UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Profiling Analysis Of GPU-Accelerated Methods For Circuit Simulation

Diploma Thesis

Zisis Balatsos

Supervisor: Nestor Evmorfopoulos

September 2025

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Profiling Analysis Of GPU-Accelerated Methods For Circuit Simulation

Diploma Thesis

Zisis Balatsos

Supervisor: Nestor Evmorfopoulos

September 2025

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ανάλυση Προφίλ Επιταχυνόμενων GPU Μεθόδων Για Προσομοίωση Κυκλωμάτων

Διπλωματική Εργασία

Ζήσης Μπαλατσός

Επιβλέπων/: Νέστωρ Ευμορφόπουλος

Σεπτέμβριος 2025

Approved by the Examination Committee:

Supervisor **Nestor Evmorfopoulos**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Fotios Plessas**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Georgios Stamoulis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Nestor Evmorfopoulos, for his guidance and support throughout this thesis. His insightful feedback and careful mentoring shaped not only the research presented here but also my approach to problem-solving and scientific communication. I am especially grateful for his professional advice and mentorship beyond the scope of this work, which have been invaluable for my development and future career.

Furthermore, I am deeply grateful to my family for their constant encouragement, understanding, and sacrifices over the years of my studies. Their support provided the stability and motivation I needed to complete this work. Without their belief in me, this thesis would not have been possible.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Zisis Balatsos

Diploma Thesis

Profiling Analysis Of GPU-Accelerated Methods For Circuit Simulation

Zisis Balatsos

Abstract

The present thesis is positioned within the domains of computer architecture and scientific computing, with a focus on enhancing the performance of numerical methods for circuit simulation. A simulation tool was developed, using C and C++ programming languages, for the analysis of analog circuits, capable of computing their DC operating point through the Modified Nodal Analysis (MNA) algorithm, as well as supporting transient and AC analyses along with waveform generation. The central objective of this research was to investigate the efficiency of direct solution methods for large sparse linear systems of the form $Ax = b$, which arise from circuit equations, and to evaluate the impact of GPU-based direct methods on solving time of such systems.

The main hypothesis is that leveraging the parallelism of modern GPUs can significantly accelerate the solving phase, which constitutes a computational bottleneck in circuit simulation. In order to asses this, two distinct GPU acceleration approaches were implemented and benchmarked against a CPU-based baseline solver using the NVIDIA's cudss and cuSOLVER libraries. The first approach utilized cuSOLVER library, which resulted in unnoticeable speedup over the optimized sequential baseline solver. The second and main approach employed cuDSS library, a GPU-accelerated direct sparse solver, along with 3 different implementations which exploit the library's advanced features. These implementations resulted in a significant reduction in computation time, demonstrating the effectiveness of GPU acceleration.

The results of this study underscore that while a straightforward port of CPU methods to GPU may not yield benefits due to overheads and algorithmic constraints, specialized GPU libraries such as cuDSS can lead to a profound improvement in simulation runtime, consequently validating the potential of GPU computing to address the scalability challenges inherent in simulating large-scale electronic circuits.

Διπλωματική Εργασία

Ανάλυση Προφίλ Επιταχυνόμενων GPU Μεθόδων Για Προσομοίωση Κυκλωμάτων

Ζήσης Μπαλατσός

Περίληψη

Η παρούσα διατριβή εντάσσεται στους τομείς της αρχιτεκτονικής υπολογιστών και των επιστημονικών υπολογισμών, με έμφαση στη βελτίωση της απόδοσης των αριθμητικών μεθόδων για την προσομοίωση κυκλωμάτων. Στα πλαίσια της παρούσας εργασίας, αναπτύχθηκε ένα εργαλείο προσομοίωσης, χρησιμοποιώντας τις γλώσσες προγραμματισμού C και C++, για την ανάλυση αναλογικών κυκλωμάτων, ικανό να υπολογίζει το σημείο λειτουργίας DC μέσω του αλγορίθμου Modified Nodal Analysis (MNA), καθώς και να υποστηρίζει μεταβατικές και AC αναλύσεις, καθώς και την δημιουργία κυματομορφών. Ο κεντρικός στόχος αυτής της έρευνας ήταν να διερευνήσει την αποτελεσματικότητα των μεθόδων άμεσης επίλυσης για μεγάλα αραιά γραμμικά συστήματα της μορφής $Ax = b$, τα οποία προκύπτουν από εξισώσεις κυκλωμάτων, και να αξιολογήσει τον αντίκτυπο των μεθόδων άμεσης επίλυσης, που βασίζονται σε GPUs, στον χρόνο επίλυσης τέτοιων συστημάτων.

Η κεντρική υπόθεση της έρευνας, είναι ότι η αξιοποίηση του παραλληλισμού των σύγχρονων GPUs μπορεί να επιταχύνει σημαντικά τη φάση επίλυσης, η οποία αποτελεί υπολογιστικό εμπόδιο στην προσομοίωση κυκλωμάτων. Προκειμένου να αξιολογηθεί αυτό, εφαρμόστηκαν δύο διαφορετικές προσεγγίσεις επιτάχυνσης GPU και συγκρίθηκαν με μια βασική μέθοδο επίλυσης βασισμένη σε CPU, χρησιμοποιώντας τις βιβλιοθήκες cuDSS και cuSOLVER της NVIDIA. Η πρώτη προσέγγιση χρησιμοποίησε τη βιβλιοθήκη cuSOLVER, η οποία είχε ως αποτέλεσμα μια μη αισθητή επιτάχυνση σε σχέση με τον βελτιστοποιημένο διαδοχικό βασικό επιλυτή. Η δεύτερη και κύρια προσέγγιση χρησιμοποίησε τη βιβλιοθήκη cuDSS, μια επιταχυνόμενη από GPU άμεση μέθοδο αραιής επίλσυς και πιο συγκεκριμένα 3 διαφορετικές εφαρμογές που εκμεταλλεύονται τις προηγμένες δυνατότητες της βιβλιοθήκης. Αυτές οι εφαρμογές είχαν ως αποτέλεσμα μια σημαντική μείωση του χρόνου υπολογισμού, αποδεικνύοντας την αποτελεσματικότητα της επιτάχυνσης GPU.

Τα αποτελέσματα αυτής της μελέτης υπογραμμίζουν ότι, ενώ η απλή μεταφορά μεθόδων CPU σε GPU ενδέχεται να μην αποφέρει οφέλη λόγω των επιπλέον δαπανών και των

αλγοριθμικών περιορισμών, εξειδικευμένες βιβλιοθήκες GPU όπως η cuDSS μπορούν να οδηγήσουν σε σημαντική βελτίωση του χρόνου εκτέλεσης της προσομοίωσης, επικυρώνοντας έτσι το δυναμικό της υπολογιστικής ισχύος GPU για την αντιμετώπιση των προκλήσεων που είναι εγγενείς στην προσομοίωση κυκλωμάτων μεγάλης κλίμακας.

Table of contents

List of figures

Abbreviations

GPU	Graphical Processing Unit
CPU	Central Processing Unit
MNA	Modified Nodal Analysis
nnz	Non-zero
SPICE	Simulation Program with Integrated Circuit Emphasis
CG	Conjugate Gradient
BiCG	Biconjugate Gradient
SPD	Symmetric and Positive Definite
CUDA	Compute Unified Device Architecture
cuDSS	CUDA Direct Sparse Solver
cuSOLVER	CUDA Solver
GSL	GNU Scientific Library
AMD	Approximate Minimum Degree
MD	Minimum Degree
ODE	Ordinary Differential Equations
BE	Backward Euler
TR	Trapezoidal
SM	Streaming Multiprocessor
MGMN	Multi-GPU Multi-Node
SIMT	Single Instruction Multiple Thread
CSC	Compressed Sparse Column Format
CSR	Compressed Sparse Row Format
RHS	Right-Hand-Side
D2H	Device-To-Host
H2D	Host-To-Device

MPI	Message Passing Interface
MIG	Multiple-Instance GPU
VRAM	Video Random-Access Memory
FLOPS	Floating point operations per second

Chapter 1

Introduction

The ongoing development of contemporary technology has resulted in the creation of ever-more intricate electronic systems. Circuit simulators, which are fundamental tools in both academic research and industrial applications, play a crucial role in evaluating these systems. Such tools work by translating a descriptive netlist of a circuit into a system of mathematical equations, a task most commonly handled by the Modified Nodal Analysis (MNA) algorithm [1]. The electrical behavior of the circuit, including its steady-state operating point or the time-domain response to a signal, can be found by solving the emerging system. However, the strength of this approach, its ability to model large circuits, is also its primary weakness. A key characteristic of these systems is that they become overwhelmingly sparse as their size increases, meaning that most of the entries in the coefficient matrix are zero. This sparsity arises from the local connectivity of the circuit components; each element typically connects only to a few nodes.

The challenge is not only the sheer scale of the computations, but also the need for robustness and accuracy. Circuit equations frequently result in sparse matrices with highly irregular structures, leading to inefficiencies when using conventional dense-matrix approaches. Furthermore, iterative methods that are sometimes employed for memory efficiency can struggle with convergence, especially for stiff systems. As a result, direct methods, despite their higher computational cost, remain essential for ensuring accurate and stable solutions. The process of solving these large, sparse linear systems consistently emerges as the dominant bottleneck, often consuming the majority of the total simulation time.

1.1 The Computational Challenge: Sparse Matrix Solving

One of the central difficulties in circuit simulation is rooted in the solution of large systems of linear equations, $Ax = b$, due to the fact that the arising matrices are of large scale and strongly sparse; the vast majority of entries are zero, while only a small part of the matrix carries nonzero values. This structure is not accidental; it reflects the local connectivity of the circuit elements, where each node interacts only with a limited number of others. As a circuit's complexity grows, with its node count scaling to millions, the dimensions of matrix A expand accordingly and its sparsity remains extremely high, with densities below one percent. Although sparsity greatly reduces the amount of memory usage compared to dense matrices, it also presents challenges. The most critical is the fill-in phenomenon during the factorization steps, where entries change from being initially zero to a nonzero value. The amount of fill-in is highly dependent on the order in which equations are processed. Consequently, a significant pre-processing step in any efficient sparse solver involves finding a permutation of the matrix that minimizes this fill-in, using algorithms like Minimum Degree (MD) [2], without destroying the sparsity of matrix A.

In summary, the computational challenge lies in storing these matrices compactly, but also in performing robust and efficient factorization and solution steps, balancing accuracy and efficiency in sparse direct and iterative solvers.

1.2 The Promise of GPU Acceleration

The rise of parallel computing and the development of modern GPUs has transformed the landscape of scientific computing, making them widely used in numerical linear algebra due to their ability to perform thousands of operations simultaneously. The parallel architecture can be harnessed to accelerate the solution bottleneck of large sparse systems. This characteristic makes them particularly well-suited to the kinds of repetitive, structured computations found in factorization and solving phases of such systems. Despite these advances, applying GPU parallelization to circuit simulation introduces several difficulties. Not all numerical algorithms can be easily adapted to massively parallel architectures, and the irregular sparsity patterns typical of circuit matrices can hinder efficient execution. Furthermore, sparse problems are memory-bound which can lead to a memory transfer of data overhead and often exhibit irregular sparsity patterns, which can be difficult to map efficiently onto a

GPU's memory. Therefore, specialized libraries and algorithms are needed to handle sparsity, factorization and solving steps to exploit GPU hardware effectively and take advantage of performance acceleration for computational challenges of circuit simulation.

1.3 Document Structure

Chapter 2, details the core architecture of the developed circuit simulator using C and C++. It begins with the mathematical modeling of circuit elements, followed by the formulation of circuit equations using the Modified Nodal Analysis (MNA) algorithm. The discussion then focuses on Direct (e.g. LU factorization, Cholesky) and Iterative methods (e.g. CG, BiCG) for solving the resulting $Ax = b$ systems, considering the cases of dense and sparse matrices. Finally, the chapter outlines the implementation of transient and AC analysis, explaining the construction and solution of the complex-valued system.

Chapter 3, presents the main focus of this thesis: using GPU parallelism to speed up the solution phase of sparse linear problems. It is divided into two main approaches. First, it explains the implementation of a hybrid solver using both CSparse factorization methods and NVIDIA's cuSOLVER library. Secondly, it describes the phased execution model (analysis, factorization, solve) of NVIDIA's cuDSS library, a dedicated Sparse Solver. The chapter covers implementation details, the integration into the simulation tool and the encountered challenges for both methods.

Chapter 4, reports the experimental evaluation of the developed solvers. Performance of CPU-based methods is compared against the two GPU-based implementations, utilizing large-scale circuits from IBM's Power Grid Benchmarks. The experimental setup and methodology for measuring execution time of symbolic factorization, numerical factorization and solving, as well as GPU memory consumption are clearly defined. The chapter concludes with a thorough discussion that interprets the results, highlighting the observed performance improvement, and validating the core hypothesis regarding the impact of specialized GPU libraries.

Chapter 2

Simulation Tool

2.1 Introduction

This chapter details the design and implementation of the circuit simulation tool that was used for testing the core primary objective of this thesis; using GPU-accelerated direct sparse solvers for optimized performance over classical CPU-based solvers. This tool takes SPICE as a reference and provides a flexible and efficient software framework for the numerical simulation of linear circuits, capable of performing fundamental types of analysis: DC Operating Point, DC Sweep, Transient and AC Analysis.

The implementation follows a modular architecture, where each core component of a traditional SPICE simulator [3] is built and integrated sequentially. The foundation of the tool is the Modified Nodal Analysis algorithm, which provides a systematic and automated method for constructing a linear system of equations that mathematically describe the interconnected circuit elements.

The workflow begins with a parser that interprets a specific netlist, based on the IBM Power Grid Benchmarks [4] format, of the circuit and the specified simulations. Then an internal data structure is formed that represents all the circuit components and their connections. This data structure is then traversed by the equation formulation module, which builds the sparse matrix system $Ax = b$ according to the rules of MNA. The specific form of this system depends on the type of analysis requested (e.g., static for DC, differential for Transient, complex for AC).

