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

Shaoyi Peng, *Student Member, IEEE,* Sheldon X.-D. Tan, *Senior Member, IEEE,*

Abstract—LU factorization for sparse matrices is the most important computing step for many engineering and scientific computing problems such as circuit simulation. But parallelizing LU factorization with the Graphic Processing Units (GPU) still remains a challenging problem due to high data dependency and irregular memory accesses. Recently GPU-based hybrid right-looking sparse LU solver, called GLU (1.0 and 2.0), has been proposed to exploit the fine grain level parallelism of GPU. However, a new type of data dependency (called double-U dependency) introduced by GLU slows down the preprocessing step. Furthermore, GLU uses fixed GPU thread allocation strategy, which limits the parallelism. In this article, we propose a new GPU-based sparse LU factorization method, called *GLU3.0*, which solves the aforementioned problems. First, it introduces a much more efficient data dependency detection algorithm. Second, we observe that the potential parallelism is different as the matrix factorization goes on. We then develop three different modes of GPU kernel which adapt to different stages to accommodate the computing task changes in the factorization. Experimental results on circuit matrices from University of Florida Sparse Matrix Collection (UFL) show that GLU3.0 delivers 2-3 orders of magnitude speedup over GLU2.0 for the data dependency detection. Furthermore, GLU3.0 achieve 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.

Index Terms—GPU, LU factorization, left-looking LU factorization, sparse matrices, GLU

I. INTRODUCTION

Sparse LU factorization plays a critical role in wide engineering and scientific computing applications such as solving differential and circuit equations. Particularly, for circuit simulation application such as the widely used SPICE program [1], the core of the computing or dominant computing is to solve the linear algebraic system, $Ax = b$, 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 solving the two triangular matrices sequentially. Also 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 of those analysis and simulation applications. As VLSI continues to grow in size, how to improve the LU factorization efficiency and scalability continues to be a challenging problem.

Graphics processing unit (GPU) provides massive and fine-grain parallelism with orders of magnitude higher throughput

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 [2], [3]. Today, in additional to gaming graphics, GPU has been widely used for more general purpose computing [4] such as EDA, deep learning/AI, finance, medical and life science etc. However, parallelizing sparse LU factorization on GPU (GPU) is not straightforward due to high data dependency and irregular memory access [5].

There exists some earlier researches targeting parallel sparse LU factorization on shared memory multi-core CPUs. For instance, SuperLU_MT [6] is the multi-threaded parallel version of SuperLU [7] for multi-core architectures. However, it is not easy to form super-node in some sparse matrix such as circuit matrix. KLU [8], which is specially optimized for circuit simulation, adopts Block Triangular Form based on Gilbert-Peierls (G/P) left-looking algorithm [9] and has become one of the standard algorithms in circuit simulation applications. The KLU algorithm has been parallelized on multi-core architecture by exploiting the column-level parallelism [10], [11].

Existing GPU based parallel LU factorization solvers mainly focus on dense matrices, including [12], [13], [14]. There also exists a few sparse matrix LU factorization methods on GPU [15], [16], [17]. But these works mainly convert the sparse matrices into many dense submatrices (blocks) and then solve them by dense matrix LU factorizations. However, such strategy may not work well for circuit matrices, which hardly have dense submatrices.

Parallel (G/P) left-looking algorithm [9] on GPU has been explored first in [18], [19]. It exploits the column-level (called task-level) parallelism due to 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 can't be completely parallelized (from line 4-8 in Algorithm 1) thus the G/P method is difficult for fine grain parallelization.

To mitigate this problem, He *et. al* proposed a hybrid right-looking sparse LU factorization on GPU, called GLU (GLU1.0) [5]. GLU keeps 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 paper), which has been reported in GLU2.0 [20] and [21]. 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.* proposed an enhanced GLU2.0 solver [21], which considers the column count difference in different level, 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 is still used and limiting performance.

