GPU and Krylov Based Parallel Model Order Reduction Algorithms for MIMO Circuit Reductions

by

Rayeed Rahman Ananda

A thesis submitted to the Office of Graduate Studies in partial fulfillment of the requirements for the degree of

Master of Applied Science
in
Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

© 2026
Rayeed Rahman Ananda

Acknowledgements

I would like to express my sincere gratitude to my thesis advisor, Professor Ram Achar for introducing me to the world of research and high-performance computing. His guidance, support and encouragement has allowed me to explore my ideas, dig deeper, and helped me build a foundational base for picking up any research topic. I would also like to thank Professor Wai Kong of UTAR University for his help in providing learning resources pertaining to programming Graphics Processing Units and shared memory architectures. I dedicate this work to my late grandfather, Dr. Nurul Alam Khan. His wisdom has been the instrumental force of guidance for me in most pursuits of my life, both academic and personal. Without him, I would not be where I am today in life. Finally, my heart-felt gratitude to the two greatest women in my life - my mother Urmi Irani Khan and my grandmother Rijia Alam for their boundless love and support all throughout.

Abstract

Analysis of modern high-speed, high-density designs with multi physics-based formulations for signal, power and thermal integrity analysis is becoming increasingly challenging due to the need for mixed frequency and time analysis. The associated timing, memory and costs of solving large sets of circuit equations become prohibitively expensive. Simulation of such systems help designers to build, analyze and characterize circuits prior to their manufacturing. For circuit designers to effectively streamline the design process, innovate products and reduce the time to market, models need to accurately capture real-world behaviors of the components in the design. However, working with full-scale models can lead to circuits with millions of internal nodes and hundreds of external ports incurring prohibitively excessive computational costs and expensive memory requirements. To circumvent such problems, Model Order Reduction (MOR) schemes utilizing Krylov Subspace based projections such as the Passive Reduced Interconnect Modeling Algorithm (PRIMA) have been widely used. PRIMA facilitates passive reduction of large linear sets of equations, generating compact, passive reduced models that accurately represent the response of the original system. Further work in this field, such as the Structure-Preserving Reduced-order Interconnect Macromodeling algorithm (SPRIM) have also been utilized to retain the block structures of the original system matrices in their reduced forms. In this thesis, we advance the PRIMA and SPRIM algorithms to the emerging computing platforms of Graphics Processing Units (GPU) with Tensor Cores (TC), yielding significant speed-up compared to using the traditional multi-core CPU (Central Processing Unit) platforms.

Table of contents

List of Figures viii

List of tables x

Acronyms / Abbreviations xi

1 Introduction 1
- 1.1 Background and Motivation . . . 1
- 1.2 Model Order Reduction . . . 2
- 1.3 Explicit Moment Matching via Padé, AWE . . . 2
- 1.4 Implicit Moment Matching Via Block Arnoldi . . . 3
- 1.5 PRIMA and SPRIM as Passive Model Order Reduction Techniques . . . 3
- 1.6 Parallelized Arnoldi Implementations . . . 4
- 1.7 Contributions . . . 5
- 1.8 Organization of the Thesis . . . 6

2 The Modern GPU Architecture and Programming Model 7
- 2.1 Introduction to GPU . . . 7
- 2.2 Terminology . . . 8
- 2.3 Memory Hierarchy . . . 10
  - 2.3.1 Global Memory . . . 11
  - 2.3.2 Local Memory . . . 11
  - 2.3.3 Constant and Texture Caches . . . 11
  - 2.3.4 Shared Memory . . . 12
  - 2.3.5 Registers . . . 12
- 2.4 Compute Framework and Scalability . . . 12
  - 2.4.1 Compute Framework . . . 12
  - 2.4.2 The Programming Process Flow . . . 14
  - 2.4.3 Scalability . . . 15

2.5 Summary 15

3 Projection-Based Model Order Reduction: PRIMA and SPRIM 16
3.1 MNA Descriptor Formulation for RLC Interconnects 16
3.2 Arnoldi/Block-Arnoldi Iterations for Implicit Moment Matching 19
3.3 PRIMA: Passive Reduced Interconnect Modeling via Projectors 20
3.4 SPRIM: Structure-Preserving Reduced Interconnect Modelling 24
3.4.1 Second–order form of the SPRIM reduced model 27
3.5 Moment Matching Considerations in PRIMA and SPRIM 28
3.5.1 PRIMA: moment matching via split congruence transformation 28
3.5.2 SPRIM: moment matching via split diagonal congruence transformation 30
3.6 Passivity Preservation with PRIMA and SPRIM Algorithms 31
3.7 Accuracy and Computational Considerations 32
3.8 Summary 33

4 Proposed Development of GPU PRIMA 34
4.1 Preliminaries 34
4.2 Proposed Implementation Pipeline Overview 35
4.3 Step 1: CPU based Matrix Pre-processing and Computation of **A** and **R** Matrices 37
4.4 Step 2: GPU-based Block Arnoldi and Projection Basis Construction . 37
4.5 Step 3: Projection to The Reduced Model (GPU) 39
4.6 Step 4: Computation of Reduced System Matrix 39
4.7 Step 5: Eigen-decomposition (GPU) 40
4.8 Step 6: Computing the Residue Factors (GPU) 40
4.9 Step 7: Formulation of Port Transfer function in GPU 40
4.10 Discussion 42
4.10.1 Data layout, precision, and use of libraries 42
4.10.2 Workload and performance overview 42
4.11 Summary 43

5 Numerical Results for the Proposed GPU-PRIMA 44
5.1 Experimental Setup 44
5.2 Example 1: Multi-port Circuit with $N_p = 40$ 45
5.2.1 Discussion 50
5.3 Example 2: Multi-port RLC Circuit with $N_p = 128$ 50

5.3.1 Discussion . . . 55
5.4 Summary . . . 55

6 Proposed Development of GPU SPRIM 56
6.1 Preliminaries . . . 56
6.2 Proposed implementation pipeline overview . . . 57
6.3 Step 1: CPU based MNA Matrix Pre-processing . . . 59
6.4 Step 2: GPU-based Block Arnoldi . . . 59
6.5 Steps 3-5: SPRIM Projector Partitioning and Block-diagonal Assembly (GPU) . . . 59
6.6 Step 6: Projection to the SPRIM Reduced Model (concurrent streams in GPU) . . . 61
6.7 Step 7: Frequency-Domain Response Computation (GPU) . . . 61
6.8 Discussion . . . 63
6.8.1 Data layout, precision, and libraries . . . 63
6.8.2 Workload and Performance Overview . . . 64
6.9 Summary . . . 65

7 Numerical Results for the Proposed GPU-SPRIM 66
7.1 Example 1: Multi-port RLC Circuit with $Np = 40$ . . . 66
7.1.1 Discussion . . . 70
7.2 Example 2: Multi-port RLC Circuit with $Np = 128$ . . . 71
7.2.1 Frequency Domain Response . . . 71
7.2.2 Discussion . . . 74
7.3 Accuracy Comparison Between Original Model Vs. GPU-PRIMA and GPU-SPRIM . . . 75
7.3.1 Discussion . . . 77
7.4 Transient Analysis: Multi-port RLC Circuit with $Np = 40$ . . . 78
7.5 Summary . . . 81

8 Conclusion 83
8.1 Future Work and Considerations . . . 84

Bibliography 85

A An Illustrative Circuit Example for PRIMA and SPRIM Formulation and Reduction 89
A.1 MNA Stamps . . . 90

A.1.1 Stamping the MNA Matrices 90
A.1.2 MNA Matrices at the Expansion Point 92
A.2 PRIMA Projector via Block Arnoldi 92
A.3 SPRIM Projector Assembly and Reduced Order Matrices 94

List of Figures

2.1 GPU vs CPU Architectures 8
2.2 The Hybrid Programming CUDA Model 9
2.3 The Memory Hierarchy model within a GPU 10
2.4 SIMD/SIMT vs MIMD Instruction Architectures 13
2.5 Illustration of multi-GPU architecture 15

3.1 An illustrative multi-port RLC excited by current sources, represented as a black-box transfer function $H(s)$. 17
3.2 Inductor Stamp 18
3.3 Partitioning the Krylov projection basis $\mathbf{X}$ according to inner blocks of $\mathbf{G}$. 24
3.4 Matrix sparsity patterns 25

4.1 Algorithmic flowchart for GPU–PRIMA. 36
4.2 An example of a 5×5 upper-triangular transfer function matrix. 41
4.3 Mapping the transfer function matrix to linearized thread indices. . . . 41

5.1 Illustration of the RLC mesh circuit configuration for an example case of 10 ports 45
5.2 Comparison of $|\mathbf{H}_{1,1}(j\omega)|$ from original system and GPU-PRIMA reduced models (40-port network) 46
5.3 Comparison of $|\mathbf{H}_{1,10}(j\omega)|$ from original system and GPU-PRIMA reduced models (40-port network) 47
5.4 Comparison of $|\mathbf{H}_{1,38}(j\omega)|$ from original system and GPU-PRIMA reduced models (40-port network) 47
5.5 Comparison of $|\mathbf{H}_{11,50}(j\omega)|$ from original system and GPU-PRIMA reduced models (128-port network) 51
5.6 Comparison of $|\mathbf{H}_{90,92}(j\omega)|$ from original system and GPU-PRIMA reduced models (128-port network) 52

5.7 Comparison of $|\mathbf{H}_{100,101}(j\omega)|$ from original system and GPU-PRIMA reduced models (128-port network) 52

6.1 Algorithmic flowchart for the proposed GPU–SPRIM 58
6.2 **H**-matrix response calculation with overlapping compute and async D2H copies 63

7.1 Comparison of $|\mathbf{H}_{1,1}(j\omega)|$ from original system and GPU-SPRIM reduced models (40-port network) 67
7.2 Comparison of $|\mathbf{H}_{1,10}(j\omega)|$ from original system and GPU-SPRIM reduced models (40-port network) 67
7.3 Comparison of $|\mathbf{H}_{1,38}(j\omega)|$ from original system and GPU-SPRIM reduced models (40-port network) 68
7.4 Comparison of $|\mathbf{H}_{11,50}(j\omega)|$ from the original system and the GPU-SPRIM reduced model (128-port network) 71
7.5 Comparison of $|\mathbf{H}_{90,92}(j\omega)|$ from the original system and the GPU-SPRIM reduced model (128-port network) 72
7.6 Comparison of $|\mathbf{H}_{100,101}(j\omega)|$ from the original system and the GPU-SPRIM reduced model (128-port network) 72
7.7 Comparison of $|\mathbf{H}_{11,50}(j\omega)|$ from the original system, GPU-PRIMA and GPU-SPRIM reduced models (128-port network) 76
7.8 Comparison of $|\mathbf{H}_{90,92}(j\omega)|$ from the original system, GPU-PRIMA and GPU-SPRIM reduced models (128-port network) 76
7.9 Comparison of $|\mathbf{H}_{100,101}(j\omega)|$ from the original system, GPU-PRIMA and GPU-SPRIM reduced models (128-port network) 77
7.10 Input Source Signal at Port 1 79
7.11 Comparison of $\mathbf{V}_4(t)$ voltage waveform from the original system, GPU-PRIMA and GPU-SPRIM reduced models (40-port network) 79
7.12 Comparison of $\mathbf{V}_{10}(t)$ voltage waveform from the original system, GPU-PRIMA and GPU-SPRIM reduced models (40-port network) 80
7.13 Comparison of $\mathbf{V}_{22}(t)$ voltage waveform from the original system, GPU-PRIMA and GPU-SPRIM reduced models (40-port network) 80

A.1 An illustrative RLC circuit 89
A.2 Element MNA stamps used to assemble the matrices **G**, **C** and **B**. . . . 90

List of tables

5.1 Comparison of Computational Performance Times for PRIMA Implementations (Example-1, 40-Port Circuit) 49
5.2 Comparison of Computational Performance Times for PRIMA Implementations (Example-2, 128-Port Circuit) 54

7.1 Comparison of Computational Performance Times for SPRIM 2^{nd} Order Implementations (Example-1, 40-Port Circuit) 69
7.2 Comparison of Computational Performance Times for SPRIM 1^{st} Order Implementations (Example-1, 40-Port Circuit) 70
7.3 Comparison of Computational Performance Times for SPRIM 2^{nd} Order Implementations (Example-2, 128-Port Circuit) 73
7.4 Comparison of Computational Performance Times for SPRIM 1^{st} Order Implementations (Example-2, 128-Port Circuit) 74
7.5 Average Frobenius norm of the magnitude error over the frequency band 0–1 GHz for the proposed implementations GPU-PRIMA and GPU-SPRIM (128-port network) 77
7.6 Average Frobenius norm of the voltage output errors over the time period 0–50 ns at 1000 points for GPU-PRIMA and GPU-SPRIM (40-port network) 81
7.7 Computation Time for Voltages at Ports 81

Acronyms / Abbreviations

ALU Arithmetic Logic Unit

Arnoldi (Block) Arnoldi Iteration

AVX2 Advanced Vector Extensions 2

AWE Asymptotic Waveform Evaluation

BLAS Basic Linear Algebra Subprograms

CAD Computer-Aided Design

CFH Complex Frequency Hopping

CHOLMOD Sparse Cholesky Factorization Library

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

D2H Device-to-Host (memory transfer)

DAE Differential–Algebraic Equation

DRAC Digital Research Alliance of Canada

EVD Eigenvalue Decomposition

FLOP Floating-Point Operation

FPU Floating-Point Unit

GEMM General Matrix–Matrix Multiply

GPU Graphics Processing Unit

H2D Host-to-Device (memory transfer)

KLU Sparse LU Factorization for Circuit Matrices

Krylov Krylov Subspace

LAPACK Linear Algebra PACKage

LU Lower–Upper Factorization

MAGMA Matrix Algebra on GPU and Multicore Architectures

MIMD Multiple Instruction, Multiple Data

MIMO Multiple-Input Multiple-Output

MMT Moment Matching Techniques

MNA Modified Nodal Analysis

MOR Model Order Reduction

PCIe Peripheral Component Interconnect Express

PDN Power Distribution Network

PRIMA Passive Reduced-order Interconnect Macromodeling Algorithm

PR Positive Real

PVL Padé Via Lanczos

QR Orthogonal–Triangular Factorization

RLC Resistor–Inductor–Capacitor

SIMT Single Instruction, Multiple Threads

SM Streaming Multiprocessor

SPICE Simulation Program with Integrated Circuit Emphasis

SPRIM Structure-Preserving Reduced-order Interconnect Macromodeling

SuiteSparse Sparse Matrix Algorithms Suite

TC Tensor Cores

TSQR Tall–Skinny QR

VLSI Very-Large-Scale Integration

Chapter 1

Introduction

1.1 Background and Motivation

With ever-shrinking transistor sizes and advancements in semiconductor fabrication, it has become possible to pack billions of transistors inside modern integrated circuit chips. Modern circuits are required to operate at high frequencies (in the gigahertz range (GHz) while consuming low power during operation. Indeed, today's VLSI chips can be regarded as entire cities of transistors at the nano-meter scales. However, with increasing chip densities and integration innovations, come a plethora of new challenges in the form of signal and power integrity issues. Signal integrity issues such as signal delay, crosstalk, ringing and skin effect come into play due to higher frequencies and fast switching, resulting in degrading signal quality. Power integrity issues such as ground bounce can significantly degrade the ability to supply clean, stable voltages through the power-distribution network (PDN) to all active devices. These issues can arise at all hierarchies of the design process and require careful analysis and evaluation before the products (chips, packages and boards) can be fabricated. This is because the semiconductor industry is associated with extremely high costs in product design life cycles (especially during fabrication). This leads to the need for CAD (computer aided design) tools for simulation and characterization, to ensure proper and reliable operation of the systems they design.

Traditionally, circuit simulation is accomplished by formulating the modified nodal analysis based equations using the component models [1, 2]. However, due to the signal and power integrity issues [3], MNA systems of equations tend to be large [2, 4] and often challenged by mixed frequency-time as well as multi physics based analysis requiring large memory and incurring prohibitively excessive computational costs. To address these challenges, model order reduction techniques were introduced [5–10] .

1.2 Model Order Reduction

The underlying philosophy of MOR (model order reduction) is to create a reduced-order model that can capture the response of an original circuit with reasonable accuracy and with limited computational cost. Subsequently, reduced order models can be re-used again for repetitive tasks of simulation and optimization instead of the computationally expensive original large circuits, thus resulting in significant computational savings. Main considerations during model-order reduction are accuracy, CPU efficiency and maintaining the passivity of the reduced-order model. Early MOR techniques such as Elmore Delay explicitly matches the first Taylor coefficient of the transfer function of RC trees for fast timing analysis [5]. However, due to the low order approximations, they suffered from low numerical accuracies. More rigorous MOR techniques were based on Padé approximation and Moment Matching Techniques (MMT). Explicit MMT methods like Asymptotic Waveform Evaluation (AWE) [11] based on expansion at the origin (DC), suffer from ill-conditioning when attempting higher-order approximations. This was addressed via Complex Frequency Hopping (CFH) [7] which uses multiple expansions to achieve higher bandwidth of accuracy. These issues are also addressed through implicit methods based on Krylov subspace methods such as Padé via Lanczos (PVL) [8] and Arnoldi algorithm [12]. However, these methods do not ensure the passivity of the reduced-order model, without which stability during transient simulation of the macromodel with the rest of the circuitry cannot be guaranteed.

1.3 Explicit Moment Matching via Padé, AWE

Classical explicit moment–matching based approaches build a Padé approximation of the multi-port transfer function around a specified frequency expansion point known as the Asymptotic Waveform Evaluation (AWE) [11]. Here the moments of $\mathbf{H}(s)$ around s_0 are computed and matched to produce a low–order rational approximation to model the original RLC networks. Early applications using AWE included approximations to RC trees and later advanced to handle RLC networks as well as lossy multiconductor transmission lines [11].

However, single-point Padé/AWE had well–documented shortcomings when used over a wider band of frequencies as the Toeplitz systems used in coefficient extraction became too ill-conditioned (typically only 6–8 reliable poles could be extracted). Many spurious unstable poles would appear with increase in the order of approximation and accuracy

degraded quickly as one moved away from the frequency point of expansion. These challenges motivated Padé based approximations using multi-point strategies [4] such as Complex Frequency Hopping.
Complex Frequency Hopping (CFH) addressed this accuracy degradation over a large bandwidth by matching moments at multiple frequency points near the imaginary axis, and unifying them into a common pole set. It employed a binary search based approach, while matching the responses at the midpoint between two adjacent frequency points. This yielded more accurate frequency responses with a minimized number of expansions [4, 7].
In a multi-port setting, Block CFH [9] approach can assemble accurate responses for each transfer function entry, then form residues from the union of driving-point poles, enabling macromodel synthesis for SPICE-like simulators.

1.4 Implicit Moment Matching Via Block Arnoldi

Accuracy issues and ill-conditioning for higher-order approximations were addressed by implicit Krylov approaches (e.g., PVL/Arnoldi) that avoid explicit moment computation; instead, they project the solution from the MNA onto a block Krylov subspace [6, 8, 13–17]. This is generally done through the use of subsequent orthonormalization, which improves the numerical robustness compared to AWE when considering higher-order approximations. The reduced transfer function matches the first q block moments about an expansion point s_0 without needing to assemble the high-order, ill-conditioned Toeplitz systems. In practice, a block Arnoldi basis and its block upper–Hessenberg form provide the reduced realization that can be used for frequency and time-domain analysis [6, 8].
It is to be noted that AWE, CFH, and Arnoldi based approaches may produce models that are not passive, and can trigger large erroneous oscillations in time domain simulations when the macromodel is re-embedded with other linear or nonlinear components during simulations [4, 7].

1.5 PRIMA and SPRIM as Passive Model Order Reduction Techniques

PRIMA (Passive Reduced-order Interconnect Macro-modeling Algorithm) [14] employs the block-Arnoldi basis with a split *congruence* projection of the MNA matrices

(these matrices are formulated such that they are semi-positive definite). Since the transformation is real and congruent, the reduced model retains the positive semi-definiteness and thereby satisfies standard positive-real passivity conditions for the reduced multiport transfer function [14, 15]. SPRIM extends PRIMA by enforcing the original MNA block structure (voltage/current/inductor) through a re-assembled block-diagonal projector. The SPRIM based projection also uses split congruent transformation, preserving passivity as well as positive semi-definiteness and the block pattern of the original MNA matrices, which can help produce reduced models that can be mapped more directly to physically realizable reduced circuits [16, 18].

1.6 Parallelized Arnoldi Implementations

To speed up the Arnoldi process, several parallelization algorithms and communication reducing strategies were considered [19–23]. On distributed-memory systems, Arnoldi process can incur many global reductions (inner products and norms within the inner loops) per iteration, which can dominate the run time. Hernandez–Roman–Tomás propose rearrangements of Arnoldi that keep the classical Gram–Schmidt process with iterative selective re-orthogonalization (for robustness) using a custom *norm estimation* [20]. This was accomplished by applying the pythagorean theorem between the projected vector, the orthogonalized vector and the projection of the orthogonalized vector onto the Krylov vectors. The implementation averages the residues/Arnoldi-coefficients and re-orthogonalizes the vectors twice. This approach also has a further implementation that *group* the global communications by reorganizing the stages that require dot products and normalization to effectively generate next stage vectors before full orthonormalization so as to reduce the number of synchronizations. Their scalability analyses show good parallel improvements with controlled loss of numerical stability [20].

Communication Avoiding (CA) Krylov ideas reduce synchronizations further by opting for approaches based on Classical Block Gram–Schmidt combined with binary tree based Tall-Skinny QR (TSQR) replacing column-by-column Modified Gram–Schmidt with a tradeoff in numerical robustness. CA-Arnoldi steps prefer orthogonalization to be cast as dense, blocked matrix–vector/matrix–matrix reductions, cutting global synchronizations while retaining stability. These techniques can be extended to selective or delayed reorthogonalization and to Lanczos/CG with benefits in reducing synchronization counts in the algorithms [21, 24].

This thesis utilizes a parallelized Block Arnoldi implementation based on Block MGS to batch together matrix-matrix multiplies and reduce synchronization points as part of PRIMA and SPRIM methods.

1.7 Contributions

In this thesis, the following contributions are made at the algorithmic and implementation levels for Krylov-based passive model order reduction of RLC circuits on GPU (Graphical Processing Unit) [25] platforms with the emerging tensor cores.

- **Formulations of PRIMA and SPRIM Algorithms Suitable for GPU Platforms:** In this contribution, PRIMA and SPRIM formulations that are suitable for modern GPU with Tensor Core Platforms were presented to (i) construct Krylov subspaces with Block Arnoldi, (ii) reduce the MNA pair via *split congruence transformation*, and (iii) evaluate frequency responses either through a pole–residue approach (PRIMA) or using solutions of direct reduced state space systems (SPRIM).
- **Communication-lean Block Arnoldi:** The Arnoldi pipeline is organized using BLAS–3 operations and tall-skinny QR (TSQR) to minimize global synchronizations and improve numerical robustness. Re-orthogonalization is handled by GPU TSQR/MGS (Modified Gram-schmidt) implementations from MAGMA (Matrix Algebra for GPU and Multi-core Architectures). Corresponding results are published in [26].
- **Structure-preserving projector assembly:** A block-diagonal projector kernel is proposed by *row-partitioning* the Arnoldi basis and re-orthonormalizing each block. This preserves the block pattern/reciprocity of the MNA and enables direct frequency-domain solutions using the reduced descriptor form. For this, custom kernels are proposed for basis packing/splitting and block-diagonal assembly of the projector matrix.
- **CPU–GPU pipeline:** A CPU-GPU pipeline with minimal transfers in-between is developed, wherein sparse preprocessing (using CHOLMOD/KLU [27]) on the CPU is performed to compute the initial system solution and all iterative and dense stages (Block Arnoldi, projection, reduced LU/eigensolves) are proposed to remain on the GPU to limit PCIe (Peripheral Component Interconnect Express) traffic.
- **Stream-level concurrency and efficient data packing:** Separate CUDA (Compute Unified Device Architecture) streams are developed to overlap computa-

tions at the following task levels: (i) computing reductions using split congruence transformations, (ii) per-block MGS operations in SPRIM, and (iii) reduced system solutions and asynchronous D2H (device-to-host) copy during frequency sweeps. Customized kernels for real→complex casting, and packed upper-triangle assembly of the port response $\mathbf{H}(j\omega)$ are also proposed.

- **Overlapped computation and batched evaluation:** Reduced system solutions during frequency sweep at thousands of points using parallel LU solutions (SPRIM) with separate CUDA streams and events to overlap computation and I/O data writing are proposed.

1.8 Organization of the Thesis

This thesis is organized as follows. Chapter 1 provides the motivations for the need for efficient and passive model order reduction (MOR) for large interconnect networks and summarizes the state-of-the-art work in explicit and implicit moment matching approaches. Chapter 2 reviews GPU architecture and the CUDA programming model, including memory hierarchy, execution model, and scalability considerations relevant to high-throughput linear algebra type computations. Chapter 3 outlines the formulations of the MNA descriptor system for RLC interconnects, explicit moment matching (Padé/AWE/CFH) and implicit Krylov approaches, and the PRIMA and SPRIM projections. Chapter 4 presents the development of the proposed GPU-PRIMA: block-Arnoldi basis construction, congruent projection, reduced system eigen-decomposition, and a batched pole–residue frequency sweep. Chapter 5 presents the validating results for the proposed GPU-PRIMA presented in Chapter 4. Chapter 6 presents the development of the proposed GPU-SPRIM algorithm: projector partitioning and block-diagonal assembly, concurrent reductions, and per-frequency reduced solutions. Corresponding validating results for the proposed GPU-SPRIM are presented in Chapter 7, along with a performance comparison between the original system, PRIMA and the SPRIM algorithms. Chapter 8 presents the conclusions, and discusses the possible future work based on the contributions of this thesis.

Chapter 2

The Modern GPU Architecture and Programming Model

GPU Architecture

2.1 Introduction

The advance of the information age depended on the evolution in the capabilities of the microprocessors. The first processors were single core systems running a single thread to process instructions sequentially, running a fetch-decode-execute instruction per clock cycle. With advances in semiconductor technologies, the number of cores were increased to handle different complex tasks concurrently. Current CPU (Central Processing Units) architectures are highly tuned for MIMD instructions (Multiple Instructions, Multiple Data) with many independent cores, each running its own threads and tasks. CPUs engage heavily in prediction, speculation, large caches and complex pipelines to speed up a thread's work. They are heavily optimized for latency since MIMD instructions require reasonably fast serial computation, irregular data access on each thread and complex branching to keep the process pipelines full and busy per cycle. In each core, each thread works on a different instruction and branches to other instructions every cycle.
In contrast to this, GPUs are more geared towards SIMT (Single Instruction, Multiple Threads) paradigm; many cores (warps) execute the *same instruction* in lock-step on different data. Any branching within the instruction is homogeneous as well; all threads will execute the same branching instructions provided the control flow is uniform and non-divergent. They are optimized for high throughput on data-parallel tasks, moving,

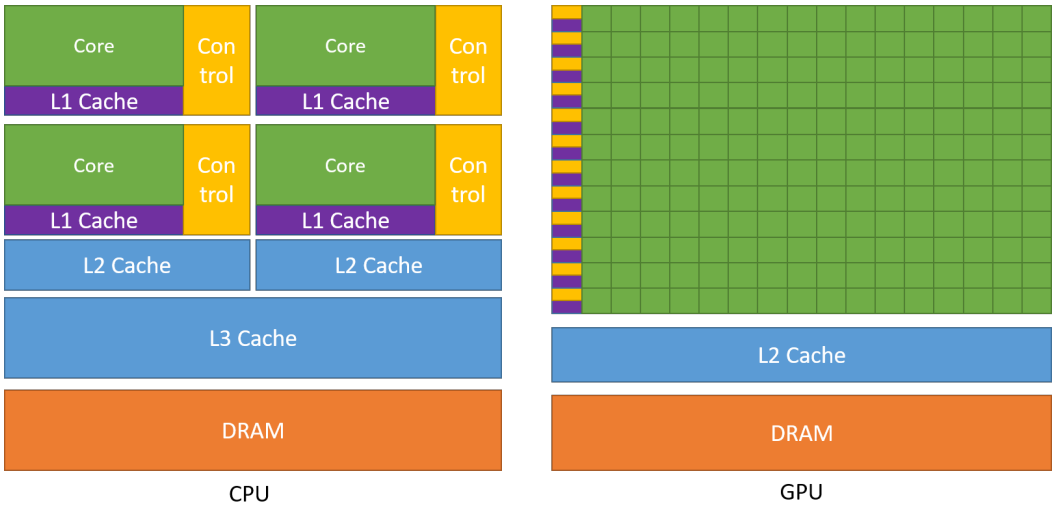


Figure 2.1: GPU vs CPU Architectures

accessing and computing on the same data per thread. A fundamentally different programming model than that of the CPU that allows divergent data and instruction paths well. As such, GPUs are quickly becoming a staple in the scientific world of high performance computing where the focus is on running algorithms for processing and computing on homogeneous data. GPUs came to prominence in 1999 when Nvidia released the first GPU, the Geforce 256. The GPU optimized 3D graphics, most widely used in video game rendering [28, 29]. Nvidia centered the design around the need to scale thousands of cores (SIMD lanes) by simplifying each core's control logic, area and power. This trade-off in control logic favors many arithmetic logc units and wide memory interfaces to increase the compute throughput on data much more. GPUs have massive parallel potential due to their thousands of cores and higher memory bandwidth, as illustrated in Fig. 2.1.

2.2 Terminology

The CUDA (Compute Unified Device Architecture framework programming model) is a hybrid computing model with both the CPU and GPU being utilized in tandem. Parallel data-heavy processing tasks are offloaded to the GPU while the CPU controls the flow of the algorithmic skeletons. The CPU is referred to as the host while the GPU is referred to as the device. The host manages kernel launches, device data allocation, and data transfers between host and device memory. A thread is the smallest basic processing unit on the GPU. Threads concurrently execute kernel code on singular elements (BLAS level-1). GPU threads are light weight and consist mainly of ALU

units and are used to primarily perform math operations on allocated device memory. A group of threads is called a *CUDA block*. A CUDA block can have up to 1024 threads. A *warp* is a block of 32 threads. Each thread in a warp executes the same set of instructions. Multiple CUDA blocks are grouped into a *grid*. *Streaming multiprocessors* (SM) can execute several blocks concurrently [30] and allocate the registers and shared memories as required by each block. A typical CUDA programming model is shown in Figure 2.2.

