Received 7 June 2022, accepted 14 July 2022, date of publication 25 July 2022, date of current version 1 August 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3193785

APPLIED RESEARCH

Algorithmic Advancements and a Comparative Investigation of Left and Right Looking Sparse LU Factorization on GPU Platform for Circuit Simulation

WAI-KONG LEE[1], (Member, IEEE), AND RAMACHANDRA ACHAR[2], (Fellow, IEEE)
[1]Department of Computer Engineering, Gachon University, Seongnam 13120, South Korea
[2]Department of Electronics, Carleton University, Ottawa, ON K1S 5B6, Canada

Corresponding author: Wai-Kong Lee (waikong.lee@gmail.com)

The work of Ramachandra Achar was supported in part by the Natural Science and Engineering Research Council of Canada (NSERC).

ABSTRACT Sparse LU factorization is a key tool in the solution of large linear set of algebraic equations encompassing a wide range of computing applications. Recent advances in this field exploit the massively parallel architecture of the GPUs via left-looking algorithm (LLA) and right-looking algorithm (RLA). In this paper, *adaptive cluster* mode is proposed to improve the state-of-the-art in LLA for GPU platforms. The proposed method takes into consideration of varying sparsity at different levels during cluster mode execution, to adaptively configure the GPU block size and the number of parallel columns. The new refinements for LLA are also integrated with the dynamic parallelism that is available in modern GPU architectures. The paper also provides a comprehensive performance comparison of the LLA and hybrid RLA along with state-of-the-art advances on the same GPU platform. The results indicate that, when implemented with similar refinements and on a same platform, LLA provides better performance compared to the hybrid-RLA. The results would be useful to the scientific community while making decision on adopting LLA or RLA algorithms for sparse LU factorization.

INDEX TERMS Circuit simulation, GPU, LU factorization, multi-core, parallel simulation, sparse matrices, SPICE.

I. INTRODUCTION

Simulation and optimization involving many engineering and scientific applications require solution of large linear algebraic system of equations. Examples of such applications include circuit simulation and multi-dimensional scaling, etc. Lower-upper ($\boldsymbol{LU}$) factorization is one of the important direct solvers widely used by the scientific community in solving linear equations. During LU factorization, a matrix $\boldsymbol{A}$ is decomposed into its lower ($\boldsymbol{L}$) and upper ($\boldsymbol{U}$) matrices. The $\boldsymbol{L}$ and $\boldsymbol{U}$ matrices are used with forward and backward substitutions to obtain the final solution [1]. Although LU factorization is faster than conventional Gauss–Jordan elimination [1], it becomes slow for larger matrices. Significant advances are made during the last several decades to exploit the sparsity of the system matrix $\boldsymbol{A}$ to speed-up the LU factorization process. For instance, a time-domain analysis using SPICE (Simulation Program with Integrated Circuit Emphasis) based circuit simulators often requires solutions at thousands of time points. Each time point solution in turn requires several Newton Raphson (NR) iterations, wherein each NR iteration needs a LU factorization involving a large but sparse matrix. Despite exploiting the sparsity, considering the huge number of LU factorizations involved in a SPICE simulation, it is necessary to further minimize the time spent in each LU factorization [1]–[5].

To address this challenge, several prior works attempted to parallelize the LU factorization over multi-core CPUs [2], [8].

The associate editor coordinating the review of this manuscript and approving it for publication was S. Khandelwal.

Also, exploiting the massive computational resources available in Graphics Processing Unit (GPU), parallel algorithms to speed up the LU factorization [9]–[11] were proposed. To further exploit the sparse nature of the circuit matrices, GPU based algorithms were proposed by Chen *et al.* [3] based on the G/P Left Looking algorithm (LLA) [12]. In [3], it was proposed to use a fixed block size (32 threads) while executing LLA in cluster as well as pipeline modes. The number of parallel columns was selected based on the size of a particular level. However, this approach is not optimal because the number of nonzeros in each column is different, and hence the amount of computation in each column also varies accordingly.

Later on, He *et al.* [4] proposed the Hybrid Right Looking algorithm (RLA), with the objective of exploiting more parallelism in GPU compared to LLA. Following up the work, Lee *et al.* [5] introduced a dynamic approach to speedup the RLA. Peng and Tan [6] improved the speed performance of RLA through a stream mode to handle the levels with very small number of parallel columns. They also proposed a technique to adaptively configure the block size as the RLA execution progresses level by level. Recently, Lee and Achar [7] presented a novel adaptive PCBSO approach that configures the number of parallel blocks and block size simultaneously, based on the size of associated submatrices at each level. This effectively addressed the resource contention issue associated with the work by Peng and Tan [6], leading to better performance.

One of the limitation of RLA is that it requires the use of atomic instructions, which in many cases can greatly limit the performance [4], [6]. In contrast, LLA algorithm does not exhibit such a limitation. Moreover, RLA requires longer levelling time compared to LLA, due to the potential ''read-after-write'' data hazard. Although significant advances are done to improve the performance of both the LLA and RLA, there is no prior work that directly compares LLA and RLA implementations with all the state-of-the-art refinements. For instance, He *et al.* [4] showed that the hybrid RLA is superior compared to LLA, but their work did not address the issue of ''read-after-write hazard'' in RLA [5] and also didn't include the benefits of pipeline mode execution in LLA. A fair comparison between LLA and RLA implementations can only be achieved if all the advanced refinements are considered while accounting for any ''read-after-write'' data hazard and properly benchmarked on the same GPU architecture.

Addressing the above discussed issues, following contributions are made in this paper to advance the state-of-the-art in the sparse LU factorization on GPU platforms:

1) First contribution focuses on addressing the performance limitation of LLA due to the fixed block size in the cluster mode. For this purpose, a novel adaptive cluster mode is proposed to improve the performance of the LLA. The proposed approach adaptively configures the block size and number of parallel columns for parallel processing based on the total number of nonzeros at a particular level.
2) Proposed advancements for the LLA are also applied with dynamic parallelism (similar to the case in RLA [7]).
3) A comprehensive performance comparison using a set of publicly available benchmark circuit examples, between LLA and Hybrid RLA [7] approaches is presented. For a fair comparison, both methods are executed on the same platform and were implemented with state-of-the-art refinements along with the ones that are presented in this paper. The results indicate that, when implemented with state-of-the-art techniques and on a same platform, LLA provides better performance compared to the Hybrid-RLA. The results are very useful to the scientific community while making the decision on which approach to use for LU factorization in GPU platforms.