In this article, we propose a new version of 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 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 a lot higher than the previous solution with little impact on performance.
- Second, we have observed a pattern of potential parallelism as the matrix factorization goes on, based on the circuit matrices we analyzed. 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 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 achieve $13.0 \times$ (arithmetic mean) or $6.7 \times$ (geometric mean) speedup over GLU 2.0 and $7.1 \times$ (arithmetic mean) or $4.8 \times$ (geometric mean) over recent proposed enhanced GLU2.0 sparse LU solver on the same set of circuit matrices.

This article is organized as follows. Section II gives a brief review of previous work on sparse matrix LU factorization, GPU programming, and GLU itself. In Section III, we present our two novel improvements. Several numerical examples and related discussions are presented in Section IV. At last, Section V concludes this work.

II. REVIEW OF LU FACTORIZATION AND CUDA

In this section, we briefly review the traditional G/P left-looking method for sparse matrices LU factorization [9] and the recently proposed hybrid right-looking algorithm used in GLU1.0, GLU2.0 and a recent GLU enhancement work [21].

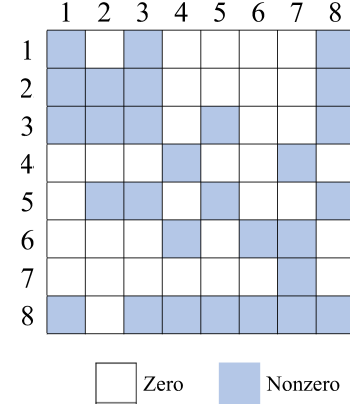[1]GLU 3.0 source codes and documents are available at [20].

Fig. 1. The example matrix

We would also briefly review the GPU architectures and NVIDIA CUDA programming system.

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

A. Left-looking factorization method

Traditional Gaussian elimination based LU factorization method (also called right-looking method) solves one row for U matrix and then one column for L matrix in each iteration. While 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, G/P left-looking method shows better performance for sparse matrices, especially those from circuit simulations [22].

Algorithm 1 shows the detailed implementation of G/P left-looking LU factorization [9]. The input of this algorithm A_s is the nonzero filled-in matrix of A after symbolic analysis. The matrix A_s is 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 (line 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 current column j. This is the reason why it is named 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.

Fig. 2 gives 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 $A_s(k, j) \neq 0$ (line 4), which are 4 and 6 (as $A_s(4, 7) \neq 0$ and $A_s(6, 7) \neq 0$). The two subfigures show these two iterations respectively. In (a), $k = 4$, so 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 the red elements in column 4 multiplying $A_s(4, 7)$. (b) shows the next iteration, where $k = 6$, column
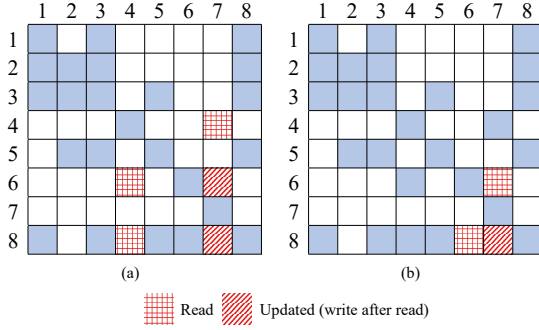
Fig. 2. The two update iterations completing factorization of the 7th column ($j = 7$) (a) update using the 4th column ($k = 4$) (b) update using the 6th column ($k = 6$)
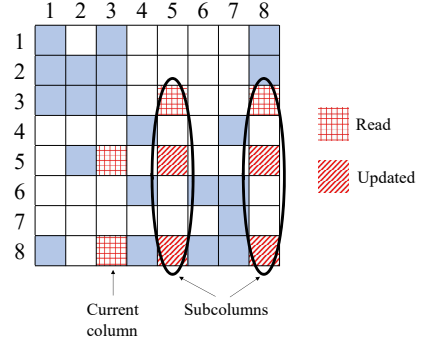

Fig. 3. Subcolumns and submatrix column 3. All highlighted elements compose the submatrix, which include elements being read and elements being updated.

3 is used to further update column 7, which can be explicitly written as $A_s(8,7) \leftarrow A_s(8,7) - A_s(8,6) * A_s(6,7)$.