A critical and computationally intensive part of the simulation is solving the resulting linear systems. This tool investigates two distinct families of solving methods:

1. Direct methods (LU, Cholesky factorization), prized for their robustness and numerical stability.

2. Iterative methods (Conjugate Gradient, Bi-Conjugate Gradient), which offer the potential for higher performance on very large sparse systems by converging to a solution.

The efficiency of these solvers is paramount, especially for large-scale circuits. Therefore, the implementation heavily leverages sparse matrices and their solving methods throughout the entire process to minimize memory usage and runtime. Finally, the tool supports Transient and AC analysis. Transient analysis employs numerical integration techniques (Backward Euler, Trapezoidal) to solve the system of differential equations governing the circuit's time-domain behavior. The AC analysis module operates in the frequency domain, solving a complex-valued linear system to determine the circuit's sinusoidal steady state. The following sections in this chapter provide a comprehensive description of the mathematical models, algorithms, and implementation for each of these stages.

2.2 Equation Formulation

The basis of the developed simulation tool is the formulation of the equation system that mathematically describes the interconnected circuit elements. This is achieved using the Modified Nodal Analysis (MNA) method.

2.2.1 Fundamental Concepts and Definitions

The circuit is represented as a directed graph:

- $V = \{0, 1, \ldots, n-1\}$,the set of nodes, where node 0 is the reference (ground) node.
- $E = \{e_1, e_2, \ldots, e_m\}$,the set of branches (edges).

The topology of the circuit is described by the reduced incidence matrix $A \in \mathbb{R}^{(n-1)\times m}$

$$a_{ij} = \begin{cases} +1, & \text{if branch } j \text{ leaves node } i, \\ -1, & \text{if branch } j \text{ enters node } i, \\ 0, & \text{if branch } j \text{ is not connected to node } i. \end{cases}$$

The electrical state of the circuit is described by three vectors:

- Vector of node voltages:

$$v(t) = \Big[v_1(t), v_2(t), \ldots, v_{n-1}(t)\Big]^T$$

- Vector of branch voltages:

$$u(t) = \Big[u_1(t), u_2(t), \ldots, u_m(t)\Big]^T$$

- Vector of branch currents:

$$i(t) = \Big[i_1(t), i_2(t), \ldots, i_m(t)\Big]^T$$

Kirchhoff's Laws can then be expressed in matrix form:

- Kirchhoff's Current Law (KCL): $Ai(t) = 0$
- Kirchhoff's Voltage Law (KVL): $u(t) = A^T\, v(t)$

2.2.2 Element Classification and Constitutive Equations

Circuit elements are divided into two groups based on the form of their equation:

- Group G1 which includes resistors, capacitors and current sources:

$$i_k(t) = g_k\, u_k(t) + c_k\, \frac{du_k(t)}{dt} + s_k(t)$$

- Group G2 which includes inductors and voltage sources.

Let $m1$ be the number of G1 elements and $m2$ the number of G2 elements:

- $m1 + m2 = m$
- $A = \Big[A_1 \quad A_2\Big], \quad A_1 \in \mathbb{R}^{(n-1)\times m_1}, \quad A_2 \in \mathbb{R}^{(n-1)\times m_2}$
- $u(t) = \begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix}, \quad u_1(t) \in \mathbb{R}^{m_1\times 1}, \quad u_2(t) \in \mathbb{R}^{m_2\times 1}$
- $i(t) = \begin{bmatrix} i_1(t) \\ i_2(t) \end{bmatrix}, \quad i_1(t) \in \mathbb{R}^{m_1\times 1}, \quad i_2(t) \in \mathbb{R}^{m_2\times 1}$

2.2.3 MNA Formulation

The constitutive equations for each group are written in matrix form.

- Group 1:

$$i_1(t) = G\,u_1(t) + C\,\frac{du_1(t)}{dt} + s_1(t)$$

  - $G : (m1 \times m1)$ diagonal matrix of conductances with non-zeros in resistor positions
  - $C : (m1 \times m1)$ diagonal matrix of capacitances with non-zeros in capacitor positions
  - $s_1(t) : (m_1 \times 1)$ vector with non-zeros in current source positions

- Group 2:

$$u_2(t) = L\,\frac{di_2(t)}{dt} + s_2(t)$$

  - $L : (m2 \times m2)$ diagonal matrix of inductances with non-zeros in inductor positions
  - $s_2(t) : (m_2 \times 1)$ vector with non-zeros in voltage source positions

MNA system which arises after succesive substitutions:

$$\begin{bmatrix} A_1 G A_1^T & A_2 \\ A_2^T & 0 \end{bmatrix} \begin{bmatrix} v(t) \\ i_2(t) \end{bmatrix} + \begin{bmatrix} A_1 C A_1^T & 0 \\ 0 & -L \end{bmatrix} \begin{bmatrix} \frac{dv(t)}{dt} \\ \frac{di_2(t)}{dt} \end{bmatrix} = \begin{bmatrix} -A_1 s_1(t) \\ s_2(t) \end{bmatrix}$$

where:

- $x(t) = \begin{bmatrix} v(t) \\ i_2(t) \end{bmatrix}, \quad G = \begin{bmatrix} A_1 G A_1^T & A_2 \\ A_2^T & 0 \end{bmatrix}, \quad C = \begin{bmatrix} A_1 C A_1^T & 0 \\ 0 & -L \end{bmatrix}$

- $b(t) = \begin{bmatrix} -A_1 s_1(t) \\ s_2(t) \end{bmatrix}$

2.2.4 Element Stamping

The matrices G and C are constructed using a stamping algorithm. C, G and b are initialized to zero and by iterating through each circuit element, a stamp is added, representing the contribution of each element to these matrices and vector. The algorithm can be studied in depth in Farid's N. Najm book, chapter 2.4.4 Modified Nodal Analysis [5].

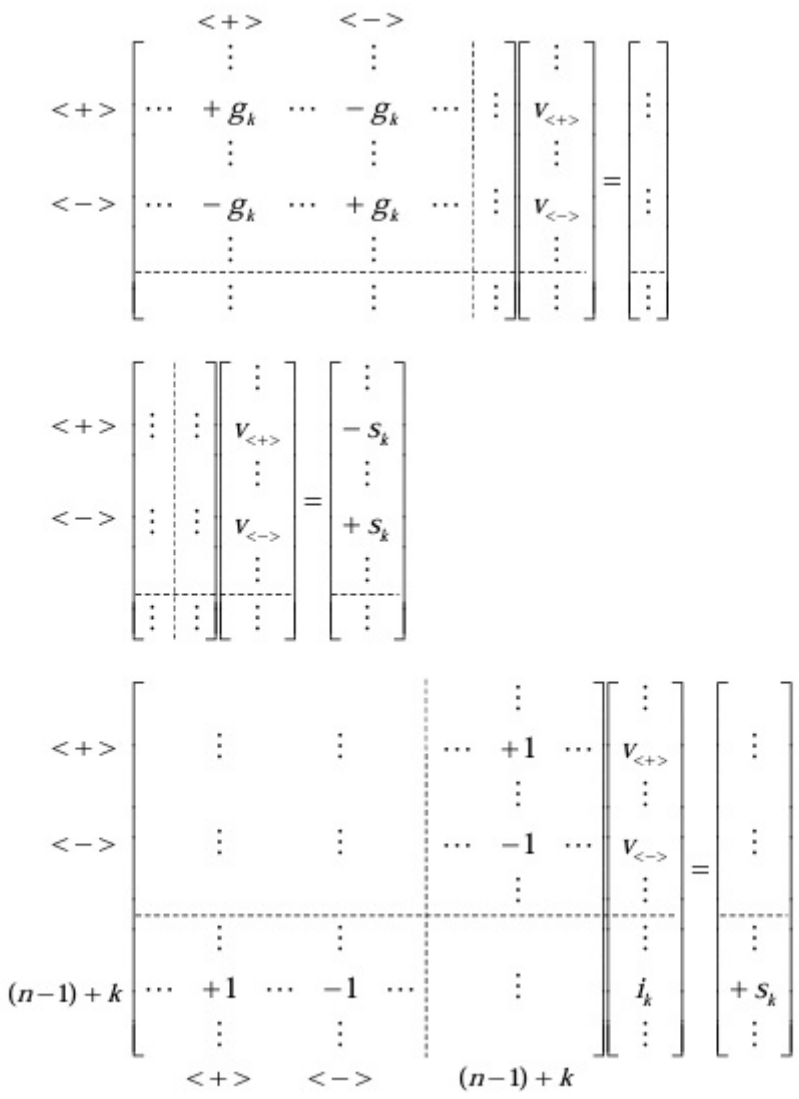

Figure 2.1: Stamps for resistor, independent current source and independent voltage source

2.2.5 DC Analysis

In DC Analysis, the time-derivative terms vanish. Capacitors are replaced by open circuits and their stamp to C is ignored, contributing nothing to G. Inductors are replaced by short circuits and they are treated as a zero-value voltage source. The solution of this DC system determines the DC operating point of the circuit:

$$\begin{bmatrix} A_1 G A_1^T & A_2 \\ A_2^T & 0 \end{bmatrix} \begin{bmatrix} v(t) \\ i_2(t) \end{bmatrix} = \begin{bmatrix} -A_1 s_1(t) \\ s_2(t) \end{bmatrix}$$

where:

- $A = \begin{bmatrix} A_1 G A_1^T & A_2 \\ A_2^T & 0 \end{bmatrix}$, $A \in \mathbb{R}^{[(n-1)+m2]\times[(n-1)+m2]}$
- $x = \begin{bmatrix} v(t) \\ i_2(t) \end{bmatrix}$, $x \in \mathbb{R}^{[(n-1)+m2]\times 1}$
- $b = \begin{bmatrix} -A_1 s_1(t) \\ s_2(t) \end{bmatrix}$, $b \in \mathbb{R}^{[(n-1)+m2]\times 1}$

2.3 Solving Methods

The choice of solving method is critical, as it directly impacts the simulation's numerical stability, execution time and memory consumption. Solvers are broadly classified into two categories: Direct Methods and Iterative Methods.

Direct methods such as Gaussian Elimination with LU factorization and Cholesky, compute the solution via a finite number of operations. Their numerical robustness and guaranteed convergence makes them highly reliable and the standard choice for the majority of circuit simulations, especially for systems that are not too large. Their primary drawback is their computational complexity, which becomes prohibitive for very large systems at $O(n^3)$ for factorization and $O(n^2)$ for solving. On the other hand, their main disadvantages are that

convergence is not guaranteed for all matrices and their performance is highly dependent on the conditioning of the system and the quality of the preconditioner.

Iterative methods such as Conjugate Gradient (CG) [6] and Biconjugate Gradient (BiCG) [7], generate a sequence of progressively improving approximations to the solution. They terminate once a specified convergence criterion is satisfied. Their key advantage is low memory consumption and a per-iteration complexity of $O(n)$ or $O(n^2)$, making them suitable for very large sparse systems where direct methods would be extremely slow or memory intensive.

The choice between these two methods is a trade-off between reliability and computational scalability. The following subsection will first detail the direct methods implemented in the tool.

2.3.1 Direct Methods

Direct methods for solving $Ax = b$ are based on factorizing matrix A into a product of simpler matrices, which are then easily solved. The two primary factorizations used are LU factorization (for general matrices) [8] and Cholesky factorization for symmetric positive definite (SPD) matrices [9].

For a square and invertible matrix $A \in \mathbb{R}^{n\times n}$, the LU Factorization with partial pivoting decomposes it as: $PA = LU$

where:

- $P \in \mathbb{R}^{n\times n}$,the permutation matrix which implements partial pivoting
- L ,the identity lower triangular matrix $(l_{ii} = 1)$
- U ,the upper triangular matrix

Algorithm 1 LU Factorization

for $k = 1$ to n **do**
  $x \leftarrow |a_{kk}|$; $m \leftarrow k$
  for $i = k + 1$ to n **do**
    if $|a_{ik}| > x$ **then**
      $x \leftarrow |a_{ik}|$; $m \leftarrow i$
    end if
  end for
  Interchange rows k and m
  for $i = k + 1$ to n **do**
    $l_{ik} \leftarrow a_{ik}/a_{kk}$
  end for
  for $i = k + 1$ to n **do**
    for $j = k + 1$ to n **do**
      $a_{ij} \leftarrow a_{ij} - l_{ik}\, a_{kj}$
    end for
  end for
end for

After the completion of LU Factorization, $PAx = LUx = Pb$ is solved by performing forward and backward substitution:

- Solve lower triangular system: $Ly = Pb$

Algorithm 2 Forward Substitution

for $i = 1$ to n **do**
  for $j = 1$ to $i - 1$ **do**
    $b[i] \leftarrow b[i] - l[i][j] \cdot y[j]$
  end for
  $y[i] \leftarrow b[i]/l[i][i]$
end for

- Solve upper triangular system: $Ux = y$

Algorithm 3 Backward Substitution

1: **for** $i = n$ to 1 **do**
2: **for** $j = i + 1$ to n **do**
3: $y[i] \leftarrow y[i] - u[i][j] \cdot x[j]$
4: **end for**
5: $x[i] \leftarrow y[i]/u[i][i]$
6: **end for**

The matrix $A \in \mathbb{R}^{n \times n}$ that arises from MNA can often be of Symmetric Positive Definite (SPD) format, if the circuti only consists of G1 group elements; resistors, independent current sources and capacitors. In such case, the more efficient Cholesky Factorization, which decomposes A without pivoting, can be used. For A to be SPD, the following relations should be satisfied:

- $A = A^T$
- $x^T Ax > 0,\ \forall x \in \mathbb{R}^n,\ x \neq 0$

Cholesky Factorization decomposes A as: $A = LL^T$

- L is a lower triangular matrix with positive diagonal entries

Algorithm 4 Cholesky Factorization

1: **for** $j = 1$ to n **do**
2: $l[j][j] \leftarrow \sqrt{a[j][j] - \sum_{k=1}^{j-1} l[j][k]^2}$
3: **for** $i = j + 1$ to n **do**
4: $l[i][j] \leftarrow \dfrac{a[i][j] - \sum_{k=1}^{j-1} l[i][k] \cdot l[j][k]}{l[j][j]}$
5: **end for**
6: **end for**

