GVF: GPU based Vector Fitting

by

Srinidhi Ganeshan, B. E.

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements for the degree of
Master of Applied Sciences

Carleton Institute of Electrical and Computer Engineering
Department of Electronics
Faculty of Engineering
Carleton University
Ottawa, Ontario, Canada

© Srinidhi Ganeshan, 2019

The undersigned recommend to the Faculty of Graduate Studies and Research
acceptance of this thesis,

”GVF: GPU based Vector Fitting”

submitted by Srinidhi Ganeshan (B.E.) in partial fulfillment of the
requirements for the degree of Master of Applied Sciences

Thesis Supervisor
Professor Ram Achar

Chairman, Department of Electronics
Professor Niall Tait

Carleton University, Sep 2019

Abstract

Vector Fitting (VF) algorithm has been widely used for system identification from multiport tabulated data. Particularly, it is of high interest to the design community focused on modeling of high-speed modules such as large number of coupled interconnects, packaging structures and variety of electromagnetic modules. Since VF and strategies based on it require many iterations to arrive at optimal number of converged poles, it is highly desired to reduce the computational cost of each VF iteration. This thesis advances the applicability of VF by exploiting the emerging massively parallel graphical processing Units (GPUs). Necessary parallelization strategies suitable for GPU platforms are developed for large problem sizes (increasing number of poles and ports). Numerical results demonstrate that the proposed method provides better speedup compared to both the multi-CPU based VF as well as the existing GPU based parallel VF techniques.

Acknowledgements

I would like to express my gratitude to my thesis advisor Professor Ram Achar of the Electronics Department at Carleton University for his helpful suggestions. His motivations in difficult moments, patience and good judgement as well as proof reading my thesis made the journey through this thesis enjoyable and successful.

I am also thankful to Professor Wai Kong Lee of UTAR, Kampar, Malaysia, for his valuable inputs on GPU and CUDA programming which helped me to have a great start in this thesis.

I have special note of thanks to Naveen Kumar Elumalai. Without his encouragement and support, working on this thesis would have been hard.

I would also like to thank my father Ganeshan Loganathan, my mother Karpagam Ganeshan and my sister Mounisha Ganeshan for the continuous encouragement and constant support throughout my years of study.

Table of Contents

1	Introduction	11
1.1	Modeling via Tabulated Data	12
1.2	System Identification via Vector Fitting	13
1.3	Parallel Vector Fitting	14
1.4	Contributions	16
1.5	Organization of the Thesis	17
2	Overview of GPU Architectures	19
2.1	GPU Architectures	20
2.1.1	Terminologies in GPU architecture	20
2.2	Memory Organization in GPUs	21
2.2.1	Global memory	22
2.2.2	Constant memory	22
2.2.3	Texture memory	22
2.2.4	Shared memory	23
2.2.5	Registers and local memory	23
2.3	CUDA Programming Model	24
2.3.1	Memory allocation	25

2.3.2	Launching a kernel	26
2.4	NVIDIA cuBLAS	26
2.5	MAGMA	27
3	Overview of Vector Fitting	29
3.1	Pole Identification Stage	29
3.2	Selecting Initial Pole-set	37
3.3	Major Computational Cost Associated with the VF Process	38
3.4	Challenges Associated with Pole Computing for Multiport Tabu- lated Subnetworks	39
4	Review of Parallel VF using CPU and GPU	40
4.1	Overview of Formulation of Multiport VF Suitable for Parallel Computing [38]	40
4.1.1	Parallel VF using multi CPU Environment	43
4.1.2	Relaxed VF	45
4.2	Parallelization Strategies for multi CPU Environment	48
4.2.1	None Splitting Strategy	48
4.2.2	All Splitting Strategy	54
4.3	Parallelization Strategies for mixed multi CPU + GPU Environment	58
4.3.1	Approach-1: Formulation of A_l in CPU followed by im- mediate QR factorization and residue evaluation in GPU for <i>None Splitting Strategy</i>	58
4.3.2	Approach-2: Transfer of block of A_l s from CPU to GPU and parallel QR factorization in CPUs and GPU for <i>None</i> <i>Splitting Strategy</i> (single scaling function)	61