Algorithm 1 The Gilbert-Peierls 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 $A_s(k,j) \neq 0$ **do**
5:     /*Vector multiple-and-add (MAC) */
6:     **for** $i = k + 1$ to n where $A_s(i,k) \neq 0$ **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 $A_s(i,j) \neq 0$ **do**
12:     $A_s(i,j) = A_s(i,j)/A_s(j,j)$
13:   **end for**
14: **end for**

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

As elaborated in [5], the G/P left-looking sparse LU factorization has one limitation that it failed to parallelize the two loops in 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.* proposed the hybrid column-based right-looking LU factorization algorithm for GLU [5]. 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*. Fig. 3 gives an example illustrating these two concepts. In the figure, $j = 3$, its subcolumns are column 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 submatrix update completes the equivalent jobs of triangular matrix solving (lines 4-9 of Algorithm 1) in advance. In the example shown in Fig. 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 will be discussed in detail in the following section, this difference enables exploiting parallelization between subcolumns.

Algorithm 2 The 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 $A_s(k,j) \neq 0$ **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 $A_s(j,k) \neq 0$ **do**
9:     **for** $i = j + 1$ to n where $A_s(i,j) \neq 0$ **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**

C. 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 [6] and NICSLU [11], elimination tree has been used to resolve this issue. Similarly, methods like dynamic application dependency graph is used in heterogeneous computing [23]. For GLU, in order 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 in the same level are independent of each other and can thus be factorized in
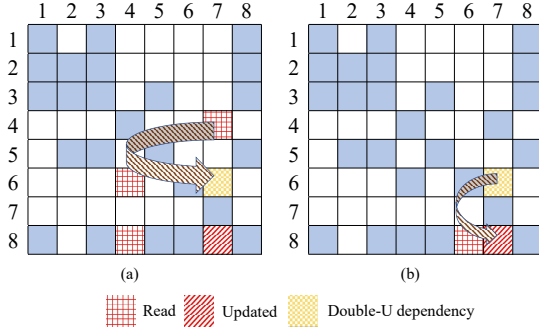
Fig. 4. An example of double-U dependency originated from element (6,7)

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 $A_s(t, i) \neq 0$ **do**
4:       **for** $j = t$ to n where $A_s(j, t) \neq 0$ **do**
5:          Store all non-zero indices of row j in I_j
6:          **if** $\exists k, k \in I_i, k \in I_j, k > t$ **then**
7:             Add i to t's dependency list
8:          **end if**
9:       **end for**
10:   **end for**
11: **end for**

parallel. Such process deriving information about levels is called *levelization*, which is a similar method to 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 [20], [21], 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 Fig. 4. In (a), $A_s(6, 7)$ is updated by column 4: $A_s(6, 7) \leftarrow A_s(6, 7) - A_s(6, 4) * A_s(4, 7)$. In (b), $A_s(6, 7)$ is used to update column 7: $A_s(8, 7) \leftarrow A_s(8, 7) - A_s(8, 6) * A_s(6, 7)$. 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 Fig. 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 a double-U dependency exists between columns i and t. In the example of Fig. 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.

Besides dependency detection and levelization, some pre-processing and symbolic analysis needs to be done on CPU ahead of factorization. The preprocessing includes MC64 and AMD (Approximate minimum degree) algorithms in order to reduce the number of final nonzero elements, as is done in NICSLU [11]. Symbolic analysis includes fill-in and levelization. Combining all this, we have the complete flow of GLU2.0 shown in Fig. 5.
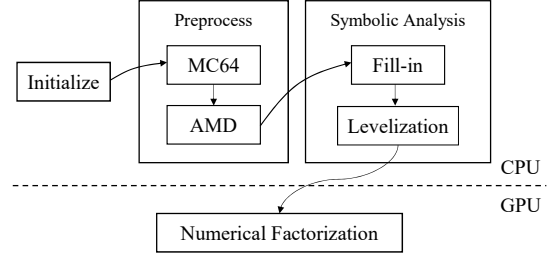

Fig. 5. Complete flow of GLU2.0

D. Enhancements to GLU2.0

Recently, Lee *et. al* proposed a method to enhance GLU2.0[21]. In detail, three different kernels were proposed, namely cluster mode, batch mode and pipeline mode. Modes are selected based on 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 speed-up. Besides 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 $1.26\times$ (geometric mean) speedup over GLU2.0.

