Michigan Technological University

Digital Commons @ Michigan Tech

Dissertations, Master's Theses and Master's Reports

2016

SCALABLE INTEGRATED CIRCUIT SIMULATION ALGORITHMS FOR ENERGY-EFFICIENT TERAFLOP HETEROGENEOUS PARALLEL COMPUTING PLATFORMS

Lengfei Han
Michigan Technological University, lengfeih@mtu.edu

Copyright 2016 Lengfei Han

Recommended Citation
Han, Lengfei, "SCALABLE INTEGRATED CIRCUIT SIMULATION ALGORITHMS FOR ENERGY-EFFICIENT TERAFLOP HETEROGENEOUS PARALLEL COMPUTING PLATFORMS", Open Access Dissertation, Michigan Technological University, 2016.
https://doi.org/10.37099/mtu.dc.etdr/86

Follow this and additional works at: https://digitalcommons.mtu.edu/etdr

Part of the Electronic Devices and Semiconductor Manufacturing Commons

SCALABLE INTEGRATED CIRCUIT SIMULATION ALGORITHMS FOR ENERGY-EFFICIENT TERAFLOP HETEROGENEOUS PARALLEL COMPUTING PLATFORMS

By

Lengfei Han

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

In Computer Engineering

MICHIGAN TECHNOLOGICAL UNIVERSITY

2016

© 2016 Lengfei Han

This dissertation has been approved in partial fulfillment of the requirements for the Degree of DOCTOR OF PHILOSOPHY in Computer Engineering.

Department of Electrical and Computer Engineering

Dissertation Advisor: *Dr. Zhuo Feng*

Committee Member: *Dr. Kuilin Zhang*

Committee Member: *Dr. Zhenlin Wang*

Committee Member: *Dr. Zhengfu Xu*

Department Chair: *Dr. Daniel R. Fuhrmann*

Dedication

To my parents, wife and son

Contents

List of Figures . . . . **xi**

List of Tables . . . . **xiii**

Preface . . . . **xv**

Acknowledgments . . . . **xvii**

Abstract . . . . **xix**

1 Introduction **1**

1.1 Post-layout RF Circuits Harmonic Balance Analysis 3

1.2 Reliability and Yield Analysis of Small Circuit 5

1.3 Overview of Chapters 7

2 Scalable Harmonic Balance Analysis of Post-Layout RF Circuits Leveraging Heterogeneous Platform **9**

2.1 Background and Overview 9

2.1.1 Review of Harmonic Balance Analysis 10

2.1.2 Graph-based Preconditioning Approaches 13

2.1.2.1 Graph Sparsification Problems 13

2.1.2.2 Ultra-sparsifier Support Graph Preconditioners . . 14

2.1.3 Overview of Proposed Support-Circuit Preconditioning Approach 16

2.2 Support-circuit Preconditioner for HB Analysis 18

2.2.1 Sparsification of Representative Laplacian Matrices 19
2.2.1.1 Extraction of Representative Laplacian Matrices . 19
2.2.1.2 Sparsification of Representative Laplacian Matrices 21
2.2.2 Sparsification Pattern Extraction 22
2.2.3 HB Jacobian Preconditioner Construction 24
2.2.4 Case Study: Double-balanced Gilbert Mixer Sparsification . 25
2.3 Parallel Block Sparse Matrix Direct Solver 27
2.3.1 LU Data Dependency Analysis 28
2.3.2 "Fake" Dependencies in LU Factorization 29
2.3.3 Parallel LU Task Scheduling 31
2.3.4 Test Matrix Factorization 31
2.3.5 The Sparse Block LU Algorithm 32
2.4 Transient Analysis Guided Sparsification 33
2.4.1 HB Simulation Runtime Profiling 34
2.4.2 Runtime Performance Modeling 36
2.4.2.1 CPU Only Platform Performance Model 38
2.4.2.2 CPU-GPU Platform Performance Model 38
2.4.3 Nearly-optimal Sparsification Scheme 40
2.5 The Scalable HB Analysis Algorithm 42
2.6 Experiment Result 45
2.6.1 Experimental Setup 45
2.6.2 Experimental Results 46
2.6.3 Scalability 49

3 Massively Repeated Small Circuit Simulation on GPU 51
3.1 Background and Overview 51
3.1.1 Nonlinear Circuit Simulation Approaches 51
3.1.2 Massively Parallel GPU Computing 52
3.1.3 Overview of our approach 53

3.2 Device Evaluation and Stamping on GPU 55
3.2.1 Device Evaluation on GPU 55
3.2.2 Jacobian Matrix Data Format and Stamping on GPU 58
3.2.2.1 Dense Jacobian Matrix and Stamping 58
3.2.2.2 GPU Sparse Jacobian Matrix and Stamping 59
3.2.3 RHS and Excitation Sources 61
3.3 Matrix Solver on GPU 62
3.3.1 GPU-based Levelized LU Factorization 63
3.3.2 Circuits Clustering 64
3.4 GPU Optimization 66
3.4.1 Data Allocation and Access Optimization 66
3.4.2 Thread Organization 67
3.4.3 Jacobian Matrix Format Determination 69
3.5 Algorithm Flow for TinySPICE 69
3.5.1 CPU and GPU Cooperation 69
3.5.1.1 CPU Setup Phase 70
3.5.1.2 GPU setup and analysis phase 71
3.5.2 NR Iteration algorithm on GPU 71
3.5.3 DC Simulation Flow 72
3.5.4 Transient Simulation Flow 73
3.6 Experiment Result 75
3.6.1 Experimental Setup 75
3.6.2 Experimental Results 76
3.6.2.1 Accuracy of Parametric 3D LUT 76
3.6.2.2 Runtime Results 79

4 Conclusion and Future Work 83
4.1 Conclusion of the dissertation 83
4.2 Future Work 84

References **87**

A Letters of Permission **95**

A.1 Permission Letters for Chapter 2 95

A.2 Permission Letter for Chapter 3 98

List of Figures

2.1 From sparsification of MNA matrix to sparsification of HB Jacobian matrix problems . . . 16

2.2 Circuit MNA matrices decomposed into Laplacian and complement matrices . . . 20

2.3 MNA matrix sparsification pattern . . . 24

2.4 HB Jacobian matrix construction . . . 25

2.5 An RF mixer design (left) and the circuit network (right) corresponding to its Laplacian matrix . . . 26

2.6 MOSFET model . . . 27

2.7 Sparsification of the Laplacian graph of an RF mixer circuit . . . 27

2.8 Upper matrix factor and its DDG . . . 30

2.9 Lower matrix factor and its improved DDG . . . 30

2.10 Runtime profiling for solving two sparsified graphs . . . 35

2.11 HB simulation runtime with different sparsity . . . 35

2.12 TR simulation runtime vs. HB simulation runtime . . . 38

2.13 Performance-guided sparsification scheme . . . 41

2.14 HB analysis runtime vs. the input power for an RF mixer circuit . . . 47

2.15 Convergence rate/time comparisons of SCPHB and ILU algorithms 48

2.16 Waveform result comparison between direct solution method and proposed iterative method . . . 49

2.17 Results on runtime scalability . . . 50

2.18 Results on memory scalability . . . 50

3.1 TinySPICE: massively parallel SPICE simulation program on GPUs 54
3.2 Vector for storing 3D LUTs. 57
3.3 MOSFET stamping location map for dense and sparse matrix format. 61
3.4 Vectors for storing excitation sources on GPU. 62
3.5 Levelized LU factorization task list. 63
3.6 GPU data structure for the LU factorization task list. 64
3.7 Circuit clustering. 65
3.8 The solution vector data access pattern on GPU. 67
3.9 The algorithm flow of TinySPICE. 68
3.10 The DC simulation flow of TinySPICE. 73
3.11 TR simulation algorithm flow. 74
3.12 The I-V characteristics obtained by parametric 3D LUT and Bsim4 model evaluations. Circles denote the LUT evaluation results. . . . 77
3.13 Scatter plot of the DC simulation results for SRAM circuits obtained by TinySPICE and the original Bsim4 SPICE simulator. 78
3.14 Comparison of DC Simulation Runtime 79
3.15 Comparison of Transient Simulation Runtime 80
3.16 Memory usage (shared memory + registers) vs. speedups 81

List of Tables

2.1 Experimental circuit descriptions 44

2.2 Results of runtime and memory cost. 45

3.1 Experimental setup of test cases. 76

3.2 DC Simulation Runtime Results of TinySPICE with Dense Format 78

3.3 Transient Simulation Runtime Results of TinySPICE with Dense Format 78

3.4 DC simulation runtime results of TinySPICE with sparse format. . 81

3.5 Transient Simulation Runtime results of TinySPICE with sparse format 81

Preface

This dissertation presents my research work in pursuing the PhD degree in Computer Engineering at Michigan Technological University. This dissertation includes previously published articles in Chapter 2 and Chapter 3. All the research works descried herein were conducted under the supervision of my advisor Professor Zhuo Feng.

Chapter 2 contains two articles published in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* and *52nd Design Automation Conference*. As the first author of both papers, with the guidance of my advisor, I completed most parts of algorithm design, implementation, and analysis. Xueqian Zhao, as the second author of the paper published in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, implemented the ultra-sparsifier generation algorithm. The two papers themselves were completed by me and my advisor.

Part of Chapter 3 was published in *50th Design Automation Conference*. As the first author, with the guidance of my advisor, I proposed the algorithm and completed its implementation. Xueqian Zhao, as the second author, provided invaluable revision advice of the paper. This article was completed by me and my advisor.

Acknowledgments

First and foremost, I would like to thank my advisor Professor Zhuo Feng for his support and guidance throughout my stay at Michigan Tech. Professor Feng introduced me to most of the topics presented in this dissertation. Without his continued support, skillful guidance, and insightful advice, most ideas presented in this dissertation would have not materialized. I am also grateful for his encouragement and patience through the ups and downs of my PhD research years.

I would like to thank all my dissertation committee members, Professor Zhenlin Wang, Professor Zhengfu Xu and Professor Kuilin Zhang for their time and invaluable advice during the dissertation process. Thanks to Professor Shiyan Hu for serving on my qualifying examination and proposal committee.

I also want to thank my friends at Michigan Tech, Xueqian Zhao, Caoyang Jiang, Jia Wang, Bojun Ma, Yonghe Guo, Liang Ma, Lin Liu and Yuchen Zhou for their help and supports. Countless discussions with Xueqian Zhao on graph sparsification technology were very helpful to insight me how to apply this new technology to the harmonic balance analysis projects. Caoyang Jiang is always energetic and passionate on discussing the GPU programming techniques.

Finally, and most importantly, I would like to thank my family. Without their unconditioned love and support, it is impossible for me to get this far. My parents are always standing behind me with unending encouragement and support. My wife, Hao Meng, spent lots of time taking care of our family, even when she had her own research work to do. I give my greatest gratitude to her.

Abstract

Integrated circuit technology has gone through several decades of aggressive scaling. It is increasingly challenging to analyze growing design complexity. Post-layout SPICE simulation can be computationally prohibitive due to the huge amount of parasitic elements, which can easily boost the computation and memory cost. As the decrease in device size, the circuits become more vulnerable to process variations. Designers need to statistically simulate the probability that a circuit does not meet the performance metric, which requires millions times of simulations to capture rare failure events.

Recent, multiprocessors with heterogeneous architecture have emerged as mainstream computing platforms. The heterogeneous computing platform can achieve high-throughput energy efficient computing. However, the application of such platform is not trivial and needs to reinvent existing algorithms to fully utilize the computing resources. This dissertation presents several new algorithms to address those aforementioned two significant and challenging issues on the heterogeneous platform.

Harmonic Balance (HB) analysis is essential for efficient verification of large post-layout RF and microwave integrated circuits (ICs). However, existing methods either suffer from excessively long simulation time and prohibitively large memory consumption or exhibit poor stability. This dissertation introduces a novel transient-simulation guided graph sparsification technique, as well as an efficient runtime performance modeling approach tailored for heterogeneous manycore CPU-GPU computing system to build nearly-optimal subgraph preconditioners that can lead to minimum HB simulation runtime. Additionally, we propose a novel heterogeneous parallel sparse block matrix algorithm by taking advantages of the structure of HB Jacobian matrices as well as GPU's streaming multiprocessors to achieve optimal workload balancing

during the preconditioning phase of HB analysis. We also show how the proposed preconditioned iterative algorithm can efficiently adapt to heterogeneous computing systems with different CPU and GPU computing capabilities. Extensive experimental results show that our HB solver can achieve up to 20X speedups and 5X memory reduction when compared with the state-of-the-art direct solver highly optimized for twelve-core CPUs.

In nowadays variation-aware IC designs, cell characterizations and SRAM memory yield analysis require many thousands or even millions of repeated SPICE simulations for relatively small nonlinear circuits. In this dissertation, for the first time, we present a massively parallel SPICE simulator on GPU, TinySPICE, for efficiently analyzing small nonlinear circuits. TinySPICE integrates a highly-optimized shared-memory based matrix solver and fast parametric three-dimensional (3D) LUTs based device evaluation method. A novel circuit clustering method is also proposed to improve the stability and efficiency of the matrix solver. Compared with CPU-based SPICE simulator, TinySPICE achieves up to $264X$ speedups for parametric SRAM yield analysis without loss of accuracy.

Chapter 1

Introduction

As relentless technology scaling reaches into the sub-16nm regime, integrated circuit (IC) designers are facing phenomenal growth of design complexity: present-day multicore/manycore microprocessors integrate billions of transistors into a single chip, while emerging three-dimensional ICs (3D-ICs)[1, 2] integrate multiple active layers in the vertical direction. Key VLSI subsystems such as embedded memory arrays and analog and mixed-signal systems may reach an unprecedented complexity of hundreds of millions of circuit components (nodes), making their modeling, analysis and verification tasks prohibitively expensive and even intractable. It is not rare to experience analog and RF circuit simulations that take a few days or weeks to finish.

Although there has been tremendous evolution in shifting traditional sequential Electronic Design Automation (EDA) tools into their parallel implementations for modern multicore computers in the past decade [3, 4, 5, 6, 7, 8, 9, 10, 11], the future multicore computing will apparently be hindered by the dramatically-increased chip power consumption and slowly-improved heat sinking capability. As a result, present-day computer architects and research community are forced to seek alternative paradigms

to sustain ever-increasing performance. The industry realized the only viable solution was to replace some of the large yet power-inefficient general purpose processors by as many as possible slimmers but much more energy-efficient co-processors on the same chip[12], building so-called heterogeneous computing platforms. Recent multiprocessors with such heterogeneous architectures have emerged as mainstream computing platforms, which typically integrate a variety of processing elements of different computing performance, programming flexibility and energy efficiency characteristics. Heterogeneous computing platforms, such as IBM/Sony Cell architectures, personal computers (PCs) with multicore CPUs and manycore GPUs, and the latest low-power heterogeneous microprocessors (e.g. APU from AMD[13], Larrabee from Intel[14], Tegra from Nvidia[15]), can theoretically achieve unprecedented high performance and high energy efficiency simultaneously. With such heterogeneous computing architectures, VLSI CAD developers will face tremendous opportunities to revolutionize EDA industry, thereby targeting much greater performance and energy efficiency.

The goal of this work is to investigate and develop scalable IC modeling, simulation, and verification methods for emerging heterogeneous parallel architectures by reinventing CAD algorithms/data structures and exploiting powerful hardware-specific computing performance/energy modeling and optimization approaches. A coherent set of VLSI CAD problems will be targeted and investigated in this dissertation as followings:

† Scalable post-layout RF circuits harmonic balance analysis

† Reliability and yield analysis of small circuit

† Hardware-specific performance modeling for heterogeneous computing architectures

1.1 Post-layout RF Circuits Harmonic Balance Analysis

The rapid growth of demand for high-performance wireless systems has increased the need for more efficient, accurate, and robust simulation method for RF circuits. Harmonic balance method is the typical choice for steady state analysis, which can captures the spectral response directly. Traditional HB methods require solving very large yet non-sparse Jacobian matrices, which can take excessively long simulation time and consume a large amount of memory resources when using direct solution methods [16]. As a result, some of existing industrial HB simulators separate the nonlinear and linear parts of the circuit such that the computational cost can be effectively reduced. Unfortunately, such splitting methods assume that the coupling effects between the linear and nonlinear circuit components are relatively weak, and therefore may not be suitable for dealing with large post-layout RF and microwave circuits that involve a lot of parasitics.

To achieve greater computing efficiency than the traditional direct solution methods, several preconditioned Krylov-subspace iterative methods have been investigated and developed in recent years [16, 17, 18, 19]. However, developing high-quality preconditioners for HB analysis has been a very challenging task, since the convergence property of the preconditioned iterative methods for HB analysis strongly depends on the effectiveness of the underlying preconditioners, especially when using the Krylov-subspace iterative methods, such as the GMRES algorithm [20]. For instance, although prior preconditioning methods have shown promising results for trading off the computational efficiency and preconditioning effectiveness, they can be inevitably facing with a variety of limitations and difficulties when handling large and strongly

nonlinear post-layout RF and microwave circuits: the block-diagonal averaging preconditioners are easy to compute but only limited to handle weakly nonlinear systems [21]; the hierarchical HB preconditioner proposed in [18] is more effective than the block-diagonal preconditioner and also suitable for parallel computing, but can lead to poor performance or divergence when handling strongly-nonlinear RF ICs since the frequency domain decomposition scheme will introduce large errors during the preconditioning step; another finite-difference Jacobian preconditioner can easily deal with strongly nonlinear systems, but will not work for more than one tones in HB analysis, as discussed in [16]; in a most recent work [16], a sparse block direct solver is developed for solving the Jacobian matrices of HB, but it will consume much more computational resources than iterative methods and cannot scale well with large RF circuit designs. As a result, there is not a preconditioning method that can work robustly for a wide variety of RF and microwave circuits analysis problems, and at the same time be computationally efficient (scalable to large problems sizes). Consequently, it is very desirable to develop efficient yet robust solvers to facilitate fast HB analysis for addressing the challenges in future large-scale RF and microwave IC design and verification procedures.

In this dissertation, recent graph sparsification and support-circuit preconditioning techniques [22, 23, 24, 25] are exploited for developing scalable Jacobian matrix solvers on Heterogeneous platform that can tackle large-scale strongly nonlinear post-layout HB analysis problems. Our approach starts with sparsifying the HB Jacobian matrix with the performance guided sparsification model. We show that the resultant sparsified Jacobian matrix can be used as a robust yet efficient preconditioner in HB analysis. Subsequently, the proposed parallel sparse block solver can solve the preconditioned system rapidly to accelerate the preconditioned Krylov-subspace iterative solving process. Our experimental results show that when compared with the state-of-the-art direct solution method [16], the proposed HB solver can more efficiently handle moderate to strong nonlinearities during the HB analysis of large

RF circuits, achieving up to $20X$ speedups and $5X$ memory reductions. The main technical contributions of this work have been summarized as follows.

1. Proposed a circuit-oriented support-circuit preconditioning approach that can scale almost linearly with large-scale strongly nonlinear post-layout RF circuit.
2. Proposed a GPU-friendly sparse block matrix solver for fast solving the preconditioner matrix.
3. Proposed a transient-analysis guided hardware-specific graph sparsification scheme to help automatically compute nearly-optimal preconditioners.

1.2 Reliability and Yield Analysis of Small Circuit

Reliability and yield analysis of embedded SRAM memory modules are critical to designs of modern microprocessors, 3D-ICs, and mixed-signal SOCs. However, nanoscale SRAM designs are significantly challenged by prohibitively high computation cost due to the extremely large number of repeated SPICE simulations considering parametric variations [26, 27, 28, 29, 30]. Additionally, current variation-aware design methodologies require extremely fast cell/driver characterization capability capturing important process, voltage supply, and temperature (PVT) variations [31, 32], which also demands for much more powerful simulation methodologies. For instance, SRAM readability, writability and stability analysis considering threshold voltage (V_{th}), effective channel length (L_{eff}) and power supply variations require tens of millions of repeated SPICE simulations for a given design, while variation-aware cell modeling and characterizations also involve constructing look-up tables (LUTs) for capturing all fast/slow corners that require running many thousands of SPICE simulations [31, 32, 33].

Although there have been works that target accelerating SPICE simulations by performing device evaluations on GPU's hundreds of streaming processors and sparse matrix solves on CPU [34], only a small fraction of the computations can be accelerated on GPU, while the overall simulation performance is still limited by the relatively low communication bandwidth and large latency between the CPU and GPU. As a result, only $2X$ speedups have been obtained when compared with the CPU-based SPICE simulator [34]. Since sparse matrices derived from general nonlinear circuits are typically large scale and asymmetric, no sparse matrix algorithm have been efficiently accelerated on GPU due to a large amount of memory accesses and complicated algorithm flow. Consequently, accelerating the entire computations involved in general-purpose SPICE simulations on GPU remains impractical considering present-day GPU computing limitations. However, there is still a strong need to consider accelerating application-specific SPICE simulations on GPU for achieving much higher computing performance.

In this work, we present a massively parallel SPICE simulator on GPU, TinySPICE, which accelerates the entire SPICE simulation computations on GPU without introducing excessive CPU-GPU data communications and device memory accesses. TinySPICE can analyze small nonlinear circuits in GPU's shared memory and thus gains unprecedentedly high computational throughput. The proposed series of highly-optimized shared-memory based sparse matrix construction and solution techniques allow TinySPICE be able to handle much larger circuits while still being able to achieve orders of magnitude speedup over traditional CPU-based SPICE-like simulation engines. We develop novel GPU-friendly data structures and efficient algorithm flow for every kernel function of the SPICE algorithm that includes device evaluations, matrix construction, linear system solving and Newton-Raphson (NR) iterations. TinySPICE is capable of solving thousands of small circuit simulation problems in GPU's shared memory concurrently, and achieves unprecedented high-performance massively parallel SPICE simulations on GPU. Compared with CPU-based SPICE

simulators, TinySPICE achieves up to $264X$ speedups for a variety of circuit analysis problems without loss of accuracy. Key contributions of this work have been summarized as follows:

1. We propose a massively parallel SPICE simulation engine which is able to perform DC and TR analysis entirely on GPU.

2. We propose a series of shared-memory based matrix storage format and matrix solution method to guarantee the simulator can utilize the GPU hardware resources in the most efficient way for different size of circuit designs.

3. We propose a novel circuits clustering/classification procedure that will allow to simulate circuits with similar statistical properties (performance) on the same streaming multiprocessor (SM) of each GPU, which can effectively minimize the rounding errors and GPU thread divergences during the sparse matrix factorization procedure.

4. We also present a series practical techniques for optimizing GPU's memory usage considering GPU-specific data structures and access patterns to achieve optimal computing throughput.

1.3 Overview of Chapters

This dissertation consists of four chapters. Chapter 1 presents the introduction of the two major problems addressed in this dissertation and the summary of our contributions. Chapter 2 presents the new harmonic balance solver which integrates a novel circuit-oriented preconditioner and parallel sparse block matrix solver. In Chapter 3, the proposed new massively parallel small circuit simulator on GPU is described in

details. The dissertation concludes with Chapter 4, which summarizes the work and discuss directions for future research.

Chapter 2

Scalable Harmonic Balance Analysis of Post-Layout RF Circuits Leveraging Heterogeneous Platform[1]

2.1 Background and Overview

We first review the basics of harmonic balance (HB) method for steady-state simulations of RF circuits. Then, we provide a brief introduction to graph sparsification theory and its applications in developing scalable preconditioned iterative matrix solvers.

