GPU-Accelerated Parallel Sparse LU Factorization Method for Fast Circuit Analysis

Kai He *Student Member, IEEE*, Sheldon X.-D. Tan *Senior Member, IEEE*
Hai Wang *Member, IEEE* and Guoyong Shi *Senior Member, IEEE*

***Abstract*—LU factorization for sparse matrices is the most important computing step for circuit simulation problems. However parallelizing LU factorization on the Graphic Processing Units (GPU) turns out to be a difficult problem due to intrinsic data dependency and irregular memory access, which diminish GPU computing power. In this article, we propose a new sparse LU solver on GPUs for circuit simulation and more general scientific computing. The new method, which is called *GLU* solver (for GPU LU), is based on a hybrid right-looking LU factorization algorithm for sparse matrices. We show that more concurrency can be exploited in the right-looking method than the left-looking method, which is more popular for circuit analysis, on GPU platforms. At the same time, the *GLU* also preserves the benefit of column-based left-looking LU method such as symbolic analysis and column-level concurrency. We show that the resulting new parallel GPU LU solver allows the parallelization of all three loops in the LU factorization on GPUs. While in contrast, the existing GPU-based left-looking LU factorization approach can only allow parallelization of two loops.**

Experimental results show that the proposed GLU solver can deliver 5.71× and 1.46× speedup over the single-threaded and the 16-threaded PARDISO solvers respectively, 19.56× speedup over the KLU solver, 47.13× over the UMFPACK solver and 1.47× speedup over a recently proposed GPU-based left-looking LU solver on the set of typical circuit matrices from University of Florida Sparse Matrix Collection (UFL). Furthermore, we also compare the proposed GLU solver on a set of general matrices from UFL, GLU achieves 6.38× and 1.12× speedup over the single-threaded and the 16-threaded PARDISO solvers respectively, 39.39× speedup over the KLU solver, 24.04× over the UMFPACK solver and 2.35× speedup over the same GPU-based left-looking LU solver. Also comparison on self-generated RLC mesh networks shows a similar trend, which further validates the advantage of the proposed method over the existing sparse LU solvers.

x

I. Introduction

Transforming a sparse matrix into its LU form is of crucial importance in linear algebra as it plays an important role in many numerical and scientific computing applications such as finite difference and finite element based methods. LU factorization operation represents the dominant computing cost in those problems and it is very important to improve the efficiency of the LU factorization algorithms. LU factorization for sparse matrices is the most important computing step for general circuit simulation problems for circuit designs. But parallelizing LU factorization on the popular many-core platforms such as Graphic Processing Units (GPU) turns out to be a difficult problem due to intrinsic data dependency and irregular memory access, which diminish GPU computing power.

Modern computer architecture has shifted towards the multi-core processor [1], [2] and many-core architectures [3]. The family of GPU is among the most powerful many-core computing systems in mass-market use [4]. For instance, the state-of-the-art NVIDIA Kepler K40 GPU with 2880 cores has a peak performance of over 4 TFLOPS versus about 80–100 GFLOPS of Intel i7 series Quad-core CPUs [5], [6]. In addition to the primary use of GPUs in accelerating graphics rendering operations, there has been considerable interest in exploiting GPUs for general purpose computation (GPGPU) [7].

Until now, dense linear algebra support on GPU is well developed, with its own BLAS library [8], but sparse linear algebra support is still limited. Modern NVIDIA GPUs are throughput-oriented many-core processors that can offer very high peak computational throughput. They favor computations exhibiting sufficient regularity of execution paths and memory access patterns. For sparse-matrice-based analysis, GPU acceleration has been applied to parallelize the shooting-Newton method for transient radio-frequency circuit analysis [9] and to speedup the generalized minimum residual analysis (GRMES) based iterative method for large-scale thermal analysis [10] in the past. However, parallelizing the sparse LU factorization operation is very difficult because of the irregular structure of matrices and the high data-dependency during the numeric LU factorization. As a result, they remain a challenge for GPU-based fine-grained parallel computing [5].

Several research efforts have been proposed for parallelizing sparse LU factorization on shared memory multi-core CPU and GPUs. SuperLU [11], [12] implemented supernode-based Gilbert-Peierls (G/P) left-looking algorithm [13], and SuperLU_MT [12] is its multi-threaded parallel version developed for shared memory multi-core architectures. However, it is not easy to form super-node in some sparse matrix such as circuit matrix. KLU [14], which is specially optimized for circuit simulation, adopts Block Triangular Form based on G/P left-looking algorithm.

Recently, the KLU algorithm has been parallelized on

Kai He and Sheldon X.-D. Tan is with Department of Electrical Engineering, University of California, Riverside, CA 92521 USA (e-mail:stan@ee.ucr.edu)

Hai Wang is with School of Microelectronics & Solid-State Electronics, University of Electronic Science & Technology of China, Chengdu, Sichuan, 610054 China

Guoyong Shi is with School of Microelectronics, Shanghai Jiao Tong University, Shanghai, China, 200240

This work is supported in part by NSF grant under No. CCF-1017090, and in part by 985 research funds from Shanghai Jiao Tong University and University of Electronic Science and Technology of China.

multi-core architecture by exploiting the column-level parallelism [15], [16]. For parallel LU factorization solvers on GPU, existing works mainly focus on dense matrices including [17], [18], [19], very few works on the sparse matrix have been proposed. Ren *et al.* recently proposed a GPU-based sparse LU solver based on the G/P left-looking algorithm [20]. It exploits the column-level parallelism due to sparse nature of the matrix. The left-looking based method, which transforms the factorization computing into a number of triangular matrix solving, seems more efficient on GPU computing. But it possesses higher data dependency coming from solving the triangular matrices. The traditional right-looking LU factorization, which involves only less data dependent vector operations, has not been well studied in GPU implementation.

In this article, we propose a new sparse LU solver on GPUs for circuit simulation and more general scientific computing. The new method, called *GLU* method, is based on a hybrid right-looking LU factorization algorithm. We show that more concurrency can be exploited in the right-looking method than the left-looking method, especially on GPU platforms. We have the following contributions:

- We propose a new column-based right-looking LU factorization method, which is shown to be more amenable for exploiting the concurrency of LU factorization. The new method preserves the benefit of column-level concurrency and symbolic analysis in the left-looking method meanwhile, it allows more parallelism to be exploited.
- We show that the new *GLU* LU solver allows the parallelization of all three loops in the LU factorization on GPUs. In contrast, the existing GPU-based left-looking LU factorization approach can only allow two-level parallelization. We conduct comprehensive studies on the new GPU LU solver on a number of published general matrices, circuit matrices and self-made large RLC circuit matrices against some existing LU solvers to demonstrate the advantage of the proposed GLU solver.