4.3.3	Approach-3: Transfer of block of A_l s from CPUs to GPU, parallel QR factorization and residue evaluation in CPUs as well as in GPU for <i>All Splitting Strategy</i> (individual scaling functions)	64
5	Development of GPU based Vector Fitting (GVF) with <i>None Splitting Strategy</i>	68
5.1	GVF based on None Splitting Strategy	69
5.2	Performance Comparison of the Proposed <i>GVF</i> with <i>None Splitting Strategy</i> on a Single GPU + CPU Platform versus PVF [38] on multiple CPUs	72
5.2.1	Data (S , g and α) transfer cost from CPU to GPU: <i>step (a)</i>	73
5.2.2	Formulation and transfer of $\hat{\Phi}$: <i>step (b)</i>	74
5.2.3	Formulation of A_l : <i>step (c)</i> , <i>step (d)</i>	74
5.2.4	QR factorization time in GPU using <i>MAGMA</i> [40] library: <i>step (e)</i>	75
5.2.5	R_l^{22} extraction: <i>step (f)</i>	75
5.2.6	Residue evaluation of the scaling function: <i>step (h)</i>	75
5.2.7	Computation of zeros of the scaling function $\hat{\sigma}(s)$: <i>step (i)</i>	76
5.3	QR factorization time in GPU using <i>cuBLAS</i> [41] library: <i>step (e)</i> .	78
5.4	Discussion on computational aspects and efficiency	91
6	Development of GPU based Vector Fitting (GVF) with <i>All Splitting Strategy</i>	92
6.1	GVF based on All Splitting Strategy	93

6.2	Performance comparison of the proposed <i>GVF</i> with <i>All Splitting Strategy</i> on a Single GPU + CPUs Platform versus PVF [38] on multiple CPUs	98
6.2.1	Data ($\mathbf{S}$, $\mathbf{g}$ and α) transfer cost from CPU to GPU: <i>step (a)</i>	99
6.2.2	Transfer of poles $p_{n,l}$: <i>step (b)</i>	100
6.2.3	Formulation of $\hat{\Phi}$: <i>step (c)</i>	100
6.2.4	Formulation of $\mathbf{A}_l$ and $\mathbf{b}_l$: <i>step (d)</i> , <i>step (e)</i> and <i>step (f)</i>	100
6.2.5	Computation time of $\mathbf{A}_l^T \times \mathbf{A}_l$, $\mathbf{A}_l^T \times \mathbf{b}_l$ using <i>MAGMA</i> [40] library: <i>step (g)</i>	101
6.2.6	LU decomposition, forward/backward substitution time in GPU using <i>MAGMA</i> [40] library: <i>step (h)</i>	101
6.2.7	Data transfer cost from GPU to CPU: <i>step (i)</i>	101
6.2.8	Computation of zeros of the scaling function $\hat{\sigma}_l(s)$: <i>step (h)</i>	102
6.3	Using <i>cuBLAS</i> [41] library for the <i>step (g)</i> and <i>step (h)</i>	103
6.4	Discussion on computational aspects and efficiency	115
7	Conclusion and Future work	116
7.1	Future work	117
	References	118

List of figures

2.1	CUDA Programming Model	21
2.2	CUDA Memory Architecture	24
4.1	Graphical illustration of <i>None Splitting Strategy</i> for multiport VF .	51
4.2	Graphical illustration of <i>All Splitting Strategy</i> for multiport VF . .	56
5.1	Flow diagram for the proposed <i>GVF</i> (for <i>None Splitting</i>) for one pole refinement iteration	70
5.2	CPU time comparison of the proposed <i>GVF</i> using <i>Approach (a)</i> with <i>cuBLAS</i> and <i>MAGMA</i> libraries versus PVF based on <i>Multi</i> <i>CPU approach (for None Splitting)</i>	80
5.3	Speed up comparison of the proposed <i>GVF</i> using <i>Approach (a)</i> with <i>cuBLAS</i> and <i>MAGMA</i> libraries versus PVF (<i>for None Splitting</i>)	81
5.4	CPU time comparison of the proposed <i>GVF</i> using <i>Approach (b)</i> with <i>cuBLAS</i> and <i>MAGMA</i> libraries versus PVF based on <i>Multi</i> <i>CPU approach (for None Splitting)</i>	82
5.5	Speed up comparison of the proposed <i>GVF</i> using <i>Approach (b)</i> with <i>cuBLAS</i> and <i>MAGMA</i> libraries versus PVF (<i>for None Splitting</i>)	82
5.6	Accuracy comparison of sample response for Example 1	83