[1] The material contained in this chapter was previously published in "*IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*" ©2015 IEEE and "*Proceedings of ACM/IEEE Design Automation Conference (DAC)*" ©2015 IEEE. See Appendix A.1 for copies of the copyright permission from IEEE.

2.1.1 Review of Harmonic Balance Analysis

Compared to time-domain analysis that can be obtained by performing transient (TR) circuit simulations, steady-state simulations of RF and microwave circuits typically require HB analysis [16, 17, 18, 19, 21, 35] that can naturally handle frequency-domain data such as S-parameters of linear networks. The basic theory of HB method is introduced as follows. Consider a non-autonomous circuit analysis problem described by the following equation:

$$\int_{-\infty}^{t} y\left(t-s\right)x\left(s\right)ds + \frac{dq\left(x\left(t\right)\right)}{dt} + f\left(x\left(t\right)\right) + b\left(t\right) = 0, \tag{2.1}$$

where $x\left(t\right) \in \Re^n$ represents a set of state variables, n is the number of unknowns, y is the matrix-valued impulse response function of frequency-domain linear circuit components (such as S-parameter models), $q(\bullet)$ denotes a function for the nonlinear charge and flux, $f(\bullet)$ represents the static (memoryless) nonlinearities, and b represents the time-dependent excitations that are assumed to be periodic with a time period T. The circuit steady-state response $x(t)$, and functions $q(\bullet)$ as well as $f(\bullet)$ will be periodic with period T. By writing the above equation in frequency domain and applying Newton-Raphson (NR) method, it can be shown that the linearized system in frequency domain becomes:

$$YX + \Omega\Gamma C\Gamma^{-1}X + \Gamma G\Gamma^{-1}X - B = 0, \tag{2.2}$$

where X and B denote the Fourier-coefficient vector of $x(t)$ and $b(t)$ respectively, Ω is a diagonal matrix denoting the frequency domain differentiation operator, Γ and Γ^{-1} are the fast Fourier transform(FFT) and inverse FFT(IFFT) matrices, while C and G are block diagonal matrices with block diagonals representing the linearizations of $q(\bullet)$ and $f(\bullet)$ at h time-domain sampled points that can be described as follows for

$i = 1, ..., h$, respectively:

$$C = diag\left\{c_i = \delta q/\delta x|_{x=x(t_i)}\right\}; \quad (2.3)$$

$$G = diag\left\{g_i = \delta f/\delta x|_{x=x(t_i)}\right\}. \quad (2.4)$$

When the double-sided FFT/IFFT are used, a total number of $h = 2k + 1$ harmonics are included to represent each signal, where k is the number of positive frequencies being considered.

In each NR step, a linearized system is solved with a Jacobian matrix of (2.2):

$$J_{hb} = Y + \Omega\Gamma C\Gamma^{-1} + \Gamma G\Gamma^{-1}. \quad (2.5)$$

The most time-consuming step in HB analysis is the one for solving the large yet non-sparse Jacobian matrix J_{hb} shown in (2.5). It can be shown that the dense blocks in J_{hb} are mainly due to the block-circulant matrices $\Gamma C\Gamma^{-1}$ and $\Gamma G\Gamma^{-1}$ [21]. For instance, the block-circulant matrix $\Gamma G\Gamma^{-1}$ can be expressed as:

$$\Gamma \begin{bmatrix} g_1 & & \\ & \ddots & \\ & & g_h \end{bmatrix} \Gamma^{-1} = \begin{bmatrix} G_1 & G_2 & \cdots & G_h \\ G_h & G_1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & G_2 \\ G_2 & \cdots & G_h & G_1 \end{bmatrix} = circulant(G_1, \cdots, G_h). \quad (2.6)$$

As a result, directly solving such a Jacobian matrix using LU-based direct solution method can be very runtime and memory costly due to the very dense matrix structure and a large number of fill-ins introduced during the factorization procedure. Consider a recent state-of-the-art HB simulator developed in [16]. It has been shown that HB analysis of a post-layout RF circuit (LNA+mixer+filter) with $44K$ nodes and 20

harmonics will result in a Jacobian matrix with more than 1.8 million unknowns that would require around 100 hours and more than $15GB$ memory when using LU-based direct solution method. On the other hand, the runtime and memory costs for another smaller test case (with half problem size) are $10X$ less than the previous case, indicating a rather poor algorithm scalability.

To avoid the direct factorization of large and dense Jacobian matrices in HB analysis, iterative methods can be applied to dramatically improve the computing efficiency. Krylov-subspace iterative methods, such as GMRES method [20], are particularly suitable for such problems since only the matrix-vector operations are needed throughout the solution procedures. It has been shown that for HB analysis, the matrix-vector product:

$$JX = YX + \Omega \Gamma C \Gamma^{-1} X + \Gamma G \Gamma^{-1} X \tag{2.7}$$

can be computed very efficiently, without explicitly forming the real Jacobian matrix.

Unfortunately, iterative methods, such as Krylov-subspace iterative methods in particular, may suffer from slow convergence or even divergence issues unless robust preconditioners are adopted. However, finding efficient yet robust preconditioners for tackling general HB simulation tasks remains an open problem. A good introduction of existing preconditioning methods for HB analysis of RF circuits can be found in [16].

2.1.2 Graph-based Preconditioning Approaches

Recently, a series of support-graph based preconditioning techniques has been introduced to solve circuit simulation problems. For instance, a support-graph preconditioned solver was presented in [22] for solving linear circuit networks, and support-circuit preconditioning techniques for nonlinear transistor-level circuit simulations were proposed in [23, 24], which have been shown to achieve nearly-linear runtime and memory efficiency. In this section, the related background and techniques will be described in details.

2.1.2.1 Graph Sparsification Problems

General linear circuit analysis problems can be converted into equivalent graph problems [36]. For instance, a linear resistive network can be represented by a weighted, undirected graph $G = (V, E, w)$, where V is a set of vertices, E is a set of edges, and w is a weight function that assigns a positive weight to every edge. The Laplacian matrix A of a weighted graph is defined as follows:

$$A(s,d) = \begin{cases} -w(s,d) & \text{if } (s,d) \in E \\ sum(s) & \text{if } (s = d) \\ 0 & \text{if } otherwise \end{cases} \tag{2.8}$$

where $sum(s) = \sum_{(s,v)\in E} w(s,v)$ denotes the sum of the incident weights of vertex s. From (2.8), we can observe that Laplacian matrix is a symmetric matrix with non-positive off-diagonals and zero row sums, which can be considered as an admittance matrix in circuit theory. For a vector $x \in \Re^V$, the Laplacian quadratic form of G is

defined to be:

$$x^T Ax = \sum_{(s,d)\in E} w_{s,d}(x(s) - x(d))^2. \tag{2.9}$$

It can be seen that Laplacian matrix A provides a measure of the smoothness of x over the edges in G [37], since the more x changes over an edge, the larger the quadratic form becomes.

Graph sparsification is a very important technique that has been playing significant roles in designing nowadays efficient graph algorithms [37, 38, 39, 40, 41]. Given a graph $G = (V, E)$, a graph sparsifier (a.k.a support graph) G' is a sparse subgraph of G that can approximate G in some measures such as the pairwise distance, cut values or the graph Laplacian. The goal of graph sparsification is to approximate a given graph G by G' on the same set of vertices such that G' can be used as a proxy for G in numerical computations without introducing too much error. A good sparsifier should have very few edges that will immediately result in significantly reduced computation and storage cost. More details of this technique can be found in recent research papers [39, 40, 41].

2.1.2.2 Ultra-sparsifier Support Graph Preconditioners

Support-graph preconditioning is to first construct a graph G according to a given graph Laplacian matrix A, and then extract the support graph G' of G that can be further used to build a preconditioner P for iterative solvers such as conjugate gradient (CG) or GMRES solvers. In practice, for a given graph a maximum or low-stretch spanning tree can be constructed and used as its support graph, which has been proposed in the past for solving linear systems with symmetric and diagonally-dominant (SDD) matrices in nearly-linear time [37, 42, 43, 44]. Support-graph preconditioning seeks to compute the preconditioner P such that the generalized eigenvalues and the

condition number of the matrix pencil (A, P) are bounded [45]. If both A and P are symmetric positive definite (SPD) matrices, the convergence of classic Krylov-subspace iterative methods depends on the condition number $\kappa(A, P)$ computed by:

$$\kappa(A, P) = \lambda_{max}(A, P)/\lambda_{min}(A, P), \tag{2.10}$$

where $\lambda(A, P)$ denotes the generalized eigenvalues. A stronger theoretical result on convergence can be derived as follows. Define the support of (A, P), denoted by $\sigma(A, P)$, as follows [42, 45]:

$$\sigma(A, P) = min\{\tau \in \mathbb{R} | x^T(\tau P - A)x \geq 0 \ for \ all \ x \in \mathbb{R}^n\}. \tag{2.11}$$

Subsequently, if one can split A and P into $A = A_1 + A_2 + ... + A_m$ and $P = P_1 + P_2 + ... + P_m$ such that all $\tau P_i - A_i$ are SPD matrices, one can show that the generalized eigenvalue of (A, P) is bounded by τ.

Compared to the original graph, the support graph has fewer edges. Since the spanning tree of a graph that includes n vertices and m $(m \geq n)$ edges retains only $n - 1$ edges, the power dissipated on the support graph is much smaller than the one on the original system. If a support-graph preconditioner preserves not only the eigenvalues but also the power dissipation of the original system, it can be more effective than the previous spanning-tree support graph preconditioner [38] in reducing the number of iterations when using Krylov-subspace iterative methods. Consequently, a much better support graph can be formed by selectively adding extra links to the spanning-tree support graph, which is also known as the ultra-sparsifier support graph[38]. When using ultra-sparsifier support graphs as preconditioners for iterative solvers, it is important to trade off the effectiveness and efficiency by carefully choosing the extra edges to be added to the spanning tree.

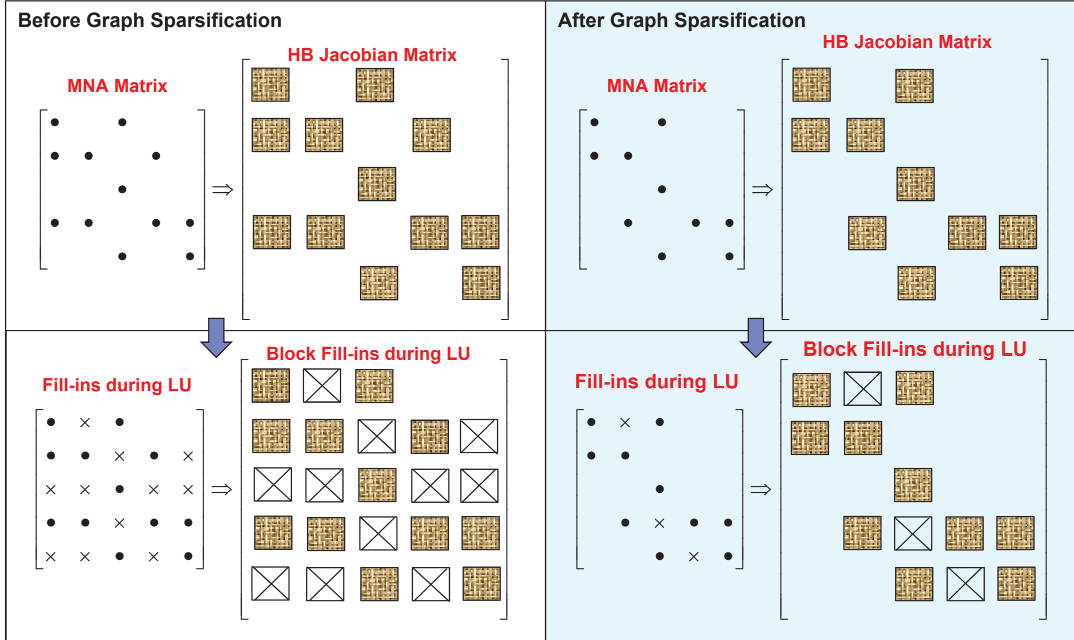


Figure 2.1: From sparsification of MNA matrix to sparsification of HB Jacobian matrix problems

2.1.3 Overview of Proposed Support-Circuit Preconditioning Approach

In this work, the graph sparsification and support graph theories will be exploited to develop scalable Jacobian matrix solvers for strongly nonlinear post-layout HB analysis. Although direct solution methods for solving the HB Jacobian matrix J_{hb} can handle strongly nonlinear problems, the fast-growing cost for solving the J_{hb} matrix in HB analysis due to the block matrix fill-ins during the block LU factorization process will make such methods computationally prohibitive [16], as shown in Fig. 2.1. In our approach, the original RF circuit is first sparsified into a support circuit (a sparsified circuit that can well approximate the original circuit) graph that has much

fewer edges and maintains a tree-like structure, such that the number of fill-ins during LU factorization procedures can be dramatically reduced. Such a *support circuit*[2] can be subsequently used as a preconditioner to facilitate fast HB analysis, thereby significantly improving the runtime and memory efficiency. Although the proposed support-circuit preconditioning process will also introduce some block fill-ins during LU factorization, the number of new blocks will increase almost linearly with the problem sizes owing to the tree-like circuit structure, while the original circuit topology will typically result in exponentially increased block fill-ins. As a result, the proposed support-circuit preconditioning approach will allow to analyze much larger RF and microwave circuits than ever before, and still maintain a decent convergence rate during Krylov-subspace iterations in the presence of moderate to strong nonlinearities.

In this dissertation, we present a novel graph sparsification approach for generating preconditioners that can effectively and efficiently facilitate HB simulations of strongly nonlinear post-layout RF circuits. In the proposed method, the system MNA matrix of each sampled time point that can be obtained from the g_i and c_i matrices in (2.3) and (2.4), will be first decomposed into a *Laplacian* (A_L) *matrix* (with stamped equivalent resistors and capacitors) and a *complement* (A_C) *matrix* (with other stamped components, such as inductors, transconductances and voltage sources, etc), as shown in Fig. 2.2. Throughout this work, we define the network derived from the Laplacian matrix as the *Laplacian network*, and the network derived from the complement matrix as the *complement network*. Subsequently, a *representative Laplacian matrix* is obtained by scaling and averaging all the sampled Laplacian matrices, which can be subsequently sparsified into a *sparsified representative Laplacian matrix* by constructing an ultra-sparsifier support graph based on its Laplacian network. In the next step, the sparsification pattern of the system MNA matrices can be obtained

[2]Support-circuit preconditioner is first introduced in [23] and has been extended to iteratively solve general SPICE-accurate simulation problems in [24].

by combining the sparsified representative Laplacian matrix with the complement matrix. Next, FFT and IFFT algorithms are applied to compute the sparsified HB Jacobian matrix in frequency domain that can be leveraged as a robust and efficient HB preconditioner in the following Krylov-subspace iterative procedures.

It should be noted that during the iterative solution procedure, the HB Jacobian matrices need not be constructed explicitly. Instead, only the matrix-vector multiplications are computed at each iteration, which is more computational and memory efficient than the original HB methods that typically express the full Jacobian matrix explicitly. Although the matrix factors L and U of the support-circuit preconditioner have to be formed explicitly, the proposed graph (circuit) sparsification technique can greatly reduce the memory and runtime consumption. The sparsified preconditioner matrix is usually much sparser than the original HB Jacobian matrix, and it thus can be more quickly solved using existing direct solution methods, such as the block LU solver proposed in [16], leading to nearly-linear runtime and memory efficiency.

2.2 Support-circuit Preconditioner for HB Analysis

This section describes the detailed procedures for computing the proposed support-circuit preconditioner for HB analysis.

2.2.1 Sparsification of Representative Laplacian Matrices

2.2.1.1 Extraction of Representative Laplacian Matrices

It is hoped that by examining spatiotemporal MNA matrix patterns obtained through transient circuit analysis, more meaningful support-circuit preconditioners can be generated for HB analysis based on graph sparsification techniques. To this end, we propose a simple method for extracting the representative Laplacian matrix by examining time-domain sampled MNA system matrices.

To extract the desired support circuit from the original RF circuit scheme, the equivalent resistors and capacitors of each NR iteration (during TR analysis) are first stamped into the Laplacian matrix A_L while the rest components are stamped into the complement matrix A_C for each sampled time point, as shown in Fig. 2.2. To properly preserve the impact of energy-storage components, a fixed time-step size which is determined by the largest harmonic in HB analysis can be adopted for computing the system MNA matrix. For example, the equivalent conductance $C/\Delta t$ for a capacitor with a value of C will be stamped into the Laplacian matrix A_L during transient analysis, where Δt is the time step size corresponding to the highest frequency harmonic component. The above stamping strategy can assure that the proposed support-circuit preconditioner will not miss the edges presenting critical energy-storage elements.

It should also be noted that for different sampled time points, although the entry values of the system MNA matrices obtained from nonlinear device evaluations can be quite different, the corresponding entry locations (patterns) remain the same. As a result, the Laplacian matrices that include the resistors and capacitors derived from

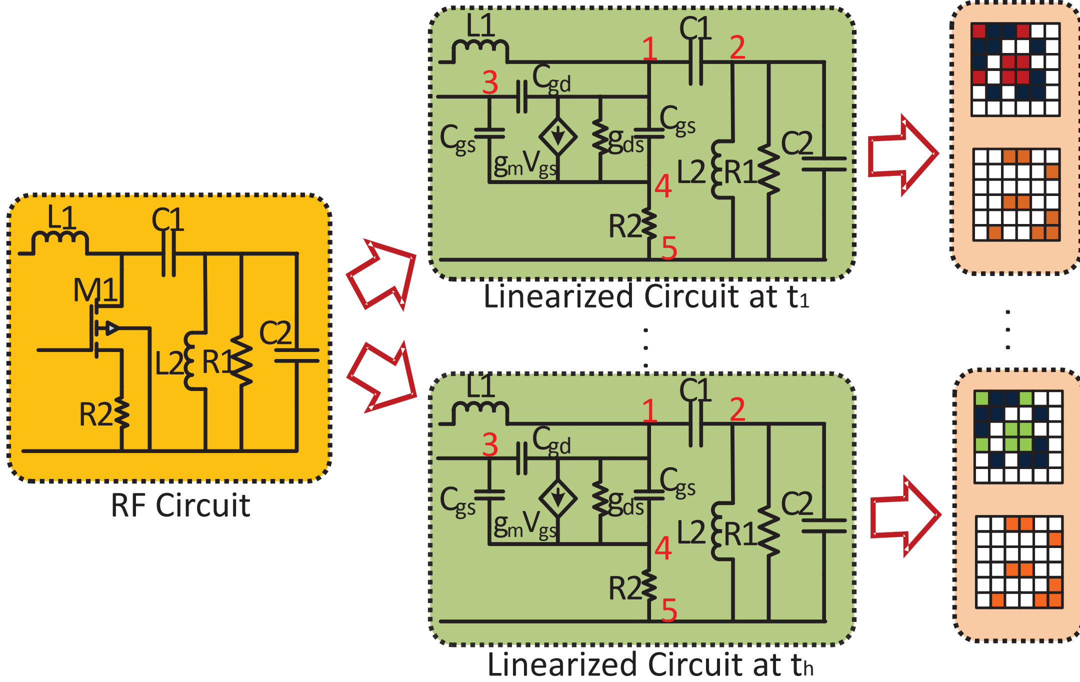


Figure 2.2: Circuit MNA matrices decomposed into Laplacian and complement matrices

nonlinear device linearizations are time-varying during NR iterations. Since the amplitudes of matrix entries sampled at different time points can be quite different from each other, directly averaging these Laplacian matrices may not effectively reflect the influence of some important circuit components. To retain the relatively important circuit components, the sampled Laplacian matrices will be normalized as follows: for each sampled Laplacian matrix obtained at a time point, all the matrix entries are scaled by a common factor such that the largest elements of these Laplacian matrices are always the same. As a result, by averaging these normalized Laplacian matrices, we can obtain a representative Laplacian matrix that can truthfully mimic the average circuit behaviors during transient simulations. Take the r_{ds} of transistor companion model as an example, as shown in Fig 2.6. At different sampled time points the resistance of r_{ds} will be quite different. By normalizing and averaging r_{ds} of all the sampled time points, the new edge weight can reflect the relative importance of

this resistor under all the harmonics.

2.2.1.2 Sparsification of Representative Laplacian Matrices

As described in Section 2.1.2.2, ultra-sparsifier preconditioner can better approximate the original system than spanning-tree preconditioner by adding extra edges to the spanning-tree support graph. However, adding excessive edges to the spanning tree may result in a rather dense graph and thus it can lead to dramatically increased computation cost when using the ultra sparsifier as a preconditioner. In order to find the most important extra edges, the conductivity of the original graph and the degree of each vertex will be analyzed using a *weighted degree* metric. For a graph Laplacian matrix A_L, the weighted degree $wd(v)$ of a vertex $v \in A_L$ is defined as[46]:

$$wd(v) = \frac{w_t(v)}{max_{u \in S(v)} w(u,v)}, \tag{2.12}$$

where $w_t(v)$ denotes the total weight (conductance) incident to the vertex v, $S(v)$ represents the set of edges connected with v, and $w(u,v)$ is the weight of the edge that connects vertex u and vertex v.

In this work, we propose an effective method to determine the edges to be added to the spanning tree as shown in Algorithm 1. In the algorithm, we define $\alpha(v)$ to be the matching factor of node weighted degree:

$$\alpha(v) = \frac{\tilde{wd}(v)}{wd(v)}, \tag{2.13}$$

where $\tilde{wd}(v)$ is the weighted degree of vertex v in the support graph. We also define α_{th} as the threshold of the weighted degree matching factor, which can be used to effectively control the approximation quality of the ultra-sparsifier graph. For a

weighted graph, the lower bound of α_{th} can be computed by setting $\tilde{wd}(v) = 1$ and $wd(v) = wd_{max}$, where wd_{max} denotes the maximum weighted degree of the original graph. Consequently, it can be shown that α_{th} will always fall within the range:

$$\frac{1}{wd_{max}} < \alpha_{th} \leq 1. \tag{2.14}$$

When α_{th} is set to be close to $\frac{1}{wd_{max}}$, the ultra-sparsifier support graph will shrink to a spanning tree; on the other hand, when $\alpha_{th} = 1$, no sparsification is performed, which will result in the original graph.

Algorithm 1 Ultra-Sparsifier Generation Algorithm

Input: Laplacian matrix $A_L \in \Re^{n \times n}$ of the representative Laplacian network.
Output: Laplacian matrix $B_L \in \Re^{n \times n}$ of the ultra sparsifier.

1: **for** node $v \in A_L$ **do**
2: Compute the weighted degree $wd(v)$ with (2.12).
3: **end for**
4: Obtain the sorted the node list $swd \in \Re^n$ in descending order according to the wd of each node.
5: Arbitrarily pick one starting node and compute the maximum spanning tree support graph of graph A_L.
6: **for** $k = 0$ **to** $n - 1$ **do**
7: Get node from the sorted node list $v = swd(k)$
8: Compute the latest weighted degree $\tilde{wd}(v)$ with (2.12) and the matching factor $\alpha(v)$ with (2.13).
9: **while** $\alpha(v) < \alpha_{th}$ **do**
10: Restore a single previously removed edge that has the largest weight.
11: Update the weighted degree $\tilde{wd}(v)$ with (2.12).
12: **end while**
13: **end for**
14: Return the Laplacian matrix B_L of the final ultra-sparsifier.

