Dynamic GPU Parallel Sparse LU Factorization for Fast Circuit Simulation

Wai-Kong Lee, *Member, IEEE*, Ramachandra Achar, *Fellow, IEEE*, and Michel S. Nakhla, *Life Fellow, IEEE*

***Abstract*— Lower–upper (LU) factorization is widely used in many scientific computations. It is one of the most critical modules in circuit simulators, such as the Simulation Program With Integrated Circuit Emphasis. To exploit the emerging graphics process unit (GPU) computing platforms, several GPU-based sparse LU solvers have been recently proposed. In this paper, efficient algorithms are presented to enhance the ability of GPU-based LU solvers to achieve higher parallelism as well as to exploit the *dynamic parallelism* feature in the state-of-the-art GPUs. Also, rigorous performance comparisons of the proposed algorithms with GLU as well as KLU, for both the single-precision and double-precision cases, are presented.**

***Index Terms*— Circuit simulation, graphics processing unit (GPU), lower–upper (LU) factorization, multicore, parallel simulation, Simulation Program With Integrated Circuit Emphasis (SPICE), sparse matrices.**

I. Introduction

MANY numerical and scientific computing applications require the solution of large linear algebraic system of equations, which can be computationally expensive. Particularly, for circuit simulation applications, one of the most critical modules is the solution of large but sparse set of algebraic equations ($\boldsymbol{Ax} = \boldsymbol{b}$) resulting from a frequency-domain analysis or a time-domain analysis. Lower–upper (LU) factorization is an efficient way to solve these equations by first transforming the matrix $\boldsymbol{A}$ into lower ($\boldsymbol{L}$) and upper ($\boldsymbol{U}$) matrices and subsequently computing the solution through forward and backward substitutions (FBSs). LU factorization remains as one of the main computational cost components in a Simulation Program With Integrated Circuit Emphasis (SPICE)-based simulator. This is because a time-domain analysis may require thousands of time points, and each time point solution in turn may need several Newton–Raphson (NR) iterations wherein each NR iteration requires an LU factorization.

Graphics processing unit (GPU) is traditionally used to accelerate the computation in graphics and video applications. Since the release of CUDA [1] in 2006, GPU is quickly becoming a preferred computational platform in many scientific applications due to its massively parallel architecture (owing to thousands of cores) and large memory bandwidth. However, parallelizing LU factorization of sparse matrices in GPU is not straightforward due to the strong data dependence and irregular memory access pattern.

There are several previous works targeting LU factorization using GPU platforms [2]–[4]. These approaches rely on forming multifrontal dense matrices [5] from the sparse matrices and utilize dense basic linear algebra subprograms [6] to perform LU factorization on these dense subblocks. However, sparse matrices in circuit simulation can hardly form dense subblocks, making these techniques not suitable for circuit application. KLU [7] is one of the most widely used sparse direct solvers specially targeting circuit matrices, which adopts the Gilbert and Peierls (G/P) left-looking algorithm [8]. Although KLU is faster than many other sequential solvers for circuit matrices [7], it does not offer any easy route for parallel execution. One of the workarounds to utilize KLU in multicore CPUs is through domain decomposition [9] and node tearing-based matrix partitioning techniques [10].

Chen *et al.* [11] proposed NICSLU, a parallel version of G/P left-looking algorithm in multicore CPU platforms. NICSLU was subsequently implemented in GPU platform [12] for LU factorization of circuit matrices. Although G/P left-looking algorithm enjoys good performance when implemented in GPU platforms, the second *for* loop (index j in Algorithm 1) [11] in the LU factorization is inherently serial, and hence it does not allow for fine grain parallelization. To address this, He *et al.* [13] proposed hybrid column-based right-looking algorithm [or GPU-based sparse LU (GLU)]. Recently, NVIDIA also released an official sparse matrix solver, cuSolver [14], but the LU factorization in it is still performed in CPU instead of GPU.

In this paper, the state of the art in GPU LU solvers is advanced so as to increase parallelism, to reduce the CPU-GPU launch overhead, and also to take advantage of the new *dynamic parallelism* feature in the newer GPUs. The contributions are listed in the following along with the context of the existing work in the literature.

1) GLU [13] utilizes the CPU to launch the GPU *kernel* level by level, until the entire matrix is factorized. This requires the CPU to manage the *kernel* launches explicitly, which can excessively consume valuable CPU resource for large circuit matrices. To address this shortcoming, we propose to offload the entire LU factorization into GPU by utilizing the *dynamic parallelism*

Manuscript received March 4, 2018; revised May 30, 2018; accepted July 1, 2018. Date of publication August 9, 2018; date of current version October 23, 2018. This work was supported by the Natural Sciences and Engineering Research Council of Canada. *(Corresponding author: Ramachandra Achar.)*

W.-K. Lee is with the Faculty of Information and Communication Technology, Universiti Tunku Abdul Rahman, Petaling Jaya 31900, Malaysia (e-mail: wklee@utar.edu.my).

R. Achar and M. S. Nakhla are with the Department of Electronics, Carleton University, Ottawa, ON K1S 5B6, Canada (e-mail: achar@doe.carleton.ca; msn@doe.carleton.ca).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2018.2858014

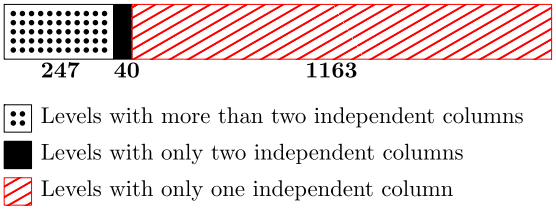

Fig. 1. Illustrative example of the number of columns in different levels in a circuit matrix (ASIC_680ks [27]: matrix of size 682712 × 682712 with 1450 levels).

TABLE I
ILLUSTRATION OF THE NUMBER OF INDEPENDENT COLUMNS IN VARIOUS DEPENDENCE LEVELS

Circuit Matrix	Matrix size (# of rows)	Total # of levels	Levels with > 2 columns	Levels with two columns	Levels with one column
rajat12	1872	37	23	**1**	**13**
memplus	17258	147	126	**5**	**16**
rajat26	51032	157	68	**1**	**88**
rajat20	86916	1216	183	**22**	**1011**
Raj1	263743	1594	493	**307**	**794**
ASIC_680ks	682712	1450	247	**40**	**1163**

feature [1] in the state-of-the-art GPUs. Under this execution framework, the CPU launches only the parent GPU *kernel*, then subsequent *kernels* are all launched and managed by the GPU itself. This approach frees up the CPU resources to be available for other tasks. In addition, this leads to further reduction in the *kernel* launch overhead, since launching *kernels* from GPU have less overhead compared to launching *kernels* from CPU.

2) In the previous work GLU [13], GPU *kernels* are launched level by level. It is to be noted that in the case of large circuit matrices, typically first few levels will have many independent columns whereas the large number of subsequent levels will have only two columns or only one column. This seriously undermines the parallel processing of columns, underutilization of GPU resources, and excessive overhead in *kernel* launches. This is further shown in Fig. 1, which shows the proportionate composition of levels with different number of columns for a practical circuit with a large number of nodes. It can be clearly seen that approximately 80% of the levels are left with only one column. Table I further shows the related statistics for circuit matrices of varying sparsity and sizes. As can be seen, as the circuit size increases, the number of levels with only one or two columns significantly increases.

In order to address this shortcoming of many levels with only one or two columns, following contributions are presented: 1) while starting to process the levels with only two columns, instead of launching the kernel level by level, it is proposed to launch these kernels in *batch mode* and compute them one after another as the sequencing permits and 2) to handle the situation of levels with only one column, a pipeline-based algorithm is presented to launch these levels in batch and then compute them in sequence. These new approaches help to reduce the kernel launch overhead and also to achieve more parallelism compared to the previously published implementations.

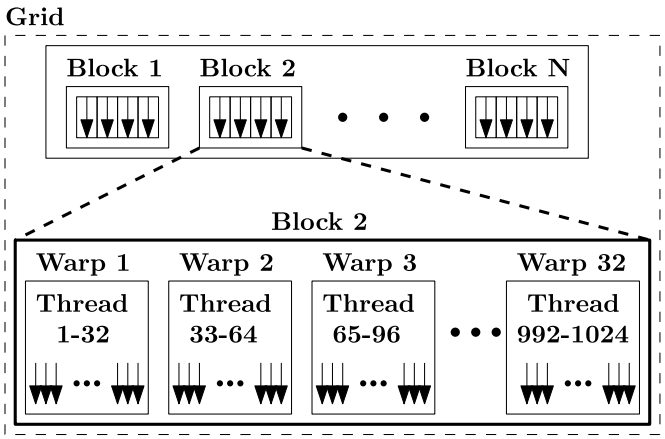

Fig. 2. CUDA programming model: grid, block, warp, and thread architecture.