5.7	CPU time comparison of the proposed <i>GVF</i> using <i>Approach (a)</i> with <i>cuBLAS</i> and <i>MAGMA</i> libraries versus PVF based on <i>Multi</i> <i>CPU approach (for None Splitting)</i>	87
5.8	Speed up comparison of the proposed <i>GVF</i> using <i>Approach (a)</i> with <i>cuBLAS</i> and <i>MAGMA</i> libraries versus PVF (<i>for None Splitting</i>)	88
5.9	CPU time comparison of the proposed <i>GVF</i> using <i>Approach (b)</i> with <i>cuBLAS</i> and <i>MAGMA</i> libraries versus PVF based on <i>Multi</i> <i>CPU approach (for None Splitting)</i>	89
5.10	Speed up comparison of the proposed <i>GVF</i> using <i>Approach (b)</i> with <i>cuBLAS</i> and <i>MAGMA</i> libraries versus PVF (<i>for None Splitting</i>)	89
5.11	Accuracy comparison of sample response for Example 2	90
6.1	Flow diagram for the proposed <i>GVF</i> with <i>All Splitting</i>	94
6.2	Flow diagram for the proposed <i>GVF</i> (<i>for All Splitting</i>)	104
6.3	CPU time comparison of the proposed <i>GVF</i> using <i>cuBLAS</i> and <i>MAGMA</i> libraries versus PVF based on <i>Multi CPU approach (for</i> <i>All Splitting)</i>	106
6.4	Speed up comparison of the proposed <i>GVF</i> using <i>cuBLAS</i> and <i>MAGMA</i> libraries versus PVF (<i>for All Splitting</i>)	107
6.5	Accuracy comparison of sample response for Example 1	108
6.6	CPU time comparison of the proposed <i>GVF</i> versus PVF based on <i>Multi CPU approach (for All Splitting)</i>	112
6.7	Speed up comparison of the proposed <i>GVF</i> using <i>cuBLAS</i> and <i>MAGMA</i> libraries versus PVF (<i>for All Splitting</i>)	112
6.8	Accuracy comparison of sample response for Example 2	114

List of Tables

5.1	Performance comparison of the proposed <i>GVF</i> using <i>MAGMA</i> library versus PVF based on multi CPU approach (for <i>None Splitting</i>)	73
5.2	Itemized cost for the proposed <i>GVF</i> using <i>MAGMA</i> library (for <i>None Splitting</i>)	77
5.3	Performance comparison of the proposed <i>GVF</i> (using <i>MAGMA</i> library and <i>Approach (b)</i>) versus PVF based on <i>Multi CPU approach</i> (for <i>None Splitting</i>)	77
5.4	Itemized cost for the proposed <i>GVF</i> using <i>cuBLAS</i> library (for <i>None Splitting</i>)	79
5.5	Performance comparison of the proposed <i>GVF</i> (using <i>cuBLAS</i> library) versus PVF [38] based <i>Multi CPU approach</i> (for <i>None Splitting</i>)	79
5.6	Itemized cost for the proposed <i>GVF</i> using <i>MAGMA</i> library (for <i>None Splitting</i>)	84
5.7	Performance comparison of the proposed <i>GVF</i> (using <i>MAGMA</i> library) versus PVF based on <i>Multi CPU approach</i> (for <i>None Splitting</i>)	85

5.8	Itemized cost for the proposed <i>GVF</i> using <i>cuBLAS</i> library (for <i>None Splitting</i>)	86
5.9	Performance comparison of the proposed <i>GVF</i> (using <i>cuBLAS</i> library) versus PVF based on <i>Multi CPU approach</i> (for <i>None Splitting</i>)	86
6.1	Performance comparison of the proposed <i>GVF</i> using <i>MAGMA</i> library versus PVF based on <i>Multi CPU approach</i> (for <i>All Splitting</i>)	99
6.2	Computational time for zeros computation in CPUs (for <i>All Splitting</i>)	102
6.3	Itemized cost for the proposed <i>GVF</i> using <i>MAGMA</i> library (for <i>All Splitting</i>)	103
6.4	Itemized cost for the proposed <i>GVF</i> using <i>cuBLAS</i> library (for <i>All Splitting</i>)	105
6.5	Performance comparison of the proposed <i>GVF</i> using <i>cuBLAS</i> library versus PVF based on <i>Multi CPU approach</i> (for <i>All Splitting</i>)	105
6.6	Itemized cost for the proposed <i>GVF</i> using <i>MAGMA</i> library (for <i>All Splitting</i>)	109
6.7	Computational time for zeros computation in CPUs (for <i>All Splitting</i>)	110
6.8	Performance comparison of the proposed <i>GVF</i> using <i>MAGMA</i> library versus PVF based on <i>Multi CPU approach</i> (for <i>All Splitting</i>)	110
6.9	Itemized cost for the proposed <i>GVF</i> using <i>cuBLAS</i> library (for <i>All Splitting</i>)	111
6.10	Performance comparison of the proposed <i>GVF</i> with <i>cuBLAS</i> library versus PVF based on <i>Multi CPU approach</i> (for <i>All Splitting</i>)	111