2.2.2 Sparsification Pattern Extraction

After the graph sparsification algorithm (Algorithm 1) is applied to compute the sparsified representative Laplacian matrix by finding the ultra-sparsifier support graph

according to its Laplacian matrix, the support-circuit preconditioner can be subsequently constructed by exploiting the sparsification pattern of the representative MNA matrix. The procedures for finding the sparsification pattern of the representative MNA matrix have been illustrated in Fig. 2.2 and Fig. 2.3, while detailed steps are described as follows:

1. After performing circuit linearizations at different time points (Section 2.1.1), the Laplacian matrices that only include the equivalent resistors and capacitors are extracted from the system MNA matrices, as shown in Fig. 2.2;

2. The representative Laplacian matrix is created by normalizing and averaging the Laplacian matrices sampled at multiple time points, as described in Section 2.2.1;

3. The representative Laplacian matrix is converted to an undirected graph for which an ultra-sparsifier support graph is created using Algorithm 1. Next, the ultra sparsifier is converted to its matrix form that is defined as the sparsified representative Laplacian matrix, as illustrated in Fig. 2.3;

4. Finally, the complement matrix is combined with the sparsified representative Laplacian matrix to create the final sparsification pattern matrix, as illustrated in Fig. 2.3.

Once the sparsification pattern matrix is obtained, it can be adopted to sparsify the HB Jacobian matrix: if one entry in the sparsification pattern matrix is removed, the corresponding block matrix (2.6) in the HB Jacobian matrix will also be eliminated. We will show that this sparsification pattern matrix can very efficiently and effectively facilitate the sparsification of large HB Jacobian matrices.

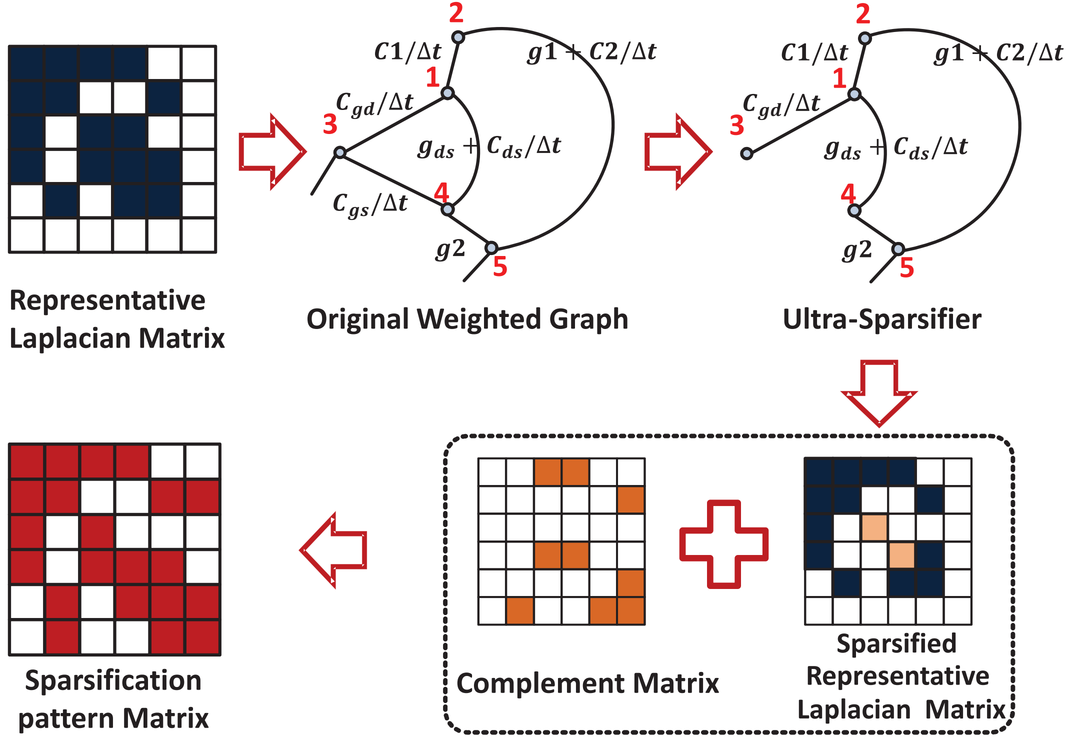


Figure 2.3: MNA matrix sparsification pattern

2.2.3 HB Jacobian Preconditioner Construction

As discussed in Section 2.1.1, the dense blocks in HB Jacobian matrix J_{hb} are mainly due to the block-circulant matrices similar to (2.6). It has been shown that by reordering the unknowns, the HB Jacobian matrix J_{hb} can be converted to an equivalent sparse block matrix that maintains the same sparsity as the MNA matrix for TR analysis [16], as illustrated in Fig. 2.4. For example, if there are three harmonics in the HB analysis, the HB Jacobian matrix can be obtained by replacing every entry in the time-domain system MNA matrix with a 3 by 3 circulant-matrix block. Similarly, it is not difficult to construct the sparsified HB Jacobian matrix when the sparsification pattern is obtained through the previous procedures. For instance, we can compute the circulant-matrix block by applying FFT to the entries of the sparsified

time-domain sampled MNA matrices, as described in (2.6).

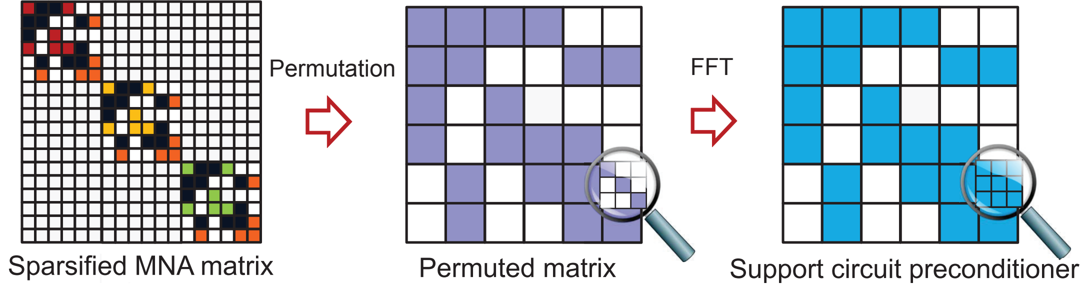


Figure 2.4: HB Jacobian matrix construction

We want to emphasize that the proposed HB preconditioner matrix can be more efficiently factorized than the original HB Jacobian matrix, since only a small number of block fill-ins will be created during the block LU matrix factorization procedures. In this work, similar to the block LU solver developed in [16], we have developed a block LU solver (descried in section 2.3) for factorizing the HB Jacobian matrix preconditioner. Due to the tree-like structure of the sparsified RF circuit, the HB Jacobian preconditioner matrix can be solved in nearly linear time. Since the proposed preconditioning method shares the advantages of prior direct solution methods [16], it can be efficiently and reliably applied to handle strong nonlinearities in HB analysis of RF circuits.

2.2.4 Case Study: Double-balanced Gilbert Mixer Sparsification

RF mixers serve as key elements in superheterodyne transceivers for wireless systems, but they are also the primary sources of nonlinear distortions. Most mixer designs (Fig. 2.5) are following the principle that a large local oscillator (LO) driven by an RF signal can modulate the incoming RF signal into an intermediate frequency (IF)

signal.

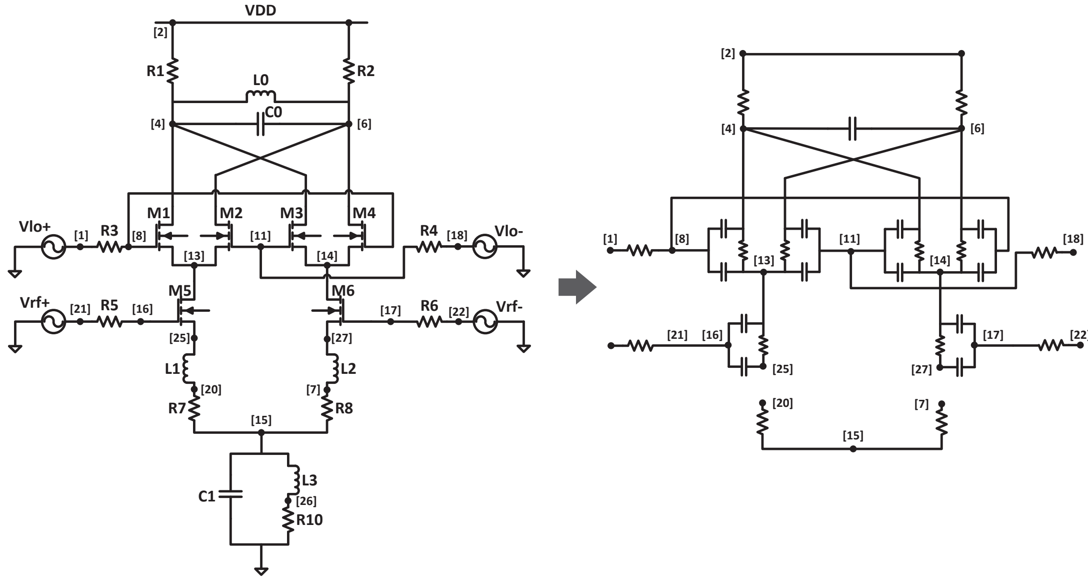


Figure 2.5: An RF mixer design (left) and the circuit network (right) corresponding to its Laplacian matrix

In this section, we demonstrate a case study to show how an RF circuit can be sparsified into a sparser support circuit for preconditioned HB analysis. First, the equivalent resistors and capacitors of the linearized RF circuit need to be extracted for forming the Laplacian matrix A_L. Consider a linearized MOSFET circuit in Fig. 2.6 that shows equivalent resistors and capacitors, such as C_{gd}, C_{gs}, and r_{ds} as well as two controlled current sources. By keeping only the non-grounded equivalent resistors and capacitors of the linearized mixer circuit, a Laplacian network can be obtained, as shown in Fig. 2.5. Like other SPICE simulation algorithms, each inductor will be treated similarly to voltage sources, in that it cannot be modeled using an I-V style Ohmic relationship, and requires an extra current variable when building the MNA matrix. As a result, all the stamped elements related to inductors will be kept in the complement network. The devices connected to ground, such as $C1$ and $R10$ in Fig. 2.5, will only introduce diagonal entries in the MNA matrix. It is also obvious that these devices will not introduce any entry to the Laplacian matrix.

Next, the Laplacian network can be converted into a weighted, undirected graph, and subsequently, an ultra-sparsifier graph can be created by following the steps in Algorithm 1.

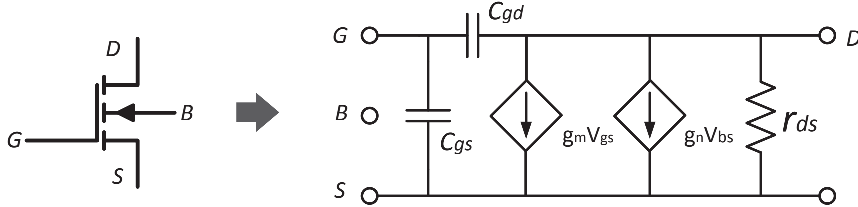


Figure 2.6: MOSFET model

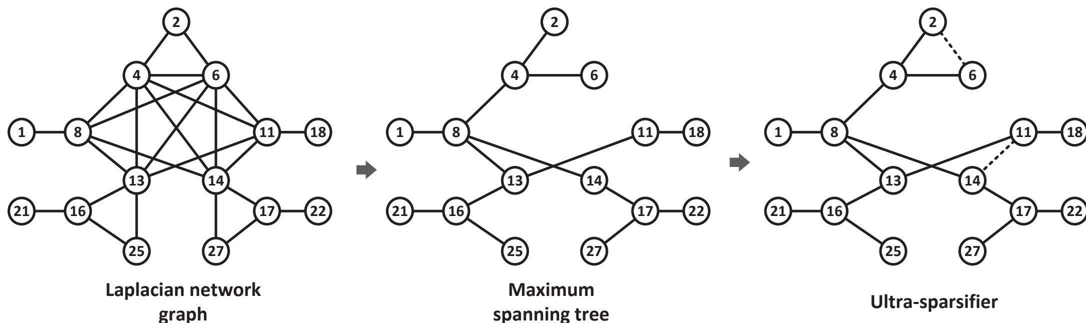


Figure 2.7: Sparsification of the Laplacian graph of an RF mixer circuit

2.3 Parallel Block Sparse Matrix Direct Solver

When using preconditioned Krylov-subspace iterative methods for HB analysis, such as the GMRES method [20], the Jacobian preconditioner matrix needs to be factorized once during each NR iteration. Therefore, the efficiency of the preconditioner factorization is critical to the overall performance of HB analysis. Inspired by the method proposed in [16, 47], we propose a parallel block sparse matrix direct solver, which can efficiently factorize the HB Jacobian preconditioner matrix on heterogeneous CPU-GPU computing platforms or CPU only platforms. Although only two

basic block matrix operations exist in the sparse block LU factorization procedure: block matrix multiplication and division, optimal acceleration of block LU solver on heterogeneous CPU-GPU computing platforms may not be trivial, since the optimal workload balancing and task assignments will not only depend the hardware specific properties of a given heterogeneous computing platform (e.g. the availability of streaming multiprocessors, the size of on-chip shared memory and registers, I/O bandwidth and latency, etc), but also the algorithm specific properties, such as the data dependencies during the sparse block LU factorization procedure.

2.3.1 LU Data Dependency Analysis

Algorithm 2 LU factorization $PJQ = LU$

Input: $J \in \Re^{n \times n}$
Output: $L, U \in \Re^{n \times n}$,$Q \in \Re^{n}$,$P \in \Re^{n}$

1: $Q = AMD(J)$ {column reordering to reduce fill-ins}
2: $L = I_n$
3: **for** $k = 1$ **to** n **do**
4: $x = \text{LTS}(L, J(:, k))$
5: $P = PartPivoting(x)$ {partial pivoting}
6: $U(1 : k, k) = x(1 : k)$
7: $L(k : n, k) = x(k : n)/U(k, k)$
8: **end for**

Algorithm 3 LTS: Lower triangular system solver $Lx = b$

Input: $L \in \Re^{n \times n}$, $b \in \Re^{n}$
Output: $x \in \Re^{n}$

1: $\theta = \{i \mid b_i \neq 0\}$
2: $\chi = Reachable_{G_L}(\theta)$ {nonzero location of x, $\chi = \{i \mid x_i \neq 0\}$ }
3: $x = b$
4: **for** $j \in \chi$ **do**
5: $x(j + 1 : n) = x(j + 1 : n) - L(j + 1 : n, j)x(j)$
6: **end for**

First,we review a classical LU factorization algorithm as shown in Algorithm 2 and

3. This left-looking LU factorization algorithm [48] includes two phases: a symbolic analysis phase and a numerical analysis phase. During the symbolic analysis phase, the columns of the matrix will be reordered to reduce fill-ins during the factorization. There are several ordering approaches available, such as the Approximate Minimum Ordering(AMD)[49],and COLAMD[50] ordering schemes. During the numerical analysis phase, the L and U factors will be calculated with partial pivoting to reorder the rows to avoid very small diagonal elements. The core of the numerical analysis phase is the Gilbert/Peiels factorization algorithm[51],which is to solve a lower triangular system $Lx = b$ repeatedly.

It is usually difficult to parallelize sparse LU factorization procedures due to the complicated data dependencies and imbalanced workloads. From line 4 of Algorithm 2 and Algorithm 3, we can observe that the column k depends on column j, when $U(j, k) \neq 0$. Because of the high data dependency between different columns, the appropriate timing order of the computation in LU factorization must be guaranteed. According to the above dependency relationship, we can derive the data dependency graph (DDG) from the U factor matrix, as shown in Fig. 2.8 where each node represents a column and each arrow denotes a dependency between two columns.

2.3.2 "Fake" Dependencies in LU Factorization

However, the dependencies between columns derived from the U factor matrix do not always introduce matrix operations. From line 5 of Algorithm 3, we can observe that if there is no nonzero entry in the non-diagonal positions of column j in the L factor matrix, the matrix multiplication operation can be skipped. We take the dependency between column 7 and column 4 as an example to illustrate this situation: from the L factor matrix, as shown in Fig. 2.9, we can observe that there is no nonzero

entry except for the diagonal entry in column 4. As a result, there will be no matrix multiplication operations introduced by the dependency between line 7 and line 4. We call such a data dependency a "fake" dependency. These "fake" dependencies can be quickly identified by checking the nonzero entries of the L factor matrix. For instance, the dotted lines with arrows in Fig 2.8 are all "fake" dependencies. Apparently, by eliminating these "fake" dependencies, we can obtain a much balancer workload for each level, as shown in Fig. 2.9.

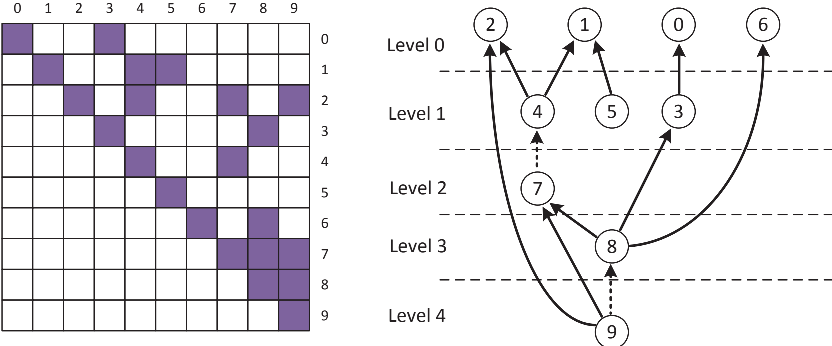


Figure 2.8: Upper matrix factor and its DDG

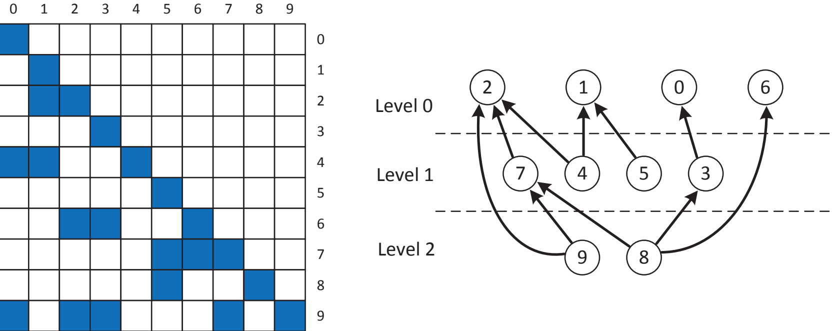


Figure 2.9: Lower matrix factor and its improved DDG

From the above discussion, it is clear that in order to get the DDG, we need to know the nonzero-entry locations of L and U factors. Fortunately, by factorizing the sparsified representative MNA matrix (a.k.a test matrix) obtained from RF circuit transient analysis [25], the nonzero entries of L and U factor matrices can be precisely

predicted before performing the actual block sparse LU factorization.

2.3.3 Parallel LU Task Scheduling

Once the LU DDG is obtained, it can be further partitioned into multiple levels. Columns at the same level are independent and can be therefore computed concurrently using parallel processors (streaming multiprocessors). It should be noted that if a column depends on several other columns, the matrix operations must be executed in a strict order starting from the left most column. After getting the levelized dependency graph, the final LU factorization task list can be generated from the top level to the bottom level. For each task level, a batch of matrix multiplications and matrix divisions will be performed as described in Algorithm 2 and 3. For each task level, there are two task lists related to matrix multiplications and matrix division respectively, and the computation order of the two task lists are also strictly enforced: the division task list has to wait for the completion of multiplication tasks. For CPU only platform, the high-performance BLAS[52] library can efficiently perform above operations with multiple threads. And for GPU platform, the highly efficient cuBLAS library [53] is able to take full advantage of the high computational capability of modern GPUs.

2.3.4 Test Matrix Factorization

As described in Section 2.3.1, a symbolic analysis procedure needs to be performed to find the nonzero entry locations of U factor matrix. It is hard to obtain good performance for this complicated process on GPU platform. Inspired by [16], we extract the nonzero entry locations by factorizing the test matrix.

Before performing block LU factorization on the preconditioner matrix, we first examine the sparsified representative MNA matrix (test matrix) obtained from TR analysis. Since the HB preconditioner matrix has the same block entry pattern as the sparsified representative MNA matrix, each entry in the test matrix will correspond to a block entry in the HB Jacobian preconditioner matrix. Consequently, by factorizing this test matrix, the possible fill-in locations and computations during the following BLU factorizations can be precisely predicted.

Besides the nonzero entry locations, we can also obtain the row and column permutation vectors during the factorization of test matrix. As a result, we can simply apply these vector to the Jacobian matrix before the factorization and avoid the rich branching pivoting procedure on GPU.

2.3.5 The Sparse Block LU Algorithm

In this subsection, we will extend the above classical LU algorithm to develop our sparse block LU algorithms (Algorithm 4 for factorizing the HB Jacobian preconditioner matrix efficiently. The key idea of our block LU algorithm is to exploit the results of the previous test matrix LU factorization. After factorizing the test matrix, the column (Q) and row (P) permutation vectors will be obtained respectively, which can be readily leveraged to factorize the HB Jacobian matrix. By replacing the element-wise multiplications and divisions with matrix-wise multiplication and divisions, the HB preconditioner matrix can be factorized very efficiently. Since each block entry of the HB preconditioner matrix is a dense matrix, the matrix-wise multiplications and divisions can be performed using aggressively optimized BLAS and cuBLAS implementations for the target architecture.

Algorithm 4 Parallel Block LU Factorization Algorithm

Input: Block sparse matrix, nonzero-entry locations of L and U factor matrices, permutation vectors

Output: L and U factor matrices

1: Perform pivoting to the block sparse matrix according to pivoting vector
2: Create the column dependency graph according to L and U nonzero-entry locations
3: Generate the matrix multiplication and division task list for each level of the dependency graph
4: Allocate memory for L and U factor matrices
5: Distribute the block sparse matrix to L and U memory buffer
6: For GPU: Transfer the L and U factor matrices to GPU
7: **for** Each level **do**
8:     Calculate matrix multiplications concurrently by BLAS or cuBLAS.
9:     Calculate matrix divisions concurrently by BLAS or cuBLAS.
10: **end for**
11: For GPU: Transfer the L and U factor matrix results back to CPU

2.4 Transient Analysis Guided Sparsification

The key to obtaining the optimal computing performance for HB analysis using the proposed support-circuit preconditioning algorithm is to find the optimal graph sparsification strategy for generating the support-circuit preconditioners. To this end, efficient yet accurate runtime performance models will be developed in this work for assessing the performance of each graph sparsification configuration, which can be subsequently leveraged to facilitate runtime performance optimization that can eventually identify the optimal graph sparsification strategy as well as the corresponding support-circuit preconditioner for minimizing HB runtime for a given computing platform.

2.4.1 HB Simulation Runtime Profiling

The total runtime of a complete NR iteration during HB analysis can be estimated as follows:

$$T_{NR} = T_{LU}(\alpha_{th}) + N(\alpha_{th}) \cdot T_{GMRES}(\alpha_{th}), \tag{2.15}$$

where T_{LU} is HB preconditioner matrix factorization time, T_{GMRES} is the runtime for one GMRES iteration and N denotes the total number of GMRES iterations during one NR iteration step, which are all functions of the preconditioner sparsity controlled by α_{th}. As illustrated in Fig. 2.10, a sparser ultra-sparsifier support graph will result in smaller preconditioner factorization time T_{LU}, but much more iterations can be expected due to the worse approximation of the original graph. On the other hand, a denser support graph will result in greater cost in factorizing the HB Jacobian preconditioner matrix that can dominate the overall simulation time. Therefore, to achieve the optimal runtime performance of HB simulation using the proposed iterative solver, the optimal sparsity (α_{th}) of the preconditioner matrix needs to be determined efficiently.

