2021.9586141

Abstract—Sparse LU factorization is one of the key building blocks of sparse direct solvers and often dominates the computing time of circuit simulation programs. Existing GPU-accelerated sparse LU factorization methods either offload relatively small dense matrix-matrix multiplications to GPU cores, or extract level-set information to parallelize elimination operations in each level. However, because of the insufficient parallelism, neither of the methods can saturate a large amount of compute units on modern GPUs.

We in this paper propose a synchronization-free sparse LU factorization algorithm called SFLU. To saturate GPU cores, our method lets each thread block eliminate a column and runs all the thread blocks at the same time. Through communicating dependency information stored on global memory, all the thread blocks either busy wait to run or get updated by their previous columns. Because elimination of all the columns work concurrently, our method avoids any barrier synchronization and saturates GPU resources. By benchmarking over 1000 sparse matrices on an NVIDIA Titan RTX GPU, our SFLU outperforms SuperLU and GLU by a factor of on average 155.71 and 8.21 (up to 3585.62 and 252.66), respectively.

Index Terms—sparse LU factorization, circuit simulation, GPU

I. Introduction

The sparse LU factorization $A = LU$ decomposes a sparse matrix A into the product of a unit lower triangular sparse matrix L and an upper triangular sparse matrix U. It may be the most studied kernel of sparse direct methods for solving linear system $Ax = b$ [1]. In a number of circuit simulation tools, such as the SPICE package [2], the most time consuming step is in general calling sparse solvers to solve systems of circuit equations generated from linear and nonlinear circuits. Thus the sparse LU factorization becomes a crucial part of fast circuit simulation on modern parallel platform.

Even though a number of parallel sparse LU factorization methods, such as MUMPS [3], SuperLU [4], NICSLU [5]–[8], as well as GLU and its variants [9]–[11], already can use modern parallel processors such as GPUs, their performance is still unsatisfactory. The main reason is that their algorithms can not well use the thousands of compute units on modern GPUs. Specifically, the MUMPS and SuperLU find multifrontal or supernodes and offload the corresponding dense matrix-matrix multiplication, i.e., GEMM, operations to GPUs. But the multifrontals or supernodes generated from circuit simulation problems are typically too small to exploit massively parallel GPUs. In addition, the NICSLU and GLU need to divide the nodes in the graph form of the matrix into multiple levels with dependencies and solve the components inside each level in parallel. However, when the sizes of those levels are relatively small, the large amount of compute units on GPUs can not be saturated.

To exploit the massive parallelism on modern GPUs, we in this paper propose a synchronization-free sparse LU factorization algorithm called SFLU. Our method assigns the elimination work of each column to one thread block in CUDA, and issues all the thread blocks in a single kernel. Then through accessing a global memory array with dependency information, all the thread blocks busy wait until their dependencies on the other columns are relieved. Once a thread block knows its dependencies is resolved, it finishes its local work and updates the global memory array to remove corresponding dependencies to other thread blocks. Through this way, all the elimination work on the columns runs at the same time to saturate GPU resources, and does not need any kernel synchronization between levels as used in the NICSLU and GLU methods. As a result, the parallelism of our SFLU can be much higher than existing sparse LU factorization work.

We compare our SFLU with the latest implementations of SuperLU [4] and GLU [11] by benchmarking 1309 sparse matrices in the SuiteSparse Matrix Collection [12] on an Intel 20-core machine and an NVIDIA Titan RTX GPU. Our experimental results show that for circuit matrices, our method obtains up to 287.61 and 6.09 speedups over SuperLU and GLU, respectively. Also, for a wider range of sparse matrices, our SFLU is on average 155.71 (up to 3585.62) and on average 8.21 (up to 252.66) faster than SuperLU and GLU, respectively.

II. Background and Motivation

A. Sparse LU Factorization

The function of LU factorization is to decompose a square matrix A into the product of a unit lower triangular matrix L and an upper triangular matrix U. The typical

method is Gauss elimination. The elements in L and U can be determined according to the following two formulations:

$$U_{ij}=A_{ij}-\sum_{k=0}^{j-1}L_{ik}U_{kj}\left\{\begin{array}{c} i=1,2,...,N \\ j=i,i+1,...,N \end{array}\right. \quad (1)$$

$$L_{ij}=\frac{1}{U_{jj}}\left(A_{ij}-\sum_{k=1}^{j-1}L_{ik}U_{kj}\right)\left\{\begin{array}{c} i=1,2,...,N \\ j=1,2,...,i-1 \end{array}\right. \quad (2)$$

When the matrix A is sparse, the factorization procedure can be divided into three steps, i.e., preprocessing, symbolic decomposition and numeric decomposition. Preprocessing in general reorders the rows and columns of the original matrix to reduce the number of fill-ins, and the symbolic and numeric factorization are used to determine the sparsity structure of L and U and to calculate the values of their nonzeros, respectively.