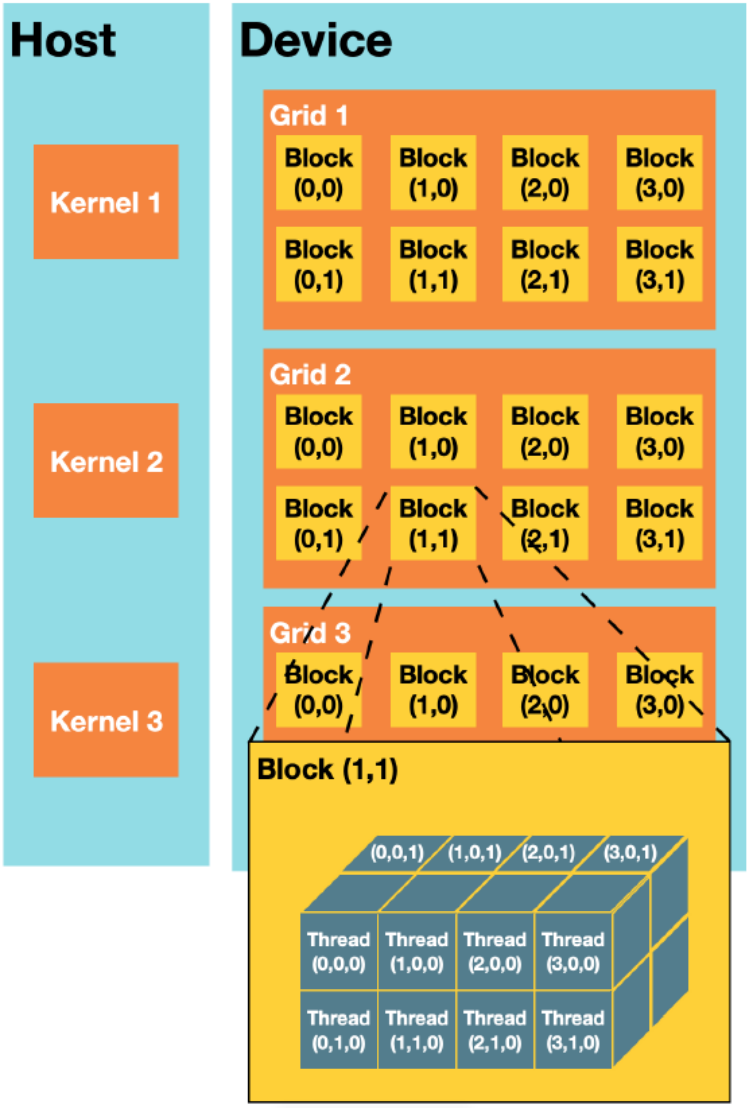


Figure 2.2: The Hybrid Programming CUDA Model

2.3 Memory Hierarchy

GPUs utilize a complex memory hierarchy consisting of on-chip storage and off-chip device memory. On-chip memory includes *registers*, *shared memory* (programmer-managed scratchpad), and hardware-managed *L1 cache*. Off-chip memory includes *global* and *local* memory residing in DRAM. Read-only *constant* and *texture* spaces are backed by device DRAM but accessed via specialized on-chip caches. On-chip memory is much faster but has limited storage capacity in comparison to off-chip memory. A typical latency ranking (fastest → slowest) is[29, 30] associated with these memories is given below:

1. Registers (per-thread, on-chip memory)
2. Shared memory (per-SM scratchpad, on-chip memory)
3. Constant / texture caches (read-only)
4. Global memory (device DRAM)
5. Local memory (per-thread storage in device DRAM; same class as global memory)

Figure 2.3 shows how threads, blocks and grids interact and access the various levels of GPU memories.

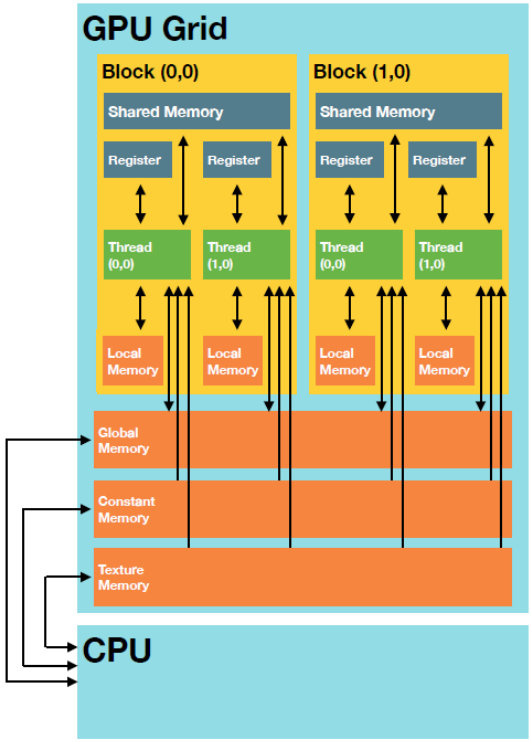


Figure 2.3: The Memory Hierarchy in a GPU

2.3.1 Global Memory

Global memory is the primary off-chip device memory in the DRAM on a GPU. The host cannot directly de-reference device global memory; data must be copied between host and device (e.g., with `cudaMemcpy`). From the device side, all blocks and threads can read/write any global memory location provided they have a valid pointer to. Global memory objects allocated with `cudaMalloc` (or declared as `__device__` variables) and persist across kernel launches for the lifetime of the object, or until explicitly freed with `cudaFree`. Because global memory has high latency, minimizing accesses (e.g., coalescing, leveraging caches, staging reusable data in shared memory) is critical to reduce memory-dependent synchronization points in a program.

2.3.2 Local Memory

Local memory is the per-thread storage that resides in device DRAM and is cached by the L1/L2 caches. It is used primarily for register data spills and for thread-local automatic variables that cannot be held in the thread registers (e.g., large arrays or variables whose address is taken). Similar to registers, its scope and visibility are limited to a single thread; other threads cannot access a thread's local memory. Because local memory is off-chip, its latency is comparable to global memory. Compilers place automatic variables in local memory when register pressure is high or when indexing patterns prevent allocation in registers.

2.3.3 Constant and Texture Caches

Constant Memory

Constant memory holds read-only data for the duration of a kernel launch. The host populates constant memory before kernel execution (e.g., `cudaMemcpyToSymbol`); during the kernel, the device can only read it. All blocks and threads may access constant memory. When all threads in a warp read the *same* constant address, the hardware broadcasts the value, yielding very low-latency, high-bandwidth access. Very often used to store input data/matrices that do not get altered during math kernels.

Texture Memory

Texture memory is another read-only path backed by device DRAM but served through specialized texture caches optimized for spatial locality (1D/2D/3D). Originally introduced for graphics (filtering, addressing modes), CUDA exposes texture fetches for

general-purpose computing as well, which can improve performance for irregular or poorly coalesced access patterns by leveraging the texture cache.

2.3.4 Shared Memory

Shared memory is an on-chip storage allocated per CUDA block. It's latency is an order of magnitude lower (often $\sim$100$\times$) than global memory, but the tradeoff is that the capacity is limited. Only threads within the *same* block can read-write to that block's shared memory. A common usage by programmers for this type of storage is to store tiles of data fetched from global memory that will be accessed repeatedly such as in MMAC operations in BLAS routines. The lifetime of shared memory spans the execution of the block, after which it is cleared.
In CUDA, shared memory is declared with the `__shared__` qualifier. Care is required when multiple threads access the same locations: use appropriate synchronization (e.g., `__syncthreads()`) to avoid race conditions. Also be mindful of bank conflicts, which can serialize accesses and degrade performance.

2.3.5 Registers

Registers are the fastest of the on-chip memories but also the smallest in capacity. They are solely local and accessible by the owning thread; threads within the same block cannot access another threads registers. Register values persist for the lifetime of the thread. If a kernel attempts to use more registers than are available (per thread), the excess is *spilled* to local memory (i.e., device DRAM via the L1/L2 caches), which incurs higher latency whenever accessed.

2.4 Compute Framework and Scalability

2.4.1 Compute Framework

CPUs and GPUs excel at different things; CPUs excel at latency-sensitive, irregular tasks (branch-heavy code, pointer chasing, operating-system functions), while GPUs can tackle massively data-parallel workloads (dense linear algebra, image processing, deep learning). In this thesis, the throughput power of GPUs is leveraged while utilizing the flow control of the algorithms on CPUs: mapping the control flow of Krylov-subspace reductions using the CPU's MIMD strengths, and offloading the dense, matrix operations to the GPU's SIMT engine (and its mixed precision Tensor Core

units) [31] for dramatic speedups. Fig. 2.4 gives a comparison of the MIMD and SIMT frameworks, highlighting how MIMD can process different instruction flows while also processing different data. Contrastingly, SIMT/SIMD architectures enable processing massively homogeneous data with a simple instruction flow [28, 29].

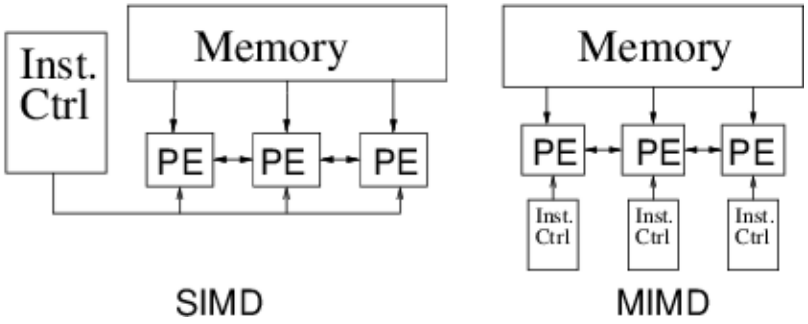


Figure 2.4: SIMD/SIMT vs MIMD

Nvidia GPUs utilize their vendor specific CUDA in order to access the data parallelism capabilities. There are three levels of abstraction in this framework: a thread group hierarchy, shared memories, and barrier synchronization. Users have programmatic access to these through a set of C/C++ language extensions [29, 30]. It allows users to achieve fine-grained data parallelism, thread level parallelism, and higher block level and task level parallelism abstractions.

These abstractions are exposed via a set of C/C++ language extensions and runtime APIs [30, 29], enabling:

1. Fine-grained data parallelism: BLAS kernels (e.g., vector addition, small matrix multiplies) map naturally onto block or warp level tiles and even allows fine-tuned thread level (single element BLAS level 1) manipulations.
2. Thread-level parallelism: Thousands of threads can be active concurrently, hiding memory latency through oversubscription.
3. Block and task-level parallelism: Independent blocks allow work to be distributed across all SMs (streaming multiprocessors) and even across multiple GPUs.

This allows fine-grained BLAS (basic linear algebra subroutines) kernels to exploit parallelism in very low data or thread levels in problem hierarchies, all the way to scheduling concurrent independent operations at the task level in the problem formulations.

2.4.2 The Programming Process Flow

In a typical CUDA program, the programmatic sequence involving device kernel launches is as follows [30]:

1. Allocate and initialize the memory needed on the host side. Invoked using `malloc`, the allocated memory is not initialized and the values are unknown. It returns typeless memory to a host-side object pointer `objPtr`:

   ```
   objPtr = (object*) malloc(num_of_objects * sizeof(object));
   ```

2. Allocate memory needed on device side and assign device side pointer (*devPtr*) to it. Similar to *malloc*:

   ```
   devPtr = (object*) cudaMalloc(void^{***} devPtr, size_t size;
   ```

3. Copy data from host to device side or vice versa. Invoking `cudaMemcpy` copies `count` bytes from the memory area pointed to by `src` (source) to the memory area pointed to by `dst` (destination), where `kind` specifies the direction of the copy:

   ```
   cudaError_t cudaMemcpy(void *dst, const void *src, size_t count,
                          cudaMemcpyKind kind);
   ```

   where, `kind` can be:
   - `cudaMemcpyHostToHost`
   - `cudaMemcpyHostToDevice`
   - `cudaMemcpyDeviceToHost`
   - `cudaMemcpyDeviceToDevice`

4. Invoke kernel(s) and pass in device pointer(s) to data as arguments. It is to be noted that device (GPU) performs computations. This is analogous to executing functions on the GPU. There are three sets of arguments needed by the kernel: the first set describes the CUDA grid/block configuration; the second set is the arguments needed by the function itself; additionally, dynamic shared memory, a third argument, can be added to the CUDA configuration specifying the amount of shared memory per block:

   ```
   kernelName<<<GridDim, BlockDim, memSize>>>(arg1, arg2, ...);
   ```

   Here, `GridDim` and `BlockDim` are of type `dim3`, a vector type provided by CUDA to specify block and grid dimensions.

5. Free host and device memory. Like host memory, device memory must be freed before termination of the allocating program. `cudaFree` frees the memory space pointed to by `devPtr`, which must have been returned by a previous call to `cudaMalloc` (or a variant). If a previously freed memory pointer is freed again, an error is generated. This freeing operation is invoked using:

   ```
   cudaError_t cudaFree(void *devPtr);
   ```

2.4.3 Scalability

CUDA also allows flexible scalability in problem sizes with multiple GPUs using the same code originally formulated for single GPU environment simply by partitioning data and launching the kernels on each device. This is demonstrated in Fig. 2.5.

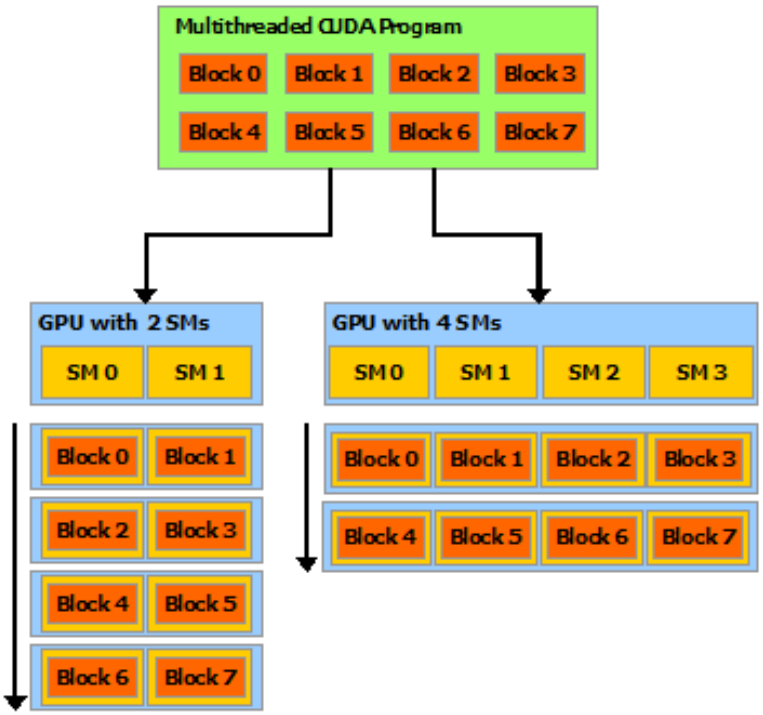


Figure 2.5: Illustration of multi-GPU architecture

2.5 Summary

In this chapter, a brief summary of the emerging GPU platforms with Tensor Cores is presented. An outline of the hybrid CUDA programming model, the memory architecture, the computing frameworks and the programming process flow while using GPUs are presented and reviewed.

Chapter 3

Projection-Based Model Order Reduction: PRIMA and SPRIM

In this chapter, a brief review of implicit moment matching techniques based on Krylov subspace based techniques, utilizing Arnoldi (Block Arnoldi) methods is presented. To overcome their limitation, mainly loss of passivity in the reduced models, PRIMA and it's block-structure preserving alternative SPRIM, as relevant to the contributions presented in this thesis, are reviewed.

3.1 MNA Descriptor Formulation for RLC Interconnects

In this thesis, analysis of large multi-port circuits such as interconnects, packages and power distribution circuits are considered. For the purpose of illustration and ease of presentation, consider the modeling of the transfer function based on an impedance matrix of a multi-port RLC network with Np ports as shown in Figure 3.1.

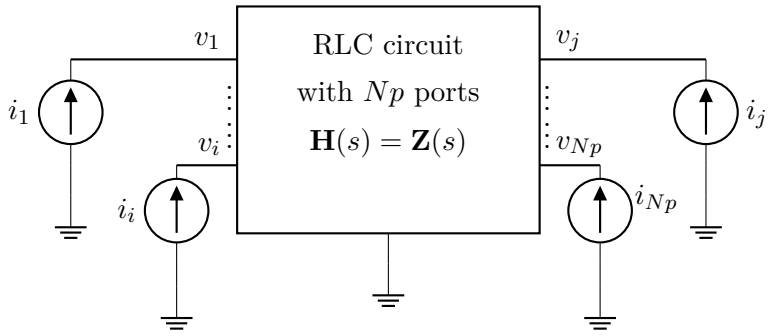


Figure 3.1: An illustrative multi-port RLC excited by current sources, represented as a black-box transfer function $H(s)$.

This linear multi-port RLC interconnect network can be formulated using Modified Nodal Analysis (MNA) [1, 4] as:

$$\mathbf{G}\,\mathbf{x}(t) + \mathbf{C}\,\dot{\mathbf{x}}(t) = \mathbf{B}\,\mathbf{I}_{Np}(t), \qquad \mathbf{V}_{Np}(t) = \mathbf{B}^{\top}\mathbf{x}(t), \tag{3.1}$$

where $\mathbf{x}(t) \in \mathbb{R}^{N_r}$ contains the MNA variables (consisting of node voltages, inductor and port currents), $\mathbf{I}_{Np}(t) \in \mathbb{R}^{Np}$ contains current source excitations at the ports, and $\mathbf{V}_{Np}(t) \in \mathbb{R}^{Np}$ contains output port voltage variables. Here N_r and Np represent the total number of MNA variables and the total number of ports, respectively.
It can be noted that the $\mathbf{G}$, $\mathbf{C}, \mathbf{B}$ matrices can be partitioned with certain inner block structures as:

$$\mathbf{G} \;=\; \begin{bmatrix} \mathbf{N} & \mathbf{E}_\ell \\ \mathbf{E}_\ell^{\mathsf{T}} & \mathbf{0} \end{bmatrix}, \qquad \mathbf{C} \;=\; \begin{bmatrix} \mathbf{Q} & \mathbf{0} \\ \mathbf{0} & -\mathcal{L} \end{bmatrix}, \qquad \mathbf{B} \;=\; \begin{bmatrix} \mathbf{B}_v \\ \mathbf{0} \end{bmatrix}, \qquad \mathbf{x} \;=\; \begin{bmatrix} \mathbf{v} \\ \mathbf{i}_\ell \end{bmatrix} \tag{3.2}$$

In (3.2), $\mathbf{N} \in \mathbb{R}^{N_{\text{int}} \times N_{\text{int}}}$, $\mathbf{Q} \in \mathbb{R}^{N_{\text{int}} \times N_{\text{int}}}$ and $\mathcal{L} \in \mathbb{R}^{N_{\text{ind}} \times N_{\text{ind}}}$ are real, sparse matrices containing resistor, capacitor and inductance stamps, respectively. These inner block matrices are symmetric with $\mathbf{N} \succeq 0$, $\mathbf{Q} \succeq 0$, $\mathcal{L} \succeq 0$ (i.e., symmetric semi-positive definite matrices). The symbol "$\succeq$" indicates semi-positive definite matrices. Here, N_{int} corresponds to the number of nodal voltage variables ($\mathbf{v}$), N_{ind} corresponds to the number of inductor current variables ($\mathbf{i}_\ell$) in the MNA variable vector $\mathbf{x}$. $\mathbf{B}_v$ contains the incidence matrix for the current sources at the ports. The size of the MNA matrix N_r can be written as:

$$N_r = \underbrace{N_{\text{int}}}_{\text{number of voltage variables}} + \underbrace{N_{\text{ind}}}_{\text{number of inductor current variables}} \tag{3.3}$$

The coupling matrix $\mathbf{E}_\ell \in \mathbb{R}^{N_{\text{ind}} \times N_{\text{int}}}$ is the incidence matrix with entries of $\{0, \pm 1\}$ corresponding to the current variables that are introduced for inductors currents, during the MNA formulation.
A direct reduction using the circuit equations in (3.1) and (3.2) using Krylov subspace techniques, may not result in *passive* reduced models. To address this for RLC circuit MNA matrices, a modified stamping of the traditional MNA inductor current variables was proposed in the PRIMA algorithm [13, 14]. The modified stampings negate the incidence entries in the `new_col` columns, as shown in Figure 3.2:

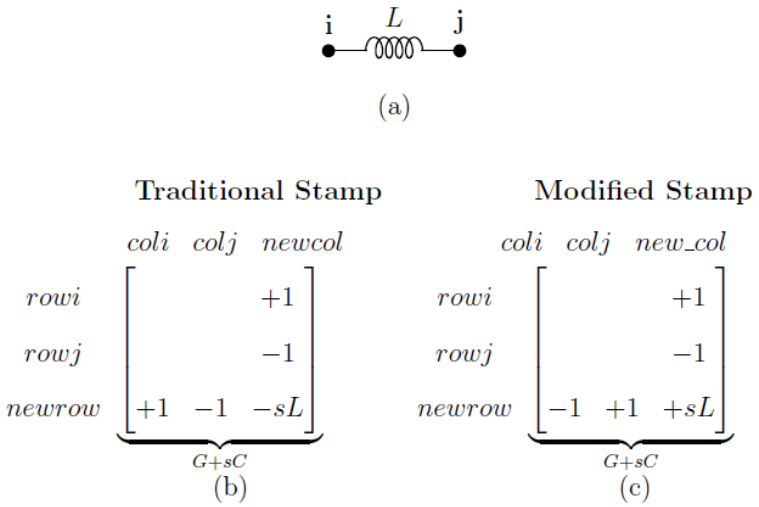


Figure 3.2: Negated stamping of inductor current incidences into MNA.

The $\mathbf{G}$, $\mathbf{C}, \mathbf{B}$ matrices inner block structures then become:

$$\mathbf{G} = \begin{bmatrix} \mathbf{N} & \mathbf{E}_\ell \\ -\mathbf{E}_\ell^{\mathsf{T}} & \mathbf{0} \end{bmatrix}, \qquad \mathbf{C} = \begin{bmatrix} \mathbf{Q} & \mathbf{0} \\ \mathbf{0} & \mathcal{L} \end{bmatrix}, \qquad \mathbf{B} = \begin{bmatrix} \mathbf{B}_v \\ \mathbf{0} \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} \mathbf{v} \\ \mathbf{i}_\ell \end{bmatrix}. \tag{3.4}$$

This modification was used to ensure passivity in the reduced system matrices (PRIMA). As an example a sample circuit with the modified stamping scheme is given in Appendix A, section A.1.1.
For RLC networks the *transfer function* $\mathbf{H}(s)$ is obtained as:

$$\begin{aligned} \mathbf{H}(s) &= \mathbf{B}^{\mathsf{T}} \left(\mathbf{G} + s\,\mathbf{C}\right)^{-1} \mathbf{B} \\ &= \mathbf{B}^{\mathsf{T}} \left[\mathbf{G}\left(\mathbf{I} - s\mathbf{A}\right)\right]^{-1} \mathbf{B} \\ &= \mathbf{B}^{\mathsf{T}} \left(\mathbf{I} - s\mathbf{A}\right)^{-1} \mathbf{G}^{-1} \mathbf{B} \\ \mathbf{H}(s) &= \mathbf{B}^{\mathsf{T}} \left(\mathbf{I} - s\mathbf{A}\right)^{-1} \mathbf{R} \end{aligned} \tag{3.5}$$

where

$$\mathbf{A} = -\mathbf{G}^{-1}\mathbf{C} \qquad \text{and} \qquad \mathbf{R} = \mathbf{G}^{-1}\mathbf{B}. \tag{3.6}$$

Note, in this thesis, with current source formulations (excitations at ports with injected currents), the transfer function $\mathbf{H}(s) = \mathbf{Z}(s)$ where $\mathbf{Z}(s)$ is the impedance matrix. For the ease of presentation in this thesis, the impedance transfer function is referred to as $\mathbf{H}(s)$ throughout.

Simulation and optimization of the systems using (3.1) and (3.5) in the context of modern VLSI circuits involve millions of nodes leading to large set of coupled equations; solution of which can be prohibitively CPU expensive. This difficulty was first addressed using explicit moment matching techniques like AWE [11]. However, they tend to be ill-conditioned for higher-order approximations. This was addressed using Krylov subspace based approaches using Arnoldi iterations [6, 13].

3.2 Arnoldi/Block-Arnoldi Iterations for Implicit Moment Matching

Using the matrices in (3.6) the *block Krylov subspace* spanning $q = (n.Np)$ columns at an expansion point $s_0 = 0$ can be obtained as:

$$\mathcal{K}_q(\mathbf{A}, \mathbf{R}) \;=\; \text{colspan}\{\, \mathbf{R},\, \mathbf{AR},\, \mathbf{A}^2\mathbf{R},\, \ldots,\, \mathbf{A}^{q-1}\mathbf{R} \,\}. \tag{3.7}$$

Here, n represents the number of block moments considered. In the block Arnoldi algorithm [21], an orthonormal basis $\mathbf{X} \in \mathbb{R}^{N_r \times q}$ of $\mathcal{K}_q(\mathbf{A}, \mathbf{R})$ and a block upper–Hessenberg matrix $\mathscr{H}_q$ are obtained such that:

$$\mathbf{X}^{\mathsf{T}}\mathbf{X} = \mathbf{I}, \qquad \mathbf{A}\,\mathbf{X} \;=\; \mathbf{X}\,\mathscr{H}_q \tag{3.8}$$

Using the projection basis $\mathbf{X}$, the system transfer function (3.5) can be approximated as:

$$\mathbf{H}_q(s) \;=\; \mathbf{L}^{\mathsf{T}}\mathbf{X}\left(\mathbf{I} - s\mathscr{H}_q\right)^{-1}\mathbf{X}^{\mathsf{T}}\mathbf{R}, \tag{3.9}$$

It can be noted that, in (3.9) the first q block moments of $\mathbf{H}(s)$ are *implicitly matched* at the expansion point $s_0 = 0$ [8, 12], with:

$$\mathbf{H}(s) = \sum_{k=0} \mathbf{M}_k s^k, \qquad \mathbf{H}_q(s) = \sum_{k=0} \mathbf{M}_{k,q} s^k, \qquad \mathbf{M}_{k,q} = \mathbf{M}_k \;\text{ for } k = 0, \ldots, q-1. \tag{3.10}$$

where

$$\mathbf{M}_k \;=\; \mathbf{B}^{\mathsf{T}}\mathbf{A}^k\mathbf{R}, \qquad \mathbf{M}_{k,q} \;=\; \mathbf{B}^{\mathsf{T}}\mathbf{X}\,\mathscr{H}_q^{\,k}\,\mathbf{X}^{\mathsf{T}}\,\mathbf{R} \tag{3.11}$$

Here $\mathbf{M}_{k,q}$ denote the moments of the block upper-Hessenburg reduced system, and $\mathbf{M}_k$ denote the moments of the original, system. The advantage of the (Block) Arnoldi iterations is that it uses implicit moment matching avoiding forming moments explicitly (as in AWE) therefore, improving the numerical robustness and conditioning. Since Arnoldi iterations create a diagonalized system and an upper block Hessenberg matrix $\mathscr{H}_q$, the system can be further eigen-decomposed to speed-up the frequency response calculation of the system. The eigen-decomposition of $\mathscr{H}_q$ gives

$$\mathscr{H}_q = \mathbf{S}\,\mathbf{\Lambda}\,\mathbf{S}^{-1}.$$

This facilitates fast evaluation of the reduced transfer function as:

$$\mathbf{H}_q(s) \;=\; \mathbf{L}^{\mathsf{T}}\mathbf{X}\,\mathbf{S}\left(\mathbf{I} - s\,\mathbf{\Lambda}\right)^{-1}\mathbf{S}^{-1}\mathbf{X}^{\mathsf{T}}\mathbf{R},$$

Here, $\left(\mathbf{I} - s\,\mathbf{\Lambda}\right)^{-1}$ is easy to calculate since it is a diagonal matrix and its inverse computation can be faster since it involves only computing the inverse of the individual diagonal entries within it.
It is to be noted that just the Arnoldi based reduction may not preserve the passivity of the reduced model. To address this issue, the PRIMA [14] algorithm was introduced.

3.3 PRIMA: Passive Reduced Interconnect Modeling via Projectors

The PRIMA [14] algorithm proceeds by building a set of orthonormal basis $\mathbf{X} \in \mathbb{R}^{N_r \times q}$ that spans the Krylov subspace

$$\mathcal{K}_q(\mathbf{A}, \mathbf{R}) \;=\; \text{span}\,\{\,\mathbf{R},\; \mathbf{AR},\; \mathbf{A}^2\mathbf{R},\; \ldots,\; \mathbf{A}^{\,q-1}\mathbf{R}\,\}. \tag{3.12}$$

obtained via the block Arnoldi algorithm (as outlined in the previous section). The basis is then used to directly reduce the system matrices in PRIMA. Since the basis projector is orthonormal, the basis projector $\mathbf{X}$ has the property $\mathbf{X}^{\mathsf{T}}\mathbf{X} = \mathbf{I}$ (equation 3.8). The reduced model matrices are obtained by using the *split congruence* projection as:

$$\mathbf{G}_q \;=\; \mathbf{X}^{\mathsf{T}}\mathbf{G}\,\mathbf{X}, \qquad \mathbf{C}_q \;=\; \mathbf{X}^{\mathsf{T}}\mathbf{C}\,\mathbf{X}, \qquad \mathbf{B}_q \;=\; \mathbf{X}^{\mathsf{T}}\mathbf{B}, \tag{3.13}$$

with the corresponding dimensions: $\mathbf{G}_q \in \mathbb{R}^{q\times q}$, $\mathbf{C}_q \in \mathbb{R}^{q\times q}$ and $\mathbf{B}_q \in \mathbb{R}^{q\times Np}$. The reduced transfer function can be obtained as:

$$\mathbf{H}_q(s) \;=\; \mathbf{B}_q^{\mathsf{T}}\,(\mathbf{G}_q + s\,\mathbf{C}_q)^{-1}\mathbf{B}_q. \tag{3.14}$$

The change in variables between the original system variables $\mathbf{x}(t) \in \mathbb{R}^{N_r\times 1}$ and the reduced set of variables $\hat{\mathbf{x}}(t) \in \mathbb{R}^{q\times 1}$ are linked by

$$\mathbf{x}(t) \approx \mathbf{X}\,\hat{\mathbf{x}}(t). \tag{3.15}$$

In the time-domain, the reduced system of MNA equations of (3.1) can be written as:

$$\begin{aligned} \mathbf{C}_q\,\dot{\hat{\mathbf{x}}}(t) \;+\; \mathbf{G}_q\,\hat{\mathbf{x}}(t) &= \mathbf{B}_q\,\mathbf{I}_{Np}(t), \\ \mathbf{V}_{Np}(t) &= \mathbf{B}_q^{\mathsf{T}}\,\hat{\mathbf{x}}(t). \end{aligned} \tag{3.16}$$

Once the system matrices are available, the steps of the PRIMA algorithm suitable for CPU-based platforms can be carried out as outlined in Algorithm 1 and Algorithm 2.

Algorithm 1 PRIMA Algorithm

Inputs: Conductance matrix $\mathbf{G} \in \mathbb{R}^{N_r \times N_r}$; capacitance matrix $\mathbf{C} \in \mathbb{R}^{N_r \times N_r}$; input voltage-source matrix $\mathbf{B} \in \mathbb{R}^{N_r \times Np}$; number of block moments matched $= n$ and number of Krylov subspace columns $q = n \times Np$; dimensions N_r (states), Np (ports).
Output: reduced matrices $\mathbf{G}_q, \mathbf{C}_q, \mathbf{B}_q$ and admittance $\mathbf{Y}(j\omega)$.
Step 1: Compute a fill-reducing ordering and factorize $\mathbf{G}$ once:

$$\mathbf{PGQ} = \mathbf{LU}.$$

Solve $\mathbf{GR} = \mathbf{B}$ for $\mathbf{R}$ using the $\mathbf{L}, \mathbf{U}, \mathbf{P}, \mathbf{Q}$ factors.