Consider an example shown in Fig. 2.11 which illustrates the plot of total HB runtime as well as LU factorization and preconditioned GMRES iteration runtimes by sweeping α_{th} on different computing platform. It is observed that the best α_{th} is near the crossing point (marked in the red square) of plots of "LU runtime" that equals to $T_{LU}(\alpha_{th})$ and "Resolve runtime" that equals to $N(\alpha_{th}) \cdot T_{GMRES}(\alpha_{th})$. To the left of the crossing point, the preconditioner is quite sparse, which leads to smaller LU factorization runtime, and as a result, the GMRES resolve runtime dominates HB simulation time. On the other hand, to the right of the crossing point, the LU factorization runtime increases dramatically when the preconditioner is getting denser. As a result, the LU factorization will dominate the overall simulation runtime.

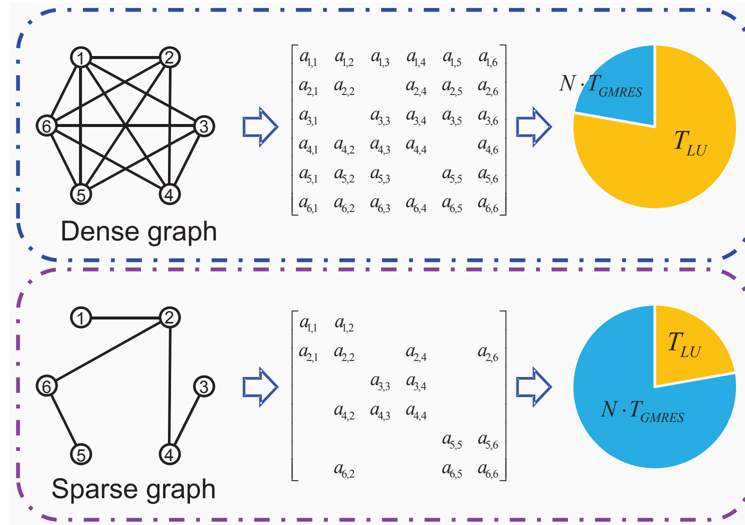


Figure 2.10: Runtime profiling for solving two sparsified graphs

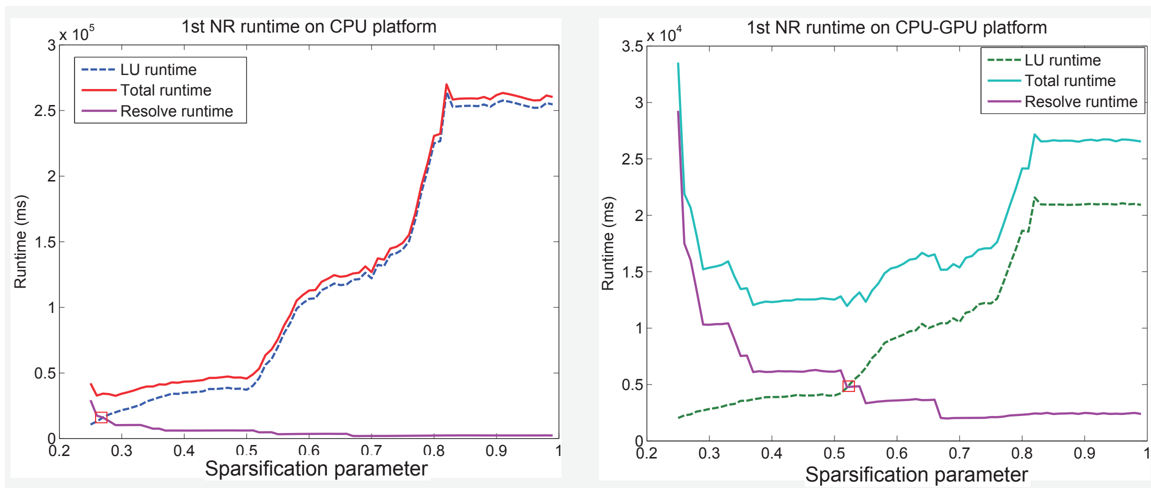


Figure 2.11: HB simulation runtime with different sparsity

It should be noted that for different computing platforms with various settings of CPUs and GPUs, the best graph sparsification configuration can be quite different. For instance, for CPU-dominated computing platforms (strong CPUs+weak GPUs), due to limited computational capability, a sparser preconditioner has to be used to reduce the LU factorization cost on CPU, whereas for GPU-dominated platforms (weak CPUs+strong GPUs) the LU factorization time may be greatly reduced due

to significantly higher computing capability of GPUs, and therefore denser support graphs should be adopted for achieving a faster convergence.

2.4.2 Runtime Performance Modeling

To identify the best possible graph sparsification strategy for generating the support-circuit preconditioner that can minimize the overall HB simulation time, we propose a systematic approach to automatically and robustly find nearly-optimal matching factor threshold α_{th} of weighted degree by using a transient analysis-guided performance modeling approach. By encapsulating hardware specific properties of a given computing system into this simple yet effective runtime performance model, the runtime performance of support-circuit preconditioned HB simulation can be quickly estimated, while the nearly-optimal graph sparsification configuration (matching factor threshold α_{th}) can be subsequently identified.

Although the matrix dimension of the TR analysis is much smaller than the matrix dimension in HB analysis, since TR analysis has the same Jacobian matrix structure as the HB analysis, it can be used as a good surrogate for estimating the efficiency of the HB preconditioner when the same representative sparse matrix pattern is used. As a result, TR analysis runtime can effectively reflect the efficiency of the corresponding HB preconditioner. For instance, if the GMRES iteration number is large during TR analysis, it indicates that the preconditioner may not be accurate enough, and therefore additional edges should be included into the sparsified graph. We want to emphasize that the TR analysis does not need to reach the steady state for performance modeling purpose. Instead, we just need to perform a few steps of TR analysis to proximately estimate the quality of the HB preconditioner based on the convergence behavior. Although the resultant HB preconditioner may not be the optimal

one during the entire HB analysis procedure, it should be able to provide a reasonably good initial preconditioner that can be improved later based on the actual numbers of GMRES iterations during the following HB analysis steps (e.g. NR iterations).

Fig.2.11 indicate that we can build a quadratic runtime performance model for predicting HB simulation runtime simulation runtime under different α_{th} values:

$$T_{HB} = a\alpha_{th}^2 + b\alpha_{th} + c, \tag{2.16}$$

where T_{HB} represents the predicted overall runtime of HB simulation and a, b, and c are the coefficients of the quadratic function. In our approach, N different matching factor thresholds, $\alpha_{th,i}\ for\ i = 1, ..., N$, are uniformly chosen within the range (2.13) described in Section 2.2.1.2. First the HB simulation runtime of each sample matching factor thresholds will be estimated by running few steps of TR simulation. Then coefficients of the quadratic function can be calculated by using curve fitting method based on the predicted HB simulation runtime. Subsequently, the identification of the optimal α_{th} for HB analysis can be efficiently performed based on the above runtime performance model. It needs to be noted that the estimate HB simulation runtime does not need to be accurate. It is sufficient to find the extreme value if the estimate HB simulation runtime is proportional to real HB simulation runtime. Since running TR analysis is much faster than running full HB simulations for RF circuits, the cost of building the proposed performance model can be negligible. It should be noted that this novel HB runtime performance modeling approach allows to automatically and robustly compute the nearly-optimal sparsified circuit networks for preconditioning purpose, while previous manually tuned sparsification algorithm in [25] may require excessive effort in finding a relatively good preconditioner. At last, we want to emphasize that the performance model of different types of platforms can be quite different due to the different operating platform between TR and HB analysis.

2.4.2.1 CPU Only Platform Performance Model

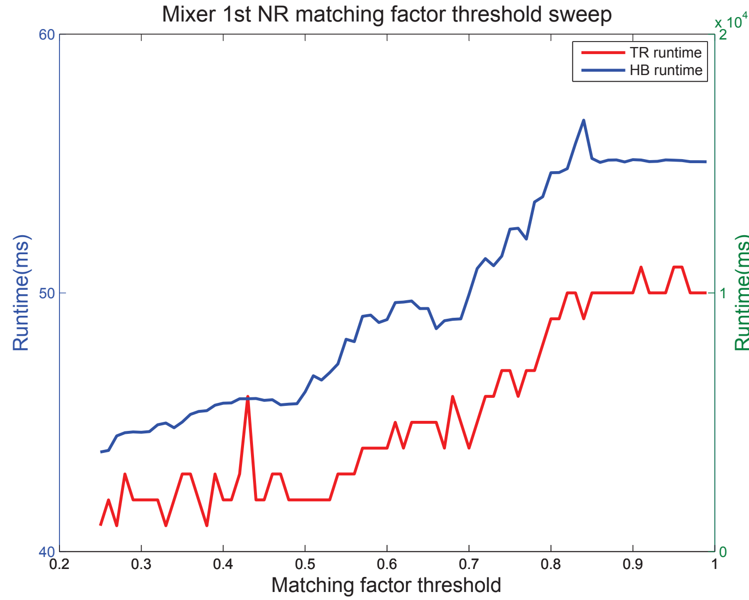


Figure 2.12: TR simulation runtime vs. HB simulation runtime

For CPU only platform, TR and HB analysis are both performed on CPU with same factorization and iterative solve algorithms. As a result, TR simulation runtime results under different α_{th} values correlate well with the corresponding HB simulation runtime results, as shown in Fig. 2.12. The above observation suggests building the quadratic performance model under the guidance of TR analysis results directly. Subsequently, the identification of the optimal α_{th} for HB analysis can be efficiently performed based on the above runtime performance model.

2.4.2.2 CPU-GPU Platform Performance Model

Unlike the CPU only platform, the runtime performance model for GPU-based sparse block LU factorization cannot be directly obtained by only performing incomplete

Algorithm 5 CPU-GPU runtime performance modeling for HB analysis

Input: maximum weighted degree (wd_{max}) of the representative Laplacian matrix graph
Output: Best matching factor threshold α_{th}

1: Generate N different matching factor thresholds $\alpha_{th,i}, i \in (1, \ldots, N)$ uniformly within the range $[\frac{1}{wd_{max}}, \ldots, 1]$;
2: **for** $\alpha_{th,i}, i = 1$ **to** N **do**
3: Perform M steps transient simulations with GMRES iterative solver and record the GMRES resolve runtime cost $T_{TRGMRES}$ during the transient simulation;
4: Generate sparsification pattern with α_{th} and sparsify the MNA matrix at each sample time point;
5: Factorize the representative MNA matrix to obtain the nonzero entry locations of L and U factor matrices;
6: Create DDG according to nonzero entry locations of L and U;
7: Create the parallel levelized LU task list $LUlist_j, j \in (1, \ldots, K)$ for GPU computing from the DDG, where k is the total levels of the LU task list;
8: Load the pre-obtained computing platform specific runtime performance lookup table;
9: **for** $LUlist_j, j = 1$ **to** K **do**
10: $THB_{LUgpu} = THB_{LUgpu} + lookup(h_j, b_j)$, where THB_{LUgpu} is the predicted HB preconditioner LU factorization runtime on GPU, h_j denotes the harmonic number and b_j represents batch number of the matrix operation;
11: **end for**
12: $THB_{GMRES} = T_{TRGMRES} * h$, where THB_{GMRES} represents the predicted HB GMRES resolve time cost, h denotes the total harmonic number.
13: $THB[i] = THB_{LUgpu} + THB_{GMRES}$, where $THB[i]$ denotes the overall HB analysis runtime with $\alpha_{th,i}$;
14: **end for**
15: Quadratic performance model generation using curve fitting method.
16: Return the α_{th} with the minimal HB total runtime.

transient simulations on CPU. The reason is that the sparse block LU factorization on GPU is operated for HB Jacobian matrix in a parallel manner, whereas the LU factorization in transient simulation on CPU deals with the time-domain MNA matrix directly. Fortunately, the CPU-based preconditioned GMRES iterations of the HB analysis and transient simulation use the same matrix sparsity pattern but different matrix sizes. Hence, we can predict the preconditioned GMRES iteration time precisely according to the incomplete transient simulation results (preconditioned GMRES iteration time for MNA matrices), as shown in line 12 of Algorithm 5. To predict the sparse block LU factorization runtime on GPU, we propose to build $2D$

lookup tables (LUTs) for predicting the runtime performance of batched matrix multiplications and divisions. The block size and the number of batched matrix operations are the inputs parameters of the 2D LUTs. As a result, once the parallel LU factorization tasks have been assigned, the GPU-based LU factorization runtime can be easily predicted using the above LUTs for each LU factorization task level. The LUTs only need to be built once for a given CPU-GPU computing platform, and can be utilized for subsequent HB simulation tasks. In the last, the performance model of GPU-based sparse block LU factorization is combined with the performance model for solving the HB Jacobian matrix using preconditioned GMRES iterations to create the total runtime performance model that can be further used to identify the optimal α_{th} value for running support-circuit preconditioned HB analysis on a given CPU-GPU heterogeneous computing platform.

2.4.3 Nearly-optimal Sparsification Scheme

Since the linearized models of nonlinear devices may change drastically during NR iterations, the sparsified circuit networks may also change accordingly, which may require an on-the-fly update of the preconditioner. In this work, we propose to apply the quadratic runtime performance modeling/optimization procedures only when the previous NR iteration runtime has changed dramatically. On the other hand, when the runtime of the latest NR iteration does not change much, the previous sparsified network topology will be reused with updated element values. Fig. 2.13 illustrates the proposed sparsification scheme during an NR iteration of HB simulation, where $T_{HB}(k)$ denotes the k_{th} NR iteration runtime and β denotes the threshold value of the changing rate for NR iteration runtime. Since the runtime performance models can be efficiently generated on-the-fly for finding the near optimal α_{th}, high robustness and efficiency of HB simulation using the proposed adaptive preconditioning approach

can be always achieved.

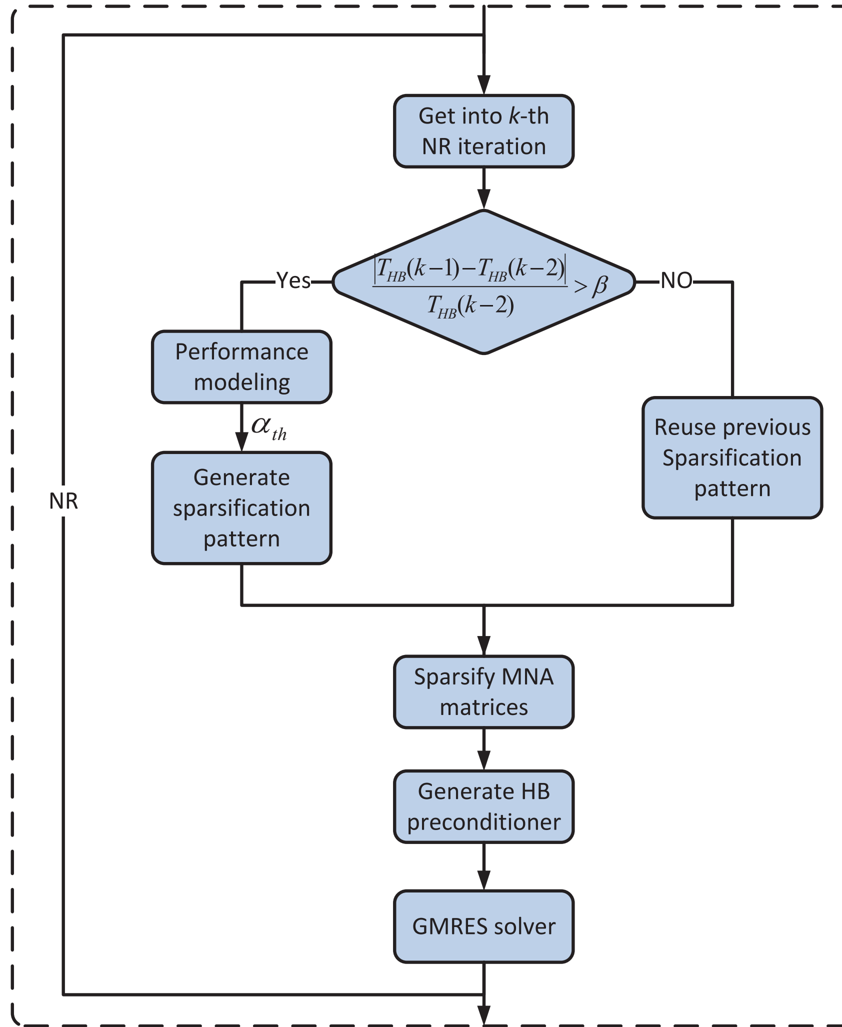


Figure 2.13: Performance-guided sparsification scheme

We want to emphasize that when the circuit becomes nonlinear device dominant, the proposed preconditioner will be very similar to the original HB Jacobian matrix. As a result, the proposed method will work like a direct solver. As the increase of parasitic components of the post-layout circuits, the proposed preconditioner will be much sparser than the original HB Jacobian matrix, and the proposed method will work as an iterative solver. The automatic and smooth switching between the direct

solver (no sparsification) and iterative solver (multiple sparsification levels) allows the proposed method to reliably and efficiently obtain the steady state solution for different RF circuits.

2.5 The Scalable HB Analysis Algorithm

Algorithm 6 The SCPHB method

Input: RF circuit netlist.
Output: Solution.

1: Set up the solver;
2: **while** performing a Newton-Raphson iteration **do**
3: Evaluate devices and compute the linearized circuit system matrices at each sample time points;
4: Create the Laplacian and complement matrices at each sample time point;
5: **if** Previous HB NR iteration runtime change drastically **then**
6: Update the sparsification pattern:
7: a) Create the representative Laplacian matrix P by scaling and averaging all the Laplacian matrices;
8: b) Create the performance model function $T_{HB} = f(\alpha_{th})$;
9: c) Compute the optimal matching factor threshold α_{th};
10: d) Extract the ultra-sparsifier support graph from the representative Laplacian matrix P;
11: e) Build the sparsified representative Laplacian matrix P_{usg};
12: f) Form the sparsification pattern by combining the sparsified representative Laplacian matrix with the complement matrix;
13: **else**
14: Reuse the previous sparsification pattern;
15: **end if**
16: Sparsify the HB Jacobian matrix;
17: Construct and factorize the Jacobian preconditioner matrix;
18: Perform preconditioned GMRES iterations;
19: Update the solution vector and transform the solution from frequency domain to time domain using IFFT;
20: Check NR convergence: if converged, stop the NR iteration; otherwise, go to the next NR iteration.
21: **end while**
22: Return the final steady-state solution.

The algorithm flow of the proposed support-circuit preconditioned HB (SCPHB) method is summarized in Algorithm 6, while the complexity of the proposed algorithm is discussed as follows. Benefited from the iterative solution method, we only need to explicitly construct the factor matrices L and U of the preconditioner. Since the ultra-sparsifier preconditioner maintains a tree-like structure, the memory and computational cost due to the fill-ins introduced during the block LU factorization procedure will scale almost linearly with the problem size. Therefore, the memory cost can be estimated by:

$$(nnzl_{sg} + nnzu_{sg})h^2$$

where $nnzl_{sg}$ and $nnzu_{sg}$ denote the nonzeros in the L and U factors of the HB Jacobian preconditioner, while h denotes the number of harmonics for HB analysis. It is obvious that the proposed method has better memory efficiency than the direct solution method whose memory consumption is given by [16]:

$$(nnzl + nnzu)h^2$$

where $nnzl$ and $nnzu$ are the nonzeros in L and U factors. Since the proposed sparsification technique can greatly reduce the number of edges/elements of the original circuit network, $(nnzl + nnzu)$ should be much greater than $(nnzl_{sg} + nnzu_{sg})$ due to the dramatically reduced fill-ins during the block LU factorization procedure.

The computation complexity of the block LU solver[16] is

$$O((nnzl_{sg}^{\beta} + nnzu_{sg}^{\beta})h^3)$$

where $\beta > 1$ but it is usually very close to 1.

We assume the preconditioned GMRES method converges in m iterations, then the

Table 2.1
Experimental circuit descriptions

CKT	Name	Node	Harmonic	Unknown
1	Mixer	302	49	14798
2			81	24462
3			99	29898
4	LNA+Mixer	344	45	15480
5			75	25800
6			105	36120
7	Mixer	1988	49	97412
8			63	125244
9			99	196812

complexity of HB matrix-vector products is

$$O(m(nnzc + nnzg)hlog(h))$$

where $nnzc$ and $nnzg$ are the numbers of nonzeros in C and G respectively. The complexity for triangle solves during GMRES iteration canbe estimated to be $O(m(nnzl_{sg}^{\beta} + nnzu_{sg}^{\beta})h^2)$. The complexity of applying the proposed preconditioner in m GMRES iterations is $O(m^2nh)$. Therefore the total runtime complexity including the preconditioner factorization time, preconditioner triangle solve time, and matrix-vector multiplication time can be estimated as:

$$\begin{aligned} &O((nnzl_{sg}^{\beta} + nnzu_{sg}^{\beta})h^3) \\ &+O(m(nnzl_{sg}^{\beta} + nnzu_{sg}^{\beta})h^2) \\ &+O(m(nnzc + nnzg)hlog(h)) \\ &+O(m^2nh). \end{aligned}$$

Based on the above analysis of runtime and memory cost using the proposed method, it is obvious that our approach will lead to much better computational efficiency than the prior direct solution method [16], especially when the number of harmonics during HB analysis is large.

Table 2.2
Results of runtime and memory cost.

CKT	Direct CPU		Direct GPU			SCPHB CPU			SCPHB GPU		
	time(s)	mem (G)	time(s)	mem (G)	speedup	time(s)	speedup	mem_r	time(s)	speedup	mem_r
1	482.3	0.24	72.5	0.2	6.7X	157.7	3.1X	2X	67.9	7.1X	2X
2	522.2	0.65	183.5	0.53	2.8X	169.2	3.1X	2.2X	127.2	4.1X	2.03X
3	3338.9	0.98	318.1	0.8	10.5X	1030.9	3.2X	2.1X	297.8	11.2X	1.95X
4	159.5	0.25	34.5	0.21	4.6X	77.5	2.1X	1.9X	36.9	4.3X	1.75X
5	551.3	0.69	96.8	0.58	5.7X	247.4	2.2X	1.9X	93.4	5.9X	1.8X
6	1409.5	1.35	201.4	1.13	7X	577	2.4X	1.92X	223.9	6.3X	1.79X
7	9316.5	2.9	890.1	2.62	10.5X	1261.8	7.4X	4X	626.7	14.9X	1.89X
8	8685.9	4.8	1267.1	4.3	6.9X	812.2	10.7X	4.4X	794	10.9X	1.26X
9	57420	11.9	N/A	N/A	N/A	8791	6.5X	4.25X	2716.5	21X	N/A

2.6 Experiment Result

2.6.1 Experimental Setup

We have tested several widely used RF circuits using the proposed support-circuit preconditioned HB method. All the test circuits are post-layout RF circuits that include various levels of parasitic elements. To demonstrate the advantages of our proposed method, the direct solution method [16] and support-circuit preconditioned HB method are evaluated both on the CPU-only platform and CPU-GPU heterogeneous platform. The detailed descriptions of the test cases are summarized in Table 2.1, where “Harmonic” denotes the total number of harmonics, and “Unknown” stands for the problem size. All experiments have been performed on RHEL 6.6 64-bit with 2.66GHz 12-core CPU and $48GB$ DRAM memory. The GPU device is Tesla $C2075$ with $5GB$ device memory.