B. Gilbert-Peierls Algorithm

The Gilbert-Peierls algorithm [13], or G-P algorithm for short, is a basic method used for sparse LU factorization. It eliminates a column by traversing the elements of the lower triangle part of its previous columns in a depth-first search way. Specifically, for computing fill-ins, the G-P method counts the nodes that can be traversed from the previous columns, and observes whether there is already a nonzero at the position of these nodes in the current column. If the position is a zero element, it needs to be filled with a nonzero. Figure 1 shows an example of factorizing a matrix of order 4 by calling the G-P method.

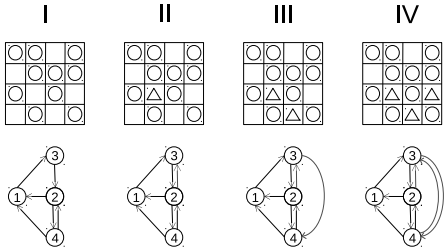


Fig. 1. The G-P method used for factorizing an example matrix of order four. The graph forms of the matrix in the four factorization steps are plotted.

C. GLU Algorithm

GLU is a recently developed GPU-accelerated sparse LU factorization package for circuit simulation and more general scientific computing. It was first proposed to mitigate the difficulty of fine-grain parallelization in the G-P method. He et al. [9] proposed a hybrid right-looking method on GPU, called GLU (also known as GLU v1.0). It solved the problem that the G-P algorithm can only work on one column at a time. Also, it has the benefits of the left-looking method for column-based parallelism and uses the same symbolic analysis routine. To compute the columns in parallel, GLU used a dependency detection algorithm, which considers the difference of the number of columns in different levels. GLU first detects data dependency between columns to factorize several columns in parallel. With complete information of dependency, columns can be grouped into levels, where all columns at the same level are independent of each other and can thus be factorized in parallel. However this also leads to synchronization time consumption.

After the algorithm updates, from GLU v1.0 to GLU v2.0 [10] and GLU v3.0 [11], GLU v3.0 still uses the column classification algorithm based on dependency detection proposed in GLU v1.0, which means the parallelization between columns in different levels may be still limited. In other words, a level has to wait until the columns in its previous levels are eliminated. It thus may cause much useless waiting time. Especially for some matrices with a large amount of level-sets, the synchronization waiting time may dominate the whole calculation time. This motivates us to design a new sparse LU factorization method that can avoid the synchronization costs.

III. SFLU: Synchronization-Free Sparse LU

As can be seen from the above analysis, we can find that synchronizations are often a bottleneck for parallel sparse LU on GPUs. In order to resolve this problem and further improve performance, we propose SFLU algorithm to avoid the synchronizations on GPUs. Our method still uses column-wise elimination as the basic working pattern, but issues all the work of the columns at the same time. We in this section will first introduce the basic idea of synchronization-free and then explain the symbolic and numeric factorization methods.

