International Journal of Innovative Technology and Exploring Engineering (IJITEE)
ISSN: 2278-3075 (Online), Volume-9 Issue-4, February 2020

Cluster-Based and GPU-Driven Parallel Computing Model to Accelerate Circuit Simulation

Shital V. Jagtap, Y.S. Rao

Abstract: ***In this digital age circuit design, analysis and validation is not only fundamental step but quite crucial in all the industries and in research. Simulation software is available for circuit analysis, but they all prove to be slower for very large circuit simulation or to execute thousands of iteration of transient analysis. Accelerating simulator is as important as speeding up circuit design. In this paper we have addressed circuit analysis using parallel computing approach on Graphics Processing Unit (GPU). Now a day's high end GPUs are available with sufficient memory in the architecture itself. Circuit processing functions are analysed to search compute intensive functions. Mathematical operations are redesigned so that it will execute in parallel. LU decomposition algorithm and complex math operations are converted in parallel form. Some mathematical operations are simplified to merge them in suitable cluster. Clustering approach is used which helps in finding kernel of uniform operations to map on GPU cores. GPU programming strategies like if-else in-lining, parallel reduction etc are useful in accelerating circuit operations. Use of look up tables in shared memory or constant memory proves to be useful in fast data access. At least 15% speed gain is achieved for operational analysis and 40% for transient analysis of regular circuits.***

Keywords **: *Clustering, GPU (Graphics Processing Unit), LU decomposition, Parallel computation, Transient analysis.***

I. INTRODUCTION

Analysis of an electronic circuit is crucial in almost all the electronic industries, academics and in research. Many circuit simulation software available for circuit design but simulation of large digital and mixed-signal integrated circuits is one of the challenges of the electronics industries[1]. Testing large circuit containing millions of components as a whole needs days or week of time. As number of components in the circuit increases, simulation time increases exponentially. Costly distributed systems or supercomputers can make it fast, but all the users can not use it.

Revised Manuscript Received on February 28, 2020.
***** Correspondence Author
Shital V. Jagtap*, Research Scholar, Computer Engineering Department, Ramrao Adik Institute Of Technology, Navi Mumbai, India. Email: svjagtap@gmail.com.
Dr. Y. S. Rao, Principal, Sardar Patel College of Engineering, Mumbai, India. Email: ysrao@spit.ac.in.

Fig. 1. Netlist parsing and kernel development

To reduce simulation time, cluster driven and GPU based parallel system is proposed in this paper.
NGSPICE simulator is widely used for circuit simulation and source code is also available for research. So this code is modified here to adapt parallel programming strategies. As per figure 1, input-netlist is first parsed using Graph DFS algorithm to find relationship and connection among the components and its parameters. Some sparse library functions are redesigned by adding GPU strategies for parallel programming. Reduction, if else in-lining, and kernel optimization are applied on all the functions that are executed in parallel. Component and method clustering algorithm is applied. Simulator APIs parse or compile netlist file that specifies the interactive relationship between different components and store parameters in different data structures. Circuit parameters are initialized with direct or calculated values. Direct values are like '0.0' or '1' or output from previous step. Calculated values are like output of simple arithmetic or logical calculations. Additionally, some complex mathematical operations (like integration) are applied on these parameters.
Figure 2 show the operations executed on GPU. All the original sequential methods cannot be executed on GPU as they are optimized for serial execution. From the complete simulation library, some compute intensive initialization, mathematical and cluster methods are selected for parallel execution.

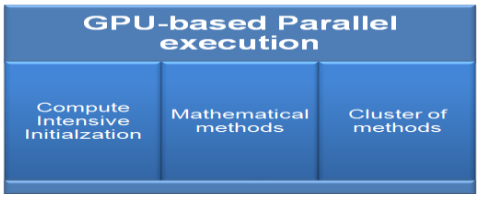

Fig. 2. Parallel execution model

Compute intensive operations and dataset derived from clustering are mapped on parallel processors. In transient analysis, these iterations use millions of clock cycles. Some operations need common processing and common parameters. Clustering approach on these parameters and their operations are helpful. Clustering is applied on these components and parameters to help mapping of functions on many processors of GPU. Uniformity in mathematical operations and relative weights are used to form clusters. These clusters are then mapped to GPU for parallel processing.

