GLU3.0: Fast GPU-Based Parallel Sparse LU Factorization for Circuit Simulation

Shaoyi Peng and Sheldon X.-D. Tan
University of California at Riverside

Editor's note:
Many scientific computing problems, including circuit simulations, rely on efficient lower–upper (LU) decomposition of sparse matrices. Prior studies took advantage of GPUs to parallelize LU decomposition, but they suffer from nontrivial data dependencies. This article presents a new method, called GLU3.0, to accelerate GPU-based sparse LU factorization.
—Umit Ogras, Arizona State University

Sparse LU factorization plays a critical role in wide engineering and scientific computing applications such as solving differential and circuit equations. In particular, for a circuit simulation application such as the widely used SPICE program, the core computing or dominant computing is to solve the linear algebraic system, $\boldsymbol{Ax} = \boldsymbol{b}$, resulting from linear or nonlinear circuits with millions or even billions of extracted components. LU factorization solves these equations by transforming matrix A into two matrices: the lower triangular matrix L and upper triangular matrix U such that $A = LU$. Then, the solution x is obtained by sequentially solving the two triangular matrices. Furthermore, for all circuit simulation problems, matrix A is very large and sparse. As a result, LU factorization of large sparse matrices becomes a central problem for those analysis and simulation applications.

Digital Object Identifier 10.1109/MDAT.2020.2974910
Date of publication: 18 February 2020; date of current version: 24 June 2020.

As very large scale integration (VLSI) continues to grow in size, improving the efficiency and scalability of LU factorization continues to be a challenging problem.

The graphics processing unit (GPU) provides massive and fine-grain parallelism with higher orders of magnitude higher than the CPUs. For instance, the state-of-the-art NVIDIA Tesla V100 GPU with 5120 cores has a peak performance of over 15 TFLOPS versus about 200–400 GFLOPS of Intel i9 series 8 core CPUs. Today, in addition to gaming graphics, GPU has been widely used for more general-purpose computing such as EDA, deep learning/AI, finance, and medical and life sciences. However, parallelizing sparse LU factorization on GPU is not straightforward due to high data dependency and irregular memory access [1].

There exist some previous research studies targeting parallel sparse LU factorization on shared memory multicore CPUs. For instance, SuperLU_MT [2] is the multithreaded parallel version of SuperLU for multicore architectures. However, it is not easy to form a super-node in some sparse matrix such as circuit matrix. KLU [3], which is specially optimized for circuit simulation, adopts block triangular form based on the Gilbert–Peierls (G/P) left-looking algorithm [4] and has become one of the standard algorithms in circuit simulation applications. The KLU algorithm has been parallelized on multicore

2168-2356/20©2020 IEEE Copublished by the IEEE CEDA, IEEE CASS, IEEE SSCS, and TTTC

architecture by exploiting the column-level parallelism [5] and named *NICSLU*.

Existing GPU-based parallel LU factorization solvers mainly focus on dense matrices in [6]. There also exist a few sparse matrix LU factorization methods on GPU [7], [8]. However, these works mainly convert the sparse matrices into many dense submatrices (blocks) and then solve them by dense matrix LU factorizations. However, such a strategy may not work well for circuit matrices, which hardly have dense submatrices.

The parallel (G/P) left-looking algorithm [4] on GPU has been explored first in [9]. It exploits the column-level (called *task-level*) parallelism due to the sparse nature of the matrix- and vector-level parallelism in the sparser triangular matrix solving in the G/P method. However, the two loops in triangular matrix solving cannot be completely parallelized (from lines 4 to 8 in Algorithm 1); thus, the G/P method is difficult for fine-grain parallelization.

To mitigate this problem, He et al. [1] proposed a hybrid right-looking sparse LU factorization on GPU, called *GLU (GLU1.0)*. GLU has the benefits of the left-looking method for column-based parallelism and uses the same symbolic analysis routine. The difference is that it performs the submatrix update once one column is factorized, which is similar to the traditional right-looking LU method. However, GLU1.0 used a fixed scheme to allocate the GPU threads and memory, which limits its parallelism. Furthermore, the right-looking feature of GLU actually introduces new data dependency (called *double-U dependency* in this article), which has been reported in GLU2.0 and [10]. Double-U dependency can lead to inaccurate results for some test cases. Detection of double-U dependency was added into GLU2.0 to fix this issue, which, however, incurred some performance degradation compared to GLU1.0. Recently, Lee et al. [10] proposed an enhanced GLU2.0 solver, which considers the column count difference in different levels, and exploits some advanced GPU features such as dynamic kernel launch to further improve the GLU kernel efficiency. However, the fixed GPU threads and memory allocation method from GLU2.0 for each kernel launch are still used and have limited performance.

In this article, we propose a new version of a GPU-based sparse LU factorization solver, called *GLU3.0*[1] for circuit simulation and more general scientific computing. It is based on existing GLU1.0/2.0 using a hybrid right-looking LU factorization algorithm. The main improvements of GLU3.0 are summarized as follows.

- First, to mitigate the slow process to detect the new double-U data dependency in existing GLU2.0 solver, GLU3.0 introduces a new dependency detection algorithm. It uses a relaxed principal to find all required dependencies, plus some redundant ones. The efficiency is much higher than the previous solution with little impact on performance.
- Second, we have observed a pattern of potential parallelism along with the matrix factorization, based on the analyzed circuit matrices. Basically, the number of columns and its associated subcolumns (updates) of each column, which are important parallel computing task units, are inversely correlated. As a result, we can use the number of columns as a good metric for resource allocation. We have developed three different modes of GPU kernel that adapt to different stages to accommodate the computing task changes. As a result, GLU3.0 can dynamically allocate GPU threads and memory based on the number of columns in a level to better balance the computing demands and resources during the LU factorization process.

Numerical results on circuit matrices from the University of Florida Sparse Matrix Collection (UFL) show that the GLU3.0 can deliver 2–3 orders of magnitude speedup over GLU2.0 for the data dependency detection. Furthermore, GLU3.0 *consistently* outperforms both GLU 2.0 and the recently proposed enhanced GLU2.0 sparse LU solver on the same set of circuit matrices. Furthermore, GLU3.0 achieves 13.0 × (arithmetic mean) or 6.7 × (geometric mean) speedup over GLU 2.0 and 7.1 × (arithmetic mean) or 4.8 × (geometric mean) over the recently proposed enhanced GLU2.0 sparse LU solver on the same set of circuit matrices.

Review of LU factorization and CUDA

In this section, we briefly review the traditional G/P left-looking method for sparse matrices LU factorization [4] and the recently proposed hybrid

[1]GLU 3.0 source codes and documents are available at https://intra.ece.ucr.edu/stan/project/glu/GLU/.