Step 2: Perform Block Arnoldi using Algorithm 2. Pass matrices $\mathbf{C}, \mathbf{R}, \mathbf{L}, \mathbf{U}, \mathbf{P}, \mathbf{Q}$ from previous step as inputs and generate Krylov vectors $\mathbf{X} \in \mathbb{R}^{N_r \times q}$ spanning

$$\mathcal{K}_n(\mathbf{A}, \mathbf{R}) = \text{span}\{\mathbf{R},\ \mathbf{AR},\ \mathbf{A}^2\mathbf{R}, \ldots\}.$$

Step 3: Reduce system matrices via generated Krylov projector:

$$\mathbf{G}_q = \mathbf{X}^\top \mathbf{G} \mathbf{X}, \quad \mathbf{C}_q = \mathbf{X}^\top \mathbf{C} \mathbf{X}, \quad \mathbf{B}_q = \mathbf{X}^\top \mathbf{B}.$$

Step 4: Form the reduced system matrix (via LU of $\mathbf{G}_q$ and triangular solves)

$$\mathbf{A}_q \leftarrow \mathbf{G}_q^{-1} \mathbf{C}_q.$$

Step 5: Eigen-decompose the reduced system matrix

$$\mathbf{A}_q = \mathbf{S}\, \mathbf{\Lambda}\, \mathbf{S}^{-1}.$$

Step 6: Form residue factors

$$\mathbf{W} \leftarrow \mathbf{X}^\top \mathbf{R}, \quad \mathbf{B}_q \leftarrow \mathbf{X}^\top \mathbf{B},$$

$$\mathbf{v} \leftarrow \mathbf{S}^{-1} \mathbf{W}, \quad \boldsymbol{\mu} \leftarrow \mathbf{S}^\top \mathbf{B}_q.$$

Step 7: For each ω, assemble a batch of admittance entries

$$\mathbf{H}_{a,b}(j\omega) = \sum_{c=1}^{q} \frac{\boldsymbol{\mu}_{(a,c)}\, \mathbf{v}_{(b,c)}}{1 - (j\omega)\, \Lambda_{c,c}}\,.$$

Return $\mathbf{G}_q, \mathbf{C}_q, \mathbf{B}_q$ and $\mathbf{H}(j\omega)$

Algorithm 2 Block Arnoldi algorithm for CPU-based platform

Inputs: Sparse $\mathbf{C} \in \mathbb{R}^{N_r \times N_r}$, sparse LU factors of $\mathbf{G}$ with permutations: $\mathbf{L}, \mathbf{U}, \mathbf{P}, \mathbf{Q}$, starting block $\mathbf{R} = \mathbf{G}^{-1}\mathbf{B} \in \mathbb{R}^{N_r \times N_p}$, number of block moments matched: n and total basis columns: $q = nN_p$.

Outputs: Orthonormal basis $\mathbf{X} = [\mathbf{X}_0, \ldots, \mathbf{X}_{n-1}] \in \mathbb{R}^{N_r \times q}$ spanning $\mathcal{K}_n(\mathbf{A}, \mathbf{R})$.

Step 1: Initialize first block: tall–skinny QR decomposition

Compute thin QR of the starting block:

$\mathbf{R} = \mathbf{X}_0 \mathbf{R}_0$ (TSQR; $\mathbf{X}_0 \in \mathbb{R}^{N_r \times N_p}$)

Initialize the global basis: $\mathbf{X} \leftarrow [\mathbf{X}_0]$.

Step 2: Arnoldi loop (for $k = 1, \ldots, n-1$)

(a) Krylov updates to form $-\mathbf{G}^{-1}(\mathbf{C}\mathbf{X}_{k-1})$

Compute the sparse-dense matrix block product:

$\mathbf{S}_k \leftarrow \mathbf{C}\,\mathbf{X}_{k-1}$ (Sparse $\mathbf{C}$, dense block $\mathbf{X}_{k-1}$)

Solve $\mathbf{G}\mathbf{W}_k = -\mathbf{S}_k$ using the input LU factors:

$\mathbf{F} \leftarrow -\,\mathbf{P}\mathbf{S}_k$ (apply row permutation)

Solve $\mathbf{L}\mathbf{Z} = \mathbf{F}$ (forward solve)

Solve $\mathbf{U}\mathbf{T} = \mathbf{Z}$ (backward solve)

$\mathbf{W}_k \leftarrow \mathbf{Q}\mathbf{T}$ (apply column permutation back)

(b) Block Modified Gram–Schmidt orthogonalization

for $j = 1, \ldots, k$ **do**

$\mathbf{H}_{k,j} \leftarrow \mathbf{X}_{k-j}^{\top}\mathbf{W}_k$

$\mathbf{W}_k \leftarrow \mathbf{W}_k - \mathbf{X}_{k-j}\mathbf{H}_{k,j}$

end for

(c) Ortho-normalize the block

$\mathbf{W}_k = \mathbf{X}_k\,\mathbf{R}_k$ (TSQR)

(d) Append block to the basis

$\mathbf{X} \leftarrow [\,\mathbf{X} \mid \mathbf{X}_k\,]$

Step 3: Collect output

Orthonormal basis $\mathbf{X} \in \mathbb{R}^{N_r \times q}$.

3.4 SPRIM: Structure-Preserving Reduced Interconnect Modelling

It is to be noted that, PRIMA reduced models do not preserve structural properties (e.g, the block pattern) that are inherent to the original MNA formulation. This often complicates the process of converting reduced matrices to physically realizable circuits. To address this, SPRIM [16, 18] was introduced. Similar to the PRIMA algorithm, a Krylov subspace spanning q columns as in (3.12) is obtained. Subsequently, in the SPRIM algorithm, the Krylov projection process is modified so that the projection basis conforms to the original MNA block structure in (3.4). This is done by partitioning the projection basis $\mathbf{X}$ as illustrated in Figure 3.3.

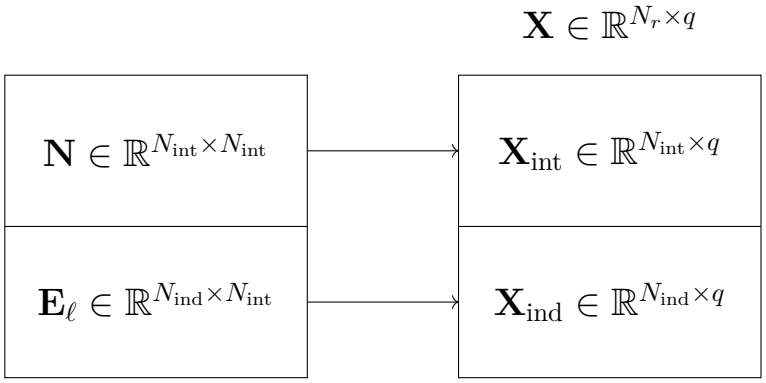


Figure 3.3: Partitioning the Krylov projection basis $\mathbf{X}$ according to inner blocks of $\mathbf{G}$.

Each block, $\mathbf{X}_{ind}$ and $\mathbf{X}_{int}$ are subjected to orthonormalization to obtain orthonormalized blocks $\widehat{\mathbf{X}}_{ind}$ and $\widehat{\mathbf{X}}_{int}$. This is followed by assembling the block diagonal projector as:

$$\widetilde{\mathbf{X}} \;=\; \text{diag}(\widehat{\mathbf{X}}_{\text{int}}, \widehat{\mathbf{X}}_{\text{ind}}) \tag{3.17}$$

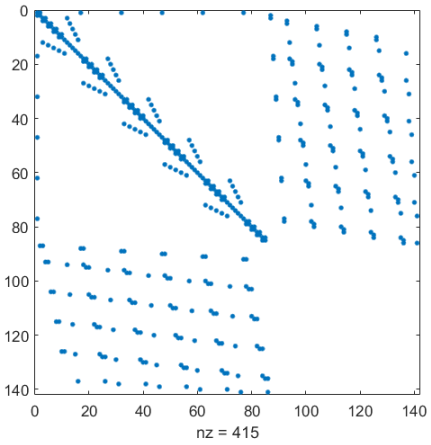
Once the block diagonal projector $\widetilde{\mathbf{X}}$ is obtained in, orthogonal split congruence transformation is used for reducing the system matrices while maintaining the positive semi-definiteness of the matrices, while as well as maintaining the underlying block structure:

$$\mathbf{G}_q \;=\; \widetilde{\mathbf{X}}^{\mathsf{T}} \mathbf{G}\, \widetilde{\mathbf{X}}, \qquad \mathbf{C}_q \;=\; \widetilde{\mathbf{X}}^{\mathsf{T}} \mathbf{C}\, \widetilde{\mathbf{X}}, \qquad \mathbf{B}_q \;=\; \widetilde{\mathbf{X}}^{\mathsf{T}} \mathbf{B}, \tag{3.18}$$

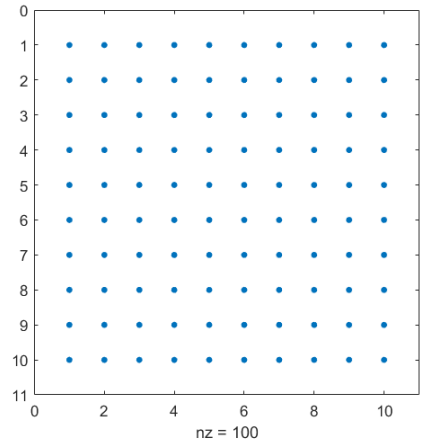
Noting that the relationship between the original ($\mathbf{x}(t)$) and reduced set of variables ($\hat{\mathbf{x}}(t)$) is given by, $\mathbf{x}(t) \approx \widetilde{\mathbf{X}}\, \hat{\mathbf{x}}(t)$, the reduced transfer function can be obtained as:

$$\mathbf{H}_q(s) \;=\; \mathbf{B}_q^{\mathsf{T}} \left(\mathbf{G}_q + s\, \mathbf{C}_q\right)^{-1} \mathbf{B}_q. \tag{3.19}$$

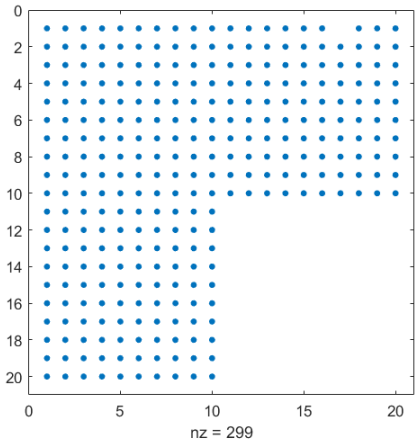
A pseudocode for the SPRIM algorithm is given in Algorithm 3.
A further illustration of preservation of block structures is demonstrated in Figure 3.4. The original MNA matrices of a 5-port network ($Np = 5$) of size 141×141, is reduced by matching $n = 2$ block moments by PRIMA and SPRIM. The corresponding PRIMA and SPRIM reduced models have dimensions 10×10 and 20×20, respectively. It is to be noted that the PRIMA results in a reduced model of size $q \times q$ with $q = Np \times$ n. On the other hand, SPRIM reduction for this results in $q_{tot} = 2q$ and consequently with a reduced model of size $2q \times 2q$.



(a) Original G matrix



(b) G_q from PRIMA (no block structure preservation)



(c) G_q after SPRIM (block structure preservation)

Figure 3.4: Sparsity structure of the original conductance matrix G and its reduced forms G_q from PRIMA and SPRIM.

Algorithm 3 SPRIM Algorithm

Inputs: conductance matrix $\mathbf{G} \in \mathbb{R}^{N_r \times N_r}$, capacitance matrix $\mathbf{C} \in \mathbb{R}^{N_r \times N_r}$, input matrix $\mathbf{B} \in \mathbb{R}^{N_r \times Np}$, counts: N_{int} (number of internal nodes), N_{ind} (number of inductors), Np (number of ports), with $N_r = N_{\text{int}} + N_{\text{ind}}$; target number of block moments matched $= n$ and number of Krylov subspace columns $q = n \times Np$.
Outputs: reduced matrices $\mathbf{G}_q, \mathbf{C}_q, \mathbf{B}_q$ and admittance $\mathbf{Y}(j\omega)$.
Step 1: Compute a fill-reducing ordering and factorize $\mathbf{G}$ once:

$$\mathbf{PGQ} = \mathbf{LU}.$$

Solve $\mathbf{GR} = \mathbf{B}$ for $\mathbf{R}$ using the $\mathbf{L}, \mathbf{U}, \mathbf{P}, \mathbf{Q}$ factors.

Step 2: Perform Block Arnoldi using Algorithm 2. Pass matrices $\mathbf{C}, \mathbf{R}, \mathbf{L}, \mathbf{U}, \mathbf{P}, \mathbf{Q}$ from previous step as inputs and generate Krylov vectors $\mathbf{X} \in \mathbb{R}^{N_r \times q}$ spanning

$$\mathcal{K}_n(\mathbf{A}, \mathbf{R}) = \text{span}\{\mathbf{R},\ \mathbf{AR},\ \mathbf{A}^2\mathbf{R}, \ldots\}.$$

Step 3: Partition $\mathbf{X}$ by *rows* according to SPRIM blocks

$$\mathbf{X} \ \rightarrow \ \begin{bmatrix} \mathbf{X}_{\text{int}} \\ \mathbf{X}_{\text{ind}} \end{bmatrix}, \quad \mathbf{X}_{\text{int}} \in \mathbb{R}^{N_{\text{int}} \times q},\ \mathbf{X}_{\text{ind}} \in \mathbb{R}^{N_{\text{ind}} \times q}$$

Step 4: *Perform Modified Gram Schmidt orthonormalization (per block)*

$$\widehat{\mathbf{X}}_{\text{int}} \leftarrow \text{MGS}(\mathbf{X}_{\text{int}}), \qquad \widehat{\mathbf{X}}_{\text{ind}} \leftarrow \text{MGS}(\mathbf{X}_{\text{ind}})$$

Step 5: Assemble the block–diagonal SPRIM projector

$$\widetilde{\mathbf{X}} \ \leftarrow \ \text{blkdiag}\big(\widehat{\mathbf{X}}_{\text{int}}, \widehat{\mathbf{X}}_{\text{ind}}\big), \qquad \widetilde{\mathbf{X}} \in \mathbb{R}^{N_r \times q_{tot}}, \quad q_{\text{tot}} = 2q$$

Step 6: Reduce the system matrices

$$\mathbf{G}_q = \widetilde{\mathbf{X}}^\top \mathbf{G}\, \widetilde{\mathbf{X}}, \qquad \mathbf{C}_q = \widetilde{\mathbf{X}}^\top \mathbf{C}\, \widetilde{\mathbf{X}}, \qquad \mathbf{B}_q = \widetilde{\mathbf{X}}^\top \mathbf{B}.$$

Step 7: Frequency–domain response over a frequency range of interest
For each ω:

$$\big(\mathbf{G}_q + (j\omega)\, \mathbf{C}_q\big)\, \mathbf{z}(\omega) = \mathbf{B}_q, \qquad \mathbf{H}(j\omega) = \mathbf{B}_q^\top\, \mathbf{z}(\omega), \qquad \mathbf{z}(\omega) \in \mathbb{R}^{q \times N_p}.$$

Return the reduced matrices $\mathbf{G}_q, \mathbf{C}_q, \mathbf{B}_q$ and $\mathbf{H}(j\omega)$.

3.4.1 Second–order form of the SPRIM reduced model

From the above discussions, it may seem that SPRIM results in reduced matrix sizes that are twice large when compared to that from PRIMA. However, this can be addressed by casting SPRIM reduced models to the second-order form which results in reduced systems of size $q \times q$, same as that of PRIMA [16]. After SPRIM projection, the reduced MNA equations of the RLC network have the block form

$$\mathbf{G}_q \;=\; \begin{bmatrix} \mathbf{N}_q & \mathbf{E}_{\ell,q} \\ -\mathbf{E}_{\ell,q}^{\mathsf{T}} & \mathbf{0} \end{bmatrix}, \qquad \mathbf{C}_q \;=\; \begin{bmatrix} \mathbf{Q}_q & \mathbf{0} \\ \mathbf{0} & \boldsymbol{\mathcal{L}}_q \end{bmatrix}, \qquad \mathbf{B}_q \;=\; \begin{bmatrix} \mathbf{B}_{v,q} \\ \mathbf{0} \end{bmatrix}, \qquad \widehat{\mathbf{x}} \;=\; \begin{bmatrix} \widehat{\mathbf{v}} \\ \widehat{\mathbf{i}}_\ell \end{bmatrix}. \tag{3.20}$$

Here $\widehat{\mathbf{v}}$ denotes the reduced node voltages, $\widehat{\mathbf{i}}_\ell$ the reduced inductor branch currents, $\mathbf{N}_q$ is the reduced resistive part, $\mathbf{Q}_q$ and $\boldsymbol{\mathcal{L}}_q$ are the reduced capacitor and inductor matrices, and $\mathbf{B}_{v,q}$ injects the port currents into the reduced voltage nodes. The corresponding first–order transfer function in the Laplace domain is

$$\mathbf{H}_q(s) \;=\; \mathbf{B}_q^{\mathsf{T}}\Big(\mathbf{G}_q + s\,\mathbf{C}_q\Big)^{-1}\mathbf{B}_q. \tag{3.21}$$

Taking the inverse of the positive–definite block $\boldsymbol{\mathcal{L}}_q$ in the second block row of the reduced MNA system, solving for $\widehat{\mathbf{i}}_\ell(s)$ in the frequency domain, and substituting back into the upper row block of (3.20), a reduced second–order operator can be obtained as:

$$\boldsymbol{\mathcal{Q}}_q(s) \;=\; s\,\mathbf{Q}_q + \mathbf{N}_q + \frac{1}{s}\,\mathbf{E}_{\ell,q}^{\mathsf{T}}\boldsymbol{\mathcal{L}}_q^{-1}\mathbf{E}_{\ell,q}. \tag{3.22}$$

Because $\mathbf{B}_q = \Big[\mathbf{B}_{v,q}^{\mathsf{T}},\, \mathbf{0}^{\mathsf{T}}\Big]^{\mathsf{T}}$, the reduced second–order transfer function becomes

$$\mathbf{H}_q(s) \;=\; \mathbf{B}_{v,q}^{\mathsf{T}}\,\boldsymbol{\mathcal{Q}}_q(s)^{-1}\,\mathbf{B}_{v,q}. \tag{3.23}$$

It can be noted from (3.22) and (3.23) that the reduced system from the SPRIM algorithm in it's *second order* form is of the same size as that from PRIMA. However, it is to be noted that if the SPRIM model is realized in second-order form, although it can be CPU efficient, it ends up losing the block structure.

3.5 Moment Matching Considerations in PRIMA and SPRIM

Consider the original MNA descriptor system in equations (3.1) and an expansion point s_0. The system matrices and the general block moments can be computed as:

$$\mathbf{A} \;=\; -\left(\mathbf{G}+s_0\mathbf{C}\right)^{-1}\mathbf{C}, \qquad \mathbf{R} \;=\; \left(\mathbf{G}+s_0\mathbf{C}\right)^{-1}\mathbf{B}. \tag{3.24}$$

The *block moments* of $\mathbf{H}(s)$ are obtained as the Taylor coefficients of $\mathbf{H}$ at s_0 [10, 14]:

$$\mathbf{H}(s) \;=\; \sum_{k=0}^{\infty}\mathbf{M}_k\left(s-s_0\right)^k, \qquad \mathbf{M}_k \;=\; \mathbf{B}^{\mathsf{T}}\mathbf{A}^k\mathbf{R}. \tag{3.25}$$

This section considers the moment matching capabilities of PRIMA and SPRIM reduced models with respect to the original system models.

3.5.1 PRIMA: moment matching via split congruence transformation

Let $\mathbf{X} \in \mathbb{R}^{N_r \times q}$ be the real basis spanning the Krylov subspace generated using Block Arnoldi, $\mathcal{K}_q(\mathbf{A},\mathbf{R}) = \text{span}\{\mathbf{R}, \mathbf{AR}, \ldots, \mathbf{A}^{q-1}\mathbf{R}\}$. The reduced system matrices are obtained using equation (3.13) with the associated transfer function as given in (3.14). It is to be noted that PRIMA matches the first q block moments at the expansion point $s_0 = 0$ [14]. The reduced system transfer function can be expanded using the Taylor expansion coefficients as:

$$\mathbf{H}_q(s) = \sum_{k=0}\mathbf{M}_{k,q}\left(s-s_0\right)^k, \qquad \mathbf{M}_{k,q} = \mathbf{B}_q^{\mathsf{T}}\mathbf{A}_q^k\mathbf{R}_q, \tag{3.26}$$

with

$$\mathbf{A}_q = \mathbf{G}_q^{-1}\mathbf{C}_q, \qquad \mathbf{R}_q = \mathbf{G}_q^{-1}\mathbf{B}_q. \tag{3.27}$$

Next, noting the relations in equation (3.8) and equation (3.13), substituting the expressions for $\mathbf{G_q}$, $\mathbf{C_q}$ and $\mathbf{B_q}$ into equation (3.27) and equating the block moments $\mathbf{M}_{k,q}$ of the reduced system with the block moments $\mathbf{M}$ of the original system we have:

$$\mathbf{A}^k\mathbf{R} = \mathbf{X}\boldsymbol{\mathcal{H}}^k\mathbf{X}^{\mathsf{T}}\mathbf{R}, \quad \mathbf{R} = \mathbf{G}^{-1}\mathbf{B}, \quad \text{and} \quad \mathbf{X}^{\mathsf{T}}\mathbf{X} = \mathbf{I}. \tag{3.28}$$

and further, it can be shown that:

$$\begin{aligned} \mathbf{A}\,\mathbf{A}^{k-1}\mathbf{R} &= \mathbf{X}\,\boldsymbol{\mathcal{H}}^{k}\mathbf{X}^{\mathsf{T}}\mathbf{R}, \\ \mathbf{G}^{-1}\mathbf{C}\,\mathbf{A}^{k-1}\mathbf{R} &= \mathbf{X}\,\boldsymbol{\mathcal{H}}^{k}\mathbf{X}^{\mathsf{T}}\mathbf{R}, \\ \mathbf{X}(\mathbf{X}^{t}\mathbf{G}\mathbf{X})^{-1}\mathbf{X}^{t}\mathbf{C}\,\mathbf{A}^{k-1}\mathbf{R} &= \mathbf{X}\,\boldsymbol{\mathcal{H}}^{k}\mathbf{X}^{\mathsf{T}}\mathbf{R}, \end{aligned} \tag{3.29}$$

Next, equating (3.29) with (3.28) the following can be obtained:

$$\mathbf{A}^{k}\mathbf{R} \;=\; (\mathbf{K})^{k}\mathbf{R}. \tag{3.30}$$

for $k = 0, \ldots, q-1$. Consequently, it can be noted that $\mathbf{K} = -\mathbf{X}(\mathbf{X}^{\mathsf{T}}\mathbf{G}\mathbf{X})^{-1}\mathbf{X}^{\mathsf{T}}\mathbf{C}$. Substituting $\mathbf{K}$ in equation (3.29), the following can be obtained:

$$\begin{aligned} \mathbf{K}^{k}\mathbf{X} &= \mathbf{X}\Big(-(\mathbf{X}^{\mathsf{T}}\mathbf{G}\mathbf{X})^{-1}\mathbf{X}^{\mathsf{T}}\mathbf{C}\mathbf{X}\Big)^{k} \\ &= \mathbf{X}\,(\mathbf{G}_{q}^{-1}\mathbf{C}_{q})^{k} \end{aligned} \tag{3.31}$$

Substituting $\mathbf{K}^{k}\mathbf{X}$ from above into the moment equation for the reduced system:

$$\mathbf{M}_{k,q} = \mathbf{B}^{\mathsf{T}}\mathbf{X}(\mathbf{G}_{q}^{-1}\mathbf{C}_{q})^{k}\mathbf{G}_{q}^{-1}\mathbf{B}_{q} = \mathbf{B}_{q}^{\mathsf{T}}\mathbf{K}^{\,k}\mathbf{X}\mathbf{R}_{q} \tag{3.32}$$

Equating (3.32) and (3.28), evaluating at $k = 0$ and then multiplying both sides by $\mathbf{X}^{\mathsf{T}}\mathbf{G}$ gives:

$$\begin{aligned} \mathbf{X}^{\mathsf{T}}\mathbf{G}\mathbf{R} &= (\mathbf{X}^{\mathsf{T}}\mathbf{G}\mathbf{X})(\mathbf{X}^{\mathsf{T}}\mathbf{R}), \\ \mathbf{R} &= \mathbf{X}\,(\mathbf{X}^{\mathsf{T}}\mathbf{G}\mathbf{X})^{-1}\mathbf{X}^{\mathsf{T}}\mathbf{B}, \\ \mathbf{R} &= \mathbf{X}\,(\mathbf{R}_{\mathsf{q}}) \end{aligned} \tag{3.33}$$

Next, combining (3.32) and (3.33) the following can be noted:

$$\mathbf{M}_{k,q} = \mathbf{B}^{\mathsf{T}}\mathbf{K}^{k}\mathbf{R} = \mathbf{M}_{k}$$

where $\mathbf{M}_{k} = \mathbf{B}^{\mathsf{T}}\mathbf{A}^{k}\mathbf{R}$ are the moments of the original circuit. PRIMA thus matches q moments of the original circuit transfer function.

3.5.2 SPRIM: moment matching via split diagonal congruence transformation

Using similar steps as the case in PRIMA, the moment matching proof for SPRIM can be obtained by using the circuit equations in (3.1) and the system matrices as defined in (3.24). The projector $\widetilde{\mathbf{X}}$ is chosen so that it spans q columns of the block Krylov space

$$\mathcal{K}_q(\mathbf{A}, \mathbf{R}) = \text{span}\{\mathbf{R},\, \mathbf{AR},\, \mathbf{A}^2\mathbf{R},\, \ldots,\, \mathbf{A}^{q-1}\mathbf{R}\},$$

According to [16, 18] there exist residual blocks $\mathbf{\Theta}_i$, $i = 1, \ldots, q$, such that

$$\mathbf{A}^{\,i-1}\mathbf{R} \;=\; \widetilde{\mathbf{X}}\,\mathbf{\Theta}_i \tag{3.34}$$

Noting that $(\mathbf{G} + s_0\mathbf{C})\mathbf{R} = \mathbf{B}$ and $\mathbf{R} = \widetilde{\mathbf{X}}\mathbf{\Theta}_1$ when $i = 1$, the following can be formulated:

$$\begin{aligned}
(\mathbf{G} + s_0\mathbf{C})\,\widetilde{\mathbf{X}}\,\mathbf{\Theta}_1 &= \mathbf{B},\\
\widetilde{\mathbf{X}}^{\mathsf{T}}(\mathbf{G} + s_0\mathbf{C})\,\widetilde{\mathbf{X}}\,\mathbf{\Theta}_1 &= \widetilde{\mathbf{X}}^{\mathsf{T}}\mathbf{B},\\
(\mathbf{G}_q + s_0\mathbf{C}_q)\,\mathbf{\Theta}_1 &= \widetilde{\mathbf{X}}^{\mathsf{T}}\mathbf{B},\\
\mathbf{\Theta}_1 \;=\; \mathbf{R}_q &= (\mathbf{G}_q + s_0\mathbf{C}_q)^{-1}\widetilde{\mathbf{X}}^{\mathsf{T}}\mathbf{B}.
\end{aligned} \tag{3.35}$$

where $\mathbf{G}_q$, and $\mathbf{C}_q$ are defined as in (3.18).

Recursive relationship for $\mathbf{\Theta}_i$ ($i \geq 2$): Using (3.34) and the expression for $\mathbf{A}$ from equation 3.34, a recursive relationship can be obtained as:

$$(\mathbf{G} + s_0\mathbf{C})\,\widetilde{\mathbf{X}}\mathbf{\Theta}_i \;=\; \mathbf{C}\,\widetilde{\mathbf{X}}\mathbf{\Theta}_{i-1}.$$

Multiplying both sides of the above with $\widetilde{\mathbf{X}}^{\mathsf{T}}$ gives

$$(\mathbf{G}_q + s_0\mathbf{C}_q)\,\mathbf{\Theta}_i \;=\; \mathbf{C}_q\,\mathbf{\Theta}_{i-1};\quad \mathbf{\Theta}_i \;=\; \mathbf{A}_q^{i-1}\,\mathbf{\Theta}_{i-1} \tag{3.36}$$

Hence, by induction with (3.35),

$$\mathbf{\Theta}_i \;=\; \mathbf{A}_q^{\,i-1}\mathbf{R}_q, \qquad i = 1, \ldots, q\,. \tag{3.37}$$

Moment equality for the reduced system and the original system: Combining (3.34) and (3.37), for $k = 0, 1, \ldots, q-1$, the following can be written:

$$\mathbf{A}^k\mathbf{R} = \widetilde{\mathbf{X}}\,\mathbf{A}_q^{\,k}\,\mathbf{R}_q, \tag{3.38}$$

Multiplying both sides by $\mathbf{B}^\mathsf{T}$ gives

$$\mathbf{B}^\mathsf{T}\mathbf{A}^k\mathbf{R} = (\widetilde{\mathbf{X}}^\mathsf{T}\mathbf{B})^\mathsf{T}\,\mathbf{A}_q^k\,\mathbf{R}_q, \tag{3.39}$$

or,

$$\mathbf{M}_k \;=\; \mathbf{B}^\mathsf{T}\mathbf{A}^k\mathbf{R} = \mathbf{M}_{k,q} \;=\; (\widetilde{\mathbf{X}}^\mathsf{T}\mathbf{B})^\mathsf{T}\,\mathbf{A}_q^k\,\mathbf{R}_q \tag{3.40}$$

SPRIM matches $2q$ moments at an arbitrary expansion point s_0 [18]. This enhanced moment matching can be stated as follows. First, note that from equations (3.38) and (3.40) the following expressions can be obtained:

$$\mathbf{B}^\mathsf{T}\mathbf{A}^{j_1}\mathbf{C} = \mathbf{B}_q^\mathsf{T}\mathbf{A}_q^{j_1}\mathbf{C}_q, \qquad j_1 = 0, 1, \ldots, q, \tag{3.41}$$

$$\mathbf{A}^{j_2}\mathbf{R} = \mathbf{A}_q^{j_2}\mathbf{R}_q, \qquad j_2 = 0, 1, \ldots, q-1. \tag{3.42}$$

Combining (3.41) and (3.42) yields

$$\mathbf{B}^\mathsf{T}\mathbf{A}^{j_1+j_2}\mathbf{R} = \mathbf{B}_q^\mathsf{T}\mathbf{A}_q^{j_1+j_2}\mathbf{R}_q, \qquad j_1 + j_2 = 0, 1, \ldots, 2q-1,$$

This can be simplified and written as:

$$\mathbf{B}^\mathsf{T}\mathbf{A}^{j}\mathbf{R} = \mathbf{B}_q^\mathsf{T}\mathbf{A}_q^{j}\mathbf{R}_q, \qquad j = 0, 1, \ldots, 2q-1. \tag{3.43}$$

Comparing (3.43) with (3.40), it can be noted that the SPRIM algorithm matches twice the moments compared to PRIMA [16, 18].