A rigorous comparative study on variety of circuits using the proposed approaches as well as the *GLU v2.0* [15] for both the single-precision and double-precision cases is presented. The results demonstrate a significant speedup using the proposed approaches. Also, a further comparison of the results with the KLU [7] is provided, which paves the way for judicious and efficient use of CPU and GPU resources for sparse LU factorization.

II. BACKGROUND AND PROBLEM FORMULATION

In this section, a brief overview of the GPU architecture, a CUDA programming model, SPICE transient simulation flow, the left looking as well as hybrid right-looking sparse LU factorization algorithms, is presented.

A. GPU Architecture and CUDA Programming Model

GPU is fast emerging as one of the preferred computing platforms for the scientific community with many innovative applications emerging from different areas. This includes artificial intelligence [16], scientific simulation [17], [18], cryptography [19], integrated circuit analysis [20], and medical imaging [21]. This section describes the main characteristics of the GPU architecture, including its programming model, memory hierarchy, and special features that are relevant to the proposed work presented in this paper.

1) CUDA Programming Model: A generic architecture for the CUDA programming model is described in Fig. 2. In this architecture, a *block* consists of multiple *threads* (up to a maximum size of 1024 threads) and a *grid* consists of multiple blocks. Threads within a block are further grouped into *warps* (32 threads form one *warp*, which is the scheduling unit in a GPU hardware). All threads within the same *warp* are scheduled to execute the same instruction. Each thread and *block* can be indexed individually using the built-in variables (*threadIdx* and *blockIdx*). The GPU follows the single instruction multiple data execution model, wherein each thread is executing the same program while operating on different sets

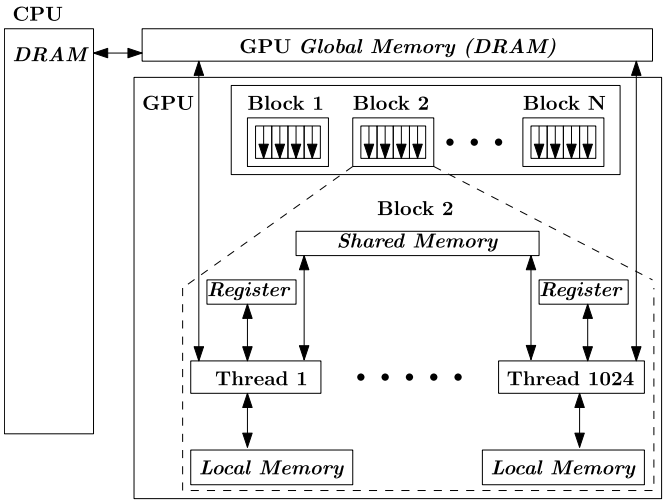

Fig. 3. GPU memory architecture.

of data. For a multi-GPU system, there will be multiple grids. The GPU program is usually referred to as a *kernel*.

2) Memory Hierarchy: The memory hierarchy in GPU has several layers of memory, which is different from that of the CPU and is described in Fig. 3. GPU memory can be divided into off-chip memory and on-chip memory, which have a huge difference in their performance.

Global memory is placed outside the chip and it provides larger storage capacity. However, it is slower in performance. The data used for GPU computation are usually transmitted from CPU DRAM to GPU global memory (which is also a DRAM). Once the GPU computes the results, they are transferred back to CPU DRAM. It is to be noted that the transfer of data between GPU and CPU happens only through their respective DRAMs.

Shared memory is an on-chip memory accessible by all threads within the same thread block. It is usually used as a user-managed cache for high-performance computations. If shared memory usage is designed carefully to avoid bank conflicts, it can provide much faster access speed compared to global memory.

Registers are on-chip and are the fastest among the memories in GPU. However, they are very limited in size and only accessible locally by each thread. If a *kernel* uses more registers than its allowed limit, the compiler will allocate these extra register usages into "local memory" that resides in the slow global memory but cached at L1 cache for faster access.

3) Dynamic Parallelism: The GPU architecture has evolved through several generations [commonly referred to as compute capability (CC)], including Fermi (CC 2.x), Kepler (CC 3.x), Maxwell (CC 5.x), and recently Pascal (CC 6.x) [1]. Each new GPU architecture offers a better performance compared to the old one by advancing the hardware design, manufacturing process, and architectural changes. Besides that, new GPU architectures also provide advanced features that can be further leveraged for accelerating the implementation of many scientific algorithms.

Conventionally, the GPU *kernel*s are launched and managed by the CPU, which can consume substantial portion of the CPU computational resources. This also implies that the workload (the number of blocks and threads) to be offloaded to the GPU has to be determined *a priori*. This becomes a serious limitation while implementing many algorithms (e.g., graph traversal and heat flow simulation), as the workload for next iteration is dynamic (i.e., difficult to predict *a priori*). Hence, the usual workaround to this limitation is to overestimate the workload so that there are always sufficient tasks to keep GPU resources occupied. The newer NVIDIA GPUs with CC 3.5 and above offer an advanced feature named *dynamic parallelism*, which allows the GPU *kernel* to launch another GPU *kernel* by itself. With *dynamic parallelism*, the CPU needs to launch the parent *kernel* only once, then the GPU *kernel* can manage the subsequent *kernel* launches within the GPU itself. This approach frees up CPU resources for other tasks and also allows the algorithm with dynamic workload to be designed for optimality as they no longer need to overestimate the workload. Unlike in older GPU architectures, *dynamic parallelism* allows recursive function call within GPU.

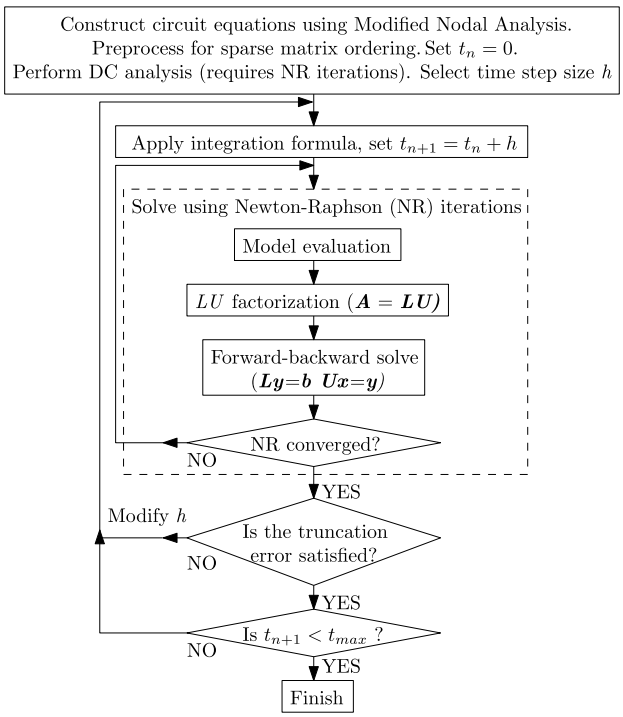

Fig. 4. Illustration of LU factorization requirements in a SPICE transient simulation.

In this paper, the proposed algorithms are designed to exploit the *dynamic parallelism* feature to launch and manage the *kernel* within the GPU. This helps to reduce the *kernel* launch overhead and also frees up the CPU for other tasks.

B. SPICE Transient Simulation

One of the critical modules during circuit simulation is the LU factorization, which may require to be repeated thousands of times during a transient simulation. Fig. 4 shows the use of NR iterations and LU factorization in a typical SPICE-based transient simulation flow.

To start with, circuit equations are formulated using a modified nodal analysis (MNA) [23], which often results in the form of nonlinear differential equations. Using an appropriate

time-step size h (to set the next time point, $t_{n+1} = t_n + h$) and a suitable integration formula (such as backward Euler, trapezoidal rule, and so on) at each time point, MNA equations are translated to a set of nonlinear algebraic equations. These equations are solved using NR iterations. Generally, depending on the required time span of simulation and the nature of the circuit, thousands of time point solutions may be required with each time point requiring several NR iterations (typically in the range of 3–4 iterations per time step). Each NR iteration results in a set of linear equations ($\boldsymbol{Ax} = \boldsymbol{b}$), which needs to be solved for the vector $\boldsymbol{x}$ containing the unknown voltages and other MNA variables. Here, $\boldsymbol{A}$ tends to be large but sparse. $\boldsymbol{b}$ is a function of the response at previous time points and the source vector. Popular approach to solving these equations in circuit simulation is to first obtain the LU factors of $\boldsymbol{A}$ and subsequently use FBS to solve for the unknown vector $\boldsymbol{x}$. However, LU factorization can be CPU expensive for large circuit matrices and can make the transient analysis a slow process, as it may require thousands of LU factorizations.