Rest of the paper is organized as follows. In Section II, a brief background of LLA and RLA along with their state-of-the-art advances are provided. Section III provides the details of the proposed adaptive cluster mode for LLA along with dynamic parallelism. Section IV provides the discussions and comparative analytical insights into the performance and implementation aspects for both the LLA and RLA. Section V and VI provide the numerical results and conclusions, respectively.

II. BACKGROUND

In this section, a brief review of state-of-the-art in LU factorization on GPU platforms is given. For an overview of GPU architectures, CUDA programming model, memory hierarchy and relevant terminologies, please refer to [5].

A. G/P LEFT LOOKING ALGORITHM

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 $\boldsymbol{U}$
4:      for $j = 1$ to $i - 1$ where $A(j, i) \neq 0$ do
5:          for $k = j + 1$ to $n$ where $A(k, j) \neq 0$ do
6:              $A(k, i) = A(k, i) - A(k, j) \times A(j, i)$
7:          end for
8:      end for
9:      // Compute column $i$ for $\boldsymbol{L}$
10:     for $k = i + 1$ to $n$ where $A(k, i) \neq 0$ do
11:         $A(k, i) = A(k, i)/A(i, i)$
12:     end for
13: end for
```

G/P [12] left-looking algorithm is widely accepted as one of the most efficient method to perform LU factorization for sparce matrices. It is also widely adopted by state-of-the-art sparse LU solvers (e.g. KLU [1] and NICSLU [2], [3]). Algorithm 1 shows the pseudocode for in-place version of G/P [12] left-looking algorithm, wherein the LU factors are directly stored into the matrix $\boldsymbol{A}$, which replaces its original

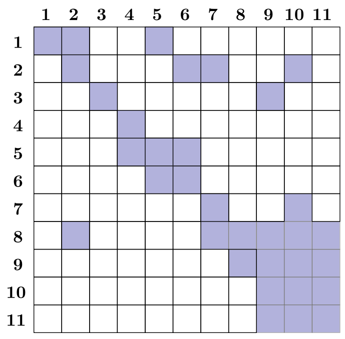

FIGURE 1. An illustrative sparse matrix.

TABLE 1. Column dependency levels.

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

values. Note that this is a slight modifications to the original G/P [12] left-looking algorithm to achieve the in-place implementation without affecting its correctness.

Referring to Algorithm 1, the left-looking method processes the matrix $\boldsymbol{A}$ column by column, from left to right (index i) and computes 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 column j in order to calculate the $\boldsymbol{L}$ and $\boldsymbol{U}$ factors for the current column (index i).

LLA by its nature, is a serial algorithm, but it can be adapted to exploit parallellism and sparsity while processing sparse matrices [1]. By knowing the nonzero pattern of the $\boldsymbol{U}$ matrix, the column dependency for the LLA can be determined *a priori*. Any independent columns can be factorized in parallel since they will not affect the results in other columns. For example, referring to Fig. 1 as an example matrix, the corresponding column dependency levels can be determined and are shown in Table 1. As can be seen in this example, there is no dependency between columns 1, 3, 4 and 8, and hence they can be executed first in parallel (level 1). Column 2 and column 5 depend on column 1, so the factorization of these columns have to be postponed to level 2. Column 6 depends on columns 2 and 5, while column 7 depends on column 2; hence they need to be placed in level 3 (i.e., after level 2 which contains columns 2 and 5). By using this method, column dependency levels for the entire matrix $\boldsymbol{A}$ can be determined before beginning the actual factorization process. This process is referred to as *dependency levelling* in the subsequent discussions.

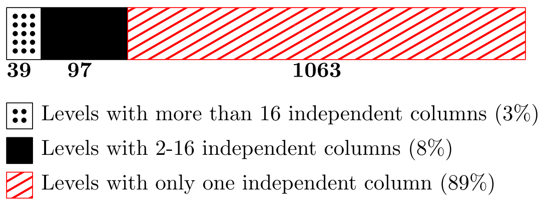

FIGURE 2. An illustrative example of number of columns in different levels for LLA execution in a circuit matrix (onetone2 [15]: matrix of size 36,057 × 36,057 with 1,199 levels).

After the dependency levelling, each level is launched in serial, while all the columns within the same level are executed in parallel. This corresponds to the outermost ***for*** loop in Algorithm 1 (index i), and is termed as *column-level parallelism* or *cluster* mode execution [2].

However, for large sparse circuit matrices, typically many levels will end up with very few independent columns (for example, see levels 3-6 in Table 1). As a further illustration, Fig. 2 shows the level information for a relatively large circuit matrix, wherein majority of the levels have less than 16 independent columns. This essentially leads to nearly serial execution of majority of the columns, since there are very few independent columns available. To further improve the performance, *"pipeline mode"* was introduced for LLA by [2], [3] to speed up the execution of levels with very few independent columns, which is summarized in Algorithm 2.

Algorithm 2 Algorithm for LLA Pipeline Mode Implementation