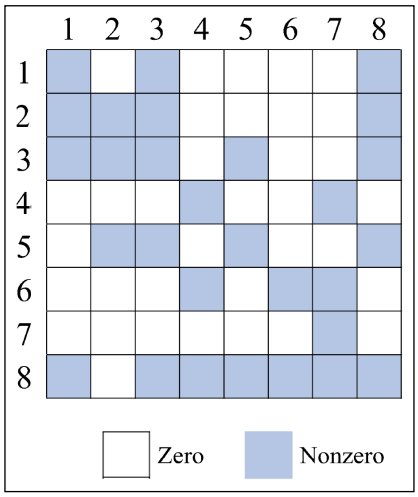

Figure 1. Example matrix.

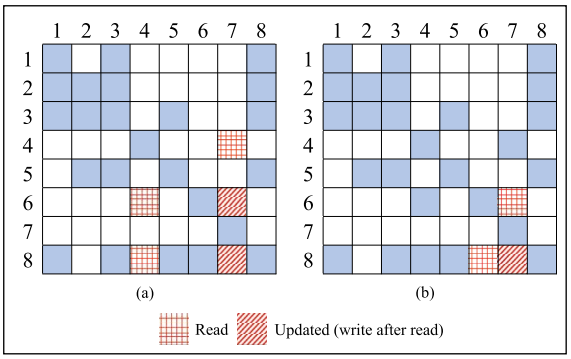

Figure 2. Two update iterations completing factorization of the seventh column (j = 7): (a) update using the fourth column (k = 4) and (b) update using the sixth column (k = 6).

right-looking algorithm used in GLU1.0, GLU2.0, and a recent GLU enhancement work [10].

We would also briefly review the GPU architectures and NVIDIA Compute Unified Device Architecture (CUDA) programming system.

An example matrix is used for illustrating important concepts and algorithms in the following discussion. The matrix is shown in Figure 1, where the colored spots denote nonzero elements.

Left-looking factorization method

The traditional Gaussian elimination-based LU factorization method (also called the *right-looking method*) solves one row for the U matrix and then one column for the L matrix in each iteration, whereas the G/P left-looking method computes one column in one iteration for both L and U instead, which is achieved by solving a lower triangular matrix. It also allows the symbolic fill-in analysis of L and U matrices before the actual numerical computing. As a result, the G/P left-looking method shows better performance for sparse matrices, especially those from circuit simulations [11].

Algorithm 1 G/P left-looking algorithm

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

Algorithm 1 shows the detailed implementation of G/P left-looking LU factorization [4]. The input of this algorithm A_s is the nonzero filled-in matrix of A after symbolic analysis. The matrix A_s is a factorized column by column (the outer j loop), and factorizing each column for both L and U contains two steps. The first step (lines 4–9) is to solve a triangular matrix. In each k loop, element-wise multiply-and-accumulate (MAC) operation is done (lines 6–8) for the partial column vector A_s (k + 1 : n, j). $A_s(i, k)$ are the elements in the factorized columns on the left of the current column j. This is the reason why it is named the *left-looking LU method*. Then, the second step (lines 10–13) is a much simpler loop that finishes the factorization of this column. Triangular matrix solving (lines 4–9) is the most essential and computationally expensive step in this algorithm.

Figure 2 shows a complete example of this step. In this example, column 7 is being factorized, meaning j = 7 in Algorithm 1. Only two k's satisfy A_s (k, 7) ≠ 0 (line 4), which are 4 and 6 [as A_s (4, 7) ≠ 0 and A_s (6, 7) ≠ 0]. The two subfigures show these two iterations, respectively. Figure 2a shows that, k = 4, so that column 4 is used to update column 7. The update operation refers to lines 6–8 of Algorithm 1, where two elements of column 7 [A_s (6, 7) and A_s (8, 7)] are updated by MAC operations with

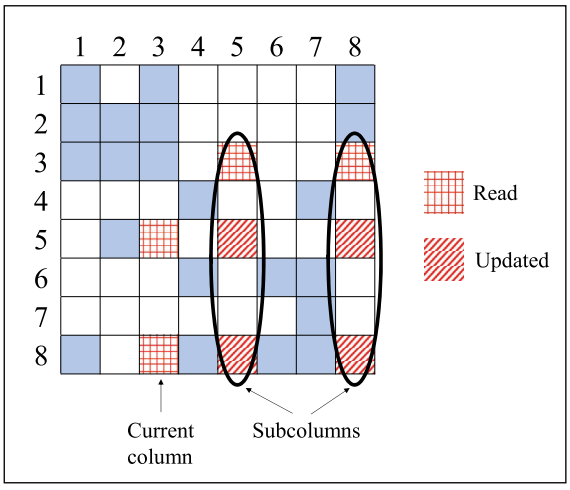

Figure 3. Subcolumns and submatrix column 3. All highlighted elements compose the submatrix, which include elements being read and elements being updated.

the red elements in column 4 multiplying $A_s(4, 7)$. Figure 2b shows the next iteration, where $k = 6$, and column 3 is used to further update column 7, which can be explicitly written as $A_s(8, 7) \leftarrow A_s(8, 7) - A_s(8, 6) * A_s(6, 7)$.

Review of the column-based right-looking algorithm used in GLU

As elaborated in [1], the G/P left-looking sparse LU factorization has one limitation that it failed to parallelize the two loops in the triangular matrix solving process (lines 4–8 of Algorithm 1). It can only work on (write) one column (current column j) at a time as indicated in line 7. To mitigate this problem, He et al. [1] proposed the hybrid column-based right-looking LU factorization algorithm for GLU. The algorithm is hybrid because it still keeps the column-based parallelism in the left-looking algorithm while updates columns on the right during factorization. Similar symbolic analysis is still applied in advance as well.

The hybrid right-looking LU factorization algorithm is listed in Algorithm 2. Similarly, the current column under computing is indexed by j. For each column, the first step is to compute the L part of the current column (lines 4–6), which is equivalent to lines 10–12 of Algorithm 1. Then, it looks right to find all columns k ($k > j$) that meet $A_s(k, j) \neq 0$ and uses the currently factorized column j to update these columns (lines 8–12). For the sake of presentation convenience without confusion, we name these columns *subcolumns* of column j. Note that these subcolumns are not part of the column j. Furthermore, this step of updating all subcolumns is called *submatrix update*, where all elements being read or updated form a *submatrix*. Figure 3 is an example of illustrating these two concepts. In the figure, $j = 3$, its subcolumns are columns 5 and 8 because $A_s(3, 5)$ and $A_s(3, 8)$ are nonzero elements. Corresponding this to the execution of Algorithm 2, during iteration $j = 3$, two k's meet the condition of line 8, which are 5 and 8.