After the completion of Cholesky Factorization, $Ax = LL^T x = b$ is solved similarly to the LU method above:

- Forward substitution: $Ly = b$

- Backward substitution: $L^T x = y$

The developed simulation tool implements both methods using GSL library [10], which provides efficient routines both for LU and Cholesky method.

```
    for(i = 0; i < rowsA; i++){
        for(j = 0; j < columnsA; j++){
            gsl_matrix_set(A, i, j, array_A[i][j]);
        }
    }

    for(i = 0; i < rowsx; i++){
            gsl_vector_set(X, i, array_x[i]);
    }

    for(i = 0; i < rowsb; i++){
            gsl_vector_set(B, i, array_b[i]);
    }

    gsl_linalg_LU_decomp(A, P, &s);

    // gsl_matrix* A, gsl_permutation* P, gsl_vector* B, gsl_vector* X
    gsl_linalg_LU_solve(A, P, B, X);
```

Listing 2.1: LU factorization using GSL

```
    for(i = 0; i < rowsA; i++){
        for(j = 0; j < columnsA; j++){
            gsl_matrix_set(A, i, j, array_A[i][j]);
        }
    }

    for(i = 0; i < rowsx; i++){
            gsl_vector_set(X, i, array_x[i]);
    }

    for(i = 0; i < rowsb; i++){
            gsl_vector_set(B, i, array_b[i]);
    }

    gsl_linalg_cholesky_decomp(A);

    // gsl_matrix* A, gsl_vector* B, gsl_vector* X
    gsl_linalg_cholesky_solve(A, B, X);
```

Listing 2.2: Cholesky factorization using GSL

2.3.2 Iterative Methods

Iterative methods, provide a potent substitute to Direct methods by producing a series of ever better approximate solutions, until a convergence criterion is reached. A solution is often achieved with significantly fewer operations and less memory usage, especially for large sparse matrices, with the application of suitable preconditioners that take advantage of the specific characteristics of A. The tool implements a Jacobi preconditioner $M = diag(A)$ for both Conjugate Gradient (CG) and Biconjugate Gradient (BiCG), which are part of Krylov subspace methods. In addition, the relative norm of the residual acts as the convergence threshold and tolerance can be configured inside the netlist file with the default value being 10^{-3}.

The Conjugate Gradient method is the leading iterative algorithm for solving systems $A\mathbf{x} = \mathbf{b}$, exploiting matrix $A \in \mathbb{R}^{n \times n}$ being Symmetric Positive Definite (SPD). The CG convergence rate is faster for well-conditioned A matrices, which approximate the I matrix. On the other hand, the convergence rate is slow when matrix A is ill-conditioned and tends

to be non invertible. To speed up convergence, a preconditioner M is almost always used. The preconditioner is an approximation of A^{-1}, that is cheap to compute and apply.

Algorithm 5 Conjugate Gradient Method

Require: Matrix A, right-hand side b, preconditioner M, tolerance `itol`, maximum iterations n

Ensure: Approximate solution x to $Ax = b$

$x \leftarrow x^{(0)}$
$r \leftarrow b - Ax$
$iter \leftarrow 0$
while $\|r\|/\|b\| > itol$ **and** $iter < n$ **do**
    $iter \leftarrow iter + 1$
    Solve $Mz = r$
    $\rho \leftarrow r^\top z$
    if $iter = 1$ **then**
        $p \leftarrow z$
    else
        $\beta \leftarrow \rho/\rho_{old}$
        $p \leftarrow z + \beta p$
    end if
    $\rho_{old} \leftarrow \rho$
    $q \leftarrow Ap$
    $\alpha \leftarrow \rho/(p^\top q)$
    $x \leftarrow x + \alpha p$
    $r \leftarrow r - \alpha q$
end while

For general, non-symmetric matrices A, the CG method is not applicable. The Biconjugate Gradient method extends the concepts of CG to this broader class of problems, but is more complex and has higher memory requirements than CG.

Algorithm 6 BiConjugate Gradient (BiCG)

Require: Matrix A, right-hand side b, preconditioner M, tolerance `itol`, maximum iterations n

Ensure: Approximate solution x to $Ax = b$

$x \leftarrow x^{(0)}$
$r \leftarrow b - Ax$
$\tilde{r} \leftarrow r$
$iter \leftarrow 0$
while $\|r\|/\|b\| > itol$ **and** $iter < n$ **do**
    $iter \leftarrow iter + 1$
    Solve $Mz = r$; Solve $M^T\tilde{z} = \tilde{r}$
    $\rho \leftarrow \tilde{r}^T z$
    if $|\rho| < \varepsilon$ **then**
        exit
    end if
    if $iter = 1$ **then**
        $p \leftarrow z$; $\tilde{p} \leftarrow \tilde{z}$
    else
        $\beta \leftarrow \rho/\rho_{old}$
        $p \leftarrow z + \beta p$; $\tilde{p} \leftarrow \tilde{z} + \beta\tilde{p}$
    end if
    $\rho_{old} \leftarrow \rho$
    $q \leftarrow Ap$; $\tilde{q} \leftarrow A^T\tilde{p}$
    $\omega \leftarrow \tilde{p}^T q$
    if $|\omega| < \varepsilon$ **then**
        exit
    end if
    $\alpha \leftarrow \rho/\omega$
    $x \leftarrow x + \alpha p$
    $r \leftarrow r - \alpha q$
    $\tilde{r} \leftarrow \tilde{r} - \alpha\tilde{q}$
end while

In summary, while direct methods are the workhorses for medium-sized problems due to their reliability, the iterative CG and BiCG methods provide an essential capability for solving the very large sparse linear systems that arise in modern circuit simulation. Their effectiveness is greatly enhanced by the application of suitable preconditioners.

2.4 Sparse Matrices

The efficient solution of linear systems represents the most computationally challenging aspect of circuit simulation, as it dominates the majority of simulation runtime. System matrices, which are produced by MNA, tend to be extremely sparse, as a consequence of element-node connectivity. Most entries are zero because each element connects only to a few nodes in the circuit. From a theoretical point of view, matrix $A \in \mathbb{R}^{n \times n}$ with n_z elements is considered to be sparse when $n_z = O(n)$. From a computational aspect, matrix A is considered sparse when its representation ensures that only elements that are not zero are stored and used in computations. Leveraging this sparsity through specialized data structures and algorithms is essential to achieve more efficient simulation times and memory usage.

2.4.1 Sparse Matrix Storage Formats

- Triplet Format (Coordinate Format)
  - It consists of two vectors i, j of type int and a vector x of type $double$, all of nz size, which contain the row indices, column indices, and the values of the nonzeros of A, respectively (in arbitrary order).
  - It provides a simple intuitive representation suitable for initial insertion of nonzero elements of the sparse matrix.
- Compressed Sparse Column Format
  - It provides memory efficiency and allows fast matrix operations, such as matrix-vector multiplication.
  - It consists of an int vector i of size nz, an int vector p of size $n + 1$, and a $double$ vector x of size nz. The vectors i, p, x contain the row indices, column pointers, and values of the nonzeros of A, respectively.

  - CSparse library of SuiteSparse suite [11], as well as other widely used libraries and GPU-based ones, use both Triplet and Compressed Column Format as their primary storage formats.

- Fill-in Reduction Ordering

  - The factorization of sparse matrices often produces fill-ins, which are new non-zero elements in the factors that were zero in the original matrix. Fill-in dramatically increases memory requirements and computation time, two critical aspects for large circuits.
  - To mitigate fill-ins, a preprocessing reordering is applied. The most popular heuristic is the Approximate Minimum Degree (AMD) algorithm, which tries to minimize the degree (number of nonzeros) of the pivot rows and columns during elimination.

2.4.2 Sparse Direct Solvers

This chapter focuses on the implementation of sparse direct methods (LU and Cholesky) for the large sparse matrices provided by IBM's Power Grid Benchmarks [4]. These two methods use CSparse methods that are available through SuiteSparse library [11] and their fundamental algorithmic structure is presented in 2.3 and 2.4.

The CSparse library provides a concise set of algorithms for solving sparse linear systems that arise in circuit simulation problems. The library implements direct solution methods that are especially effective for the sparse matrices generated by Modified Nodal Analysis (MNA) in circuit simulation. The implementation follows a structured workflow: (1) matrix initialization in triplet format, (2) conversion to compressed column storage, (3) symbolic analysis, (4) numerical factorization, and (5) solution of the linear system. This approach efficiently handles the extreme sparsity of circuit matrices while minimizing fill-in during factorization.

```
cs* sparse_A;
cs* sparse_C;
double* sparse_b;
double* y;

sparse_A = cs_spalloc(rowsA, columnsA, non_zeros, 1, 1);
sparse_A->nz = non_zeros;

sparse_C = cs_compress(sparse_A);
cs_dupl(sparse_C);

css* S = cs_sqr(2, C, 0);
csn* N = cs_lu(C, S, 1.0);

cs_ipvec(N->pinv, b, y, n);      // Apply permutation: y = P*b
cs_lsolve(N->L, y);              // Solve L*y = (P*b)
cs_usolve(N->U, y);              // Solve U*x = y
cs_ipvec(S->q, y, x, n);         // Apply column permutation: x = q*x
```

Listing 2.3: LU factorization using CSparse

- (12) : Performs symbolic analysis for LU factorization. The first parameter (2) specifies the use of column approximate minimum degree (COLAMD) ordering to reduce fill-in during factorization. The third parameter (0) indicates that LU factorization is performed. Symbolic analysis determines the elimination tree and the pattern of non-zero elements in the factors, which optimizes the subsequent numerical factorization.

- (13) : Computes the numerical LU factorization with partial pivoting. The third parameter (1.0) specifies the pivot tolerance for partial pivoting. This function decomposes the matrix into the product of a lower triangular matrix L and an upper triangular matrix U, with permutations in the $N \rightarrow pinv$ vector. The factorization handles numerical stability issues through careful pivot selection.

- (15) : Applies the row permutation to the right-hand side vector b using the permutation vector from the LU factorization.

- (16) : Solves the lower triangular system $Ly = b$.

- (17) : Solves the lower triangular system $Ux = y$.
- (18) : Applies the column permutation to obtain the final solution x.

```
cs* sparse_A;
cs* sparse_C;
double* sparse_b;
double* y;

sparse_A = cs_spalloc(rowsA, columnsA, non_zeros, 1, 1);
sparse_A->nz = non_zeros;

sparse_C = cs_compress(sparse_A);
cs_dupl(sparse_C);

css* S = cs_schol(1, sparse_C);
csn* N = cs_chol(sparse_C, sparse_S);

cs_ipvec(sparse_S->pinv, sparse_b, sparse_y, sparse_rowsA);
cs_lsolve(sparse_N->L, sparse_y);
cs_ltsolve(sparse_N->L, sparse_y);
cs_pvec(sparse_S->pinv, sparse_y, sparse_b, sparse_rowsA);
```

Listing 2.4: Cholesky factorization using CSparse

- (12) : Performs symbolic analysis for Cholesky factorization. The first parameter (1) specifies the use of approximate minimum degree (AMD) ordering to reduce fill-in. This function analyzes the nonzero pattern of the matrix and determines an efficient elimination order, which is crucial for maintaining sparsity during factorization.
- (13) : Computes the numerical Cholesky factorization. This decomposition is only applicable to symmetric positive definite (SPD) matrices, which commonly occur in circuit simulation when the circuit contains only G1 group elements. The function returns a factor L such that $A = LL^T$.
- (15) : Applies the permutation to the right-hand side vector b using the permutation vector from the LU factorization.

- (16) : Solves $Ly = Pb$.
- (17) : Solves $L^T x = y$.
- (18) : Applies the inverse permutation to obtain the final solution x.

2.5 Transient Analysis

Transient analysis computes the time-domain response of a circuit to time-varying voltage and curent source inputs, revealing its dynamic behavior. Unlike DC analysis, which finds a single operating point, transient analysis solves a system of differential-algebraic equations (DAEs) obtained from Modified Nodal Analysis (MNA) over a sequence of time points. Furthermore, Transient analysis requires initial conditions which stem from DC operating point analysis at $t_0 = 0$ and the DC solution $x(t_0)$.

2.5.1 Equation Formulation

In the case of Transient analysis, the arising MNA system is a first order algebraic differential equation system, which must be solved with initial conditions $(x(t_0) = x_0)$ of the following form: $\tilde{G}x(t) + \tilde{C}\dfrac{dx(t)}{dt} = e(t)$

where:

- $\tilde{G} = \begin{bmatrix} A_1 G A_1^T & A_2 \\ A_2^T & 0 \end{bmatrix}$, is the conductances matrix
- $x(t) = \begin{bmatrix} v(t) \\ i_2(t) \end{bmatrix}$, is the vector of unknown node voltages and G2 element currents
- $\tilde{C} = \begin{bmatrix} A_1 C A_1^T & 0 \\ 0 & -L \end{bmatrix}$, is the capacitances and inductances matrix
- $e(t) = \begin{bmatrix} -A_1 s_1(t) \\ s_2(t) \end{bmatrix}$, is the stimulation vector of voltage sources and current sources

2.5.2 Numerical solution methods for ODE

- Backward Euler approximation (BE):

$$(\tilde{G} + \frac{1}{h}\tilde{C})x(t_k) = e(t_k) + \frac{1}{h}\tilde{C}x(t_{k-1}) \ , \ k = 1, 2, ..., m$$

- Trapezoidal approximation (TR):

$$(\tilde{G} + \frac{2}{h}\tilde{C})x(t_k) = e(t_k) + e(t_{k-1}) - (\tilde{G} - \frac{2}{h}\tilde{C})x(t_{k-1}) \ , \ k = 1, 2, ..., m$$

In practice, TR usually gives much better accuracy than Backward Euler for the same time step size, at the cost of solving a slightly different linear system. Although Backward Euler may be preferred for very stiff circuits to avoid numerical oscillations.