Numerical results show that the proposed GLU solver can deliver 5.71× and 1.46× speedup over the single-threaded and the 16-threaded PARDISO solvers [21] respectively, 19.56× speedup over the KLU solver [14], 47.13× over the UMFPACK solver [22] and 1.47× speedup over a recently proposed GPU-based left-looking LU solver [20] on the set of typical circuit matrices from University of Florida Sparse Matrix Collection (UFL) [23]. Furthermore, we also compare the proposed GLU solver on a set of general matrices from UFL, GLU achieves 6.38× and 1.12× speedup over the single-threaded and the 16-threaded PARDISO solvers respectively, 39.39× speedup over the KLU solver, 24.04× over the UMFPACK solver and 2.35× speedup over the same GPU-based left-looking LU solver. Also comparison on self-generated RLC mesh networks shows a similar trend, which further validates the advantage of the proposed method over the existing sparse LU solvers.

This article is organized as follows. Section II reviews previous work that has been done to factorize sparse matrices into LU form on GPU, in particular the left-looking algorithm, GPU architecture and CUDA programming. In Section III, we present the new column-based right-looking algorithm and its parallel implementation on GPU. Several numerical examples and discussions are presented in Section IV. Last, Section V concludes.

II. Review of LU factorization algorithms and CUDA

Before we present our new approach, we first review the two main-stream LU factorization methods: the left-looking G/P factorization algorithm [13] and a variant of the right-looking algorithms such as the Gaussian elimination method. We then review some recent works on LU factorizations on GPU and the NVIDIA CUDA programming system.

The LU factorization of a $n \times n$ matrix, A, has the form $A = LU$, where L is a lower triangular matrix and U is an upper triangular matrix. For a full matrix, LU factorization has $O(n^3)$ complexity as it has three embedded loops.

A. Right-looking factorization method

The right-looking LU factorization is the traditional factorization including the Gaussian elimination method. The algorithm can be explained by the following equation:

$$\begin{bmatrix} l_{11} & \\ l_{21} & L_{22} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ & U_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & A_{22} \end{bmatrix}, \quad (1)$$

where $l_{11} = 1$ is the a scalar, and l_{21} and u_{12} are the column and row vectors respectively, and L_{22} and U_{22} are the $(n-1) \times (n-1)$ submatrices. They can be computed by $u_{11} = a_{11}$, $u_{12} = a_{12}$, $l_{21} = a_{21}/u_{11}$. After this, we end up with a $(n-1) \times (n-1)$ equation to solve: $L_{22}U_{22} = A_{22} - l_{21}u_{12}$. The process repeats until we reach a 1×1 equation to solve. As we can see, the traditional right-looking method solves one row for U matrix and then one column for L matrix at each iteration. Then it updates the $(n-1)\times(n-1)$ submatrix A_{22} on the right part of the whole matrix and solves the reduced matrix recursively (so it is called the right-looking method). Note that the right-looking method requires that A_{ii} is first factored before we can factor $A_{i-1,i-1}$, which indicates the sequential data dependency of this algorithm and its limits for potential parallel implementations (although the multifrontal based hierarchical schemes can be exploited for parallelization [24]). Note that we ignore all the reordering steps for fill-in reduction and numerical pivoting as well as symbolic analysis steps as we will visit them later.

B. Left-looking factorization method

The G/P left-looking method shows better performances for sparse matrices and easier implementation than the traditional Gaussian elimination based methods. It also allows the symbolic fill-in analysis of L and U matrices before the actual numerical computing. Instead of computing one row of U and one column of L, the left-looking method computes one column for both L and U instead. This is achieved by solving a lower triangular matrix. This lower triangular solution is repeated n times during the entire factorization (where n is the size of the matrix) and each solution step computes a column

of the L and U factors. In this method, the matrix is traversed by columns from left to right. To compute current column, the algorithm has to look at all the previous computed columns on the left part of the L and U. So it is called left-looking method. Algorithm 1 shows one detailed implementation of the left-looking LU factorization. In this pseudo code, the current column is indexed by j, and the columns to the left of the current column are indexed by k. To compute the current column j, the algorithm looks left and finds all already factored column k ($k < j$), where $A_s(k, j) \neq 0$, and then uses these columns to update current column j. $A_s(x, y)$ indicates the LU symbolically factorized A matrix, where all the fill-ins and non-zero elements are assigned with non-zero initial values and their memories are allocated. Fig. 1 illustrates the basic idea of the left-looking algorithm. The key operation of the left-looking algorithm is the triangular matrix solving, which is actually performed by the so-called vector multiple-and-add (MAD) operations sequentially.

One important observation for the left-looking algorithm is that since all the fill-in patterns of factored matrices are exploited, we know some columns can be solved independently and in parallel, which is called column-level parallelism in the existing approaches. Such concurrency does not exist in the existing traditional right-looking algorithms due to the recursive nature of the algorithm as we mentioned before.

Algorithm 1 The Gilbert-Peierls left-looking algorithm

```
1:  for j = 1 to n do
2:    /*Triangular matrix solving*/
3:    for k = 1 to j − 1 where A_s(k, j) ≠ 0 do
4:      /*Vector multiple-and-add*/
5:      for i = k + 1 to n where A_s(i, k) ≠ 0 do
6:        A_s(i, j) = A_s(i, j) − A_s(i, k) ∗ A_s(k, j)
7:      end for
8:    end for
9:    /*Compute column j for L matrix*/
10:   for i = j + 1 to n where A_s(i, j) ≠ 0 do
11:     A_s(i, j) = A_s(i, j)/A_s(j, j)
12:   end for
13: end for
```

C. Related works

The G/P left-looking method shown in Fig. 1 has been parallelized on GPU recently [20]. This method exploits the two-level concurrency in the left-looking algorithm due to the sparsity patterns of the matrices. First, it exploits the column-level parallelism in the left-looking algorithm as mentioned earlier. Based on the matrix sparsity pattern, the independent columns can be grouped into levels. So the outer j loop of Algorithm 1 can be parallelized by processing columns level by level. The so-called *cluster mode* in this algorithm is for levels with many independent columns, while the *pipeline mode* is for levels with only a few columns. It also explores the parallelism within the vector MAD operation, which is reflected in the i loop of Algorithm 1. However, the middle k loop of column-by-column update, which is the key operation, is still in serial. The reason is that there is only one column j of the U matrix and updating this column must be done sequentially.

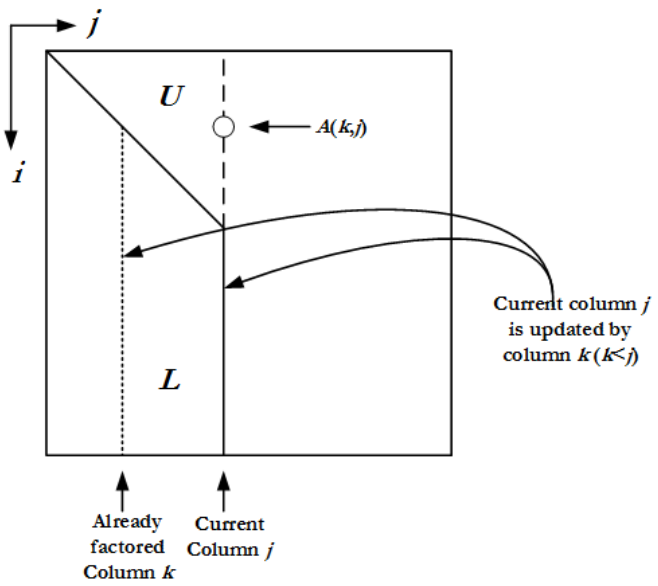

Fig. 1. Left-looking update for column j

D. Review of GPU Architecture and CUDA programming

In this subsection, we review the GPU architecture and CUDA programming. CUDA, short for Compute Unified Device Architecture, is the parallel programming model for NVIDIA's general-purpose GPUs. The architecture of a typical CUDA-capable GPU is consisted of an array of highly threaded streaming multiprocessors (SM) and comes with up to a huge amount of DRAM, referred to as global memory. Take the Tesla C2070 GPU for example. It contains 14 SMs, each of which has 32 streaming processors (SPs, or CUDA cores called by NVIDIA), 4 special function units (SFU), and its own shared memory/L1 cache. The structure of a streaming multiprocessor is shown in Fig. 2.