The key difference between this right-looking algorithm and the left-looking one is that the submatrix update completes the equivalent jobs of triangular matrix solving (lines 4–9 of Algorithm 1) in advance. In the example shown in Figure 2, both update operations are completed while $j = 7$. However, in the case of the right-looking algorithm, the update in (a) is done while $j = 4$, and update in (b) is done while $j = 6$. As discussed in the following section, this difference enables exploiting parallelization between subcolumns.

Additional data dependency in GLU: The fix in GLU2.0

Data dependency is an important issue in parallel computing or general high-performance computing. It puts hard requirements in the orders of operations. In SuperLU [2] and NICSLU [5], the elimination tree has been used to resolve this issue. For GLU, to factorize several columns in parallel, data dependency between columns needs to be detected in the first place. With complete information of dependency, columns can be grouped into *levels*, where all columns at the same level are independent of each

Algorithm 2 Hybrid column-based right-looking algorithm for GLU1.0/2.0

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

other and can thus be factorized in parallel. Such a process deriving information about levels is called *levelization*, which is a similar method to the elimination tree. In the left-looking LU factorization method, levelization is done by studying the sparsity pattern of the U matrix. Any $U(i, j) \neq 0$, $i < j$ results in column j being dependent on i because of the triangular matrix solving (lines 4–9 in Algorithm 1). This dependency detection algorithm was also used in GLU1.0.

However, as reported in GLU2.0 and [10], the hybrid right-looking algorithm used in GLU leads to a new column dependency named *double-U dependency*, originating from the read–write hazard during parallel submatrix updates. An example of this can be found in columns 4 and 6 of the example matrix with the details highlighted in Figure 4. In (a), A_s (6,7) is updated by column 4: A_s (6,7) ← A_s(6, 7) − A_s (6,4) ∗ A_s (4,7). In (b), A_s(6,7) is used to update column 7: A_s (8, 7) ← A_s (8, 7) − A_s (8, 6) ∗ A_s (6, 7). In the scheme of GLU1.0, both updates are executed in parallel. However, A_s (6, 7) is written in (a) and read in (b), which forms a read–write hazard when they are executed in parallel. To ensure correctness, the write operation in (a) must finish before the read operation in (b). As a result, an additional dependency between columns 4 and 6 needs to be introduced undesirably.

Such read–write dependency is called *double-U dependency* in GLU2.0 as it originates from two overlapped U-shaped dependencies as shown in Figure 4. To detect this new dependency, GLU2.0 introduced a different dependency detection process as shown in Algorithm 3. This algorithm directly looks for double-U dependency. Suppose k is found for given i, t, and j, A_s (t, k) is updated by A_s (t, i) while it is also used to update A_s (j, k). As a result,

Algorithm 3 Double-U dependency detection algorithm used in GLU2.0

```
1:  for i = 1 to n do
2:      Store all non-zero indices of row i in I_i
3:      for t = i to n where A_s(t, i) ≠ 0 do
4:          for j = t to n where A_s(j, t) ≠ 0 do
5:              Store all non-zero indices of row j in I_j
6:              if ∃k, k ∈ I_i, k ∈ I_j, k > t then
7:                  Add i to t's dependency list
8:              end if
9:          end for
10:     end for
11: end for
```

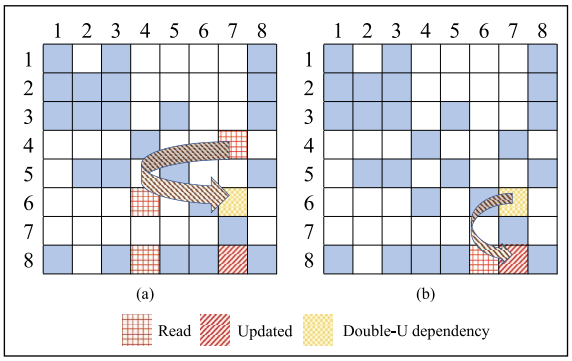

Figure 4. Example of double-U dependency originated from element (6,7). It can be concurrently (a) updated by column 4, (b) read by column 6.

a double-U dependency exists between columns i and t. In the example of Figure 4, $i = 4$, $t = 6$, $j = 8$, and $k = 7$, respectively.

However, this detection algorithm can be quite expensive because of the three nested loops that have $O(n^3)$ complexity. In comparison, there are only two for loops in the U matrix pattern-based dependency detection algorithm. It leads to performance degradation compared to GLU1.0.

In addition to dependency detection and levelization, certain preprocessing and symbolic analysis needs to be done on CPU ahead of factorization. The preprocessing includes MC64 and Approximate Minimum Degree (AMD) algorithms to reduce the number of final nonzero elements, as is done in NICSLU [5]. The symbolic analysis includes fill-in and levelization. Combining all these, we have the complete flow of GLU2.0 as shown in Figure 5.

Enhancements to GLU2.0

Recently, Lee et al. [10] proposed a method to enhance GLU2.0. In detail, three different kernels were proposed, namely cluster mode, batch mode, and pipeline mode. Modes are selected based on a different number of columns in levels. In batch mode and pipeline mode, overlapped execution of different levels is achieved to some extent, which contributes to the speedup. In addition to this, kernel launches are managed by a small kernel instead of CPU, which is called *dynamic parallelism* enabled by CUDA compute capability 3.0. Combining these techniques, the enhanced GLU has achieved 1.26 × (geometric mean) speedup over GLU2.0.

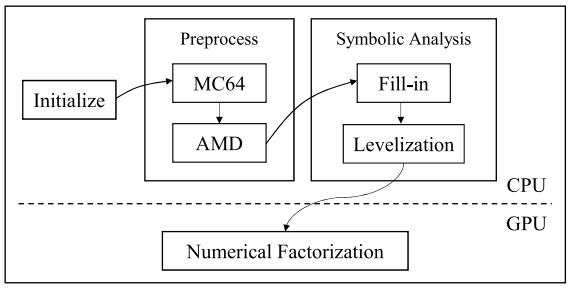

Figure 5. Complete flow of GLU2.0.

Review of GPU architecture and CUDA programming

CUDA 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 a huge amount of DRAM, referred to as *global memory*. Take the GTX TITAN X GPU for example. It contains 24 SM, each of which has 128 SM (SPs, or CUDA cores called by NVIDIA), 8 special function units (SFU), and its own shared memory/L1 cache. The architecture of the GPU and SM is shown in Figure 6.

As the programming model of GPU, CUDA extends C into CUDA C and supports such tasks as threads calling and memory allocation, which allows programmers to explore most of the capabilities of GPU parallelism. In the CUDA programming model, illustrated in Figure 7, 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, 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 reveals 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 a unique ID to all blocks and 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. Figure 7 shows an example of 2-dim blocks and 2-dim threads in a grid, and the block ID and thread ID are indicated by their row and column positions.