Focus of this paper is on adopting the GPU platform for efficient LU factorization for circuit simulation. For this purpose, as outlined in the literature, the matrix $\boldsymbol{A}$ is first reordered to reduce the fill-in, then prescaled using the HSL MC64 algorithm [24], [25] to improve the numerical stability. Next, the first LU factorization is performed to obtain the information on nonzero locations. These steps are performed "one time" in CPU in order to obtain the dependence-level information of all columns [24], [25]. The subsequent LU factorizations without pivoting (also called *refactorization*) are performed using GPU. Since the structure of the circuit matrix is fixed during circuit simulation, the same nonzero patterns obtained in the first LU factorization are used for subsequent refactorizations. The preprocessed matrix $\boldsymbol{A}$, together with the dependence-level information, is transferred to the GPU global memory for refactorization. After the refactorization (which will not generate any further new fill-ins), GPU transfers the LU factors back to the CPU, followed by FBS in CPU to solve for the unknown vector $\boldsymbol{x}$. The refactorization and the FBS are repeated many times until the NR iterations are converged and the transient simulation is completed. In Section II-C, state of the art in GPU-based LU factorization is reviewed.

C. Sparse LU Factorization Algorithms

G/P [8] left-looking algorithm for sparse matrices has been widely adopted by state-of-the-art sparse LU solvers, including KLU [7] and NICSLU [11], [12]. Algorithm 1 shows the pseudocode for the in-place version of G/P [8] left-looking algorithm, in which the $\boldsymbol{L}$ and $\boldsymbol{U}$ factors are stored directly into the matrix $\boldsymbol{A}$ replacing its original values.

Referring to Algorithm 1, the left-looking method scans each column from left to right (index i) and calculates both $\boldsymbol{L}$ and $\boldsymbol{U}$ factors for the corresponding column. This is achieved by processing a triangular matrix (lines 3–8) to obtain the $\boldsymbol{U}$ factors and subsequently computing the $\boldsymbol{L}$ factors by dividing by the corresponding diagonal element (line 11). Hence, the algorithm *always looks left* for all previously computed columns j in order to calculate the $\boldsymbol{L}$ and $\boldsymbol{U}$ factors for the current column (index i).

Algorithm 1 G/P Left-Looking Algorithm

```
1:  // Scan each column from left to right
2:  for i = 1 to n do
3:      // Compute triangular solve for U
4:      for j = 1 to i - 1 where A(j,i) ≠ 0 do
5:          for k = j + 1 to n where A(k,j) ≠ 0 do
6:              A(k,i) = A(k,i) − A(k,j) × A(j,i)
7:          end for
8:      end for
9:      // Compute column i for L
10:     for k = i + 1 to n where A(k,i) ≠ 0 do
11:         A(k,i) = A(k,i)/A(i,i)
12:     end for
13: end for
```

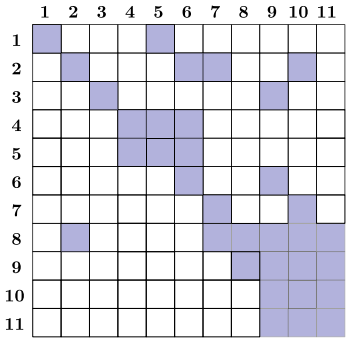

Fig. 5. Example matrix.

TABLE II
COLUMN DEPENDENCE LEVELS

Level	Task Nodes (Columns)	
	LLA [8], [12]	RLA [13]
1	1, 2, 3, 4, 8	1, 2, 3, 4
2	5, 7	5, 7
3	6	6, 8
4	9	9
5	10	19
6	11	11

The column dependence for the left-looking algorithm can be determined by knowing the nonzero pattern of the $\boldsymbol{U}$ matrix. This algorithm benefits from a symbolic fill-in analysis of $\boldsymbol{L}$ and $\boldsymbol{U}$ factors before the actual numerical factorization begins. In other words, since the dependence between each column is known before the numerical factorization, any independent columns can be factorized in parallel. For the purpose of illustration, Fig. 5 shows an example matrix that is grouped into multiple levels based on its column dependence (shown in Table II). For example, columns 1–4 and 8 have no dependence on any other and hence they can be executed first in parallel (level 1). Column 5 depends on columns 1 and 4, and hence it needs to be postponed to level 2. Column 6 depends on columns 2, 4, and 5; hence it needs to be placed in level 3 (i.e., after level 2 which contains column 5). Similarly, dependence level of other columns can be determined.

Each level is launched in serial, while all the columns within the same level are executed in parallel (since there are no dependences among the columns in a given column).

Algorithm 2 Hybrid Column-Based RLA [13]

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

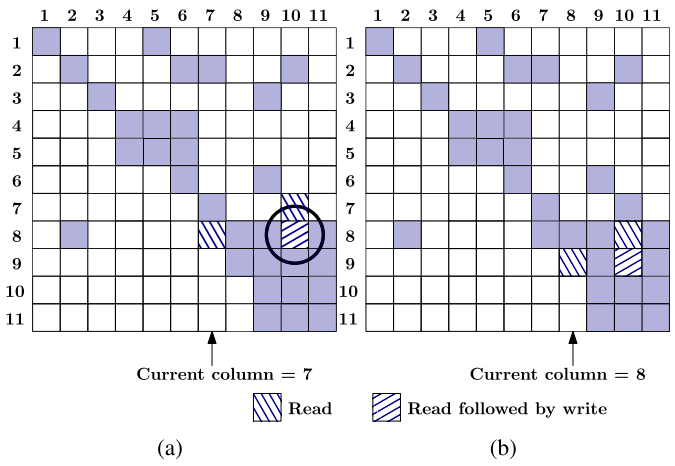

Fig. 6. Additional dependence for hybrid column-based RLA.

This corresponds to the outer most *for* loop in Algorithm 1 (index i), which is usually termed *column-level parallelism* or *cluster* mode launch [11].

However, one of the major difficulties encountered in this implementation is that, for large sparse circuit matrices, typically many levels will end up with only one column after the first few levels (for example, see Table I and Fig. 1 and also levels 3–6 in Table II). This essentially leads to serial execution of majority of the columns. To alleviate this situation, *"pipeline mode parallelism"* was adopted in [11] and [12] to speed up the execution of these levels with only one column. In this scheme, once a column completes its computation, the corresponding results are immediately forwarded to the column that depends on it. This allows the dependent column to start its execution partially in a pipeline fashion. Also, the multiply-and-accumulate (MAC) operation (lines 5–7) is executed in parallel to enable more parallelism. However, the j loop (line 4) remains the major bottleneck, which still has to be executed column by column, in a serial fashion.

He *et al.* [13] proposed hybrid column-based right-looking algorithm (RLA) to provide additional parallelism in GPU platforms, which is described in Algorithm 2. This algorithm starts by computing the $\boldsymbol{L}$ factors for the current column, then updates the submatrix on the right of the current column via the MAC operations (hence it is named "right looking method"). Unlike the conventional right-looking method, this algorithm suggested exploiting the column-based parallelism similar to the left-looking method [8] and parallelization of the two *for* loops in submatrix update. The corresponding implementation of this algorithm was released by the authors to the public domain as "*GLU v1.0*," whose speedup was reported in [13]. However, this implementation faced difficulty due to the inaccurate dependence detection, leading to inaccurate results. This is because the dependence leveling of the RLA is different from that of the left-looking method, as it not only depends on the nonzero pattern of the $\boldsymbol{U}$ matrix but also the nonzero pattern of the $\boldsymbol{L}$ matrix. This is shown in Fig. 6 using the same example matrix of Fig. 5.

Referring to Fig. 6, if the left-looking method is used, computing column 7 does not affect the values in column 8, leaving columns 7 and 8 to be independent, resulting in the dependence levels as in Table II. However, if RLA is used, column 8 cannot be placed in a level prior to the level containing column 7 due to the "read-after-write" data hazard. This is illustrated with respect to the circled element in Fig. 6. If column 7 is in action, after it computes its $\boldsymbol{L}$ matrix update, it proceeds to update the circled element [MAC operations, see Fig. 6(a)]. However, if column 8 was also to be run in parallel at the same time, it could end up reading the nonupdated circled element to perform its own MAC operation [see Fig. 6(b)] to update the element below the circled element. Such a "read-after-write" data hazard can lead to inaccurate results and should be avoided, if there is dependence between these two columns. As a result, the column 8 needs to be moved to level 3 (so as to process it after column 7 completes, as shown in Table II). To accommodate these concerns, He *et al.* [13] released "*GLU v2.0*" [15]. However, the performance and the speedup from "*GLU v2.0*" were lower than the one reported in [13].