3.6 Passivity Preservation with PRIMA and SPRIM Algorithms

The reduced models need to be passive for use in a time domain simulation. Physically, a non-passive model implies energy generation from a system on it's own without external stimulus. For a model to be *passive*, it must satisfy the three following conditions:

(i) **Condition 1: no poles in the open right half-plane** $\mathbf{H}_q(s)$ must be analytic for all complex s with $\Re(s) > 0$ (equivalently, it has no poles in the open right half-plane).

(ii) **Condition 2: conjugate symmetry**

$$\mathbf{H}_q(s^*) = \mathbf{H}_q^*(s) \tag{3.44}$$

the above condition implies that the transfer function $\mathbf{H}_q(s)$ satisfies the complex conjugation for real-valued physical systems (this condition is typically satisfied if the complex poles are included in the transfer function along with their complex conjugates in the model).

(iii) **Condition 3: positive matrix in the right half-plane** $\mathbf{H}_q(s)$ is a positive matrix, that is

$$\mathbf{z}^{\mathsf{T}}\Big(\mathbf{H}_q(s) + \mathbf{H}_q(s^*)^{\mathsf{T}}\Big)\mathbf{z}^* \ \geq 0 \tag{3.45}$$

for all complex s with $\Re(s) > 0$ and any complex vector $\mathbf{z}$.

To satisfy the passivity for both algorithms, both $\mathbf{X}$ and $\widetilde{\mathbf{X}}$ from PRIMA and SPRIM, respectively, are generated as *real projection matrices* thus ensuring the reduced matrices are *real*. This guarantees the second condition in (3.44) for passivity for the reduced system transfer function.
To satisfy the third requirement for passivity, the MNA stamps for inductor currents are negated when building the MNA matrices as shown earlier in Figure 3.2.
The modified stamping ensures the MNA matrices satisfy the property [14]:

$$(\mathbf{G} + \mathbf{G}^T) \succeq 0, \quad (\mathbf{C} + \mathbf{C}^T) \succeq 0$$

This fact is used to establish the condition in (3.45) for the reduced models. The complete passivity proof applicable to both PRIMA and SPRIM can be found at [14].

3.7 Accuracy and Computational Considerations

It is to be noted that, when the Krylov subspace is constructed by expanding at $s = 0$. The related matrices in (3.6) are obtained as: $\mathbf{A} = \mathbf{G}^{-1}\,\mathbf{C}, \mathbf{R} = \mathbf{G}^{-1}\,\mathbf{B}$. This allows the bulk of the computation to remain in real arithmetic. For example, the projector generation $\mathbf{X}$, the reduced matrices $\mathbf{G}_q, \mathbf{C}_q$, and the eigen-decomposition step $\text{eig}(\mathbf{G}_q^{-1}\mathbf{C}_q)$ in GPU-PRIMA are all computed in real arithmetic, saving roughly half the flops and storage that would otherwise be needed when using complex calculations. On the other hand, expanding only at $s = 0$ allows for accurate responses over a limited bandwidth of frequencies when using the reduced models. To improve accuracy

over a wider band, Krylov subspaces can be constructed at other complex expansion points $s_0 = \mathrm{j}\omega$ (rational Krylov / multipoint strategies) [15, 23]. In this case, the implementations require complex arithmetic, with most operations carried out in complex double precision using the corresponding `MAGMA` [32] and `CUDA` [30] libraries. Correspondingly, the related matrices are formulated as

$$\mathbf{A} = (\mathbf{G} + \mathrm{j}\omega\mathbf{C})^{-1}\,\mathbf{C}, \qquad \mathbf{R} = (\mathbf{G} + \mathrm{j}\omega\mathbf{C})^{-1}\,\mathbf{B}. \tag{3.46}$$

where ω is the frequency expansion point.
To obtain *real* and thus automatically passive reduced matrices when using complex expansion points, a real projector is formed by concatenating the real and imaginary parts of the complex Arnoldi basis as

$$\mathbf{X}_{\mathbb{R}} \;=\; \mathrm{orth}\Big(\big[\,\Re(\mathbf{X})\;\;\Im(\mathbf{X})\,\big]\Big), \tag{3.47}$$

and performing the congruence projection with $\mathbf{X}_{\mathbb{R}}$ so that $\mathbf{G}_q = \mathbf{X}_{\mathbb{R}}^{\mathsf{T}}\mathbf{G}\,\mathbf{X}_{\mathbb{R}}$, $\mathbf{C}_q = \mathbf{X}_{\mathbb{R}}^{\mathsf{T}}\mathbf{C}\,\mathbf{X}_{\mathbb{R}}$, and $\mathbf{B}_q = \mathbf{X}_{\mathbb{R}}^{\mathsf{T}}\mathbf{B}$ are all real. This ensures the reduced model remains passive by effecting a real congruence transformation.

3.8 Summary

In this chapter, a brief summary of model order reduction techniques based on Krylov subspaces is given. At the beginning, an outline of MNA formulation is provided. A brief review of implicit moment matching techniques based on Krylov subspace based techniques, utilizing Arnoldi (Block Arnoldi) methods is presented. To overcome their limitation, mainly loss of passivity in the reduced models, PRIMA and it's block-structure preserving alternative SPRIM as relevant to the contributions presented in this thesis, are reviewed.

Chapter 4

Proposed Development of GPU PRIMA

In Chapter 3, a brief review of the PRIMA algorithm was given. In this chapter, details of the proposed GPU-PRIMA is presented providing for extensive parallelization at both the task and thread levels, leading to speed-ups compared to multi-core CPU based implementations.

4.1 Preliminaries

Consider the modified nodal analysis (MNA) formulation of (3.1) and the system matrices as in (3.27). For a multi-port system of N_p ports, PRIMA is used to obtain a reduced model of size $q = N_p \cdot \mathtt{n}$. The target is to construct a block Krylov subspace using the Block Arnoldi algorithm and subsequently obtain the Krylov vector sequence $\mathbf{X} \in \mathbb{R}^{N_r \times q}$ spanning the subspace:

$$\mathcal{K}_q(\mathbf{A}, \mathbf{R}) = \text{span}\{\mathbf{R},\ \mathbf{AR},\ \mathbf{A}^2\mathbf{R},\ \dots\}. \tag{4.1}$$

The orthonormal basis $\mathbf{X}$ of this subspace yields the reduced matrices

$$\mathbf{G}_q = \mathbf{X}^\top \mathbf{G}\mathbf{X}, \qquad \mathbf{C}_q = \mathbf{X}^\top \mathbf{C}\mathbf{X}, \qquad \mathbf{B}_q = \mathbf{X}^\top \mathbf{B}, \qquad \mathbf{R}_q = \mathbf{X}^\top \mathbf{R}. \tag{4.2}$$

In the proposed GPU implementation, $\mathbf{X}$ is formed using the block Arnoldi algorithm, and $\mathbf{G}_q, \mathbf{C}_q$ are obtained using the split congruence transformation as above. Subsequently, $\mathbf{G}_q^{-1}\mathbf{C}_q$, is diagonalized, and the port transfer function matrix $\mathbf{H}(\mathrm{j}\omega)$ is

assembled in batches from the computed poles and residues for obtaining the frequency response at the points of interest.

4.2 Proposed Implementation Pipeline Overview

The proposed implementation uses the host side (CPU) to control the flow of the algorithm while off-loading data heavy processing to GPU for parallelization. The CPU is used for computations that are not data-heavy, non-iterative or for computations that are incurred only once. For example the CPU is used to do a single LU factorization of the original system prior to launching the iterative Krylov solvers. All other iterative and compute heavy processes are centered around utilizing the GPU. A simple overview of the application pipeline is given below for clarity:

1. *Sparse CPU LU pre-processing:* Factorize $\mathbf{G}$ and compute $\mathbf{A} = \mathbf{G}^{-1}\mathbf{C}$, $\mathbf{R} = \mathbf{G}^{-1}\mathbf{B}$ using SuiteSparse [27]. This step is computed once for a single expansion point $s = 0$ (or at each subsequent expansion point).
2. *GPU Block Arnoldi:* Generate $\mathbf{X}$ via BLAS–3 and tall-skinny QR utilizing MAGMA libraries. The first bulk of the computation time dominates here since it is an iterative process with large matrix-matrix and matrix-vector computations.
3. *Projection (GPU):* Form $\mathbf{G}_q, \mathbf{C}_q$ on the GPU using MAGMA GEMMs.
4. *Eigen-decomposition (GPU):* Compute eigen-pairs of $\mathbf{G}_q^{-1}\mathbf{C}_q$. An iterative QR based solver *magma_zgeev* is used for diagonalizing the reduced system matrix to it's Schur form.
5. *Frequency sweep (GPU):* $\mathbf{H}(\mathrm{j}\omega)$ is assembled with a custom CUDA kernel for a particular band of frequencies.

The algorithmic flowchart for the GPU-based task-parallelized PRIMA algorithm in Fig. 4.1.

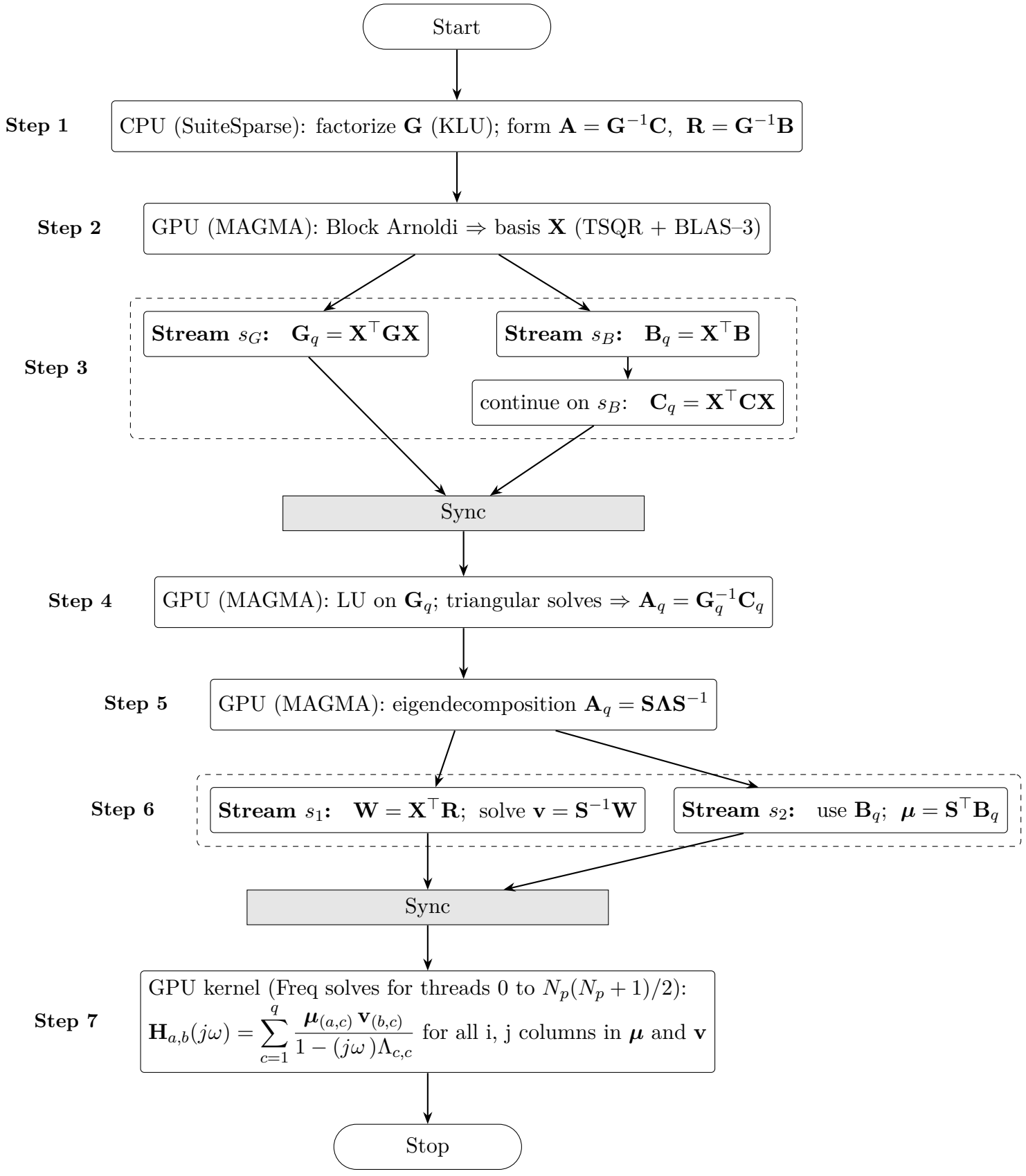


Figure 4.1: Algorithmic flowchart for GPU–PRIMA.

4.3 Step 1: CPU based Matrix Pre-processing and Computation of A and R Matrices

The input Matrices $\mathbf{G}, \mathbf{C}, \mathbf{B}$ of the RLC circuit MNA formulation are read from text files that are saved in COO (Coordinate format). Each row of the matrix data structure contains the row index, column index and the corresponding numerical value at that index. Due to the scarcity of readily available, direct sparse LU-decomposition based solvers for GPU, the first step uses CPU-based libraries. The CHOLMOD library is used to convert $\mathbf{G}$ to CSC (Compressed Sparse Column) matrix storage format and the KLU library [27] is used for an initial symbolic factorization to map out the fill-ins, independent columns and sparsity pattern, and to handle row or column permutations prior to any factorization. Once the symbolic analysis is done, the indices for fill-ins and numerical calculations are used to efficiently compute the L and U factors for the system matrices. The KLU factors are then used to apply multi-RHS sparse solvers to obtain

$$\mathbf{A} \leftarrow \mathbf{G}^{-1}\mathbf{C}, \qquad \mathbf{R} \leftarrow \mathbf{G}^{-1}\mathbf{B}. \tag{4.3}$$

At the end of this phase, $\mathbf{A} \in \mathbb{R}^{N_r \times N_r}$ and $\mathbf{R} \in \mathbb{R}^{N_r \times N_p}$ are stored as dense, column-major arrays compatible with MAGMA/LAPACK. Here N_r denotes the dimension of the square system matrices of the original MNA, and N_p denotes the number of port voltage sources.

4.4 Step 2: GPU-based Block Arnoldi and Projection Basis Construction

The size of each block of Arnoldi vectors equals the port count, Np. Let $n = q/Np$ be the total number of Arnoldi blocks generated. The high throughput and massively parallel cores of the GPU are used to speed up the Arnoldi workflow that is dominated with matrix-matrix and matrix-vector products. A pseudocode of the proposed block Arnoldi process is given in Algorithm 4.

Algorithm 4 GPU Block Arnoldi (block size N_p) for PRIMA Basis Construction

Inputs: $\boldsymbol{A} = \boldsymbol{G}^{-1}\boldsymbol{C} \in \mathbb{R}^{N_r \times N_r}$, $\boldsymbol{R} = \boldsymbol{G}^{-1}\boldsymbol{B} \in \mathbb{R}^{N_r \times N_p}$, number of block moments matched $= n$ and number of Krylov subspace columns $q = n \times Np$.

Outputs: Orthonormal basis $\boldsymbol{X} = [\boldsymbol{X}_0, \ldots, \boldsymbol{X}_{n-1}] \in \mathbb{R}^{N_r \times q}$ forming the sequence of Block Krylov vectors.

Step 1: Initialization (tall–skinny QR)

Factorize the starting block: $\boldsymbol{R} = \boldsymbol{X}_0 \boldsymbol{R}_0$ (TSQR using Magma on GPU).

Initialize the global basis: $\boldsymbol{X} \leftarrow [\boldsymbol{X}_0]$.

Step 2: Arnoldi loop (for $k = 1, \ldots, n-1$)

(a) Krylov update

$\boldsymbol{W}_k \leftarrow -\boldsymbol{A}\boldsymbol{X}_{k-1}$ (BLAS–3 GEMM on GPU)

(b) Block Modified Gram–Schmidt orthogonalization

for $j = 1, \ldots, k$ **do**

$\boldsymbol{H}_{k,j} \leftarrow \boldsymbol{X}_{k-j}^{\top}\boldsymbol{W}_k$ (BLAS–3 GEMM on GPU)

$\boldsymbol{W}_k \leftarrow \boldsymbol{W}_k - \boldsymbol{X}_{k-j}\boldsymbol{H}_{k,j}$ (BLAS–3 GEMM on GPU)

end for

(c) Re-orthogonalize / normalize

$\boldsymbol{W}_k = \boldsymbol{X}_k \boldsymbol{R}_k$ (TSQR using Magma on GPU)

(d) Append block to basis

$\boldsymbol{X} \leftarrow [\boldsymbol{X} \mid \boldsymbol{X}_k]$

Step 3: Outputs

Orthonormal basis $\boldsymbol{X} \in \mathbb{R}^{N_r \times q}$.

The implementation utilizes BLAS–3 (`dgemm`) for forming the base block of vectors and utilizes `magma_dgegqr_gpu` for tall skinny QR (running a modified gram-schmidt orthogonalization scheme) factorization of the $\boldsymbol{W}_k$ generated blocks, maximizing GPU throughput. The data packing into $\mathbf{X}$ is done by a CUDA kernel and basic pointer arithmetic.

4.5 Step 3: Projection to The Reduced Model (GPU)

With $\mathbf{X}$ saved on the GPU by default, the reduced matrices are formed using *separate CUDA streams* to maximize GPU utilization and overlap. In stream s_G, the reduced matrix G_q is evaluated as:

$$\mathbf{G}_q \;=\; \mathbf{X}^\top \mathbf{G}\, \mathbf{X}, \tag{4.4}$$

while concurrently in stream s_B, B_q is computed:

$$\mathbf{B}_q \;=\; \mathbf{X}^\top \mathbf{B}. \tag{4.5}$$

Once the computation of $\mathbf{G}_q$ is finished, the same stream s_G is used to compute

$$\mathbf{C}_q \;=\; \mathbf{X}^\top \mathbf{C}\, \mathbf{X}. \tag{4.6}$$

CUDA programming constructs called CUDA events synchronize the streams before any subsequent steps that may have data dependencies from the previous steps (e.g., using $\mathbf{G}_q$ and $\mathbf{C}_q$).

4.6 Step 4: Computation of Reduced System Matrix

Next, to obtain $\mathbf{G}_q^{-1}\mathbf{C}_q$, $\mathbf{G}_q$ is first factorized using Magma's dense LU kernel and triangular solvers are applied on $\mathbf{C}_q$ to obtain:

$$\mathbf{G}_q = \mathbf{L}\mathbf{U}, \qquad \mathbf{A}_q = \mathbf{U}\backslash\Big(\mathbf{L}\backslash\mathbf{C}_q\Big). \tag{4.7}$$

The routines used for this are `magma_dgetrf_expert_gpu_work` and `magma_dgetrs_gpu`. The congruence transformation $\mathbf{X}$, the MNA matrices $\mathbf{G}$, $\mathbf{C}$, $\mathbf{B}$ used in the projection steps are real, double-precision matrices. This leads to real matrices that need to be cast to complex form. The matrix $\mathbf{A}_q \in \mathbb{R}^{q\times q}$ is therefore cast to complex form and passed to a complex eigen-solver.

4.7 Step 5: Eigen-decomposition (GPU)

In this step, eigen-decomposition of the system is then performed as:

$$\mathbf{A}_q = \mathbf{S}\mathbf{\Lambda}\mathbf{S}^{-1}, \qquad \mathbf{\Lambda} = \mathrm{diag}(w_1, \ldots, w_q) \in \mathbb{C}^{q\times q}, \tag{4.8}$$

using a complex, non-Hermitian eigensolver (from the Magma library: `magma_zgeev`). Here, $\mathbf{\Lambda}$ is the diagonal eigenvalue matrix and $\mathbf{S}$ is the matrix of right eigen-vectors returned by the routine. Once the reduced system is eigen-decomposed and diagonalized, the resulting eigenvectors and eigenvalues are used to calculate the residue factors to construct the transfer function matrix.

4.8 Step 6: Computing the Residue Factors (GPU)

The residue factors are generally dense and benefit from GPU accelerated parallelization. Two streams are utilized to overlap the matrix multiply phases to compute $\mathbf{v}$ and μ. This phase utilizes separate streams *s1* and *s2* for the matrix-matrix multiplications using BLAS–3 `dgemm` routines in conjunction with `magma_dgetrf_expert_gpu_work` and `magma_dgetrs_gpu` for forming the inverse of the right eigenvectors (on stream *s1*). Using the eigen-decomposition of (4.8), the residue factors are calculated as:

$$\mathbf{W} = \mathbf{X}^\top\mathbf{R} \in \mathbb{R}^{q\times N_p}, \qquad \mathbf{B}_q = \mathbf{X}^\top\mathbf{B} \in \mathbb{R}^{q\times N_p}. \tag{4.9}$$

$$\mathbf{v} = \mathbf{S}^{-1}\mathbf{W} \in \mathbb{C}^{q\times N_p}, \qquad \boldsymbol{\mu} = \mathbf{S}^\top\mathbf{B}_q \in \mathbb{C}^{q\times N_p}. \tag{4.10}$$

4.9 Step 7: Formulation of Port Transfer function in GPU

For each angular frequency ω, the port transfer function is evaluated using the above computed residues as

$$\mathbf{H}_{a,b}(\mathrm{j}\omega) = \sum_{c=1}^{q} \frac{\boldsymbol{\mu}_{(a,c)}\ \mathbf{v}_{(b,c)}}{1-(j\omega\,)\Lambda_{c,c}}, \qquad \text{with } a, b = 1, 2, 3, \ldots, N_p \text{ and } i \neq j \tag{4.11}$$

Here $\mathbf{v}_{(a,:)}$ is the a-th column of $\mathbf{v}$ and $\boldsymbol{\mu}_{(b,:)}$ is the b-th column of $\boldsymbol{\mu}$ (the ":" operator indicates all elements within the column). The resulting transfer function matrix is

always symmetric, enabling us to halve the compute work and storage required. For this purpose, a custom CUDA kernel computes only the upper triangle of **H** in packed form of length $N_p(N_p + 1)/2$ using a one-dimensional grid structure. The implementation pre-calculates the upper triangle packed indices to (a, b) pairs using two integer arrays. This allows each thread to have it's own unique (a, b) mappings, despite being declared as a single dimensional grid of threads. An example of a pre-calculated 2-array index mapping is given for a 5x5 matrix in Figure 4.2 and Figure 4.3.
For each frequency point, a single entry in the upper triangle is obtained by accumulating contributions over the q rows of the residue and eigenvalue matrices. The host (CPU) manages the frequency sweeps: for every frequency ω, a batch of **H** transfer function values is computed, and asynchronous copy kernels are launched to write the results back to an output file, overlapping the data transfers with the ongoing computations.

0,0	0,1	0,2	0,3	0,4
	1,1	1,2	1,3	1,4
		2,2	2,3	2,4
			3,3	3,4
				4,4

Figure 4.2: An example of a 5×5 upper-triangular transfer function matrix.

a	0	0	0	0	0	1	1	1	1	2	2	2	3	3	4
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
b	0	1	2	3	4	1	2	3	4	2	3	4	3	4	4
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
threads	0,0	0,1	0,2	0,3	0,4	1,1	1,2	1,3	1,4	2,2	2,3	2,4	3,3	3,4	4,4
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Figure 4.3: Mapping the transfer function matrix to linearized thread indices.

4.10 Discussion

4.10.1 Data layout, precision, and use of libraries

All dense arrays are considered in column-major order, and the workflow is designed to minimize host–device transfers by keeping $\mathbf{X}$, $\mathbf{G}$, $\mathbf{C}$, and the reduced quantities on the GPU during projection and eigen-decomposition phases. Double precision is used throughout (real or complex) to maintain numerical stability in orthogonalization and dense factorization. The following libraries are used during the implementation:

- **SuiteSparse (CPU):** CHOLMOD for sparse I/O/format conversion; KLU for symbolic/numeric factorization of $\mathbf{G}$ and subsequent sparse solves to form $\mathbf{A}, \mathbf{R}$ (Step 1).
- **MAGMA (GPU):** BLAS–3 (`dgemm`, used in Step 2, Step 3 and Step 6), tall-skinny QR (`dgegqr_gpu`), LU/solve for $\mathbf{G}_q^{-1}$ (used in Step 4 and Step 6), and complex eigensolver (`zgeev`, used in Step 5).
- **CUDA (GPU):** Custom kernels for packing the global basis $\mathbf{X}$, custom kernels for real→complex casting, and assembling $\mathbf{H}(\mathrm{j}\omega)$ (used in Steps 1-3 and Step 7).

4.10.2 Workload and performance overview

The implementation reports timings for the sparse factorization phases, Arnoldi/QR phases, $\mathbf{G}/\mathbf{C}$ reduction phase, dense inversion of $\mathbf{G}_q$, eigen-decomposition in GPU, and the final $\mathbf{H}$-assembly. The computational workload can be divided into the following components:

- *Sparse preprocessing (CPU):* This stage is dominated by the KLU factorization [27] and the subsequent multi-RHS triangular solves, so its overall cost is primarily determined by the sparsity pattern of the matrices and the chosen ordering (Step 1).
- *Block Arnoldi (GPU):* At each Arnoldi step, the dominant operation is a single $\mathcal{O}(N_r^2 N_p)$ matrix–matrix product for updating the Krylov subspace, accompanied by k projection products of cost $\mathcal{O}(N_r N_p^2)$, and a tall–skinny QR factorization of an $N_r \times N_p$ matrix (Step 2).
- *Projection (GPU):* The projection phase requires two GEMMs per reduced matrix, and the total work in this stage scales as $\mathcal{O}(N_r^2)$ (Step 3).
- *Dense stages (GPU/CPU):* In the dense part of the algorithm, the LU factorization of the reduced $q \times q$ matrix $\mathbf{G}_q$ has a computational cost of $\mathcal{O}(q^3)$, and the subsequent eigensolve similarly incurs a cost of $\mathcal{O}(q^3)$ (Steps 4–5).

- *Frequency sweep (GPU):* For each frequency point, the total work amounts to $\mathcal{O}\Big(q\,N_p(N_p+1)/2\Big)$ complex dot-product multiplies and accumulates, and these operations are parallelized both across the individual matrix entries and across different frequencies (Step 7).
- Only right eigenvectors are formed in this implementation, since the residues are constructed directly from the eigenvector matrix $\mathbf{S}$ and its inverse $\mathbf{S}^{-1}$, as required by the PRIMA algorithm (Steps 5–6).
- The $\mathbf{H}$ kernel only computes the packed upper triangle of the transfer function matrix, exploiting the fact that, in linear RLC circuits, the matrix is symmetric and the explicit computation of the lower triangle is therefore unnecessary (Step 7).
- For very large frequency batches, a pre-allocated host buffer and multiple CUDA streams are employed so that computation and PCIe transfers of the transfer function matrix can be effectively overlapped (Step 7).
- The threads of the custom kernels load array and matrix data into shared memory locations in order to reduce accesses to global memory, following the standard design pattern used in most high-performance GPU kernels to further accelerate data movement (Step 7).

4.11 Summary

In this chapter, details of the proposed GPU-accelerated PRIMA algorithm is given. The algorithm mainly involves the steps of: (i) constructing $\mathbf{A} = \mathbf{G}^{-1}\mathbf{C}$ and $\mathbf{R} = \mathbf{G}^{-1}\mathbf{B}$ on the CPU using sparse KLU solvers, (ii) generating a block Arnoldi basis $\mathbf{X}$ entirely on the GPU, (iii) projecting the system matrices to reduced matrices forming $\mathbf{G}_q$ *and* $\mathbf{C}_q$, (iv) forming the new system matrix $\mathbf{G}_q^{-1}\mathbf{C}_q$, (v) computing it's eigen-decomposition to extract the dominant poles and eigenvectors of the reduced system, (vi) computing the residue factors $\mathbf{v}$ and μ and finally, (vii) assembling $\mathbf{H}(\mathrm{j}\omega)$ for a large frequency range via a custom kernel in batches. The proposed implementation uses BLAS level 3 intensity and minimal host–device data transfer traffic, yielding a practical, scalable GPU implementation suitable for multi-port interconnect models.

Chapter 5

Numerical Results for the Proposed GPU-PRIMA

In this chapter, numerical results validating the proposed GPU-PRIMA algorithm are presented.

5.1 Experimental Setup

Hardware Environment All experiments were run with double precision (64-bit floating-point arithmetic). The examples were executed on the CPU/GPU servers hosted by DRAC (Digital Research Alliance of Canada). The compute cluster used for executing our implementations was the Narval cluster hosted by DRAC, which contains NVIDIA A100 GPUs featuring both Tensor and CUDA cores, 40 GB of RAM, 40 MB of L2 cache, 6912 CUDA cores, and 1.6 TB/s of graphics memory bandwidth. The CPU is a third-generation AMD EPYC 7413 with 2.65 GHz base clock speed, 128 MB of L3 cache, and 24 cores (48 threads in total). These processors support the AVX2 instruction set and have a maximum boosted clock speed of 3.6 GHz.
The proposed reduction algorithms are executed for different circuit configurations on both CPU and GPU platforms and the performance metrics are compared. Two validating circuits with varying MNA sizes and port counts are considered: $N_p \in \{40, 128\}$. Circuit configuration corresponding to a power grid mesh with RLC components is used in these experiments. An illustration of the circuit configuration with $Np = 10$ ports is given in Figure 5.1. The following parameters from both the reduced and original models are tested and compared:

- **Frequency response:** The frequency responses of the original MNA and the reduced models are plotted and compared over a bandwidth of 0-1.3 GHZ.

- **Performance:** A comparison of run-times using both the reduced and original models for obtaining the frequency-domain responses is given. In addition, CPU/GPU times for various stages while using the proposed GPU-PRIMA are recorded and compared:
  (i) Sparse CPU pre-processing,
  (ii) Block Arnoldi,
  (iii) Projection to reduced matrices,
  (iv) Dense reduced matrix inverses,
  (v) Eigen-decomposition times,
  (vi) Poles, residues calculation, and
  (vii) Computing the transfer function matrix.