```
1: // Launch the kernel for levels with very less columns in
   batch
2: // Triangular solve for $\boldsymbol{U}$
3: for all subcols to the left of current col do
4:     while (pipeFlag[subcols]==false);
5:     Perform triangular solve using the elements
   (line 4 - 8 in Algorithm 1) from subcol in parallel
6: end for
7: Synchronize all threads
8: for Current col in parallel do
9:     Compute $\boldsymbol{L}$ matrix for current textitcol
10: end for
11: // Inform the next column in the queue to start
12: pipeFlag[col]=true;
```

Referring to Algorithm 2, the GPU *kernels* for levels with very few independent columns are launched together. During the triangular solve process, each column checks *if the columns on which it is dependent on* have completed their execution (line 4), by using a tracking array, *pipeFlag*. A column can immediately start its execution (line 5) once the columns on which it is dependent on complete their part of the computation. Towards the end of the execution, each column updates the *pipeFlag* array (line 12), indicating that the update for this particular column is completed. Other columns that depend on this particular column can start their exeuction immediately after detecting this update in the *pipeFlag* array. As a result, this mechanism creates a pipeline execution to speed up the levels with only few independent columns. Note that the *pipeFlag* array has to be resides on the global memory so that it is visible to all parallel blocks.

Also, the multiply-and-accumulate (MAC) operation (lines 5-7) are executed in parallel to enable fine-grain parallelism, since each MAC operation is independent from the other. However, the j loop (line 4) in LLA cannot be parallelized as each j iteration is dependent on the values computed in the previous iteration. For levels with many independent columns, all columns are executed in parallel, which is referred to as "cluster mode" [2], [3]. The threshold (N_{th}) to select between the *cluster* and *pipeline* mode is determined experimentally [2], [3]. The implementation results presented by Chen *et al.* [3] represents the state-of-the-art for LLA in GPU platform.

B. HYBRID RIGHT LOOKING ALGORITHM

In the case of RLA algorithm, columns in the matrix are scanned from left to right. He *et al.* [4] proposed hybrid column based right-looking algorithm (RLA, refer to Algorithm 3), with the intention of exploiting more parallelism in GPU platforms. As can be noted, the algorithm starts by computing the $\boldsymbol{L}$ factors for the current column, and subsequently updates the submatrix on the right of the current column via the MAC operations (hence is the name "right looking method"). Similar to LLA, RLA is also able to exploit the column based parallelism and parallelization of the two ***for*** loops in submatrix update. In other words, RLA not only inherits the advantage of LLA (to execute in cluster mode), but also allows for more fine-grain parallelism.

However, RLA requires different dependency levelling compared to LLA, 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 can cause a "read-after-write" data hazard which can lead to inaccurate results and should be avoided if there is dependency between the two columns under consideration. Related details can be found in [5] and for the benefit of the reader, it is summarized in Appendix A.

Algorithm 3 Hybrid Column Based RLA [4]

```
1:  // Scan each column from left to right
2:  for $i = 1$ to $n$ do
3:      // Compute column $i$ for $\boldsymbol{L}$
4:      for $j = i + 1$ to $n$ where $A(j, i) \neq 0$ 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 $A(i, k) \neq 0$ do
9:          for $j = i + 1$ to $n$ where $A(j, i) \neq 0$ do
10:             $A(j, k) = A(j, k) - A(j, i) \times A(i, k)$
11:         end for
12:     end for
13: end for
```

After the proper dependency levelling process in RLA, many levels end up containing very few independent columns, causing it to suffer from the similar disadvantages of the left-looking algorithm [2], [3]. To alleviate this problem, Lee *et al.* [5] proposed pipeline mode execution for levels with only one column. They also proposed *batch* mode execution to improve the execution speed for levels with only two columns. For levels with low number of independent columns but have more than two independent columns, the batch mode did not provide additional benefit, mainly due to the intensive memory accesses involved.

Peng and Tan [6] proposed a *stream* mode to process levels with low number of independent columns. In stream mode, one column is processed by the entire GPU grid instead of one block, and multiple columns are executed in parallel through the multiple streams. This allows for more parallelism as the execution in different streams in GPU can overlap with each other. They also proposed *small block* and *large block* modes, where the block size increases from small to large as the LU factorization progresses to higher levels. The *large block* mode in fact represents the maximum case of small block mode where the block size is 32 warps or 1024 threads. Although these techniques tend to improve the speed performance compared to [5], they suffered from resource contention in GPU, as the maximum number of parallel columns were set to a fixed number. Since GPUs have limited computing and memory resources, the increase of block size may introduce more resource contention if the number of parallel blocks are not adjusted accordingly.

To address the above issue, Lee and Achar [7] proposed a novel adaptive PCBSO mode to adaptively configure the block size and the number of parallel blocks, based on the size of associated submatrices in each level. This can effectively avoid the resource contention problem in GPU, hence improving the speed performance of RLA. At the same time, they also proposed the Dynamic PCBSO mode to reduce the kernel launch time overhead in PCBSO mode, through the use of dynamic parallelism feature in GPU.

III. PROPOSED ADVANCEMENT FOR LEFT-LOOKING ALGORITHM IN GPU PLATFORM

In this section, first, adaptive cluster mode to improve the performance of LLA in GPUs is first presented. Subsequently, parallel implementation of the proposed advancement is discussed.

A. PROPOSED ADAPTIVE CLUSTER MODE FOR LEFT LOOKING ALGORITHM

The concept of *cluster* mode and *pipeline* mode were first introduced by Chen *et al.* [2] to process LLA efficiently in multi-core CPUs. Subsequently, they also proposed implementation of these techniques to fully exploit the massive parallelism offered by the GPUs [3]. In particular, the authors [3] commented that in *cluster* mode, columns are usually very sparse and hence the block size should be as small as possible to avoid threads being idle. Setting small block size in *cluster*