This paper addresses clustering-based GPU-accelerated circuit simulation. Section I is introduction to the topic. Section II gives the details of previous approaches to accelerate circuit simulation. Section III explores GPU-parallel programming strategies used for the system. Section IV explains circuit data analysis and clustering methodology. Section V and Section VI display the result, performance discussion, and conclusion.

II. RELATED WORK

Circuit simulators are quite crucial and can accelerate research and development. Several research papers have proposed various models for acceleration using algorithm or design optimization, parallel or distributed computing etc. GPU gained popularity by 2007 and has been in use for general purpose compute intensive operations. Gulati and Croix used GPU for VLSI circuit simulation in 2009 for the first time [2]. They identified transistor model as time consuming in circuit testing and converted it into GPU executable parallel model using basic GPU execution strategies such as if-else in-lining, forming optimum kernel, coalesced memory access etc.

As it was not completely model specific, several researchers extended this idea to various circuit acceleration models. However, as many circuit components use different mathematical methods, this system is somehow circuit specific. Chatterjee and DeOrio implemented first event-driven, gate-structure partitioning-based circuit acceleration [3], [4]. This simulator considers gates that get event signal rather than wasting time in whole circuit processing. This idea is also good to extend the concept for making gate-cluster and process it in parallel. Rather than finding compute intensive, functions design-level partitioning is used and executed in parallel. LU decomposition on large sparse matrix is always time consuming, so many strategies are used to speed up the decomposition and substitution to find variables.

Guiming and Gregory invented hardware technique such as blocking LU decomposition for FPGA circuit [5]. Some more FPGA-based techniques are also available. Shao and Jiang used pivoting operations on FPGA [6] and Manish and Nitin developed scalable block LU decomposition technique [7]. However, this approach is difficult to embed in some simulators. Compared to FPGA techniques, GPU parallel programming is simple and cost effective. Leandro and Anderson showed the importance of managing element access from memory of GPU to speed up data access for LU decomposition [8]. Besides these techniques, some solvers have been developed such as NICSLU, PARDISO, and so on that are equipped with powerful software tools to accelerate speed [9], [10]. NICSLU utilizes a column-level-dependence graph to plan the task [5]. The dependence is identified from the symbolic structure of the factors. Elimination tree (ET) is used to represent the dependence as the structure of the factors cannot be determined before factorization. It follows all the basic steps such as numerical update, pruning with proper pivoting, and others. These steps consume extra time to process simple dense small size matrix. Chen and Wang used pivoting reduction technique for LU sparse solver [10], which is a modified version of traditional solvers. Chen and Wang modified it for circuit simulation but for large circuit, matrix column dependency causes slower simulation. They have also modified these solvers to take advantage of GPU and implemented parallel processing [11], [12, [13]. Davis and Natarajan proposed KLU data structures and algorithm that can be used in circuit data storage and processing for sparse matrices [14]. They give stability in matrix processing. For small circuit simulation, it proves to be slow but has been adopted by many simulators such as NGSPICE. Murray accelerated Runge-Kutta integrators using GPU [15] that are useful in various integration applications. Saol, Vuducl, and Xiaoye proposed distributed approach to solve sparse matrix [16]. It is costly when compared to GPU-based parallel execution. Bandara and Ranasinghe compared various algorithms of LU decomposition for parallel processing on GPU [17]. To accelerate circuit simulation, the researchers in this experiment have adapted left looking algorithm. Dong and team suggested GPU-based LU decomposition algorithm for small matrices [18]. Circuit matrices are almost sparse, else we can even use LU decomposition algorithm given by Galoppo [19]. So instead of using available sparse library, there is scope in modifying it for parallel processing.