Chapter 1

Introduction

The rapid growth in VLSI technology coupled with the ever increasing quest for high-speed applications presents significant challenges for design and CAD techniques at both the chip and package levels. The timing and power requirements are becoming very stringent, demanding sharper excitations and denser layouts and low power consumption. This has highlighted the effects of interconnects such as signal delay, reflections and crosstalk, ringing, signal distortion, ground bounce and current distribution issues such as skin, edge and proximity effects [1] - [5]. In today's designs, interconnects are significantly responsible for most of the signal degradation in high-speed systems [6] - [9].

High-speed interconnect problems are not handled appropriately by conventional circuit simulators, such as SPICE [6]. These high-speed interconnect effects if not considered at the early design stage, may lead to failed and prolonged product cycles. Hence it becomes extremely important for designers to simulate the entire design along with interconnect sub-circuits in an efficient way while retaining the accuracy of simulation.

Interconnections can be from various levels of design hierarchy, such as on-chip, packaging structures, multichip modules printed boards and backplanes. Depending on the operating frequency and nature of the structure, these interconnects are typically modelled as *lumped components* (RC or RLC), *distributed elements* (fre-

quency dependent/dependent RLGC parameters, lossy, coupled) [1] - [3], [11].

1.1 Modeling via Tabulated Data

Owing to the complex nature of high-speed modules [1] - [3], the modelling and characterization based on tabulated data is gaining popularity in the recent years for signal integrity modeling and validations. The tabulated data representing the multiport parameters can be of various types, such as, admittance (Y), impedance (Z), scattering (S) and hybrid (T) parameters [12]. These data can be obtained from electromagnetic tools through full wave simulations [13] or measurements. Prominent high-speed modules that are characterized by tabulated data include high-speed interconnects, electronic packages, vias, connectors, non-uniform transmission lines and passive components, such as inductors and transformers. Time-domain simulation of multiport data is a computationally expensive process due to the mixed frequency/time problem as well as higher bandwidth needs.

Several approaches are reported in the literature to address these difficulties. A straightforward approach is based on first translating the frequency-domain data into time-domain by using IFFT and subsequently using the convolution with non-linear load characteristics and time-domain input to obtain the transient response [14] - [16]. However, such a process can be inefficient in the presence of large number of ports and large bandwidth requirements [3].

Alternative approaches include macromodeling based on model order reduction [17] - [21]. In this approach large linear circuit including the tabulated data is reduced in the frequency-domain to obtain a reduced-order model. Subsequently,

this reduced model is translated to the time-domain to obtain a macromodel in the form of ordinary differential equations [22] - [23] which is simulated along with the non-linear load to obtain the time-domain response. These time-domain simulations can also be accomplished using recursive convolution [24] - [25].

One of the major challenges associated with the macromodeling based approaches is preserving the passivity of the macromodel [26] - [32]. Lack of passivity can lead to spurious oscillations during transient simulations. A passive model cannot generate more energy than it is supplied with. A non-passive macromodel even though it is stable, when connected to other passive circuit, may lead to an overall unstable circuit. To ensure asymptotic stability of the transient simulations, a macromodel is not only required to be stable, but also passive. Several iterative methods can be found in the literature based on first-order perturbation techniques to restore the passivity of the macromodel [16] - [17].

Rational-function synthesis based least-square approximation approaches were used for system identification from tabulated data [13], [18], [20] - [22], [33] - [39]. One of the major difficulty in these approaches is the ill-conditioning encountered during high-order approximations for broadband modelling. This was successfully handled by the Vector Fitting (VF) approach for system identification [35] from tabulated data.