TABLE 2. Average number of nonzeros at different levels.

Level	rajat12	onetone2	ASIC_100ks	G3_circuit
1	3	2	4	2
2	5	5	5	5
3	11	8	6	9
4	11	9	9	14
5	17	10	10	20
6	13	13	12	28
7	15	13	15	38
8	20	13	18	52
9	19	18	21	68
10	18	18	24	84
11 to 100	17 - 839	13 - 229	29 - 400	97 - 2542
101 to 300	N/A	187 -555	406 - 1005	N/A
301 to 600	N/A	359 - 743	822 - 1823	N/A
> 601	N/A	122 - 1179	1081 - 2912	N/A

mode can also allow more number of columns to be processed concurrently. On the other hand, *pipeline* mode processes columns with many nonzeros, due to which threads always have sufficient workload; block size does not matter much in such a scenario. Given these reasons, they suggested to set the block size as one warp (32 threads), balancing the needs in both the modes. At the same time, the maximum number of parallel blocks to be processed by *cluster* and *pipeline* modes was set to be constant throughout the LU factorization in GPU.

However, careful observation reveals that this may not be the optimal configuration to fully exploit the parallelism during LLA implementation in GPUs. Referring to Table 2, one can observe that the average number of nonzeros across different levels are not constant, and majority of the cases tend to have more nonzeros at higher levels. For the purpose of illustration, taking ASIC_100ks as a sample case (see Table 2, column 4), the average number of nonzeros increases from 4 to 24 when it progresses from level 1 to level 10; the circuit becomes even more denser for levels beyond 11. Note that the *cluster* mode is assigned to handle levels with many independent columns, which happens to be the case at the first 1-20 levels. The remaining (higher) levels with less independent columns are computed through the *pipeline* mode. The above observations provide two important insights which can be summarized as below:

1) In *cluster* mode, the sparsity of columns varies significantly for different matrices. In other words, the number of nonzeros for each level is not constant and hence setting a fixed block size (32 threads) while executing in this mode may not be optimal. If the block size is set arbitrarily much larger than the number of nonzeros, some of the threads may become idle as there may not be enough workload to perform. On the contrary, small block size may not be able to fully exploit the parallelism potential of the GPU.
2) The columns in *pipeline* mode have relatively more nonzeros compared to the columns in the *cluster* mode. Hence, it is more advantageous to allocate a larger block size to handle each column in parallel, in order to fully exploit the parallelism in GPU.

To address the above difficulties following advancements are proposed. First, we propose to configure the block size in *cluster* mode adaptively, based on the average number of nonzeros in each level. This can ensure that there is always sufficient workload to be computed by each block. For calculating the adaptive block size (W_{AD}), following relation is developed:

$$W_{AD} = st \times \lfloor \frac{avgNonzeros_m}{32} \rfloor \tag{1}$$

where $avgNonzeros_m$ refers to the average number of nonzeros in level m. st is a scaling factor which is set based on the GPU architecture and available resources (i.e., GPU cores). It is determined experimentally by varying $st = 1, 2, 4, 8, \ldots, 32$; whichever among them provides the best performance is selected. This is done only once and it can be used for simulating various circuits.

At the same time, the number of parallel columns (N_{AD}) to be processed is also configured adaptively based on the above computed W_{AD}. It is obstained as inversely proportional to W_{AD} in order to avoid potential resource contention in GPU, using the following relation:

$$N_{AD} = \frac{max\_parallel\_cols \times sb}{32 \times W_{AD}} \tag{2}$$

where sb refers to the scaling factor and has the similar purpose and relavence as st. $\boldsymbol{max\_parallel\_cols}$ refers to the maximum allowable columns to be processed in parallel, which is calculated as:

$$max\_parallel\_cols = \frac{Maximum\ global\ memory\ available}{Number\ of\ \ rows\ in\ the\ matrix\ A} \tag{3}$$

For the *pipeline* mode, the block size (W_{PL}) is selected to be between 4 - 32 in order to allow sufficient parallelism. The value for W_{PL} is determined through experiments as it is mainly affected by the GPU resources and its architectures. The value of W_{PL} remains constant throughout the entire *pipeline* mode execution.

Proposed adaptive cluster mode executing steps are presented as a pseudocode in Algorithm 4. Initially, some pre-processing steps are completed prior to LLA based LU factorization. First, after scanning through each level (line 3), sum of nonzeros in every column at each level (lines 4-6) is obtained. Next, the average number of nonzeros at each level is calculated and stored (lines 11-12) in an array (*avgNonzeros*). Subsequently, the threshold level (***thres_lev***) that distinguishes between *cluster* and *pipeline* modes is determined. Referring to lines 7-9, the number of independent columns at each level is compared with a threshold value N_{th}. The ***thres_lev*** is set as the level m that contains number of columns lesser than N_{th}. In other words, all levels before ***thres_lev*** is being processed in the *cluster* mode, while the rest of the columns are proceed in *pipeline* mode.

Algorithm 4 Proposed Adaptive Cluster Mode for LLA LU Factorization

Definition:
num_level: Total number of levels in matrix ***A***
num_col: An array of size *num_level*, storing the number of columns in each level
curr_col_sz: A two dimensional array of size *num_level* × *num_col*, storing the size of nonzeros of each column
max_parallel_cols: The maximum parallel columns that can be factorized in parallel
avgNonzeros: An array of size *num_level*, storing the average size of nonzeros at each level
st: Scaling factor to determine the number of parallel threads
sb: Scaling factor to determine the number of parallel blocks
N_{th}: The threshold (number of independent columns) to select between *cluster* and *pipeline* mode
thres_lev: The threshold level that divides between *cluster* and *pipeline* mode. All levels before ***thres_lev*** is processed in *cluster* mode, and the subsequent levels will be processed in *pipeline* mode