As the programming model of GPU, CUDA extends C into CUDA C and supports such tasks as threads calling and memory allocation, which makes programmers able to explore most of the capabilities of GPU parallelism. In CUDA programming model, illustrated in Fig. 3, threads are organized into blocks; blocks of threads are organized as grids. CUDA also assumes that both the host (CPU) and the device (GPU) maintain their own separate memory spaces, which are referred to as host memory and device memory respectively. For every block of threads, a shared memory is accessible to all threads in that same block. The global memory is accessible to all threads in all blocks. Developers can write programs running millions of threads with thousands of blocks in parallel. This massive parallelism forms the reason that programs with GPU acceleration can be much faster than their CPU counterparts. CUDA C provides its extended keywords and built-in variables, such as `blockIdx.{x,y,z}` and `threadIdx.{x,y.z}`, to assign unique ID to all blocks and

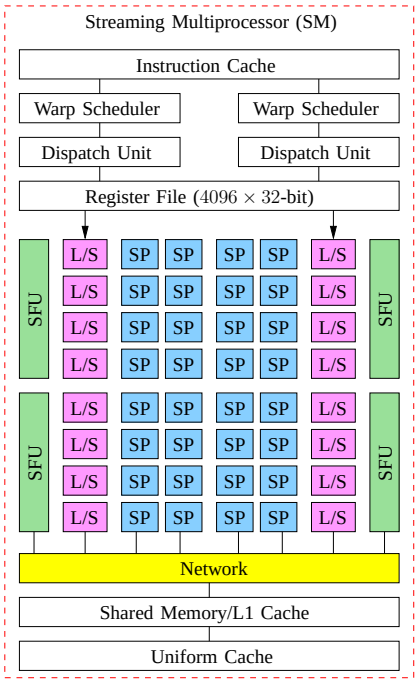

Fig. 2. Diagram of a streaming multiprocessor in NVIDIA Tesla C2070. (SP is short for streaming processor, L/S for load/store unit, and SFU for Special Function Unit.)

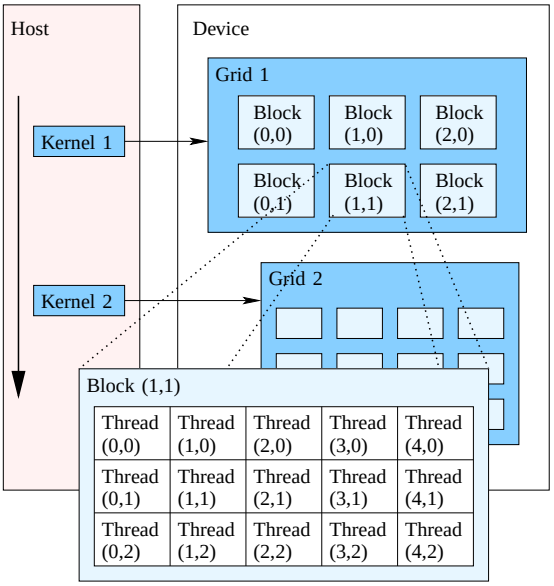

Fig. 3. The programming model of CUDA.

threads in the whole grid partition. Therefore, programmers can easily map the data partition to the parallel threads, and instruct the specific thread to compute its own responsible data elements. Fig. 3 shows an example of 2-dim blocks and 2-dim threads in a grid, the block ID and thread ID are indicated by their row and column positions.

III. Proposed GLU solver based on the hybrid column-based right-looking LU method on GPU platforms

In this section, we explain our new hybrid column-based right-looking sparse LU factorization method on the GPUs – GLU solver. GLU solver was originally inspired by the observation that the existing left-looking LU factorization has inherent limitations for concurrency exploitions due to the required solving of triangular matrices. To mitigate this problem, we look at the traditional right-looking LU factorization method, which seems more amenable for parallelization especially on GPU platforms. But we also want the benefits of symbolic analysis for storage management of factorized matrices and column-level concurrency in the left-looking based method. The resulting method is the hybrid column-based right-looking LU method, which will be discussed in the following.

A. The column-based right-looking algorithm

Our starting point is still the left-looking algorithm as we want to keep the column-concurrency and symbolic analysis and we still compute one column for both L and U matrices. But unlike the left-looking algorithm, once a column of L is computed, its impacts on the yet-to-be-solved columns will be updated right away (so we now start to look right in this sense). Algorithm 2 shows the hybrid column-based right-looking LU factorization algorithm, which turns out to be more amenable for GPU parallelization. In this pseudo code, the current column are indexed by k, and the columns to the right of the current column, which are updated immediately after the current column has been computed, are indexed by j. After current column k is computed, the algorithm looks right and finds all column j ($j > k$) in the submatrix, where $A_s(k, j) \neq 0$, and then uses column k to update these columns. $A_s(x, y)$ indicates the LU symbolically factorized A matrix, where all the fill-ins and non-zero elements are assigned with non-zero initial values and their memories are allocated. Fig. 4 illustrates the basic idea of the hybrid column-based right-looking algorithm. The key operation of the right-looking algorithm becomes submatrix update now. But such change makes a major difference in terms of concurrency exploitation as we will show soon.

In this column-based right-looking algorithm, we still have three loops. The outer k-loop chooses the current column k that will be factorized; the middle j-loop chooses the column j in the submatrix right to column k that depends on column k; and the inner i-loop is used to perform MAD operations between column k and column j. But now, we will show that these three loops can be parallelized because the submatrix is updated by the i and j loops, which is more amenable for parallelization than the solving triangular matrices in the left-looking method (see details in Subsection III-C).