1.2 System Identification via Vector Fitting

Vector fitting algorithm [35] which was introduced nearly two decades ago, provided an efficient method for system identification via rational-function modeling of tabulated data. It overcame the ill-conditioning faced by the traditional least

squares based curve-fitting for rational-function modeling by assuming an initial set of poles. An iterative process is developed [35] using a partial fraction basis with the initial assumed set of poles representing the poles of the partial fraction. Based on this formulation and a scaling function with the same initial poles, residues of the scaling functions are evaluated. These residues are used to calculate the zeros of the scaling function, which are used as the new set of poles for the successive iterations until the poles converge. A major advantage of this method is the well-conditioned equations when initial poles are carefully chosen [36]. This allows for higher-order and broadband approximation, unlike the traditional least square approximation. If the number of poles selected is appropriate, then the VF algorithm converges in few iterations, if not, number of poles is changed and the iterations are executed again. Many improvements to VF since then can be found in the literature [36] - [39]. To improve the efficiency of VF, block QR-based approach was proposed in [37].

1.3 *Parallel Vector Fitting*

Parallelization of the VF algorithm using multi-CPU environment was proposed in [38] to reduce the computational cost involved in VF. Particularly, authors provided two important parallelization strategies: *None Splitting* and *All Splitting* Strategies.

- (a) *None Splitting Strategy*: This approach is based on computing a common pole set for the entire multiport matrix. In this approach, all VF steps are done in parallel using available CPUs (formulation and QR factorization of matrices corresponding to each S-parameter matrix elements). In the

last step of the iteration, the residues and zeros of the scaling function are computed in a single CPU (note that in this approach, only one scaling function is used for the entire matrix).

- (b) *All Splitting Strategy*: This approach is based on computing an individual pole set for each S-element of the multiport matrix. In this approach, all steps of the VF (formulation, QR factorization and computation of residues of the scaling function) are done in parallel across available CPUs. In this approach, individual scaling functions are assumed for individual S-elements.

In [39], use of the emerging platform of GPU was considered and parallelization strategies that are suitable for GPU + CPU platforms were proposed with the following approaches:

1. *Approach-1*: This approach is based on the *None Splitting Strategy* [38] for mixed CPU + GPU environment, in which the GPU does the QR factorization of the matrices corresponding to the individual S-elements that are supplied *sequentially* from CPU. Computation of the scaling function residues had to wait until all the QR factorization are computed.
2. *Approach-2*: This approach is also based on the *None Splitting Strategy*, wherein instead of supplying the matrices to GPU sequentially as in *Approach-1*, a certain percentage of the number of S-elements were sent in a block to GPU for parallel QR factorization. At the same time, the remaining S-elements are processed parallelly in the available CPUs. Residues of the scaling function are computed in a CPU after collecting all $\mathbf{R}$ factors.

3. *Approach-3*: This approach is based on the *All Splitting Strategy* [38] in which for part of the S-elements, GPU does the parallel QR factorization and evaluation of the residues of the individual scaling functions of matrices corresponding to a block of the S-elements. While at the same time, the remaining S-elements are processed in the available CPUs.

In all the above approaches, *cuBLAS* [41] routines were used for the QR factorization. The above approaches did not fully exploit the massively parallel architecture of GPUs and did not scale well with the increasing size of S-elements.

1.4 Contributions

To address the above difficulties, in this thesis, novel contributions are made to advance the applicability of VF to exploit the emerging massively parallel graphical processing units (GPUs) so as to significantly reduce the computational time. Necessary parallelization strategies suitable for GPU platforms are developed. Specifically, an algorithm *GVF: GPU based Vector Fitting* [53] is developed further advancing the concepts in [39] with the following new contributions in this thesis.

1. GVF based on *None Splitting Strategy* is proposed. In this algorithm, enhanced parallelization strategies for formulation of $\mathbf{A}$, QR factorization of the individual $\mathbf{A}_i$ elements, extraction of required subset of $\mathbf{R}$ matrices in GPU and computation of scaling function residues are developed.
2. GVF based on *All Splitting Strategy* is also developed. In this algorithm, strategies are developed for parallel formulation, QR factorization and the

residue of the scaling function in GPU for individual S-elements (note that individual scaling function are used for each of the multiport matrix elements).