```
1:  //Pre-processing
2:  totNonzeros = 0; // Variable used in performing summation of size of all associated submatrices at a given level
3:  for $m = 1$ to num_level do
4:      for $c = 1$ to num_col[m] do
5:          // Sum up the total nonzeros size in this level
6:          totNonzeros = totNonzeros + curr_col_sz[m][c]
7:          if num_col[m] < $N_{th}$ then
8:              thres_lev = m;
9:          end if
10:     end for
11:     //Compute the average nonzeros at a given level
12:     avgNonzeros[m] = totNonzeros / num_col[m]
13: end for

14: // Parallel LU Factorization
15: for $m = 1$ to thres_lev do
16:     // The number of columns to be processed at each level
17:     lev_size = num_col[m]
18:     Adaptive cluster mode
19:     num_warps ($W_{AD}$) = st × ⌊avgNonzeros[m]/32⌋
20:     // Minimum 1 warp and maximum 32 warps
21:     num_threads = num_warps × 32
22:     // Minimum 1 block and maximum max_parallel_cols
23:     block
24:     num_parallel_cols ($N_{AD}$) =
            max_parallel_cols × sb / (32 × $W_{AD}$)
25:     while level_size > 0
26:         // Configure GPU to factorize all col in this level.
27:         // Compute num_parallel_cols blocks in parallel,
28:         // each block consists of num_threads.
29:         LU≪<num_parallel_cols , num_threads≫>
30:         level_size = level_size - num_parallel_cols
31:     end while
32: end for
33: // Total number of columns in all levels that execute in
34: // pipeline mode
35: for $m = 1$ to num_level do
36:     pipeline_size = pipeline_size + num_col
37: end for
38: Pipeline mode
39: // Configure GPU to factorize all remaining columns
40: // Constant block size and parallel columns
41: num_threads = $W_{PL}$ × 32
42: num_parallel_cols = max_parallel_cols
43: while pipeline_size > 0
44:     // Compute num_parallel_cols blocks in parallel,
45:     // each block consists of num_threads.
46:     LU_pipe≪<num_parallel_cols , num_threads≫>
47:     pipeline_size = pipeline_size - num_parallel_cols
48: end while
```

The LU facotrization in *adaptive cluster* mode starts by processing level 1 to level ***thres_lev***. The number of columns to be processed at each level is first obtained (line 17), followed by the calculation of adaptive block size (W_{AD}) and number of parallel columns (N_{AD}) (lines 19-24). Once this process completes, the GPU is configured (lines 26-29) to factorize all the columns at this particular level.

Subsequently, the algorithm moves on to compute the total number of columns to be factorized in *pipeline* mode (lines 35-37). The block size (W_{PL}) and the number of parallel columns are configured to a fixed value (lines 42-43), which are determined through experiments. Next, the GPU is configured to factorize the remaining levels (lines 44-48). Note that in *adaptive cluster* mode, the LU factorization proceeds level by level, due to the strict dependency between each level. However, in *pipeline* mode, all the remaining columns are factorized at the same time. The dependency between each level still persists in *pipeline* mode, but the LU factorization process can be executed in pipeline fashion (refer to Algorithm 2 for detailed explanation).

1) DISCUSSION

Note that Lee and Achar presented PCBSO mode in [7] for RLA, wherein the size of associated submatrices are used as a metric to configure the block size and parallel columns adaptively. The PCBSO mode is derived based on the observation that the parallelism within a block is heavily influenced by the workload (i.e., size of associated submatrices). In contrast, the LLA algorithm does not update the submatrices to the right; it only updates the current column based on the existing values from the left of current column. Hence, the parallelism within a block in LLA is only influenced by the number of nonzeros that exists in the current column. This provides the basis for the development of the proposed *adaptive cluster* mode, which is clearly different from the one proposed in [7].

B. PROPOSED DYNAMIC LEFT LOOKING ALGORITHM

Note that in the proposed *adaptive cluster* mode, while each level is being executed (Algorithm 4, lines 15-32); within each level, the GPU kernels are launched in batches due to the limitation of maximum allowable parallel columns. Similarly, the GPU kernels to be executed in the *pipeline* mode are also launched in batches (Algorithm 4, lines 44-49). This has the potential to cause the bottleneck due to the following reasons:

1) In the *adaptive cluster* mode, the execution proceeds level by level taking into account of the dependency requirements. The CPU needs to wait for the current level to complete before launching kernels for the next level. This consumes CPU resources and slows down the LU factorization performance.
2) In the *pipeline* mode, the dependency between levels can be pipelined, so the CPU does not need to proceed level by level, which can be more efficient. However, all the kernel launches are still managed by the CPU, which makes the overhead to be relatively high.

Correspondingly, to further improve the speed performance, these kernel launching activities can be executed entirely in GPU using the Dynamic Parallelism feature [13], which reduces the context switching overhead. Dynamic Parallelism is a feature offered by state-of-the-art GPUs; which allows the parent kernels in GPU to launch new child kernels without going through the CPU. This enables many useful tasks to be carried out more efficiently by the GPU. For instance, the GPU can perform recursive function calls; manage dynamic task allocation entirely inside the GPU, wherein the workload size is unknown apriori; reduce the context switching overhead for kernel launch, etc.

Using the similar approach in [5], [7], CPU first starts a GPU parent *kernel*, then continues to execute other tasks. By exploiting the Dynamic Parallelism for LLA, the GPU parent *kernel* then launches all the subsequent kernel within the GPU. This launching *kernels* within GPU reduces the need to transfer execution control between CPU and GPU [13] and consequently, reduces the context switching overhead. Also, since CPU will be responsible for launching one parent *kernel*; it is no longer required to monitor the GPU execution closely, which in turn frees up the CPU resources for other tasks. This technique can be used to manage the kernel launches in *adaptive* as well as *pipeline* modes.

IV. ANALYTICAL INSIGHTS FOR IMPLEMENTATION AND PERFORMANCE ANALYSIS OF LLA AND RLA APPROACHES

In this section, analytical insights on implementation aspects of LLA and RLA approaches on GPU platforms are presented. It also provides discussion on the performance of both the algorithms. RLA [4], [5] shows many advantages compared to LLA [3], but a direct comparison is lacking in the literature. He *et al.* [4] presented that RLA is faster than LLA, but comparison is not on an equal-footing as it does not take into account different execution modes (*cluster* and *pipeline*) in LLA. In the following part of this section, the strength and weaknesses of LLA and RLA in *cluster* mode execution are discussed. Also, certain limitations of RLA in performing *pipeline* mode execution are noted.