E. Review of 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 a huge amount of DRAM, referred to as global memory. Take the GTX TITAN X GPU for example. It contains 24 SMs, each of which has 128 streaming multiprocessors (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 streaming multiprocessors is shown in Fig. 6.

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. 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.
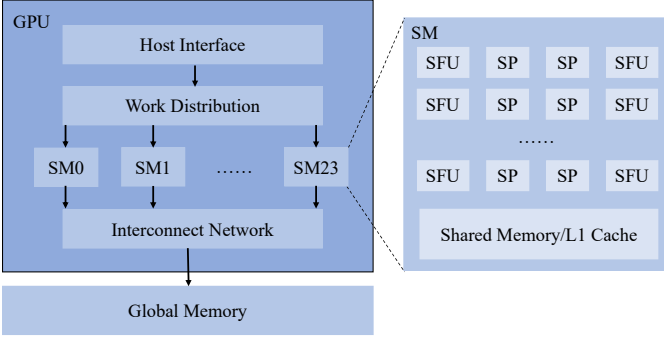
Fig. 6. Diagram of NVIDIA TITAN X and the streaming multiprocessor. (SP is short for streaming processor, L/S for load/store unit, and SFU for Special Function Unit.)
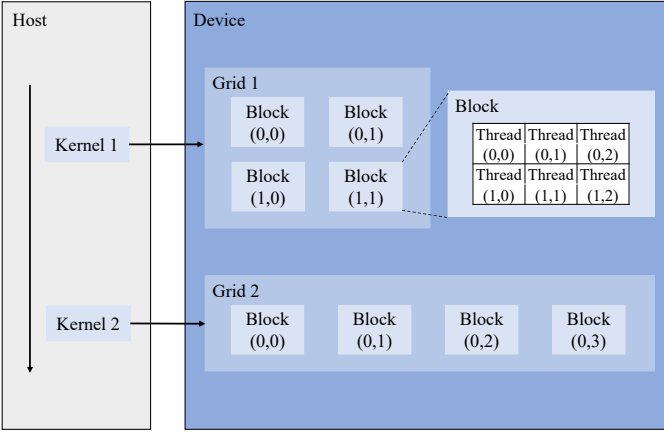

Fig. 7. The programming model of CUDA.

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 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. 7 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. New GPU sparse LU solver: GLU 3.0

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

A. Relaxed data dependency detection method for GLU

As mentioned in Section II-C, the prior dependency detection algorithm introduced to cover double-U dependency slowed down the factorization a lot. In this work, we solve this problem by proposing a better dependency detection algorithm, called *relaxed* column 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 diagonal element in the L matrix. In the example in Fig. 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 depending on column 4, as it is the reason that $A_s(6,7)$ gets updated, and $A_s(6,7)$ is the very element that induces the double-U dependency.

Based on this observation, the new method simply just look for nonzero elements on the left of 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. Fig. 8 compares the result of them by applying both methods to column 6. As there is no nonzero element in column 6 of U matrix, "Looking up" from $A_s(6,6)$ will find no depended 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.
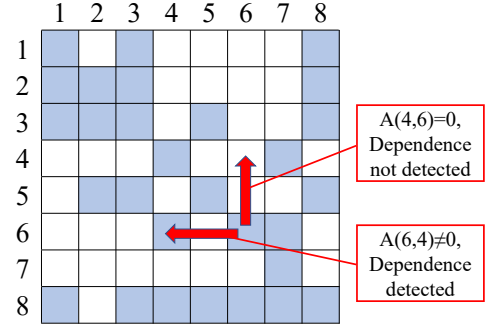

Fig. 8. Comparison of left looking and up looking, left looking is able to detect double-U dependency.