New GPU sparse LU solver: GLU 3.0

As introduced above, the workflow of factorizing a sparse matrix with GLU can be divided into two parts: the preprocessing and symbolic analysis of CPU and the numeric factorization on GPU. The second part on GPU might be repeated many times when solving a nonlinear equation with the Newton–Raphson method in circuit simulation. In this work, we significantly improve both the symbolic analysis and numeric factorization.

Relaxed data dependency detection method for GLU

As mentioned in the "Additional data dependency in GLU: the fix in GLU2.0" section, the prior dependency detection algorithm introduced to cover double-U dependency slowed down the factorization to a great extent. In this work, we solve this problem by proposing a better dependency detection algorithm, called the *relaxed column-based dependency detection method*, which can reduce the process down to two loops. The new algorithm is based on the observation that a *necessary condition* for such additional dependency is the existence of nonzero elements on the left of the diagonal element in the L matrix. In the example shown in Figure 4, such dependency exists between columns 4 and 6. The nonzero element $A_s(6, 4)$ on the left of diagonal element $A_s(6, 6)$ is the necessary condition that column 6

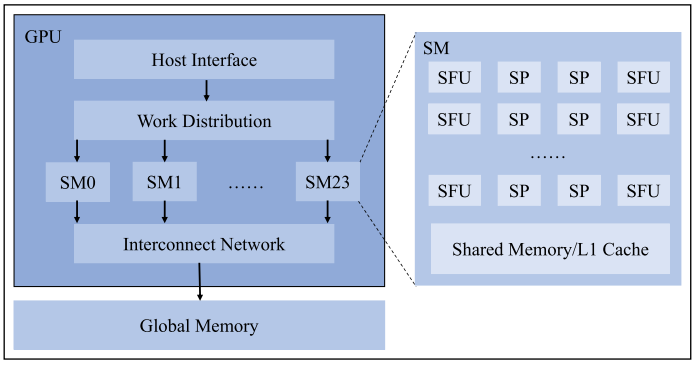

Figure 6. Diagram of NVIDIA TITAN X and the SM. (SP is short for streaming processor, L/S for load/store unit, and SFU for special function unit.)

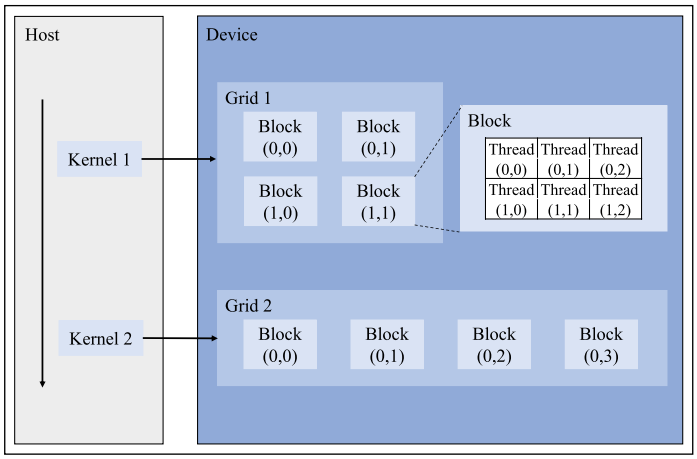

Figure 7. Programming model of CUDA.

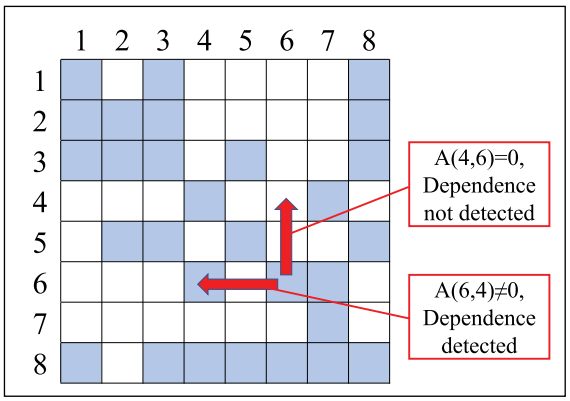

Figure 8. Comparison of left looking and up looking, and left looking is able to detect double-U dependency.

depends on column 4, which is due to that A_s (6, 7) gets updated and is the very element that induces the double-U dependency.

Algorithm 4 Proposed relaxed column dependency detection method

```
1:  for k = 1 to n do
2:    /* Look up for all nonzeros in column k of U */
3:    for i = 1 to k − 1 where A_s(i, k) ≠ 0 do
4:      if Column i of L is not empty then
5:        Add i to k's dependency list
6:      end if
7:    end for
8:    /* Look left for all nonzeros in row k of L */
9:    for i = 1 to k − 1 where A_s(k, i) ≠ 0 do
10:     Add i to k's dependency list
11:   end for
12: end for
```

Based on this observation, the new method simply just looks for nonzero elements on the left of the diagonal element, which can be called simply as *left looking*, to find such new dependency. It is very similar to the "up looking" in the U matrix-based dependency detection method employed in the left-looking factorization algorithm. Figure 8 shows the comparison of those results by applying both methods to column 6. As there is no nonzero element in column 6 of the U matrix, "looking up" from A_s (6, 6) will find no dependent column of column 6. On the other hand, "looking left" from the same element, a nonzero element in column 4 can be seen, which is interpreted as the new dependency between columns 4 and 6 that is the double-U dependency as expected. The complete algorithm incorporating the new dependency detection method is listed in Algorithm 4. Lines 8–11 are the additional "left looking" part that is added to the original dependency detection algorithm from GLU1.0.

To compare the aforementioned three dependency detection methods, they are applied to the example matrix from Figure 2 and the results are shown in Figure 9, respectively. An edge $x \rightarrow y$ indicates that column x depends on column y. Comparing (a) and (b), the extra dependencies $1 \rightarrow 2$ and $4 \rightarrow 6$ (marked by the blue line) are the double-U dependencies. Further comparing (b) and (c), we can see that the proposed method is able to detect all required column dependencies, plus a few redundant ones marked by red. Despite the redundant dependencies, the result of levelization is exactly the same, which means the same numerical performance on GPU can be expected. This example shows that the redundant dependencies have minor, if none, impacts on parallelism exploration of GLU. The reason why this dependency detection method is called *relaxed* is that it does not detect the exact set of dependencies, but a sufficient one possibly with some redundant dependencies. More examples on this will be reported later in the "Numerical results and discussion" section.

New numerical kernels

Before we discuss our new GPU kernels, we would like to first review the submatrix update in GLU, which is a key step in Algorithm 2.