Further observations and analysis are based on the published literature [13] and the software (*GLU v2.0*) [15]. It is to be noted that, even in this implementation, after the proper dependence leveling process, many levels end up containing only one or two columns, causing the algorithm to suffer from the similar disadvantages of the left-looking algorithm [11], [12]. GLU launches the GPU *kernel* level by level (serially), which implies that the GPU is mostly idle if there are only one or two columns available for computation. Moreover, launching many small *kernel*s in GPU can be expensive as each *kernel* launch introduces its own overhead, which is likely to accumulate when the circuit size grows (i.e., more levels with only one column). In order to address these difficulties, some new algorithms are presented in Section III.

III. Development of the Proposed Dynamic Parallel Sparse LU Factorization on GPU

In this section, we present the proposed dynamic parallel LU factorization along with pipeline and *batch mode* algorithms to improve the hybrid column-based RLA.

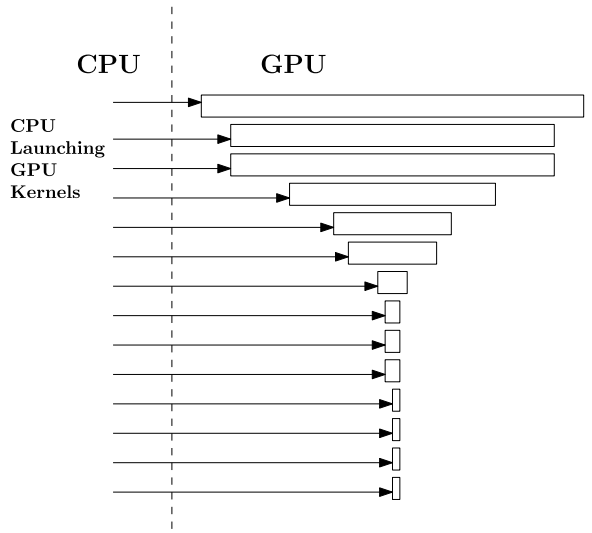

Fig. 7. CPU-managed *kernel* launch [13].

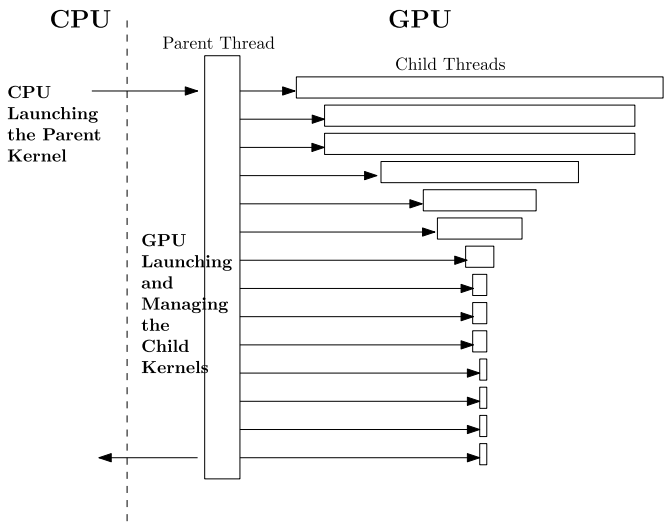

Fig. 8. Proposed GPU-managed *kernel* launch.

A. Proposed GPU Managed GPU Kernel Launches

The previous implementation [13], [15] utilized CPU to launch the GPU *kernels* level by level, as shown in Fig. 7. The CPU first launches the GPU *kernels* to compute many independent columns in the first level, then waits for it to complete before launching the second level. This process continues until all levels are completed. We observed that the above process hinders the performance as the CPU needs to closely monitor the GPU *kernels* in each level and time their launch in a serial manner.

However, as discussed in Section II-C, in most cases, the number of levels quickly grows with the increasing circuit sizes (refer to Fig. 1 and Table I), making the CPU managed *kernel* launches very inefficient.

To address the above-mentioned difficulties, we propose to utilize the *dynamic parallelism* feature to launch and manage the GPU *kernels* (for more details see Section II-A3), leading to the following major benefits.

1) Launching *kernels* within GPU creates lesser overhead compared to launching *kernels* from CPU, because the need to transfer execution control between CPU and GPU is reduced [1], thus reducing the communication through the PCIe bus. This benefit becomes significant when the number of levels (which influences the number of *kernel* launches) increases due to increase in circuit sizes.
2) The CPU only needs to launch a parent *kernel* to manage the subsequent *kernel* launches; it is no longer required to monitor the GPU execution closely. The subsequent *kernel* launches are all performed by the GPU parent *kernel*. This frees up the CPU resources to be available for other tasks.
3) In the previous implementations, the workload for the GPUs needed to be determined *a priori*. This is a hindrance in achieving higher performance as the optimum load design is quite challenging, since it heavily depends on the sparsity of the matrix as well as the number of columns in a level. With the proposed use of the *dynamic parallelism* feature, the workload for the GPUs no longer need to be determined *a priori* (i.e., the number of blocks, warps, and so on). This allows for efficient implementation of the proposed batch and *pipeline mode* algorithms for the right looking LU factorization.

The execution flow of the proposed GPU-managed *kernel* launches is shown in Fig. 8. This technique is applied to all the execution modes (cluster, batch, and pipeline) that are presented in Section III-B.

B. Proposed Batch and Pipeline Modes for Hybrid Right-Looking Algorithm

As is evident from the discussions in the introduction (see Fig. 1 and Table I), after the dependence leveling phase, there are potentially many levels with only one or two columns. The previous works from [13] and [15] proposed to launch the GPU *kernels* level by level, which will become inefficient for cases with a large number of levels with only one or two columns, as they do not fully utilize the GPU resources. Also, launching many GPU *kernels* in serial may introduce serious overhead. Hence, we are motivated to improve these aspects by proposing the *batch mode* for levels with only two independent columns and the *pipeline mode* for levels with only one column. For this reason, in the proposed approach, we further classify the levels into three types: *more than two columns, two columns*, and *one column*. Based on this classification, different types of execution modes are adopted, which are shown in Fig. 9, and the corresponding proposed execution methods are detailed in the following.

1) Cluster Mode Execution: The levels with many columns are executed in the *cluster mode*, similar to the one in previous works [12], [13], [15]. Referring to Algorithm 2, each *GPU block* (see Section II-A for details on *block* and *thread* organization) computes one column, which corresponds to the parallelization of *i* loop. Within each block, factorization of a given column starts with the computation of the $\boldsymbol{L}$

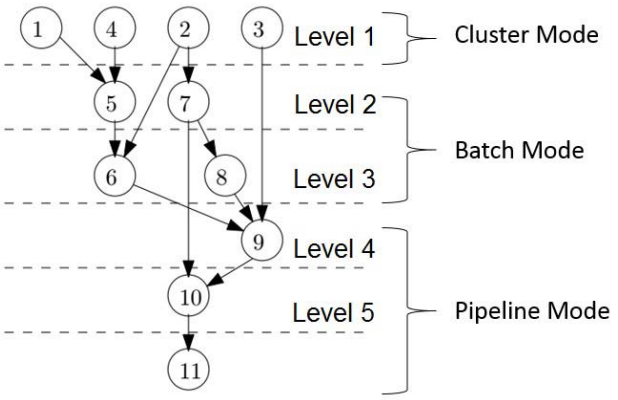

Fig. 9. Classification of execution modes.

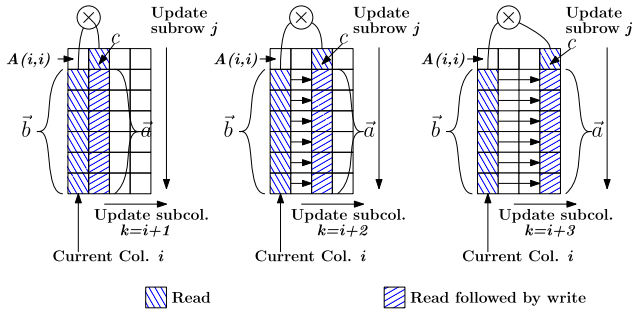

Fig. 10. Submatrix update in hybrid column-based RLA.