Researchers in this paper focus on utilizing parallel processing computational power of GPU for complex computations not just by redesigning circuit operations but also by using clustering that makes it automated and faster. Compared to gate-level partitioning, component clustering is easy to group all types of components. New faster GPUs are coming up in the market with better architectural features. Hence, this research is still persistent in accelerating simulation.

III. PARALLEL COMPUTING STRATEGIES

The modern GPU is available with several programmability features and at a very low cost. This GPU is preferred for performance growth in variety of complex compute-intensive applications. GPU gives a SIMD parallel computing platform not only for traditional graphics but also for general-purpose heavy computations. GPU includes thousands of many-core, multi-threaded, SIMD processors with inbuilt memory hierarchy of different sizes and access time. They also have high-bandwidth memory interfaces. Each CUDA kernel launches grid and blocks of threads. These threads execute themselves in parallel, asynchronously from CPU. Compute intensive operations, equipped to transfer data for to-and-from memory, are preferred for GPU implementation.

Sequential code is not directly executable on GPU; it needs many different strategies to make it executable on parallel device. Same sequential algorithm should not be executed on parallel device; else it may take more time than sequential algorithm.

Some common redesign strategies have been used and some have been modified that adapt themselves to circuit analysis. These strategies are also suitable for cluster execution on parallel device. Kernel code optimization is required in data storage/access through memory, number of resources such as thread, avoiding divergence, and so on.

- Parallel reduction to reduce time complexity from O(n) to O(log n). If mathematical operation follows commutative or associate property then parallel reduction is applicable.
- Optimum kernel size with respect to available memory and data requirement for kernel.
- Use of proper synchronization is needed. Also in selection of warp, threads should not be idle. Operations can be accelerated using if-else in-lining, coalesced memory access.
- Avoid thread and warp divergence
- Copy Structure array, instead of copying individual item in GPU memory.
- Copying standard lookup table and frequently used data in nearest level of memory.

IV. CIRCUIT ANALYSIS AND CLUSTERING

To accelerate circuit analysis, two different approaches have been merged as per Figure 3.

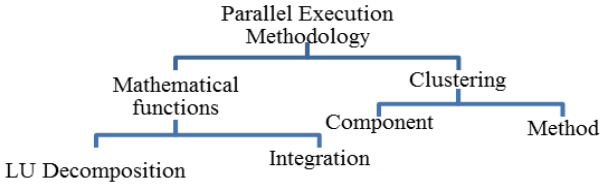

Fig. 3. Parallel execution methodology

First approach is to redesign mathematical functions used in the circuit processing. For example, LU decomposition algorithm is redesigned to execute in parallel. Many simple mathematical functions such as arithmetic operations, logical operations, finding maximum, and others are executed in parallel, with the condition that it is executable on large dataset. Second approach is to use machine learning methods. Circuit analysis functions are classified based on uniformity in components and methods. Clusters are formed based on suitability for parallel processing. Those component and method clusters are modified to adapt GPU programming strategies, which is suitable for circuit analysis and removes data dependency.

Finding compute-intensive and time consuming functions of circuit analysis is essential as these are the most relevant contributions to accelerate speed. All the 'load' functions in simulator are compute-intensive. Millions of basic parameters initialization through direct values or calculated values is involved in load function. Direct values are like current temperature or voltage from voltage source etc. Direct values also include '0' or '1' initialisation. Indirect values are derived from direct values, by applying some mathematical function. It initializes parameters such as temperature, TCL, length, pointers, coefficients, positive-negative voltage, and current direction of millions of components; finds conductance values and loads itself into the simulation matrix and its execution time sums up to approximately 54% of total analysis time [1]. Initialization of hundreds of basic parameters cannot be skipped, else result cannot be accurate. Some parameters need to be re-initialized for every transient iteration.