1) The submatrix update revisited. The submatrix update is explicitly listed in Algorithm 5. Specifically, we can write the submatrix to be updated as

$$A_{\text{sub}} = \begin{bmatrix} A_s(j+1,j+1) & \dots & A_s(j+1,n) \\ \vdots & \ddots & \vdots \\ A_s(n,j+1) & \cdots & A_s(n,n) \end{bmatrix} \quad (1)$$

where the size of the submatrix is $N \times N$, with $N = n - j$. The submatrix update operation can be further represented in the following format:

$$A_{\text{sub}} \leftarrow A_{\text{sub}} - \begin{bmatrix} A_s(j+1,j) \\ \vdots \\ A_s(n,j) \end{bmatrix} . [A_s(j,j+1),\dots,A_s(j,n)] \quad (2)$$

where the size of the two vectors is $N \times 1$ and $1 \times N$. Both two vectors and A_{sub} matrix are sparse. From (2), we can see that the submatrix update consists of two operations: 1) vector tensor multiplication (the second item on the right-hand side) and 2) matrix addition.

In the implementation of GLU, the submatrix update

$$A_{\text{sub}} - \begin{bmatrix} A_s(j+1,j) \\ \vdots \\ A_s(n,j) \end{bmatrix} . [A_s(j,j+1),\dots,A_s(j,n)]$$

is done column-wise as depicted as follows:

$$\vec{A}_s(j+1{:}n,i) - \vec{A}_s(j+1{:}n,j) \cdot A_s(j,i), \quad \text{for } i = [j+1,\dots,n] \quad (3)$$

where

$$\vec{A}_s(j+1{:}n,i) = [A_s(j+1,i),\dots,A_s(n,i)]^T$$
$$\vec{A}_s(j+1{:}n,j) = [A_s(j+1,j),\dots,A_s(n,j)]^T.$$

As we can see, the submatrix update consists of vector operations or *subcolumn update*. Each time, we can update one *subcolumn* i as shown in (3). This can be parallelized in GPU where each resulting element can be computed using one thread, where the operation is MAC operation. There are two levels of parallelism, namely, 1) the vector operations (or subcolumn updates) for different vectors as shown in (3) and 2) element-wise MAC operations in each vector or subcolumn. On the contrary, the left-looking algorithm only has element-wise MAC operation parallelism in the triangular matrix solving process.

2) New adaptive GPU kernel. The second contribution we made is to significantly improve the GPU kernel computing efficiency for GLU. GLU1.0/2.0 used a fixed resource allocation strategy in the GPU kernel. However, as the matrix size increases, the fixed resource allocation strategy will significantly restrict the potential parallelism in GPU.

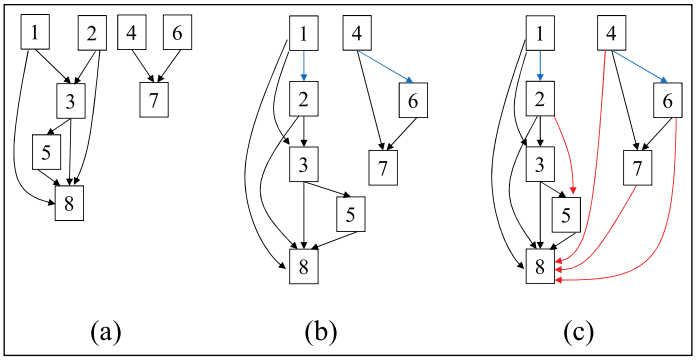

Figure 9. Dependency graph generated from three methods: (a) GLU1.0: incorrect result, (b) GLU2.0: correct result, and (c) this work: the relaxed data dependency.

Before going into further details, we define several terms for the ease of discussion. All columns in the same level that can be factorized in parallel referred to as *parallelizable*; the *size* of a level is the number of parallelizable columns in this level. In other words, a *large* level has many parallelizable columns, while a *small* level has few columns.

As defined, all columns in one level are parallelizable, and each column has many associated parallelizable subcolumns. This two-level parallelism distinguishes GLU from other parallel sparse matrix LU factorization algorithms. Two metrics can be used to describe the potential parallelism respectively, namely the size of one level and the maximum number of subcolumns for all columns in one level because they are the basic units that get parallelized.

The potential parallelism keeps changing across the levels, which is the key reason for the fixed resource allocation strategy being inefficient. The trend of potential parallelism is shown in Figure 10. An important observation is that levels generally fall into three categories, which are also labeled in the figure. Type A levels in the beginning stage of factorization have a huge number of parallelizable columns, while each column has very few associated subcolumns. For higher throughput, parallelizing

Algorithm 5 Submatrix update in GLU

```
/*Update the submatrix for next iteration*/
for k = j + 1 to n where A_s(j, k) ≠ 0 do
    for i = j + 1 to n where A_s(i, j) ≠ 0 do
        A_s(i, k) = A_s(i, k) − A_s(i, j) ∗ A_s(j, k)
    end for
end for
```

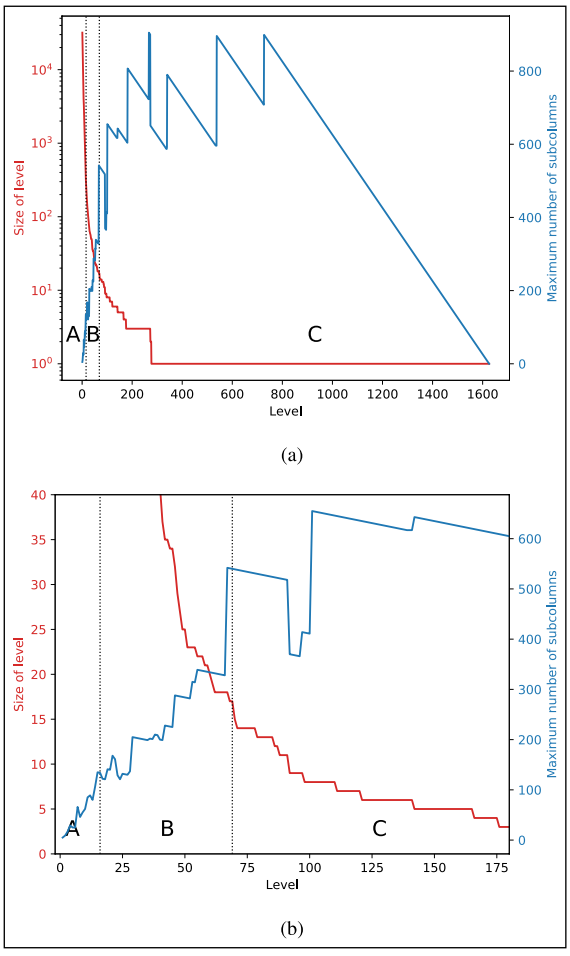

Figure 10. Number of columns and subcolumns of different levels. Maximum number of subcolumns is used for each level. The matrix is ASIC100ks from [12]. (a) Level versus its size and the maximum number of subcolumns and (b) zoomed-in view.