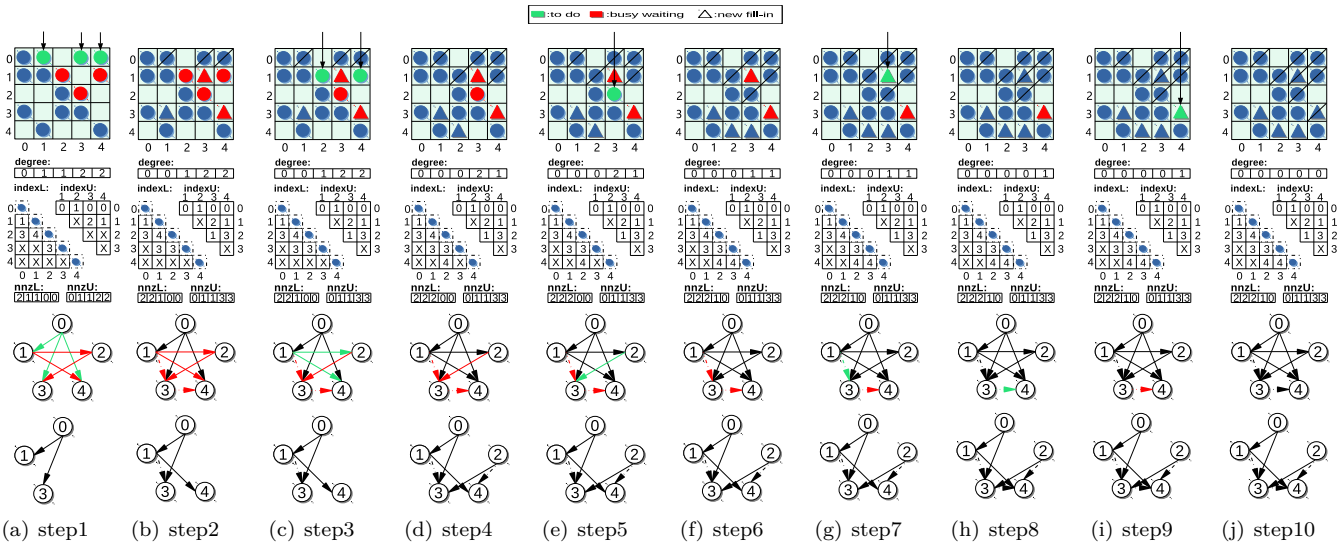
A. Basic Idea of Synchronization-Free Method

The way to avoid synchronization costs of parallel programs has always been one of the mostly studied topics for improving parallelism. To the best of our knowledge, in the field of matrix factorization, our work for the first time avoids synchronization cost in parallel algorithm design.

In a synchronization-free algorithm, a global memory space is allocated in advance, and the working state of a CUDA thread block in the parallel program is determined by a value in this memory space. The value will also change as the thread block's task is completed, which in turn changes the working status of other thread blocks until the whole task is finished. During the running of the program, all thread blocks can access this global memory space. Thus they know when to start, otherwise the thread blocks will be in a busy waiting state. Because the thread blocks work simultaneously through one single CUDA kernel call, the synchronization cost between kernels is removed. Also, since the thread blocks are self scheduled, higher parallelism can be obtained.

B. Symbolic Factorization

SFLU can be divided into symbolic and numeric factorization phases. Symbolic factorization runs first to



(a) step1 (b) step2 (c) step3 (d) step4 (e) step5 (f) step6 (g) step7 (h) step8 (i) step9 (j) step10

Fig. 2. The SFLU method used for factorizing an example matrix of order five. The graph forms of the matrix in the ten factorization steps are plotted. The following directed acyclic graph is composed of the columns whose elements have been eliminated(degree=0),and corresponding to the process graph of element elimination above.The green arrow of the matrix points to the working point, the red arrow points to the busy waiting point, the black arrow points to the finished point, and the dotted line points to the new non-zero element.

Algorithm 1 Symbolic factorization of SFLU

```
1: /*All columns are executed in parallel*/
2: for  all columns in parallel do
3:     /*Set the column be k*/
4:     for  i = 0 to k where the A[i,k] != 0 do
5:         /*Eliminate elements above the diagonal*/
6:         while degree[i] != 0 do
7:             //Busy wait
8:         end while
9:         Eliminate A[i,k] by using ith column
10:        /*After elimination, update the value of degree*/
11:        atomicSubtract(degree[k], 1)
12:    end for
13: end for
```

determine the final nonzero element structures of the factors L and U, and then the numeric factorization is performed to calculate the values of the nonzero element in the resulting matrices.