Other time consuming element, which is actual matrix solving, results into unknown circuit node parameters that constitutes about 36% [1]. Most of the elements of a matrix resulting from MNA(Modified Nodal Analysis) are zero, whereas non-zero pattern has no regularity. So to complete the analysis of time spent in the transient analysis, there may be 5% error of time in circuit calculation. In addition, 2% time in NI iterations, the code for Newton-Raphson iterations, 2% in NI integration, and 1% is spent in rest of simulator. Sparse matrix nature may affect output stability, speed, and accuracy, if matrix on parallel device is executed directly.

A. LU Decomposition

LU decomposition on GPU is really a challenging problem as there is high data dependency and irregular memory access. Out of all mathematical methods of circuit analysis, LU decomposition is more time consuming, and execution time increases with increase in components. So it has been used selectively for parallel execution. LU decomposition forms upper and lower triangular matrices of pre-processed circuit matrix. Left looking LU decomposition algorithm is more suitable for parallel conversion as both upper and lower triangular matrices proceed column by column. Finally using backward and forward substitution, it generates parameter values. Equation (1) and (2) show operations involved and executed in many iterations. Here 'a' is the matrix element whereas 'm' is the multiplication factor generated from different 'n' size column elements. Gaussian Elimination zero out sub-diagonal elements by some row combinations to generate lower-upper triangular elements using following equations:

$$m_{i,k} = \frac{a_{i,k}}{a_{k,k}} \quad (1)$$

$$a_{i,j} = a_{i,j} - m_{i,k} a_{k,j} \quad (2)$$

Here variable index range is like $k = 1$ to n and $j = k + 1$ to n. During the kth submatrix modification, the element $a_{i,j}^{(k)}$ is computed based on the multiplier $m_{i,k}^{(k-1)}$. While the multiplier $m_{i,k}^{(k)}$ is calculated using $a_{i,j}^{(k)}$in the k+1th column updation. So due to the dependent relation of the data, the column calculation and the sub-matrix modification must be done serially. During the kth sub-matrix modification, the elements of different columns $a_{i,,k+1}^{(k)}, a_{i,k+2}^{(k)}, \ldots, a_{i,n}^{(k)}$ are calculated using the same multiplier $m_{i,k}^{(k-1)}$. So for different columns, the sub-matrix modification can be performed on parallel level. Further, during the kth sub-matrix modification, when the element $a_{i,k+1}^{(k)}$ is obtained, it can be used to perform the k+1th column calculation and is not required to wait for the kth iteration to complete.

In algorithmic form, sequential LU decomposition algorithm is shown in algorithm 1 and its worst case execution time complexity is O(N^3). Here 'N' indicates approximate component count (or sometimes nodes).

```
for j = 0 to N
  for i = 0 to N
   if i < = j
    for k = 0 to i
      Uij = Uij - Lik * Ukj
   else
    for k = 0 to j
     Lij = Lij - Lik * Ukj
  Lij = Lij/ Ujj
```

Algo 1. LU decomposition algorithm (Sequential)

It updates matrix data column by column to generate lower and upper triangular matrix. Here, each succeeding column is dependent on previous column. For small circuit, matrix size is small, while for large circuit 'N' can exceed 1000. To process big matrices, GPU-based matrix operations are found very effective. Simplified operation for LU decomposition is shown in Figure 4 and (3).

Fig. 4. Column access in parallel

$$A_i = A_i - C * B_i \quad (3)$$

In LU decomposition, calculations of column elements are independent of each other. Each succeeding column is dependent on data available in previous columns. By finding independency, if we map one column to processor set and calculate all values of one column in parallel, it gives accurate result. As shown in Figure 4, P1 processor reads data B1, performs calculation and the result is stored at A1. Similarly, other processors work on data available at consecutive locations. Modified parallel algorithm is shown in algorithm 2. In this algorithm, variable j indicates thread number that points to column of matrix.

```
for i = 0 to N
   if i < = j
    for k=0 to i
     Uij = Uij - Lik * Ukj
   else
    for k=0 to j
     Lij = Lij - Lik * Ukj
Lij = Lij/ Ujj
```