We remark the hybrid LU factorization method is similar to the multifrontal based right-looking LU factorization method, in the sense that each independent column and its connected columns can form a frontal matrix [24]. But in our approach,

Algorithm 2 The hybrid column-based right-looking algorithm

```
1: for k = 1 to n do
2:    /*Compute column k of L matrix*/
3:    for i = k + 1 to n where A_s(i,k) ≠ 0 do
4:       A_s(i,k) = A_s(i,k)/A_s(k,k)
5:    end for
6:    /*Update the submatrix for next iteration*/
7:    for j = k + 1 to n where A_s(k,j) ≠ 0 do
8:       for i = k + 1 to n where A_s(i,k) ≠ 0 do
9:          A_s(i,j) = A_s(i,j) − A_s(i,k) * A_s(k,j)
10:      end for
11:   end for
12: end for
```

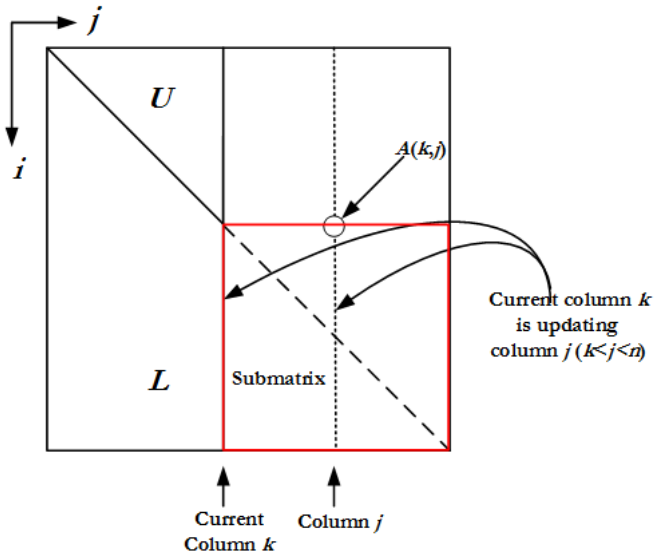

Fig. 4. The illustration of the hybrid column-based right-looking algorithm and the submatrix update at each iteration

no elimination tree is used to build the frontal matrices and the hierarchical matrix analysis structure. The column-level parallelization is mainly based on the dependency graph (to be discussed later).

B. Preprocessing and symbolic analysis

As we mentioned earlier that the proposed method combines the benefits of both left-looking method and the right-looking methods. As a result, it still follows the preprocessing and symbolic analysis steps to improve the factorization efficiency. Hence the new factorization algorithm can still be split into three phases. In the sequel, we give a brief description of the first two steps for the self-contained purpose. Then we analyze the related data dependency from the symbolic analysis step for GPU computing.

First, the preprocessing phase preorders the matrix A to minimize fill-in and to ensure a zero-free diagonal. Second, the symbolic phase performs symbolic factorization and determines the structure of lower triangular matrix L and upper triangular matrix U. Then it groups independent columns into levels. Third, the numerical phase obtains the resulting lower and upper sparse triangular factors by solving the columns level by level. The preprocessing phase and symbolic phase are performed only once on CPU (which will be discussed in this section). The numerical phase can be performed multiple times on GPU. For the completion of the algorithm, we also present the first two phases.

In the preprocessing phase, we use HSL MC64 [25] to decrease the likelihood of encountering tiny pivots and AMD (Approximate Minimum Degree) algorithm [26] to reduce the fill-ins. The nonzero structure of the sparse matrix may dramatically change in course of LU factorization. In this step, we perform a left-looking algorithm based symbolic analysis [13] to determine the nonzero patterns of L and U. The core operation of left-looking algorithm is to solve the lower triangular system $L_k x = b$ in order to compute the kth column, where L_k is lower matrix representing the already computed $(k-1)$ columns and the vector b is the kth column of matrix A. This pseudo core operation is shown in Algorithm 3.

Algorithm 3 Forward substitution for solving sparse triangular matrices

```
1: x = b
2: for j = 0 to k − 1 where b(j) ≠ 0 do
3:    for i = j + 1 to n where L(i,j) ≠ 0 do
4:       x(i) = x(i) − L(i,j)x(j)
5:    end for
6: end for
```

From the pseudo code, we can see that entries in x can become nonzero in only two places, the first and fourth lines. We can represent these two relationships as a directed graph $G = (V, E)$, where the nodes $V = 1...n$ represent the rows and the edges $E = (j, i)$ where $L(i, j) \neq 0$. Thus, line 1 is equivalent to marking all nodes that are nonzeros in the vector b, whereas line 4 implies that if a node j is marked and it has an edge to a node i, then the latter must be also marked. Fig. 5 graphically highlights these two relationships. Therefore, the nonzero pattern can be computed by determining the nodes that are reachable from the nodes of vector b, which is also the computed column vector of U from the previous iteration of the left-looking method. This reachability problem can be solved using a classical depth-first search in G. Then we can determine the nonzero pattern of the new matrices L and U.

Fig. 6 shows a sparse matrix A and the predicted nonzero pattern of the LU factors of A after symbolic analysis (L and U share the same space of A), in which the black circle and white circle represent original entries of A and fill-in entries respectively. As a result, the matrix A now contains the fill-ins and elements in the original matrix. When we copy A to GPU's memory, we actually copy the L and U matrices (with their values yet to be determined). After the factorization, the A becomes the resulting L and U matrices physically and it doesn't contain the original matrix any more.

Another important problem is the column dependencies. It is clear that any column dependencies in the overall left-looking algorithm only arise from the *sparse triangular solve* step, the

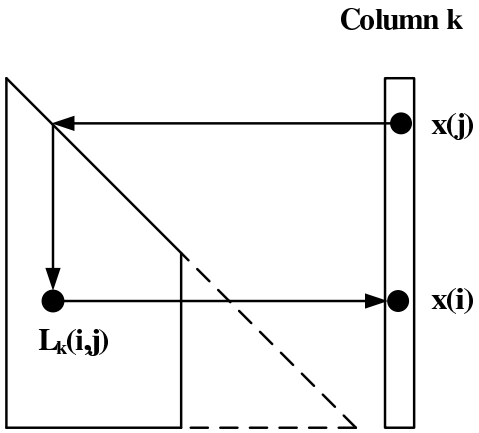

Fig. 5. Nonzero pattern for a sparse triangular solver.

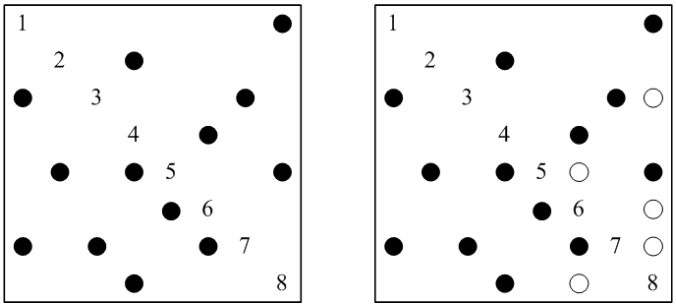

Fig. 6. The original matrix A (left) and the matrix A (right) after symbolic analysis with predicted nonzero pattern of LU factors of A

line 2 of Algorithm 3. However, when we compute column k, not all the columns to its left are needed, as it was illustrated in Algorithm 3. In fact, the factorization of column k only depends on the columns that satisfy $a_{ij} \neq 0$ for $i < j$. In other words, the dependencies between rows are defined by the sparsity pattern of the upper triangular matrix U and are independent of the lower triangular matrix L.