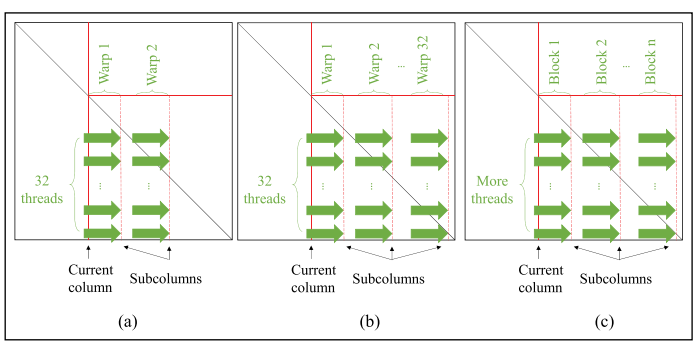

Figure 11. Comparison of the concurrency layout for one column in different kernels: (a) small block mode, (b) large block mode, and (c) stream mode.

columns should be prioritized for these types of levels. Type C levels, On the contrary, have a limited number of columns, while each column generally has a large number of subcolumns until the very end of the factorization process. As a result, parallelizing subcolumns is more important for these types of levels. Type B levels, in the transitional stage, have great numbers of columns with many subcolumns. Therefore, parallelism should be naturally balanced between them.

Furthermore, the second important observation we have is that for one level the numbers of parallelizable columns and their associated subcolumns are inversely correlated in general. As a result, we can use the size of the level as a good estimation of the associated subcolumn numbers to dynamically allocate the computing resources to further improve the GPU kernel computing efficiency. Based on this observation, we propose three computing modes of GLU kernels, which are chosen based on the level sizes in a progressive way to accommodate the three types of levels.

Small block mode. This mode is designed for type A levels. A convention we have followed from GLU for this mode is that one block takes care of a column, and one warp is assigned to a subcolumn. In this mode, as shown in Figure 11a, fewer warps are assigned to a CUDA block, which is why we name it small block mode. As the total number of warps is fixed for a given GPU, more blocks, or equivalently the factorization of more columns, can be carried out in parallel, which fits the requirement of type A levels: huge number of columns with a few subcolumns. Another important observation from Figure 10 is that the number of subcolumns is gradually increasing, and the level size is decreasing rapidly. To adapt to this change, the number of warps assigned to a block is gradually increased, assisting the growing number of subcolumns while trying to make full use of the available warps. The number of warps assigned to a block grows from 2 to 4, 8, and eventually to 32, which is the number in the next mode. The exact number of warps assigned to a block is determined by a number of columns in a level using the following expression:

$$W = \frac{\text{Total number of warps}}{\text{Level size}} \tag{4}$$

where W is the number of warps assigned to each block. Another factor limiting the number of possible

parallel columns is memory. As the columns being factorized are stored as a dense form in global memory, too many columns from a big level can overflow the memory. Specifically, during the factorization of each column, an array of size n is allocated for caching. As a result, the maximum parallelizable columns N can be calculated as

$$N = \frac{\text{Max global memory allowed}}{n * \text{size of (float)}} \quad (5)$$

where n is the number of rows of the matrix.

Large block mode. This mode takes care of type B levels, and it is similar to the kernel used in previous GLU versions. Similar to the small block mode, each block takes care of one column and each warp is assigned to a subcolumn. In this mode, the number of subcolumns still continues growing, and the number of threads that each subcolumn gets (32, one warp) becomes insufficient. However, the maximum number of a thread block (1024) prohibits any further increase in this number.

Stream mode. To tackle the maximum warp size (32) problem, stream mode is proposed for type C levels in this work. In this mode, blocks instead of warps are assigned to each subcolumn, and therefore kernel calls instead of blocks are assigned to each column, and a block is assigned to each subcolumn now, as shown in Figure 11c. To fully exploit parallelism within the same level, CUDAStreams are used, which allows parallel kernel execution through streams in a GPU. Although the number of CUDAStreams could also be dynamical, it has been observed that creating more CUDAStreams sometimes has a negative effect on performance. As a result, the number of CUDAStreams has been set to a fixed number, 16. This number is able to produce optimal results based on our experimental results which will be discussed later. Accordingly, this mode begins as long as the level size drops to 16.

We remark that the three GPU kernel modes we proposed are quite different than the three modes proposed in [10]. First, our approach is based on the observation of both parallelizable column count and associated subcolumn count change with different levels, while Lee's work is only based on the column count in each level. Second, we propose to dynamically allocate GPU computing resources (different numbers of warps and threads in each block, etc.) based on that information, while Lee's work exploits some advanced GPU features such dynamic kernel launch. Third, Lee's work focuses more on exploiting parallelism between different levels, while GLU 3.0 focuses on dynamically changing parallelisms in one level over the course of the factorization process.

Numerical results and discussion

The proposed GLU3.0 is implemented in C++ and CUDA- C and compiled with optimization level 3 (-O3). The tests were run on a server equipped with Intel(R) Xeon(R) CPU E5-2698 v3, 128 GB RAM, and NVIDIA GTX TITAN X (3072 CUDA cores, 12 GB GDDR5 memory, and Maxwell architecture). The test matrices occurs from the UFL [12], and the similar ones tested in [10] are used for the sake of comparison. Single precision floating point is used for computation as the Maxwell architecture does not support atomic operations for double precision. On newer GPU platforms that allow double atomic operations, the performance of GLU with double precision is expected on average 30% slower compared to that of the single-precision version [10].

First, we test the new relaxed data dependency detection method proposed in the "Relaxed data dependency detection method for GLU" section. It is applied to the test matrices to perform levelization. The results of levelization are presented in Table 2. For the purpose of saving space, more details of test matrices such as a number of nonzeros can be found in Table 1.

From this table, we have two observations. First, the number of additional levels resulting from the new dependency detection method is just a few or even zero. As the number of levels is the most decisive parameter of the runtime of the GPU kernel, this means the proposed new leveling algorithm would have marginal impacts on the runtime of numerical factorization on GPU. Second, the runtime of levelization (Algorithm 4) has improved dramatically on all test matrices compared to the existing method. The previous method of levelization used in GLU2.0 (Algorithm 3) has to explicitly find all double-U dependency, which has $O(n^3)$ complexity and thus makes the runtime of preprocessing nonnegligible (compared to the LU factorization time). However, with an average speedup ratio of 8804.1 (arithmetic average) or 3145.8 (geometric mean), the proposed new method is able to reduce the preprocessing runtime back into the similar time frame of

Table 1. Solver runtimes of GLU3.0 versus previous works, where *nz* stands for a number of nonzeros before fill-in, and *nnz* stands for a number of nonzeros after fill-in.