In order to compare the aforementioned three dependency detection methods, they are applied to the example matrix from Fig. 2 and the results are shown in Fig. 9 respectively. An edge $x \rightarrow y$ indicates that that column x depends on column y. Comparing (a) and (b), the extra dependencies $1 \rightarrow 2$ and $4 \rightarrow 6$ (marked by 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
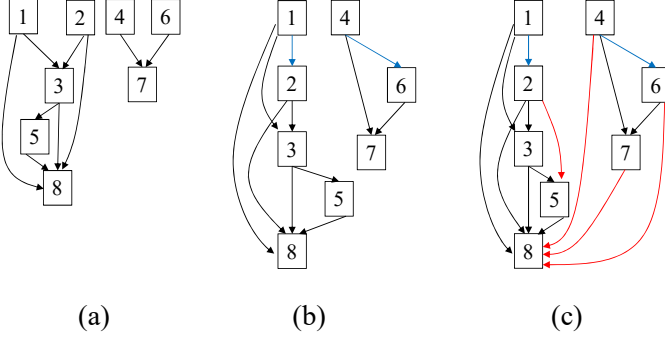
Fig. 9. Dependency graph generated from 3 methods: (a) GLU1.0: incorrect result (b) GLU2.0: correct result (c) This work: the relaxed data dependency

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 about this will be reported later in Section IV.

Algorithm 4 The 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 $A_s(i, k) \neq 0$ **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 $A_s(k, i) \neq 0$ **do**
10:       Add i to k's dependency list
11:    **end for**
12: **end for**

B. 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 below. Specifically, we can

---

Algorithm 5 The submatrix update in GLU

1: /*Update the submatrix for next iteration*/
2: **for** $k = j + 1$ to n where $A_s(j, k) \neq 0$ **do**
3:    **for** $i = j + 1$ to n where $A_s(i, j) \neq 0$ **do**
4:       $A_s(i, k) = A_s(i, k) - A_s(i, j) * A_s(j, k)$
5:    **end for**
6: **end for**

---

write the submatrix to be updated as

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

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:

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

where the size of the two vectors are $N \times 1$, 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: (a) vector tensor multiplication (the second item on the right hand side); (b) matrix addition.

In the implementation of GLU, the submatrix update

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

is done in a column-wise way as depicted in (3):

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

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 multiply-accumulate (MAC) operation. There are two levels of parallelism: namely (a) the vector operations (or subcolumn updates) for different vectors as shown in (3) and (b) element-wise MAC operations in each vector or subcolumn. In contrast, 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 fixed resource allocation strategy in the GPU kernel. However, as the matrix size grows, the fixed resource allocation strategy will significantly restrict the potential parallelism in GPU.

Before going into 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 of the fixed resource allocation strategy being inefficient. The trend of potential parallelism is shown in Fig. 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 huge number of parallelizable columns, while each column has very few associated subcolumns. For higher throughput, parallelizing columns should be prioritized for this type of levels. Type C levels, in contrast, have limited number of columns, while each column generally has large number of subcolumns until very end of the factorization process. As a result parallelizing subcolumns is more important for this type of levels. Type B levels, in the transitional stage, have great numbers of columns, and at the same time columns also have many subcolumns. So parallelism should be naturally balanced between them.

Furthermore, the second important observation we have is rev*that the in one level number of parallelizable columns and their associated subcolumns are inversely correlated in general*. As a result, we can use the size of a 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.

1. **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 Fig. 11(a), 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 Fig. 10 is that the number of subcolumns is gradually increasing, and the level size is decreasing quickly. In order to adapt to this change, the number of warps assigned to a block is gradually increased, assisting the growing number of subcolumns and trying to make full use of available warps at the same time. 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 number of columns in a level using following expression:

$$W = \frac{\text{Total number of warps}}{\text{Level size}} \quad (4)$$

where W is the number of warps assigned to each block. Another factor limiting the number of possible parallel columns is memory. Because 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 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:

$$N = \frac{\text{Max global memory allowed}}{n * \text{sizeof(float)}} \quad (5)$$
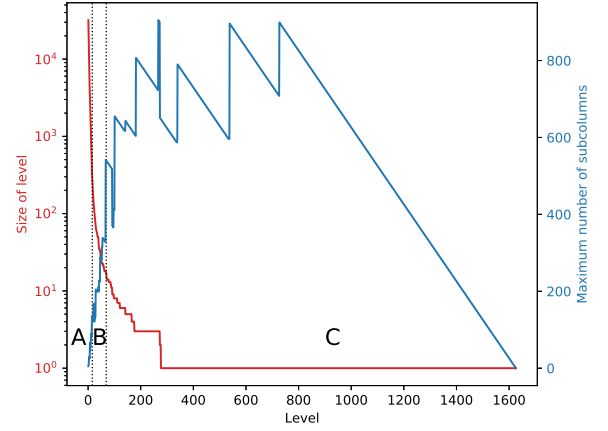
(a) Level versus its size and the maximum number of subcolumns
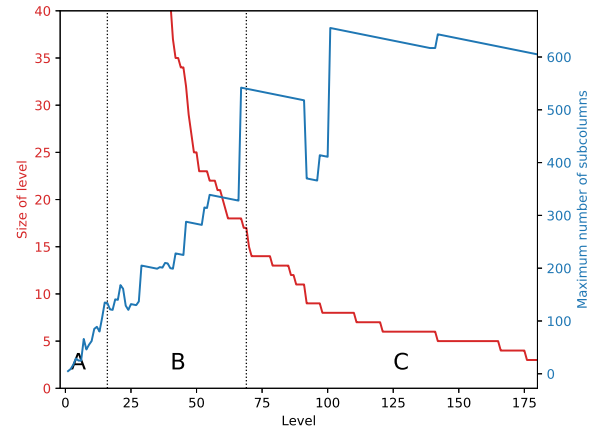
(b) Zoomed in view

Fig. 10. Number of columns and subcolumns of different levels. Maximum number of subcolumns is used for each level. The matrix is ASIC100ks from [24].

where n is the number of rows of the matrix.

2. **Large block mode:** This mode takes care of type B levels, and it is similar to the kernel used in previous GLU versions. Same as 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 keep growing, the number of threads each subcolumn gets (32, one warp) becomes insufficient. However, the maximum number of a thread block (1024) prohibits any further increase in this number.

3. **Stream mode:** To tackle 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 is shown in Fig. 11(c). 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 dynamically, it has been observed that creating more CUDAStreams sometimes has a negative effect in performance. As a result, the number of CUDAStreams has been set to a
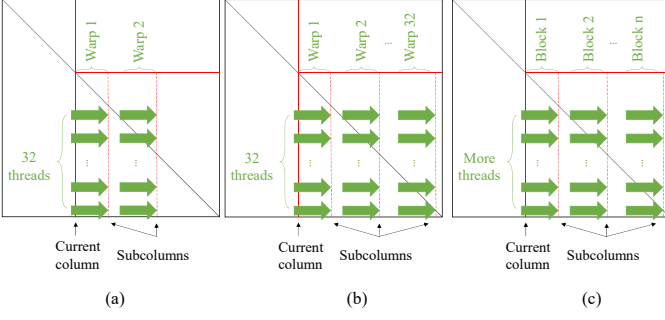
Fig. 11. Comparison of the concurrency layout for one column in different kernels: (a) Small block mode (b) Large block mode (c) Stream mode

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 then three modes proposed in [21]. 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 number of warps and threads in each blocks etc) based on those information, while Lee's work exploits some advanced GPU features such dynamic kernel launch. Third, Lee's work focus more on exploit parallelism between different levels, while GLU 3.0 focus on dynamically changing parallelisms in one level over the course of factorization process.

IV. NUMERICAL RESULTS AND DISCUSSIONS

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, 128GB RAM, and NVIDIA GTX TITAN X (3072 CUDA cores, 12GB GDDR5 memory, Maxwell architecture). The test matrices come from the University of Florida Sparse Matrix Collection [24], and the same ones tested in [21] 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 [21].

First we test the new relaxed data dependency detection method proposed in Section III-A. It is applied to the test matrices to perform levelization. The results of levelization are presented in Table II. For the purpose of saving space, more details of test matrices such as number of non-zeros can be found in Table I.