We use a directed acyclic graph (also called dependency graph) to represent the data dependencies in the LU factorization of the matrix in Fig. 6. In which, if column k depends on column i, then a directed edge exists from node i to node k, where $i < k$. Fig. 7 (top figure) illustrates the column dependencies of example matrix A. The graph was computed using predicted nonzero structure of matrix U only. All the columns in the same level are independent and can be computed in parallel. For instance, column 1, 2, 3, 5 can be evaluated in parallel; however, column 6 cannot be processed until columns 4 and 5 are computed.

Note that the concurrent computation resources (warps per streaming multiprocessor (SM), shared memory per block, threads per block) on GPU are limited. As a result, the number of columns, which can be solved in parallel, in each level should be limited. Hence, we propose a resource-aware levelization scheme, in which the number of columns of each level will be limited by a fixed number. For instance, Fig. 7 (bottom figure) shows the levelization result from the top figure in which the maximum number of allowed columns is 3. This resource-aware levelization scheme will be applied to parallelize the outer k-loop of the proposed right-looking algorithm.

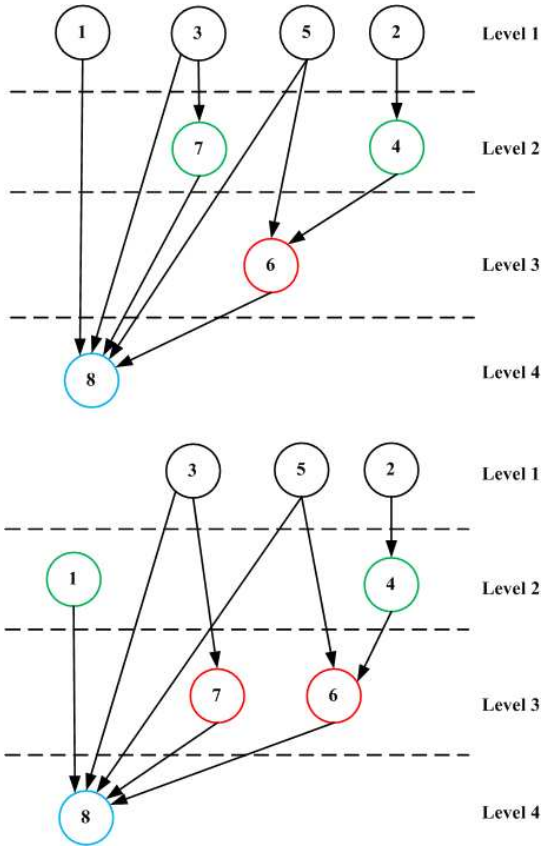

Fig. 7. The illustration of the resource-aware levelization scheme.

C. Numerical computing phase

The algorithm based symbolic analysis can be altered to expose the column-level parallelism. Despite this exposed column-evaluation concurrency, in the numerical factorization phase we can explore more parallelism available in each level using multiple threads.

On GPU, the global memory access from the same CUDA warp (one warp consists of 32 threads and it is scheduling unit on one SM on GPU) can be coalesced if they are visiting the consecutive memory address. However for sparse LU factorization, irregular nonzero pattern leads to many uncoalesced global memory accesses, which greatly degrades the performance. To maximize the coalescence, we use compressed sparse column (CSC) format to store the A matrices (L and U share the same storage of A) and record all nonzeros in L and U. In addition, to maximize parallelization during the factorization, we also use compressed sparse row (CSR) format to record the nonzero positions of symbolic U (but not its values), and its usage will become clear soon.

Now let us look at how the three loops in the proposed right-looking method can be parallelized in GPU platforms. Algorithm 4 is the pseudo code for the proposed parallel column-based right-looking algorithm. The first loop is to choose a number of columns of L matrix in one level, which

can be factorized in parallel. Both the proposed method and the left-looking method enjoy this *column-level parallelism* as the proposed method is also based on the symbolic left-looking level analysis. The difference is in the other two loops of the two algorithms. Next, let us look at the computing steps inside the first loop (between line 2 and line 14). There are two stages. In the first stage, we compute the current column col of the L matrix by vector-scalar division and it can be performed in parallel easily. Due to the first column-level parallelism, we may have several current column col's to be updated.

In the second step, we perform the submatrix update (MAD operations) for the current column col. We are concerned col's that are needed by other columns of the submatrix (means that $A(k, j) \neq 0$ in line 7 of Algorithm 2) . We call the columns in submatrix which depends on current column col, the $subcol$. To facilitate locating those $subcol$'s, we need to access the dependency graph, which is represented by the symbolic upper triangular matrix U. For instance, the $subcol$'s of a current column, say k, can be found by using the nonzero position information of row k of L. This also explains why we need to have symbolic U in the CSR format. Note that the current column col needs to be stored into a un-compressed array and the $subcol$'s can access the un-compressed array to get the col information to update themselves. Since the $subcol$'s only read information from the un-compressed column, there's no conflict.

Notice that each $subcol$ only needs to be updated once by the current column. As a result, all $subcol$'s in one submatrix can be updated in parallel. This parallelism in the loop is called *submatrix update parallelism*. In the third loop, the core operation is vector MAD operation, which is used to update a $subcol$. In contrast, the current column col needs to be updated by all solved and relevant columns to its left in the left-looking algorithm in the left-looking algorithm and the updates to column k must be performed sequentially. Hence it cannot enjoy the *submatrix update parallelism*. As a result, we parallelize essentially all the loops in LU factorization in the proposed new method.

Algorithm 4 Parallel column-based right-looking algorithm on GPU

```
1:  for level 1 to level m do
2:    /*column-level parallelism*/
3:    for all col's in current level in parallel do
4:      compute current col of L matrix
5:    end for
6:    synchronize threads
7:    for all col's in current level in parallel do
8:      /*submatrix update parallelism*/
9:      for all subcol's in current submatrix which depends
        on col in parallel do
10:       /*vector MAD operation parallelism*/
11:       update elements in one subcol
12:     end for
13:   end for
14:   synchronize threads
15: end for
```

D. Parallel implementation on GPU

In parallel implementation of sparse LU factorization, the CPU is responsible for initializing the matrix and doing the symbolic analysis. GPU only concentrates on numerical factorization. CPU is also responsible for allocating device memory, copy host inputs to device memory, and copy the computed device results back to the host.

In the proposed algorithm, the matrix A is divided into several levels. With the resource-aware levelization scheme, the optimal number of thread blocks can be easily determined (see experimental section for some study results). We use one thread block to process one column in a level. In the first stage, we use one warp to compute one current column in the L matrix. Multiple current columns will lead to multiple blocks invoked in GPU in each kernel launch as each block can execute independently. Then we synchronize the threads within the block to ensure the current column is solved. In the second stage, we use one warp to update one $subcol$. There may be multiple active warps in one block now. Within one block, there is no data conflict among warps because they update different $subcol$. However, memory access conflict may occur between different blocks. For example, column j depends on both column k_1 and k_2, while column k_1 and k_2 are in the same level. In this case, the updates to the column j must be performed using atomic floating point operations. Finally, each element in $subcol$'s is updated by one thread. In this way, we can take the full advantage of the GPU powers. Fig. 8 illustrates the difference for concurrency exploitation and warp scheduling schemes between the left-looking and the proposed column-based right-looking algorithm. It can be seen that there is only one warp sequentially updating current column with the already factored columns in the left-looking algorithms. However in the proposed right-looking algorithm, multiple warps can use the current column to update many different sub-columns concurrently.