Matrix	Number of rows	nz	nnz	CPU time (ms)		Numerical factorization time (ms)					
				GLU3.0	GLU2.0	GLU3.0 (GPU)	GLU2.0 (GPU)	NICSLU (CPU) [5]	Speed-up over GLU2.0	Speed-up over [10]	Speed-up over [5]
rajat12	1879	12926	13948	3.999	13.998	2.237	2.44883	3.99	1.1	1.0	1.78
circuit_2	4510	21199	32671	7.998	59.991	4.144	8.36301	6.66	2.0	1.9	1.61
memplus	17758	126150	126152	15.997	377.943	6.672	6.90432	26.91	1.0	0.9	4.03
rajat27	20640	99777	143438	21.997	404.939	10.539	23.8673	34.44	2.3	2.0	3.27
onetone2	36057	227628	1306245	353.946	36729.4	60.964	550.598	432.69	9.0	8.3	7.10
rajat15	37261	443573	1697198	423.936	18461.2	71.135	458.611	356.90	6.4	6.1	5.02
rajat26	51032	249302	343497	76.988	2011.69	32.366	104.12	88.77	3.2	4.2	2.74
circuit_4	80209	307604	438628	295.955	4662.29	68.944	394.995	118.23	5.7	9.1	1.71
rajat20	86916	605045	2204552	2190.67	121207	241.822	2538.24	245.63	10.5	8.8	1.02
ASIC_100ks	99190	578890	3638758	2052.69	316998	215.493	2652.79	357.53	12.3	14.1	1.66
hcircuit	105676	513072	630666	67.99	6279.05	46.996	243.846	221.50	5.2	9.5	4.71
Raj1	263743	1302464	7287722	7240.9	140008	845.189	7969.05	825.38	9.4	8.7	0.98
ASIC_320ks	321671	1827807	4838825	2336.64	410679	216.517	5632.8	765.35	26.0	21.3	3.53
ASIC_680ks	682712	2329176	4957172	1747.73	686421	210.697	11771.7	614.75	55.9	18.4	2.92
G3_circuit	1585478	4623152	36699336	9728.52	1764580	878.153	38780.9	9232.618	44.2	8.2	10.51
								Arithmetic mean	13.0	7.1	3.51
								Geometric mean	6.7	4.8	2.81

the preprocessing time in the plain left-looking-based method.

Then, we test the performance of GPU kernels of GLU3.0 and compare it with GLU2.0 and NICSLU [5], which is a parallel sparse matrix LU factorization solver based on CPU. Thirty-two threads are used when testing the performance of NICSLU. The results are presented in Table 1. The speedup ratio of the proposed work [10] is calculated based on its reported speedup ratio against GLU2.0 using the same testing matrices. The runtime measured includes the time completing memory copy. CPU time that comprises preprocessing and symbolic analysis is compared as well. As can be seen from the table, despite slightly more levels as reported in Table 2, the proposed new GPU kernel still demonstrates a steady speedup over the kernels from GLU2.0 and the improved version from [10]. At least 5× speed-up can be achieved on average. Furthermore, more significant improvement can be expected when it comes to bigger matrices, starting from circuit_4 with a row number of 80209. The reason is that the computational tasks of small matrices are so light that the GPU still allows more parallelizable tasks. On the other hand, when factorizing larger matrices, the limited GPU computation power will throttle full parallelization in the GLU. In these cases, the proposed adaptive kernels can utilize the GPU in a better way so that more parallelism and shorter runtime are achieved.

Table 2. Levelization runtimes.

Matrix	Number of levels		Levelization Time (ms)		
	GLU2.0	this work	GLU2.0	this work	speed-up
rajat12	37	39	3.048	0.035	87.1
circuit_2	101	102	17.187	0.074	232.3
memplus	147	147	345.568	0.234	1476.8
rajat27	123	125	272.216	0.32	850.7
onetone2	1213	1213	4009.51	1.589	2523.3
rajat15	968	968	3680.02	2.224	1654.7
rajat26	157	158	1703.92	0.711	2396.5
circuit_4	228	229	5053.39	0.944	5353.2
rajat20	1216	1219	15931.2	3.389	4700.9
ASIC_100ks	1626	1626	36388.8	5.301	6864.5
hcircuit	144	145	6122.57	1.206	5076.8
Raj1	1594	1595	56580.9	11.102	5096.5
ASIC_320ks	1669	1669	168979	8.573	19710.6
ASIC_680ks	1450	1450	530478	10.642	49847.6
G3_circuit	652	688	1741860	66.508	26190.2
				Arithmetic mean	8804.1
				Geometric mean	3145.8

To further validate the improvement from the proposed three modes of kernels, another experiment was conducted, where either one of the two newly proposed modes (small block mode and stream mode) is disabled, to show the degradation of performance without them. The results are listed in Table 3. In case 1, the small mode is disabled, while in case 2, the stream mode is disabled. The number of three different types of levels is also listed. Comparing GLU3.0 with case 1, we can see that small block mode benefits most matrices except G3_circuit. Although the number of type A levels is generally small, small block mode can still lead to decent improvement. The reason for G3_circuit being slower without small block mode is probably that the number of blocks assigned

in small block mode is less than optimal because of the limitation of (5). In this case, more warps should be assigned to a block as the total number of blocks is limited. Then, comparing GLU3.0 with case 2, a more significant improvement can be seen from stream mode. Furthermore, stream mode tends to benefit all matrices, as the results of GLU3.0 are either much faster or at worst equivalent. In particular, the improvement is more significant for large matrices such as ASIC_100ks and Raj1.

In the "New numerical kernels" section, we mentioned that stream mode starts when level size decreases to 16. This number is also selected based on an experiment. The results can be found in Figure 12. For better understanding, instead of using all matrices used in the previous experiments, only those that benefit significantly from stream mode are selected. In the figure, N stands for the threshold of a level size where stream mode begins, and the values plotted are GPU kernel runtimes with different N compared with that with $N = 5$. It can be seen that the runtime keeps reducing until $N = 16$. Except for matrix Raj1, experiments with all other matrices show slower or equivalent results for larger N, which proves that $N = 16$ is a good choice.