Algo 2. LU decomposition algorithm (Parallel)

Launching N threads perform operations in parallel. As one column is dependent on previous column, execution of two or more columns in parallel is not possible. One column of matrix is assigned in the kernel and executed at a time on parallel cores. One thread calculates one element of column. Proper synchronization is required to avoid any data dependent operation. Worst-case time complexity of parallel LU algorithm is $O(N^2)$. Constants or fixed data such as diagonal of lower triangular matrix can be initialized prior to save execution cycles. Minimum 'if' statement has been tried to use as far as possible as it may cause warp or thread divergence. Proper initialization of L and U matrices can reduce some of the 'if' statements. GPU contains on-chip levels of memory with different access speed. Shared memory accelerates the speed as compared to global memory access in GPU.

B. Clustering Model

To select functions for parallel computing, machine learning approach has been used. Instead of manually assigning the procedures and functions on GPU, clustering helps in automated or dynamic decision making and is suitable for every circuit. In large circuit, there are 1000+ parameters with different data types for each component. Various filters are required to decide when to process that parameter. So clustering is a systematic approach for parallel mapping. It helps in generating automated system for component and method selection, which will be applicable for every circuit analysis. Here, we create clusters of components and methods. Clustering means to group similar elements together and one with different property has to be added in different cluster. Depth First Search (DFS) algorithm is used to trace all the components in the circuit. Component clusters help in grouping of items having same properties whereas method clusters help in merging uniform methods together. K-means clustering is used as the centroids are pre-decided, simple to understand. As per accuracy threshold, number of iterations is decided. In this project, two types of clustering have been used.

Consider a dataset D which contain n number of components/methods and having k number of clusters. These objects are organized into k partitions such that k<=n and each partition represents a cluster. It is assumed that each data item present belongs to one cluster and each cluster has one data item present. Intension of this method is to reduce the variance between the objects of the cluster and also, at the same time, have a large variance between objects of different clusters.

K-means because each of the k clusters present is represented by the type of the electronic component present within the clusters. It is also known as the centroid method, where the centroid of each cluster is determined and then the points of the clusters are reallocated to the cluster whose centroid is the closest to it.

Benefits of Clustering for circuit analysis-

- Scalable to any circuit.
- Generate set of uniform components and methods suitable for SIMD parallel execution.
- Produce results in such a manner that the results are usable and comprehensible for parallel analysis.
- Ability to deal with any circuit component with any number of transient iterations.
- Ability to deal with any data type.
- Ability to deal with outliers.

1. Component Clustering

In component clustering, similar components are clustered together. Advantage of component clustering is that the same type of processing is applicable to all the instances of cluster. Processing includes parameter initialization, setup, loading, conductance calculation and so on. If components are same, sub-operations are exactly the same. It helps in accelerating speed as compared to processing with other components in parallel. Property structure, number of component properties, and its data type are the parameters used to perform classification and clustering. Variance is calculated using formula (4).

$$variance = \sum \frac{(x - x')^2}{n} \quad (4)$$

Variance of all similar components is almost zero. For dissimilar components, variance is nonzero large value. We can set desired threshold to accept it in cluster or not. If parameters processing for different components are same, they are also grouped in one cluster. For example, different voltage sources can be grouped together.

2. Method Clustering

Method clustering is more complex and time consuming than the component clustering. However, method clustering is more suitable for GPU computing, as many thread instance formations are possible due to SIMD nature of GPU. Component analysis involves many processing methods such as initialization, load, mathematical operations, and so on. Also, many components have common and simple processing methods such as initializing parameters with value '0' or some temperature-based calculation etc. Such common methods can be grouped together in one cluster. Initialization, load, setup, and the methods that do not have any data dependency can be grouped together. We tried to increase intra-cluster similarity compared to inter-cluster similarity. Figure 5 shows the method cluster of common uniform executable methods.