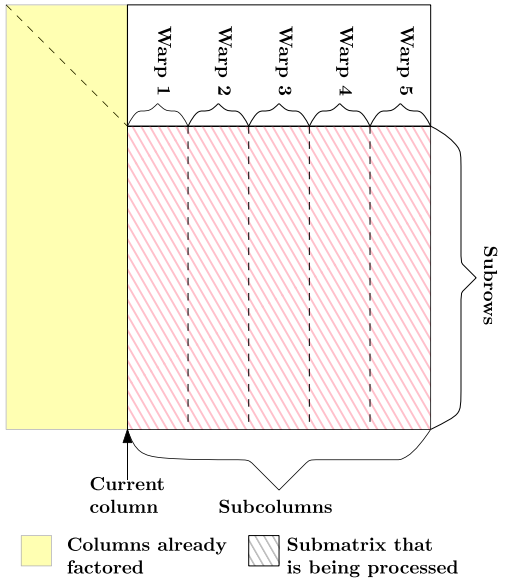

Fig. 11. Processing of a column in the cluster mode.

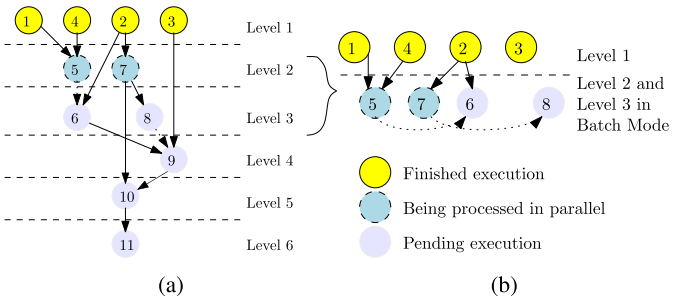

Fig. 12. Proposed *batch mode* execution.

matrix update (lines 4–6). For this purpose, each thread in the block is first assigned to compute the $\boldsymbol{L}$ matrix update independently. This corresponds to parallelizing the first j loop in Algorithm 2.

The next step is to perform submatrix update for all columns to the right of the current column i and is shown in Fig. 10. This update corresponds to the j and k loops (lines 8–12) in Algorithm 2. The main computation in these steps is the MAC operation, which is represented as $\vec{a} \leftarrow \vec{a} - \vec{b} \times c$ in the ensuing discussions. As shown in Fig. 10, vector $\vec{a}$ is first being read and then updated by subtracting its value by the multiplication of $\vec{b}$ [the vector of elements below the diagonal $\boldsymbol{A(i, i)}$] and the element c. This operation is indicated by the vertical downward arrow with the tag "update subrow j" in Fig. 10. These operations correspond to the j loop and can be performed in parallel. Note that the update of each subcolumn is also independent of other subcolumns, and hence the k loop can also be executed in parallel (indicated by horizontal rightward arrow at the bottom of figures in Fig. 10, with the tag "update subcol k").

Processing of each column is further shown in Fig. 11. Here, each *warp* (consists of 32 threads) is assigned to compute a subcolumn within the submatrix; multiple *warps* update the submatrices in parallel (corresponds to the k loop). Within each warp, 32 threads perform the atomic MAC operation (line 10) in parallel, while executing the j loop (lines 9–11).

2) Proposed Batch Mode Execution: For levels with only two independent columns, we propose to launch their *kernels* in the batch mode instead of level by level (i.e., a separate *kernel* for each level is launched at the same time, thus reducing the total *kernel* launch time), while columns in them are executed in GPU in parallel or sequence based on the dependence among them. For the purpose of illustration, consider Fig. 12(a), where levels 2 and 3 (each of which have two independent columns) are executed in the *batch mode*. Under the *batch mode* operation, the GPU *kernels* for these two levels are launched at the same time, wherein the execution of columns 5 and 7 starts immediately, followed by columns 6 and 8, as shown in Fig. 12(b). Since the columns are already arranged in a proper order based on the dependence graph, the result is guaranteed to be correct if the proper sequence is followed during the execution. This approach gains performance by minimizing the launch and process times associated with both the *kernels*.

A pseudocode for the *batch mode* execution for levels with only two columns is given in Algorithm 3). An array (*batchFlag*) is used to keep track of the execution status of each column in the queue. The first element in *batchFlag* is always set to true, while the rest of the elements are initialized to false. This implies that the first column in the queue always gets executed immediately, while other columns wait; all columns check the status in *batchFlag* continuously through a blocking while loop (line 2). Once the first column completes the

Algorithm 3 Algorithm for Batch Mode Implementation

```
1: // Launch the levels with only two columns in batch
2: while(batchFlag[blockIdx]==false);
3: for Current col in parallel do
4:     Compute L matrix for current col
5: end for
6: Synchronize all threads
7: // Update the submatrix
8: for all subcols in current col in parallel do
9:     Update elements in one subcol
10: end for
11: // Inform next column in the queue to start
12: batchFlag[blockIdx+1]=true;
```

execution, it updates the *batchFlag* array; then, the next column in the queue gets started.

Although this approach can offer a better performance compared to the existing implementation [13], we do not recommend extending this to other cases where the levels contain three or more independent columns. This is because the batch mode execution requires every thread to access the queuing array *batchFlag* frequently, which introduces some overhead. In case of batching levels with three or more columns, the corresponding performance gain due to the reduction in launch time is not often sufficient to offset the increased overhead in accessing the queuing array.

3) Proposed Pipeline Mode Execution: In the proposed method, GPU *kernels* for levels with only one column are also being launched in batches instead of level by level, wherein the columns are executed in a *pipeline mode* in GPU. This is based on the observation that the computation of $\boldsymbol{L}$ matrix update (lines 4–6 in Algorithm 2) for the next column can immediately start once its diagonal element is processed by the previous column, without waiting for the previous column to complete its full MAC operations to update the submatrix right to it.

The proposed *pipeline mode* execution is illustrated using the same example of Fig. 5 in Fig. 13. The *kernels* for column 9–11 are launched simultaneously, but only the execution of column 9 starts immediately [see Fig. 13(a)], while the execution of columns 10 and 11 is kept pending. Once column 9 has updated the entire subcolumn 10 [see Fig. 13(b)], it continues to update the subcolumn 11 [see Fig. 13(c)]; at the same time, column 10 can start computing the $\boldsymbol{L}$ matrix update [see Fig. 13(b)]. However, column 10 needs to wait for column 9 to complete the computation on subcolumn 11 before it can proceed to update its submatrix, since they are updating the same location [see Fig. 13(c)] to avoid the read-after-write data hazard. Once column 9 had updated the subcolumn 11, column 10 can begin its execution for updating column 11 [see Fig. 13(d)]. This creates a pipeline work flow and yields a superior performance by reducing the number of separate GPU *kernal* launches, while executing levels with only one column.

A pseudocode for the proposed *pipeline mode* approach is given in Algorithm 4. Here, two arrays (*pipeFlagA*)

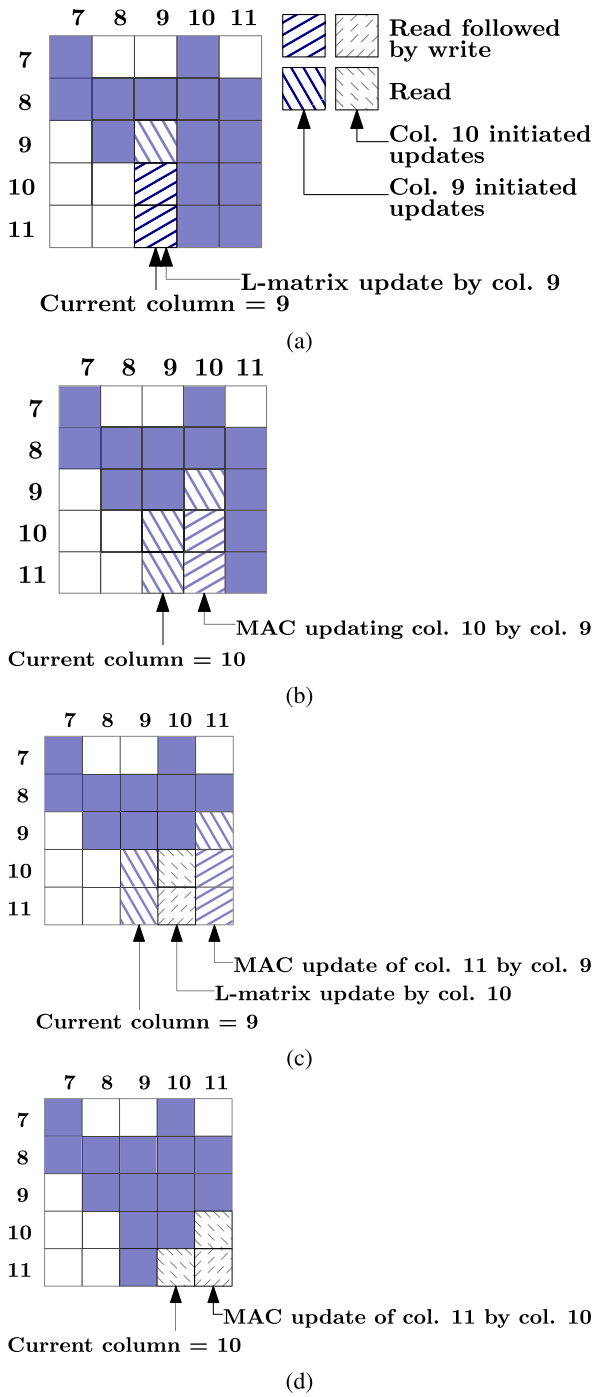

Fig. 13. Proposed *pipeline mode* execution for levels with only one column.

and (*pipeFlagB*) are used to keep track the execution status of each column in the queue. Similar to batch mode, the first element in these two arrays are always set to true to allow the first column in the queue to start immediately. Once the first column had computed the diagonal element of the next column in the queue and the elements below it (lines 11–13), it updates the *pipeFlagA*. At this point of time, next column can start its execution to compute the $\boldsymbol{L}$ matrix update, then wait again at line 7. Once the first column completes the computation of all its submatrix, it updates *pipeFlagB*, so that the next waiting column can start its computation. The same process continues until all other columns in the queue are updated.