2.5.3 Element Stamping

The algorithm can be studied in depth in Farid's N. Najm book, chapter 4 Solution of Nonlinear Algebraic Circuit Equations [5].

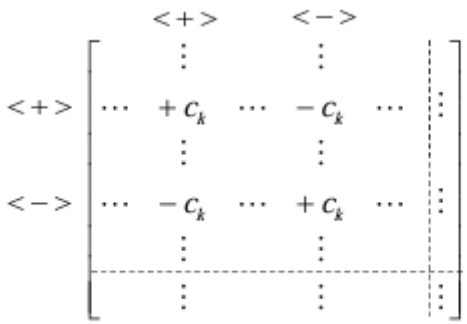

Figure 2.2: Contributions of capacitances to $\tilde{C}$

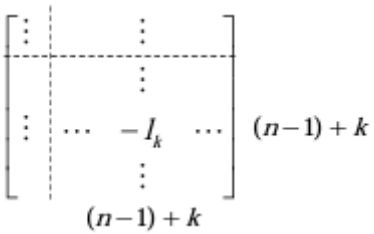

Figure 2.3: Contributions of inductances to $\tilde{C}$

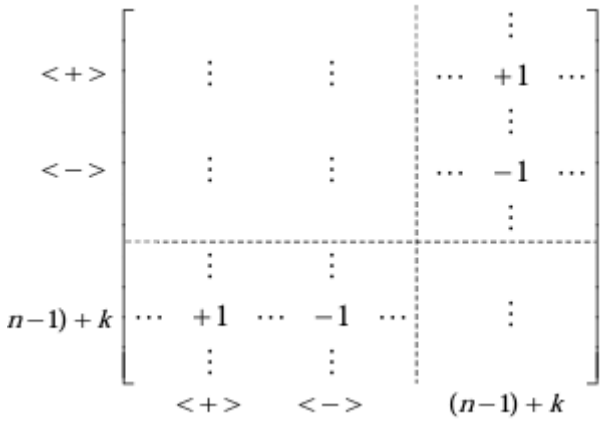

Figure 2.4: Contributions of inductances to $\tilde{G}$

2.5.4 Source Modeling

Transient analysis supports various time-dependent sources with different waveform types:

$$\textbf{EXP:}\begin{cases} i_1, & 0 \le t \le td1 \\ i_1 + (i_2 - i_1)\left(1 - e^{-(t-td1)/tc1}\right), & td1 \le t \le td2 \\ i_1 + (i_2 - i_1)\left(e^{-(t-td2)/tc2} - e^{-(t-td1)/tc1}\right), & td2 \le t \le t_{\text{fin}} \end{cases}$$

$$\textbf{SIN:}\begin{cases} i_1 + i_a \cdot \sin\left(2\pi \frac{\varphi}{360^\circ}\right), & 0 \le t \le td \\ i_1 + i_a \cdot \sin\left(2\pi f_r (t - td) + 2\pi \frac{\varphi}{360^\circ}\right) \cdot e^{-(t-td)d_f}, & td \le t \le t_{\text{fin}} \end{cases}$$

$$\textbf{PULSE:}\begin{cases} i_1, & 0 \le t \le td \\ i_1 \to i_2 \text{ (linearly)}, & td + k \cdot per \le t \le td + tr + k \cdot per \\ i_2, & td + tr + k \cdot per \le t \le td + tr + pw + k \cdot per \\ i_2 \to i_1 \text{ (linearly)}, & td + tr + pw + k \cdot per \le t \le td + tr + pw + tf + k \cdot per \\ i_1, & td + tr + pw + tf + k \cdot per \le t \le td + per + k \cdot per \end{cases}$$

$$(k = 0, 1, 2, \dots)$$

2.6 AC Analysis

AC analysis computes the small-signal frequency response of a circuit by solving the system equations in the frequency domain. All independent sources are sinusoidal at a common frequency ω, thus converting time domain equations into phasor domain ones. The time-based MNA system transforms to a complex MNA system. In addition, this type of analysis complements the time-domain perspective of Transient analysis.

2.6.1 Complex MNA Formulation

The system equations of G1 and G2 elements, which make use of the phasor format, are represented in the sinusoidal steady state as:

- $\hat{i}_1(t) = (G + j\omega C)\hat{u}_1(t) + \hat{s}_1 \quad (1)$

- $\hat{u}_2 = j\omega L\hat{i}_2 + \hat{s}_2 \quad (2)$

Substituting KVL and KCL equations in (1) and (2), the following complex MNA system emerges:

$$\begin{bmatrix} A_1(G + j\omega C)A_1^T & A_2 \\ A_2^T & -j\omega L \end{bmatrix} \begin{bmatrix} \hat{v} \\ \hat{i_2} \end{bmatrix} = \begin{bmatrix} -A_1\hat{s_1} \\ \hat{s_2} \end{bmatrix}$$

where:

- $\tilde{G(j\omega)} = \begin{bmatrix} A_1(G + j\omega C)A_1^T & A_2 \\ A_2^T & -j\omega L \end{bmatrix}$

- $\hat{x} = \begin{bmatrix} \hat{v} \\ \hat{i_2} \end{bmatrix}$

- $\hat{e} = \begin{bmatrix} -A_1\hat{s_1} \\ \hat{s_2} \end{bmatrix}$

2.6.2 Element Stamping

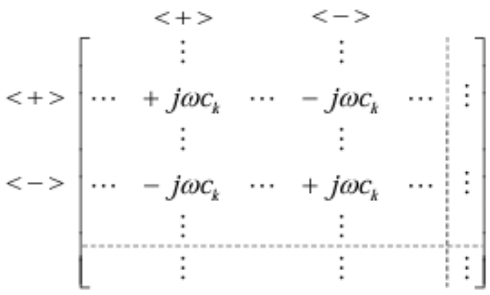

Figure 2.5: Contributions of capacitances to $G(\tilde{j\omega})$

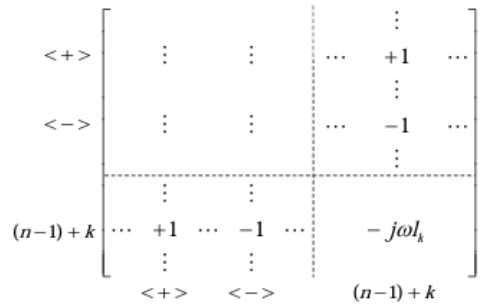

Figure 2.6: Contributions of inductances to $G(\tilde{j\omega})$

2.6.3 Solving Complex Systems

Complex linear systems can be solved via modifying the factorization and iterative solution algorithms, analyzed in previous chapters, as follows:

- Sparse LU factorization:
  - For sparse complex matrices, the CXSparse extension of CSparse, which transforms the functionality used in 2.3, into the corresponding complex format, without changing the algorithmic steps.
- BiCG:
  - In order for BiCG to be implemented for AC analysis, the following replacements are made in 6:

- $\tilde{r}^T z, \ \tilde{p}^T q$ are replaced by $\tilde{r}^H z, \ \tilde{p}^H q$
- $A^T \tilde{p}$ is replaced by $A^H \tilde{p}$
- $M^T \tilde{z} = \tilde{r}$ is replaced by $M^H \tilde{z} = \tilde{r}$
- $\tilde{p} = \tilde{z} + \beta \tilde{p}, \ \tilde{r} = \tilde{r} - \alpha \tilde{q}$ are replaced by $\tilde{p} = \tilde{z} + \overline{\beta} \tilde{p}, \ \tilde{r} = \tilde{r} - \overline{\alpha} \tilde{q}$

- Cholesky factorization and CG method, can not be used in AC analysis, due to the fact that $\tilde{G}(j\omega)$ is not Hermittian (i.e. $\tilde{G}(j\omega) \neq \tilde{G}^H(j\omega)$).

Chapter 3

GPU-based Solvers

3.1 Introduction

The computational bottleneck of modern circuit simulation lies in the solution of large sparse linear systems of the form $\mathbf{Ax} = \mathbf{b}$, which most of the time are generated by the Modified Nodal Analysis (MNA) method. As integrated circuits grow in complexity; the size of these systems scales accordingly, often reaching millions of nodes. CPU-based direct solvers, using highly optimized libraries like CSparse [11], face fundamental limitations due to their reliance on sequential algorithms and memory bandwidth constraints, leading to prohibitively long runtime simulations.

GPU architectures, with their massive thread-level parallelism consisting of thousands of efficient cores, are exceptionally well-suited for the data-parallel operations that dominate linear algebra computations. Furthermore, GPUs provide much higher device-memory bandwidth than commodity CPUs, making them attractive for the most demanding solution phases (symbolic and numeric factorization).

This chapter details the implementation and evaluation of GPU-accelerated solvers, specifically using the NVIDIA cuDSS (CUDA [12] Direct Sparse Solver) library [13], to tackle the circuit netlists from IBM's Power Grid benchmarks. Three distinct implementation strategies are proposed: a baseline approach, using cuDSS functions and data types and two methods that rely on two advanced feature modes of the library. The first is the Multi-GPU Multi-Node (MGMN) mode, which allows multiple GPU devices to use numerous processes and distributed calculations simultaneously. The second one is a Multi-Threaded (MT) CPU-controlled model, which adds CPU-side threading (via a threading layer such as GNU OpenMP),

which can accelerate host-side phases even in single-GPU runs.

NVIDIA's cuDSS (CUDA Direct Sparse Solver) is a production GPU library that implements a modern, parallel direct solver pipeline (symbolic factorization → numerical factorization → solving) and reports substantial gains versus CPU-only solvers. Although absolute gains depend on the matrices and platform, these results underscore that GPU-accelerated direct solvers are a practical alternative for the sparse systems arising in circuit simulation.

3.2 cuSOLVER (supplementary)

This section is provided only for completeness purposes and it does not constitute a competitive or recommended method for this work. It summarizes why the legacy cuSOLVER [14] sparse path is not a viable LU based solution for very large Modified Nodal Analysis (MNA) systems and why its QR route underperforms compared with both CPU baselines and modern GPU direct solvers. Results of this approach are intentionally excluded from 4.

NVIDIA's current guidance is clear; the sparse-direct functionality historically present in cuSolverSp is deprecated, and it is strongly recommended to migrate to cuDSS for sparse direct solves. The official cuSOLVER documentation explicitly notes the deprecation and points to cuDSS with transition samples. In addition, cuDSS's own documentation reiterates that relation and migration path. Concretely, cuSOLVER today provides the following:

- Device sparse QR front-end $cusolverSpDcsrlsvqr()$ for certain problem classes.
- Host sparse LU factorization $csrlsvluHost()$,which means that the LU path runs on the CPU, not on the GPU.

3.2.1 cuSOLVER QR

Besides this library being outdated and currently no longer supported by NVIDIA, a simple implementation was developed and tested, as shown in 7 proving the scaling and performance inefficiency mainly for largest circuits. Timing improvement in comparisson with CSparse method, was also not observed even for smaller circuits (i.e., ibmpg1). All three reordering methods (symrcm, symamd, metis) were tested, with metis showing a slight speedup.

Due to these limitations, this thesis does not present performance charts for the cuSOLVER QR method in 4. The method proved inefficient in timing and unable to complete the solving workflow of the largest benchmarks (e.g., ibmpg3 from IBM's Benchmarks) under the same constraints, whereas cuDSS offers precisely the features needed to address scale.

Algorithm 7 cuSOLVER QR decomposition

Require: CSC matrix A_{CSC}, RHS b on host, nnz

Ensure: Overwrite $b \leftarrow x$ (host) solving $Ax = b$

Host_preprocessing

$A_{\text{CSR}} \leftarrow$ CS_Transpose(A_{CSC})

Device_allocation_and_H2D

Allocate `d_rowPtr[`m`+1]`, `d_colInd[nnz]`, `d_vals[nnz]`, `d_b[`m`]`, `d_x[`m`]`

Copy `rowPtr`, `colInd`, `vals` $\rightarrow$ device

copy b (host) $\rightarrow$ `d_b`

cuSOLVER/cuSPARSE setup

Create `cusolverSpHandle_t()` and `cusparseMatDescr_t()`

with `CUSPARSE_MATRIX_TYPE_GENERAL`, `CUSPARSE_INDEX_BASE_ZERO`

QR solve

Choose tolerance `tol` (e.g., 10^{-12}) and `reorder` (e.g., `symamd, metis`)

`cusolverSpDcsrlsvqr(` m, nnz, `descrA`, `d_vals`, `d_rowPtr`, `d_colInd`, `d_b`, `tol`, `reorder`, `d_x`, `singularity )`

D2H

Copy `d_x` $\rightarrow$ b (host)

Destroy cuSOLVER/cuSPARSE objects, free device memory

3.2.2 cuDSS: the practical replacement

cuDSS is NVIDIA's actively developed sparse direct solver stack with the three-phase workflow (reordering/symbolic, factorization, solve), plus extended capabilities for large circuit-scale problems: MGMN (multi-GPU, multi-node), multi-threaded (MT) host-side acceleration. A complete implementation of the developed methods, based on these advanced features, is presented in the following sections.

3.3 Architectural Overview: CUDA and cuDSS

3.3.1 CUDA Parallel Computing Platform

NVIDIA's CUDA platform is a parallel computing architecture that enables usage of NVIDIA GPUs for general-purpose processing. The key to its performance lies in its hierarchy of many lightweight parallel threads, which are organized into thread blocks that execute on streaming multiprocessors (SMs). A typical NVIDIA GPU, like the nvidia A10 [15] used for testing the developed methods, comprises multiple SMs, each containing multiple of CUDA cores. In addition thousands of threads are executed concurrently, grouped into warps (32 threads) that execute the same instruction synchronously. This Single Instruction Multiple Thread (SIMT) model is ideal for applying the same operation (e.g. an arithmetic operation on a matrix element) to large data sets in parallel.