2.6.2 Experimental Results

First, we would like to demonstrate the runtime efficiency of the proposed parallel sparse block LU solver as shown in Table 2.2. In this result table, all the speedup results are compared with multithreading (8 threads) parallel direct solution method. In Table 2.2, "mem" denotes the memory consumption and "mem_r" represents the memory reduction. The "mem_r" of "SCPHB CPU" method is compared with the CPU-based direct method "Direct CPU", and the "mem_r" of "SCPHB GPU" method is compared with the GPU-based direct method "Direct GPU". From Table 2.2, we can observe that with the increase in the problem size, the runtime cost of direct solution method on CPU increase dramatically, which reveals poor scalability of the algorithm. Table 2.2 also shows that the direct method with our GPU-based block matrix solver ("Direct GPU") can run up to $10.5X$ times faster than multithreading direct method on CPUs. However, for the very large problem size (the last test case), "Direct GPU" method cannot be used due to the insufficient GPU memory resources. The results of SCPHB method on CPUs ("SCPHB CPU") demonstrate very satisfactory runtime and memory efficiencies, showing up to $11X$ speedup and $4.4X$ memory reduction, which are benefited from the proposed Jacobian matrix sparsification method. The proposed "SCPHB GPU" method benefits from both the sparsification and the high computational capability of GPU platform, and achieves up to $21X$ speedup comparing with "Direct CPU" method and up to $2X$ memory reduction comparing with "Direct GPU" methods.

It can be also observed that when the node number of the circuit is relatively small, the GPU-accelerated LU factorization contributes most of the speedups. When the node number of post-layout RF circuit is large, the contributions of sparsification start to be more notable. It also needs to be noted that the proposed transient-analysis guided graph sparsification framework will only introduce negligible runtime

overhead. For example, we observed only less than 1% runtime overhead during all the HB simulations shown in this section.

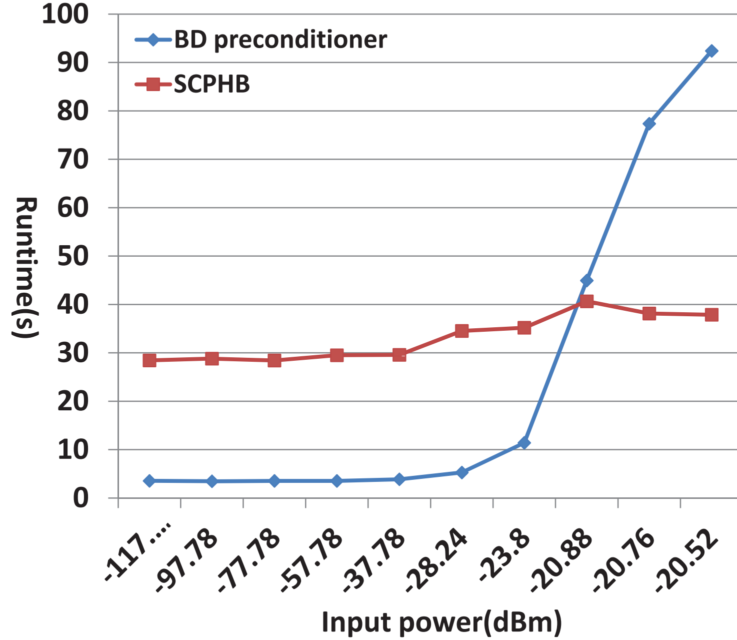


Figure 2.14: HB analysis runtime vs. the input power for an RF mixer circuit

Fig. 2.14 shows the comparison of the simulation runtime between the proposed support-circuit preconditioner and the BD preconditioner for an LNA+mixer with two tones of equal power applied to the input. We observe that in low input-power region (RF circuit exhibits weakly nonlinear behaviors), BD preconditioning method is faster than the proposed support-circuit preconditioning method. Since for weakly nonlinear circuits, there are almost no large off-diagonal entries in the block circulant matrices of the HB Jacobian matrix, so the BD preconditioner matrix can well approximate the properties of the original HB Jacobian matrix. However, as the input power increases (circuit system becomes strongly nonlinear), the simulation runtime of the BD preconditioning method and the number of GMRES iterations for each NR iteration will grow dramatically. On the other hand, the proposed support-circuit preconditioning method always results in the similar numbers of GMRES iterations

and achieves desirable nearly-linear runtime and memory efficiency even for these strongly nonlinear RF circuits.

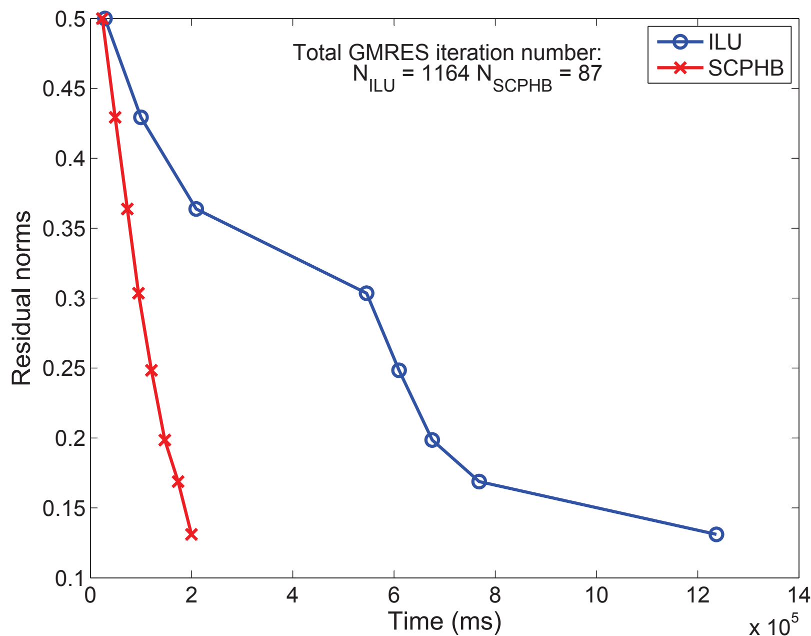


Figure 2.15: Convergence rate/time comparisons of SCPHB and ILU algorithms

Fig. 2.15 compares the performance between our SCPHB solver and the Incomplete LU factorization (ILU) preconditioned GMRES iterative solver, for test case CKT 6, by showing the runtime and numbers of GMRES iterations for the first few Newton-Raphson steps. The ILU preconditioner is built from PETSc [54] with ILU factorization level set to be 5. When the ILU levels are set to be $0-4$, the ILU-preconditioned GMRES solver cannot converge for the first NR step. From the figure, we observe that the proposed SCPHB method can converge much faster than the ILU-preconditioned method with much smaller iteration number and total runtime.

In the last, we would like to demonstrate the accuracy of the proposed support-circuit preconditioning method. Fig. 2.16 illustrates the voltage waveforms of double

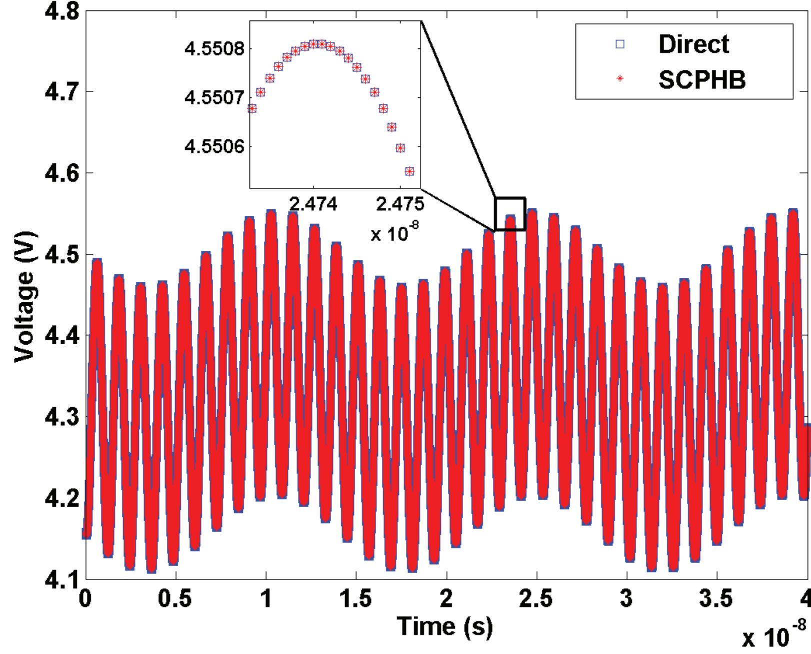


Figure 2.16: Waveform result comparison between direct solution method and proposed iterative method

balanced mixer circuit showing that the results of our proposed support-circuit preconditioning method can accurately match the one obtained by the direct method. Although other preconditioning methods can also achieve the same level of accuracy, our method can usually converge much faster as observed in our experimental results.

2.6.3 Scalability

In this section, we would like to demonstrate the scalability of our proposed method. Fig. 2.17 illustrates the total HB simulation runtime results of various problem sizes (unknowns). Fig. 2.18 shows the peak memory consumptions of different problem sizes. It is obvious that the proposed SCPHB method has much better scalability than direct solution method. In Fig. 2.17, with the increase in the problem size, the runtime cost of direct solution method grows much faster than the proposed method.

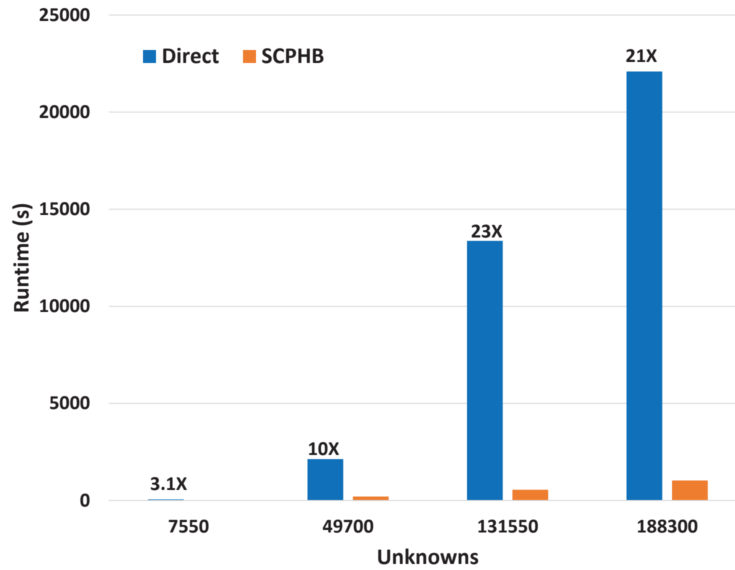


Figure 2.17: Results on runtime scalability

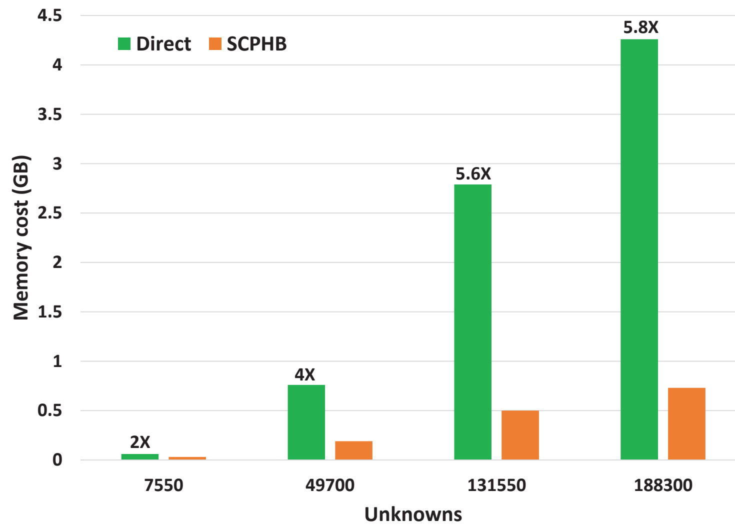


Figure 2.18: Results on memory scalability

We observe $3X$ to $20X$ speedups and memory cost reductions from $2X$ to $5.8X$ from Fig. 2.17 and Fig. 2.18.

Chapter 3

Massively Repeated Small Circuit Simulation on GPU [1]

3.1 Background and Overview

3.1.1 Nonlinear Circuit Simulation Approaches

General nonlinear electronic circuit simulation techniques rely on Newton-Raphson (NR) method to solve the following nonlinear differential equations [36]:

$$f\left(x\left(t\right)\right)+\frac{d}{dt}q\left(x\left(t\right)\right)+u\left(t\right)=0, \tag{3.1}$$

[1] The material contained in this chapter was previously published in "*Proceedings of ACM/IEEE Design Automation Conference (DAC)*" ©2013 ACM. See Appendix A.2 for a copy of the copyright permission from ACM.

where $f(\cdot)$ and $q(\cdot)$ denote the static and dynamic nonlinearities, $x(t)$ is a vector including nodal voltages as well as branch currents, and $u(t)$ is the input excitation vector. Sophisticated numerical methods can be used to solve the above nonlinear differential equations by first linearizing the nonlinear circuit system at a given solution point, and subsequently solving the corresponding linear matrix problems. For instance, after linearizing the system, conductance matrix $G\left(x^k\right) = \frac{\delta f}{\delta x}\Big|_{x^k}$ and capacitance matrix $C\left(x^k\right) = \frac{\delta q}{\delta x}\Big|_{x^k}$ can be easily obtained which are typically asymmetric matrices. The dominant computational cost for solving small circuit problems is mainly due to the nonlinear device evaluations, while for much larger circuits solving the asymmetric Jacobian matrices using direct solution method can be much more expensive due to the exponentially increased runtime and memory cost.

3.1.2 Massively Parallel GPU Computing

Recent Fermi GPU from Nvidia has increased the number of streaming processors (SPs) in each streaming multi-processor (SM) from 8 to 32, boosting the total number of streaming processors to 512[55]. According to CUDA programming model [56], 32 threads are formed into a warp, and will execute the same instruction every four clock cycles, resulting in a very light overhead (one instruction issuing is followed by 32 thread executions). When a kernel function is launched on GPU, the task (data) is further divided into many thread blocks (1D, 2D or 3D) based on the problem size and available on-chip hardware resources. Each thread block may include multiple warps of threads. Subsequently, each SM will work on a few thread (data) blocks with its eight SPs. The new GPU model also supports high performance double-precision computing and concurrent kernel executions. Up to 16 kernels can be launched concurrently on the 16 SMs for Fermi GPUs, while in previous GPU architectures only one kernel can be launched at the same time on GPU, which allows for more flexible

and efficient GPU computing.

GPU's on-chip memory (shared memory and registers) is very fast, but the available on-chip memory resource can be quite limited, whereas the off-chip device memory (global memory) is sufficiently large but can be much slower than on-chip memories. Additionally, coalesced GPU global memory accesses are important since random memory accesses are typically much slower. The device memory bandwidth can be up to $100Gb/s$ if accessed in a coalesced pattern but may also reduce to $10X$ lower if accessed in a random manner [56]. If a random memory access is needed, texture memory on GPU (like the L1 and L2 caches for CPU) should be used, though a good memory access pattern is still desired such that threads of a warp can access the neighboring memory locations.

GPU's hardware and software properties impose the following challenges when designing streaming data parallel computing algorithms: (1) the dependencies among different tasks (data) should be minimized or avoided; (2) excessive global data sharing and shared memory (register) bank conflicts should be eliminated; (3) the arithmetic intensity that is defined as the number of floating point operations per data reading/writing should be maximized; and (4) the algorithm control flow should be simplified.

3.1.3 Overview of our approach

TinySPICE is an SPICE-accurate nonlinear circuit simulator that leverages CPU and GPU for fast repeated small circuit analysis. To leverage the powers of GPU streaming processors (SP) for data parallel computing, we propose a novel GPU massively parallelized algorithm with GPU friendly data structures to accelerate the

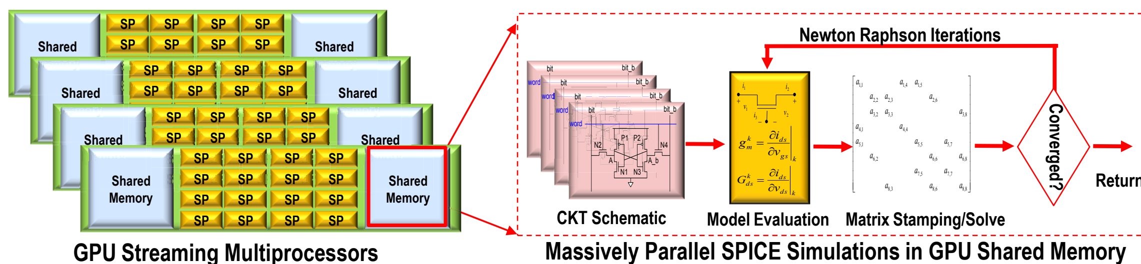


Figure 3.1: TinySPICE: massively parallel SPICE simulation program on GPUs

repeated small nonlinear circuit simulations. For different test circuits, the data format (dense or sparse) will be chosen automatically to optimally utilize the GPU hardware resources.

It is important to note that by running each circuit simulation using one GPU thread, it allows hundreds or thousands of independent simulations executed on GPU simultaneously. Moreover, since limited memory is required for each circuit simulation, the data related to simulation of a single circuit can be entirely stored in GPU's shared memory, as illustrated in Fig. 3.1. By using shared memory and coalesced global memory access, the carefully designed algorithm can achieve extremely high computing throughout. As a result, our TinySPICE can run in a much faster way on GPUs than conventional SPICE simulators, especially for massively repeated Monte Carlo small circuit simulations.

In order to handle relatively large circuit,we propose a series of novel GPU-friendly data structures and memory access strategies for setting up (updating) sparse MNA matrices, as well as a data-parallel sparse matrix solution algorithm flow for solving hundreds of sparse MNA matrices concurrently to significantly boost the computing performance of massively parallel SPICE circuit simulations. In order to reduce computation rounding error during LU factorizations and to avoid GPU threads divergence during NR iterations, a novel circuits clustering procedure is introduced to classify circuits to several groups and extract the common pivoting patterns and data

dependency graphs of LU factorizations for each group.

3.2 Device Evaluation and Stamping on GPU

3.2.1 Device Evaluation on GPU

As introduced in Section 3.1.1, during SPICE simulation, all the devices need to be evaluated to find the relationship between current and voltage in every NR iteration. So, the efficiency of device evaluation is critical to the SPICE simulation, especially for the small circuit designs.

The elements of linear devices such as resistors, capacitors and inductors are with constant values, the corresponding system matrix entries will not change throughout the entire simulation. Consequently, they can be pre-evaluated and stored in a linear system matrix, which can be subsequently combined with the matrix generated from nonlinear devices evaluations.

For nonlinear devices, such as transistors. The traditional BSIM4 model-based evaluations involve many complex device-oriented formulas, which make the nonlinear devices evaluation computational and runtime inefficient.In order to more effectively parallelize the device evaluations on GPU during circuit simulations, a 3D LUT modeling method is adopted in this work.

In order to meet requirements for both accuracy and runtime efficiency, the parametric 3D LUT models will be constructed for evaluating transistors during circuit simulations. LUT-based evaluation of a smooth function derived from the truncated Taylor expansion can be formulated as follows:

$$T_n(x) = f(c) + \sum_{k=1}^{d} \frac{f^{(k)}(c)}{k!}(x-c)^k \tag{3.2}$$

where $f(c)$ denotes the evaluation function and $f^{(k)}(c)$ denotes the k-th order derivatives at reference point c, x is the evaluation point, and d is the degree of the Taylor polynomial. The approximated evaluation can be carried out by looking up a pre-calculated LUT for coefficients associated with $(x-c)^k$. For the second order Taylor polynomial expansion, which means $d = 2$, we can get the second order parametric 3D LUTs evaluation function:

$$\begin{aligned} LUT = LUT_{base} + LUT_{V_{th}} \cdot \Delta V_{th} + LUT_{L_{eff}} \cdot \Delta L_{eff} \\ + LUT_{V_{th2}} \cdot \Delta V_{th}^2 + LUT_{L_{eff2}} \cdot \Delta L_{eff}^2 \\ + \Delta V_{th} \cdot \Delta L_{eff} \cdot LUT_{V_{th}L_{eff}} \end{aligned} \tag{3.3}$$

where LUT_{base} represents the base LUT generated based on the transistor nominal parameters. $LUT_{V_{th}}$ and $LUT_{L_{eff}}$ are the first order coefficient LUTs for transistor threshold voltage and effective channel length respectively. Similarly $LUT_{V_{th2}}$ and $LUT_{L_{eff2}}$ are the second order coefficient LUTs. $LUT_{V_{th}L_{eff}}$ is the coefficient LUT derived from the partial derivative of V_{th} and L_{eff}. In order to reduce the complexity, this cross term is ignored in our implementation. ΔV_{th} and ΔL_{eff} mean the variation of the threshold voltage and effective channel length. So the base LUT and two coefficient LUTs compose the whole parametric 3D LUTs of a transistor. The number and order of coefficient LUT can be adjusted according to the number of critical input parameters and accuracy requirement. However it is not always necessary to introduce the higher order LUTs for each parameter. The proposed TinySPICE only apply second order LUTs to those parameter whose variation greater than a

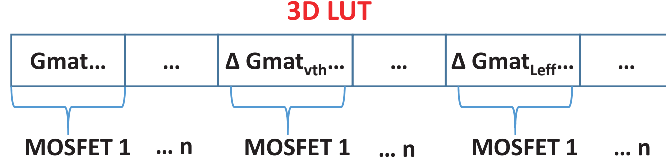


Figure 3.2: Vector for storing 3D LUTs.

threshold. Benefited from the coefficient LUTs that can capture the variations of transistor parameters, we do not need to update the LUTs at every NR iteration. In other words, only one-time data transferring of the parametric transistor LUTs from CPU to GPU is required, which can significantly reduce the overhead of CPU-GPU communication.

The proposed TinySPICE first parses standard SPICE-like circuit netlist, and evaluates the BSIM4 transistor models to build parametric 3D LUTs for all nonlinear transistors. When building the parametric 3D LUTs, we use the ΔV_{th},ΔL_{eff}, V_{ds}, V_{gs} and V_{bs} as the input variables, where V_{ds}, V_{gs} and V_{bs} denote the terminal voltages of MOSFET devices. To get coefficient LUTs, $LUT_{V_{th}}$, $LUT_{V_{L_{eff}}}$,$LUT_{V_{th2}}$ and $LUT_{V_{L_{eff2}}}$ are also calculated after generating the LUT_{base}. The parametric 3D LUTs outputs include all the required elements for stamping the conductance and capacitance matrices obtained from linearizing (3.1) during the simulations, such as conductance, capacitance, currents and charges. Output elements can be obtained based on the input voltages of the transistor terminals.

After extracting all the data required by these parametric 3D LUTs using thousands of BSIM4 model evaluations, we store all the data in a long vector to allow GPU's coalesced device memory accesses, as shown in Fig. 3.2. Considering the huge amount of data (more than forty elements) computed in one transistor evaluation, we store the data in such a way that good data locality can be well preserved to ensure GPU's efficient texture memory accesses during LUTs' trilinear data interpolations using neighboring eight points.