Algorithm 4 Algorithm for Pipeline Mode Implementation

```
// Launch the levels with only one column in batch
while(pipeFlagA[blockIdx]==false);
for Current col in parallel do
    Compute L matrix for current textitcol
end for
Synchronize all threads
while(pipeFlagB[blockIdx]==false);
// Update submatrix
for all subcols in current col in parallel do
    Update elements in one subcol
    if Diagonal element for next col is updated then
        pipeFlagA[blockIdx+1]=true;
    end if
end for
// Inform the next column in the queue to start
pipeFlagB[blockIdx+1]=true;
```

C. Parallel Implementation in GPU

It is to be noted that the available resources for concurrent computation in GPU (active warps, shared memory, and threads per block) are limited, and hence the number of concurrent columns to be factorized should also be limited. For cluster mode execution, we follow the similar approach in GLU [15] that only a fixed number of columns N_{par} will be executed in parallel. If a particular level contains many independent columns which exceeds N_{par}, the factorization process then breaks into multiple executions. We extend this technique to the batch as well as pipeline modes (i.e., batch size is also limited to N_{par}). For the experiments performed in Section IV, we fixed $N_{\text{par}} = 32$, as this configuration shows the best results for our experimental platform (Pascal P100 GPU). For other GPU platforms, the optimal value for N_{par} can be determined experimentally.

Referring to Algorithm 2, the $\boldsymbol{L}$ matrix update of a current column (i.e., elements below the diagonal element) is used to update the submatrix to the right. The previous implementation in GLU [15] stores the values of $\boldsymbol{L}$ as a temporary vector in global memory to facilitate the subsequent submatrix update process. This temporary vector needs to be cleared to zero (using *cudaMemset* instruction) after factorization of the entire column, because the subsequent columns will use the same memory space for computation. However, we noticed that *cudaMemset* instruction is very time-consuming, as it clears the memory in a bytewise manner. Also noting the fact that $\boldsymbol{L}$ is sparse for most of the circuit matrices, not all elements in this temporary vector are affected, so we do not need to explicitly clear the entire memory space. Instead, we propose to launch an additional *kernel* after columnwise factorization is completed to clear the affected elements only. This provides additional improvement for the overall performance of GPU-based LU factorization.

As discussed in Section II-B, circuit simulation process typically requires several rounds of LU factorization in every NR iteration until the iterations converge. This process is repeated at every time point until the full simulation completes. For the numerical stability of the results, numerical precision plays an important role to ensure the accuracy of LU factors computed. The industrial grade solvers, such as KLU [7], are implemented in double precision. For this reason, we have implemented our proposed algorithm in both the double precision (to compare the results with KLU) as well as in single precision [to compare the results with the previously published work (GLU [13], [15]), which is available in single precision]. Numerical validation of the proposed methods is given in Section IV.

IV. Experimental Results and Discussion

The proposed algorithms are implemented in C language under CUDA 8.0 SDK and compiled with optimization level 3 (-O3). The performance was evaluated using the Compute Canada platform (a national computing grid) [28], which has a module configuration of four CPU cores (Intel Xeon E5-2650), 64-GB RAM, and a GPU. The GPU (Pascal P100) consists of 3584 cores, 4096 KB of L2 cache memory, and 12-GB DRAM and operates at 1.328-GHz clock frequency; 15 circuit matrices from the University of Florida Sparse Matrix Collection [27] are used to evaluate the proposed GPU-based LU factorization and compared with the *GLU v2.0 [15]* as well as CPU-based implementation (KLU [7]).

A. Performance Comparison of the Proposed Method and GLU V2.0 [15] (Single Precision)

In this experiment, we compare the performance of the proposed algorithms with the *GLU v2.0*, which is based on single-precision accuracy [15]). While performing this experiment, the matrices are stored in the CSR format and transferred to GPU global memory in every refactorization; the LU factors are written directly to the original matrix $\boldsymbol{A}$, which is also known as in-place computation. After refactorization, the LU factors are transferred back to CPU to solve for the unknown vector using FBS in CPU.

The results are presented in Table III, where *n* and *nnz* represent the matrix dimension (the number of rows) and the number of nonzeros in the matrix. The time shown in this table represents the time for GPU LU factorization, which includes the time spent for transferring the data from the CPU DRAM to GPU global memory, time taken for computing the LU factors in GPU, and the time taken for transferring the data (LU factors) from the GPU global memory back to the CPU DRAM.

As can be seen from the table, the proposed algorithms consistently yield a better performance compared to the *GLU v2.0*. Referring to Table III, it is to be noted that the reported speedup is due to all the algorithmic improvements proposed in this paper, including the dynamic parallelism, GPU-managed Kernel launches, and pipeline/batch mode execution enhancements. Also to be noted that the performance of LU factorization is highly dependent on the matrix characteristics (size of the circuit matrix, the nature of the circuit-sparsity pattern, and the number of nonzeros) and ordering techniques. For example, here *G3_circuit* is a relatively large matrix with many strongly connected nodes, which explains why the levels with one and two columns are very less. However, since the

TABLE III

PERFORMANCE COMPARISON WITH *GLU v2.0* (SINGLE PRECISION)

Matrix	Matrix size: # of rows (n)	No. of nonzero (nnz)	Dependency Level Information (a, b, c)*	Time (ms) Single precision		
				GLU v2.0	Pro-posed	Speed -up
rajat12	1879	12818	(23,1,13)	18	17	**1.06**
circuit_2	4510	21199	(63,6,32)	52	48	**1.08**
memplus	17758	99147	(16,5,126)	96	85	**1.13**
rajat27	20640	97353	(61,10,52)	112	97	**1.15**
onetone2	36057	222596	(140,61,1012)	3258	2997	**1.09**
rajat15	37261	443573	(257,155,554)	3176	2987	**1.06**
rajat26	51032	247528	(68,1,88)	601	413	**1.46**
circuit_4	80209	307604	(30,9,213)	1665	1583	**1.05**
rajat20	86916	604299	(183,22,1011)	19824	14352	**1.38**
ASIC _100ks	99190	578890	(277,4,1350)	15666	14891	**1.05**
hcircuit	105676	513072	(57,9,78)	774	453	**1.71**
Raj1	263743	1302464	(493,307,794)	67923	63182	**1.08**
ASIC _320ks	321671	1827807	(236,15, 1418)	27363	22433	**1.22**
ASIC _680ks	682712	2329176	(247,40, 1163)	45740	15038	**3.04**
G3_ circuit	1585478	7660826	(641,5,6)	110029	20384	**5.4**
					Arithmetic Mean	**1.33**
					Geometric Mean	**1.26**

* a: >2 columns; b: 2 columns; c: 1 column.