3.3.2 cuDSS Library

cuDSS is a high-performance GPU-accelerated library for solving sparse linear systems using direct methods. It is designed to handle the very large sparse matrices found in computational science and engineering simulations, exactly like those in circuit simulation. The library is designed around a three-phase workflow that must always be executted in the following order:

- CUDSS_PHASE_ANALYSIS

  Analysis(Reordering and symbolic factorization): Involves reordering the matrix to minimize fill-in (using algorithms like AMD [16] or METIS [17]), building the elimination tree, and determining the optimal parallel execution plan for the subsequent numerical phase on the GPU. Reordering and Symbolic factorization can be executed separately in different phases but it that specific order.

- CUDSS_PHASE_FACTORIZATION

  Factorization(Numerical factorization): Performs the actual LU or Cholesky decomposition. This phase is highly compute-intensive and benefits tremendously from GPU acceleration.

- CUDSS_PHASE_REFACTORIZATION

Refactorization(Optional): Numerical refactorization, used if the reordering algorithm is specified and not the default one.

- CUDSS_PHASE_SOLVE

  Solving: Uses factored matrices to solve for one or more right-hand sides (RHS) using a combination od forward,backward substitution and diagonal solve. The full solving can only ,though this is the optimal solving technique.

simple cuDSS Workflow ($Ax = b$)

0. Prerequisites: User-allocated device memory buffers for matrix A and vectors $x,\ b$
1. Initialize library handle: `cudssCreate()`
2. Create matrix wrappers: `cudssMatrixCreateCsr()` for A , `cudssMatrixCreateDn()` for b and x
3. Create config objects: `cudssConfigCreate()`, `cudssConfig_t`
4. Create data objects: `cudssDataCreate()`, `cudssData_t`
5. (Optional) Adittional Settings: `cudssSetStream()`, `cudssConfigSet()`, `cudssDataSet()`
6. Analysis: `cudssExecute(..., CUDSS_PHASE_ANALYSIS, ...)`
7. Factorization: `cudssExecute(..., CUDSS_PHASE_FACTORIZATION, ...)`
8. Solve: `cudssExecute(..., CUDSS_PHASE_SOLVE, ...)`
9. Destroy objects: `cudssMatrixDestroy()`, `cudssDataDestroy()`, `cudssConfigDestroy()`
10. Destroy library handle: `cudssDestroy()`

3.4 Baseline cuDSS method

The solution of sparse linear systems represents the most computationally intensive phase in circuit simulation. Although the CSparse library provides robust CPU-based factorization, its execution time becomes prohibitive for the massive matrices (dimension > 100,000) found in modern circuit analysis. NVIDIA's cuDSS (CUDA Direct Sparse Solver) library is designed to offload the entire direct solving process; symbolic analysis, numerical fac-

torization and full solving phase onto the GPU, leveraging its massive parallelism and high memory bandwidth to achieve significant performance gains.

This section details the implementation of a baseline solver that uses cuDSS to accelerate the solution of sparse systems $Ax = b$ arising from MNA. The implementation follows a structured pipeline: (1) matrix format conversion from CSC to CSR, (2) device memory allocation and data transfer from Host to Device, (3) cuDSS handle and descriptor configuration, and (4) the execution of the three core phases of direct solving. This baseline implementation serves as the foundation for the more advanced multi-GPU and multi-threaded variants discussed in subsequent sections. A more precise implementation of the algorithm is can be found at 8.

3.4.1 Algorithmic Description and Implementation

1. **Input:** A square matrix in Compressed Sparse Column format, dimension n of A, right-hand side vector b
   Output: Solution vector x

2. **Matrix Format conversion: CSC to CSR**

   - cuDSS requires the input matrix in Compressed Sparse Row (CSR) format for optimal performance.
   - The CSparse function $cs\_transpose()$ is used to convert the matrix from CSC to CSR. This is a necessary pre-processing step performed on the CPU.

3. **Device Memory Allocation and Data Transfer**

   - Allocate GPU memory for the matrix (CSR arrays: row pointers, column indices, values) and the vectors (RHS b, solution x).
   - Copy the CSR matrix data and the RHS vector from host (CPU) to device (GPU) memory. This is a synchronous and blocking operation.
   - Performance Note: This data transfer overhead is amortized over the subsequent computationally intensive phases.

4. **Handle and Descriptor Configuration**

   - Handle Creation ($cudssCreate()$): Initializes the cuDSS library context.

- Config Creation ($cudssConfigCreate()$): Creates a configuration object to set solver parameters.
- Data Creation ($cudssDataCreate()$): Creates an object to hold the internal data structures (factors, permutation vectors) generated during the solving process.
- Matrix Descriptors ($cudssMatrixCreateCsr(), cudssMatrixCreateDn()$): Wrap the raw device pointers for the matrix and vectors into cuDSS descriptor objects. This abstraction allows cuDSS to interpret the data correctly.
  - **CUDSS_MTYPE_GENERAL**: Specifies a general matrix (non-symmetric).
  - **CUDSS_BASE_ZERO**: Indicates the CSR indexing uses a 0-based base.
- $cudssConfigSet()$ allows tuning the solver; the choice of reordering algorithm (e.g. CUDSS_ALG_DEFAULT, CUDSS_ALG_METIS) can significantly impact the fill-in and thus the factorization time and memory usage. This is a critical knob for optimizing performance across different matrix types.

5. **Three-Phase Workflow**

(α′) Reordering and Symbolic Factorization (CUDSS_PHASE_ANALYSIS):

- Performs symbolic analysis on the matrix structure. This includes finding a fill-reducing permutation (e.g., using the METIS algorithm, specified here by **CUDSS_ALG_DEFAULT**) and analyzing the dependency graph to plan the parallel numerical factorization.
- While this phase often involves sequential steps, cuDSS offloads suitable tasks to the GPU, but it may still leverage the CPU. **CUDSS_ALG_DEFAULT** ,allows the library to select the best heuristic for the given matrix.

(β′) Numerical Factorization (CUDSS_PHASE_FACTORIZATION):

- Computes the sparse LU factorization on the GPU. This is the most computationally intensive phase and benefits most from massive parallelism.
- The factorization of independent sub-matrices (supernodes) is scheduled across thousands of GPU threads, dramatically accelerating this operation.

(γ′) Solving (CUDSS_PHASE_SOLVE):

- Solves the system $LUx = Pb$ using forward, backward substitution and diagonal solve. The permutations from the analysis phase are applied automatically.
- The solution vector x is copied from the device (d_x) back to the host b.

3.4.2 Expected Performance

The baseline cuDSS implementation is expected to have a significant speedup over the CPU-based CSparse solver, primarily due to the parallelization of the numerical factorization phase. The analysis phase may show a more modest speedup or even be slower for smaller matrices due to its partially sequential nature and data transfer overhead. The solve phase, while faster, will typically show a smaller speedup than factorization due to its lower arithmetic intensity, especially for larger circuit designs. The overall speedup is a product of these three phases and the data transfer costs, which this baseline implementation successfully minimizes by keeping all data on the GPU between phases.

3.5 Multithreaded cuDSS method

While the baseline cuDSS implementation effectively leverages GPU parallelism for the numerical phases of sparse factorization, the entire process remains bound to a single CPU thread for control flow, data preparation, and API calls. This can become a bottleneck in complex simulation workflows. The cuDSS Multi-Threaded (MT) mode addresses this limitation. It allows the cuDSS library to be called concurrently from multiple host CPU threads, enabling a hybrid parallel paradigm. In addition, it also allows phases that perform graph processing and metadata preparation (e.g., reordering and symbolic analysis) to leverage the CPU cores, reducing the wall-time before numeric factorization launches on the GPU. Both factorizations and solving remain GPU kernels; MT primarily shortens the host-driven portions and any CPU-side orchestration. This is particularly powerful for:

- Pipelined Execution: Overlapping the data preparation and transfer for the next system with the GPU's solution of the current system, hiding host-side overhead effectively.
- Integration with Multi-Threaded Simulation Frameworks: Allowing the linear solver to fit seamlessly into a broader CPU-parallel simulation environment without becoming

a serialization point.

3.5.1 Algorithmic Description and Implementation

Algorithm 9 shows in detail the multi-threaded variant. Its structure and workflow sequence are similar to the baseline but includes critical thread configuration, such as choosing the threading layer library and setting optimal environment tuning. OpenMP library [18] was used to test the performance of this method along with the specified environment variables for threading control and stable affinity: OMP_NUM_THREADS, OMP_PLACES=cores, OMP_PROC_BIND=close. The following steps differentiate this method from the baseline method which was presented in the previous section.

1. **Device Selection**

   - Explicitly set the target GPU device via $cudaSetDevice()$. This is crucial in multi-GPU systems or when multiple threads manage different devices.

2. **cuDSS Handle and Threading Layer Configuration**

   - This is the core addition for multi-threading. After creating the $cudssHandle\_t$ and $cudssConfig\_t$, the function explicitly loads a threading layer library.
   - The library $libcudss\_mtlayer\_gomp.so$ is an OpenMP-based layer that makes the cuDSS context managed by this handle thread-safe. This allows this same handle to be used from multiple CPU threads, or for multiple handles to co-exist without interference.
   - NVIDIA also provides different layers for different threading runtimes (e.g., one for Intel's TBB).

3.5.2 Expected Performance

This Multithreaded method is expected to present an overall acceleration in comparisson with the implemented cuDSS baseline method; especially with the largest gains to appear on bigger matrices. For example, if the host application creates M CPU threads (e.g., M=16), it can assign batches of problems to each thread. Each thread manages its own cuDSS handle and GPU stream, transforming the GPU-accelerated solver from a high-performance serial component into a scalable, throughput-oriented parallel component.

3.6 MGMN cuDSS method

The computational demands of large scale circuit simulation problems with matrix dimensions exceeding 1 million nodes, push beyond the memory and processing capabilities of even the most powerful single GPU. To tackle these challenging problems, the cuDSS Multi-GPU Multi-Node (MGMN) mode provides a distributed, massively parallel memory approach to sparse direct solving. This implementation leverages a cluster of compute nodes, each equipped with one or more GPUs, to collaboratively factor and solve a single large sparse linear system by distributing the matrix data and computational workload across all available devices. This implemented method is essential for overcoming two fundamental limitations of single GPU solvers:

- Memory Capacity: The aggregate global memory of multiple GPUs can accommodate the massive storage requirements for the factors of billion-scale matrices, which would be impossible to fit on a single device.
- Computational Throughput: By parallelizing the factorization across many GPUs, the MGMN approach can achieve solve times that would be unattainable with a single processor, effectively leveraging the combined FLOPs and memory bandwidth of an entire cluster.

The following section details the implementation of a distributed solver using cuDSS's MGMN capabilities in conjunction with the Message Passing Interface (MPI), Open MPI [19] in this implementation. The implementation follows a row-based 1D domain decomposition strategy, where each MPI rank (typically mapped to one GPU) is responsible for a contiguous block of rows of the global matrix. The algorithm is presented in detail in 12 and 13 as well as in 11 and 10 , which constitute the two helper functions for extracting the rows $[firstRow, lastRow]$ from the global CSR matrix (including all non-zero elements in those rows), and computing each rank's contiguous row interval $[first, last]$ respectively.

3.6.1 Algorithmic Description and Implementation

1. **Input:** A square matrix in Compressed Sparse Column format, dimension n of A, right-hand side vector b (valid only in root rank)
   Output: Solution vector x stored in vector b on the root rank (rank 0).

2. **MPI Initialization and Process Layout**

   - Initialize the MPI environment if it hasn't been already. Request MPI_THREAD_MULTIPLE support to ensure compatibility with other threading models.
   - Determine the world size $(worldSize)$ and rank $(worldRank)$ within MPI_COMM_WORLD.
   - Create a new communicator $(nodeComm)$; grouping ranks on the same physical node. This is used to manage GPU assignments per node.

3. **GPU Device Assignment**

   - Determine the number of available GPUs per node $(ngpu)$ and the local rank within the node.
   - Enforce one-to-one mapping between MPI ranks and GPUs. Each rank sets its CUDA device to $(localRank \ \% \ ngpu)$. This ensures optimal GPU utilization and avoids resource contention.

4. **Global Matrix Conversion and Distribution**

   - The root rank (rank 0) converts the global matrix from CSC to CSR format.
   - The matrix is partitioned into contiguous row blocks across all MPI ranks using a balanced block distribution. The helper function $compute\_row\_block()$ calculates the start and end row for each rank.

5. **Local CSR Slice Construction**

   - Each rank constructs its local portion of the global matrix. The helper function $build\_csr\_slice()$ extracts the rows $[firstRow, lastRow]$ from the global CSR matrix, including all non-zero elements in those rows.
   - The right-hand side vector b is broadcast from the root rank to all other ranks to ensure each process has a complete copy.

6. **Device Memory Allocation and Data Transfer**

   - Each rank allocates GPU memory for its local matrix slice and the global vectors $b,\ x$.

- The local CSR data (row pointers, column indices, values) and the RHS vector are copied to the respective GPU device.

7. **cuDSS Configuration for Distributed Solving**

   - Initialize cuDSS and configure it to use the MPI communicator for inter-rank communication. This is done by loading the appropriate communication layer library (e.g., $libcudss\_commlayer\_openmpi.so$) and passing the MPI communicator to cuDSS.
   - Create cuDSS matrix descriptors for the distributed matrix and vectors. Specifically, inform cuDSS of the data distribution by calling $cudssMatrixSetDistributionRow1d()$ with the global row indices owned by this rank.

8. **Distributed Execution of Solving Phases**

   - Execute the three standard cuDSS phases (ANALYSIS, FACTORIZATION, SOLVE). The key difference is that cuDSS now handles all inter-node and inter-GPU communication internally during these phases to coordinate the distributed factorization and solution process.
   - After solving, the solution vector x is distributed across all GPUs. The root rank gathers the entire solution vector to its device memory and then copies it to the host, overwriting the input vector b.