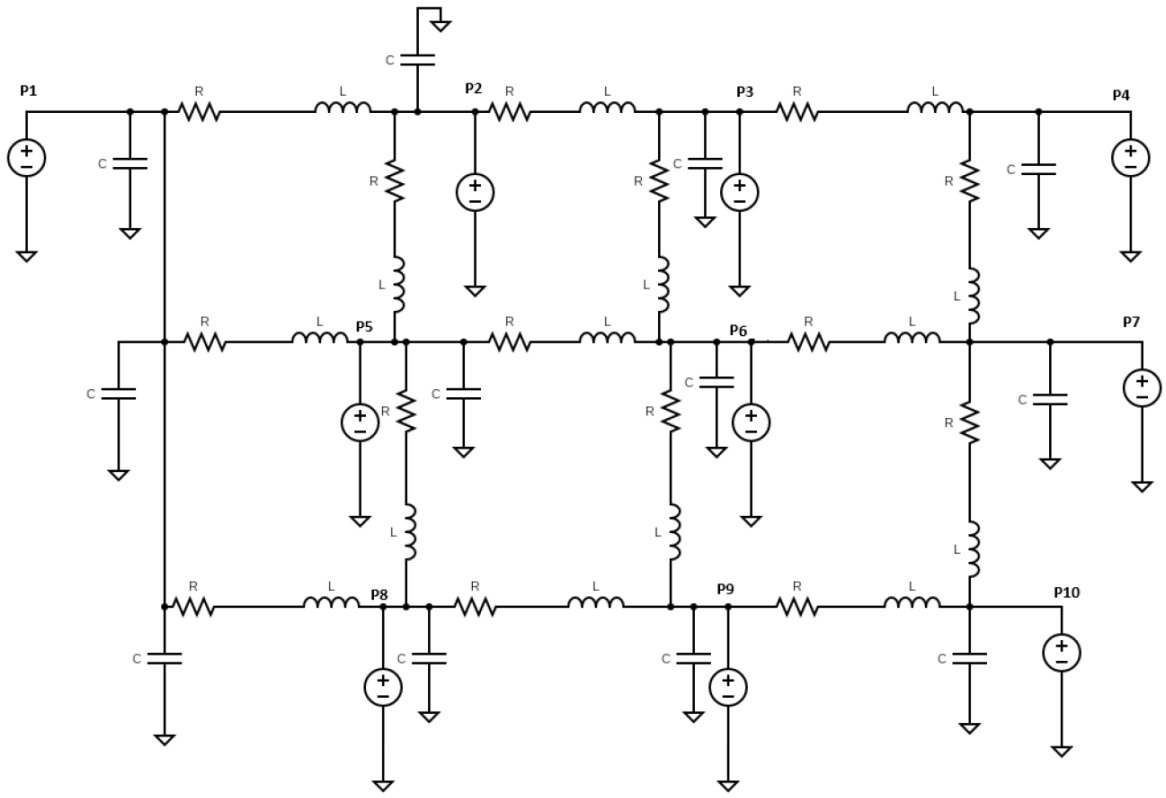


Figure 5.1: Illustration of the RLC mesh circuit configuration for an example case of 10 ports

5.2 Example 1: Multi-port Circuit with $N_p = 40$

In this example, a 40-port network with a similar RLC network configuration as in Figure 5.1 is considered (a longer mesh depth in both the x- and y-directions is used).

The resulting MNA matrix is of size 11683 × 11683 with 7022 internal nodes. The circuit frequency response is first solved for the original system and then compared with the response of the proposed reduced models (implemented on single-core CPU, multi-core CPU and GPU platforms). A frequency sweep is performed using 2000 frequency points over a bandwidth up to 1.3 GHZ. For this example, 25 moments per port ($q = 25N_p$) are matched, yielding reduced models of size 1000 × 1000. Sample comparisons of frequency responses between the original and GPU-PRIMA reduced models are provided in Figures 5.2–5.4.

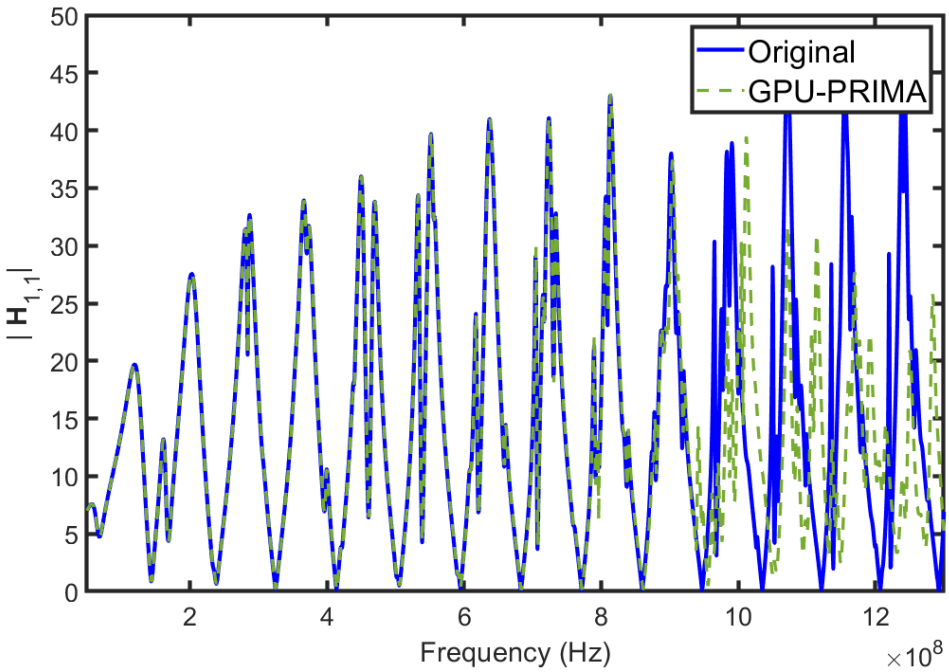


Figure 5.2: Comparison of $|\mathbf{H}_{1,1}(j\omega)|$ from original system and GPU-PRIMA reduced models (40-port network)

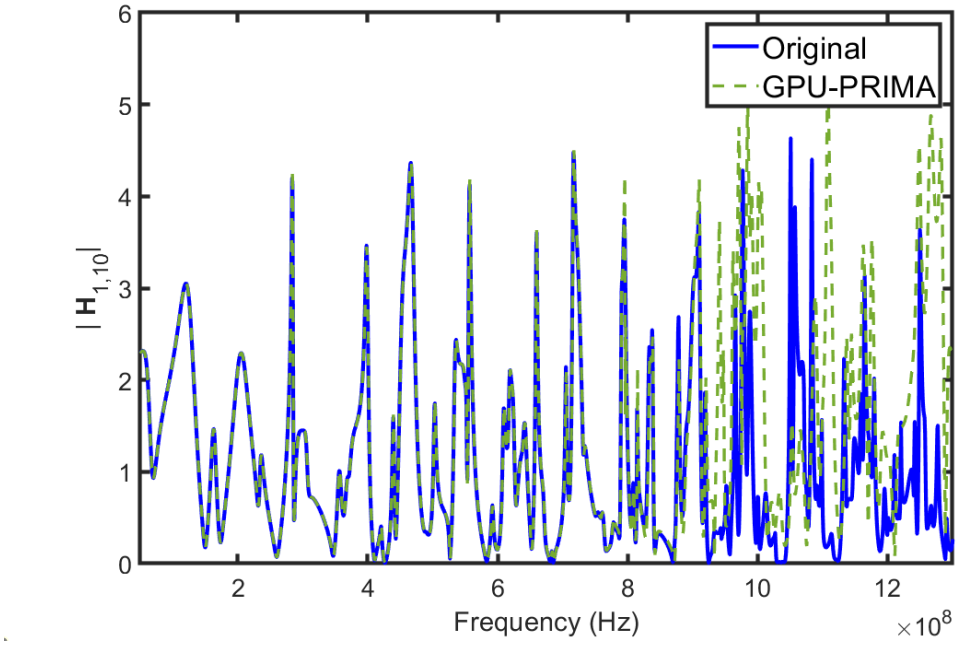


Figure 5.3: Comparison of $|\mathbf{H}_{1,10}(j\omega)|$ from original system and GPU-PRIMA reduced models (40-port network)

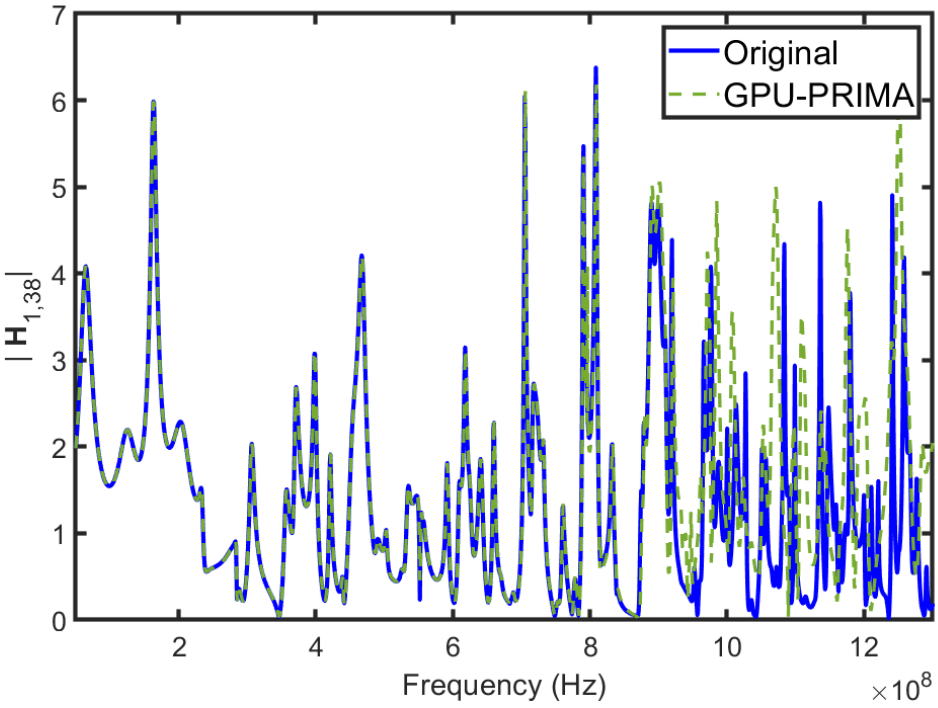


Figure 5.4: Comparison of $|\mathbf{H}_{1,38}(j\omega)|$ from original system and GPU-PRIMA reduced models (40-port network)

As can be seen, the responses from the proposed GPU-PRIMA (in the dotted lines) match reasonably well with the original system response (in solid line). Responses from the single-core and multi-core CPU PRIMA are not plotted as the responses are identical to that of GPU-PRIMA responses. The accuracy of the reduced models will increase if we increase the number of moments matched. A comparison of computational times is provided in Table 5.1.

Table 5.1: Comparison of Computational Performance Times for PRIMA Implementations (Example-1, 40-Port Circuit)

Step	**Single-core PRIMA (s)**	**Multi-core PRIMA (s)**	**Proposed GPU-PRIMA (s)**
Computation of $\mathbf{R}$ (in CPU) - CPU-PRIMA [Step 1]	0.012	0.01	–
Computation of $\mathbf{A}$ and $\mathbf{R}$ (in CPU) - GPU-PRIMA [Step 1]	–	–	1.11
Block Arnoldi [Step 2]	5.81	2.00	0.60
$\mathbf{G}, \mathbf{C}$ and $\mathbf{B}$ reductions [Step 3]	18.94	1.91	0.22
Total time taken for obtaining a reduced state space system	*24.76*	*3.92*	*1.93*
Speed-up (while generating the reduced state space models)	–	**6.32**	**12.83**
Dense $\mathbf{G}_q$ inversion [Step 4]	0.66	0.61	0.07
Eigen-decomposition of $\mathbf{G}_q^{-1}\mathbf{C}_q$ [Step 5]	4.77	4.31	2.14
Poles, residues calculation [Step 6]	4.83	0.42	0.36
Frequency-sweep $\mathbf{H}$ matrix computation [Step 7]	1.60	0.31	
Total time taken including pole-residue computation and obtaining frequency response	*36.62*	*9.57*	*4.50*
Overall Speed-up (including CPU costs for frequency response generation)	–	**3.83**	**8.14**

5.2.1 Discussion

In this example (with $N_p = 40$ and $N_r = 11683$), the reduced model produces the responses with reasonably good accuracy, comparable with that from the original network. As can be seen from the sample frequency responses in Figures 5.2–5.4, the magnitude responses $|\mathbf{H}_{1,1}(j\omega)|$, $|\mathbf{H}_{1,10}(j\omega)|$, and $|\mathbf{H}_{1,38}(j\omega)|$ from GPU-PRIMA closely track the original circuit response over the 0–800 MHz range.

With $q = 25N_p$, PRIMA yields a 1000×1000 reduced model. This reduced order is therefore less than 10% of the original state dimension while still achieving the frequency-domain accuracies as shown in Figures 5.2–5.4. The bandwidth of accuracy can be further increased by increasing the number of moments matched, but the tradeoff is an increase in the size of the reduced models.

It is to be noted that, while passivity is preserved, sparsity is lost in the reduced model. The computational timings in Table 5.1 highlight the advantages as well as the dominant bottlenecks in the proposed algorithm. The proposed GPU-PRIMA Algorithm achieves speedups of approximately 8.14 compared to the single-core CPU-PRIMA implementation and 2.13 times compared to the multi-core CPU-PRIMA implementation (4.50 s vs. 36.62 s and 9.57 s, respectively). The dominant costs of GPU-PRIMA are the dense matrix-matrix multiplications of the block Arnoldi step, the $\mathbf{G}$, $\mathbf{C}$, and $\mathbf{B}$ projections, and the eigen-decomposition and frequency response calculation steps, which are data-heavy, dense operations. These stages map well to the GPU, where massively parallel dense linear algebra kernels substantially reduce the compute time compared with the CPU implementations. For an original system of size $N_r = 11683$ with a 1000×1000 reduced model, these timings and speedups are consistent with the expected cubic cost of dense matrix–matrix multiplications and the high throughput of the A100 GPU.

5.3 Example 2: Multi-port RLC Circuit with $N_p = 128$

In this example, a 128-port RLC network with a larger spatial mesh structure is considered. The resulting MNA matrices are of size $44,353 \times 44,353$, with $26,632$ internal nodes and $17,721$ inductor current variables. As before, the circuit frequency response is first computed for the original system and then compared against the responses of the reduced models obtained by PRIMA on single-core CPU, multi-core CPU, and GPU platforms. A frequency sweep of 2000 points is performed over a

frequency band up to 1.3 GHz. While constructing the reduced model, 30 moments are matched ($q = 30N_p$), resulting in GPU-PRIMA reduced models of size 3840×3840. Representative comparisons of frequency responses between the original and GPU-PRIMA reduced models are shown in Figures 5.5–5.7.

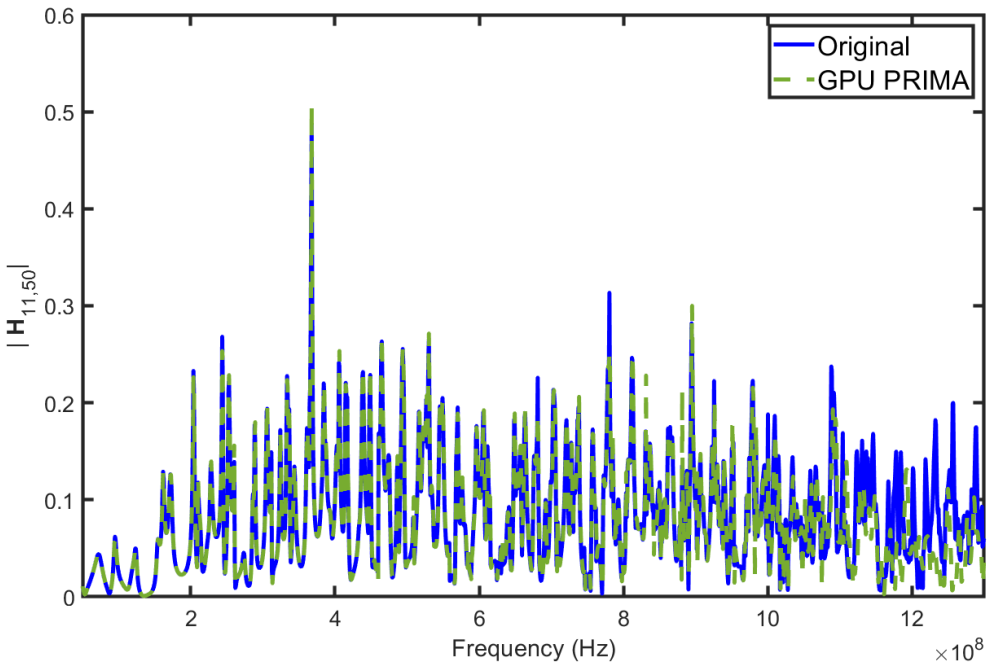


Figure 5.5: Comparison of $|\mathbf{H}_{11,50}(j\omega)|$ from original system and GPU-PRIMA reduced models (128-port network)

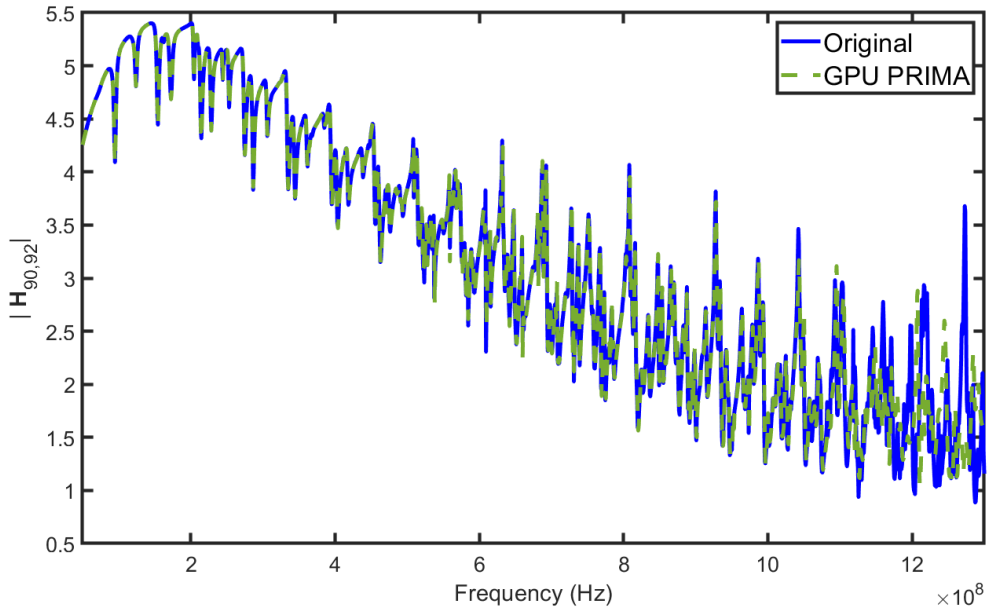


Figure 5.6: Comparison of $|\mathbf{H}_{90,92}(j\omega)|$ from original system and GPU-PRIMA reduced models (128-port network)

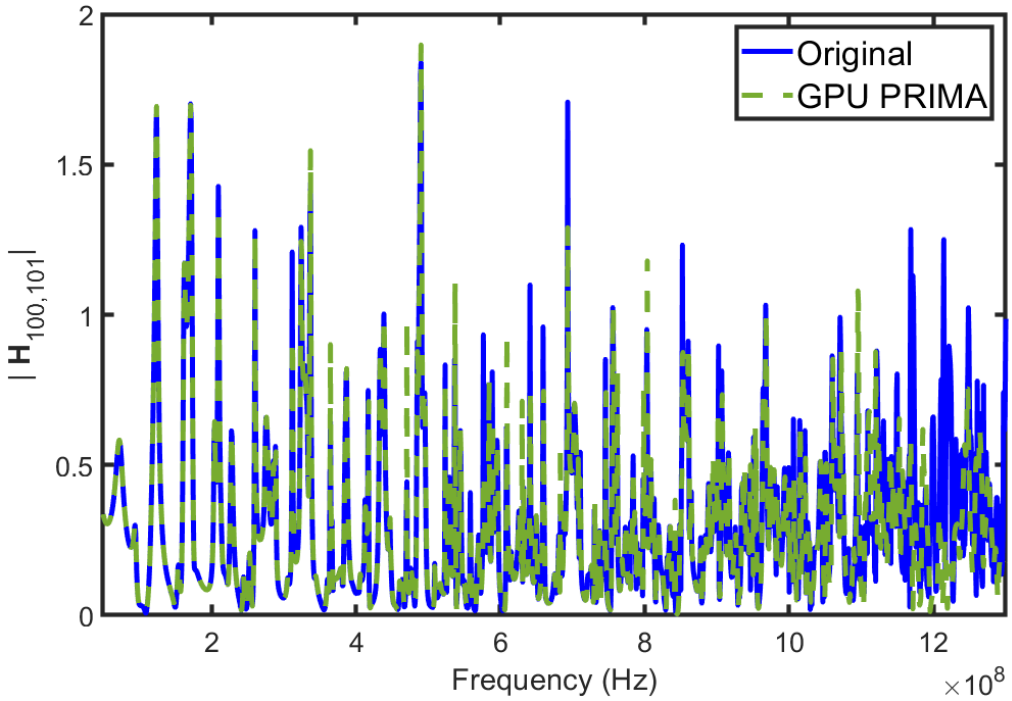


Figure 5.7: Comparison of $|\mathbf{H}_{100,101}(j\omega)|$ from original system and GPU-PRIMA reduced models (128-port network)

From Figures 5.5–5.7 it can be seen that, the responses from the proposed GPU-PRIMA reduced models (dashed green) matches reasonably well with the original system responses (solid blue) over most of the frequency bandwidth. A computational performance comparison for all three implementations is given in Table 5.2.

Table 5.2: Comparison of Computational Performance Times for PRIMA Implementations (Example-2, 128-Port Circuit)

Step	**Single-core PRIMA (s)**	**Multi-core PRIMA (s)**	**Proposed GPU-PRIMA (s)**
Computation of **R** (in CPU) - CPU-PRIMA [Step 1]	0.11	0.09	–
Computation of **A** and **R** (in CPU) - GPU-PRIMA [Step 1]	–	–	18.16
Block Arnoldi [Step 2]	96.56	28.20	6.27
G, **C** and **B** matrix reductions [Step 3]	754.06	58.66	4.86
Total time taken for obtaining a reduced state space system	*850.73*	*86.95*	*29.29*
Speed-up (while generating the reduced state space system)	–	**9.78**	**29.05**
Dense **G** inversion [Step 4]	15.85	1.99	0.71
Eigen-decomposition of $\mathbf{G}_q^{-1}\mathbf{C}_q$	612.87	407.15	71.18
Poles, residues calculation [Step 6]	135.82	9.89	1.38
Frequency-sweep **H** matrix computation [Step 7]	403.36	30.95	
Total time taken including pole-residue computation and obtaining frequency response	*2018.63*	*536.93*	*102.56*
Overall Speed-up (including compute costs for frequency response generation)	–	**3.76**	**19.68**

5.3.1 Discussion

For the 128-port circuit, the sample responses in Figures 5.5–5.7 demonstrate that GPU-PRIMA produces reasonably well matching responses with the original system up to the 800 MHz band. Using $q = 30N_p$ for $N_p = 128$ produces a 3840×3840 reduced model, which is only a small fraction of the original state dimension, yet still tracks the original frequency responses with reasonable accuracy. The timing breakdown in Table 5.2 shows that the dominant costs in the single-core and multi-core CPU implementations are the block Arnoldi step, the $\mathbf{G}$, $\mathbf{C}$, $\mathbf{B}$ reductions, followed by the eigenvalue decomposition of $\mathbf{G}_q^{-1}\mathbf{C}_q$ and the frequency response calculation from generating and using the pole-residue model. On the GPU, the dense, data-intensive stages—block Arnoldi, $\mathbf{G}$, $\mathbf{C}$ and $\mathbf{B}$ projections, and dense $\mathbf{G}_q$ inversion—are accelerated substantially, reducing the reduced state-space generation time from 850.73 s (single-core) and 86.95 s (multi-core) down to 29.29 s on the GPU. This corresponds to speedups of approximately 9.78 times for the multi-core CPU implementation and 29.05 times for the GPU-PRIMA implementation during state-space generation. When the pole–residue computation and full $\mathbf{H}$-matrix evaluation are included, the overall run time is reduced from 2018.63 s (single-core) and 536.93 s (multi-core) to 102.56 s on the GPU, yielding effective speedups of about 19.68 times and 5.23 times, for GPU-PRIMA and multi-core CPU PRIMA respectively. These results confirm that the proposed GPU-PRIMA framework provides substantial acceleration in the presence of large circuits.

5.4 Summary

In this chapter, numerical examples are presented to demonstrate the efficiency and accuracy of the proposed GPU-PRIMA. Two examples are presented (with MNA sizes of $11,683 \times 11,683$ and $44,353 \times 44,353$). It is demonstrated that the proposed GPU-PRIMA produces passive reduced models and provides significant speed-ups compared to single-core and multi-core based PRIMA implementations by exploiting the parallel computational framework of modern GPUs with Tensor Core based platforms.

Chapter 6

Proposed Development of GPU SPRIM

In Chapter 4, a parallel framework for implementing the PRIMA algorithm, suitable for use in GPU-based platforms was presented. In this chapter, a parallel framework for the SPRIM algorithm in GPU platforms is presented.

6.1 Preliminaries

Consider the MNA formulation of an RLC circuit and it's system matrices $\mathbf{G}$ and $\mathbf{C}$ in equation (3.1) and the matrices $\mathbf{A}$ and $\mathbf{R}$ matrices in (3.24). For the proposed GPU-based SPRIM algorithm, the first few steps are similar to that of the proposed GPU-PRIMA algorithm presented in the previous chapter. First, a Krylov subspace with block size N_p (equaling the port count) and with total block width of $q = N_p \cdot \mathtt{n}$ is constructed, producing an orthonormal basis $\mathbf{X} \in \mathbb{R}^{N_r \times q}$. SPRIM differs from PRIMA in the sense that the *structure-preserving* projection in SPRIM requires partitioning the basis to retain the block structure of the $\mathbf{G}$ and $\mathbf{C}$ matrices (while also retaining the passivity requirements for the reduced model). The algorithm involves the following steps: (i) compute the projection basis $\mathbf{X}$ by using Block Arnoldi; (ii) assemble a block-diagonal projector $\widetilde{\mathbf{X}}$ that partitions the *output* $\mathbf{X}$ according to *internal* blocks of $\mathbf{G}$, (iii) form $\mathbf{G}_\mathrm{q}, \mathbf{C}_\mathrm{q}, \mathbf{B}_\mathrm{q}$; and (iv) evaluate $\mathbf{H}(\mathrm{j}\omega)$ via LU decomposition and forward-backward solution steps. A pseudocode of the SPRIM algorithm is given as Algorithm 3 in Chapter 3.

6.2 Proposed implementation pipeline overview

In the proposed GPU-SPRIM implementation, first, the sparse system matrices are constructe in the CPU. Next, all matrix-matrix and matrix-vector multiply and accumulate processes are executed in the GPU. Separate CUDA streams schedule independent operations concurrently to exploit parallelism. Details of the proposed GPU-based algorithm are given in the subsequent steps, and an algorithmic flowchart for the proposed GPU-based SPRIM implementation is given in Figure 6.1, as well as outlined below:

1. *Sparse CPU LU preprocessing:* Factorize $\mathbf{G}$ with SuiteSparse/KLU; form the system matrices $\mathbf{A}$, $\mathbf{R}$ as in (3.24).
2. *Block Arnoldi (GPU):* Generate $\mathbf{X} \in \mathbb{R}^{N_r \times q}$ using GEMM plus TSQR libraries in MAGMA corresponding to step 2.
3. *SPRIM projector (GPU):* Partition $\mathbf{X}$ into blocks based on SPRIM formulation and build the block-diagonal projector $\widetilde{\mathbf{X}} = \mathrm{diag}(\widehat{\mathbf{X}}_{\mathrm{int}}, \widehat{\mathbf{X}}_{\mathrm{ind}})$ as in (3.17). This partitioning is done in GPU using custom cuda kernels outlined in Algorithm 5.
4. *Orthogonlize the partitioned blocks (GPU):* If, the resulting blocks of the partitioned projector $\widetilde{\mathbf{X}}$ end up rank deficient, their columns must be orthogonalized again. This step is important since the un-orthogonalized columns can inject noise and/or produce incorrect time domain currents for the reduced models during transient simulations. This is done in GPU using a proposed kernel outlined in Algorithm 6.
5. *Projector Assembly (GPU):* The SPRIM projector $\widetilde{\mathbf{X}}$ is assembled using custom kernel in GPU.
6. *Split congruence projection (GPU):* Compute $\mathbf{G}_{\mathrm{q}}$, $\mathbf{C}_{\mathrm{q}}$, and $\mathbf{B}_{\mathrm{q}}$ matrices as in (3.18). This is done on the GPU on separate streams to increase parallelization and throughput.
7. *Frequency sweep (GPU):* For each ω, solve $(\mathbf{G}_{\mathrm{q}} + \mathrm{j}\omega\,\mathbf{C}_{\mathrm{q}})\mathbf{X} = \mathbf{B}_{\mathrm{q}}$, followed by computing $\mathbf{H}(\mathrm{j}\omega) = \mathbf{B}_{\mathrm{q}}^{\top}\mathbf{X}$.

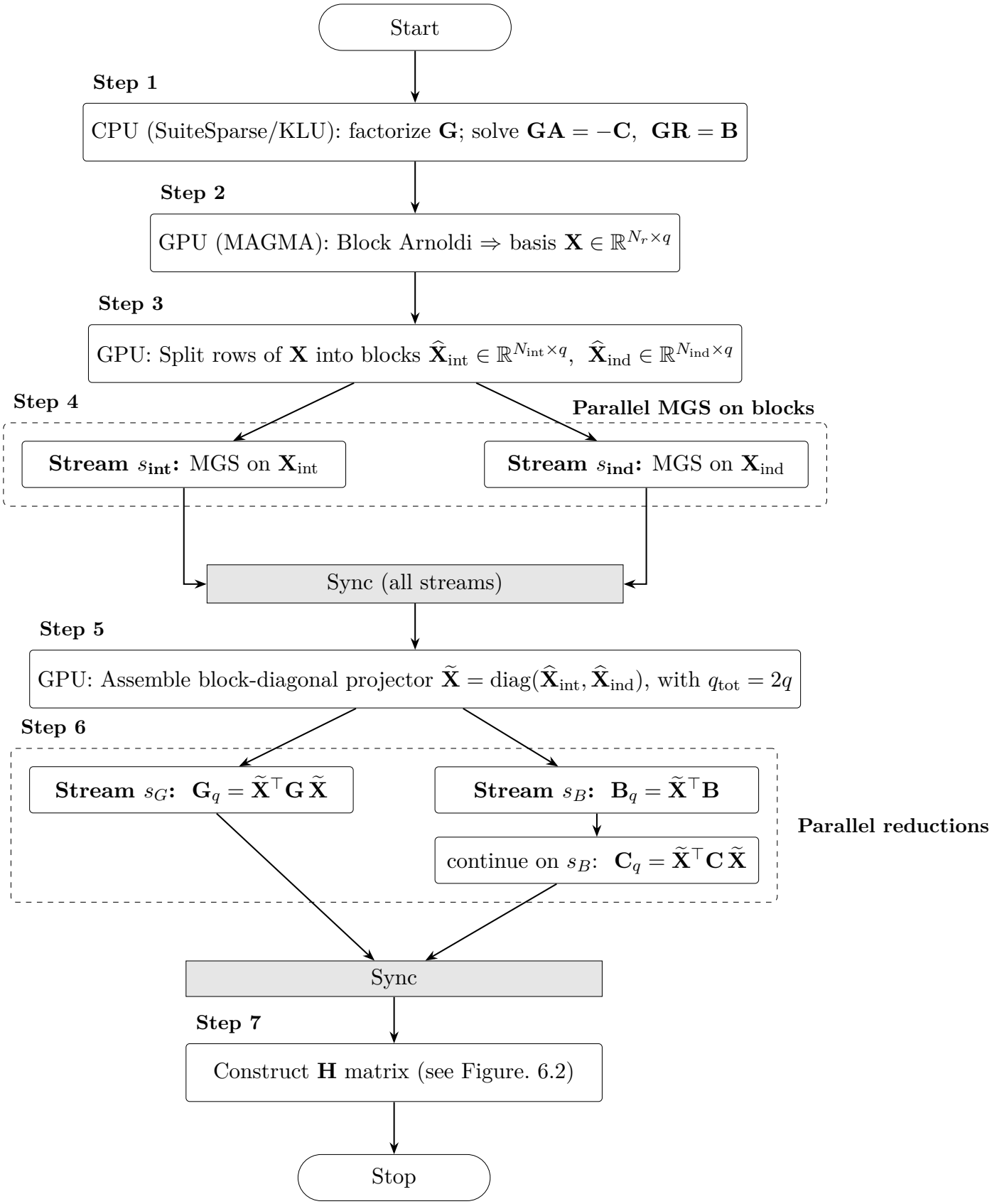


Figure 6.1: Algorithmic flowchart for the proposed GPU–SPRIM

6.3 Step 1: CPU based MNA Matrix Pre-processing

Similar to the proposed GPU-PRIMA (from the previous chapter), the sparse input matrices are read in COO format. CHOLMOD library is used to convert $\mathbf{G}$ to CSC format, and KLU library is used to perform the symbolic and numeric factorizations. Using the LU factors, forward and backward substitutions are applied to obtain dense, column-major arrays $\mathbf{A} \in \mathbb{R}^{N_r \times N_r}$ and $\mathbf{R} \in \mathbb{R}^{N_r \times N_p}$, which are compatible with MAGMA/LAPACK.