A. CLUSTER MODE EXECUTION IN LLA AND RLA

Both LLA and RLA can support the execution in *cluster* mode, wherein the independent columns can be executed in parallel to achieve more parallelism. However, due to potential "read-after-write" data hazard, RLA needs to employ atomic instruction to perform the MAC operation (Algorithm 3, line 10). This limits the performance of RLA, as atomic instruction has lower throughput compared to other general instruction [13].

B. COMPARATIVE DISCUSSION OF PIPELINE/STREAM MODE EXECUTION IN LLA AND RLA

For better speed performance, LLA is preferred to be executed in *pipeline* mode for levels with very less independent columns. Assuming that N_{th} is the threshold value to separate between *cluster* and *pipeline* mode executions, the value for N_{th} is usually determined through experiment [2], [3]. Levels with independent columns greater than N_{th} are executed in *cluster* mode. This allows LLA to exploit more parallelism for levels that come with lesser independent columns (for details of *pipeline* mode execution, see Algorithm 2).

RLA can also achieve the similar parallelism through the *stream* mode execution proposed by Peng and Tan [6]. In *stream* mode, multiple streams are being launched, wherein each stream processes one column using the entire GPU grid. This can effectively improve the parallelism in RLA compared to the original proposal by He *et al.* [4]. However, unlike *pipeline* mode in LLA, *stream* mode in RLA needs to process level by level in strict order. In other words, even though some columns in the current level may have already been solved, but they cannot be used by the next level yet. This is the main bottleneck in RLA *stream* mode, which can limit it's performance compared to LLA.

C. FINE-GRAIN PARALLELISM IN LLA AND RLA

Both RLA and LLA compute the parallel columns using the parallel blocks in GPU (the outer i loop in Algorithm 1 and 3). The remaining j and k loops are executed through the parallel threads within a block, which exploit the available fine-grain parallelism in LU factorization. RLA seem to be able to exploit more fine-grain parallelism compared to LLA, as the j and k loops in RLA can be parallelized, compared to only k loop in LLA. However, since the maximum parallel threads in a block is limited to 1024, the benefit of having more parallelism in RLA is also limited by this factor. In practice, we can see that the LLA can already achieve quite good parallelism on the early levels. Referring to Table 2, after level 10, rajat12 and G3_circuit already have many nonzeros in each column, which can effectively exploit the parallelism within a block. This implies that the higher parallelism in RLA may not always be an advantage compared to LLA, when it is implemented on a GPU.

V. EXPERIMENTAL RESULTS

A comprehensive computational comparison is performed in this section between LLA and RLA with state-of-the-art implementations that are proposed in this paper. The results are also compared against the widely used industrial grade KLU solver [1] and a multi-core CPU based solver, NICSLU [2]. To carry out the experiment, the proposed algorithms are implemented in C language under CUDA 10.2 SDK and compiled with optimization level 3 (-O3). The workstation used for experiments consisted of eight CPU cores (Intel Xeon i7-9700F), 16GB RAM, a GPU (Turing RTX2060) with 1920 cores and 6GB DRAM. Fifteen randomly picked circuit matrices from SuiteSparse Matrix Collection [15] are used to evaluate the proposed advancements on LLA and the state-of-the-art RLA based LU factorization [7]. Note that in our experiments, the KLU, NICSLU and all the GPU implementations are configured to perform only re-factorization, which does not involve pivoting. The pivoting process is only performed on the first factorization. This procedure is commonly used in circuit simulation and the GPU-based LU solvers [3]–[7].

The threshold (N_{th}) to divide between the *cluster* and *pipeline* mode is determined experimentally [2], [3]. For experiments performed in this section, we fixed $N_{th} = 128$ as it gave the best performance for both LLA and RLA in our GPU platform. Similarly, we set $W_{PL} = 8$ and the block size for pipeline mode is $num\_threads = 256$, due to its superior performance against other configurations in our GPU platform. For other GPU platforms, the optimal values for N_{th} and W_{PL} may be different from our values and it has to be determined experimentally.

A. PERFORMANCE COMPARISON OF LLA [3] AND THE PROPOSED LLA WITH DYNAMIC ADAPTIVE CLUSTER MODE

In this experiment, we apply the proposed *adaptive cluster* mode (ACM) with dynamic parallelism to improve the performance of LLA [3] on GPUs. Table 3 shows the GPU times using both the implementations, which are executed with double precision on the same platform for a fair comparison. Note that the work from Chen *et al.* [3] is not open source. To have a fair comparison, we have followed their implementation with the best available information, and then applied our proposed advancements onto it. As can be seen from Table 3, the proposed advancement shows better performance compared to the implementation by Chen *et al.* [3], with an average speedup of 1.36 (arithmetic mean) and 1.27 (geometric mean). Although the proposed ACM technique (column 5) on works on the cluster mode, it is still achieving

TABLE 3. Performance comparison of LLA [3] and the proposed LLA.

Matrix	Matrix size:	No. of nonzero	Time (msec)			Speedup (Dynamic
	# of rows (n)	(nnz)	LLA-Chen et al. [3]	LLA with Proposed ACM Mode	LLA with Dyn. + ACM Mode	+ ACM vs Chen et al. [3])
rajat12	1879	12818	1.73	1.38	1.37	**1.26**
circuit_2	4510	21199	2.08	1.44	1.43	**1.44**
memplus	17758	99147	1.95	1.77	1.77	**1.10**
rajat27	20640	97353	3.78	4.2	4.26	**0.89**
onetone2	36057	222596	15.22	11.08	11.61	**1.31**
rajat15	37261	443573	19.79	17.65	17.78	**1.11**
rajat26	51032	247528	5.46	6.37	6.17	**0.88**
circuit_4	80209	307604	13.32	13.95	13.82	**0.96**
rajat20	86916	604299	68.52	47.68	46.47	**1.43**
ASIC _100ks	99190	578890	102.79	56.83	49.83	**2.06**
hcircuit	105676	513072	7.38	9.13	8.13	**0.91**
transient	178866	961368	66.54	62.01	60.01	**1.11**
ASIC _320ks	321671	1827807	110.61	63.18	61.18	**1.81**
ASIC _680ks	682712	2329176	86.83	69.46	66.46	**1.31**
G3_ circuit	1585478	7660826	124264	36614	33657	**3.69**
Arithmetic Mean						**1.42**
Geometric Mean						**1.31**

TABLE 4. Performance comparison with adptive PCBSO based RLA [7] and the proposed dynamic ACM based LLA.

Matrix	Matrix size: # of	No. of nonzero	Time (msec) **Double precision**		Speedup
	rows (n)	(nnz)	RLA - Lee et al. [7]	LLA with proposed refinements	
rajat12	1879	12818	3.16	1.37	**2.31**
circuit_2	4510	21199	3.98	1.43	**2.78**
memplus	17758	99147	6.05	1.77	**3.42**
rajat27	20640	97353	8.38	4.26	**1.97**
onetone2	36057	222596	61.77	11.61	**5.32**
rajat15	37261	443573	58.44	17.78	**3.29**
rajat26	51032	247528	13.26	6.17	**2.15**
circuit_4	80209	307604	35.44	13.82	**2.56**