Algorithm 8 Baseline cuDSS Sparse LU on a Single GPU

Require: CSC matrix A_{CSC} (CSparse `cs* C`), RHS b (host), sizes $m = \text{rows}(A_{\text{CSC}})$, nnz

Ensure: Solution x overwriting b on host

1: **Host_prep:** $A_{\text{CSR}} \leftarrow \text{CS\_Transpose}(A_{\text{CSC}})$; **exit** on failure

2: **Extract:** `rowPtr[0..m]`, `colInd[0..nnz)`, `vals[0..nnz)` from A_{CSR}

3: **Device_alloc:** `cudaMalloc→d_rowPtr[m+1], d_colInd[nnz], d_vals[nnz], d_b[m], d_x[m]`

4: **H2D_copies:** `rowPtr→d_rowPtr, colInd→d_colInd, vals→d_vals,` $b \rightarrow$`d_b`

5: **cuDSS_objects:** `cudssCreate(handle), cudssConfigCreate(config), cudssDataCreate(handle,data)`

6: **Matrix_wrappers:** `cudssMatrixCreateCsr(cudssA, m, m, nnz, d_rowPtr, nullptr, d_colInd, d_vals, CUDA_R_32I, CUDA_R_64F, CUDSS_MTYPE_GENERAL, CUDSS_MVIEW_FULL, CUDSS_BASE_ZERO)`

7: `cudssMatrixCreateDn(cudssB, m, 1, m, d_b, CUDA_R_64F, CUDSS_LAYOUT_COL_MAJOR)`

8: `cudssMatrixCreateDn(cudssX, m, 1, m, d_x, CUDA_R_64F, CUDSS_LAYOUT_COL_MAJOR)`

9: **tuning(optional):** `cudssConfigSet(config, CUDSS_CONFIG_REORDERING_ALG, CUDSS_ALG_DEFAULT, sizeof(alg))`

10: **Analysis:** `cudssExecute(handle, CUDSS_PHASE_ANALYSIS, config, data, cudssA, cudssX, cudssB)`

11: **Factorization:** `cudssExecute(handle, CUDSS_PHASE_FACTORIZATION, config, data, cudssA, cudssX, cudssB)`

12: **Solve:** `cudssExecute(handle, CUDSS_PHASE_SOLVE, config, data, cudssA, cudssX, cudssB)`

13: **D2H copy:** `cudaMemcpy(` $b \leftarrow$ `d_x )`

14: **Destroy_cuDSS:** `cudssMatrixDestroy(cudssA), cudssMatrixDestroy(cudssB), cudssMatrixDestroy(cudssX), cudssDataDestroy(handle, data), cudssConfigDestroy(config), cudssDestroy(handle)`

15: **Free:** `cudaFree(d_b), cudaFree(d_x), cudaFree(d_rowPtr), cudaFree(d_colInd), cudaFree(d_vals), cs_spfree(`A_{CSR}`)`

Algorithm 9 Multithreaded cuDSS method

Require: Same inputs as Alg. 8: CSC matrix C (CSparse `cs*`), RHS b (host)

Ensure: Solution x overwriting b on host

1: **Reuse_from_Alg. 8:** CSC→CSR via `cs_transpose()`
`device allocations, H2D copies, create handle/config/data`
`create CSR/Dense matrix wrappers (A,B,X)`
2: **Enable_MT_threading_layer:**
`cudssSetThreadingLayer(handle, "libcudss_mtlayer_gomp.so")`
3: **OpenMP_runtime:** `set OMP_NUM_THREADS={8, 12, 16},`
`OMP_PLACES=cores, OMP_PROC_BIND=close`
4: **Reordering (same as Alg. 8):**
`cudssConfigSet(config, CUDSS_CONFIG_REORDERING_ALG,`
`&CUDSS_ALG_DEFAULT, sizeof(alg))`
5: **Execute_phases (same API as Alg. 8):**
6: `cudssExecute(handle, CUDSS_PHASE_ANALYSIS, config,`
`data, cudssA, cudssX, cudssB)`
7: `cudssExecute(handle, CUDSS_PHASE_FACTORIZATION, config,`
`data, cudssA, cudssX, cudssB)`
8: `cudssExecute(handle, CUDSS_PHASE_SOLVE, config, data,`
`cudssA, cudssX, cudssB)`
9: **Copy_back:** `cudaMemcpy(b, d_x, m*sizeof(double),`
`cudaMemcpyDeviceToHost)`
10: **Destroy_and_Free:** `destroy(cudssA, cudssB, cudssX, data,`
`config, handle),free(device_buffers)`

Algorithm 10 Row-1D Block Partitioning (compute_row_block())

Require: totalRows N, worldRank r, worldSize P

$base \leftarrow \lfloor N/P \rfloor, \quad rem \leftarrow N \bmod P$

$$add \leftarrow \begin{cases} 1 & \text{if } r < rem \\ 0 & \text{otherwise} \end{cases}$$

$offset \leftarrow r \cdot base + \min(r, rem)$

$\texttt{first} \leftarrow offset$

$\texttt{last} \leftarrow \texttt{first} + base + add - 1$

Algorithm 11 Build Local CSR Slice (build_csr_slice())

Require: Global_CSR A, $\texttt{rowPtr}_{local}[0..N]$, $\texttt{colInd}_{local}[0..\texttt{nnz})$, $\texttt{vals}_{local}[0..\texttt{nnz})$, interval $[\texttt{firstRow}, \texttt{lastRow}]$

if $\texttt{lastRow} < \texttt{firstRow}$ **then**
  $\texttt{rowPtr}_{local} \leftarrow [0]$, $\texttt{colInd}_{local} \leftarrow [\,]$, $\texttt{vals}_{local} \leftarrow [\,]$, **return**
end if
$n_{local} \leftarrow \texttt{lastRow} - \texttt{firstRow} + 1$
Resize $\texttt{rowPtr}_{local}[0..n_{local}]$, $\texttt{rowPtr}_{local}[0] \leftarrow 0$
$rowPtr_{global} \leftarrow (A \to p)$, $colInd_{global} \leftarrow (A \to i)$, $vals_{global} \leftarrow (A \to x)$
$\texttt{nnz}_{local} \leftarrow \sum_{i=\texttt{firstRow}}^{\texttt{lastRow}} \left(\texttt{rowPtr}_{global}[i{+}1] - \texttt{rowPtr}_{global}[i]\right)$
Resize $\texttt{colInd}_{local}[0..\texttt{nnz}_{local})$, $\texttt{vals}_{local}[0..\texttt{nnz}_{local})$
$\texttt{idx} \leftarrow 0$
for $i = \texttt{firstRow}$ **to** $\texttt{lastRow}$ **do**
  $start \leftarrow \texttt{rowPtr}_{global}[i]$, $end \leftarrow \texttt{rowPtr}_{global}[i{+}1]$, $len \leftarrow end - start$
  if $len > 0$ **then**
    copy $\texttt{colInd}_{global}[start..end)$ to $\texttt{colInd}_{local}[\texttt{idx}..\texttt{idx}{+}len)$
    copy $\texttt{vals}_{global}[start..end)$ to $\texttt{vals}_{local}[\texttt{idx}..\texttt{idx}{+}len)$
  end if
  $\texttt{idx} \leftarrow \texttt{idx} + len$
  $\texttt{rowPtr}_{local}[(i - \texttt{firstRow}) + 1] \leftarrow \texttt{idx}$
end for

Algorithm 12 cuDSS MGMN

Require: Global CSC matrix A (CSparse `cs*`), right-hand side b on rank 0

Ensure: On rank 0: solution x overwrites b

1: **MPI_init:** `MPI_Init_thread(..., MPI_THREAD_MULTIPLE, ...)`

2: **Ranks_and_topology:** `worldRank, worldSize, localRank, localSize`
`MPI_Comm_split_type(..., MPI_COMM_TYPE_SHARED, ...)`

3: **Device_mapping (1 rank / GPU):** `cudaGetDeviceCount(ngpu)`
`cudaSetDevice(localRank % ngpu)`

4: **Reuse_from_Alg. 8:** CSC→CSR via `cs_transpose()`

5: **Globals:** $rows_{global} \leftarrow (A_{\text{csr}} \rightarrow m)$, $cols_{global} \leftarrow (A_{\text{csr}} \rightarrow n)$,
$\texttt{nnz}_{\text{global}} \leftarrow (A_{\text{csr}} \rightarrow p[m])$

6: **Row1D_partition:** $(\texttt{firstRow}, \texttt{lastRow}) \leftarrow$ compute_row_block$(m, \texttt{worldRank}, P)$ *(Alg. 10)*

7: **Local_CSR_slice (host):**
$(\texttt{rowPtr}, \texttt{colInd}, \texttt{vals}) \leftarrow$ build_csr_slice$(A_{\text{csr}}, \texttt{firstRow}, \texttt{lastRow})$ *(Alg. 11)*

8: `nnz_loc` ← `colInd.size()`

9: **RHS_distribution:** on rank 0, set `b_full`← b
on others, allocate `b_full`$[m]$; `MPI_Bcast`(`b_full`, ..., `MPI_DOUBLE`, ...)

10: **cuDSS_setup (with comm layer):**

11: `cudssCreate(handle), cudssConfigCreate(config),`
`cudssDataCreate(handle,data)`

12: `cudssSetCommLayer(handle,"libcudss_commlayer_openmpi.so")`

13: `cudssDataSet(handle,data, CUDSS_DATA_COMM, MPI_COMM_WORLD,`
`sizeof(MPI_Comm))`

Algorithm 13 cuDSS MGMN (continued)

Matrix_wrappers:

```
cudssMatrixCreateCsr(A, m,n,nnz_glob, d_rowPtr,
nullptr, d_colInd, d_vals, CUDA_R_32I, CUDA_R_64F,
CUDSS_MTYPE_GENERAL, CUDSS_MVIEW_FULL, CUDSS_BASE_ZERO)
cudssMatrixCreateDn(B, m,1,m, d_b, CUDA_R_64F,
CUDSS_LAYOUT_COL_MAJOR)
cudssMatrixCreateDn(X, m,1,m, d_x, CUDA_R_64F,
CUDSS_LAYOUT_COL_MAJOR)
cudssMatrixSetDistributionRow1d(A, firstRow, lastRow)
```

Analysis (all ranks):

```
cudssExecute(handle, CUDSS_PHASE_ANALYSIS, config, data,
A, X, B)
```

Factorization (all ranks):

```
cudssExecute(handle, CUDSS_PHASE_FACTORIZATION, config,
data, A, X, B)
```

Solve (all ranks):

```
cudssExecute(handle, CUDSS_PHASE_SOLVE, config, data, A,
X, B)
```

Result on rank 0: copy `d_x`→host and overwrite b

Chapter 4

Results

4.1 Introduction

In this chapter, performance profiling analysis of the implemented methods is presented in detail, accompanied by the corresponding figures. Three from the five implemented solving workflows of $Ax = b$ systems arising from Modified Nodal Analysis (MNA) are evaluated:

1. CPU baseline: CSparse-based, single-threaded, presented in 2.4.2.

2. cuDSS baseline: cuDSS-based, single-threaded, presented in 3.4.

3. cuDSS MT: cuDSS-based, multi-threaded, same algorithmic pipeline as cuDSS baseline, with an OpenMP threading layer, mainly for Analysis phase, presented in 3.5.

4. cuDSS MGMN: multi-GPU, multi-node, functionally verified but not included in profiling analysis due to hardware limitations. NVIDIA's A10 GPU [15], which was used to test all methods, does not support Multiple Instance GPU (MIG) for partitioning one single physical GPU. The developed code remains validated for possible multi-GPU implementations in the future. The algorithmic steps are shown at 3.6.

5. cuSOLVER: CUDA-based, currently outdated, without a supported sparse LU routine for general unsymmetric matrices. The only available sparse solver of QR delivered worse timing results than the baseline CPU method and failed to scale to the largest IBM Benchmarks; thus it is excluded from the comparison. However, it can be studied at 3.2.

4.2 Methodology

Publicly available IBM Power Grid Benchmarks [4] were used as input circuit designs for all methods developed. The matrices that arise from these benchmarks are very sparse ($density \approx 10^{-2} - 10^{-4}$) and structurally different due to different circuit elements. The workstation where testing was performed, consisted of 16 CPU cores, $64\ GB$ RAM and NVIDIA A10 ($24\ GB$ GDDR6). All the methods were based in cuDSS library version v0.6.0, which was the latest release at the time of writing this thesis. For each method, the following metrics were measured:

- Time per phase: Two timers were developed for measuring elapsed time for Symbolical factorization, Numerical factorization and Solving phases individually. The metrics used for CPU baseline method and GPU methods, was wall-clock time and CUDA events correspondingly. In Numerical factorization and Solving phases, GPU methods provide acceleration in timing; a Speedup parameter $Speedup = T_{CPUbaseline}/T_{GPUbaseline}$ is provided and used as a metric to represent the large timing performance differences of GPU acceleration.

- Factor Structure: Calculated matrix dimensions of L and U along with the number of non-zero elements in the arising matrices from Symbolic and Numeric factorization phases, in order to measure the density of factors $p = nnz(L, U)/n^2$.

- Memory usage: Estimated Host RAM peak usage for CPU baselie and GPU methods, as well as Device VRAM peak for GPU methods, to report the impact of factorization phases in hardware resources. For the GPU methods, only cuDSS baseline is reported due to the fact that all GPU methods showed identical or within tiny noise peak memory usage. This arises from the fact that all the GPU methods produce the same factor density; which is the result similar reordering and pivoting techniques, leading to the same algrebraic (LU) structure, by applying the same default and optimized parameters during the factorization phases.