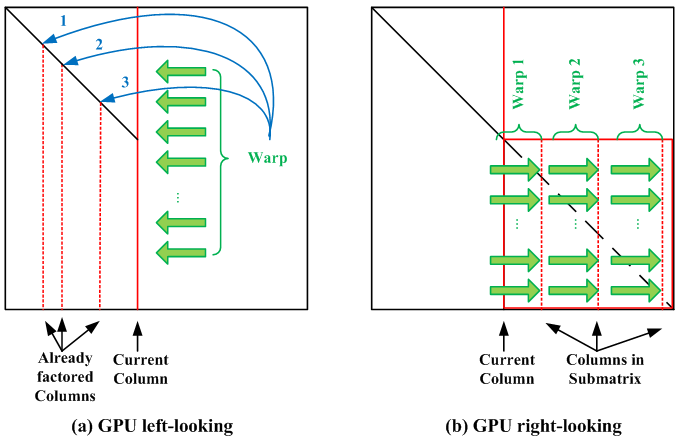

Fig. 8. The comparison of the concurrency exploitation on GPU in terms of warp scheduling.

IV. Numerical Results and Discussions

The proposed *GLU* LU factorization algorithm is implemented in C programming language. The GPU part is incorporated into the main program with CUDA C programming

TABLE II
THE PERFORMANCE COMPARISON OVER BENCHMARK MATRICES

Bechmark name	GLU Runtime (s)	PARDISO T=1 (s)	PARDISO T=16 (s)	KLU Runtime (s)	UMFPACK Runtime (s)	GPU-LL Runtime (s)	Speedup over PARDISO T=1	Speedup over PARDISO T=16	Speedup over KLU	Speedup over UMFPACK	Speedup over GPU-LL
General matrices											
cage11	5.875	23.778	3.180	781.360	163.240	16.840	4.04	0.54	133.00	27.79	2.87
cant	3.662	3.543	0.491	33.741	18.1	11.03	0.97	0.13	9.21	4.94	3.01
barrier2-11	7.434	38.622	4.491	475.690	204.059	23.667	5.20	0.60	63.99	27.45	3.18
FEM_3D_thermal2	0.898	5.658	0.864	85.201	40.94	2.583	6.30	0.96	94.88	45.59	3.71
thermomech_dK	0.053	1.501	0.274	7.090	4.527	0.238	28.32	5.17	133.77	85.42	4.49
mc2depi	0.049	0.587	0.325	25.970	20.986	0.197	11.97	6.63	530.00	428.29	4.02
epb3	0.043	0.210	0.051	0.652	0.560	0.047	4.88	1.19	15.16	13.02	1.09
apache1	0.490	2.318	0.415	14.241	7.437	0.560	4.73	0.85	29.06	15.17	1.14
ecology2	6.940	6.604	0.928	38.131	25.330	8.990	0.95	0.13	5.49	3.64	1.30
thermal2	0.066	7.668	1.043	0.466	0.943	0.112	116.18	15.80	7.06	14.29	1.70
Arithmetic mean							18.36	3.20	102.16	66.56	2.65
Geometric mean							6.38	1.12	39.39	24.04	2.35
Circuit matrices											
circuit_2	0.008	0.002	0.004	0.004	0.006	0.011	0.25	0.50	0.54	0.78	1.43
rajat15	0.163	0.090	0.020	0.293	0.447	0.203	0.55	0.12	1.79	2.74	1.24
bcircuit	0.009	0.052	0.013	0.065	0.205	0.030	5.78	1.44	7.22	22.78	3.33
ASIC_100ks	0.031	0.292	0.090,	1.660	1.871	0.032	9.42	2.90	54.25	61.15	1.05
hcircuit	0.009	0.048	0.018	0.030	0.253	0.014	5.33	2.00	3.33	28.11	1.55
scircuit	0.056	0.13	0.031	0.339	0.829	0.106	2.32	0.55	6.05	14.80	1.89
raj1	0.189	0.355	0.078	73.842	125.799	0.211	1.88	0.41	390.69	665.60	1.12
ASIC_320ks	0.058	1.328	0.264	3.703	9.156	0.081	22.90	4.55	63.75	157.63	1.39
rajat30	1.234	6.309	1.468	0.317	230.59	1.864	1.19	0.26	15.58	186.86	1.51
ASIC_680ks	0.054	20.246	2.526	1.298	4.679	0.070	374.93	46.78	24.09	86.86	1.30
G3_circuit	0.672	26.464	3.467	516.882	133.358	1.054	39.38	5.16	769.16	198.44	1.57
Freescale1	0.235	4.004	0.689	13.486	67.414	0.284	17.04	2.93	57.45	287.20	1.21
Arithmetic mean							40.08	5.63	116.16	142.74	1.55
Geometric mean							5.71	1.46	19.56	47.13	1.47
Self-generated general RLC mesh networks											
rlc1	0.080	0.730	0.326	0.558	1.746	0.135	9.13	4.08	7.02	21.96	1.70
rlc2	0.180	1.567	0.419	1.263	6.499	0.267	8.71	2.33	7.04	36.21	1.49
rlc3	0.213	2.493	0.684	1.504	6.892	0.359	11.70	3.21	7.05	32.39	1.69
rlc4	0.626	5.770	1.545	4.638	**fail**	0.996	9.22	2.47	7.41	-	1.59
rlc5	1.274	15.539	3.279	9.807	**fail**	1.985	12.20	2.57	7.70	-	1.56
Arithmetic mean							10.19	2.93	7.24	30.19	1.61
Geometric mean							10.09	2.87	7.24	29.53	1.60

Note: the GPU time includes the numeric factorization time and the time cost of data transfers between host and device.

interface. The proposed method has been prototyped in CUDA 5.0 and the experimental results are carried out in a Linux server with two 8-Core Xeon E5-2670 CPUs, DDR3-1600 64GB memory. The server also consists of one K40 GPU and one K20 GPU, which serve as the GPU platforms for the proposed algorithms. Note that all the GPU results are obtained from the K40 GPU.

A. Performance comparisons

The benchmark matrices are listed in Table I. The general matrices set and the circuit matrices set are from University of Florida Sparse Matrix Collection [23], which are used to evaluate the proposed GPU sparse LU factorization against other LU solvers. We also include a set of self-generated RLC mesh networks, which are used for providing comparison results on large circuit matrices. In this table, N is the matrix size, *NNZ(A)* means the number of nonzeros of the original matrix A, *NNZ(A)/N* shows the average number of nonzeros per row and *NNZ(L+U-I)* shows the number of nonzeros of the L and U matrices. Within each set, they are ranked with increasing number of N from top to bottom. Although our intention is for circuit matrices, we also include some matrices from wide applications to show that this GLU sparse solver can be applied for wide scientific and engineering applications.