We introduce a global integer array *degree*, which is of length n, where n is the order of the matrix A. The values in the array *degree* corresponds to the number of elements not deleted above the diagonal of each column of the matrix. From the data dependence of symbolic factorization of left-looking algorithm, we can know that when the elements above the diagonal of a column k are eliminated (at this time, $degree[k] = 0$), it is proved that the column has completed the symbolic factorization, and the elements above the diagonal line of the kth row can be eliminated based on the elements of column k. We assign a thread block to each column of the matrix, which is responsible for the actual factorization of the column. Thread 0 of the thread block is responsible for observing the change of the values in the *degree* array. When the corresponding *degree* value changes to 0, other threads of the thread block will start to decompose. Otherwise, thread 0 will make the whole thread block in a busy waiting state. The actual factorization work will be completed by all threads except number 0, such as adding symbolic factorization, adding nonzero elements and numerical calculation of numeric factorization. When an element of the column is eliminated, thread 0 of the thread block is responsible for atomically subtracting the value of *degree* of this column by 1. If the value of *degree* in this column finally becomes 0, this update will change the working state of the thread block responsible for other columns (from busy waiting state to working state).

Our algorithm is based on the left-looking algorithm, in which the processing sequence logics of symbolic and numeric factorization are basically the same. We use a 5x5 matrix example to introduce the processing logic of the symbolic factorization in Figure 2, in which the 10 subfigures represent the elimination action. We use circles to represent the elements already existing in the matrix, triangles to represent newly filled elements, and colors to indicate the state of the elements (green represents the elements being eliminated, and red represents elements in a busy waiting status). Also, we respectively use two arrays indexL and indexU to represent the corresponding row index of each column element in L and U, and nnzL and nnzU to denote the number of elements in each column in L and U. We introduce two graphs to represent the algorithm action and matrix state. The graph at the top represents the action of the algorithm, in which the green

arrow line from vertex a to b is used to eliminate column b; the red arrow line from a to b represents that column b is waiting to be eliminated by column a; the black line represents the completed elimination work; and the dotted line in the middle represents the work change due to the addition of nonzero elements. The directed acyclic graph below represents the state of the matrix, which consists of columns with corresponding *degree* values of zero.

In the first step, the value of *degree*[0] is 0, which indicates that the nonzero elements in the 0th row can be eliminated (i.e., the corresponding thread block releases the waiting state and starts working), and the nonzero elements added in the elimination process are filled into the resulting factors (lines 6-9 in Algorithm 1). The diagram of the second step shows the matrix state after the elimination of the first step, three new nonzero elements are filled into the matrix. Meanwhile, the values of each array are updated according to the changes (line 11 in Algorithm 1). In the third step, since the value of *degree*[1] is changed to 0, we can eliminate the elements above the diagonal of the first row. It is worth noting that the storage logic of the new element is at the end of the column, and the thread block of each column is to eliminate the sequence according to the row index indexU of the column. Therefore, we can understand why the element a_{13} is still in the waiting state. There are two conditions to eliminate it: the value of *degree*[1] is 0 (achieved) and a_{23} is eliminated (not achieved). In step 4, we complete the elimination of the two elements in the second line without filling in the new elements. The element a_{23} can be eliminated due to the change of *degree*[3] in step 5. In the sixth step, after eliminating a_{23}, a new element is filled in, but the value of *degree* does not change. At the same time, the conditions for eliminating a_{13} have been achieved. We eliminate it in step 7 and complete it in step 8, where the value of *degree*[3] is 0. We eliminate the element a_{34} in step 9 and complete all symbolic decomposition in step 10.

C. Numeric Factorization

Before the numeric factorization, we need to sort the indices of each column in L and U. The purpose is to ensure that the numeric factorization of any column is carried out in the logical order of each column from top to bottom. The processing logic of numeric factorization is basically the same as the symbolic factorization shown before. Each thread block is also assigned to each column for its numeric factorization. The working state of each column's thread block is guided by the value of the array *degree* as well.

From the pseudo code in algorithm 2, it can be seen that the numeric calculation of each column can be divided into two parts. The first part is to eliminate the upper triangular U (including the diagonal), and the second part is to compute the lower triangular L. The first part of SFLU's numeric factorization also involves the idea