4.3 Observations

4.3.1 Timing: phase-by-phase behavior

1. Symbolical factorization (Analysis):

    - GPU baseline(single-threaded host) is slower than the CSparse-based CPU baseline throughout all the benchmarks. cuDSS Analysis phase (symbolic and reordering) is executed primarily on the host side, including GPU handshake and initialization overheads.
    - GPU MT (multi-threaded host) directly reduces Analysis time without changing factor structure. This implies in both cases of using 8 and 16 cores, which still remain slower than CPU baseline but the gap narrows substantially.
    - Interpretation: Symbolic is dominated by serial or lightly parallel host-side work. cuDSS's MT layer spreads that host work across $(8-16)$ CPU threads, shrinking the gap, while leaving the GPU-accelerated numeric phase unchanged (see the top chart of Figs.: 4.1, 4.2, 4.3 and 4.4). This is why all cuDSS variants have identical density and memory (as seen in Fig. 4.5, and Fig. 4.6): MT does not change permutation and fill ordering; only how fast the host performs the same analysis.

2. Numerical factorization:

    - This phase drives the overall acceleration. cuDSS reduces numeric factorization from tens to hundreds of seconds on CPU to ($\sim 0.1s$) on the A10, yielding three orders of magnitude speedups from ($6x\ to\ 1500x$)).
    - Multithreaded version shows nearly the same numeric time as the GPU baseline, because numeric factorization phase is GPU-bound; host threading does not materially affect it. The line series in the numeric charts is almost flat across cuDSS modes (see the middle chart of Figs.: 4.1, 4.2, 4.3 and 4.4).
    - Notable observation: Although ibmpg3 $(nodes = 1,401,572)$ and ibmpg4 $(nodes = 1,560,645)$ benchmarks are smaller than ibmpg3 $(nodes = 1649002)$ both in size and corresponding non zeros, CPU's baseline method fill-in for ibmpg3 and ibmpg4 is worse, inflating its factorization time. On the other

hand cuDSS's LU remains compact on mentioned circuits, so the Speedup becomes larger on the smaller benchmarks (see the middle chart of Figs.: 4.2, 4.3 and 4.4). This is corroborated by the factor structure diagram (as seen in Fig. 4.5), where the number of nonzero elements for L and U for CPU_baseline(ibmpg3, ibmpg4) is much larger than cuDSS(ibmpg3, ibmpg4).

3. Solving:

   - Solving is a small fraction of total runtime on both the CPU and GPU, but cuDSS methods still show a clear advantage (see the bottom chart of Figs.: 4.1, 4.2, 4.3 and 4.4).
   - Multithreaded version MT has negligible effect. All cuDSS variants are plotted as similar, short bars with the same line value.

4.3.2 Factor structure and density

- GPU factors are consistently smaller and sparser (as seen in Fig. 4.5). This is a consequence of fewer floating point operations per second (FLOPS) and lower memory traffic in Numerical factorization which directly explains the obvious Speedup in Numerical charts. All cuDSS methods coincide here because they share the same reordering/pivoting and therefore identical non-zeros and density. Multithreading alters only the time to prepare the same structure.

4.3.3 Memory usage: host vs device

- CPU baseline host memory scales with factor size and reaches multi-GB usage (as seen in Fig. 4.6).
- VRAM memory of cuDSS device stays within ($\sim 0.8 \ \ to \ \ \sim 1.3$) GB for these cases; host-side cuDSS footprint is modest ($\sim 0.07 \ \ to \ \ \sim 0.2$) GB.
- Interpretation: cuDSS shifts the heavy memory burden to device VRAM (where bandwidth is much higher), while reducing total factor storage via better permutations and GPU-oriented dataflows. For very large designs, this can free several GB of host RAM.

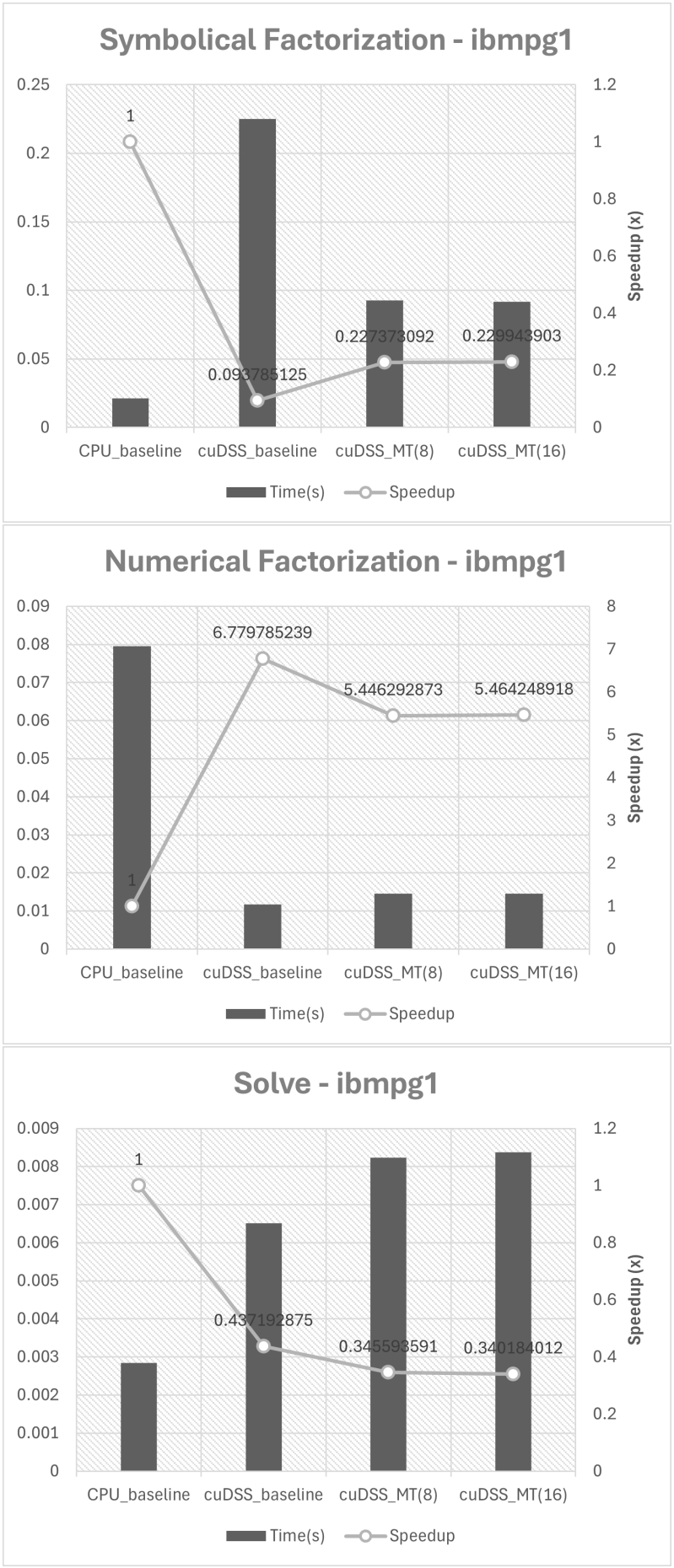

Figure 4.1: Timing per phase for ibmpg1. Top: Symbolic Factorization linear time). Middle: Numerical Factorization (linear time). Bottom: Solve (linear time).

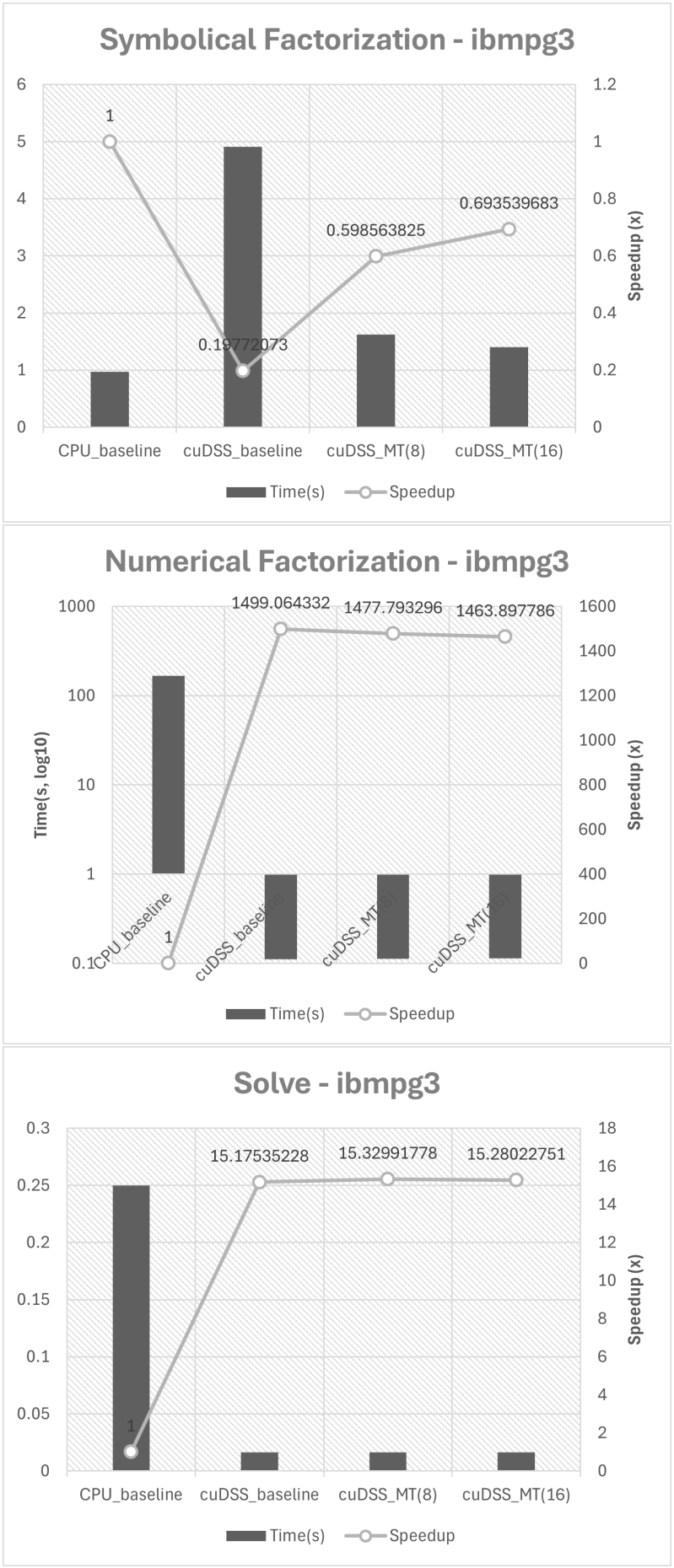

Figure 4.2: Timing per phase for ibmpg3. Top: Symbolic Factorization (linear time). Middle: Numerical Factorization (log time). Bottom: Solve (linear time).

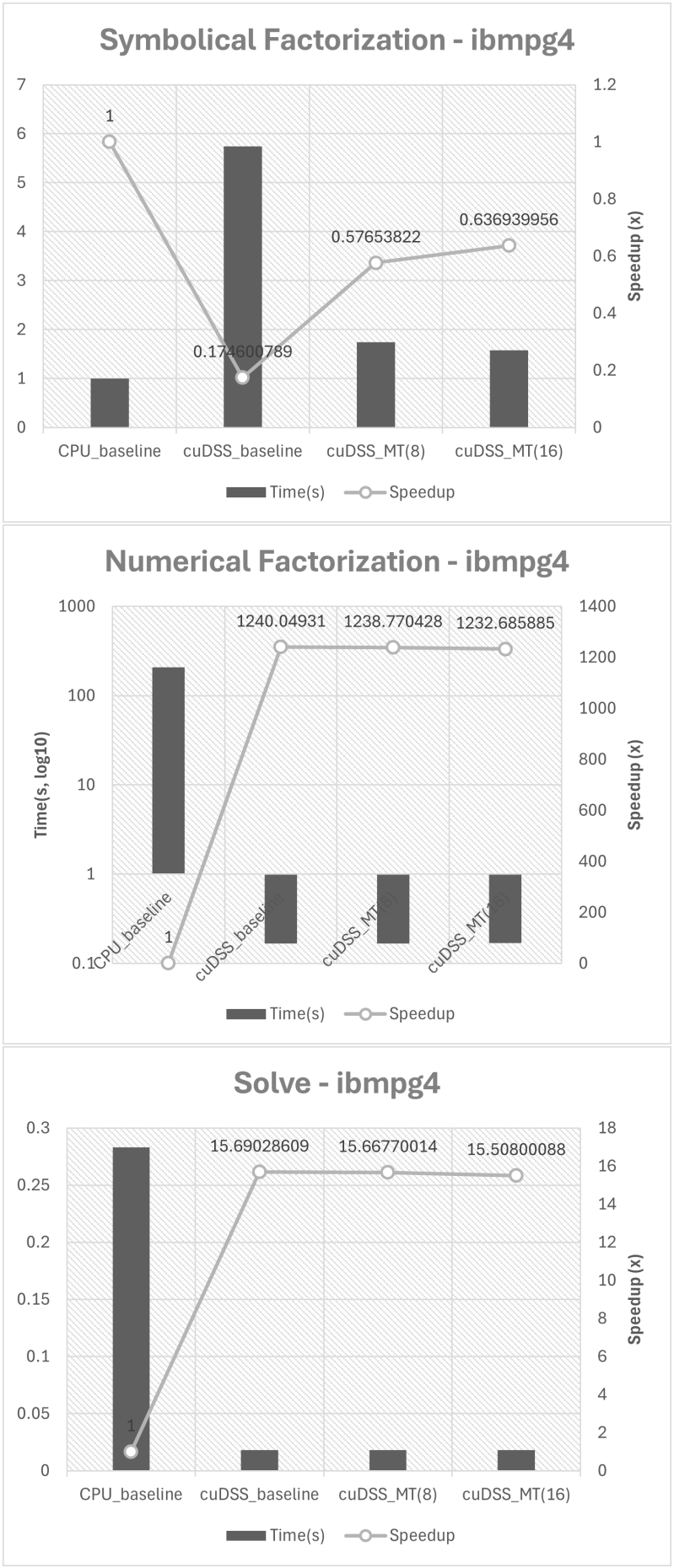

Figure 4.3: Timing per phase for ibmpg4. Top: Symbolic Factorization (linear time). Middle: Numerical Factorization (log time). Bottom: Solve (linear time).