6.4 Step 2: GPU-based Block Arnoldi

In this step, a similar GPU Block Arnoldi kernel as in GPU PRIMA is developed to compute the projector $\mathbf{X}$. In each iteration, the block of vectors $\mathbf{A}\mathbf{X}_{k-1}$ (using GEMM operations) are generated. Next, block MGS projections (GEMMs) are performed to orthogonalize them against all previous block vectors followed by an application of a tall–skinny QR method (`magma_dgegqr_gpu`, MGS) by re-orthonormalizing inner column vectors. The resulting projector $\mathbf{X} = [\mathbf{X}_0, \ldots, \mathbf{X}_{n-1}]$ will have q columns.

6.5 Steps 3-5: SPRIM Projector Partitioning and Block-diagonal Assembly (GPU)

This step utilizes the projector generated from the Block Arnoldi algorithm. The rows N_r of the MNA conductance (and susceptance) matrix $\mathbf{G}$ are split as

$$N_r \;=\; n_{\text{int}} \;+\; n_{\text{ind}} \tag{6.1}$$

with n_{int} corresponding to the number of *internal* nodes, n_{ind} as circuit MNA *inductor current* variables. Correspondingly, $\mathbf{X}$ is partitioned by rows as:

$$\mathbf{X} \;=\; \begin{bmatrix} \mathbf{X}_{\text{int}} \\ \mathbf{X}_{\text{ind}} \end{bmatrix}, \qquad \mathbf{X}_{\text{int}} \in \mathbb{R}^{n_{\text{int}} \times q},\ \mathbf{X}_{\text{ind}} \in \mathbb{R}^{n_{\text{ind}} \times q} \tag{6.2}$$

In the SPRIM based projection, a modified Gram–Schmidt (MGS) procedure is performed on the columns of each of the blocks so that they have full column rank. The resulting blocks are then appended diagonally:

$$\widetilde{\mathbf{X}} \;=\; \mathrm{diag}\Big(\widehat{\mathbf{X}}_{\mathrm{int}},\, \widehat{\mathbf{X}}_{\mathrm{ind}}\Big) \;\in\; \mathbb{R}^{N_r \times q_{tot}}, \qquad q_{tot} \;=\; 2q. \tag{6.3}$$

Proposed Realization in GPU

On the device (GPU) (i) *split* $\mathbf{X}$ into $\mathbf{X}_{\mathrm{int}}$, $\mathbf{X}_{\mathrm{ind}}$ with a 2-D copy kernel; (ii) perform a parallelized Modified Gram–Schmidt (MGS) over the columns of each block in parallel across different CUDA streams (one stream per block) to improve inter-block orthogonality; (iii) *assemble* $\widetilde{\mathbf{X}}$ by writing the newly orthogonalized blocks $[\widehat{\mathbf{X}}_{\mathrm{int}} \,|\, \widehat{\mathbf{X}}_{\mathrm{ind}}\,]$ into a new buffer of size $N_r \times (2q)$. The MGS updates for computing the blocks $\widehat{\mathbf{X}}_{\mathrm{int}}$, $\widehat{\mathbf{X}}_{\mathrm{ind}}$ are executed concurrently. It is important to note that the inner loop of the MGS algorithm is parallelized to achieve as much concurrency and GPU utilization as possible. This sequence of operations indicated in this section corresponds to steps 3-5 of the flowchart in figure 6.1. The proposed algorithms for this parallel MGS procedure is given in Algorithms 5 and 6:

Algorithm 5 Split–Orthogonalize–Assemble Kernel (SPRIM without port block)

Inputs: global basis $\mathbf{X} \in \mathbb{R}^{N_r \times q}$; row-partition sizes n_{int}, n_{ind} with $n_{\mathrm{int}} + n_{\mathrm{ind}} = N_r$.
Outputs: $\widetilde{\mathbf{X}} \in \mathbb{R}^{N_r \times 2q}$ (block-diagonal).

Step a) (Split Original Projector): Launch a 2-D copy kernel to extract

$$\mathbf{X}_{\mathrm{int}} \leftarrow \mathbf{X}(1{:}n_{\mathrm{int}}, :), \quad \mathbf{X}_{\mathrm{ind}} \leftarrow \mathbf{X}(n_{\mathrm{int}}{+}1{:}N_r, :).$$

Step b) (Concurrent MGS): On CUDA streams s_1, s_2, run *column-wise MGS concurrently* on $\mathbf{X}_{\mathrm{int}}$ and $\mathbf{X}_{\mathrm{ind}}$ using Algorithm 6.

(Each stream applies Algorithm 6 to its block independently.)

Step c) (Assemble): Allocate $\widetilde{\mathbf{X}} \in \mathbb{R}^{N_r \times 2q}$. Launch a single assembly kernel to write

$$\widetilde{\mathbf{X}}(1{:}n_{\mathrm{int}},\, 1{:}q) \leftarrow \widehat{\mathbf{X}}_{\mathrm{int}},$$
$$\widetilde{\mathbf{X}}(n_{\mathrm{int}}{+}1{:}N_r,\, q{+}1{:}2q) \leftarrow \widehat{\mathbf{X}}_{\mathrm{ind}}.$$

Return $\widetilde{\mathbf{X}}$.

Algorithm 6 Column-wise Modified Gram–Schmidt (MGS)

Input: $\mathbf{X} \in \mathbb{R}^{n \times m}$ ▷ block with m columns
Output: $\widehat{\mathbf{X}}$ with orthonormal columns

for $i = 1$ to m **do**
  $r_{ii} \leftarrow \|\mathbf{X}(:, i)\|_2$
  $\widehat{\mathbf{X}}(:, i) \leftarrow \mathbf{X}(:, i)/r_{ii}$

  for $j = i + 1$ to m **do** in parallel for each column
    $r_{ij} \leftarrow \widehat{\mathbf{X}}(:, i).\mathbf{X}(:, j)$
    $\widehat{\mathbf{X}}(:, j) \leftarrow \mathbf{X}(:, j) - r_{ij}\,\widehat{\mathbf{X}}(:, i)$
  end for
end for

6.6 Step 6: Projection to the SPRIM Reduced Model (concurrent streams in GPU)

With $\widetilde{\mathbf{X}}$ computed on the GPU, the reduced matrices of equation (3.18) are computed on GPU. The projection step is largely unchanged from GPU-PRIMA implementation where $\mathbf{B}_\mathrm{q}$ is reduced in a *second CUDA stream* while the first stream performs the $\mathbf{C}$ and $\mathbf{G}$ reductions. The resulting matrices will retain their block structure similar to the original system matrices, thus facilitating the retention of enhanced sparsity in the reduced matrices.

6.7 Step 7: Frequency-Domain Response Computation (GPU)

It is to be noted that, SPRIM reduced models differs from PRIMA as it retains the differential-algebraic structures of the original matrices within the reduced matrices. Consequently, direct frequency sweeps using a pole-residue method is challenging using eigenvalue-decomposition/diagonalization. Thus the frequency domain port responses are obtained via direct solutions using:

$$\left(\mathbf{G}_\mathrm{q} + \mathrm{j}\omega\,\mathbf{C}_\mathrm{q}\right)\mathbf{X}(\omega) = \mathbf{B}_\mathrm{q}, \tag{6.4}$$

$$\mathbf{H}(\mathrm{j}\omega) = \mathbf{B}_\mathrm{q}^\top\,\mathbf{X}(\omega). \tag{6.5}$$

For this purpose, the following operations are carried out on the GPU for each frequency ω. First, the matrix $\mathbf{G}_\mathrm{q} + \mathrm{j}\omega\,\mathbf{C}_\mathrm{q}$ is formed in complex arithmetic using an aaxpy (scaled vector-vector addition) update [31]. Next, an LU factorization is computed using `zgetrf` (MAGMA routine). This is followed by solving for $\mathbf{X}$ via q_{tot}-by-q_{tot} triangular solution routine `zgetrs` (from MAGMA). Finally, the batch of port transfer function entries is accumulated as $\mathbf{H} = \mathbf{B}_\mathrm{q}^\top \mathbf{X}$ using a complex GEMM operation.
Here, the host code (CPU side) overlaps computations in GPU and I/O (writing $\mathbf{H}$) with asynchronous copies. For this purpose, CUDA *events* are used to record when each batch of the transfer function matrix $\mathbf{H}$ becomes available, so that asynchronous device-to-host (D2H) transfers can be initiated immediately. This asynchronous copy does not block the GPU and it is free to start a new batch of computations for the transfer function matrix immediately. Figure 6.2 demonstrates how computation and asynchronous copies are overlapped.

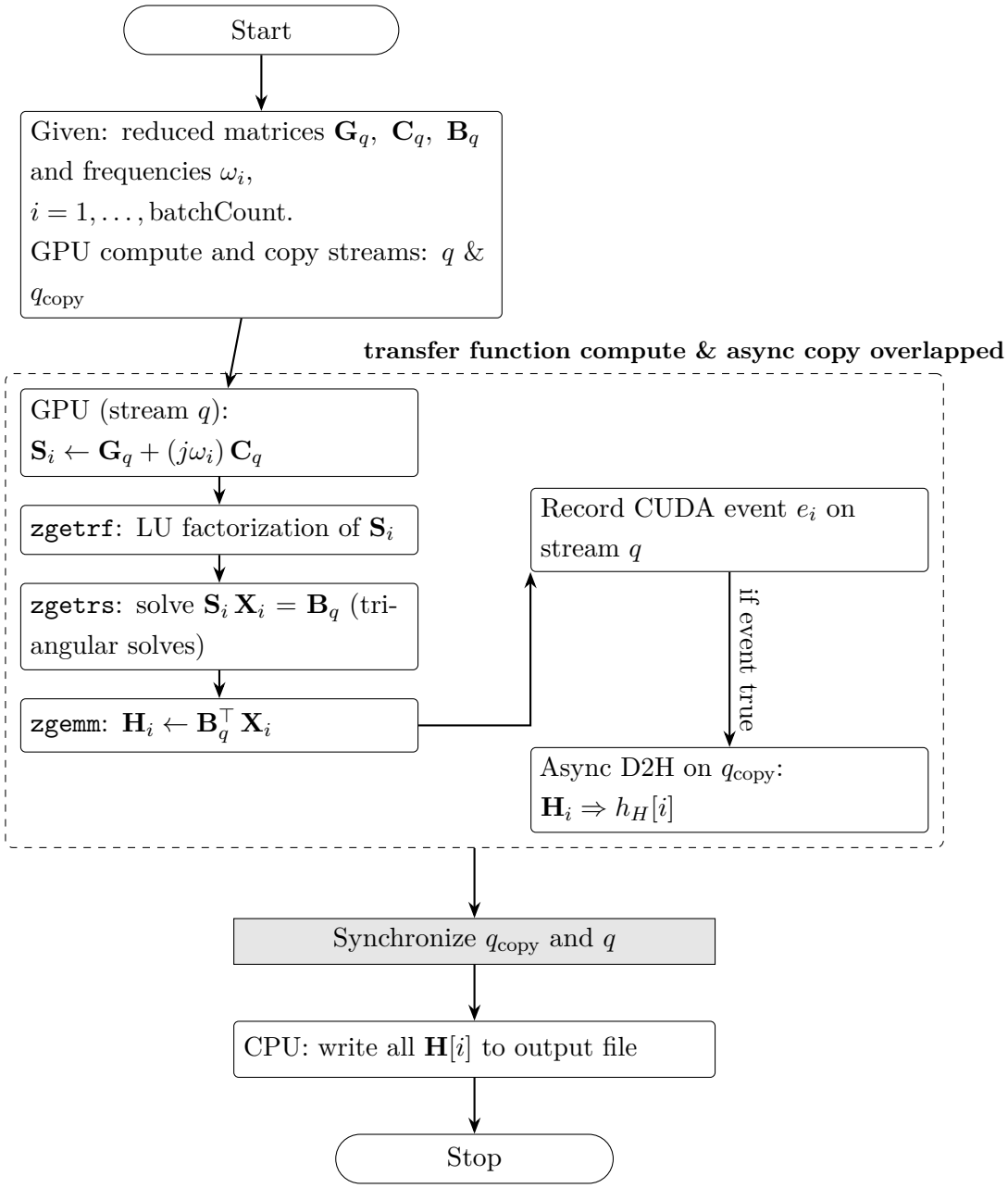


Figure 6.2: **H**-matrix response calculation with overlapping compute and async D2H copies

6.8 Discussion

6.8.1 Data layout, precision, and libraries

As described in section 4.10.1 of chapter 4, all dense arrays are stored in column-major format. Matrices $\widetilde{\mathbf{X}}$, $\mathbf{G}$, $\mathbf{C}$, $\mathbf{B}$ and all the reduced matrices are kept on the GPU

to minimize PCIe traffic and data transfers as much as possible. Double precision (real/complex) arithmetic is used for numerical accuracy.
The library stack utilized in the implementation also remains relatively unchanged:

- **SuiteSparse (CPU):** CHOLMOD for I/O format conversion; KLU for $\mathbf{G}$ factorization and sparse solves (Step 1).
- **MAGMA (GPU):** BLAS–3 GEMMs; tall–skinny QR; LU/solve in complex; vector operations for norms/scaling (Steps 2, 6, 7).
- **CUDA (GPU):** custom kernels for block splitting/assembly of $\widetilde{\mathbf{X}}$, streamed MGS, and real→complex casts (Steps 3–5, 7).

6.8.2 Workload and Performance Overview

The overview of computational workload and performance behavior of the proposed GPU–SPRIM framework are briefly summarized. Each major phase of the algorithm—Block Arnoldi generation, projector construction, and frequency-domain reduction—is examined with respect to its computational characteristics and the methods used to adopt concurrency and memory efficiency. The following notes outline the main performance considerations and implementation details observed during the development of the GPU-accelerated SPRIM model reduction pipeline.

- *Block Arnoldi:* This step is similar to PRIMA until $\mathbf{X}$ is available (Step 2).
- *SPRIM projector:* The projector block vector splitting $\mathbf{X} \rightarrow \mathbf{X}_{\text{int}}, \mathbf{X}_{\text{ind}}$ and MGS add $\mathcal{O}(q^2(n_{\text{out}} + n_{\text{ind}}))$ flops but are bandwidth-light and overlap across streams. Two CUDA streams are used both in the projector orthogonalization (separate MGS on $\mathbf{X}_{\text{out}}$, $\mathbf{X}_{\text{ind}}$) and in the reduction phase ($\mathbf{B}_{\text{q}}$ overlapped with $\mathbf{C}_{\text{q}}, \mathbf{G}_{\text{q}}$) (Steps 3–6).
- *Projection:* This step is similar to GPU–PRIMA, using overlapped GEMMs to form $\mathbf{B}_q$, $\mathbf{C}_q$, and $\mathbf{G}_q$, improving GPU utilization (Step 6).
- *Frequency sweep (per ω):* one $q_{tot} \times q_{tot}$ LU of $\mathbf{G}_q + \mathrm{j}\omega\,\mathbf{C}_q$, followed by $q_{tot} \times N_p$ triangular solutions, and a final by $q_{tot} \times N_p$ GEMM to form $\mathbf{H}$. Here $q_{tot} = 2q$ (Step 7).
- The reduced models from SPRIM typically have a larger memory footprint compared to their PRIMA counterparts, but the moment matching capabilities are improved for the same level of computational complexities. This will be further examined in the numerical example sections.

6.9 Summary

In this chapter, development of the proposed GPU-SPRIM algorithm is given. GPU-SPRIM follows GPU-PRIMA through sparse pre-processing and Block Arnoldi, then diverges by constructing a block-diagonal projector $\widetilde{\mathbf{X}}$ with total reduced size $q_{tot} \times q_{tot}$ ($q_{tot} = 2q$). Using concurrent streams, the reduced matrices are formed and at each frequency, the reduced system $(\mathbf{G}_\mathrm{q} + \mathrm{j}\omega\,\mathbf{C}_\mathrm{q})\mathbf{X} = \mathbf{B}_\mathrm{q}$ is solved and assembled to get $\mathbf{H}_q(\mathrm{j}\omega) = \mathbf{B}_\mathrm{q}^\top \mathbf{X}$. The implemented algorithm preserves the structure of the original system block, improves moment matching capabilities and exploits BLAS–3 intensity and GPU concurrency for scalable performance.

Chapter 7

Numerical Results for the Proposed GPU-SPRIM

In this chapter, numerical results validating the efficiency of the proposed GPU-SPRIM algorithm is given. For the experiments here, the same hardware setup and circuit cases that are used for validating the proposed GPU-PRIMA in Chapter 5 are considered.

7.1 Example 1: Multi-port RLC Circuit with $Np = 40$

In this experiment, the same multi-port RLC circuit with $Np = 40$ ports (Section 5.2, Chapter 5) is considered. As stated previously, the resulting MNA matrix size is 11683×11683 with 7022 internal nodes. While constructing the Krylov projector using the proposed GPU-SPRIM, the same number of moments are matched (25 moments per port ($q = 25Np$). In this case, the resulting first-order reduced-order model size is 2000×2000 ($q_{tot} = 2q = 2 \times 25Np$). However, this can be easily translated to a second-order form [16], whereby a reduced-order model of size 1000×1000 can be obtained, which is used in generating the frequency responses. Frequency sweep is performed over a bandwidth up to 1.3 GHZ and at 2000 frequency points (similar to the case in Example 1 in Section 5.2 of Chapter 5). The corresponding frequency responses from both the original circuit (solid line) and the reduced models (dash-dot line) are shown in Figure 7.1, Figure 7.2 and Figure 7.3.

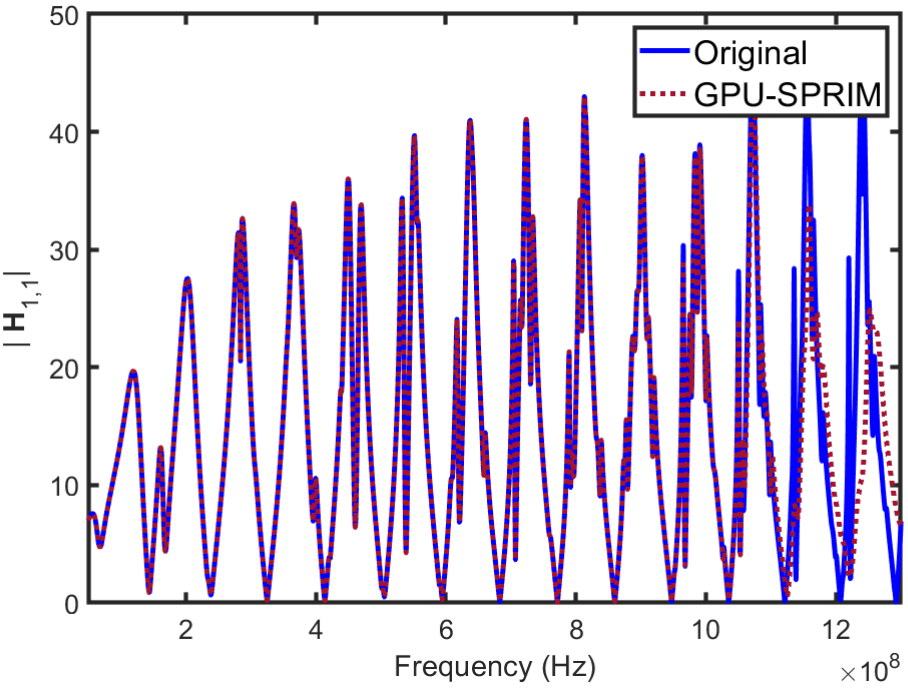


Figure 7.1: Comparison of $|\mathbf{H}_{1,1}(j\omega)|$ from original system and GPU-SPRIM reduced models (40-port network)

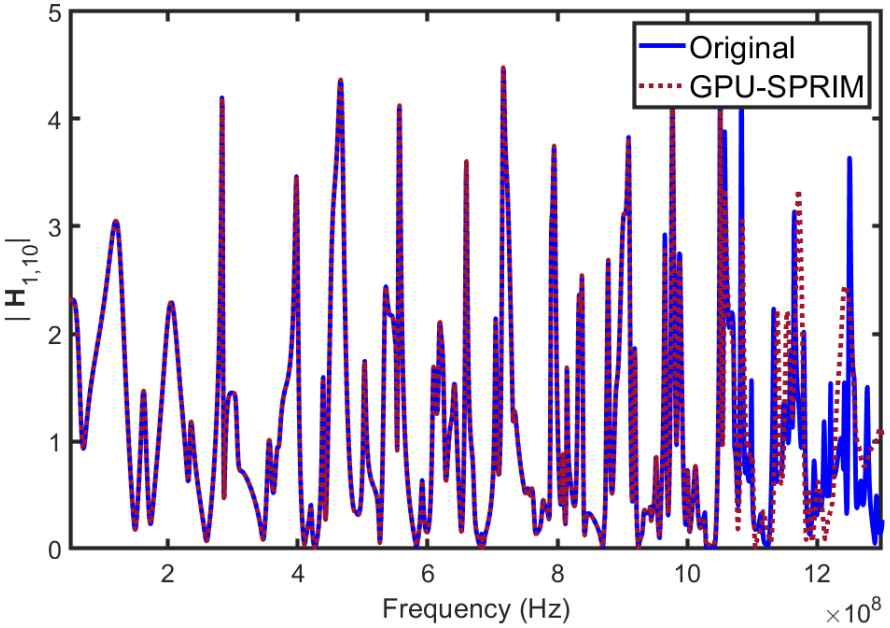


Figure 7.2: Comparison of $|\mathbf{H}_{1,10}(j\omega)|$ from original system and GPU-SPRIM reduced models (40-port network)

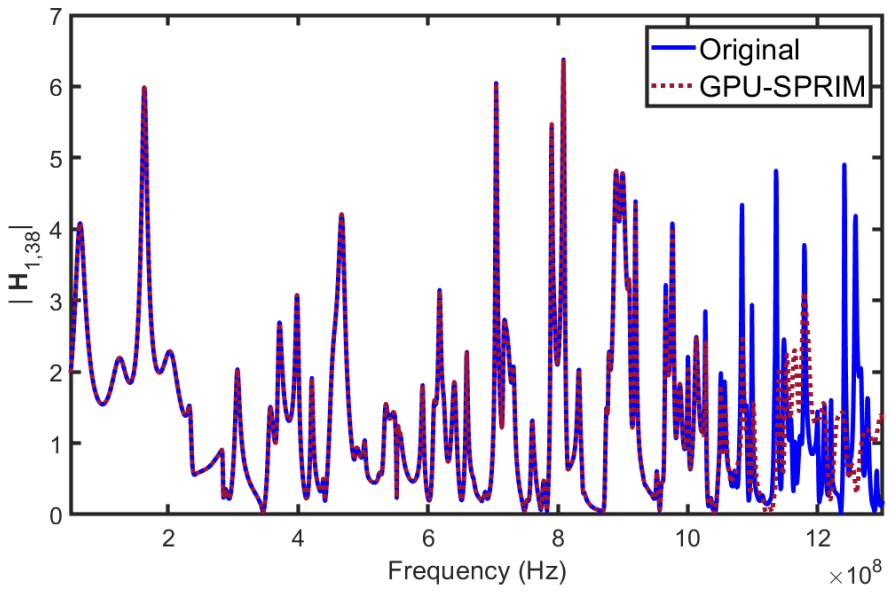


Figure 7.3: Comparison of $|\mathbf{H}_{1,38}(j\omega)|$ from original system and GPU-SPRIM reduced models (40-port network)

As can be seen, the responses from both the approaches match reasonably well. The responses from single-core and multi-core based SPRIM are omitted since they are identical to that of the proposed GPU-SPRIM. A computational cost comparison of all three approaches is provided in Table 7.1.

Table 7.1: Comparison of Computational Performance Times for SPRIM 2^{nd} Order Implementations (Example-1, 40-Port Circuit)

Step	**Single-core SPRIM (s)**	**Multi-core SPRIM (s)**	**Proposed GPU-SPRIM (s)**
Computation of **R** (in CPU) - CPU-PRIMA [Step 1]	0.01	0.009	–
Computation of **A** and **R** (in CPU) - GPU-PRIMA [Step 1]	–	–	1.01
Block Arnoldi [Step 2]	7.18	2.63	0.58
Partition **X** [Step 3]	0.20	0.01	0.0005
Parallel MGS and projector re-assembly [Step 4 – Step 5]	7.38	2.51	0.67
G, **C** and **B** reductions [Step 6]	26.39	1.87	0.62
Total time taken to generate reduced state space models	*41.16*	*7.03*	*2.88*
Speed-up (while generating reduced state space system)	–	**5.86**	**14.29**
Frequency-sweep **H** matrix computation (including time to transform system to 2^{nd} Order) [Step 7]	224.48	109.72	30.44
Total time taken (including frequency response computation)	*265.64*	*116.75*	*33.32*
Overall Speed-up (including frequency response computation)	–	**2.28**	**7.97**

Table 7.2: Comparison of Computational Performance Times for SPRIM 1^{st} Order Implementations (Example-1, 40-Port Circuit)

Step	**Single-core SPRIM (s)**	**Multi-core SPRIM (s)**	**Proposed GPU-SPRIM (s)**
Frequency-sweep **H** matrix computation	955.70	320.67	54.55

7.1.1 Discussion

In this example (with $Np = 40$ and $N_r = 11683$), the GPU-SPRIM reduced model (in the second-order form) produces responses with reasonably good accuracy in comparison to the original circuit responses. As shown in Figures. 7.1–7.3, the magnitude responses $|\mathbf{H}_{1,1}(j\omega)|$, $|\mathbf{H}_{1,10}(j\omega)|$, and $|\mathbf{H}_{1,38}(j\omega)|$ from both the approaches closely match over the frequency band up to approximately 1.2 GHz.

The itemized timing breakdown in Table 7.1 highlights that the GPU-SPRIM implementation achieved a significant speed-up of approximately 14.3× compared to the single-core CPU based approach (41.16 s vs. 2.88 s) while generating the reduced model, and about 2.4× compared to the multi-core implementation (7.03 s vs. 2.88 s). This is possible since the matrix multiplication stages are accelerated substantially on the GPU. The table also provides the timings for frequency-domain response computation using second-order form of the reduced models (3.22). As can be seen, the *frequency-domain* **H**-matrix solutions are the most expensive part of the GPU-SPRIM algorithm (30.44 s, $\approx 91\%$ of the total GPU runtime). They scale cubically with the reduced model dimension (approximately $O(q^3)$ for this case), diminishing the time savings achieved from fast Arnoldi and reduction phases. The proposed GPU-SPRIM algorithm preserves the RLC block structure of the original MNA formulation while still providing substantial acceleration over CPU-only implementations (a speed-up of approximately 8.0× versus the single-core CPU implementation and about 3.5× versus the multi-core CPU implementation for computations including frequency response evaluation).

A further investigation directly using the first-order form of SPRIM reduced model is done, and the corresponding CPU timings are given in Table 7.2. It is to be noted that the first-order SPRIM reduced model is of roughly twice the size of the second-order form and consequently, the frequency sweep becomes significantly more expensive:

the GPU **H**-matrix computation increases from 30.44 s to 54.55 s. On the CPU, the corresponding times increase from 224.48 s to 955.70 s (single-core) and from 109.72 s to 320.67 s (multi-core).

7.2 Example 2: Multi-port RLC Circuit with $Np = 128$

7.2.1 Frequency Domain Response

The second test case is the same as the 128-port RLC circuit network used in Chapter 5 (Section 5.3). For GPU-SPRIM, 30 moments per port are matched ($q = 30Np$), which produces a reduced model of size 3840×3840 (in second-order form). The same frequency sweep setup as in Example 7.1 is used. The transfer function responses are evaluated at 2000 frequency points up to 1.3 GHz. The representative magnitude responses are plotted in Figures 7.4–7.6. In each figure, the solid curve corresponds to the original full-order system, whereas the dash-dot curves shows the responses from the proposed GPU-SPRIM reduced models.