Since device evaluations using 3D LUTs are based on eight-point trilinear data interpolations, device evaluated by LUTs requires much less computation time than the BSIM4 model evaluations that involve very complex device-based formulas. Consequently, even the LUT-based method can achieve a faster device evaluation when running on CPU. We observe that for most digital circuit modeling and analysis applications, the accuracy level obtained using LUT-based SPICE simulator (with first-order parametric LUTs) is very satisfactory, though for analog circuits the convergence may become more difficult.

It needs to be noted that generating LUTs from BSIM4 transistor models is difficult to realize on GPU. Therefore, LUTs need to be created on CPU and then transferred to GPU at the very beginning of the simulation. Moreover, traditional LUT model suffers from several limitations. For instance, the direct LUT is generated based on nominal transistor parameters. However, due to the impact of process variations, transistor parameters such as effective channel length L_{eff} and threshold voltage V_{th} may deviate significantly from their nominal values. As a result, direct LUT needs to be updated frequently on CPU according to varying transistor parameters that could lead to communications of high frequency. It is also known that frequent data transfer between CPU and GPU can result in large latency and less runtime efficiency due to the limited bandwidth of the PCI bus.

3.2.2 Jacobian Matrix Data Format and Stamping on GPU

3.2.2.1 Dense Jacobian Matrix and Stamping

Dense matrix format supports easy direct access to its elements. By using dense matrix format, the matrix solver algorithm can factorize the matrix in a straightforward

way without considering the fill-ins during the factorization. This simple algorithm flow is very suitable for GPU's single-instruction-multiple-thread (SIMT) architecture. The disadvantage of dense matrix format is the extravagant computation cost and memory consumption introduced by storing and processing the zero elements. Since the memory consumption of very small circuit analysis is typically very low, those extra costs will not be significant.

It should be noted that, in order to obtain stamping locations of nonlinear elements in the system matrix, it is necessary to store the terminal indices of each nonlinear device. In this work, we propose to store terminal indices of all transistors in a long index-mapping vector, as shown in left side of Fig. 3.3. In the Mos_map vector, "Idx" stands for the corresponding LUT storage index of a transistor. "P/N" is a flag indicating PMOS or NMOS. "d,g,s,b" represent the index of each transistor's terminal in the system matrix respectively. With such information, device evaluation results from LUTs can be directly written into the system matrix as well as the right hand side vector (RHS).

3.2.2.2 GPU Sparse Jacobian Matrix and Stamping

Dense matrix data format can be a good choice for very small circuit simulations, since the entries of dense matrix can be accessed according to the index information in a rather straightforward manner. However, with the increasing circuit size, dense matrix format may result in rapidly growing memory and computation cost, though typical circuit MNA matrices are rather sparse, even after being factorized by direct matrix solvers [48], which motivates us to consider sparse matrix data format and solution techniques to dramatically improve the scalability of GPU-based SPICE simulators.

To this end, the proposed TinySPICE simulation engine leverages the Compressed Sparse Column (CSC) sparse matrix format for storing the nonzero entries of the Jacobian matrix during a SPICE simulation. The CSC sparse matrix stores all necessary data into three one-dimensional vectors including the column pointers, row indices, and entry values. To enable the CSC matrix format in TinySPICE, novel matrix stamping and LU solution steps on GPU will be introduced in the following sections.

Since linear devices such as resistors, capacitors, and inductors have constant values, their corresponding entries in the Jacobian matrix just need to be stamped once on CPU. However, nonlinear device evaluation results need to be updated and stamped into the Jacobian matrix during every NR iteration. Fig. 3.3 demonstrates the stamping procedure of a transistor, where the "O" represents the stamping entry for the transistor and "X" denotes stamping entries for other devices. Unlike the dense matrix format, the CSC sparse matrix format does not allow efficient access to the entry locations given the information of terminal indices. For instance, in order to obtain the nonlinear device stamping locations, it is necessary to traverse all the row indices for a specific column. To avoid the above frequently repeated and costly traversal procedure during device stamping procedures, we propose to store the entry indices of a value array in a mapping vector that includes all necessary MOSFET stamping locations, as shown in Fig. 3.3. With this mapping vector, GPU threads can directly read/write the corresponding nonlinear device evaluation results from/to the CSC sparse matrix.

It should be noted that when a transistor terminal connects to ground (reference) node, the related evaluation result does not need to be stamped into the Jacobian matrix using traditional MNA matrix stamping method. However, in order to avoid branching instructions executed by GPU's SPs for checking the grounded connections, these terminals are always treated as regular nodes connected to ground through a

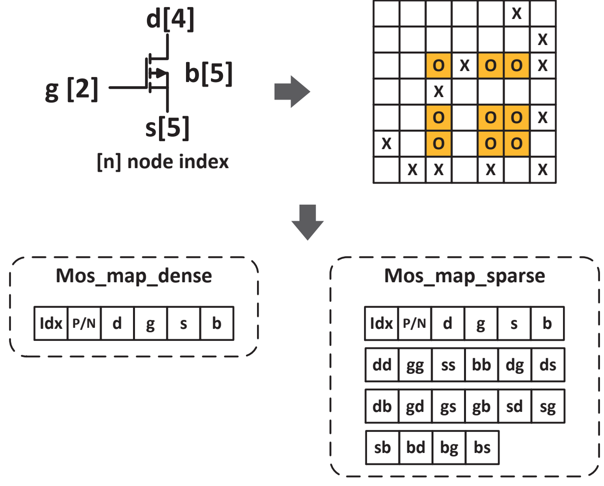


Figure 3.3: MOSFET stamping location map for dense and sparse matrix format.

zero voltage source.

3.2.3 RHS and Excitation Sources

The voltage and current excitation source values will also be extracted and stored in memory before the simulation start on GPU. The source with constant value will be stamped into RHS vector directly. The time-varying excitation source will be stored in vectors shown in Fig.3.4.

In VS_map vector, node a and node b denote regular terminal nodes' indices. In modified nodal analysis (MNA) [36], each voltage source requires including a Pseudo node into the index-mapping vector for representing the current flowing through the device.

VS_step vectors including all the values of time-varying voltage and current sources at each time step of transient simulations. VS_step will be then combined with constant excitation vector to form the final RHS vectors.

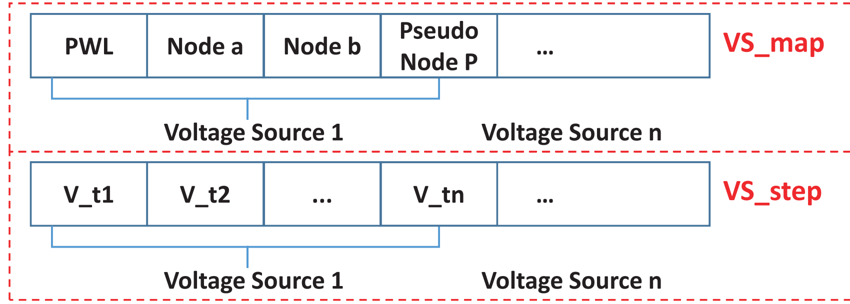


Figure 3.4: Vectors for storing excitation sources on GPU.

3.3 Matrix Solver on GPU

As mentioned in Section 3.2.2.1, with dense Jacobian matrix the classical direct LU solver can be easily implemented on GPU such as Doolittle algorithm[57]. Without considering fill-ins during LU factorization, the solver is able to avoid the GPU threads branch divergence.

For relatively large circuits, directly solving the sparse MNA matrices using LU algorithm can still be a very costly procedure during SPICE simulations. Existing work on GPU-based sparse matrix solver research mainly focuses on developing parallel LU factorization algorithms for solving a single large sparse matrix system using many GPU threads [47]. On the other hand, the proposed TinySPICE aims to utilize each single GPU thread to work on a circuit simulation task, so that many sparse Jacobian matrices can be factorized by thousands of GPU threads concurrently. To this end, we proposed a massively parallel sparse matrix solver that is enabled by a novel GPU-friendly LU algorithm flow, allowing to avoid GPU thread divergence effects that are normally unavoidable for prior GPU-based LU solution techniques.

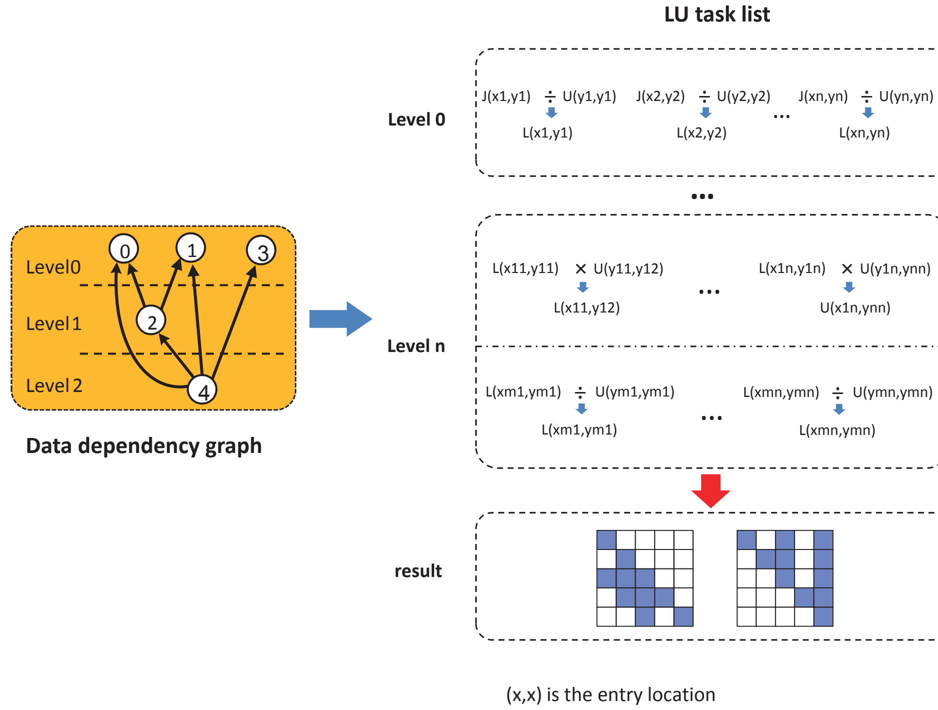


Figure 3.5: Levelized LU factorization task list.

3.3.1 GPU-based Levelized LU Factorization

In Chaper2 Section 2.3, a block sparse matrix LU solver on GPU is proposed to solve the huge harmonic balance analysis Jacobian matrix. Although the statistical simulation problem is quite smaller than the HB Jacobian matrix, the only difference between these two matrices in the structure is the matrix entry data structure. For HB Jacobian matrix, the entry is a dense matrix block. However for statistical simulation problem, the entry is just a scalar. As a result, we can use the similar LU algorithm by replacing the matrix multiplication and division operations with scalar operations shown in Fig.3.5.

Fig. 3.6 demonstrates the GPU data structure for storing the LU factorization task list, where “Level count” represents task level index in the DDG. For each task level,

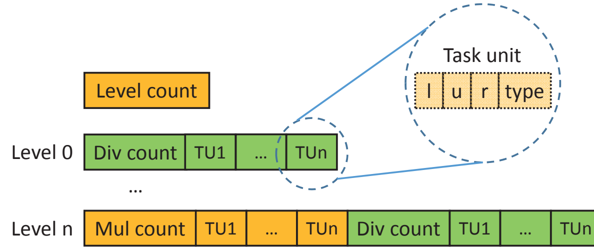


Figure 3.6: GPU data structure for the LU factorization task list.

there are two task lists related to multiplication and division operations respectively. "Div count" and "Mul count" denote the division and multiplication task numbers in the current task level. "TUx" represents a specific task unit "x", which includes the location information of the elements that involve computations. "l", "u" and "r" are the element indices in the value array of the CSC format sparse matrix, while "Type" indicates whether it is an "L" or "U" matrix factor. By keeping the value indices of the elements instead of the original coordination information, the traversal procedures for locating the value index can be avoided when running on GPU. We want to emphasize that for each task level, the computation order of the two task lists are also strictly enforced. For example, the division tasks have to wait for the completions of multiplication tasks.

3.3.2 Circuits Clustering

From the LU algorithm analysis in Section 2.3 of Chaper2, it is clear that the computation sequence required by the LU algorithm is depending on the pivoting pattern and the nonzero locations indicated by L and U matrix factors. In statistical SPICE circuit simulations, all simulation tasks will result in exactly the same nonzero entry locations for all system Jacobian matrices. The main differences are due to the stamping elements associated with nonlinear devices influenced by different input parameter variations. As a result, for circuits with dramatically different parameter

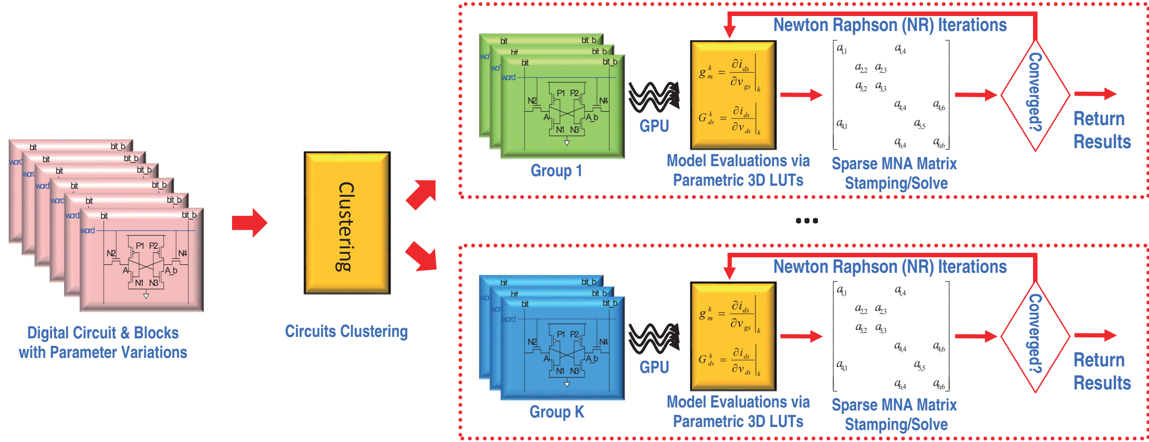


Figure 3.7: Circuit clustering.

variation settings, it is very likely that the required pivoting patterns and nonzero entry locations during LU factorizations are quite different. In this case, if same pivoting patterns and nonzero locations are used during LU factorizations, it may result in large rounding errors or even simulation failures (e.g. divergent NR iterations) when solving the corresponding dramatically different linearized systems.

In order to assure the robustness of the proposed sparse LU solver for massively parallel statistical SPICE simulations, we introduce a simple yet effective circuit clustering procedure that will group similar circuits together before performing statistical SPICE simulations, as shown in Fig. 3.7. The proposed circuit clustering procedure has also been described in Algorithm 7. The classical $k - means$ method is adopted in this proposed work for classifying the circuits with different variation parameters, such as effective channel length (L_{eff}), threshold voltage (V_{th}), and supply voltage (V_{DD} and V_{SS}). As a result of this circuit clustering procedure, within each circuit group, the operation regions of nonlinear devices (transistors) should be quite similar, and the linearized system matrices will also be very close to each other, which allows the common pivoting pattern obtained from the centroid parameter vector to be suitably applied for robust and accurate LU factorizations during large number of statistical SPICE simulations within the same group.

Algorithm 7 TinySPICE Circuits Clustering Algorithm

1: Obtain the set of data describing parameter variations.
2: Cluster the parameter variations into K groups.
3: **for** $i = 1 \rightarrow k$ parameter groups **do**
4: 1. Get the centroid parameter vector $P[i]$.
5: 2. Generate the Jacobian matrix $J[i]$ based on $P[i]$.
6: 3. Factorize $J[i]$ using LU algorithm.
7: 4. Obtain the pivoting vector $Piv[i]$ and the nonzero entry pattern $Nnz[i]$ of L and U matrix factors.
8: 5. Create the LU factorization task list $LUTask[i]$ for parameter group i.
9: **end for**

3.4 GPU Optimization

3.4.1 Data Allocation and Access Optimization

CUDA devices have several types of memories that exhibit different data access latencies and bandwidths which may greatly influence the GPU kernel execution performance. To more efficiently handle relatively large circuits using GPU's on-chip memory resources, a series of GPU-friendly data structures, as well as memory allocations and accesses have been carefully designed and optimized for TinySPICE. We summarized the proposed GPU memory allocation strategy for TinySPICE as follows.

† *Texture memory:* The parametric 3D LUTs are read-only for all the GPU threads, so it is preferred to store them in GPU's read-only texture memory to reduce the memory data access latency.

† *Shared memory:* The initial RHS vectors, linear system matrices, index-mapping vectors for transistor and voltage sources, pivoting vectors and GPU LU task list are shared among all GPU threads, so they are stored in GPU's

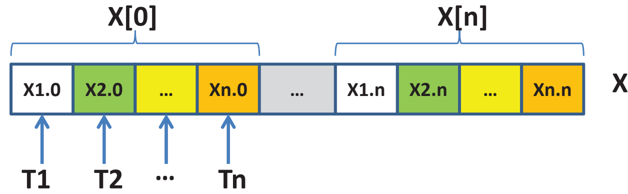


Figure 3.8: The solution vector data access pattern on GPU.

shared memory.

† *Global memory:* The device parameter vector, value array of the Jacobian matrix (sparse matrix format), RHS vector and solution vector need to occupy a large amount of memory for both read and write operations, so they are stored in GPU's global memory and carefully allocated for coalesced memory accesses during the simulations.

In order to obtain the best parallel GPU computing performance, coalesced device memory (global memory) accesses should be satisfied for all GPU threads. Take the solution vector as an example, to enable coalesced device memory accesses, we organize the memory storage of solution vectors in such a way, that for total n circuits, the memory spaces of all the n solution vectors are continuous, as shown in Fig. 3.8, where T_k denotes the k-th GPU thread, and $xi.m$ denotes the m-th element of solution vector of circuit i. This GPU-friendly data storage obviously allows for efficient coalesced global memory accesses, which can significantly reduce the GPU device memory access overhead.

3.4.2 Thread Organization

Since each GPU's streaming multiprocessor (SM) has very limited memory resources, the number of circuits to be analyzed at the same time should be carefully determined based on the circuit sizes and on-chip memory usage (e.g. Nvidia GeForce GTX480

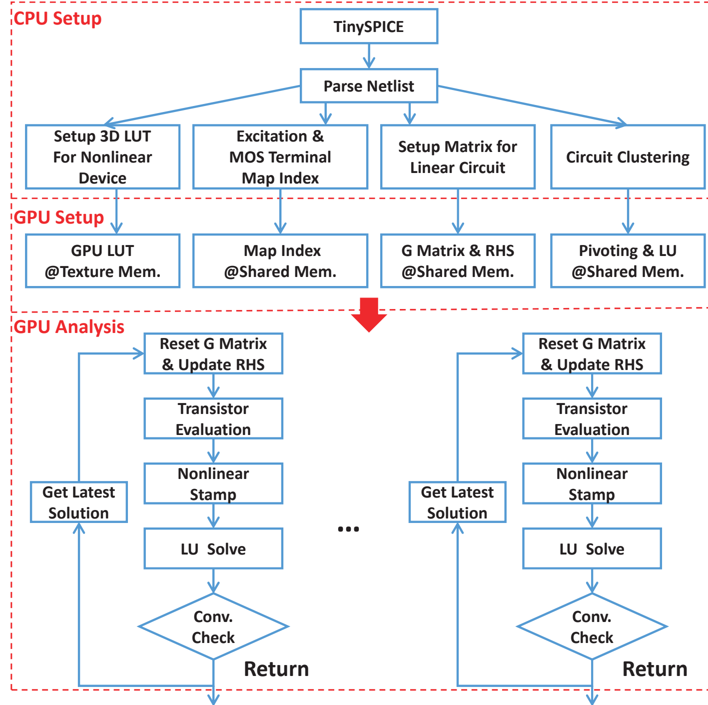


Figure 3.9: The algorithm flow of TinySPICE.

GPU has 15 Multiprocessors, 48k shared memory and 32k registers per SM). The limited memory can impact the number of GPU threads running on each SM. To achieve the best simulation performance, TinySPICE first finds out the optimal thread block sizes and grid sizes by evaluating simple memory-cost functions (for computing the maximum number of circuits that can be analyzed in one SM). Then the proper thread organization and assignment can be determined, and final simulation code can be compiled for a given circuit design. It is worth noting that different circuit analysis problems may result in different GPU thread settings, and therefore different speedups compared to CPU-based SPICE simulations.

3.4.3 Jacobian Matrix Format Determination

As described in Section 3.2, for different circuit size, TinySPICE will adopt different data format for the system matrix. For various circuit design, the memory and computation requirement can be quite different. And different GPU device also has different memory and computation capability (e.g. GTX480 GPU has 480 CUDA cores and 1.5GB device memory, Tesla C2075 has 448 CUDA cores and 6GB device memory). As a result, the best matrix format for different test cases and GPU devices can be quite different. To achieve the best simulation performance, TinySPICE choose the matrix format by examining the sparsity and dimension of the system matrix. If the system matrix itself is very dense, the dense matrix format is preferred to take the advantage of easy data access and simple solver algorithm. On the contrary, if the system matrix is very sparse and has large dimension, sparse matrix format is a better choice, which can reduce the computation and memory cost significantly. If system matrix is very sparse but has very small dimension, dense matrix format still would be preferred to avoid the complex algorithm flow of the sparse matrix solver.

3.5 Algorithm Flow for TinySPICE

3.5.1 CPU and GPU Cooperation

The complete procedures of TinySPICE simulator can be summarized into three major phases: CPU setup phase, GPU setup phase, and GPU analysis phase, as illustrated in Fig. 3.9, where G denotes the system matrix, and RHS stands for right hand side vector.

3.5.1.1 CPU Setup Phase

The main task of the CPU setup phase is to prepare GPU-friendly data structures for SPICE simulations on GPU.We conclude the CPU setup phase for TinySPICE as follows:

† Build parametric 3D LUTs for all transistors according to a user-defined accuracy level. A suitable discretization step size can be selected based on the circuit design information and specific simulation requirements. More accurate LUTs typically require larger memory space and characterization time. The parametric 3D LUTs are stored in a long 1D vector (as shown in Fig. 3.2) that will be transferred to GPU's device memory for one time before the simulation starts.

† Create the 1D terminal index-mapping vectors Mos_map to store the node indices for all nonlinear devices, as shown in Fig. 3.3. Mos_map is used to help stamp nonlinear devices into the system matrices that only needs to be constructed and transferred to GPU for one time.

† Create the linear system matrices by stamping all linear devices. The linear system matrices will also be stored in a 1D vector and sent to GPU memory. Once GPU kernel functions are launched, linear system matrices will be loaded to GPU's shared memory at the initial step and will be combined with the nonlinear device evaluation matrices to form the final system matrices for subsequent NR iterations.

† Create the VS_map vectors including information for all excitation sources such as voltage and current sources.

† Create the VS_step vectors including all the values of time-varying voltage and

current sources at each time step of transient simulations.

† Classify the test circuits to several groups and obtain the pivoting vector and LU factorization task schedule for each group.

3.5.1.2 GPU setup and analysis phase

The main task of the GPU setup phase is to prepare proper simulation environment for the subsequent circuit analysis on GPU, which includes device memory allocations, and data transmission from host (CPU) to device (GPU). The SPICE simulation will be performed in GPU analysis phase which will be described in the following sections.