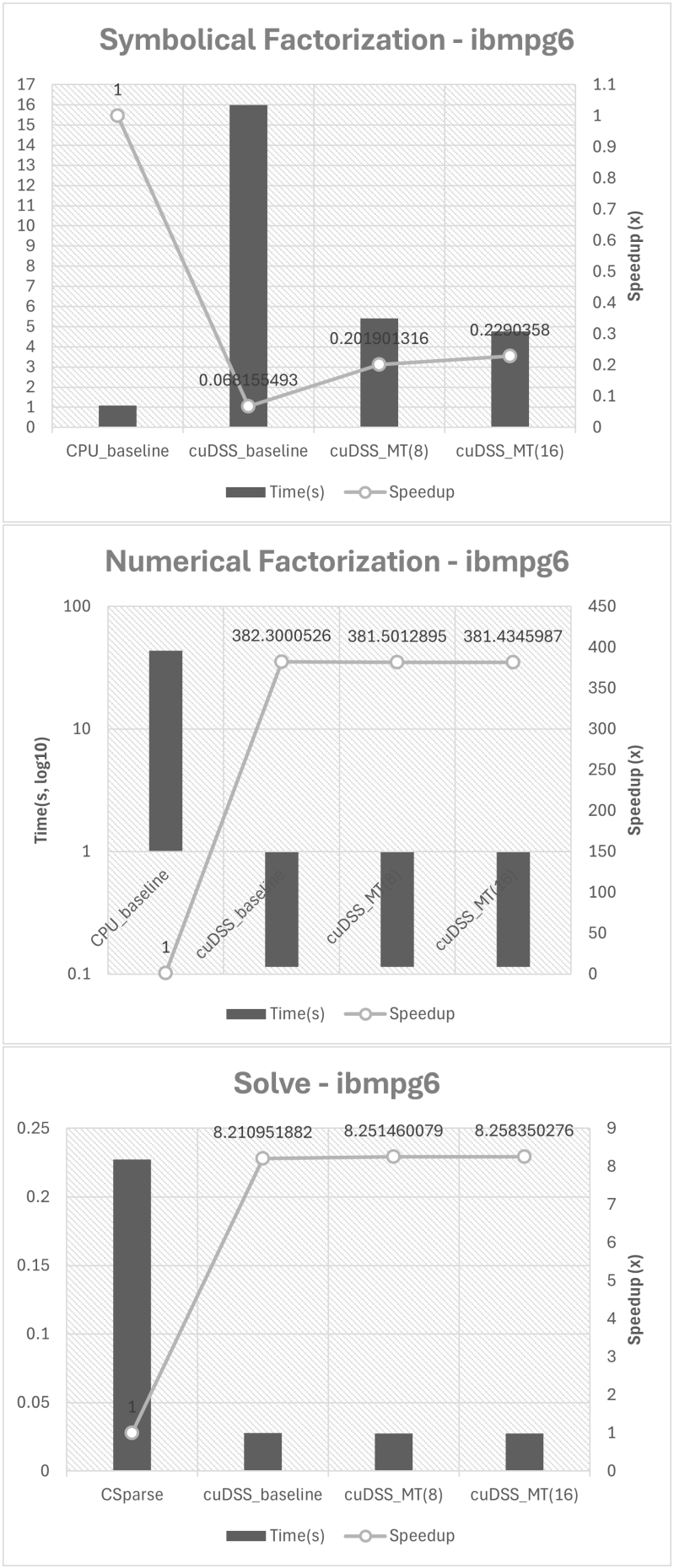

Figure 4.4: Timing per phase for ibmpg6. Top: Symbolic Factorization (linear time). Middle: Numerical Factorization (log time). Bottom: Solve (linear time).

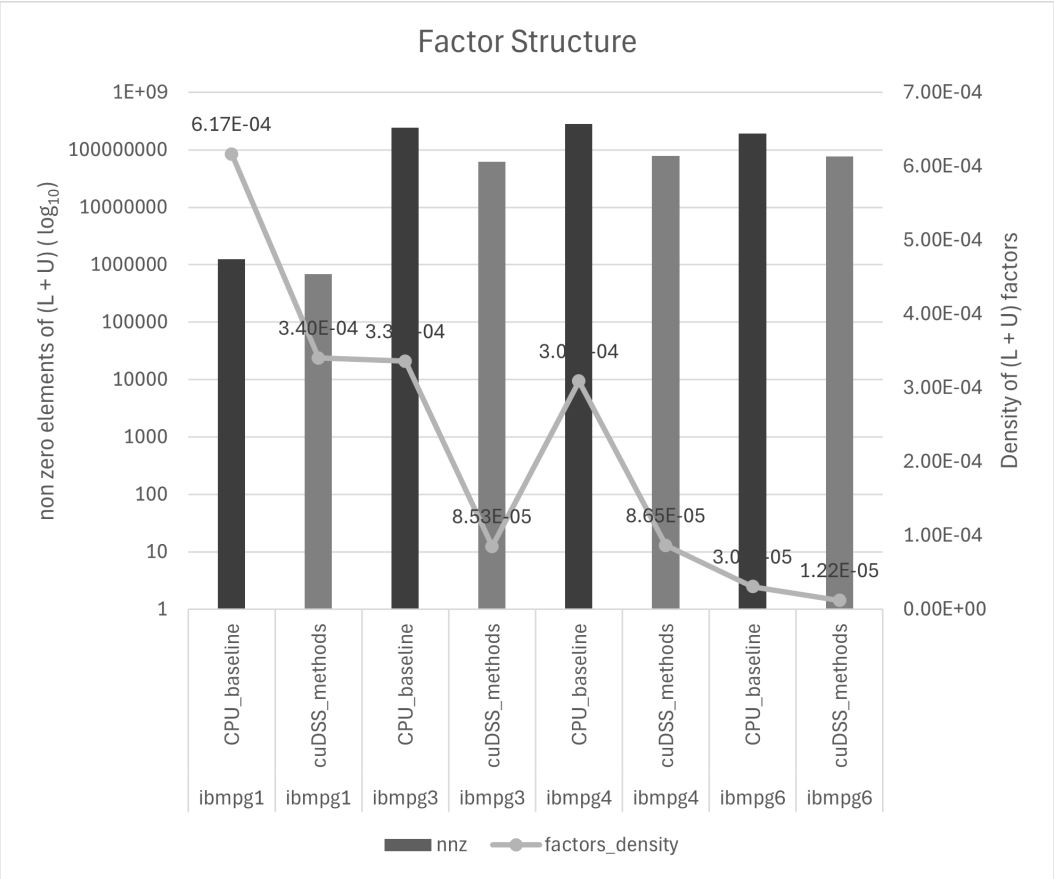

Figure 4.5: Structure of L and U Factors across benchmarks for CPU and GPU methods

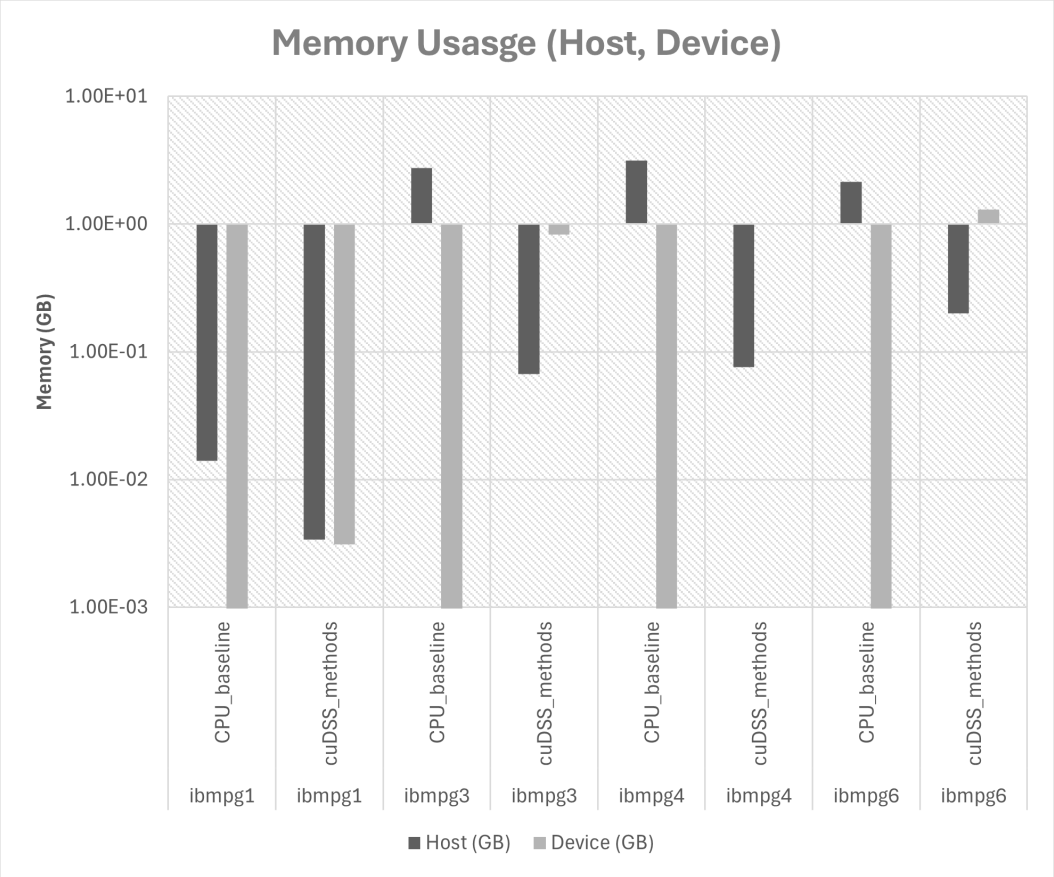

Figure 4.6: Peak memory by method and benchmark. Host RAM (CPU arrays and cuDSS host) and GPU VRAM (cuDSS device) reported separately

4.3.4 Across-benchmark trends

- The dominant cost in circuit simulation of sparse linear solves is the numeric factorization; cuDSS reduces it by two to three orders of magnitude.
- Small matrices (ibmpg1): cuDSS methods accelerate Numeric, but Symbolic and Solving may not amortize GPU overheads; Host MulTithreading improves Symbolic without changing factor density or memory usage, but total runtime is dominated by (small) absolute costs.
- Stability and robustness: cuDSS LU (with default/optimized reordering and pivoting) was robust across all tested IBM circuits.
- Smaller factor size (non-zeros and density) under cuDSS translates to less work and less memory—a twofold gain.
- Overall, for real Power Grid circuits, GPU-centric sparse direct solving (cuDSS) is a clear, practical acceleration path for circuit simulation tools.

4.4 Conclusion

This thesis set out to answer a simple question with practical weight for circuit simulation: can GPU parallelization, via modern GPU–based sparse direct solvers, materially accelerate the solution of the large, sparse linear systems that arise in Modified Nodal Analysis (MNA)? The evidence gathered across IBM power–grid benchmarks, together with an implementation study of cuDSS and reference CPU baseline exploiting CSparse library, shows that the answer is yes, with important caveats and a clear picture of where the gains come from.

The profiling analysis performed in this thesis, consistently shows that GPU-accelerated sparse direct solving methods with cuDSS delivers order of magnitude reductions in numeric factorization time, without inflating memory usage. The Analysis phase benefits from multithreading but does not change the algebraic work (hence identical factor non-zeros/density and memory across cuDSS methods). The dominant driver of time and memory is the size and density of the factors; cuDSS's reorder/pivot strategy typically yields smaller LU than the CPU baseline, further amplifying GPU advantages.

Overall, the results validate the central hypothesis of the thesis: for very large, sparse MNA systems, GPUs and GPU-based libraries provide compelling acceleration for sparse

LU factorization and solving phases, and form a practical path forward for high-performance circuit simulations.

Bibliography

[1] MNA. https://en.wikipedia.org/wiki/Modified_nodal_analysis.

[2] A. George and J. W. H. Liu. The evolution of the minimum degree algorithm. *SIAM Review*, 31(1):1–19, 1989.

[3] SPICE. https://www.cadence.com/en_US/home/explore/spice-simulation.html.

[4] IBM Power Grid Benchmarks. https://web.ece.ucsb.edu/~lip/PGBenchmarks/ibmpgbench.html.

[5] F. N. Najm. *Circuit Simulation*. Wiley, Hoboken, New Jersey, 2010.

[6] Conjugate gradient method. https://en.wikipedia.org/wiki/Conjugate_gradient_method.

[7] Biconjugate gradient method. https://en.wikipedia.org/wiki/Biconjugate_gradient_method.

[8] LU. https://en.wikipedia.org/wiki/LU_decomposition.

[9] Cholesky factorization. https://en.wikipedia.org/wiki/Cholesky_decomposition.

[10] Gnu Scientific Library (GNU) - Reference Manual. https://www.gnu.org/software/gsl/doc/latex/gsl-ref.pdf.

[11] SuiteSparse library. https://github.com/DrTimothyAldenDavis/SuiteSparse/tree/dev/CSparse.

[12] CUDA. https://developer.nvidia.com/cuda-toolkit.

[13] NVIDIA cuDSS library. https://docs.nvidia.com/cuda/cudss/index.html.

[14] cusolver. https://docs.nvidia.com/cuda/archive/12.3.0/cusolver/contents.html.

[15] NVIDIA A10 GPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a10/pdf/A10-Product-Brief.pdf.

[16] P. R. Amestoy, T. A. Davis, and I. S. Duff. Algorithm 837: Amd, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Software*, 30(3):381–388, 2004.

[17] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. SCI. COMPUT.*, 20(1):359–392, 1998.

[18] OpenMP. https://www.openmp.org/.

[19] OpenMPI. https://www.open-mpi.org/.