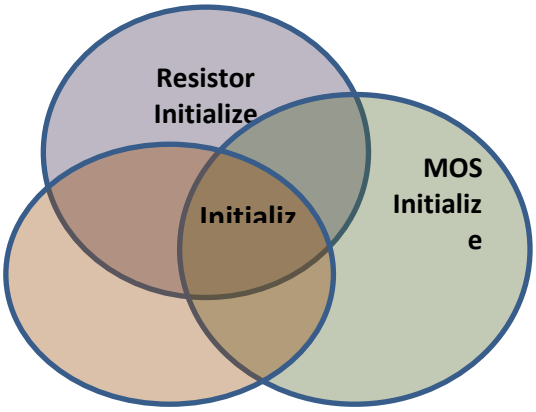

Fig. 5. Method clusters

There are limitations on GPU memory size and extra time is needed in loading and unloading all the parameters. Hence, if serial time is much more as compared to parallel cluster execution time plus loading time, it is suitable to make it parallel. Once clusters are formed, these are executed on GPU. GPU programming optimization strategies are applicable to modify and redesign the code.

V. EXPERIMENTAL VERIFICATION

To test circuit analysis, we used popular NGSPICE simulator. Source code of sequential NGSPICE simulator was modified to execute it on NVIDIA graphics card using CUDA programming. Graphics card used for testing was GeForce 840M card with 384 cores with compute capability 5.0 and 2 GB graphics card memory. Ubuntu 14.04 version was used for NGSPICE coding. Circuit netlist file is executed in two steps. First step is netlist parsing to understand components and their specifications. Netlist parsing time is constant for all the circuits on serial and parallel device, so it is neglected. Second step is actual circuit execution. Serial and parallel versions on operational and transient analysis are considered for comparison. Total execution time considered is for clustering, parameter setup, load, and mathematical operations. In transient analysis 1000+ iterations are considered to get accurate intermediate results.

Serial execution on CPU is tested verses parallel execution time on GPU. Speed gain is calculated using the equation (5)

$$\text{Percentage speed gain} = 100 * \frac{(\text{serial} - \text{parallel time})}{\text{parallel time}} \quad (5)$$

Minimum execution time is considered for every circuit execution.

Two types of analysis are tested: (1) operational analysis and (2) transient analysis. Large and small circuits are considered for operational and transient analysis respectively.

Table I shows serial and parallel execution time of operational analysis of some of the netlist having 100+ components or sub-circuits. It shows that serial execution time is more than parallel execution time.

Table- I: Serial and parallel execution time of operational analysis

Net-List	Circuit	Serial execution (MicroSec)	Parallel execution (MicroSec)	Speed gain (%)
1.	AD629B	227	195	16
2.	AD1580	285	235	21
3.	AD584	337	287	17
4.	SSM2212	361	310	16
5.	AD587	487	396	23
6.	PM1012	1112	901	23
7.	AD5144	2680	2197	22
8.	AD588	7353	6157	19

As number of components in the circuit increases, parameter copying time from CPU to GPU memory also increases. However, average speed gain is more than 15%.

Table- II: Serial and parallel execution time of transient analysis

Net-List	Circuit	No. of iterations	Serial execution (Sec)	Parallel execution(Sec)	Speed gain (%)
1	AC sine wave voltage	10000	6.38	3.75	70
2	RC circuit	1008	0.853	0.58	47
3	Full wave bridge rectifier	10000	12.54	8.23	52
4	Common source JFET amplifier	10000	8.71	4.26	104
5	Integrator with square wave input	2520	2.115	1.4875	42