According to profiling results, unused warp is the main challenge for this problem. In fact, the newly proposed three modes of kernels have greatly improved the utilization of threads in SM, despite some remaining mismatch due to unpredictable sparsity pattern of the matrix. This hurts the performance of the stream mode most significantly. As in other modes, the warp occupancy is as high as 80%, while in-stream mode the average is 40%. However, it is also worth noting that in the ending stage of factorization, as the submatrix size decreases, warp occupancy would drop naturally.

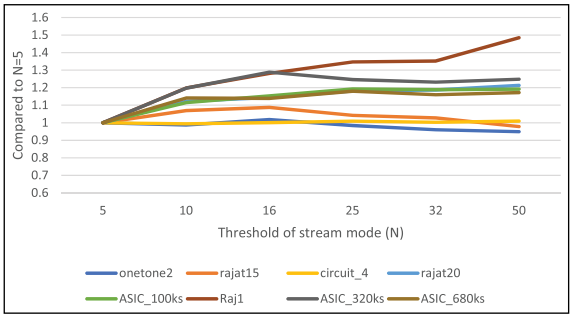

Figure 12. Performance of the GPU kernel with different stream mode threshold settings.

Table 3. GPU kernel runtimes without enabling all three kernel modes, compared to case 1 where small block mode is disabled, and case 2 where stream mode is disabled.

Matrix	GPU time (ms)			Level distribution		
	GLU3.0	Case 1	Case 2	A	B	C
rajat12	2.237	2.776	2.158	2	4	33
circuit_2	4.144	4.871	4.650	1	10	91
memplus	6.672	9.364	7.187	4	3	140
rajat27	10.539	13.069	10.665	6	23	96
onetone2	60.964	66.126	173.863	14	33	1166
rajat15	71.135	82.677	163.947	11	96	861
rajat26	32.366	43.697	35.330	8	36	114
circuit_4	68.944	170.49	103.515	7	9	213
rajat20	241.822	571.95	1019.12	11	41	1167
ASIC_100ks	215.493	246.84	1047.78	13	56	1557
hcircuit	46.996	59.103	47.761	10	14	121
Raj1	845.189	2611.12	2115	29	223	1343
ASIC_320ks	216.517	311.778	1094.78	14	50	1605
ASIC_680ks	210.697	279.784	721.589	14	55	1381
G3_circuit	878.153	783.592	877.444	104	327	257

We note that driver overhead is also significant in many of our tests. Take ASIC_100ks as an example, the first CUDA function call (including invisible set-up works) takes 40% of all GPU time (215 ms). For larger matrices, this problem should be less severe for larger matrices or in real simulation scenarios where the factorization kernel is called repeatedly.

We have proposed a new sparse LU solver on GPU for circuit simulation and more general scientific computing. The new sparse LU solver, called *GLU3.0*, has two main improvements. First, a more efficient data dependency detection algorithm is introduced. Second, three different kernel operation modes are developed based on a different number of columns in a level, as the LU factorization progresses. They enable dynamic allocation of GPU blocks to better balance the computing demands and resources during the LU factorization process. Numerical results on the set of typical circuit matrices from the UFL have shown that GLU3.0 can deliver 2–3 orders of magnitude speedup over GLU2.0 for the data dependency detection. Furthermore, GLU3.0 achieves 13.0 × (arithmetic mean) or 6.7 × (geometric mean) speedup over GLU2.0 and 7.1 × (arithmetic mean) or 4.8 × (geometric mean) over the recently proposed enhanced GLU2.0 sparse LU solver on the same set of circuit matrices. ■

Acknowledgments

This work was supported in part by NSF under Grant CCF-1527324 and Grant CCF-1816361.

References

[1] K. He et al., "GPU-accelerated parallel sparse LU factorization method for fast circuit analysis," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 3, pp. 1140–1150, Mar. 2016.

[2] J. W. Demmel, J. R. Gilbert, and X. S. Li, "An asynchronous parallel supernodal algorithm for sparse Gaussian elimination," *SIAM J. Matrix Anal. Appl.*, vol. 20, no. 4, pp. 915–952, 1999.

[3] T. A. Davis and E. Palamadai Natarajan, "Algorithm 907: KLU, a direct sparse solver for circuit simulation problems," *ACM Trans. Math. Softw.*," vol. 37, no. 3, pp. 36:1–36:17, Sep. 2010.

[4] J. R. Gilbert and T. Peierls, "Sparse partial pivoting in time proportional to arithmetic operations," *SIAM J. Sci. Stat. Comput.*, vol. 9, no. 5, pp. 862–874, 1988.

[5] X. Chen, Y. Wang, and H. Yang, "NICSLU: An adaptive sparse matrix solver for parallel circuit simulation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 2, pp. 261–274, Feb. 2013.

[6] N. Galoppo et al., "LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware," in *Proc. ACM/IEEE Supercomput. (SC) Conf.,* 2005, p. 3.

[7] D. Yu Chenhan, W. Wang, and D. Pierce, "A CPU-GPU hybrid approach for the unsymmetric multifrontal method," *Parallel Comput.*, vol. 37, no. 12, pp. 759–770, 2011.

[8] T. George et al., "Multifrontal factorization of sparse SPD matrices on GPUs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.,* 2011, pp. 372–383.

[9] X. Chen et al., "GPU-accelerated sparse LU factorization for circuit simulation with performance modeling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 3, pp. 786–795, Mar. 2015, doi: 10.1109/TPDS.2014.2312199.

[10] W.-K. Lee, R. Achar, and M. S. Nakhla, "Dynamic GPU parallel sparse LU factorization for fast circuit simulation," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 11, pp. 99:1–12, Nov. 2018.

[11] T. A. Davis, *Direct Methods for Sparse Linear Systems.* Philadelphia, PA, USA: SIAM, 2006.

[12] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, p. 1, 2011.

Shaoyi Peng is currently pursuing a PhD with the Department of Electrical and Computer Engineering, University of California at Riverside, Riverside, CA. His research interest includes VLSI reliability effect modeling and simulation, finite element method analysis, and numerical methods. He received a BS in microelectronics from Fudan University, Shanghai, China (2016). He is a Student Member of the IEEE.

Sheldon X.-D. Tan is a Professor with the Department of Electrical Engineering, University of California at Riverside (UCR), Riverside, CA. He is also a cooperative faculty member with the Department of Computer Science and Engineering, UCR. His research interests include VLSI reliability modeling, optimization and management at circuit and system levels, hardware security, thermal modeling, optimization and dynamic thermal management for manycore processors, parallel computing, and adiabatic and Ising computing based on GPU and multicore systems. He received a PhD in electrical and computer engineering from the University of Iowa, Iowa City, IA (1999). He is a Senior Member of the IEEE.

■ Direct questions and comments about this article to Department of Electrical and Computer Engineering, University of California at Riverside, Riverside, CA 92521 USA; stan@ece.ucr.edu.