3. A detailed computational cost analysis of each of the steps in VF iterations in both the above algorithms is performed. Also, a comparison of the *MAGMA* [40] and *cuBLAS* libraries [41] for the computational cost is made.

Particularly, compared to [18], strategies are developed in this thesis for most of the steps in VF including formulation, QR factorization and minimizing the data transfer costs. Also batched GPU codes are developed/used in this thesis which provide significant speedup compared to [39]. In addition, this thesis investigated the comprehensive use of *MAGMA* [40] libraries which provided significant speedup compared to using *cuBLAS* [41] in [39]. For large problem sizes (with increasing number of ports and poles) numerical results demonstrate that the proposed GPU based vector fitting (GVF) provides better speedup compared to the existing techniques.

1.5 Organization of the Thesis

The thesis is organized as follows. Chapter 2 presents a brief overview of GPU architecture and CUDA programming. In Chapter 3, an overview of system identification using vector fitting algorithm from a given set of tabulated data is presented. Chapter 4 presents an overview of parallelization algorithms for vector fitting using multi CPU environment and mixed multi CPU + GPU environment. Chapter 5 presents the development of the proposed *GVF* algorithm based on *None Splitting Strategy*. Chapter 6 presents the development of the proposed *GVF*

algorithm based on *All Splitting Strategy*. Conclusion and Future work possibilities are presented in Chapter 7.

Chapter 2

Overview of GPU Architectures

Early generations of CPUs were developed as a single core, meaning that only one operation could be performed at a time. With the significant advances in chip design and semiconductor fabrication, the computing market has evolved to multi-core CPUs which provide superior computing performance as they can perform several tasks in parallel. However, to solve many numerical problems in scientific applications, which require the solution of large linear algebraic system of equations, even the multicore CPUs become computationally expensive. This has led to the evolution of Graphics Processing Units (henceforth referred to as GPUs in this thesis) in computing applications.

Initially, GPUs were designed mainly to accelerate computation in graphics and video applications. The first GPU released in 1999, was used for graphics processing. Later, GPUs were enhanced to accelerate geometric calculations. Soon after the release of CUDA in 2006, a parallel computing architecture by Nvidia, GPUs were also adopted for mathematical computations by scientific community, with many innovative applications emerging from different areas. These include artificial intelligence [42], scientific simulation [43], [44], cryptography [45], integrated circuit analysis [46] and medical imaging [47], etc.

In this chapter, a brief overview of GPU architecture, memory hierarchy, concepts of CUDA programming model is given.

2.1 GPU Architectures

In this section, a brief review of the GPU architectures, including memory hierarchy and special features, that are relevant to the proposed work is given. GPU architectures have evolved through several generations [commonly referred to as compute capability (CC)], including Fermi (CC 2.x), Kepler (CC 3.x), Maxwell (CC 5.x), Pascal (CC 6.x) and recently Volta (CC 7.x). Volta is the codename for a GPU microarchitecture developed by Nvidia, succeeding Pascal (CC 6.x). Each new GPU architecture offers a better performance compared to the old one by advancing the hardware design, manufacturing process, and architectural changes. Besides that, new GPU architectures also provide advanced features that can be further leveraged for accelerating the implementation of many scientific algorithms.

2.1.1 Terminologies in GPU architecture

The CUDA programming model is a heterogeneous model in which both the CPU and GPU are used. In CUDA, the host refers to the CPU and its memory, while the device refers to the GPU and its memory. Host can manage memory on both the host and device, and also launches kernels which are functions executed on the device. These kernels are executed by many GPU threads in parallel.

A *thread* in the GPU is a basic element of the data to be processed. Unlike in the CPUs, a *thread* in a GPU is less light weighted. Several primitive '*threads*' will simultaneously execute a kernel program (i.e., parallel GPU program to be run on kernel). A *thread block* consists of batches of these primitive *threads*. A

block consists of multiple *threads*, up to a maximum size of 1024 *threads*. A *grid* is a collection of *thread blocks* of same thread dimensionality which all execute the same kernel. Also, *grid* consists of multiple blocks, up to a maximum of 1024 *blocks*. Also, the *threads* within a *block* are grouped into *warps*. *Warp* is a collection of 32 *threads* that share the same code and follow the same execution path with minimal divergences. A generic architecture of CUDA Programming model is described in Figure 2.1.