rajat20	86916	604299	208.37	46.47	**4.48**
ASIC _100ks	99190	578890	187.38	49.83	**3.76**
hcircuit	105676	513072	11.87	8.13	**1.46**
transient	178866	961368	390.43	60.01	**6.51**
ASIC _320ks	321671	1827807	216.05	61.18	**3.53**
ASIC _680ks	682712	2329176	184.32	66.46	**2.77**
G3_ circuit	1585478	7660826	41385.13	33657.00	**1.23**
Arithmetic Mean					**3.17**
Geometric Mean					**2.89**

reasonably good performance. This is because the previous work [3] only sets the block size to a warp (32 threads), which seriously limited the parallelism in cluster mode. In contrary, the ACM mode allows adaptive configuration of the blokc size to fully exploit the suitable parallelism in cluster mode.

As can be seen from Table 3, columns 5 and 6, application of dynamic parallelism is able to improve the performance with proposed *adaptive cluster* mode only slightly. Compared to the work by Lee and Achar [7] where dynamic

TABLE 5. Performance comparison between the proposed LLA, RLA (with state-of-the-art advancements), NICSLU and KLU.

Matrix	Matrix size: # of rows (n)	No. of nonzero (nnz)	Time (msec) **Double precision** KLU (1-core)	Time (msec) **Double precision** Proposed LLA	Speedup Proposed LLA vs KLU [1]	Time (msec) **Double precision** NICSLU (8-core)	Speedup Proposed LLA vs NICSLU [2]
rajat12	1879	12818	2.08	1.37	**1.52**	1.20	**0.88**
circuit_2	4510	21199	4.15	1.43	**2.90**	1.44	**1.01**
memplus	17758	99147	6.22	1.77	**3.51**	2.08	**1.18**
rajat27	20640	97353	8.14	4.26	**1.91**	3.12	**0.73**
onetone2	36057	222596	1021.66	11.61	**88.00**	112.15	**9.66**
rajat15	37261	443573	105.09	17.78	**5.91**	23.42	**1.32**
rajat26	51032	247528	15.17	6.17	**2.46**	4.32	**0.70**
circuit_4	80209	307604	32.24	13.82	**2.33**	15.58	**1.13**
rajat20	86916	604299	1535.98	46.47	**33.05**	699.24	**14.4**
ASIC _100ks	99190	578890	1005.67	49.83	**20.18**	159.37	**3.20**
hcircuit	105676	513072	32.37	8.13	**3.98**	11.07	**1.36**
transient	178866	961368	588.16	60.01	**9.8**	286.95	**4.78**
ASIC_320ks	321671	1827807	3718.43	61.18	**60.78**	414.12	**6.77**
ASIC_680ks	682712	2329176	1205.28	66.46	**18.14**	218.43	**3.29**
G3_circuit	1585478	7660826	205474.19	33657	**6.10**	81104.17	**2.41**
				Arithmetic Mean	**17.34**		**3.52**
				Geometric Mean	**7.57**		**2.19**

parallelism provide significant additional improvement over the adaptive PCBSO mode for RLA, the gain provided by dynamic parallelism in LLA is not comparatively significant. This can be explained through the following observation. In RLA [7], the threshold to select between adaptive PCBSO and stream mode is the number of streams (*n_streams*), which is capped at 32 (maximum available streams) due to the hardware limitation [13]. All levels that have columns greater than *n_streams* are executed in adaptive PCBSO mode; otherwise they are executed in stream modes. Similarly, N_{th}, which is the threshold between *adaptive cluster* and *pipeline* mode, which is set based on experiments concerning GPU architectures. Comparing these two thresholds (*n_streams* and N_{th}), it is obvious that, for a given same circuit, there are more levels being executed in adaptive PCBSO mode (RLA) compared to *adaptive cluster* mode in LLA. Note that both the adaptive PCBSO RLA and the proposed *adaptive cluster* mode LLA require executions level by level, which is slow if the kernel launches are managed by the CPU. This implies the performance of these two modes can be further improved greatly by employing dynamic parallelism. However, since adaptive PCBSO RLA processes more levels than *adaptive cluster* mode (LLA), the performance gain obtained from dynamic parallelism is also more significant in the case of RLA. On the other hand, stream mode (RLA) and *pipeline* mode do not benefit from dynamic parallelism as they do not need to be synchronized level by level.

B. PERFORMANCE COMPARISON OF BETWEEN LLA AND RLA

In this experiment, LLA with state-of-the-art refinements is compared with the state-of-the-art implementation of RLA (Lee and Achar [7]), which includes both *batch* and *pipeline* modes with adaptive PCBSO mode. As can be seen from Table 4, the proposed LLA refinements outperform RLA with state-of-the-art implementation. The LLA with proposed improvements provide an average speedup of 3.17 (arithmetic mean) and 2.89 (geometric mean) compared to [7]. This shows that when the LLA and RLA are implemented on the same GPU with state-of-the-art refinements, LLA can perform much better than RLA.

C. PERFORMANCE COMPARISON OF THE PROPOSED LLA AND RLA [7], NICSLU [2] AND KLU [1]

Restuls are summarized in Table 5. As can be seen from Table 5, LLA implementation in GPU provides superior results compared to KLU in all the circuits that we have investigated, with an average speedup of 17.37 (arithmetic mean) and 7.57 (geometric mean). The speedup tends to be larger when the matrix size increases, since the GPU performs well in the presence of increased workload, fully utilizing its massively parallel architecture. The proposed LLA implementation also performs better than NICSLU (configured with eight-core), providing a speedup of 3.52 (arithmetic mean) and 2.19 (geometric mean).

D. DISCUSSION: LOOKING LEFT (LLA) OR RIGHT (RLA) FOR GPU BASED LU FACTORIZATION

At the onset, RLA [4], [5], [7] may look advantageous compared to LLA [3] in terms of parallelism. Referring to Algorithm 1, the j loop (line 4) in LLA is not parallelizable; in contrast, RLA can achieve full parallelization for all three **for** loops (refer to Algorithm 3). Based on this observation, one may be tempted to claim that RLA can be faster than LLA when it is implemented in a massively parallel architecture like GPU. However, RLA has several shortcomings, particularly the read-after-write hazard, which seriously affects its performance compared to LLA.

Race condition may occur in RLA (Algorithm 3), because the submatrix update for each column (lines 8-11) affects

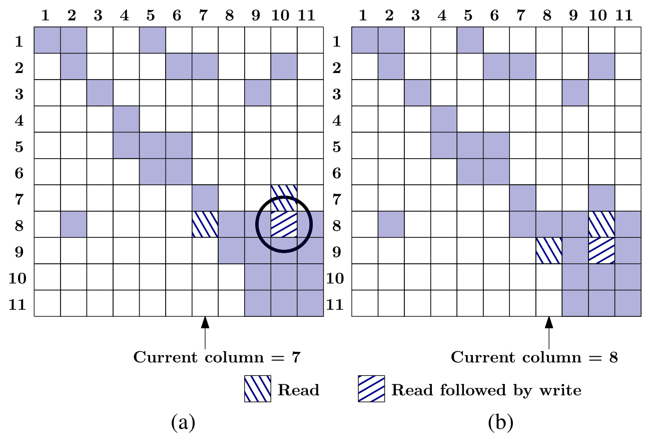

FIGURE 3. Additional dependency for hybrid column based RLA. (a) A block processing column 7. (b) Another block processing column 8 simultaneously.

the computations of other columns to the right. To avoid potential data hazard, the MAC operation in RLA (line 10) requires an atomic instruction, which only allows one thread to access and modify the data, one at a time. Under such situation, if multiple threads attempt to modify the data at the same time, the execution for this atomic instruction will be serialized to avoid data hazard. This greatly impacts the overall performance of RLA, since the MAC operation (line 10) is executed very frequently. On the contrary, all operations in LLA (Algorithm 1) do not introduce any race condition, and hence atomic operations are not required. Based on this observation and the performance comparisons, the disadvantages of RLA in cluster mode execution are obvious. These observations are also verified through our numerical experiments, which demonstrate that LLA can achieve much better speed performance compared to RLA within the same GPU platform.

VI. CONCLUSION