Algorithm 2 Numeric factorization of SFLU

```
1: /*All columns are executed in parallel*/
2: for  all columns in parallel do
3:     /*Set the column be k*/
4:     for  i = 0 to k where the A[i,k] != 0 do
5:         /*Calculate the value of the upper triangular U
6:         (including diagonal)*/
7:         while degree[i] != 0 do
8:             //Busy wait
9:         end while
10:        for  j=i+1 to n where A(j,k) != 0 in parallel do
11:            A[j,k] = A[j,k] - A[j,i]*As[i,k]
12:        end for
13:        /*After calculation, update the value of degree*/
14:        atomicSubtract(degree[k], 1)
15:    end for
16:    for  i = k + 1 where A[i,k] != 0 in parallel do
17:        /*Calculate the value of the lower triangular L*/
18:        A[i,k] = A[i,k] / A[k,k]
19:    end for
20:    /*Complete factorization, update the value of degree*/
21:    atomicSubtract(degree[k], 1)
22: end for
```

of synchronization-free parallel processing. The algorithm will determine the working state of the column through the value of *degree*, and sequentially solve the elements above the diagonal. It is worth noting that in this process, after solving, the algorithm will perform related operations on the nonzero elements below the element (lines 10-12 in Algorithm 2). In this process, every time an element is solved, the *degree* value corresponding to the column will be reduced by 1 in an atomic way (line 14 in Algorithm 2). When all the elements above the diagonal (including the diagonal) are solved, the *degree* value of this column is 1. At this point, the second part of the factorization can be executed for computing the elements below the diagonal. All nonzero elements below the diagonal will be divided by the value of the diagonal element in this column (lines 16-19 in Algorithm 2). At this time, the numeric decomposition of this column has been completed, and the *degree* value corresponding to this column is subtracted by 1 still in the atomic way (lines 21 in Algorithm 2). The thread block of some columns will stop busy waiting due to the change of *degree*.

IV. Experimental Results

A. Experimental Setup

We implement the SFLU algorithm with CUDA and run tests on an Intel 20-core machine and an NVIDIA Titan RTX GPU (Turing architecture, 4608 CUDA cores and 24GB GDDR6 memory). We use in total 1309 sparse matrices downloaded from the SuiteSparse Matrix Collection [12] as the benchmark suite. In particular, we also select a number of representative matrices from circuit simulation problems to show the effectiveness of our method. Besides our SFLU work, we also test two existing sparse LU factorization packages SuperLU_DIST [4] and GLU for performance comparison.

TABLE I
A detailed performance comparison on circuit matrices

Martrix	n	nnz	Symbolic factorization time (ms)					Numeric factorization time (ms)				
			SFLU	GLU [11]	Speedup over [11]	SuperLU [4]	Speedup over [4]	SFLU	GLU [11]	Speedup over [11]	SuperLU [4]	Speedup over [4]
adder_dcop_42	1813	11246	1.75	4.93	2.82	1.12	0.64	2.28	1.97	0.87	14.32	6.29
circuit_2	4510	21199	3.74	12.61	3.37	2.36	0.63	2.06	3.07	1.49	48.64	23.65
fpga_dcop_13	1220	5892	0.16	1.24	7.60	0.52	3.18	0.35	1.68	4.73	5.43	15.33
Hamrle2	5952	22162	6.85	5.09	0.74	3.31	0.48	2.42	14.75	6.09	42.38	17.50
memplus	17758	126150	20.51	21.24	1.04	3.15	0.15	1.45	5.09	3.53	29.09	20.13
rajat27	20640	99777	5.71	24.58	4.31	6.07	1.06	3.91	7.79	1.99	151.57	38.79
rajat22	39899	197264	10.52	69.81	6.64	9.06	0.86	8.49	14.76	1.74	115.92	13.65
mult_dcop_03	25187	193216	25.77	972.41	37.73	50.88	1.97	41.66	38.94	0.93	11983.03	287.61

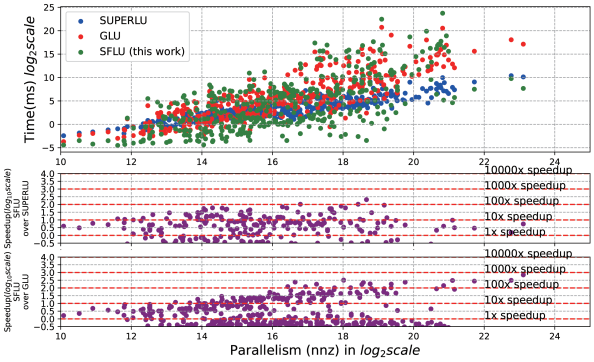


Fig. 3. Performance comparison of symbolic factorization.