From this table, we have two observations. First, the number of additional levels resulting from the new dependency detection method are just a few or even zero. As the number of levels is the most decisive parameter of 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 non-negligible (compared to the LU factorization time). However, with an average speed-up 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 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 [11], which is a parallel sparse matrix LU factorization solver based on CPU. 32 threads are used when testing the performance of NICSLU. The results are presented in Table I. The speed-up ratio of the proposed work over [21] is calculated based on its reported speed-up ratio against GLU2.0 using the same testing matrices. The runtime measured includes the time completing memory copy. CPU time that comprises of preprocessing and symbolic analysis is compared as well. As can be seen from the table, despite slightly more levels as reported in Table II, the proposed new GPU kernel still demonstrates a steady speedup over the kernels from GLU2.0 and the improved version from [21]. At least 5x 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 is achieved.

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) are disabled, to show the degradation of performance without them. The results are listed in Table III. In case 1, small mode is disabled. While in case 2, stream mode is disabled. The number of three different types of levels are 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 of 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 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 tend to benefit all matrices, as the results of GLU3.0 are either much faster or at worst equivalent. Especially, the improvement is more significant for large matrices such as ASIC_100ks and Raj1.

In Section III-B, we mentioned that stream mode starts when level size decreases to 16. This number is also selected based on experiment. The results can be found in Figure 12. For the purpose of making the figure more clear, instead of using all matrices used in previous experiments, only the ones that benefit significantly from stream mode are selected. In the

TABLE I
SOLVER RUNTIMES OF GLU3.0 VS PREVIOUS WORKS, WHERE *nz* STANDS FOR NUMBER OF NONZEROS BEFORE FILL-IN, AND *nnz* STANDS FOR 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) [11]	Speed-up over GLU2.0	Speed-up over [21]	Speed-up over [11]
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

TABLE II
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

TABLE III
GPU KERNEL RUNTIMES WITHOUT ENABLING ALL 3 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

figure, N stands for the threshold of 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 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, there being unused warps is the main challenge for this problem. Actually the newly proposed three modes of kernels have greatly improved utilization of threads in SM, despite some remaining mismatch due to unpredictable sparsity pattern of the matrix. This hurts the performance of 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 is decreasing, warp occupancy would drop naturally.

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 (215ms). For larger matrices, this problem
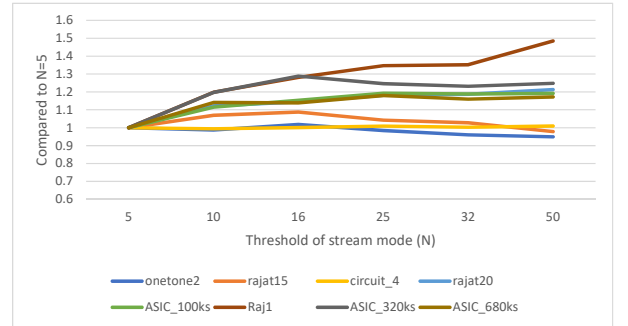

Fig. 12. Performance of GPU kernel with different stream mode threshold settings

should be less severe for larger matrices or in real simulation scenarios where the factorization kernel is called repeatedly.

V. CONCLUSION

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 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 University of Florida Sparse Matrix Collection (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 $\times$ (arithmetic mean) or 6.7 $\times$ (geometric mean) speedup over GLU2.0 and 7.1 $\times$ (arithmetic mean) or 4.8 $\times$ (geometric mean) over recent proposed enhanced GLU2.0 sparse LU solver on the same set of circuit matrices.