TABLE IV

PERFORMANCE COMPARISON WITH *GLU v2.0* (DOUBLE PRECISION)

Matrix	Matrix size: # of rows (n)	No. of nonzero (nnz)	Dependency Level Info. (a, b, c)*	Time (ms) Double precision		
				GLU v2.0	Pro-posed	Speed -up
rajat12	1879	12818	(23,1,13)	23	22	**1.05**
circuit_2	4510	21199	(63 6 32)	58	54	**1.07**
memplus	17758	99147	(16,5,126)	116	105	**1.10**
rajat27	20640	97353	(61,10,52)	172	151	**1.14**
onetone2	36057	222596	(140,61,1012)	4209	3883	**1.08**
rajat15	37261	443573	(257,155,554)	3310	3192	**1.04**
rajat26	51032	247528	(68,1,88)	638	428	**1.49**
circuit_4	80209	307604	(30,9,213)	3299	3132	**1.05**
rajat20	86916	604299	(183,22,1011)	28147	21294	**1.32**
ASIC _100ks	99190	578890	(277,4,1350)	24971	23688	**1.05**
hcircuit	105676	513072	(57,9,78)	854	504	**1.69**
Raj1	263743	1302464	(493,307,794)	87152	73155	**1.19**
ASIC _320ks	321671	1827807	(236,15, 1418)	34179	26975	**1.27**
ASIC _680ks	682712	2329176	(247,40, 1163)	54141	16819	**3.22**
G3_ circuit	1585478	7660826	(641,5,6)	141017	23737	**5.94**
					Arithmetic Mean	**1.34**
					Geometric Mean	**1.27**

* a: >2 columns; b: 2 columns; c: 1 column.

matrix is large, the number of kernel launches is also large, so the benefit gains from *dynamic parallelism* are higher in this case, whereas the benefits from the batch and pipeline mode enhancements are minimal. On the other hand, circuits *onetone2, rajat15* and *ASIC_100ks* are relatively smaller (with matrix size significantly smaller than that of the *G3_circuit*), but having a large number of levels with one and two columns. In such cases, the performance gain is mainly due to the batch- and pipeline-mode enhancements.

B. Performance Comparison of the Proposed Method and GLU (Double Precision)

Noting that the circuit simulation requires higher precision for numerical stability as well as for accuracy of results, in this experiment, a performance validation of the proposed algorithms is made in its double-precision-based implementation. Since *GLU v2.0* is single-precision-based, we modified it to have double-precision-based implementation. The corresponding performance comparison is given in Table IV. As can be seen, even for the double-precision case, the proposed algorithms perform consistently better and yield significant speedup.

The above results show that the proposed batch and *pipeline mode* execution along with the proposed GPU managed *kernel* launch method discussed in Section III can speed up the GPU LU factorization. Since the proposed techniques focus on reducing the number of individual *kernel* launches (through batch and pipeline modes) while also minimizing the *kernel* launch overhead (through GPU managed *kernel* launch), the performance gain becomes even more prominent when the circuit is large and contains many levels.

C. Performance Comparison of the Proposed Method and KLU (Double Precision)

In this experiment, a performance comparison of the proposed GPU-based LU factorization with a CPU-based KLU (a widely used serial LU factorization software) is made. The corresponding results are given in Table IV. For accuracy comparison, LU factors obtained using the proposed method are checked against the LU factors from KLU by verifying each element in the solution vector $\boldsymbol{x}$ in the system of equations $\boldsymbol{Ax} = \boldsymbol{b}$, both of which matched reasonably accurately [29]. As can be seen from Table IV, the proposed GPU-based LU factorization provides a superior performance compared with KLU in most cases with an average speedup of 4.75 (arithmetic mean) and 2.22 (geometric mean). The speedup tends to be higher for circuits that are large and with a higher number of nonzeros.

It is to be noted that, it may be difficult to draw a definitive conclusion from Table V regarding the superior performance,

TABLE V

PERFORMANCE COMPARISON WITH KLU (DOUBLE PRECISION)

Matrix	Matrix size: # of rows (n)	No. of nonzero (nnz)	Time (ms) Double Precision		
			KLU	Proposed	Speed-Up
rajat12	1879	12818	11	22	**0.5**
circuit_2	4510	21199	47	54	**0.87**
memplus	17758	99147	2921	105	**27.82**
rajat27	20640	97353	252	151	**1.67**
onetone2	36057	222596	18657	3883	**4.8**
rajat15	37261	443573	3371	3192	**1.06**
rajat26	51032	247528	240	428	**0.56**
circuit_4	80209	307604	6669	3132	**2.13**
rajat20	86916	604299	198027	21294	**9.3**
ASIC_100ks	99190	578890	91629	23688	**3.87**
hcircuit	105676	513072	640	504	**1.27**
Raj1	263743	1302464	108340	73155	**1.48**
ASIC_320ks	321671	1827807	97962	26975	**3.63**
ASIC_680ks	682712	2329176	198108	16819	**11.78**
G3_ circuit	1585478	7660826	318212	23737	**13.41**
				Arithmetic Mean	**5.05**
				Geometric Mean	**2.49**

since CPU and GPU are two totally different computing platforms with wide varying capabilities, parameters, and purposes. As evident from Table V, a performance better than KLU may not always be guaranteed using a GPU platform, which can be due to several factors such as the nature of the circuit, size of the circuit matrix, number of nonzeors, column dependence, number of levels, and the capabilities of the CPU and GPU hardware. On the other hand, the above results firmly establish the fact that GPUs can be a good alternative computing platform to the CPUs for LU factorization. Also, the above results provide a good insight for making a judicious choice of CPU or GPU for LU factorization in a system with both CPUs and GPUs.

V. Conclusion

In this paper, a dynamic parallel GPU-based algorithm for sparse LU factorization with an emphasis on circuit simulation is presented. The proposed algorithm exploits the *dynamic parallelism* feature in newer GPUs to reduce the CPU overhead in launching and managing the GPU *kernels*. Also, *batch mode*- and *pipeline mode*-based algorithms are proposed to efficiently handle the levels with fewer columns (such as with one or two columns). A performance comparison of the proposed algorithms with the existing GPU LU factorization methods as well as the CPU-based KLU is presented. The presented results provide a good mechanism and way forward for a judicious choice of GPU-based LU factorization or CPU- and KLU-based LU factorization.

Acknowledgment

The authors would like to thank Universiti Tunku Abdul Rahman and Carleton University for supporting this collaboration work between the two universities. They would also like to thank the group of Prof. S. X.-D. Tan at the University of California at Riverside, particularly L. Wang for his inputs on the GLU software. Part of the research work presented in this paper was performed by W.-K. Lee during his stay at Carleton as visiting scholar.

References

[1] (Jan. 2017). *CUDA Programming Guide V8.0*. Accessed: Sep. 2017. [Online]. Available: https://developer.nvidia.com/cuda-80-ga2-download-archive

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

[3] T. George, V. Saxena, A. Gupta, A. Singh, and A. R. Choudhury, "Multifrontal factorization of sparse SPD matrices on GPUs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, Anchorage, AK, USA, May 2011, pp. 372–383.

[4] R. F. Lucas, G. Wagenbreth, D. M. Davis, and R. Grimes, "Multifrontal computations on GPUs and their multi-core hosts," in *High Performance Computing for Computational Science—VECPAR* (Lecture Notes in Computer Science), vol. 6449. New York, NY, USA: Springer, 2011, pp. 71–82.

[5] J. W. H. Liu, "The multifrontal method for sparse matrix solution: Theory and practice," *SIAM Rev.*, vol. 34, no. 1, pp. 82–109, 1992.

[6] *Dense Linear Algebra on GPUs*. Accessed: Mar. 2018. [Online]. Available: https://developer.nvidia.com/cublas

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

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

[9] N. Fröhlich, B. M. Riess, U. A. Wever, and Q. Zheng, "A new approach for parallel simulation of VLSI circuits on a transistor level," *IEEE Trans. Circuits Syst. I, Fundam. Theory Appl.*, vol. 45, no. 6, pp. 601–613, Jun. 1998.

[10] D. Paul, M. S. Nakhla, R. Achar, and N. M. Nakhla, "Parallel circuit simulation via binary link formulations (PvB)," *IEEE Trans. Compon. Packag. Manuf. Technol.*, vol. 3, no. 5, pp. 768–782, May 2013.

[11] 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.

[12] X. Chen, L. Ren, Y. Wang, and H. Yang, "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.

[13] K. He, S. X.-D. Tan, H. Wang, and G. Shi, "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.

[14] *Cusolver Library*, document DU-06709-001_v9.0, Jun. 2017.

[15] GLU. *GPU-Accelerated Sparse Parallel LU Factorization Solver Version 2.0*. Accessed: Jul. 2017. [Online]. Available: http://www.ee.ucr.edu/~stan/project/glu/glu_proj.htm

[16] K. H. Jin, M. T. McCann, E. Froustey, and M. Unser, "Deep convolutional neural network for inverse problems in imaging," *IEEE Trans. Image Process.*, vol. 26, no. 9, pp. 4509–4522, Sep. 2017.

[17] S. Hussain, R. C. P. Silva, and D. A. Lowther, "Implementation of iron loss model on graphic processing units," *IEEE Trans. Magn.*, vol. 52, no. 3, pp. 1–4, Mar. 2017.

[18] Y. Liang, X. Xing, and Y. Li, "A GPU-based large-scale Monte Carlo simulation method for systems with long-range interactions," *J. Comput. Phys.*, vol. 338, pp. 252–268, Jun. 2017.