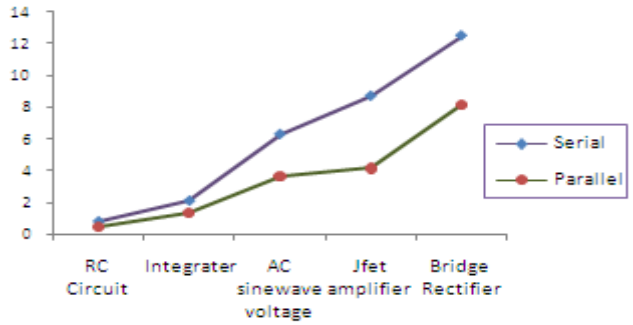

Fig. 6. Transient analysis for serial vs parallel executions time

Table II and Figure 6 show serial and parallel execution time of transient analysis of fundamental circuit netlist such as RC, Integrator, AC sine wave voltage generator, JFET amplifier, bridge rectifier having few components or subcircuits. Here, serial time is measured in NGSPICE version having KLU data structure [20]. Circuit is tested for thousands of transient iterations. It shows that basic GPU strategies and clustering approach accelerate circuit processing at least by 40% and increases subsequently for more iterations.

VI. CONCLUSION

In sequential processing, analysis time increases exponentially if we embed more components. But in parallel handling, average analysis time increases linearly as we add more components in the circuit. Cluster size and cluster formation time varies with number of circuit components. Speed gain in operational analysis is more than 15%.

Speed gain in transient analysis is good as compared to operational analysis and goes beyond 40%. In transient analysis, speed gain increases as we increase number of iterations.