3.5.2 NR Iteration algorithm on GPU

The NR algorithm flow of TinySPICE is summarized in Algorithm 8. At the beginning of each NR iteration, TinySPICE evaluates all of nonlinear devices (linearize the system) using LUT-based trilinear interpolations according to the latest solution results. After the device evaluations, the computed elements of nonlinear devices are stamped into nonlinear system matrices based on the terminal indices stored in the index-mapping vectors. RHS vectors also need to be updated based on the latest solution results. After building the nonlinear system matrices and RHS vectors, the final system matrices can be created by combining the nonlinear system matrices with the linear devices matrices that have been built from the very beginning. Subsequently, GPU-based LU decomposition algorithm is applied to factorize the system matrices.

It should be noted that, in order to reduce GPU thread divergence considering GPU's single-instruction-multiple-thread (SIMT) scheme, the convergence condition is not

Algorithm 8 Newton-Raphson (NR) Iteration Algorithm Flow on GPU

Allocate system matrix and RHS in registers for each GPU thread.
Load linear system matrix, RHS vectors, index-mapping vectors from GPU's texture memory to shared memory.
for $i = 1 \rightarrow n$ NR iterations **do**
 1. Reset system matrix and RHS vector by loading initial data from shared memory.
 2. Evaluate nonlinear devices.
 3. Stamp system matrix and compute the RHS vector.
 4. Factorize system matrix of k-th circuit and solve for the solution vector X_k.
 5. Apply a damping factor for the solution ΔX_k if needed.
end for
if NR does not converge **then**
 Perform another n iterations of steps 1-5.
end if
Return solution if NR converged. Otherwise return an error flag.

checked at every NR step. Instead, we check the convergence after several NR steps. Although this method will result in an overhead of NR steps, it may efficiently reduce the divergence issue of GPU threads.

3.5.3 DC Simulation Flow

The DC simulation algorithm flow of TinySPICE is demonstrated in Fig. 3.10. The preprocess on GPU for dense matrix format(left side) and sparse matrix format (right side) are quite different. In common, the parametric $3D$ LUT, index-mapping vector, linear Jacobian matrix will be constructed no matter which matrix format is adopted. For sparser matrix format, TinySPICE will first classify the test circuits to several groups. Then pivoting vector and LU factorization task list will be generated for each group.

In the GPU simulation step, for the beginning of each NR iteration, TinySPICE will first evaluate all of nonlinear devices using LUT-based trilinear interpolations according to the latest solution results. Next, the computed elements of nonlinear

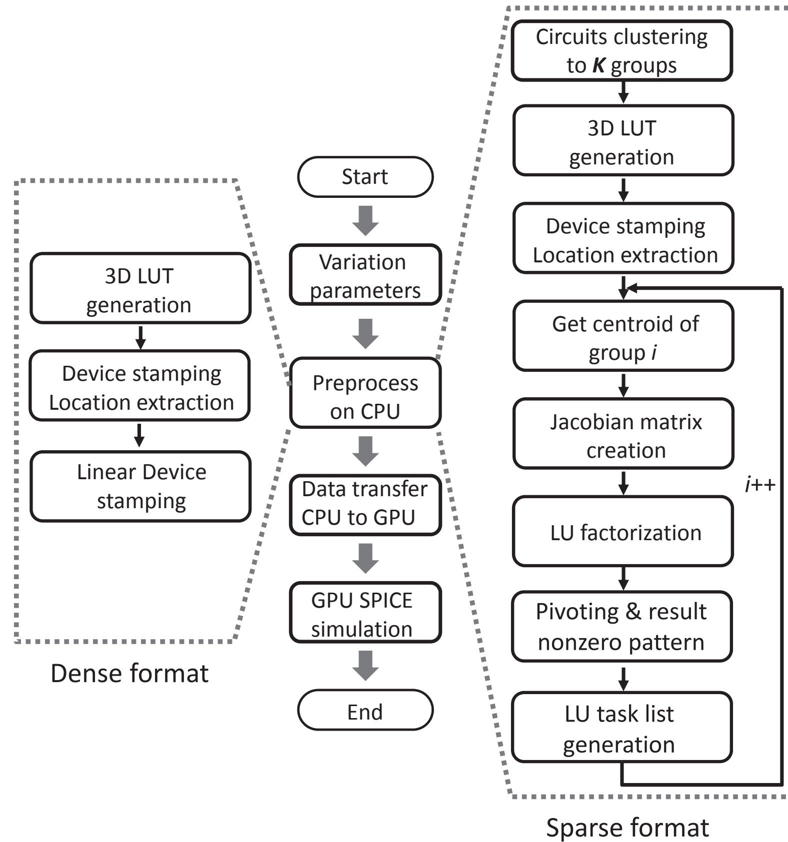


Figure 3.10: The DC simulation flow of TinySPICE.

devices are stamped into nonlinear system matrices based on the entry indices stored in the index-mapping vectors, while the RHS vectors will also be updated based on the latest solution results. Subsequently, the final system matrices will be created by combining the nonlinear system matrices with the linear device matrices that have already been built in advance, which is followed by LU factorization and solution procedures for solving linear systems of equations.

3.5.4 Transient Simulation Flow

The proposed TR simulation algorithm flow of TinySPICE is shown in Fig. 3.11, which first performs DC operating point simulations to get the initial circuit operating

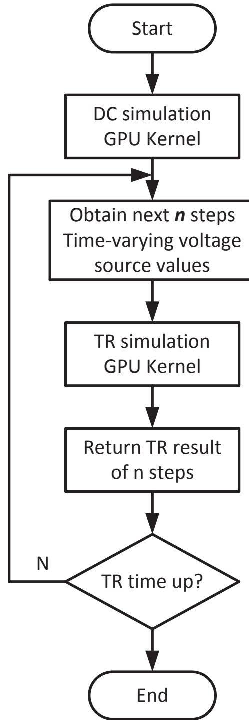


Figure 3.11: TR simulation algorithm flow.

conditions. The results of DC simulations are kept in the global memory of GPU. As introduced in Section 3.4.1, the time-varying voltage sources values are stored in the shared memory. Since it is impractical to store the voltage values for all TR simulation time steps in GPU's shared memory, TinySPICE will keep only a small number of steps (e.g. $n = 10$) of TR simulations during each GPU kernel launch. The number of steps to be kept in the shared memory can be determined by examining the available memory resources as well as circuit size. For instance, when choosing a smaller n, the simulator can handle much larger circuits, while resulting greater overhead due to the increased number of GPU kernel launches.

3.6 Experiment Result

3.6.1 Experimental Setup

In the experiments, several widely used digital circuits have been tested by TinySPICE on GPU. To demonstrate the benefit of our GPU-based TinySPICE simulator, traditional CPU-based SPICE simulation methods and TinySPICE on CPU are implemented and evaluated. Detailed characteristics of test cases are summarized in Table 3.1, where "NL_Num" denotes the number of nonlinear devices, "Node_Num" represents the number of nodes in the circuit, "Vs_Num" represents the number of independent voltage sources, and "Unk_Num" denotes the number of unknowns of the nonlinear system. We set up both the first order and second order parametric 3D LUTs in our experiments. Those LUTs have been tested using different resolutions. Throughout the following experiments, we use a high LUT resolution to guarantee that the final solution of TinySPICE is matching the SPICE solution. Under the high resolution, the direct and first order LUTs totally cost 27MB memory for a single transistor, while using the second order LUTs will double the memory cost. It should be also noted that, in the experimental results, the LUTs setup time is not included. Averagely generating the direct and first order LUTs for a single transistor costs 0.435s, while using second order LUTs will double the setup time cost. Compare with the whole simulation runtime, the LUTs setup time is much smaller and it is a one time cost. Furthermore, the LUTs generation process can be easily parallelized to reduce the LUTs setup time. Since the accuracy level with first order parametric LUTs is very satisfactory, all the following experiments use the first order LUTs to reduce the memory and runtime cost. The SRAM array circuits are tested on RHEL 6.6 64-bit with 2.66GHz 12-core CPU, 48GB DRAM memory and Tesla C2075 GPU

Table 3.1
Experimental setup of test cases.

Circuit	NL_Num	Node_Num	Vs_Num	Unk_Num
6T-SRAM	6	8	5	12
D-Latch	8	9	5	13
D-Flip-Flop	16	12	5	16
Invertor-Chain	32	20	3	22
4:1 Mux	24	27	9	35
SRAM Array 1	60	36	15	50
SRAM Array 2	120	65	25	89
SRAM Array 3	180	95	35	129
SRAM Array 4	240	125	45	169

with $5GB$ device memory. The experiments of rest circuits have been performed on Ubuntu8.04 64-bit with 2.66GHz quad-core CPU, $6GB$ DRAM memory, and one Nvidia GeForce $GTX480$ GPU with $1.5GB$ device memory.

3.6.2 Experimental Results

3.6.2.1 Accuracy of Parametric 3D LUT

The circuit for the static random access memory (SRAM) cell is simulated to show the accuracy of parametric 3D LUTs. For each test, we sweep the input from 0 to VDD. At each sweep point, 1000 ΔV_{th} and ΔL_{eff} are generated randomly and separately for each transistor following a normal distribution. For the normal distribution, parameter nominal values are chosen as the mean value, and the 10% of the nominal values are set to be the standard deviation σ. 1000 circuit DC simulations are carried out at each sweep point. CPU-based simulator using BSIM4 model evaluations generates the reference results, and will be compared with TinySPICE simulators implemented for CPU and GPU computing platforms.

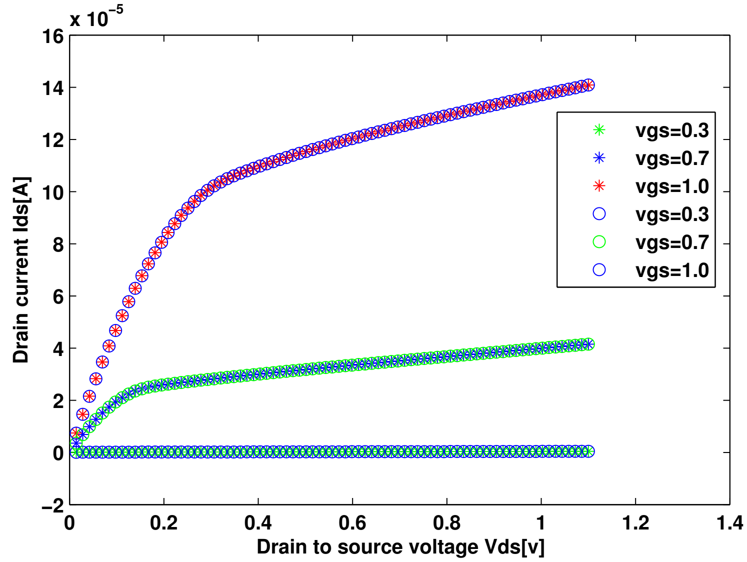


Figure 3.12: The I-V characteristics obtained by parametric 3D LUT and Bsim4 model evaluations. Circles denote the LUT evaluation results.

Fig. 3.12 shows the I-V characteristics simulation result of NMOS transistor. In the figure, asterisks represent the I-V characteristics using Bsim4 model evaluations, and the circles represent the results obtained using parametric LUTs. As observed, the results obtained from parametric LUTs model are very close to the results generated using Bsim4 models. In our experiment, several different V_{gs} values are chosen, such as 0.3,0.7,1.0, to show the accuracy.

Fig. 3.13 demonstrates the DC simulation results (for an internal node voltage) of the parametric SRAM analysis. The solid line in red is the base line. The results show that our TinySPICE simulator matches well with the original SPICE simulator, and can capture the parametric variations accurately. The average relative error is measured as 0.29%. The second order LUTs have also been tested for DC simulation. The average relative error has dropped to 0.289%.

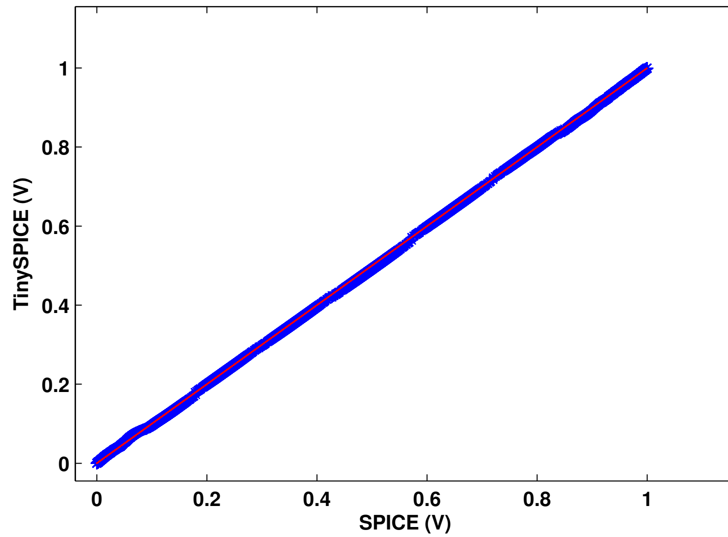


Figure 3.13: Scatter plot of the DC simulation results for SRAM circuits obtained by TinySPICE and the original Bsim4 SPICE simulator.

Table 3.2
DC Simulation Runtime Results of TinySPICE with Dense Format

Circuit	CPU BSIM4(s)	CPU LUT(s)	GPU LUT(s)
6T-SRAM	768.153	403.200	2.902(264X)
D-Latch	1212.979	527.155	5.727(211X)
D-Flip-Flop	2027.827	982.579	10.677(189X)
Invertor Chain	4377.600	1981.440	41.863(104X)
4:1 Mux	3686.400	1812.480	81.366(45X)

Table 3.3
Transient Simulation Runtime Results of TinySPICE with Dense Format

Circuit	CPU BSIM4(s)	CPU LUT(s)	GPU LUT(s)
6T-SRAM	30720	7679.82	163.59(187X)
D-Latch	41472	10751.95	186.42(222X)
D-Flip-Flop	69120	18432.15	341.3(202X)
Invertor Chain	121344	41472	755.76(160X)
4:1 Mux	256512	33792.15	2658.3(96X)

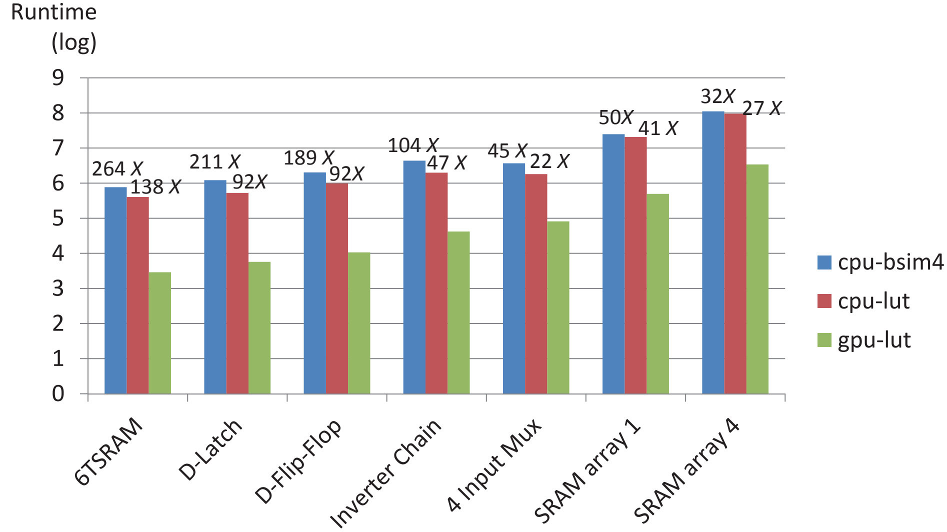


Figure 3.14: Comparison of DC Simulation Runtime

3.6.2.2 Runtime Results

First, we show the DC and transient simulation runtime results of our TinySPICE by comparing them with the results obtained by CPU-based simulators. The runtime results of all simulators are obtained by running $1,536,000$ simulations of different circuits with different excitations and circuit design parameters.

The runtime results by using dense matrix format are illustrated in Table 3.2 and 3.3, where "CPU-LUT" denotes the runtime for LUT-based SPICE simulation on CPU, "CPU BSIM4" denotes the runtime for SPICE simulation with BSIM4 models on CPU, "GPU-LUT" denotes the runtime for proposed TinySPICE on GPU. Speedups are calculated by comparing to the "CPU BSIM4". We can observe that CPU-based SPICE simulator using LUTs can achieve up to $2X$ speedups for DC simulations and $7X$ speedups for transient simulations, compared to traditional SPICE simulator "CPU BSIM4". The reason is that the device evaluation cost for parametric 3D LUTs interpolation is much cheaper than the evaluation of BSIM4 models. Moreover, compared to CPU-based SPICE simulator using LUTs, when doing DC simulation using

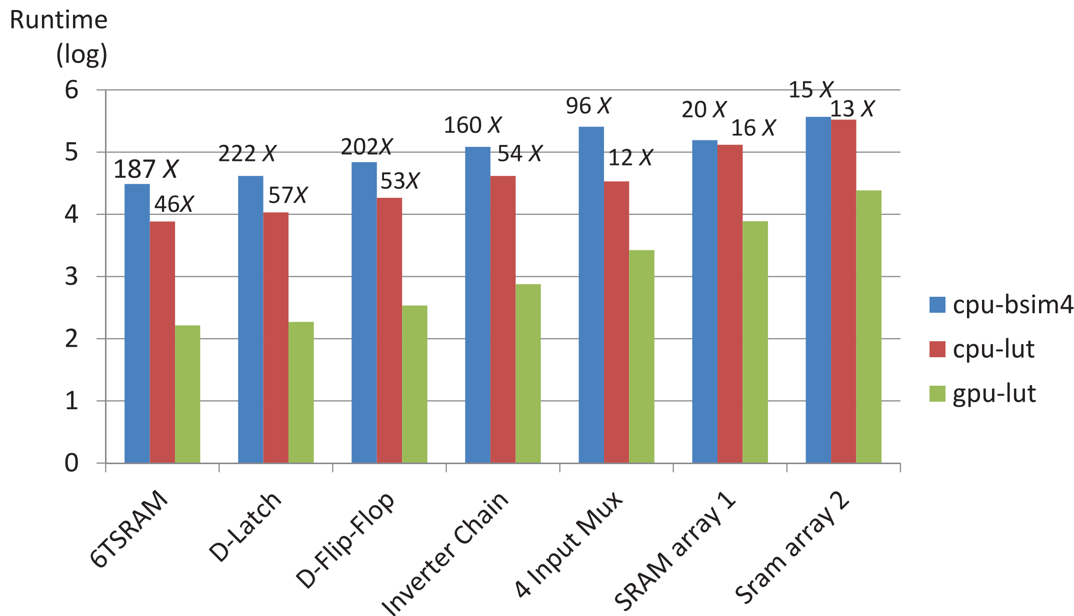


Figure 3.15: Comparison of Transient Simulation Runtime

TinySPICE on GPU, we can achieve up to 138X speedups. Thus, TinySPICE on GPU runs 264X faster than traditional SPICE simulator. For transient simulations, TinySPICE on GPU runs 222X faster than the traditional SPICE simulator(shown in Fig. 3.15). It should be noted that, once the circuit problem size increases, the memory consumption of each GPU thread will also increase. As a result, the total number of GPU threads will decrease due to the limited GPU on-chip memory resources, such as registers and shared memory. For instance, the "4:1 Mux" test case has 35 unknown variables, and the speedups obtained by GPU is only 22X in DC simulations,as illustrated in Fig. 3.14. This corresponds to a much lower simulation performance on GPU than the result obtained from the "6T-SRAM" circuit that has only 12 unknown variables. By using sparse matrix format, TinySPICE can handle circuits including up to 240 transistors and get a 30X speedup.

Table 3.4 and 3.5 present the runtime results of TinySPICE with sparse matrix format. By adopting the optimized share-memory based sparse matrix format and solver, TinySPICE is able to handle circuits with up to 240 transistors and still get a 30X

Table 3.4
DC simulation runtime results of TinySPICE with sparse format.

Circuit	CPU BSIM4(s)	CPU LUT(s)	GPU LUT(s)	Speedup
SRAM Array 1	24835.5	20721	493.7	50.3X
SRAM Array 2	52402.5	43645.5	1005.2	52.1X
SRAM Array 3	80373	67201.5	2563.1	31.3X
SRAM Array 4	111069	94909.5	3420	32.4X

Table 3.5
Transient Simulation Runtime results of TinySPICE with sparse format

Circuit	CPU BSIM4(s)	CPU LUT(s)	GPU LUT(s)	Speedup
SRAM Array 1	155632	131608	7742	20.1X
SRAM Array 2	368853	333654	24266	15 X

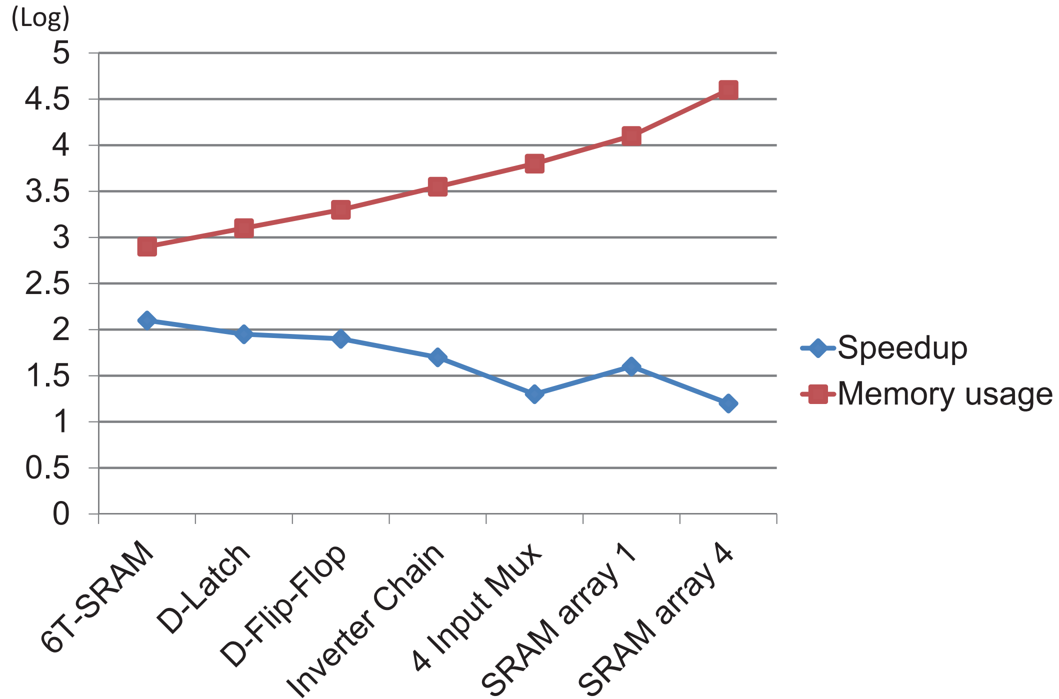


Figure 3.16: Memory usage (shared memory + registers) vs. speedups

speedup.