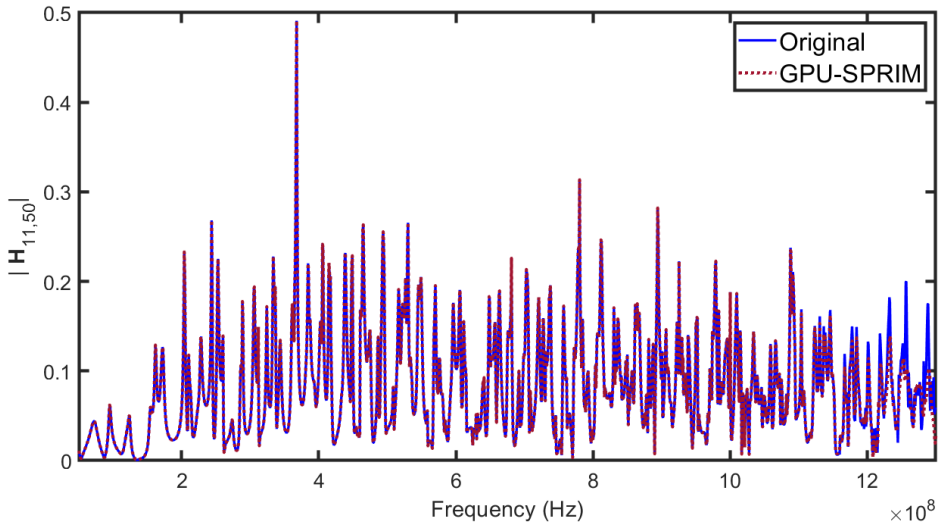


Figure 7.4: Comparison of $|\mathbf{H}_{11,50}(j\omega)|$ from the original system and the GPU-SPRIM reduced model (128-port network)

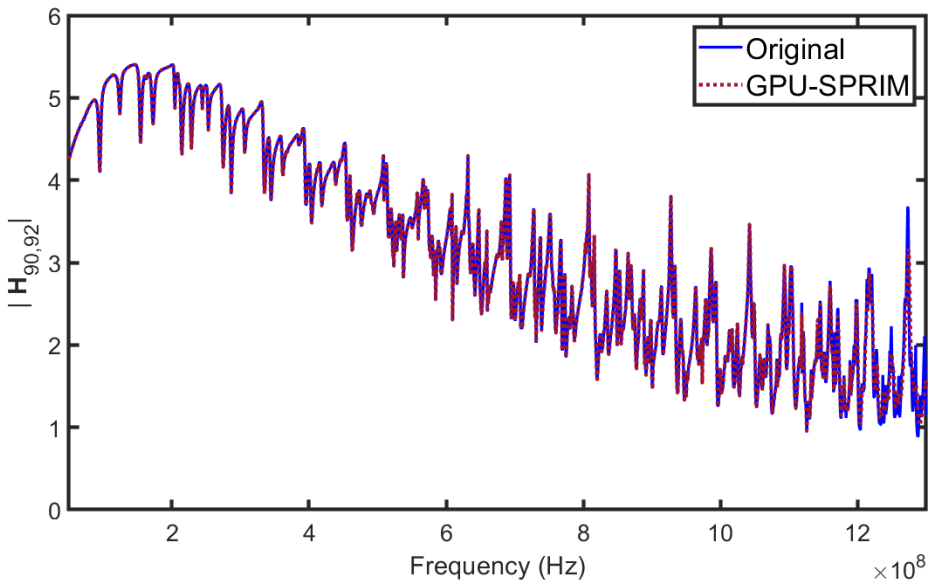


Figure 7.5: Comparison of $|\mathbf{H}_{90,92}(j\omega)|$ from the original system and the GPU-SPRIM reduced model (128-port network)

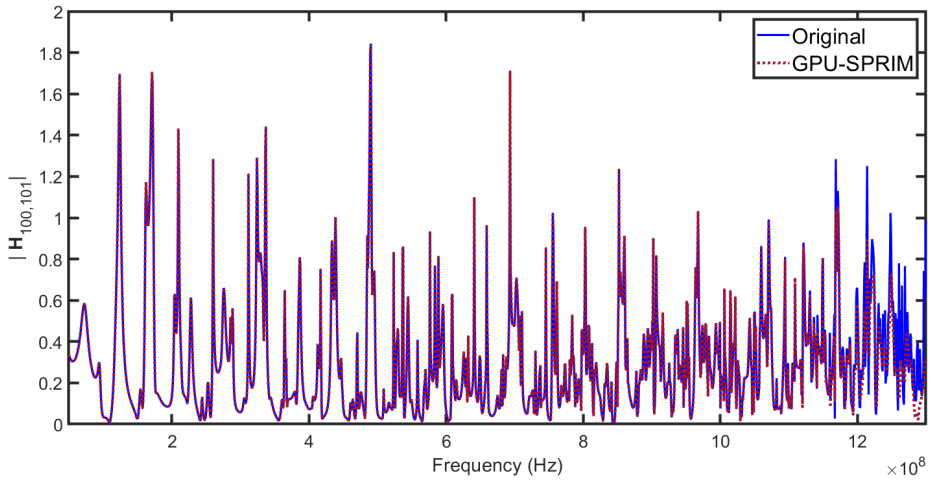


Figure 7.6: Comparison of $|\mathbf{H}_{100,101}(j\omega)|$ from the original system and the GPU-SPRIM reduced model (128-port network)

The plots in Figures 7.4–7.6 show that the reduced model responses follows the original system behavior closely. It is noted that, the single-core and multi-core CPU SPRIM implementations yield the same frequency responses as the proposed GPU-SPRIM, and are therefore not shown separately. An itemized listing of the run-times for all three implementations is given in Table 7.3.

Table 7.3: Comparison of Computational Performance Times for SPRIM 2^{nd} Order Implementations (Example-2, 128-Port Circuit)

Step	**Single-core SPRIM (s)**	**Multi-core SPRIM (s)**	**Proposed GPU-SPRIM (s)**
Computation of **R** (in CPU) - CPU-PRIMA [Step 1]	0.15	0.095	–
Computation of **A** and **R** (in CPU) - GPU-PRIMA [Step 1]	–	–	18.11
Block Arnoldi [Step 2]	87.32	27.33	6.59
Partition **X** [Step 3]	0.44	0.19	0.0061
Parallel MGS and projector re-assembly [Step 4 - Step 5]	341.74	64.22	6.83
G and **C** reductions [Step 6]	758.16	58.61	4.37
Total time taken to generate reduced state space models	*1187.81*	*150.45*	*35.91*
Speed-up (while generating reduced state space system)	–	**7.90**	**33.08**
Frequency-sweep **H** matrix computation (including time to transform system to 2^{nd} Order) [Step 7]	1559.21	1102.37	99.70
Total time taken (including frequency response computation)	*2747.02*	*1252.82*	*135.61*
Overall Speed-up (including frequency response computation)	–	**2.19**	**20.26**

Table 7.4: Comparison of Computational Performance Times for SPRIM 1^{st} Order Implementations (Example-2, 128-Port Circuit)

Step	**Single-core SPRIM (s)**	**Multi-core SPRIM (s)**	**Proposed GPU-SPRIM (s)**
Frequency-sweep **H** matrix computation [Step 6]	6667.41	4044.37	201.10

7.2.2 Discussion

For the 128-port example, the GPU-SPRIM approach provides reasonably well matching responses with that from the original system. The timing breakdown in Table 7.3 illustrates how the workload is distributed for the different stages for this larger RLC circuit. On the CPU, the block Arnoldi step dominates the reduction phase: it requires 87.32 s on a single core and 27.33 s on the multi-core implementation. This step dominates the bulk of the reduced-model generation time. The GPU implementation reduces this to 6.59 s, giving more than one order of magnitude speed-up versus the single-core implementation, and also a substantial gain over the multi-core implementation. The overall time to construct the reduced state-space model drops from 1187.81 s (single-core) and 150.45 s (multi-core) to 35.91 s on the GPU. As stated previously, the dense, matrix-matrix multiplication heavy pipeline of GPU-SPRIM (Arnoldi and the subsequent reductions) map very well to the GPU. For the 3840×3840 reduced model (second-order form), the frequency-domain **H**-matrix solutions become the main cost: 99.70 s on the GPU, compared with 1102.37 s on the multi-core CPU. These factorizations are responsible for the majority of the total runtime on both platforms. Including the frequency sweep, the total GPU runtime is 135.61 s, compared to 2747.02 s and 1252.82 s for the single-core and multi-core SPRIM implementations, respectively. Thus, for this 128-port circuit, the proposed GPU-SPRIM achieves an overall speed-up of about 31.38 times relative to the single-core CPU and a speed-up of about 2.68 times relative to the multi-core CPU, while maintaining close agreement with the original circuit responses.

CPU and GPU timings for the corresponding first-order SPRIM realization are given in Table 7.4. The first-order formulation has roughly twice the state-space dimension of the second-order model and, as a consequence, the frequency sweep becomes significantly more expensive: the GPU **H**-matrix computation increases from 99.70 s to 201.10 s. On

the CPU, the corresponding single-core and multi-core times increase from 1559.21 s to 6667.41 s and from 1102.37 s to 4044.37 s, respectively. This behavior underlines the CPU advantage of using the second-order form of SPRIM reduced model for large multi-port interconnects.

7.3 Accuracy Comparison Between Original Model Vs. GPU-PRIMA and GPU-SPRIM

An accuracy comparison is provided for the original model and the proposed GPU-PRIMA and for the original model and the proposed GPU-SPRIM. The Example-2 results from Chapter 5 (Section 5.3) and Chapter 7 (Section 7.2) are used to demonstrate the accuracies between these methods. For each frequency point ω_k, let $\mathbf{H}(j\omega_k)$ and $\mathbf{H}_{\mathrm{q}}(j\omega_k)$ denote the original system and reduced system transfer function matrices respectively. The corresponding *error matrix* is defined as

$$\mathbf{\Psi}(j\omega_k) = \mathbf{H}_{\mathrm{q}}(j\omega_k) - \mathbf{H}(j\omega_k), \tag{7.1}$$

where the absolute value is taken entrywise. The scalar magnitude error $\alpha(\omega_k)$ at the frequency point ω_k is then defined as the Frobenius norm of $\mathbf{\Psi}(\omega_k)$:

$$\alpha(j\omega_k) \;=\; \|\mathbf{\Psi}(j\omega_k)\|_F = \sqrt{\sum_{a=1}^{Np}\sum_{b=1}^{Np}\left|\Psi_{ab}(j\omega_k)\right|^2} = \sqrt{\sum_{a=1}^{Np}\sum_{b=1}^{Np}\left(\left|H_{\mathrm{q},ab}(j\omega_k) \;- H_{ab}(j\omega_k)\right|\right)^2}. \tag{7.2}$$

The Frobenius–based magnitude errors $\alpha(\omega_k)$ are evaluated at frequency points up to 1 GHz. The average scalar error over this band is given by

$$\overline{\alpha} = \frac{1}{N_\omega}\sum_{k=1}^{N_\omega}\alpha(j\omega_k). \tag{7.3}$$

Here, N_ω denotes the number of frequency points (1520) within the bandwidth defined above. Figure 7.7–Figure 7.9 provide a sample comparison of the responses between the original model response (solid blue line) and the responses obtained from GPU-PRIMA (dashed green) and GPU-SPRIM (dotted red) models.

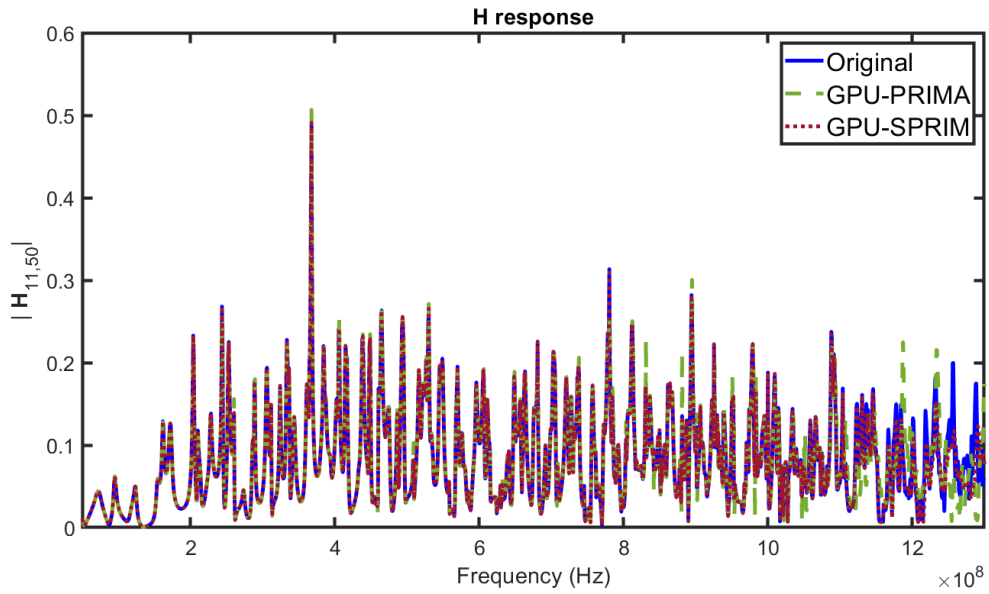


Figure 7.7: Comparison of $|\mathbf{H}_{11,50}(j\omega)|$ from the original system, GPU-PRIMA and GPU-SPRIM reduced models (128-port network)

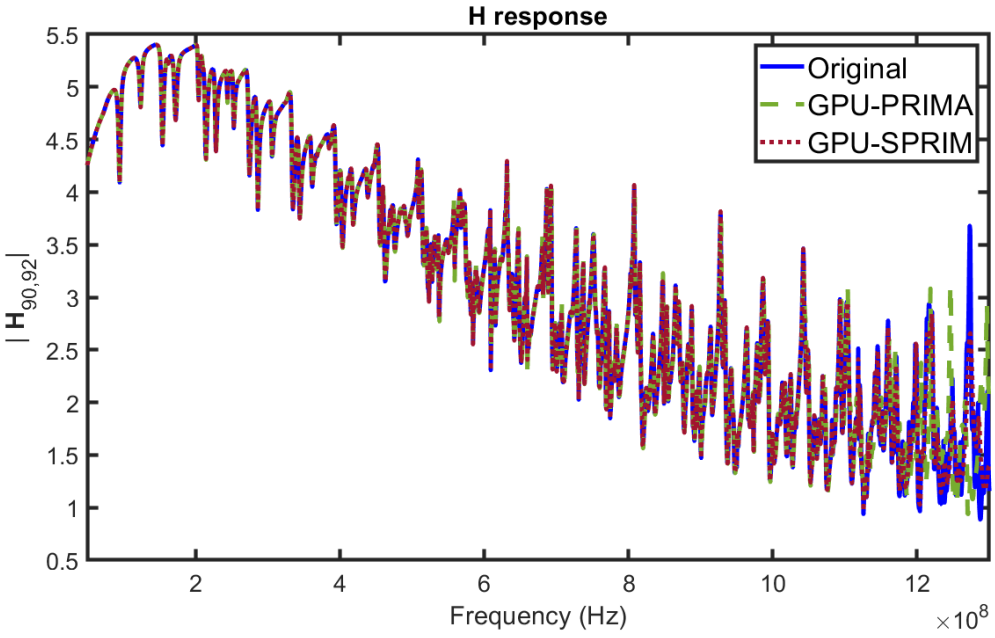


Figure 7.8: Comparison of $|\mathbf{H}_{90,92}(j\omega)|$ from the original system, GPU-PRIMA and GPU-SPRIM reduced models (128-port network)

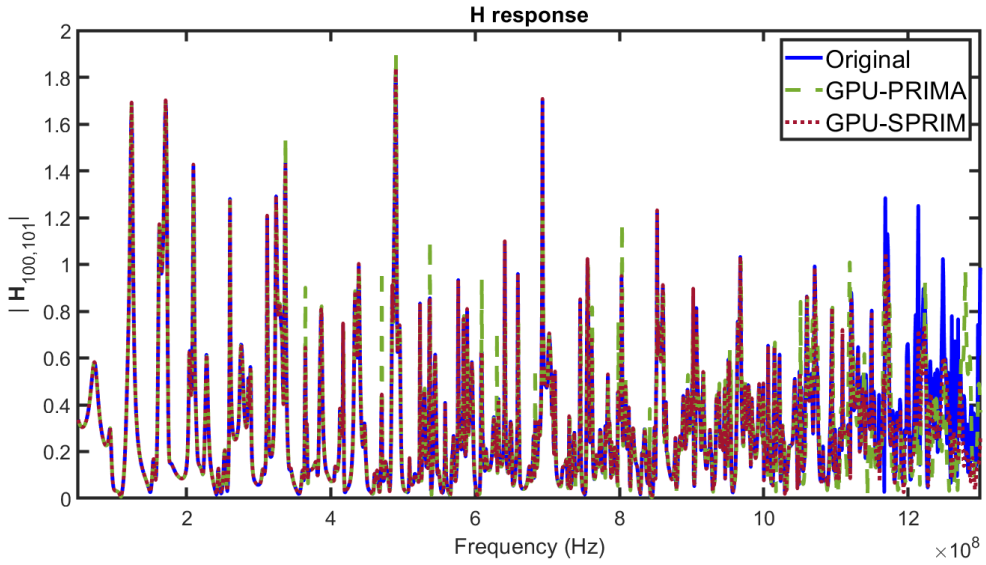


Figure 7.9: Comparison of $|\mathbf{H}_{100,101}(j\omega)|$ from the original system, GPU-PRIMA and GPU-SPRIM reduced models (128-port network)

To quantify the accuracy of the reduced-order models over all the ports, the Frobenius norm of the error matrix was evaluated at each sampled frequency. For a given angular frequency ω_k, the error matrix defined in (7.1) and its Frobenius norm given by (7.2) are evaluated. The average Frobenius norm up to 1 GHz is then computed according to (7.3).

Table 7.5 reports the resulting averaged Frobenius norms of the magnitude error for the transfer functions for both the proposed GPU-PRIMA and the proposed GPU-SPRIM reduced models against the original system models. Table 5.2 and Table 7.3 contain the speed ups for the 128-port circuit considered here.

Table 7.5: Average Frobenius norm of the magnitude error over the frequency band 0–1 GHz for the proposed implementations GPU-PRIMA and GPU-SPRIM (128-port network)

Transfer Function Frobenius Error	**GPU-PRIMA:**	**GPU-SPRIM:**
$\overline{\alpha}(\omega)$	4.39e-01	8.63e-04

7.3.1 Discussion

The averaged Frobenius norms of the magnitude error, defined in (7.1)–(7.3), provide a measure of how the reduced models reproduce the multi-port transfer function behavior

compared to the full $Np \times Np$ transfer function. For the 128-port example, proposed GPU-PRIMA reduced model yields an average Frobenius error of approximately 4.39×10^{-1} up to the 1 GHz band, whereas the proposed GPU-SPRIM model reduces this to about 8.63×10^{-4}. These results are consistent with the frequency-response plots, where both reduced models track the original responses closely, with SPRIM exhibiting better matching all throughout the band. Overall, the Frobenius-norm metrics show that the structure-preserving GPU-SPRIM projection improves the accuracy of the reduced models due to it's $2q$ moment matching capabilities when compared to the proposed GPU-PRIMA based reduction. However, this comes at an increased computation cost during reduction due to slightly larger sized reduced order models which take longer time for evaluation.

7.4 Transient Analysis: Multi-port RLC Circuit with $Np = 40$

The multi-port RLC circuit with $Np = 40$ ports from (Section 5.2, Chapter 5) is considered for demonstrating the feasibility and efficiency of the reduced order models from the proposed GPU-PRIMA and GPU-SPRIM algorithms in a time-domain analysis. For this purpose, reduced models are generated by matching 12 block moments ($n = 12$). In this case, GPU-PRIMA reduced model is of size $12 \times Np = 480 \times 480$, whereas the GPU-SPRIM reduced model is of size $2 \times 12 \times Np = 960 \times 960$. The reduced order models from both implementations are considered in the first-order form for use in a time-stepping Trapezoidal Integration scheme [33]. The input excitation considered is a finite-duration voltage pulse at port 1 with a **rise time** of 1 ns, a **pulse width** of 10 ns, and a **fall time** of 1 ns as shown in Figure 7.10. The corresponding open circuit transient responses are computed at the ports. Figures 7.11- 7.13 show a sample comparison of the time domain output voltage waveforms at different ports .

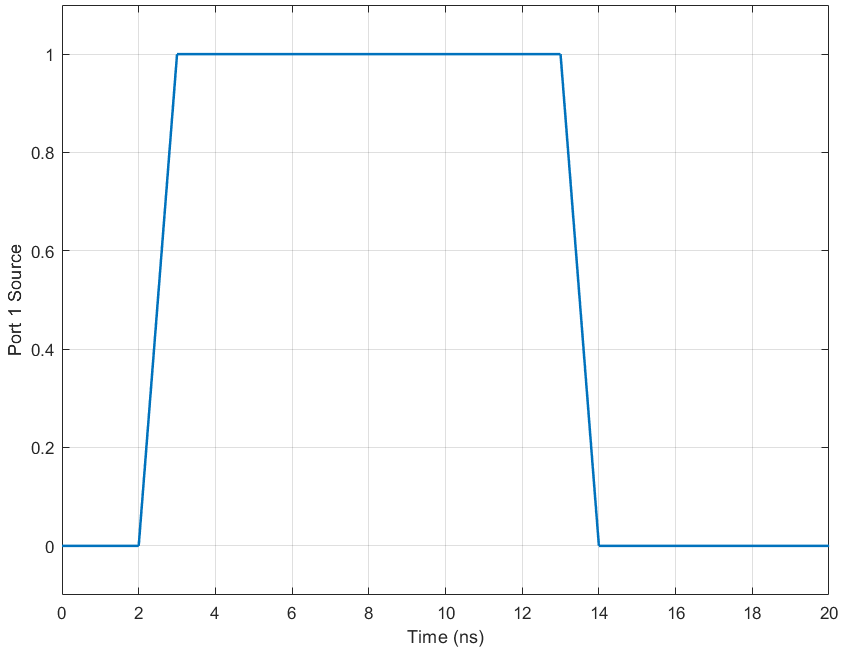


Figure 7.10: Input Source Signal at Port 1

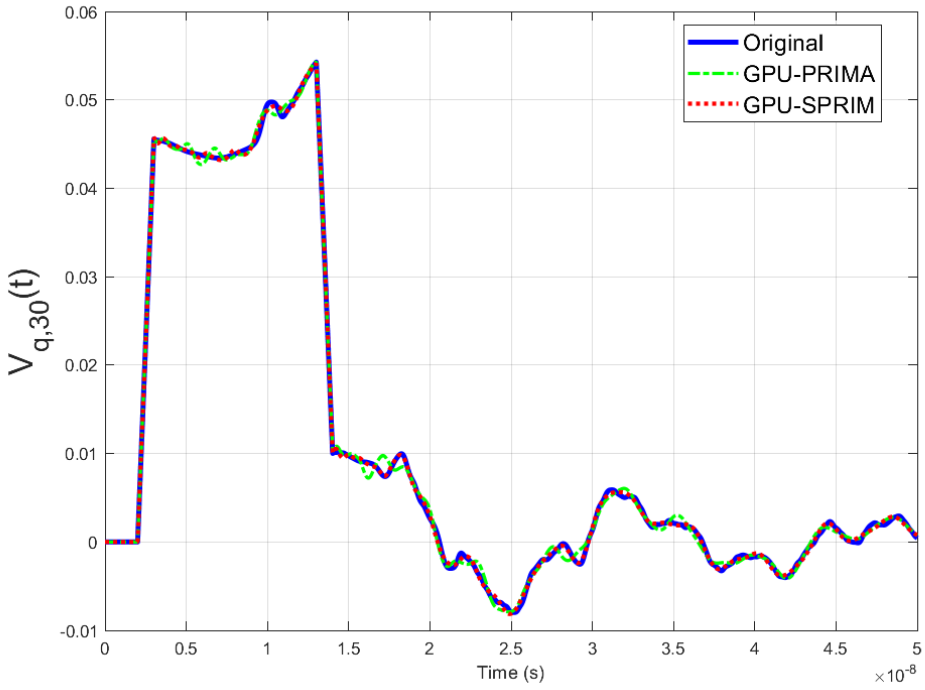


Figure 7.11: Comparison of $\mathbf{V}_4(t)$ voltage waveform from the original system, GPU-PRIMA and GPU-SPRIM reduced models (40-port network)

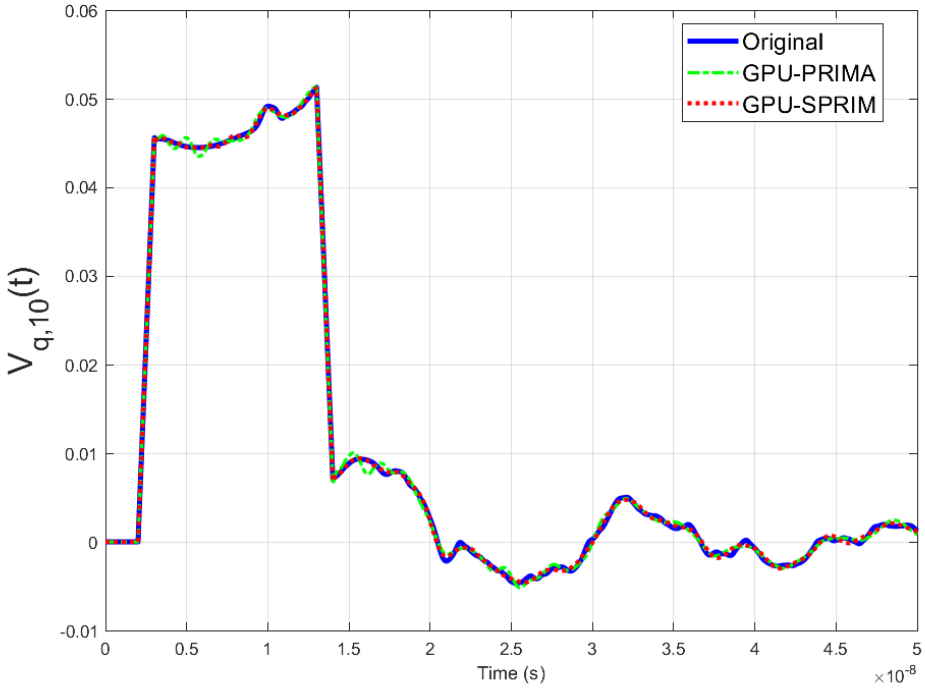


Figure 7.12: Comparison of $\mathbf{V}_{10}(t)$ voltage waveform from the original system, GPU-PRIMA and GPU-SPRIM reduced models (40-port network)

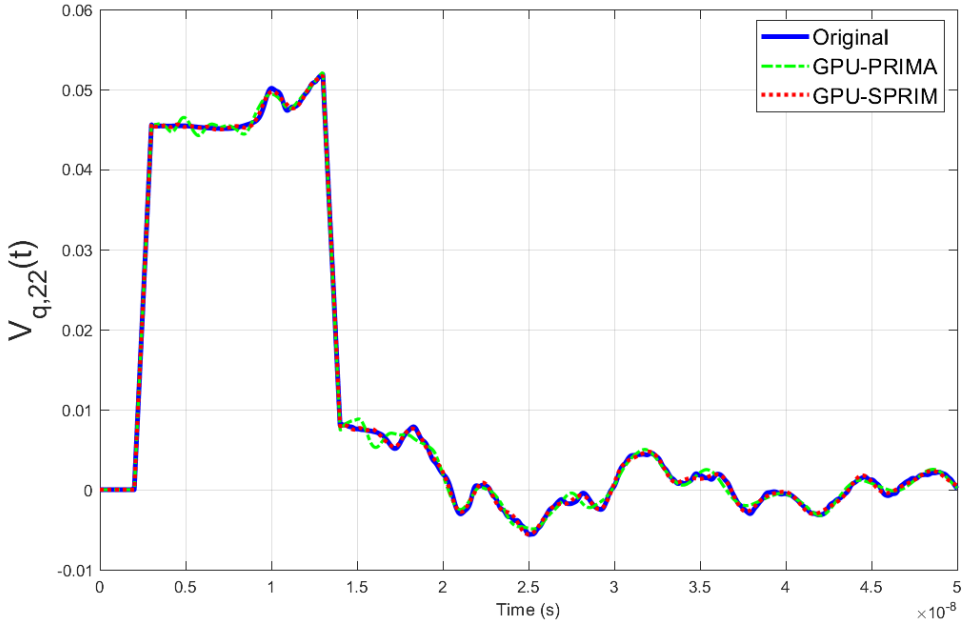


Figure 7.13: Comparison of $\mathbf{V}_{22}(t)$ voltage waveform from the original system, GPU-PRIMA and GPU-SPRIM reduced models (40-port network)

As can be seen from the responses, due to GPU-SPRIM matching $2q$ moments in comparison to GPU-PRIMA, the GPU-SPRIM reduced models exhibit better accuracy than that of the GPU-PRIMA models. The Frobenius norms for the error matrix averaged over the considered time points is given in Table 7.6. The CPU times to generate the voltage responses at the ports (in MATLAB) is given in Table 7.7.

Table 7.6: Average Frobenius norm of the voltage output errors over the time period 0–50 ns at 1000 points for GPU-PRIMA and GPU-SPRIM (40-port network)

Error	**GPU-PRIMA**	**GPU-SPRIM**
Frobenius norm $\overline{\alpha}(t)$	`2.652603e-02`	`4.787083e-03`

Table 7.7: Computation Time for Voltages at Ports

Time	**Original**	**GPU-PRIMA**	**GPU-SPRIM**
Total time taken	`3.36 s`	`0.51 s`	`1.86 s`

As can be seen from Table 7.7 due to the fact that GPU-SPRIM reduced models are twice as large compared to GPU-PRIMA, CPU time taken for simulation is much higher.

7.5 Summary

This chapter validated the performance of the proposed GPU-SPRIM framework through two multi-port RLC benchmarks (with $Np = 40$ and $Np = 128$). The results show that the proposed GPU-SPRIM yields reduced models whose frequency responses agree closely with those of the original large-scale circuits, while preserving the underlying block structures of the original system. At the same time, the GPU implementation significantly reduces the cost of the Arnoldi and projection stages, leading to an order-of-magnitude reduction in total runtime compared with single-core and multi-core SPRIM implementations. A comparison between PRIMA and SPRIM models reveals that SPRIM reduced models (while considering the same number of block moments) exhibit better accuracy as they are capable of matching $2q$ moments compared to PRIMA reduced models. However, SPRIM results in larger sized reduced

models losing efficiency during frequency and time-domain simulations. If the second-order reduced models from SPRIM are used, it ends up losing the advantages of the preserved block structures.

Chapter 8

Conclusion

This thesis proposed new parallel frameworks to exploit the emerging GPU platforms with Tensor Cores and strategies for end-to-end model order reduction based on Krylov subspaces. Particularly PRIMA and its structure-preserving alternative, SPRIM are considered in this thesis for parallelization. The frameworks were designed around minimizing CPU–GPU data communications and employing task-based parallelization to overlap distinct computational costs and/or combine computation with data transfers and writes. Custom kernels were developed to:

(a) Calculate batched frequency responses with efficient data packing for GPU-PRIMA.
(b) Use CUDA stream-based concurrency objects to efficiently overlap the computation of different tasks.
(c) Accelerate BLAS operations using GPU-based parallelization.
(d) Develop kernels with separate CUDA streams to orthogonalize Krylov vectors from the Block Arnoldi process in GPU-SPRIM, where the inner loops of the orthogonalization are parallelized for each stream.
(e) Batch and overlap computation with data write-outs for transfer function calculation in GPU-SPRIM.

Using the developed framework, the efficiency and accuracy of the proposed GPU-PRIMA and GPU-SPRIM algorithms are compared with their existing multi-core CPU approaches. Results using the proposed approaches demonstrate significant speedups compared to multi-core CPU-based approaches.

A comparison between PRIMA and SPRIM models reveals that SPRIM reduced models exhibit better accuracy (while considering same number of block moments) as they are capable of matching $2q$ moments compared to PRIMA reduced models. However, SPRIM results in larger sized reduced models losing efficiency during frequency and

time-domain simulations. If the second order reduced models from SPRIM are used, it ends up losing the advantages of the preserved block structures.