REFERENCES

1. S.V. Jagtap and Y.S. Rao, "Clustering and Parallel Processing on GPU to Accelerate Circuit Transient Analysis", Springer Advances in Intelligent Systems and Computing (AISC, volume 870), book chapter. Available: doi>https://doi.org/10.1007/978-981-13-2673-8.
2. Kanupriya Gulati, John F. Croix, Sunil P. Khatri, and Rahm Shastry, "Fast Circuit Simulation on Graphics Processing Units", IEEE XPlore, Available: doi>10.1109/ASPDAC.2009.4796514.
3. Debapriya Chatterjee, Andrew DeOrio, and Valeria Bertacco, "Event Driven Gate-Level Simulation with GP-GPUs", ACM Digital Library. Available: https://dl.acm.org/citation.cfm?id=1630056.
4. Debapriya Chatterjee, Andrew DeOrio, and Valeria Bertacco, "Gate-Level Simulation with GPU Computing", ACM Transactions on Design Automation of Electronic Systems, ACM Digital Library. Available: https://dl.acm.org/citation.cfm?id=1970363.
5. Guiming Wu, Yong Dou, and Gregory D. Peterson, "Blocking LU Decomposition for FPGAs", 2010, 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines. Available: https://dl.acm.org/citation.cfm?id=1827863.
6. Yi Shao, Liehui Jiang, Qiuxia Zhao, and Yuliang Wang, "High Performance and Parallel Model for LU Decomposition on FPGAs", 2009 Fourth International Conference on Frontier of Computer Science and Technology, Shanghai, China, Available: IEEE XPlore, doi>10.1109/FCST.2009.66.
7. Manish Kumar Jaiswal and Nitin Chandrachoodan, "FPGA-Based High Performance and Scalable Block LU Decomposition Architecture", IEEE Transactions on Computers, Vol. 61, No. 1, Jan 2012. Available: IEEE XPlore, doi>10.1109/TC.2011.24.
8. Leandro F. Cupertino, Anderson P. Singulani, Cleomar P. da Silva, and Marco Aur´elio C, "Pacheco. LU Decomposition on GPUs: The Impact of Memory Access", IEEE XPlore Digital Library. Available: IEEE XPlore, https://ieeexplore.ieee.org/document/5645390.
9. Xiaoming Chen, Yu Wang and Huazhong Yang, "NICSLU: An Adaptive Sparse Matrix Solver for Parallel Circuit Simulation", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 32, No. 2, February 2013. Available: IEEE XPlore, doi>10.1109/TCAD.2012.2217964.
10. Xiaoming Chen, Yu Wang, and Huazhong Yang, "A Fast Parallel Sparse Solver for SPICE-based Circuit Simulators", CiteSeerX. Available: doi>http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.724.4153.
11. Ling Ren, Xiaoming Chen, Yu Wang, Chenxi Zhang, and Huazhong Yang, "Sparse LU Factorization for Parallel Circuit Simulation on GPU", IEEE XPlore Digital Library. Available: https://ieeexplore.ieee.org/document/6241646.
12. Xiaoming Chen, Yu Wang, and Huazhong Yang, "An Adaptive LU Factorization Algorithm for Parallel Circuit Simulation", IEEE XPlore Digital Library. Available: https://ieeexplore.ieee.org/document/6164974.
13. Xiaoming Chen, Du Su, Yu Wang, and Huazhong Yang, "Nonzero Pattern Analysis and Memory Access Optimization in GPU-based Sparse LU Factorization for Circuit Simulation", CiteSeerX. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.719.1828.
14. Timothy Davis and E. P. Natarajan, "Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems", ACM Transactions on Mathematical Software, Vol. 37, No.3, Article 36, Sept 2010. Available: doi>10.1145/1824801.1824814.
15. Lawrence Murray, "GPU Acceleration of Runge-Kutta Integrators", IEEE Transactions on Parallel and Distributed Systems, Vol. 23, No. 1, Jan 2012. Available: ACM Digital Library, doi>10.1109/TPDS.2011.61.
16. Piyush Sao, Richard Vuduc, and Xiaoye Li, "A distributed CPU GPU sparse direct solver", Berkeley Lab. Available: https://crd-legacy.lbl.gov/~xiaoye/europar14.pdf.
17. H. M. D. M. Bandara and D. N. Ranasinghe, "Effective GPU Strategies for LU Decomposition", IEEE International Conference on High Performance Computing, HiPC. Available: https://hipc.org/hipc2011/studsym-papers/1569512927.pdf
18. Tingxing Dong, Azzam Haidar, Piotr Luszczek, James Austin Harris, Stanimire Tomov, and Jack Dongarra, "LU Factorization of Small Matrices: Accelerating Batched DGETRF on the GPU", ACMDL Digital Library. Available: https://dl.acm.org/citation.cfm?id=2760899
19. Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha, "LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware", ResearchGate. Available: https://www.researchgate.net/publication/220782520_LU-GPU_Efficient_Algorithms_for_Solving_Dense_Linear_Systems_on_Graphics_Hardware
20. Francesco Lannnutti, Paolo Nanzi, and Mauro Olivieri, "KLU Sparse Direct Linear Solver Implementation into NGSPICE", 19th International Conference on Mixed Design of Integrated Circuits and Systems, May 24-26, 2012, Poland.
21. Shaoyi Peng, Sheldon X.-D Tan,"GLU3.0:Fast GPU-based Parallel Sparse LU Factorization for Circuit Simulation", arXiv:1908.00204v2[cs.DC], 2019. https://arxiv.org/pdf/1908.00204.pdf

AUTHORS PROFILE

Shital, holds Masters degree in computer engineering and pursuing her PhD from Mumbai University. She has been in the teaching profession for over 19 years, of which she has been associated with Vishwakarma Institute of technology, Pune for more than 13 years. Her research interest is High performance computing especially in parallel processing using CUDA on GPU. She has accelerated processing of many applications using CUDA programming.

Dr. Y. Srinivasa Rao, holds a PhD degree from IIT-Bombay and has a work experience of more than 10 years in the industry. His research interest is in VLSI design and power electronics. He has been in the teaching profession for over 25 years, of which he has been associated with Bhartiya Vidya Bhavans' Sardar Patel College of Engineering and subsequently, Sardar Patel Institute of Technology for more than 17 years.

He has served as the Head of Department several times and is currently the Principal as well as the Dean of R&D at Sardar Patel Institute of Technology. He is also a guest faculty for various institutes. Dr. Y. S. Rao has been an expert lecturer at Short Term Teachers Training Program (STTP) of ISTE and TEQIP programs of AICTE.