In the following, the relationship between the runtime speedups and GPU on-chip memory consumption (shared memory and registers) will be analyzed. As illustrated

in Fig. 3.16, the blue curve denotes the memory usage for different circuits, and the red curve denotes the speedups of GPU-based SPICE simulator using LUTs obtained by comparing with CPU-based LUTs SPICE simulator. We observe that, when the number of unknowns of a circuit increases linearly, the memory consumption will dramatically increase for dense matrix format, which is due to the storage requirement of the dense system matrix. By adopting sparse matrix format, the performance of TinySPICE turn better for larger circuits. Obviously, for each GPU thread, the dominant on-chip GPU memory is consumed by the system matrices. Since the total memory available for each SM is limited, once more on-chip memory is consumed by a single GPU thread, much fewer GPU threads can be assigned onto a GPU's SM. As a result, the GPU computing resources may not be fully utilized or there may not exist enough active GPU threads, which in turn dramatically reduce the runtime speedups.

Chapter 4

Conclusion and Future Work

4.1 Conclusion of the dissertation

This dissertation presents algorithms for scalable integrated circuit simulation on heterogeneous parallel computing platforms. Two major circuit modeling and analysis problems have been addressed:

1. A framework for accelerating the harmonic balance analysis on heterogeneous CPU-GPU computing systems has been proposed. The proposed method allows to adaptively balance the computational tasks of HB analysis between CPUs and GPUs by optimally sparsifying the HB Jacobian matrix preconditioner. By leveraging a novel transient-analysis guided graph sparsification approach, nearly-optimal support-circuit preconditioners can be obtained. Extensive experiment results show that our HB solver can achieve up to $20X$ speedups and $5X$ memory reduction when compared with the state-of-the-art parallel direct solution method highly optimized for multicore CPUs.

2. A graphics processing unit(GPU) accelerated massively parallel SPICE-accurate nonlinear circuit simulation engine has been proposed for efficient parametric embedded memory and standard cell array analysis. By accelerating the entire flow of SPICE simulation algorithm on GPU's on-chip memory, such as shared memory and registers, and employing parametric 3D LUTs, SRAM yield analysis and standard cell variation-aware characterization can be performed in a much faster way than ever before. Compared with standard CPU-based SPICE simulation engines, our extensive experimental results show that TinySPICE simulation engine achieves up to $264X$ speedups for parametric SRAM simulations, and more than $150X$ speedups for standard cell simulations, without sacrificing solution accuracy. By introducing the GPU-friendly sparse matrix data format and solution algorithms, TinySPICE is capable of dealing with larger circuits with up to 240 transistors. Additionally, a novel statistical circuit clustering procedure is proposed to dramatically improve the robustness of the proposed massively parallel sparse LU algorithm.

4.2 Future Work

There are several research directions to be pursued in the future to extend or improve the work of this dissertation.

1. The sparse block solver can be improved. The proposed methods in this dissertation leverage cuBLAS for the matrix block multiplication and division batch operations on GPU. Although cuBLAS can efficiently perform the batch operations, this general purpose library can not take advantage of unique properties of circulant matrix. In fact, the multiplication and division of circulant matrix can be performed by vector operation and shifting. As a result, an enormous

amount of multiplication and division operations can be saved, especially for multi-tone strongly nonlinear circuits, which need to consider a large number of harmonics during the simulation.

2. The HB ultra-sparsifier is based on the cutting based graph sparsification technology. The edge recovery process during the ultra-sparsifier construction is not very efficient for some cases. It is possible to improve the preconditioner by adopting the recent sampling based spectral sparsification technology. Spectral graph sparsification can well preserve the spectrum of a matrix. This possible extension may generate much sparser and efficient preconditioner for HB analysis.

3. The support graph method should be able to be applied to other areas. VLSI simulation is not the only area which needs to solve the large matrix. For example, the famous PageRank algorithm from Google needs to compute the principle eigenvector of the Web graph adjacency matrix. By adopting the support graph method plus iterative solver, it is possible to accelerate the whole process.

References

[1] K. Banerjee, S. Souri, P. Kapur, and K. Saraswat, "3-D ICs: a novel chip design for improving deep-submicrometer interconnect performance and systems-on-chip integration," *Proceedings of the IEEE*, vol. 89, no. 5, pp. 602–633, 2001.

[2] W. Davis, J. Wilson, S. Mick, J. Xu, H. Hua, C. Mineo, A. Sule, M. Steer, and P. Franzon, "Demystifying 3D ICs: the pros and cons of going vertical," *IEEE Design and Test of Computers*, vol. 22, no. 6, pp. 498– 510, 2005.

[3] B. C. Catanzaro, K. Keutzer, and B.-Y. Su, "Parallelizing CAD: a timely research agenda for EDA," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pp. 12–17, 2008.

[4] W. Dong, P. Li, and X. Ye, "Wavepipe: parallel transient simulation of analog and digital circuits on multi-core shared-memory machines," in *Proceedings of IEEE/ACM Design Automation Conference (DAC)*, pp. 238–243, 2008.

[5] F. Gong, H. Yu, and L. He, "PiCAP: A parallel and incremental capacitance extraction considering stochastic process variation," in *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, pp. 764 –769, Jul. 2009.

[6] Y. Lu, H. Zhou, L. Shang, and X. Zeng, "Multicore parallelization of min-cost flow for cad applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 10, pp. 1546–1557, 2010.

[7] Y. Lu, H. Zhou, L. Shang, and X. Zeng, "Multicore parallel min-cost flow algorithm for cad applications," in *Proceedings of IEEE/ACM Design Automation Conference (DAC)*, pp. 832–837, 2009.

[8] S. S. Sapatnekar, E. Haritan, K. Keutzer, A. Devgan, D. Kirkpatrick, S. Meier, D. Pryor, and T. Spyrou, "Reinventing EDA with manycore processors," in *Proceedings of IEEE/ACM Design Automation Conference (DAC)*, pp. 126–127, 2008.

[9] H. K. Thornquist, E. R. Keiter, R. J. Hoekstra, D. M. Day, and E. G. Boman, "A parallel preconditioning strategy for efficient transistor-level circuit simulation," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pp. 410–417, 2009.

[10] X. Ye, W. Dong, P. Li, and S. Nassif, "Maps: multi-algorithm parallel circuit simulation," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 73–78, 2008.

[11] X. Ye, W. Dong, P. Li, and S. Nassif, "Hierarchical multialgorithm parallel circuit simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, pp. 45 –58, Jan. 2011.

[12] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, pp. 56–67, October 2009.

[13] AMD Corporation, "AMD Fusion Family of APUs: Enabling a Superior, Immersive PC Experience," *AMD whitepaper*, vol. [Online]. Available: http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx, 2011.

[14] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," in *ACM SIGGRAPH*, pp. 18:1–18:15, 2008.

[15] Nvidia Corporation, "Bringing High-End Graphics to Handheld Devices," *Nvidia whitepaper*, 2011.

[16] A. Mehrotra and A. Somani, "A robust and efficient harmonic balance (HB) using direct solution of HB Jacobian," in *Proc. IEEE/ACM DAC*, pp. 370–375, 2009.

[17] P. Li and L. T. Pillegi, "Efficient harmonic balance simulation using multi-level frequency decomposition," in *Proc. IEEE/ACM ICCAD*, pp. 677–682, 2004.

[18] W. Dong and P. Li, "Hierarchical harmonic-balance methods for frequency-domain analog-circuit analysis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 12, pp. 2089–2101, 2007.

[19] W. Dong and P. Li, "A parallel harmonic-balance approach to steady-state and envelope-following simulation of driven and autonomous circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 4, pp. 490–501, 2009.

[20] Y. Saad and M. Schultz, "GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, vol. 7, no. 3, pp. 856–869, 1986.

[21] P. Feldmann and B. M. D. Long, "Efficient frequency domain analysis of large nonlinear analog circuits," in *Proc. IEEE CICC*, pp. 461–464, 1996.

[22] X. Zhao, J. Wang, Z. Feng, and S. Hu, "Power grid analysis with hierarchical support graphs," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 543–547, 2011.

[23] X. Zhao and Z. Feng, “Towards efficient SPICE-accurate nonlinear circuit simulation with on-the-fly support-circuit preconditioners,” in *Proc. IEEE/ACM DAC*, pp. 1119–1124, 2012.

[24] X. Zhao and Z. Feng, “GPSCP: a general-purpose support-circuit preconditioning approach to large-scale SPICE-accurate nonlinear circuit simulations,” in *Proc. IEEE/ACM ICCAD*, pp. 429–435, 2012.

[25] L. Han, X. Zhao, and Z. Feng, “An efficient graph sparsification approach to scalable harmonic balance (HB) analysis of strongly nonlinear RF circuits,” in *Proc. IEEE/ACM ICCAD*, pp. 494–499, 2013.

[26] R. Kanj, R. V. Joshi, and S. R. Nassif, “Mixture importance sampling and its application to the analysis of SRAM designs in the presence of rare failure events,” in *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, pp. 69–72, 2006.

[27] K. Agarwal and S. R. Nassif, “Statistical analysis of SRAM cell stability,” in *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, pp. 57–62, 2006.

[28] A. Bansal, R. N. Singh, R. Kanj, S. Mukhopadhyay, J. Lee, E. Acar, A. Singhee, K. Kim, C. Chuang, S. R. Nassif, F. Heng, and K. K. Das, “Yield estimation of SRAM circuits using ”Virtual SRAM Fab”,” in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pp. 631–636, 2009.

[29] J. Wang, S. Yaldiz, X., and L. T. Pileggi, “SRAM parametric failure analysis,” in *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, pp. 496–501, 2009.

[30] J. Wang, A. Singhee, R. A. Rutenbar, and B. H. Calhoun, “Two Fast Methods for Estimating the Minimum Standby Supply Voltage for Large SRAMs,”

IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 29, no. 12, pp. 1908–1920, 2010.

[31] C. Amin, C. Kashyap, N. Menezes, K. Killpack, and E. Chiprout, "A multi-port current source model for multiple-input switching effects in cmos library cells," in *Proceedings of the 43rd annual Design Automation Conference*, pp. 247–252, 2006.

[32] P. Li, Z. Feng, and E. Acar, "Characterizing Multistage Nonlinear Drivers and Variability for Accurate Timing and Noise Analysis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 11, pp. 1205–1214, 2007.

[33] N. Menezes and C. V. Kashyap and C. S. Amin, "A "true" electrical cell model for timing, noise, and power grid verification," in *DAC*, pp. 462–467, 2008.

[34] K. Gulati, J. F. Croix, S. P. Khatri, and R. Shastry, "Fast circuit simulation on graphics processing units," in *Proceedings of the 14th Asia South Pacific Design Automation Conference*, pp. 403–408, 2009.

[35] R. J. Gilmore and M. B. Steer, "Nonlinear circuit analysis using the method of harmonic balance-a review of the art. part i. introductory concepts ," *International Journal on Microwave and Millimeter Wave Computer Aided Engineering*, vol. 1, no. 1, pp. 22–37, 1991.

[36] L. Pillage, R. Rohrer, and C. Visweswariah, *Electronic circuit & system simulation methods.* McGraw-Hill, 1995.

[37] D. Spielman, "Algorithms, graph theory, and linear equations in laplacian matrices," in *Proc. ICM*, 2010.

[38] D. Spielman and S. Teng, "Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems," in *Proc. ACM STOC*, pp. 81–90, 2004.

[39] D. Spielman and N. Srivastava, "Graph sparsification by effective resistances," in *Proc. ACM STOC*, pp. 563–568, 2008.

[40] A. Kolla, Y. Makarychev, A. Saberi, and S. Teng, "Subgraph sparsification and nearly optimal ultrasparsifiers," in *Proc. ACM STOC*, pp. 57–66, 2010.

[41] W. Fung, R. Hariharan, N. Harvey, and D. Panigrahi, "A general framework for graph sparsification," in *Proc. ACM STOC*, pp. 71–80, 2011.

[42] E. Boman and B. Hendrickson, "Support theory for preconditioning," *SIAM J. Matrix Anal. Appl.*, vol. 25, pp. 694–717, 2003.

[43] E. Boman, D. Chen, B. Hendrickson, and S. Toledo, "Maximum-weight-basis preconditioners," *Numerical Linear Algebra and Applications*, vol. 11, pp. 695–721, 2004.

[44] E. Boman, B. Hendrickson, and S. Vavasis, "Solving elliptic finite element systems in near-linear time with support preconditioners," *SIAM J. Numer. Anal.*, vol. 46, no. 6, pp. 3264–3284, 2004.

[45] M. Bern, J. R. Gilbert, B. Hendrickson, N. Nguyen, and S. Toledo, "Support-graph preconditioners," *SIAM J. Matrix Anal. Appl.*, vol. 27, pp. 930–951, 2006.

[46] I. Koutis, G. Miller, A. Sinop, and D. Tolliver, "Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing," tech. rep., CMU, 2009.

[47] L. Ren, X. Chen, Y. Wang, C. Zhang, and H. Yang, "Sparse lu factorization for parallel circuit simulation on gpu," in *Proc. IEEE/ACM DAC*, 2012.

[48] T. Davis and E. P. Natarajan, "Algorithm 907: KLU, a direct sparse solver for circuit simulation problems," *ACM Trans. Math. Softw.*, vol. 37, pp. 36:1–36:17, September 2010.

[49] P. Amestoy, Enseeiht-Irit, T. Davis, and I. Duff, “Algorithm 837: Amd, an approximate minimum degree ordering algorithm,” *ACM Trans. Math. Softw.*, vol. 30, no. 3, pp. 381–388, 2004.

[50] T. Davis, J. Gilbert, S. Larimore, and E. Ng, “Algorithm 836: Colamd, a column approximate minimum degree ordering algorithm,” *ACM Trans. Math. Softw.*, vol. 30, no. 3, pp. 377–380, 2004.

[51] J. Gilbert and T. Peierls, “Sparse partial pivoting in time proportional to arithmetic operations,” *SIAM J. Sci. Stat. Comput.*, vol. 9, no. 5, pp. 862–873, 1988.

[52] G. Kazushige and A. Robert, “High-performance implementation of the level-3 blas.,” 2008.

[53] *Nvidia CUBLAS library user guide v5.5.* [Online]. Available:http://docs.nvidia.com/cuda/cublas/index.html, 2013.

[54] *PETSc: portable, extensible toolkit for scientific computation.* [Online]. Available: http://www.mcs.anl.gov/petsc/index.html.

[55] Nvidia Corporation, *Fermi compute architecture white paper.* [Online]. Available: http://www.nvidia.com/object/fermi_architecture.html, 2010.

[56] *Nvidia CUDA programming guide.* [Online]. Available: http://www.nvidia.com/object/cuda.html, 2007.

[57] R. Burden and J. Faires, *Numerical Analysis,9th Edition.* Brooks Cole, 2010.

[58] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, third ed., 1999.

[59] L. Han, X. Zhao, and Z. Feng, "An Adaptive Graph Sparsification Approach to Scalable Harmonic Balance Analysis of Strongly Nonlinear Post-Layout RF Circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 2, pp. 173–185, 2015.

[60] L. Han, X. Zhao, and Z. Feng, "TinySPICE: a parallel SPICE simulator on GPU for massively repeated small circuit simulations," in *Proc. IEEE/ACM DAC*, pp. 89:1–89:8, 2013.

[61] L. Han and Z. Feng, "Transient-simulation guided graph sparsification approach to scalable harmonic balance (HB) analysis of post-layout RF circuits leveraging heterogeneous CPU-GPU computing systems," in *Proc. IEEE/ACM DAC*, pp. 185:1–185:6, 2015.

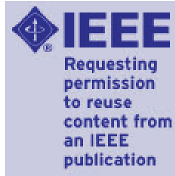
Appendix A

Letters of Permission

A.1 Permission Letters for Chapter 2

Copyright Clearance Center RightsLink®

Home | Create Account | Help | Live Chat



Title: An Adaptive Graph Sparsification Approach to Scalable Harmonic Balance Analysis of Strongly Nonlinear Post-Layout RF Circuits

Author: Lengfei Han; Xueqian Zhao; Zhuo Feng

Publication: Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on

Publisher: IEEE

Date: Feb. 2015

Copyright © 2015, IEEE

LOGIN

If you're a copyright.com user, you can login to RightsLink using your copyright.com credentials. Already **a RightsLink user** or want to learn more?

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

BACK | CLOSE WINDOW

Copyright © 2016 Copyright Clearance Center, Inc. All Rights Reserved. Privacy statement. Terms and Conditions.
Comments? We would like to hear from you. E-mail us at customercare@copyright.com

Copyright Clearance Center RightsLink®

Home | Create Account | Help | Live Chat

Title: Transient-simulation guided graph sparsification approach to scalable Harmonic Balance (HB) analysis of post-layout RF circuits leveraging heterogeneous CPU-GPU computing systems

Conference Proceedings: Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE

Author: Lengfei Han; Zhuo Feng

Publisher: IEEE

Date: 8-12 June 2015

Copyright © 2015, IEEE

LOGIN

If you're a copyright.com user, you can login to RightsLink using your copyright.com credentials. Already **a RightsLink user** or want to learn more?

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

BACK | CLOSE WINDOW

Copyright © 2016 Copyright Clearance Center, Inc. All Rights Reserved. Privacy statement. Terms and Conditions.
Comments? We would like to hear from you. E-mail us at customercare@copyright.com

A.2 Permission Letter for Chapter 3

ACM CONFERENCE COPYRIGHT FORM AND AUDIO/VIDEO RELEASE

Title of the Work: TinySPICE: A Parallel SPICE Simulator on GPU for Massively Repeated Small Circuit Simulations

Publication and/or Conference Name: DAC '13: The 50th Annual Design Automation Conference 2013 Proceedings

Author/Presenter(s): Lengfei Han;xueqian zhao

Auxiliary Materials (provide filenames and a description of auxiliary content, if any, for display in the ACM Digital Library. The description may be provided as a ReadMe file):

I. COPYRIGHT TRANSFER

Copyright to the Work and to any supplemental files integral to the Work which are submitted with it for review and publication such as an extended proof, a PowerPoint outline, or appendices that may exceed a printed page limit, (including without limitation, the right to publish the Work in whole or in part in any and all forms of media, now or hereafter known) is hereby transferred to the ACM (for Government work, to the extent transferable **-see Part I. B. below**) effective as of the date of this agreement, on the understanding that the Work has been accepted for publication by ACM.

Employer / Author(s) Retained Rights. Each of the Employer/Author(s) retains the following rights:

1. All other proprietary rights to the work such as patent;
2. The right to reuse any portion of the Work, without fee, in future works of the Authors (or Authors Employers) own, including books, lectures and presentations in all media, provided that the ACM citation, notice of the Copyright and the ACM DOI are included (See Section 4 below). Requests made on behalf of others, however (i.e. contributions to the work of other authors or other editors), usually require payment of a fee;
3. The right to revise the work. (See Policy 2.4 Definitive Versions and Revisions);
4. The right to post author-prepared versions of the Work covered by the ACM copyright in a personal collection on their own home page, on a publicly accessible server of their employer and in a repository legally mandated by the agency funding the research on which the Work is based. Such posting is limited to noncommercial access and personal use by others, and must include the following notice both embedded within the full text file and in the accompanying citation display as well:
   " ACM, (YEAR). This is the authors version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in PUBLICATION, {VOL#, ISS#, (DATE)} http://doi.acm.org/10.1145/{nnnnnn.nnnnnn}".
   (You may find the nnnnnn.nnnnnn number for your article DOIs on its citation page in the ACM Digital Library.)
5. The right of an employer who originally owned the copyright to distribute definitive copies of its author-employees Work within its organization. Posting these works for access outside of the employers organization requires explicit permission from ACM.

Authors should understand that consistent with ACMs policy of encouraging dissemination of information, each work published by ACM appears with the ACM copyright and the following notice:

"Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee."

☑ A. Assent to Assignment. I hereby represent and warrant that I am the sole owner (or authorized agent of the copyright owner(s)), with the exception of third party materials detailed in section III below. I have obtained permission for any third-party material included in the Work.

☐ B. Declaration for Government Work. I am an employee of the National Government of my country and my Government claims rights to this work, or it is not copyrightable (Government work is classified as Public Domain in U.S. only)

Are any of the co-authors, employees or contractors of a National Government?
○ Yes ◉ No
Country:

II. PERMISSION FOR CONFERENCE TAPING AND DISTRIBUTION (Check A and, if applicable, B)
A. Audio /Video Release

I hereby grant permission for ACM to include my name, likeness, presentation and comments in any and all forms, for the Conference and/or Publication.

I further grant permission for ACM to record and/or transcribe and reproduce my presentation as part of the ACM Digital Library, and to distribute the same for sale in complete or partial form as part of an ACM product on CD-ROM, DVD, webcast, USB device, streaming video or any other media format now or hereafter known.

I understand that my presentation will not be sold separately by itself as a stand-alone product without my direct consent. Accordingly, I give ACM the right to use my image, voice, pronouncements, likeness, and my name, and any biographical material submitted by me, in connection with the Conference and/or Publication, whether used in excerpts or in full, for distribution described above and for any associated advertising or exhibition.

Do you agree to the above Audio/Video Release? ◉ Yes ○ No

B. Auxiliary Materials, not integral to the Work

I hereby grant ACM permission to serve files named below containing my Auxiliary Material from the ACM Digital Library. I hereby represent and warrant that my Auxiliary Material contains no malicious code, virus, trojan horse or other software routines or hardware components designed to permit unauthorized access or to disable, erase or otherwise harm any computer systems or software, and I hereby agree to indemnify and hold harmless ACM from all liability, losses, damages, penalties, claims, actions, costs and expenses (including reasonable legal expense)

arising from the use of such files.

☐ I agree to the above Auxiliary Materials permission statement

III. Third Party Materials

In the event that any materials used in my presentation or Auxiliary Materials contain the work of third-party individuals or organizations (including copyrighted music or movie excerpts or anything not owned by me), I understand that it is my responsibility to secure any necessary permissions and/or licenses for print and/or digital publication, and cite or attach them below.

◉ We/I have not used third-party material.
○ We/I have used third-party materials and have necessary permissions.

IV. Artistic Images

If your paper includes images that were created for any purpose other than this paper and to which you or your employer claim copyright, you must complete Part IV and be sure to include a notice of copyright with each such image in the paper.
◉ We/I do not have any artistic images.
○ We/I have have any artistic images.

V. Liability Waivers & Indeminifcations

* Your Liability Waiver is conditional upon you agreeing to the terms set out below.

Because I retain certain rights in my work under the ACM Copyright Transfer Agreement and under the ACM Permissions Release Form, such as patent and moral rights, I therefore hereby agree not to assert any of my rights against ACM in connection with ACM's use of my work as agreed to herein, and I further acknowledge that ACM is under no obligation to exercise all of the rights I have granted.

I hereby agree to indemnify ACM and its agents and assigns against any and all losses incurred in connection with any claim or proceedings asserting that I have violated a prior agreement in presenting my work at an ACM event and/or in granting ACM rights to publish my work.

I hereby agree to indemnify ACM and its agents and assigns against any and all losses incurred in connection with any claim or proceeding asserting plagiarism and/or copyright infringement if the investigation carried out by ACM according to its Plagiarism Policy (see: http://www.acm.org/publications/policies/plagiarism_policy) determines that my work is the plagiarizing or infringing work. [Note, in accordance with its policies, ACM generally defends its authors against charges of plagiarism and will reasonably investigate others on behalf of the author who plagiarize a work copyrighted and published by ACM.]

All permissions and releases granted by me herein shall be effective in perpetuity and shall extend and apply to ACM and its agents and assigns.

All permissions and releases granted by me herein shall be effective in perpetuity and extend and apply to ACM and its assigns, contractors, sublicensed distributors,

successors, and agents.

☑ I agree to the above liability waiver

DATE: **03/08/2013** sent to hanlengfei@gmail.com at **17:03:23**