GRID

Figure 2.1: CUDA Programming Model

2.2 Memory Organization in GPUs

The memory hierarchy in GPU has several layers of memory and it is divided into on-chip memory and off-chip memory. The memories in the GPU chip are *registers*, *shared* and *texture memories*. The *global*, *local* and *constant memories*

reside off-chip. For the computation to happen in GPU, data is first transferred from CPU DRAM to GPU *global memory* (which is also a DRAM). It is to be noted that transfer of data from GPU to CPU happens only through their respective DRAM's.

2.2.1 *Global memory*

Global memory can be accessed and modified from both the host and the device. Note that the global memory resides off-chip. The data in the *global memory* is visible to any thread and block, which have the pointer to that region of the memory. For the computation to happen in GPU, data is transferred from CPU DRAM to GPU *global memory* and viceversa. *Global memory* allocations can persist for the lifetime of the program.

2.2.2 *Constant memory*

Constant memory is used only when data does not change over a course of kernel execution. Using *Constant memory* (i.e., cached *Global memory*) rather than *Global memory* will reduce the required memory bandwidth and improves the performance, as the warp of threads reads from the same location. It is a read only memory which is accessible by all threads and has the lifetime of the program.

2.2.3 *Texture memory*

Texture memory is a read only memory just like *Constant memory*. It is cached on chip, therefore, it provides higher effective bandwidth by reducing memory requests to off-chip DRAM (*Constant memory*) in some situations. Specifically,

texture cache is designed for graphics applications. *Texture memory* is used when a thread reads from an adjacent memory which can reduce memory traffic and increase performance compared to *global memory*.

2.2.4 *Shared memory*

In GPUs, *Shared memory* resides on-chip. It can be accessed by all threads in the same thread block. *Shared memory* provides a powerful feature for writing well optimized CUDA code. Access to *shared memory* is much faster than *global memory* as it resides on chip. It is allocated per thread block, and all the threads in the block have access to the same *shared memory*. *Shared memory* has a number of uses, such as user-managed data caches and high-performance cooperative parallel algorithms (parallel reductions).

2.2.5 *Registers and local memory*

Registers are very limited in size but are the fastest among the memories in GPU. They are only accessible locally by each thread. They reside on-chip. Data stored in the *register* lasts only for the duration of the lifetime of the thread.

If a kernel uses more *Registers* than its allowed limit, the compiler will allocate these extra register usages into “*local memory*” that resides off-chip in the slow *global memory* but cached at L1 cache for faster access.

Figure 2.2: CUDA Memory Architecture

2.3 CUDA Programming Model

Conventionally, GPU kernels are launched and managed by the CPU which can consume substantial portion of GPU resources. This also implies that workload (number of blocks and threads) to be offloaded to the GPU has to be determined *a priori*. Hence the workload needs to be estimated so that there is always sufficient tasks to keep the GPU occupied. Given the heterogeneous nature of the CUDA programming model, a typical sequence of operations for a CUDA C program is:

- (a) Declare and allocate host and device memory.
- (b) Initialize host data.
- (c) Transfer data from the host to the device.
- (d) Execute one or more kernels.
- (e) Transfer results from the device to the host.
- (f) Free the device and host memory

The GPU follows the single instruction multiple data execution model, wherein each thread is executing the same program while operating on different sets of data. Here, each thread and block can be indexed individually using the built-in variables *threadIdx* and *blockIdx*. Here, *blockDim* gives the total number of threads in a block. In the following sections, the terminologies of CUDA programming model is given.

2.3.1 Memory allocation

The device memory is allocated with the *cudaMalloc* function from the CUDA runtime API. The host and device in CUDA have separate memory spaces, both of which can be managed from host code. Next, the data is copied from host memory to the corresponding device memory using *cudaMemcpy*, which works just like the standard C *memcpy* function, except that it takes a fourth argument which specifies the direction of the copy.

cudaMemcpy(void dst, const void* src, size_t count, enum cudaMemcpyKind kind)*

After the kernel execution, the data is copied back from device memory to host memory using *cudaMemcpy*. After the execution of the program and copying the results back to the host, *cudaFree()* function is used, to free allocated device memory.