[19] W.-K. Lee, R. C.-W. Phan, G.-S. Poh, and B.-M. Goi, "SearchaStore: Fast and secure searchable cloud services," *Cluster Comput.*, pp. 1–14, 2017.

[20] X.-X. Liu, H. Yu, and S. X.-D. Tan, "A GPU-accelerated parallel shooting algorithm for analysis of radio frequency and microwave integrated circuits," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 3, pp. 480–492, Mar. 2015.

[21] B. D. de Vos, J. M. Wolterink, P. A. de Jong, T. Leiner, M. A. Viergever, and I. Isgum, "ConvNet-based localization of anatomical structures in 3-D medical images," *IEEE Trans. Med. Imag.*, vol. 36, no. 7, pp. 1470–1481, Jul. 2017.

[22] I. S. Duff and J. Koster, "On algorithms for permuting large entries to the diagonal of a sparse matrix," *SIAM J. Matrix Anal. Appl.*, vol. 22, no. 4, pp. 973–996, 2000.

[23] K. Singhal and J. Vlach, *Computer Methods for Circuit Analysis and Design*, 2nd ed. Boston, MA, USA: Kluwer, 1993.

[24] R. I. Bahar *et al.*, "Algebric decision diagrams and their applications," *Formal Methods Syst. Des.*, vol. 10, nos. 2–3, pp. 171–206, 1997.

[25] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Comput.*, vol. 36, nos. 5–6, pp. 232–240, Jun. 2010.

[26] J. Kurzak, P. Luszczek, M. Faverge, and J. Dongarra, "LU factorization with partial pivoting for a multicore system with accelerators," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 8, pp. 1613–1621, Aug. 2013.

[27] T. Davis. *The University of Florida Sparse Matrix Collection*. Accessed: Jun. 2017. [Online]. Available: http://www.cise.ufl.edu/research/sparse/

[28] *Compute Canada*. Accessed: Jul. 2017. [Online]. Available: https://www.computecanada.ca/home/

[29] *Floating Point and IEEE 754 Compliance for NVIDIA GPUs*. Accessed: May 15, 2018. [Online]. Available: https://docs.nvidia.com/cuda/floating-point/index.html

Wai-Kong Lee (M'13) received the B.Eng. degree in electronics and the M.Sc. degree from Multimedia University, Cyberjaya, Malaysia, in 2006 and 2009, respectively, and the Ph.D. degree in engineering from University Tunku Abdul Rahman, Petaling Jaya, Malaysia, in 2018.

He was a Visiting Scholar with OTH Regensburg, Regensburg, Germany, in 2015, Carleton University, Ottawa, ON, Canada, in 2017, and Feng Chia University, Taichung, Taiwan, in 2016 and 2018. His current research interests include cryptography, numerical algorithms, GPU computing, and energy harvesting.

Dr. Lee had served as a reviewer for some international journals, such as the IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING from 2016 to 2017 and *Computer and Electrical Engineering* in 2017.

Ramachandra Achar (S'95–M'00–SM'04–F'13) received the B.Eng. degree in electronics engineering from Bangalore University, Bengaluru, India, in 1990, the M.Eng. degree in microelectronics from the Birla Institute of Technology and Science at Pilani, Pilani, India, in 1992, and the Ph.D. degree in electrical engineering from Carleton University, Ottawa, ON, Canada, in 1998.

He is currently a Professor with the Department of Electronics Engineering, Carleton University. Prior to joining the Carleton University Faculty in 2000, he served in various capacities in leading research labs, including the Indian Institute of Science, Bengaluru, in 1990, the Central Electronics Engineering Research Institute, Pilani, in 1992, Larsen and Toubro Engineers Ltd., Mysore, in 1992, and the T. J. Watson Research Center, IBM, Yorktown Heights, NY, USA, in 1995. He has authored over 200 peer-reviewed articles in international transactions/conferences, six multimedia books on signal integrity, and five chapters in different books. His current research interests include signal/power integrity analysis, circuit simulation, parallel and numerical algorithms, EMC/EMI analysis, microwave/RF algorithm, and mixed-domain analysis.

Dr. Achar is a fellow of the Engineers Institute of Canada. He is a Founding Faculty Member of the Canada–India Center of Excellence. He received several prestigious awards, including the Carleton University Research Achievement Awards in 2010 and 2004, the Natural Science and Engineering Research Council Doctoral Medal in 2000, the University Medal for the Outstanding Doctoral Work in 1998, the Strategic Microelectronics Corporation Award in 1997, and the Canadian Microelectronics Corporation Award in 1996. He was a co-recipient of the IEEE Best Transactions Paper Award for the IEEE TRANSACTIONS ON ADVANCED PACKAGING in 2007 and the IEEE TRANSACTIONS ON COMPONENTS, PACKAGING AND MANUFACTURING TECHNOLOGY in 2013. His students have won numerous best student paper awards in international forums. He was the General Chair of HPCPS from 2012 to 2017, the General Co-Chair of NEMO in 2015, SIPI in 2016, and EPEPS in 2010 and 2011, and an International Guest Faculty on the invitation of the Department of Information Technology, Government of India, under the SMDP-II Program, in 2012. He currently serves as the Chair of the Distinguished Lecturer (DL) Program of the EMC Society and a DL for the Electron Devices Society. He is the Chair of the joint chapters of CAS/EDS/SSC Societies of the IEEE Ottawa Section. He also previously served as the DL for the CAS Society from 2011 to 2012 and the EMC Society from 2015 to 2016. He currently serves on the executive/steering/technical-program committees of several leading IEEE international conferences, including EPEPS, EDAPS, SPI, HPCPS, and SIPI and in the technical committees, including EDMS (TC-12 of CPMT), computer aided design (MTT-1), and SIPI (TC-10 of EMCS). He previously served as a Guest Editor for the IEEE TRANSACTIONS ON COMPONENTS, PACKAGING AND MANUFACTURING TECHNOLOGY, for two special issues on Variability Analysis and 3D-ICs/Interconnects in 2015. He is a consultant for several leading industries focused on high-frequency circuits, systems, and tools. He is a Practicing Professional Engineer of Ontario.

Michel S. Nakhla (S'73–M'75–SM'88–F'98–LF'12) received the Ph.D. degree in electrical engineering from the University of Waterloo, Waterloo, ON, Canada.

From 1976 to 1988, he was a Senior Manager with the Computer-Aided Engineering Group, Bell-Northern Research, Ottawa, ON, Canada. In 1988, he joined Carleton University, Ottawa, as a Professor, where he held the Computer-Aided Engineering Senior Industrial Chair established by Bell-Northern Research and the Natural Sciences and Engineering Research Council of Canada. He is the Founder of the High-Speed computer aided design (CAD) Research Group, Carleton University, where he is currently a Chancellor's Professor of Electrical Engineering. He has authored or coauthored over 360 peer-reviewed research articles, two books, and seven chapters in different books. His current research interests include modeling and simulation of high-speed circuits and interconnects, uncertainty quantification, model-order reduction, nonlinear circuits, radio frequency and microwave circuits, parallel processing, and multidisciplinary optimization.

Dr. Nakhla is a fellow of the Canadian Academy of Engineering. He was recognized as a fellow of the Canadian Academy of Engineering in 2016 for trend-setting achievements and contributions to the state of the art in computer-aided design of high-speed VLSI and microwave circuits and systems and for providing an exemplary leadership and service as an Educator, a Research and Development Engineering Manager, and a Researcher. He received the 2017 IEEE Canada A.G.L. McNaughton Gold Medal for outstanding contributions to signal integrity of high-speed systems and interconnects, the Best Transactions Paper Award for the IEEE TRANSACTIONS ON COMPONENTS, PACKAGING AND MANUFACTURING TECHNOLOGY in 2014 and the IEEE TRANSACTIONS ON ADVANCED PACKAGING in 2007, and the IEEE 2002 Microwave Prize. He served as Associate Editor for the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS. He is currently an Associate Editor of the IEEE TRANSACTIONS ON COMPONENTS, PACKAGING AND MANUFACTURING TECHNOLOGY. He is on various international committees, including the Standing Committee of the IEEE Signal and Power Workshop, the Technical Program Committee of the IEEE International Microwave Symposium, the Technical Program Committee of the Advanced Packaging and Systems Symposium, the Technical Program Committee of the IEEE International Conference on Numerical Electromagnetic and Multiphysics Modeling and Optimization, and the CAD Committee of the IEEE Microwave Theory and Techniques Society. He also served on several Canadian and international government-sponsored research grant selection panels. He is a Practicing Professional Engineer of Ontario.