References

[1] SPICE – the general-purpose circuit program for analog integrated circuits . http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/.
[2] David B. Kirk and Wen-Mei Hwu. *Programming Massively Parallel Processors: A Hands-on Approach, 2ed*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 2013.
[3] NVIDIA Tesla's Servers and Workstations. http://www.nvidia.com/object/tesla-servers.html.
[4] Dominik Göddeke. General-purpose computation using graphics hardware. http://www.gpgpu.org/, 2011.
[5] K. He, S. X.-D. Tan, H. Wang, and G. Shi. GPU-accelerated parallel sparse LU factorization method for fast circuit analysis. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 24(3):1140–1150, March 2016.
[6] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.
[7] http://crd.lbl.gov/ xiaoye/superlu/.
[8] T. A. Davis and E. Palamadai Natarajan. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Trans. Mathematical Software*, pages 36:1–36:17, September 2010.
[9] J. R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Statist. Comput.*, pages 862–874, 1988.
[10] 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*, (10):702–706, October 2011.
[11] Xiaoming Chen, Yu Wang, and Huazhong Yang. NICSLU: An adaptive sparse matrix solver for parallel circuit simulation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 32:261–274, February 2013.
[12] N Galoppo, N.K 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*, page 3, 2005.
[13] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, June 2010.
[14] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense Linear Algebra Solvers for Multicore with GPU Accelerators. In *Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.
[15] D Yu Chenhan, Weichung Wang, and Danl Pierce. A cpu–gpu hybrid approach for the unsymmetric multifrontal method. *Parallel Computing*, 37(12):759–770, 2011.
[16] Thomas George, Vaibhav Saxena, Anshul Gupta, Amik Singh, and Anamitra R Choudhury. Multifrontal factorization of sparse spd matrices on gpus. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 372–383. IEEE, 2011.
[17] Robert F Lucas, Gene Wagenbreth, Dan M Davis, and Roger Grimes. Multifrontal computations on gpus and their multi-core hosts. In *International Conference on High Performance Computing for Computational Science*, pages 71–82. Springer, 2010.
[18] Ling Ren, Xiaoming Chen, Yu Wang, Chenxi Zhang, and Huazhong Yang. Sparse LU factorization for parallel circuit simulation on GPU. In *Proc. Design Automation Conf. (DAC)*, pages 1125–1130, 2012.
[19] Xiaoming Chen, Ling Ren, Yu Wang, and Huazhong 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.
[20] GLU – GPU-Accelerated Sparse Parallel LU Factorization Solver. http://www.ee.ucr.edu/~stan/project/glu/glu_proj.htm.
[21] Wai-Kong Lee, Ramachandra Achar, and Michel S Nakhla. Dynamic gpu parallel sparse lu factorization for fast circuit simulation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, (99):1–12, 2018.
[22] T. A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, Philadelphia, 2006.
[23] Y. Xiao, Y. Xue, S. Nazarian, and P. Bogdan. A load balancing inspired optimization framework for exascale multicore systems: A complex networks approach. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 217–224, Nov 2017.
[24] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 2011.

Shaoyi Peng (S'17) received his B.S degree in Microelectronics from Fudan University, Shanghai, China in 2016. He is currently a Ph.D. candidate in the Department of Electrical and Computer Engineering at the University of California, Riverside. His research interest includes VLSI reliability effect modeling and simulation, finite element method analysis and numerical methods.
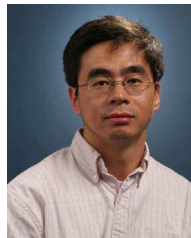
Sheldon X.-D. Tan (S'96-M'99-SM'06) received his B.S. and M.S. degrees in electrical engineering from Fudan University, Shanghai, China in 1992 and 1995, respectively and the Ph.D. degree in electrical and computer engineering from the University of Iowa, Iowa City, in 1999. He is a Professor in the Department of Electrical Engineering, University of California, Riverside, CA. He also is a cooperative faculty member in the Department of Computer Science and Engineering at 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 many-core processors, parallel computing and adiabatic and Ising computing based on GPU and multicore systems. He has published more than 300 technical papers and has co-authored 6 books on those areas.

Dr. Tan received NSF CAREER Award in 2004. He also received three Best Paper Awards from ICSICT'18, ASICON17, ICCD'07, DAC09. He also received the Honorable Mention Best Paper Award from SMACD18. He was a Visiting Professor of Kyoto University as a JSPS Fellow from Dec. 2017 to Jan. 2018. He is serving as the TPC Chair of ASPDAC 2021, amd TPC Vice Chair of ASPDAC 2020. He is serving or served as Editor in Chief for Elseviers Integration, the VLSI Journal, the Associate Editor for three journals: IEEE Transaction on VLSI Systems (TVLSI), ACM Transaction on Design Automation of Electronic Systems (TODAES) and Elseviers Microelectronics Reliability.