We compare the proposed *GLU* solver against the recently proposed GPU left-looking algorithm (GPU-LL) [20], the UMFPACK solver [22], which is a righ-looking multi-frontal solver, the KLU solver [14] and PARDISO [21], which is a state of the art parallel sparse LU solver, over the benchmark matrices in Table I. In our performance evaluation, we use the CPU time reported by UMFPACK 5.6.2, KLU 1.2.0 and PARDISO 5.0.0.

Table II summarizes the performance comparison results over benchmark matrices. And the listed time is only for numeric factorization, excluding preprocessing and symbolic analysis because the numeric factorization can be done many times in circuit simulation and consume most of the simulation time while the other steps are not significant. For UMFPACK and KLU, it is difficult to see which one is better: KLU performs relatively better for circuit matrices, but are slower for matrices which take longer time to factor; UMFPACK beats KLU for general matrices which cost hundreds seconds in their factorization, but is much slower for circuit matrices; single-threaded PARDISO performs very well on general matrices

TABLE I
THE BENCHMARK MATRICES

Name	N	NNZ(A)	NNZ(A)/N	NNZ(L+U-I)
General matrices				
cage11	39082	59722	14.3	165367120
cant	62451	4007383	64.2	57830341
barrier2-11	115625	3897557	72.1	193180399
FEM_3D_thermal2	147900	3489300	23.6	93097092
thermomech_dK	204316	2846228	13.9	29085006
mc2depi	525825	2100225	4.0	54346411
epb3	643994	6175377	9.6	31698731
apache1	715176	4817870	3.9	8679783
ecology2	999999	4995991	5.0	45752523
thermal2	1228045	8580313	4.0	6330643
Circuit matrices				
circuit_2	4510	21199	4.7	35612
rajat15	37261	443573	11.9	2028124
bcircuit	68902	375558	5.5	982513
ASIC_100ks	99190	578890	5.8	4271846
hcircuit	105676	513072	4.8	625958
scircuit	170998	958936	5.6	2518316
raj1	263743	1302464	4.9	10771367
ASIC_320ks	321671	1827807	5.7	4838888
rajat30	643994	6175377	9.6	31698731
ASIC_680ks	682712	2329176	7.2	4957172
G3_circuit	1585478	7660826	4.8	376618798
Freescale1	3428755	17052626	5.0	61281350
Self-generated RLC mesh networks				
rlc1	1970204	5930208	3.0	10169818
rlc2	3940404	11900408	3.0	21669968
rlc3	5890604	15731208	2.7	29761652
rlc4	15880404	47720408	3.0	81367568
rlc5	35621204	95082408	2.7	180026381

which are more dense than circuit matrices; The 16-threaded PARDISO is very fast on some of the cases but not so impressive on some large circuit matrices like *ASIC_680ks* and all of our self-generated RLC mesh networks; GPU-based left-looking solver, GPU-LL, has very stable performance on all of the three sets.

The proposed GLU algorithm outperforms the above solvers on most the matrices cases with various structures. For the general matrices set, compared to the KLU and UMFPACK, our speedup can achieve 39.4× and 24.0× on geometric mean, respectively. In addition, we can see in some cases such as *mc2depi*, the speedup over KLU solver can be as high as 530×. Compared to the single-threaded and the 16-threaded PARDISO solver, the speedup can be 6.38× and 1.12× on geometric mean, respectively. Compared to the GPU-LL solver, the speedup ranges from 1.09× to 4.49×, with 2.35× on geometric mean, which is still quite significant as the new solver is faster for all the matrices. On the other hand, we also notice that the speedup highly depends the structures of benchmark matrices.

Speedup in some cases such as *barrier2-11* is due to the fact that there are many denormal floating point numbers (extremely small real numbers) when factorizing this kind of matrix. CPU deals denormal numbers much slower than with normal represented numbers [27]. In contrast, the GPU can handle these numbers at the same speed as normal numbers [20]. So the GPU speedups for these matrices are very high. The performance comparison on these matrices clearly demonstrates the advantage of the proposed method.

We then compare the proposed method against other solvers on a set of typical circuit matrices which are also from UFL sparse matrix collection [23]. Compared with the single-threaded and the 16-threaded PARDISO, KLU, UMFPACK and GPU-LL, the proposed method achieves about 5.71×, 1.46×, 19.56×, 47.13× and 1.47× speedup, respectively, which further validates the advantage of the proposed method over the existing LU solvers. Please note that the proposed GLU can be slower than other solvers on very small circuit like *circuit_2* and *rajat15*. The possible reason is that the computation time is quite small and overheads become more significant, which was also observed in [28].

Last, we perform the comparison on a set of self-generated general RLC meshed networks. We notice that the KLU solver is highly optimized for the such circuit matrices and it indeed shows better performance. But still the 16-threaded PARDISO gives the best results among all the CPU sparse solver. However the new GLU method outperforms KLU about 7.24× on average. And the UMFPACK solver runs out of memory on two largest matrices. For the other 3 matrices, the proposed method also delivers about 29.53× speedup compared to UMFPACK solver. Also GLU outperforms single-threaded and 16-threaded PARDISO with 10.09× and 2.87× on average. Compared with the GPU-LL solver, GLU achieves about 1.60× speedup and it again consistently outperform the GPU-LL solver on all the examples, which further demonstrates the advantage of the proposed method over the existing GPU-LL method.

B. Impacts of warp number on performance

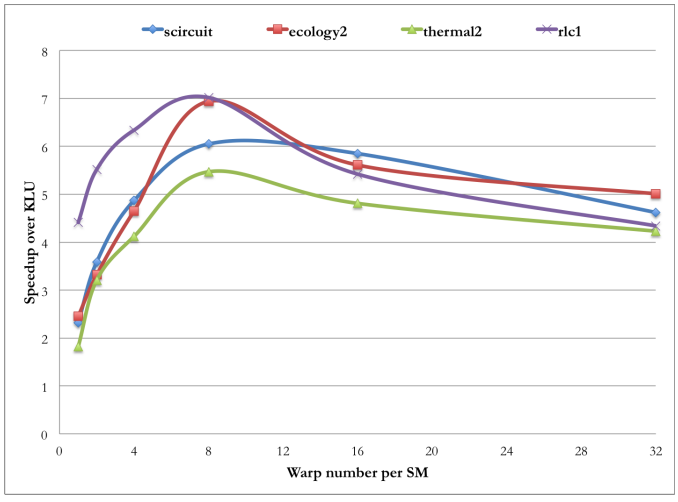

Fig. 9. Speedup over KLU vs. number of warps per SM on K40c.

Next, we study one important design parameter and its impacts on performance of the GLU solver. We observe that one important parameter for the proposed solver is the number of warps allowed for each block or streaming multiprocessor (SM). Fig. 9 shows the speedup over KLU on four matrices on K40 GPU, with different number of warps per SM. The best performance is achieved when the number of resident warps per SM is around 8.