8.1 Future Work and Considerations

While traditional Krylov algorithms such as PRIMA [14] are well documented and widely adopted in industry-leading tools that address signal integrity, power distribution networks, RLCK parasitics, and interconnect modeling, the structure-preserving Krylov algorithms remain largely within academic research domains. More notably, SPRIM [16, 18], Block SAPOR [17, 34], and similar structure-preserving projection techniques have primarily been explored in academic contexts. These algorithms often produce larger reduced models and incur higher computational costs. Future research directions may include the following:

- Further sparsifying the reduced models for potential computational speed-ups [17, 35, 36].
- Utilizing second-order Krylov techniques or transforming the original DAE problems from first to second order for eigenvalue/pole analysis. SAPOR [17, 35] attempts this approach, although its limitation is that $\mathbf{G}$ matrices must exclude $\mathbf{E}_v$ stamps, meaning circuits cannot include voltage sources connected between nodes.
- Extending SAPOR and related Krylov subspace techniques into GPU-based high-performance computing frameworks.
- Investigating methods to enable SPRIM models, which retain their DAE structures, to be eigen-decomposed efficiently for fast pole-residue model formulation, as achieved in PRIMA systems.
- Revisiting the original motivation behind structure-preserving MOR algorithms, which aimed to synthesize real, reduced circuits. Currently, synthesizing PRIMA and SPRIM models often yields non-physical or non-realizable circuit elements, such as voltage- or current-dependent sources. These synthesized circuits function as black boxes, offering limited physical intuition to circuit designers. Prior work [34, 36], has attempted to create physically viable reduced circuits; however, resulting models frequently contain negative resistances (due to the "un-stamping" procedure) or require configurations where no inductors are connected to current source nodes. This ensures that port and inductor matrix blocks, $\mathbf{E}_I$ and $\mathbf{E}_l$, remain distinct (not sharing nodes) for diagonalization techniques to operate correctly. This can be explored further in a future research.

Bibliography

[1] C.-W. Ho, A. Ruehli, and P. Brennan, "The modified nodal approach to network analysis," *IEEE Transactions on Circuits and Systems*, vol. 22, no. 6, pp. 504–509, 1975.

[2] L. W. Nagel, "SPICE2: A computer program to simulate semiconductor circuits," Ph.D. dissertation, University of California, Berkeley, 1975, eRL Memo No. UCB/ERL M520.

[3] E. Bogatin, *Signal and power integrity–simplified*, 2nd ed., ser. Prentice Hall modern semiconductor design series. Upper Saddle River, NJ: Prentice Hall, 2010.

[4] R. Achar and M. Nakhla, "Simulation of high-speed interconnects," *Proceedings of the IEEE*, vol. 89, no. 5, pp. 693–728, 2001.

[5] W. C. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers," *Journal of Applied Physics*, vol. 19, no. 1, pp. 55–63, 01 1948.

[6] D. Boley, "Krylov space methods on state-space control models," *Circuits, Systems and Signal Processing*, vol. 13, pp. 733–758, 1994. [Online]. Available: https://api.semanticscholar.org/CorpusID:15006029

[7] E. Chiprout and M. Nakhla, "Analysis of interconnect networks using complex frequency hopping (CFH)," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 2, pp. 186–200, 1995.

[8] R. Freund and P. Feldmann, "Small-signal circuit analysis and sensitivity computations with the PVL algorithm," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 43, no. 8, pp. 577–585, 1996.

[9] R. Achar, "*Model-Reduction Techniques for High-Speed Interconnect Analysis*," Ph.D. dissertation, Carleton University, Ottawa, ON, Canada, 1998.

[10] B. Salimbahrami and B. Lohmann, "Krylov subspace methods in linear model order reduction: Introduction and invariance properties," 2002. [Online]. Available: https://api.semanticscholar.org/CorpusID:17762239

[11] L. Pillage and R. Rohrer, "Asymptotic waveform evaluation for timing analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 4, pp. 352–366, 1990.

[12] R. W. Freund, "Krylov-subspace methods for reduced-order modeling in circuit simulation," *Journal of Computational and Applied Mathematics*, vol. 123, no. 1, pp. 395–421, 2000, numerical Analysis 2000. Vol. III: Linear Algebra. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377042700003964

[13] L. Miguel Silveira, M. Kamon, I. Elfadel, and J. White, "A coordinate-transformed Arnoldi algorithm for generating guaranteed stable reduced-order models of RLC circuits," in *Proceedings of International Conference on Computer Aided Design*, 1996, pp. 288–294.

[14] A. Odabasioglu, M. Celik, and L. Pileggi, "PRIMA: passive reduced-order interconnect macromodeling algorithm," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 8, pp. 645–654, 1998.

[15] A. Elfadel and D. Ling, "A Block Rational Arnoldi algorithm for multipoint passive model-order reduction of multiport RLC networks," in *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, 1997, pp. 66–71.

[16] R. Freund, "SPRIM: structure-preserving reduced-order interconnect macromodeling," in *IEEE/ACM International Conference on Computer Aided Design*, 2004, pp. 80–87.

[17] Y. Su, J. Wang, X. Zeng, Z. Bai, C. Chiang, and D. Zhou, "SAPOR: Second-order Arnoldi method for passive order reduction of RCS circuits," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2004, pp. 74–79.

[18] R. W. Freund, *The SPRIM Algorithm for Structure-Preserving Order Reduction of General RCL Circuits*. Dordrecht: Springer Netherlands, 2011, pp. 25–52. [Online]. Available: https://doi.org/10.1007/978-94-007-0089-5_2

[19] Z. Bai, D. Hu, and L. Reichel, "A Newton basis GMRES implementation," *IMA Journal of Numerical Analysis*, vol. 14, no. 4, pp. 563–581, 10 1994. [Online]. Available: https://doi.org/10.1093/imanum/14.4.563

[20] V. Hernandez, J. Roman, and A. Tomas, "Parallel Arnoldi eigensolvers with enhanced scalability via global communications rearrangement," *Parallel Computing*, vol. 33, no. 7, pp. 521–540, 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819107000622

[21] M. F. Hoemmen, "*Communication-Avoiding Krylov Subspace Methods*," Ph.D. dissertation, University of California, Berkeley, 2010, technical Report No. UCB/EECS-2010-37, April 2, 2010.

[22] L. Chen and S. Petiton, "A TSQR based Krylov basis computation method on hybrid GPU cluster," in *2015 IEEE International Conference on Cluster Computing*, 2015, pp. 106–109.

[23] M. Berljafa and S. Güttel, "Parallelization of the Rational Arnoldi algorithm," *SIAM Journal on Scientific Computing*, vol. 39, no. 5, pp. S197–S221, 2017. [Online]. Available: https://doi.org/10.1137/16M1079178

[24] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. A. Yelick, "Avoiding communication in computing Krylov subspaces," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2007-123, October 2007.

[25] D. Luebke, "CUDA: Scalable parallel programming for high-performance scientific computing," in *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, 2008, pp. 836–838.

[26] R. Anando, A. Bhunia, and R. Achar, "GPU-based PRIMA algorithm for passive model order reduction," in *Proceedings of IEEE Electrical Design of Advanced Packaging and Systems (EDAPS)*, 2024, pp. 1–3.

[27] T. A. Davis and E. Palamadai Natarajan, "Algorithm 907: KLU, a direct sparse solver for circuit simulation problems," *ACM Trans. Math. Softw.*, vol. 37, no. 3, Sep. 2010. [Online]. Available: https://doi.org/10.1145/1824801.1824814

[28] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming*. Hoboken, NJ: John Wiley & Sons, 2014.

[29] D. B. Kirk and W. Hwu, *"Programming Massively Parallel Processors: A Hands-on Approach"*, 3rd ed. Morgan Kaufmann, 2016.

[30] NVIDIA Corporation, "*CUDA C++ Programming Guide*," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2025, accessed: 2025-12-05.

[31] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA tensor core programmability, performance precision," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 522–531.

[32] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5, pp. 232–240, 2010, parallel Matrix Algorithms and Applications. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819109001276

[33] J. Vlach and K. Singhal, *Computer Methods for Circuit Analysis and Design*, 2nd ed. USA: John Wiley & Sons, Inc., 1993.

[34] F. Yang, X. Zeng, Y. Su, and D. Zhou, "RLC equivalent circuit synthesis method for structure-preserved reduced-order model of interconnect in vlsi," *COMMUNICATIONS IN COMPUTATIONAL PHYSICS Commun. Comput. Phys. / Commun. Comput. Phys*, vol. 3, pp. 376–396, 02 2008.

[35] B. Liu, X. Zeng, Y. Su, J. Tao, Z. Bai, C. C. Chiang, and D. Zhou, "Block SAPOR: block second-order arnoldi method for passive order reduction of multi-input multi-output RCS interconnect circuits," in *Proceedings of the 2005 Conference on Asia South Pacific Design Automation, ASP-DAC 2005, Shanghai, China, January 18-21, 2005*, T. Tang, Ed. ACM Press, 2005, pp. 244–249. [Online]. Available: https://doi.org/10.1145/1120725.1120815

[36] T. S. Nguyen, T. L. Duc, S. T. Tran, J.-M. Guichon, and O. Chadebec, "Circuit realization method for reduced order inductive peec modeling circuits," *Compel-the International Journal for Computation and Mathematics in Electrical and Electronic Engineering*, vol. 35, pp. 1203–1217, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:112742057

[37] V. Kukutla, R. Achar, and W. Lee, "TC-GVF: Tensor core GPU-based vector fitting via accelerated tall-skinny qr solvers," *IEEE Transactions on Components, Packaging and Manufacturing Technology*, vol. 15, no. 1, pp. 54–63, 2025.

[38] N. Lindquist, P. Luszczek, and J. Dongarra, "Accelerating restarted GMRES with mixed precision arithmetic," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 1027–1037, Apr. 2022.

[39] J. A. Moríñigo, A. Bustos, and R. Mayo-García, "CPU and GPU performance of a restarted GMRES with randomized-SVD-based preconditioning," *The Journal of Supercomputing*, vol. 81, no. 16, p. 1563, 2025.

Appendix A

An Illustrative Circuit Example for PRIMA and SPRIM Formulation and Reduction

In this example, an illustrative circuit example is provided giving the details of the formulation and reduction using PRIMA and SPRIM algorithms.

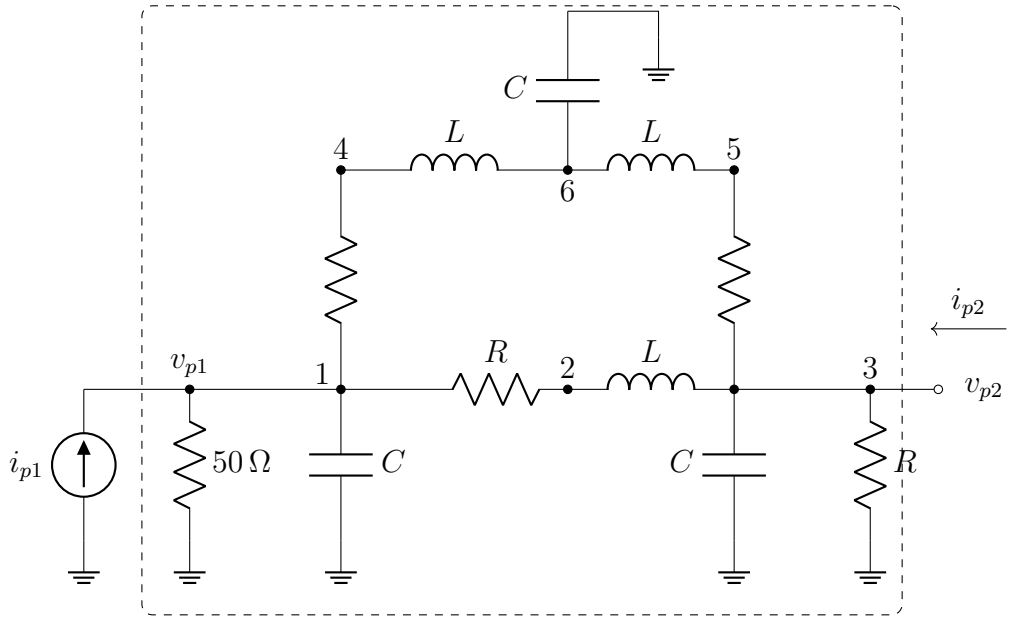


Figure A.1: An illustrative RLC circuit

The simple RLC ladder of Figure A.1 with $Np = 2$ ports is used as an example to illustrate the MNA formulation and the construction of PRIMA/SPRIM projectors. The circuit within the dashed box is reduced with PRIMA/SPRIM and the corresponding

G_q, C_q and B_q matrices are formed. In the illustration, the impedance matrix $\mathbf{Z}(s)$ of the multi-port circuit is modeled. The circuit can then be connected to external terminations and the reduced matrices can be used in the global MNA to carry out transient simulations. The parameters used for the circuit are:

$$R = 1.1719\text{e}{-3}\,\Omega, \qquad L = 4.5875\text{e}{-8}\,\text{H}, \qquad C = 1.8499\text{e}{-13}\,\text{F}.$$

The input current source i_{p1} is connected to node 1 through a parallel resistance of $50\,\Omega$.

A.1 MNA Stamps

The MNA matrices **G**, **C** and **B** are assembled by stamping the contributions of each element according to the MNA stamps in Figure A.2 [1, 14].

$$\underbrace{\left[\begin{array}{c|cc} & \text{i} & \text{j} \\ \hline \text{i} & +1/R & -1/R \\ \text{j} & -1/R & +1/R \end{array}\right]}_{\mathbf{G}}$$

(a) Resistor between nodes i and j

$$\underbrace{\left[\begin{array}{c|cc} & \text{i} & \text{j} \\ \hline \text{i} & +C & -C \\ \text{j} & -C & +C \end{array}\right]}_{\mathbf{C}}$$

(b) Capacitor between nodes i and j

$$\underbrace{\left[\begin{array}{c|ccc} & \text{i} & \text{j} & \text{k} \\ \hline \text{i} & 0 & 0 & +1 \\ \text{j} & 0 & 0 & -1 \\ \text{k} & -1 & +1 & 0 \end{array}\right]}_{\mathbf{G}} \quad \underbrace{\left[\begin{array}{c|c} & \text{k} \\ \hline \text{k} & L \end{array}\right]}_{\mathbf{C}}$$

(c) Modified inductor stamp between nodes i and j with unknown current flowing from node i to j

$$\underbrace{\left[\begin{array}{c|cc} & \text{i} & \text{j} \\ \hline \text{i} & 0 & 0 \\ \text{j} & 0 & 0 \end{array}\right]}_{\mathbf{G}} \quad \underbrace{\left[\begin{array}{c} -1 \\ \\ +1 \end{array}\right]}_{\mathbf{B}}$$

(d) Current flowing from node i to node j

Figure A.2: Element MNA stamps used to assemble the matrices **G**, **C** and **B**.

A.1.1 Stamping the MNA Matrices

Using these stamps to insert the numerical values of the elements in Figure A.1, the resulting MNA matrices are given below, (note that for inductor currents, the rows are negated prior to PRIMA and SPRIM reduction to preserve the semi-positive

definiteness of the resulting reduced matrices). The dashed regions show the inner block structures of the MNA matrices as in equation (3.4) in Chapter 3.

$$\mathbf{G} = \begin{bmatrix} \begin{bmatrix} 1.7067e3 & -853.3151 & 0 & 0 & -853.3151 & 0 \\ -853.3151 & 853.3151 & 0 & 0 & 0 & 0 \\ 0 & 0 & 853.3151 & -853.3151 & 0 & 0 \\ 0 & 0 & -853.3151 & 853.3151 & 0 & 0 \\ -853.3151 & 0 & 0 & 0 & 853.3151 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & -1 & -1 \end{bmatrix} \\ \begin{bmatrix} 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{bmatrix}.$$

$$\mathbf{C} = \begin{bmatrix} \begin{bmatrix} 3.6998e-13 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1.8499e-13 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.8499e-13 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 4.5875e-08 & 0 & 0 \\ 0 & 4.5875e-08 & 0 \\ 0 & 0 & 4.5875e-08 \end{bmatrix} \end{bmatrix}.$$

$$\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ i_{L1} \\ i_{L2} \\ i_{L3} \end{bmatrix}.$$

A.1.2 MNA Matrices at the Expansion Point

Once the MNA matrices are available the system matrices $\mathbf{A}$ and $\mathbf{R}$ are computed using (3.6) at an expansion point $s = 0$. $\mathbf{A}$ and $\mathbf{R}$ are correspondingly obtained as:

$$\mathbf{A} = \begin{bmatrix} -1.6820e-12 & 0 & -8.4085e-13 & 0 & 0 & -8.4092e-13 & 2.7803e-08 & 1.3901e-08 & -1.3901e-08 \\ -1.6817e-12 & 0 & -8.4086e-13 & 0 & 0 & -8.4086e-13 & 4.3095e-08 & -1.3901e-09 & 1.3901e-09 \\ -1.6817e-12 & 0 & -8.4086e-13 & 0 & 0 & -8.4086e-13 & -2.7803e-09 & -1.3901e-09 & 1.3901e-09 \\ -1.6818e-12 & 0 & -8.4086e-13 & 0 & 0 & -8.4100e-13 & 1.2511e-08 & -1.6682e-08 & 1.6682e-08 \\ -1.6818e-12 & 0 & -8.4086e-13 & 0 & 0 & -8.4100e-13 & 1.2511e-08 & 2.9193e-08 & -2.9193e-08 \\ -1.6818e-12 & 0 & -8.4086e-13 & 0 & 0 & -8.4100e-13 & 1.2511e-08 & -1.6682e-08 & -2.9193e-08 \\ -2.2423e-13 & 0 & 1.1211e-14 & 0 & 0 & -5.0451e-14 & -1.3049e-05 & 1.3048e-05 & -1.3048e-05 \\ -1.1211e-13 & 0 & 5.6057e-15 & 0 & 0 & 6.7269e-14 & 1.3048e-05 & -1.3049e-05 & 1.3049e-05 \\ 1.1211e-13 & 0 & -5.6057e-15 & 0 & 0 & 1.1772e-13 & -1.3048e-05 & 1.3049e-05 & -1.3049e-05 \end{bmatrix},$$

$$\mathbf{R} = \begin{bmatrix} 4.5461 & 4.5454 \\ 4.5454 & 4.5455 \\ 4.5454 & 4.5455 \\ 4.5457 & 4.5454 \\ 4.5457 & 4.5454 \\ 4.5457 & 4.5454 \\ 0.6061 & -0.0606 \\ 0.3030 & -0.0303 \\ -0.3030 & 0.0303 \end{bmatrix}.$$

A.2 PRIMA Projector via Block Arnoldi

The next step is to apply the (block) Arnoldi algorithm to the pair $(\mathbf{A}, \mathbf{R})$. The number of block moments matched is 2 (and hence $q = 2Np$). For this purpose, start by forming the first Krylov vector via the QR-decomposition of the $\mathbf{R}$ matrix:

$$\mathbf{X}_0 = \mathrm{qr}(\mathbf{R}) = \begin{bmatrix} -0.4074 & -0.0266 \\ -0.4073 & -0.0276 \\ -0.4073 & -0.0276 \\ -0.4073 & -0.0271 \\ -0.4073 & -0.0271 \\ -0.4073 & -0.0271 \\ -0.0543 & 0.8147 \\ -0.0272 & 0.4073 \\ 0.0272 & -0.4073 \end{bmatrix}.$$

Iteration 1. Next, a new Krylov *block* moment is generated by applying $\mathbf{A}$ and then orthogonalizing and normalizing with respect to the current block $\mathbf{X}_0$:

$$\widetilde{\mathbf{X}}_1 = -\,\mathbf{A}\,\mathbf{X}_0,$$

$$\mathbf{H}_{10} = \mathbf{X}_0^{\mathsf{T}}\widetilde{\mathbf{X}}_1,$$

$$\mathbf{W}_1 = \widetilde{\mathbf{X}}_1 - \mathbf{X}_0\,\mathbf{H}_{10},$$

$$\mathbf{W}_1 = \mathbf{X}_1\,\mathbf{H}_{11} \quad \text{(thin QR factorization)},$$

Here, $\mathbf{X}_1$ is the Q-factor obtained from the tall-skinny QR of $\mathbf{W}_1$ with $\mathbf{H}_{11}$ being the corresponding $\mathbf{R}$-factor.
The resulting $\mathbf{X}_1$ is:

$$\mathbf{X}_1 = \begin{bmatrix} -0.3790 & -3.3552e-04 \\ -0.3790 & -3.3556e-04 \\ 0.5307 & -5.2465e-04 \\ 0.5307 & 3.0397e-04 \\ -0.3790 & 4.9318e-04 \\ 0.0758 & 3.9858e-04 \\ 2.0730e-04 & 3.4867e-05 \\ 8.9891e-05 & -0.7071 \\ -1.1741e-04 & -0.7071 \end{bmatrix}.$$

In this illustration, the Arnoldi process is stopped after this first iteration (2 moments matched). The Arnoldi projector is then formed by collecting the block basis vectors. This is the Krylov projector (3.8) for split congruent transformation to be used in PRIMA:

$$\mathbf{X} = [\,\mathbf{X}_0\;\;\mathbf{X}_1\,] = \begin{bmatrix} -0.4074 & -0.0266 & -0.3790 & -3.3552e-04 \\ -0.4073 & -0.0276 & -0.3790 & -3.3556e-04 \\ -0.4073 & -0.0276 & 0.5307 & -5.2465e-04 \\ -0.4073 & -0.0271 & 0.5307 & 3.0397e-04 \\ -0.4073 & -0.0271 & -0.3790 & 4.9318e-04 \\ -0.4073 & -0.0271 & 0.0758 & 3.9858e-04 \\ -0.0543 & 0.8147 & 2.0730e-04 & 3.4867e-05 \\ -0.0272 & 0.4073 & 8.9891e-05 & -0.7071 \\ 0.0272 & -0.4073 & -1.1741e-04 & -0.7071 \end{bmatrix}.$$

The PRIMA reduced models (3.13) are given by

$$\mathbf{G}_q \;=\; \mathbf{X}^{\mathsf{T}}\mathbf{G}\,\mathbf{X}, \qquad \mathbf{C}_q \;=\; \mathbf{X}^{\mathsf{T}}\mathbf{C}\,\mathbf{X}, \qquad \mathbf{B}_q \;=\; \mathbf{X}^{\mathsf{T}}\mathbf{B},$$

with the numerical values

$$\mathbf{G}_q = \begin{bmatrix} 0.0365 & 0.0024 & -0.1143 & 6.0880e-05 \\ 0.0024 & 0.0013 & 1.1090 & -2.2805e-04 \\ 0.0340 & -1.1144 & 0.0592 & -5.2895e-05 \\ 3.0066e-05 & 2.3419e-04 & -5.3337e-05 & 0.0012 \end{bmatrix}.$$

$$\mathbf{C}_q = \begin{bmatrix} 2.0308e-10 & -3.0446e-09 & -7.6326e-13 & 8.2603e-16 \\ -3.0446e-09 & 4.5672e-08 & 1.1622e-11 & -1.1486e-14 \\ -7.6326e-13 & 1.1622e-11 & 1.0929e-13 & 8.9255e-13 \\ 8.2603e-16 & -1.1486e-14 & 8.9255e-13 & 4.5875e-08 \end{bmatrix}.$$

$$\mathbf{B}_q = \begin{bmatrix} -0.4082 & -0.4082 \\ -0.0022 & -0.0031 \\ -0.3790 & 0.5307 \\ -2.6915\text{e}{-04} & -6.1760\text{e}{-04} \end{bmatrix}.$$

Next, the Krylov-based projector generation process is modified for use in a SPRIM algorithm, and the details are given in the next section.

A.3 SPRIM Projector Assembly and Reduced Order Matrices

For SPRIM, the Krylov projector is partitioned across the rows according to the block structures of the original MNA matrices $\mathbf{G}$ and $\mathbf{C}$, as follows (3.17):

$$\mathbf{G} = \left[\begin{array}{cccccc|ccc} 1.7067\text{e}3 & -853.3151 & 0 & 0 & -853.3151 & 0 & 0 & 0 & 0 \\ -853.3151 & 853.3151 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 853.3151 & -853.3151 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & -853.3151 & 853.3151 & 0 & 0 & 0 & 0 & 1 \\ -853.3151 & 0 & 0 & 0 & 853.3151 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 \\ \hline 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 \end{array}\right] \begin{array}{l} \\ \\ \\ \\ \\ \\ \uparrow \\ \\ \downarrow \end{array} \begin{array}{l} \\ \\ \\ n_{\text{int}} \\ \\ \\ \\ \\ n_{\text{ind}} \end{array}$$

$$\mathbf{C} = \left[\begin{array}{cccccc|ccc} 3.6998e-13 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1.8499e-13 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.8499e-13 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 4.5875e-08 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4.5875e-08 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4.5875e-08 \end{array}\right] \begin{array}{l} \updownarrow n_{\text{int}} \\ \\ \\ \\ \\ \\ \updownarrow n_{\text{ind}} \\ \\ \end{array}$$

Here,

$$N_r \;=\; n_{\text{int}} \;+\; n_{\text{ind}}$$

with $n_{int} = 6$ corresponding to the number of *internal* nodes, $n_{ind} = 3$ corresponding to the circuit MNA *inductor current* variables.
Next, partition the $\mathbf{X}$ projector into two blocks $\mathbf{X}_1$ and $\mathbf{X}_2$ at the rows as indicated by the dashed box:

$$\mathbf{X} = \left[\begin{array}{cccc} -0.4074 & -0.0266 & -0.3790 & -3.3552e-04 \\ -0.4073 & -0.0276 & -0.3790 & -3.3556e-04 \\ -0.4073 & -0.0276 & 0.5307 & -5.2465e-04 \\ -0.4073 & -0.0271 & 0.5307 & 3.0397e-04 \\ -0.4073 & -0.0271 & -0.3790 & 4.9318e-04 \\ -0.4073 & -0.0271 & 0.0758 & 3.9858e-04 \\ \hline -0.0543 & 0.8147 & 2.0730e-04 & 3.4867e-05 \\ -0.0272 & 0.4073 & 8.9891e-05 & -0.7071 \\ 0.0272 & -0.4073 & -1.1741e-04 & -0.7071 \end{array}\right] \begin{array}{l} \longrightarrow \mathbf{X}_1 \in \mathbb{R}^{n_{\text{int}} \times q} \\ \\ \\ \\ \\ \\ \longrightarrow \mathbf{X}_2 \in \mathbb{R}^{n_{\text{ind}} \times q} \\ \\ \end{array}.$$

Running an economy QR decomposition on the partitioned blocks $\mathbf{X}_1$ and $\mathbf{X}_2$ separately, and assembling the two blocks into the SPRIM block diagonal projector we obtain:

$$\widetilde{\mathbf{X}} = \begin{bmatrix} -0.4083 & 0.6931 & 0.1692 & 0.5695 & 0 & 0 & 0 \\ -0.4082 & -0.4951 & 0.5639 & 0.1423 & 0 & 0 & 0 \\ -0.4082 & -0.4951 & -0.3947 & 0.4271 & 0 & 0 & 0 \\ -0.4082 & 0.0990 & -0.5921 & -0.2373 & 0 & 0 & 0 \\ -0.4082 & 0.0990 & 0.3665 & -0.5220 & 0 & 0 & 0 \\ -0.4082 & 0.0990 & -0.1128 & -0.3797 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -0.8165 & 0.5774 & -2.2024e-06 \\ 0 & 0 & 0 & 0 & -0.4082 & -0.5773 & 0.7071 \\ 0 & 0 & 0 & 0 & 0.4082 & 0.5774 & 0.7071 \end{bmatrix}.$$

The final first-order SPRIM reduced matrices are given by (3.18):

$$\mathbf{G}_q \;=\; \widetilde{\mathbf{X}}^{\mathsf{T}}\mathbf{G}\,\widetilde{\mathbf{X}}, \qquad \mathbf{C}_q \;=\; \widetilde{\mathbf{X}}^{\mathsf{T}}\mathbf{C}\,\widetilde{\mathbf{X}}, \qquad \mathbf{B}_q \;=\; \widetilde{\mathbf{X}}^{\mathsf{T}}\mathbf{B},$$

where,

$$\mathbf{G}_q = \begin{bmatrix} \begin{bmatrix} 0.0067 & -0.0113 & 0.0051 & -0.0116 \\ -0.0113 & 1.8070e03 & -600.2953 & 649.5704 \\ 0.0051 & -600.2953 & 199.4270 & -215.7907 \\ -0.0116 & 649.5704 & -215.7907 & 1.5491e03 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ -6.6481e-12 & -9.3125e-12 & -1.1466e-11 \\ -1.1740 & 3.0934e-12 & 3.8090e-12 \\ 0.3488 & -6.2438e-05 & 1.9629e-12 \end{bmatrix} \\ \begin{bmatrix} 0 & 6.6481e-12 & 1.1740 & -0.3488 \\ 0 & 9.3125e-12 & -3.0934e-12 & 6.2438e-05 \\ 0 & 1.1466e-11 & -3.8090e-12 & -1.9629e-12 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{bmatrix}$$

$$\mathbf{C}_q = \begin{bmatrix} \begin{bmatrix} 1.2333e-13 & -7.4778e-14 & 1.2775e-14 & -8.9605e-14 \\ -7.4778e-14 & 2.2489e-13 & 7.7464e-14 & 9.9964e-14 \\ 1.2775e-14 & 7.7464e-14 & 4.1762e-14 & 1.2374e-14 \\ -8.9605e-14 & 9.9964e-14 & 1.2374e-14 & 1.8041e-13 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 4.5875e-08 & -3.3087e-24 & 1.6544e-24 \\ -3.3087e-24 & 4.5875e-08 & -6.6174e-24 \\ 1.6544e-24 & -9.9262e-24 & 4.5875e-08 \end{bmatrix} \end{bmatrix}$$

$$\mathbf{B}_q = \begin{bmatrix} \begin{bmatrix} -0.4083 & -0.4082 \\ 0.6931 & -0.4951 \\ 0.1692 & -0.3947 \\ 0.5695 & 0.4271 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \end{bmatrix}.$$

As seen from the dashed box regions above, the inner block structures of the SPRIM reduced MNA matrices have been preserved according to Chapter 3 (3.20).

The corresponding second-order matrices according to equation (3.22) that can be used in equation (3.23) to generate the frequency response are obtained as:

$$\mathbf{N}_q = \begin{bmatrix} 0.0067 & -0.0113 & 0.0051 & -0.0116 \\ -0.0113 & 1.8070e03 & -600.2953 & 649.5704 \\ 0.0051 & -600.2953 & 199.4270 & -215.7907 \\ -0.0116 & 649.5704 & -215.7907 & 1.5491e03 \end{bmatrix},$$

$$\mathbf{Q}_q = \begin{bmatrix} 1.2333e-13 & -7.4778e-14 & 1.2775e-14 & -8.9605e-14 \\ -7.4778e-14 & 2.2489e-13 & 7.7464e-14 & 9.9964e-14 \\ 1.2775e-14 & 7.7464e-14 & 4.1762e-14 & 1.2374e-14 \\ -8.9605e-14 & 9.9964e-14 & 1.2374e-14 & 1.8041e-13 \end{bmatrix},$$

$$\mathbf{E}_{\ell,q} = \begin{bmatrix} 0 & -6.6481e-12 & -1.1740 & 0.3488 \\ 2.2766e-17 & -9.3125e-12 & 3.0934e-12 & -6.2438e-05 \\ 0 & -1.1466e-11 & 3.8090e-12 & 1.9629e-12 \end{bmatrix},$$

$$\boldsymbol{\mathcal{L}}_q = \begin{bmatrix} 4.5875e-08 & -3.3087e-24 & 1.6544e-24 \\ -3.3087e-24 & 4.5875e-08 & -6.6174e-24 \\ 1.6544e-24 & -9.9262e-24 & 4.5875e-08 \end{bmatrix},$$

$$\mathbf{B}_{v,q} = \begin{bmatrix} -0.4083 & -0.4082 \\ 0.6931 & -0.4951 \\ 0.1692 & -0.3947 \\ 0.5695 & 0.4271 \end{bmatrix}.$$