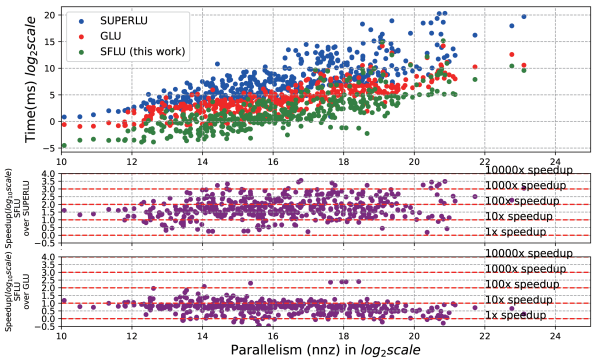


Fig. 4. Performance comparison of numeric factorization.

For all the benchmarks, we report the performance of their symbolic and numeric decomposition phases in Figures 3 and 4, respectively. The three subfigures in Figures 3 and 4 are the runtime of SuperLU, GLU and SFLU, the speedups of SFLU over SuperLU, and of SFLU over GLU, respectively.

B. Symbolic Factorization

In the comparison of the symbolic decomposition of the three methods, it can be seen from Figure 3 that our SFLU algorithm has obvious performance gain compared with SuperLU and GLU. It is worth to note that the symbolic decomposition of SuperLU and GLU adopts CPU serial algorithm, and normally gives degraded performance. But for some matrices originally with limited parallelism and many fill-ins, the sequential execution can give the best performance. For example, the performance of symbolic decomposition of SFLU in the matrix group *adder_dcop* is better than that of GLU, but is slightly slower than SuperLU. The performance of SFLU in the matrix group *fpga_dcop* is always better than the other two algorithms. Specifically, compared with SuperLU, the highest speedup of 205.14 appears in the matrix TSC_OPF_300. Compared with GLU, the highest speedup achieves a factor of 701 at the matrix *human_gene*2.

C. Numeric Factorization

As for the numeric phase, we can draw a conclusion from Figure 4 that the SFLU algorithm significant outperforms SuperLU and GLU. Specifically, our SFLU is faster than the other two methods in all matrices tested. Compared with SuperLU, our method is 3585.25 times faster, which is the highest speedup and appears on the matrix $c-52$. The reason is that in this matrix SuperLU cannot form relatively large supernodes and GEMM operations for saturating GPUs. Compared with GLU, the highest speedup achieved by SFLU is 252.2 on the matrix *nemth*18. The reason is that GLU in this case generated 9497 level-sets and thus need much synchronization cost. But our SFLU does not need such barrier synchronizations.

D. Circuit Matrix Performance

In order to study the performance of the SFLU algorithm for circuit matrices, we select different types of circuit matrix performance data from the whole benchmark suite and form Table I. The table includes the number of rows and nonzero elements of the matrix, as well as the time consumed by SuperLU, GLU and SFLU, and the performance speedup ratio of SFLU to the other two algorithms. In the table, we can get two observations. First, the performance of SuperLU symbolic factorization is better than that of SFLU and GLU. The reason may be related to the different preprocessing

operations performed. SFLU uses the same preprocessing method as GLU, and the symbolic decomposition performance of SFLU is significantly better than GLU. Among them, when decomposing matrix A, SFLU is 37.73 times faster than GLU, which proves that the advantages of SFLU symbolic decomposition are also applicable to circuit matrices. The second point is that, in the numeric factorization part, the performance of SFLU is mostly better than the other two algorithms. Compared with SuperLU, the highest speedup achieved by SFLU is 287.61 times on the matrix *mutl_dcop_*03. On the other hand, for matrix $Hamrle2$, SFLU is 6.09 times faster than GLU. Through the experiments, we can conclude that our SFLU gives state-of-the-art performance for circuit matrices.

V. Related Work

The irregularity of circuit simulation problems brings difficulties to utilizing GPUs for accelerating EDA programs. Fortunately, Garland [14] first pointed out that sparse matrix computations in circuit simulation can well utilize GPU in spite of their irregularity. Deng et al. [15] implemented fast sparse matrix multiplication algorithms for EDA tools on GPUs. Liu et al. [16] developed a synchronization-free method for sparse triangular solve on GPUs. Croix and Khatri [17] reviewed the use of GPU in EDA tools and provided insight into the type of problem best suited for the GPU architectures.