As we mentioned in Section III-D, we use one warp to process one sub-column. Although more active warps can attain a higher parallelism, processing too many sub-columns simultaneously may decrease the performance. There are several reasons for this. First, there may not be enough sub-columns as the matrix is sparse, while more resident warps means more overhead on SM. Second, it may lead to more memory access conflicts as the threads need to perform slow atomic operations. With a threshold warp number set to 8, the performance reaches its best in terms of those trade-offs. When we further increase the number of warp, although there are more threads, the parallelism will not be further exploited so the performance starts to degrade. Note that such golden number of warp number depends on the structure of sparse matrix, and a few tries are needed to find the optimal number.

V. Conclusion

We have proposed a new sparse LU solver on GPUs for circuit simulation and more general scientific computing. The new algorithm is based on a hybrid right-looking LU factorization method, which we showed, is more suitable for GPU computing as it can exploit more parallelism than the widely used left-looking LU factorization algorithm. We further showed how the three loops of LU factorization can be parallelized based on the GPU thread structures, while the existing GPU left-looking LU factorization method can only parallelize two loops. Numerical results show that the proposed GLU solver can deliver 5.71× and 1.46× speedup over the single-threaded and the 16-threaded PARDISO solvers respectively, 19.56× speedup over the KLU solver, 47.13× over the UMFPACK solver and 1.47× speedup over a recently proposed GPU-based left-looking LU solver on the set of typical circuit matrices from University of Florida Sparse Matrix Collection (UFL). Furthermore, we also compare the proposed GLU solver on a set of general matrices from UFL, GLU achieves 6.38× and 1.12× speedup over the single-threaded and the 16-threaded PARDISO solvers respectively, 39.39× speedup over the KLU solver, 24.04× over the UMFPACK solver and 2.35× speedup over the same GPU-based left-looking LU solver. Also comparison on self-generated RLC mesh networks shows a similar trend, which further validates the advantage of the proposed method over the existing sparse LU solvers.

VI. acknowledgement

The authors would like to thank Hengyang Zhao for providing some comparison data from the PARDISO solver.

References

[1] Intel Corporation, “Intel multi-core processors, making the move to quad-core and beyond (White Paper),” 2006. http://www.intel.com/multi-core.
[2] AMD Inc., “Multi-core processors—the next evolution in computing (White Paper),” 2006. http://multicore.amd.com.
[3] S. Borkar, “Thousand core chips: a technology perspective,” in *Proc. Design Automation Conf. (DAC)*, pp. 746–749, 2007.
[4] NVIDIA Corporation, 2011. http://www.nvidia.com.
[5] D. B. Kirk and W.-M. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach, 2ed*. San Francisco, CA: Morgan Kaufmann Publishers Inc., 2013.
[6] “NVIDIA Tesla’s Servers and Workstations.” http://www.nvidia.com/object/tesla-servers.html.
[7] D. Göddeke, “General-purpose computation using graphics harware.” http://www.gpgpu.org/, 2011.
[8] NVIDIA Corporation, “CUBLAS library v5.0.” https://developer.nvidia.com/cublas.
[9] X. Liu, S. X.-D. Tan, and H. Yu, “A gpu-accelerated parallel shooting algorithm for analysis of radio frequency and microwave integrated circuits,” *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 23, March 2015.
[10] X. Liu, K. Zhai, Z. Liu, K. He, S. X.-D. Tan, and W. Yu, “Parallel thermal analysis of 3D integrated circuits with liquid cooling on CPU-GPU platforms,” *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, pp. 575–579, March 2015.
[11] “http://crd.lbl.gov/ xiaoye/superlu/.”
[12] J. W. Demmel, J. R. Gilbert, and X. S. Li, “An asynchronous parallel supernodal algorithm for sparse gaussian elimination,” *SIAM J. Matrix Analysis and Applications*, vol. 20, no. 4, pp. 915–952, 1999.
[13] J. R. Gilbert and T. Peierls, “Sparse partial pivoting in time proportional to arithmetic operations,” *SIAM J. Sci. Statist. Comput.*, pp. 862–874, 1988.
[14] T. A. Davis and E. P. Natarajan, “Algorithm 907: KLU, a direct sparse solver for circuit simulation problems,” *ACM Trans. Mathematical Software*, pp. 36:1–36:17, September 2010.
[15] X. Chen, W. Wu, Y. Wang, H. Yu, and H. Yang, “An escheduler-based data dependence analysis and task scheduling for parallel circuit simulation,” *IEEE Trans. on Circuits and Systems II: Express Briefs*, pp. 702–706, October 2011.
[16] X. Chen, Y. Wang, and H. Yang, “NICSLU: An adaptive sparse matrix solver for parallel circuit simulation,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, pp. 261–274, February 2013.
[17] N. Galoppo, N. Govindaraju, M. Henson, and D. Manocha, “LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware,” in *2005 Proceedings of the ACM/IEEE Supercomputing (SC) Conference*, p. 3, 2005.
[18] S. Tomov, J. Dongarra, and M. Baboulin, “Towards dense linear algebra for hybrid GPU accelerated manycore systems,” *Parallel Computing*, vol. 36, pp. 232–240, June 2010.
[19] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, “Dense Linear Algebra Solvers for Multicore with GPU Accelerators,” in *Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pp. 1–8, IEEE, 2010.
[20] L. Ren, X. Chen, Y. Wang, C. Zhang, and H. Yang, “Sparse LU factorization for parallel circuit simulation on GPU,” in *Proc. Design Automation Conf. (DAC)*, pp. 1125–1130, 2012.
[21] O. Schenk, A. Waechter, and M. Hagemann, “Matching-based Preprocessing Algorithms to the Solution of Saddle-Point Problems in Large-Scale Nonconvex Interior-Point Optimization. Journal of Computational Optimization and Applications,” *Journal of Computational Optimization and Applications*, vol. 36, no. 2-3, pp. 321–341, 2007.
[22] “UMFPACK.” http://www.cise.ufl.edu/research/sparse/umfpack/.
[23] T. Davis, “The University of Florida sparse matrix collection.” http://www.cise.ufl.edu/research/sparse/.
[24] T. A. Davis, *Direct Methods for Sparse Linear Systems*. SIAM, Philadelphia, 2006.
[25] I. S. Duff and J. Koster, “The design and use of algorithms for permuting large entries to the diagonal of sparse matrices,” *SIAM J. Matrix Anal and Applics*, no. 4, pp. 889–901, 1997.
[26] P. R. Amestoy, Enseeiht-Irit, T. A. Davis, and I. S. Duff, “Algorithm 837: AMD, an approximate minumum degree ordering algorithm,” *ACM Trans. Mathematical Software*, pp. 381–388, September 2004.
[27] L. de Soras, “Denormal numbers in floating point signal processing applications,” 2002.
[28] X. Chen, L. Ren, Y. Wang, and H. Yang, “GPU-accelerated sparse LU factorization for circuit simulation with performance modeling,” *IEEE Trans. on Parallel and Distributed Systems*. http://doi.ieeecomputersociety.org/10.1109/TPDS.2014.2312199.