In this paper, algorithmic advancements are proposed to improve the performances of LLA implementations in GPU. The proposed *adaptive cluster* mode configures the block size and parallel columns adaptively based on the average number of nonzeros in each level. This improves the performance of LLA compared to the previous work by Chen *et al.* [3]. It also presents insights into the implementation and associated performance issues of both RLA and LLA. The experimental results show that LLA provides superior performance compared to the RLA when they are implemented on the same GPU platform with state-of-the-art advancements.

APPENDIX A
DEPENDENCY LEVELLING IN LLA AND RLA

This is illustrated in Fig. 3 using the same example matrix of Fig. 1. Referring to Fig. 3, 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 dependency levels as in Table 1. However, if RLA is used, column 8 can't 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. 3. 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. 3(a)). However, if column 8 were also to be run in parallel at the same time, it could end up reading the non-updated circled element to perform its own MAC operation (see Fig. 3(b)) to update the element below the circled element.

ACKNOWLEDGMENT

The authors would like to thank the Dr. Xiaoming Chen of Institute of Computing Technology, Chinese Academy of Sciences for his inputs on the NICSLU and its GPU implementation.

REFERENCES

[1] 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, pp. 1–17, Sep. 2010.

[2] 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.

[3] 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.

[4] 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.

[5] W.-K. Lee, R. Achar, and M. S. Nakhla, "Dynamic GPU parallel sparse LU factorization for fast circuit simulation," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 11, pp. 2518–2529, Nov. 2018.

[6] S. Peng and S. X.-D. Tan, "GLU3.0: Fast GPU-based parallel sparse LU factorization for circuit simulation," *IEEE Design Test*, vol. 37, no. 3, pp. 78–90, Jun. 2020.

[7] W.-K. Lee and R. Achar, "GPU-accelerated adaptive PCBSO mode-based hybrid RLA for sparse LU factorization in circuit simulation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 11, pp. 2320–2330, Nov. 2020.

[8] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. G. Shet, G. Chrysos, and P. Dubey, "Design and implementation of the linpack benchmark for single and multi-node systems based on Intel Xeon Phi coprocessor," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, May 2013, pp. 126–137.

[9] 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.

[10] 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.

[11] 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* (Lecture Notes in Computer Science), vol. 6449. Berlin, Germany: Springer, 2011, pp. 71–82.

[12] 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.

[13] (Jun. 2021). *CUDA Programming Guide v11.4*. Accessed: Jun. 2021. [Online]. Available: https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf

[14] (Jul. 2018). *NVIDIA CUDA Compiler Driver NVCC, TRM-06721-001_v9.2*. [Online]. Available: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html

[15] T. Davis. *SparseSuite Matrix Collection*. Accessed: Jun. 2021. [Online]. Available: https://sparse.tamu.edu/

WAI-KONG LEE (Member, IEEE) received the B.Eng. degree in electronics and M.Sc. degree from Multimedia University, in 2006 and 2009, respectively, and the Ph.D. degree in engineering from Universiti Tunku Abdul Rahman (UTAR), Malaysia, in 2018. He was a Visiting Scholar at Carleton University, Canada, in 2017, Feng Chia University, Taiwan, in 2016 and 2018, and OTH Regensburg, Germany (2015 and 2018–2022). He is currently a Postdoctoral Researcher at Gachon University, South Korea. Before that, he served as an Assistant Professor at UTAR. Prior to joining the academia, he worked in several multi-national companies (including Agilent Technologies), as a Research and Development Engineer. His research interests include cryptography, numerical algorithms, GPU computing, deep learning quantization, and the Internet of Things.

RAMACHANDRA ACHAR (Fellow, IEEE) received the B.Eng. degree in electronics engineering from Bangalore University, India, in 1990, the M.Eng. degree in micro-electronics from the Birla Institute of Technology and Science, Pilani, India, in 1992, and the Ph.D. degree in electrical engineering from Carleton University, in 1998.

He is currently a Professor with the Department of Electronics Engineering, Carleton University. Prior to joining Carleton University Faculty in 2000, he served in various capacities in leading research laboratories, including T. J. Watson Research Center, IBM, Ossining, NY, USA, in 1995; Larsen and Toubro Engineers Ltd., Mysore, in 1992; the Central Electronics Engineering Research Institute, Pilani, in 1992; and the Indian Institute of Science, Bangalore, India, in 1990. He has published over 200 peer-reviewed papers in international transactions/conferences, six multimedia books on signal integrity, and five chapters in different books. His 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 Founding Faculty Member of the Canada–India Center of Excellence, the Chair of the joint chapters of CAS/EDS/SSC societies of the IEEE Ottawa Section, and a Consultant for several leading industries focused on high-frequency circuits, systems, and tools. He is also a practicing Professional Engineer of Ontario and a fellow of the Engineers Institute of Canada. He received several prestigious awards, including the Carleton University Research Achievement Awards, in 2010 and 2004, the Natural Science and Engineering Research Council (NSERC) Doctoral Medal, in 2000, the University Medal for the Outstanding Doctoral Work, in 1998, the Strategic Microelectronics Corporation (SMC) Award, in 1997, and the Canadian Microelectronics Corporation (CMC) Award, in 1996. He was also a co-recipient of the IEEE Best Transactions Paper Award of IEEE TRANSACTIONS ON ADVANCED PACKAGING (T-AdvP), in 2007, and IEEE TRANSACTIONS ON COMPONENTS, PACKAGING AND MANUFACTURING TECHNOLOGY (T-CPMT), in 2013. His students have won numerous best student paper awards in international forums. He currently serves as the Chair for the Distinguished Lecturer (DL) Program of the EMC Society and DL for the Electron Devices Society. He also previously served as the DL for the CAS Society, in 2011 and 2012, and EMC Society, in 2015 and 2016. He also serves on the executive/steering/technical-program committees of several leading IEEE international conferences, including EPEPS, EDAPS, SPI, and in the technical committees of EDMS (TC-12 of CPMT). He previously served as the General Chair for HPCPS, from 2012 to 2017; the General Co-Chair for 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 of Government of India, under the SMDP-II program, in 2012. He previously served as a Guest Editor for IEEE TRANSACTIONS ON COMPONENTS, PACKAGING AND MANUFACTURING TECHNOLOGY (CPMT), for two Special Issues on Variability Analysis and 3D-ICs/Interconnects, in 2015.