As for sparse LU on GPUs, based on the multifrontal and supernodal styles, packages MUMPS [3] and SuperLU [4] can gather columns of the similar structures into dense matrices of certain sizes (in general pretty small) and call fast dense matrix-matrix multiplication. However, sparse matrices from circuit analysis often do not have such regular multifrontal and supernodal structure, and thus seldom behave well on those packages. This is why the KLU tool [18] focused on single threaded computations.

The other group of work fully uses GPUs for sparse LU in the level-set way. The NICSLU package developed by Ren et al. [19] and Chen et al. [5]–[8], and the GLU package developed by He et al. [9], Lee et al. [10], and Peng and Tan [11] are representatives in this direction. Their methods need to find the parallelizable level information in the L and U after symbolic phase, and issue one GPU kernel for one level-set. As a result, the synchronizations between the kernel calls often takes much time. Though multiple optimizations have been proposed, the cost for synchronizations is still not negligible. In contrast, the SFLU method proposed in this work avoids kernel synchronizations and brings higher parallelism for sparse LU.

VI. Conclusion

We have proposed SFLU: a synchronization-free algorithm for sparse LU factorization on GPUs. SFLU deals with synchronizations through communicating on global memory with atomic operations on GPUs. The new method avoided the global synchronizations between levels in the existing methods and increased the amount of parallelizable work for GPUs of a large mount of compute units. Our experimental results demonstrated that SFLU is on average 155.7 and 8.21 (up to 3585.2 and 252.5) faster than SuperLU and GLU packages, respectively.

VII. Acknowledgement

We deeply appreciate the invaluable comments from the reviewers. Zhou Jin is the corresponding author of this paper. This work was supported by the National Natural Science Foundation of China under Grant No. 61972415.

References

[1] I. S. Duff, A. M. Erisman, and J. K. Reid, Direct Methods for Sparse Matrices, 2nd ed. Oxford University Press, Inc., 2017.
[2] L. W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," Ph.D. dissertation, EECS Department, University of California, Berkeley, 1975.
[3] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster, "MUMPS: A General Purpose Distributed Memory Sparse Solver," in Applied Parallel Computing. New Paradigms for HPC in Industry and Academia, 2001.
[4] X. S. Li, "An Overview of SuperLU: Algorithms, Implementation, and User Interface," ACM Trans. Math. Softw., 2005.
[5] X. Chen, Y. Wang, and H. Yang, "An adaptive LU factorization algorithm for parallel circuit simulation," in ASP-DAC '12, 2012.
[6] ——, "NICSLU: An Adaptive Sparse Matrix Solver for Parallel Circuit Simulation," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2013.
[7] X. Chen, L. Ren, Y. Wang, and H. Yang, "GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling," IEEE Transactions on Parallel and Distributed Systems, 2015.
[8] X. Chen, L. Xia, Y. Wang, and H. Yang, "Sparsity-oriented sparse solver design for circuit simulation," in DATE '16, 2016.
[9] K. He, S. X. . Tan, H. Wang, and G. Shi, "GPU-Accelerated Parallel Sparse LU Factorization Method for Fast Circuit Analysis," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2016.
[10] W. Lee, R. Achar, and M. S. Nakhla, "Dynamic GPU Parallel Sparse LU Factorization for Fast Circuit Simulation," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2018.
[11] S. Peng and S. X. . Tan, "GLU3.0: Fast GPU-based Parallel Sparse LU Factorization for Circuit Simulation," IEEE Design & Test, 2020.
[12] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," ACM Trans. Math. Softw., 2011.
[13] J. R. Gilbert and T. Peierls, "Sparse partial pivoting in time proportional to arithmetic operations," SIAM Journal on Scientific and Statistical Computing, 1988.
[14] M. Garland, "Sparse Matrix Computations on Manycore GPU's," in DAC '08, 2008.
[15] Y. S. Deng, B. D. Wang, and S. Mu, "Taming Irregular EDA Applications on GPUs," in ICCAD '09, 2009.
[16] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter, "A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves," in Euro-Par '16, 2016.
[17] J. F. Croix and S. P. Khatri, "Introduction to GPU Programming for EDA," in ICCAD '09, 2009.
[18] T. A. Davis and E. Palamadai Natarajan, "Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems," ACM Trans. Math. Softw., 2010.
[19] L. Ren, X. Chen, Y. Wang, C. Zhang, and H. Yang, "Sparse LU Factorization for Parallel Circuit Simulation on GPU," in DAC '12, 2012.