2.3.2 *Launching a kernel*

In the CUDA programming model when launching a kernel with a grid of thread blocks, it requires two arguments. The first argument in the execution configuration specifies the number of thread blocks in the grid, and the second specifies the number of threads in a thread block that are to be executed in parallel. In CUDA, we define kernels with any name using the *global* declaration specifier:

```
global KernelFunctionName < < < GridDim, ThreadBlockDim> > > (@params)
```

For numerical computations, the following two libraries are widely used in the industry: *cuBLAS* [41] from NVIDIA and *MAGMA* [40]. A brief discussion of both these libraries are given in the following sections.

2.4 *NVIDIA cuBLAS*

In this section, a brief overview of *cuBLAS* libraries is given. The *cuBLAS* library is an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA CUDA runtime. It allows users to access the computational resources of GPU. When using the *cuBLAS* functions API in the application, first, required matrices and vectors are allocated in the GPU memory space from the host. Next,

the data is copied from host memory to global memory (device). Finally, the required sequence of *cuBLAS* function is called for computation. After the kernel execution, the results are copied back from the GPU global memory space to the host CPU DRAM memory.

The *cuBLAS* library API is included using the header file *cublas_v2.h*. The handle to the *cuBLAS* library context is initialized using the function `cublasCreate()` and is explicitly passed to every subsequent library function call. Once the *cuBLAS* function finishes its computation in GPU, `cublasDestroy()` function is called to release the resources associated with the *cuBLAS* library. The error status `cublasStatus_t` is returned by all *cuBLAS* library function calls.

2.5 *MAGMA*

MAGMA stands for Matrix Algebra on GPU and Multi-core Architectures. *MAGMA* dense linear algebra library takes advantage of the heterogeneous/hybrid architectures of the CPU system and it combines the power of GPU systems with multi cores CPU to make the applications run faster and use memory efficiently. This computational advantage gives *MAGMA* library the edge over the *LAPACK* [45] and *cuBLAS* libraries in hybrid systems.

The memory management functions of *MAGMA* include `magma_malloc`, `magma_free`, `magma_setmatrix` and `magma_getmatrix`, which are briefly outlined below.

magma_malloc_cpu

The *magma_malloc_cpu* allocates desired size bytes on the CPU memory. The purpose of using the *magma_malloc_cpu* instead of *malloc* is to properly align arrays for vector instructions. *magma_free_cpu()* is used to free this CPU memory.

magma_malloc

The *magma_malloc* allocates memory on the GPU. While using this memory, CUDA imposes a memory synchronization. Also similar to *magma_malloc_cpu*, *magma_free()* is used to free the GPU memory.

magma_setmatrix and magma_getmatrix

Both the *magma_setmatrix* and *magma_getmatrix* work just like *cudaMemcpy* function (refer 2.2.1). *magma_setmatrix* copies data from host memory to device memory and *magma_getmatrix* copies the results back from device memory to host.

An overview of GPU architectures, its memory hierarchy and basic concepts of CUDA programming model is presented. Also, a brief discussion of GPU libraries *cuBLAS* [41] and *MAGMA* [40] is given.

Chapter 3

Overview of Vector Fitting

Vector fitting algorithm [35] which was introduced nearly two decades ago, provided an efficient method for system identification via rational-function modeling from the given tabulated data. It overcame the ill-conditioning faced by the traditional least square based curve-fitting for rational-function modeling by assuming an initial set of poles and iteratively refining the poles. If the number of poles selected is adequate then the VF iterations typically converge in few iterations, if not, number of poles is changed and VF iterations are executed again. In the next section, a brief overview of the steps in VF is presented.

3.1 Pole Identification Stage

Consider the rational-function in the form of pole residue formulation, as follows

$$f(s) = \sum_{n=1}^N \frac{\tilde{r}_n}{s - p_n} + d + se \quad (3.1)$$

here the residues ($\tilde{r}_n$) and poles (p_n) can be real or complex conjugate pairs, N is the total number of poles and residues, d and e are real. Evaluation of the unknown parameters in (3.1) can be formulated as a least square based nonlinear optimization problem [35], which is solved using the VF algorithm. For this purpose, an initial set of poles $\overline{p}_n$ is assumed and a scaling function $\sigma(s)$ is defined using the known poles $\overline{p}_n$ and unknown residues c_n as

$$\sigma(s) = \sum_{n=1}^N \frac{c_n}{s - \bar{p}_n} + 1 \quad (3.2)$$

Using (3.2), a scaled function is defined at all frequency samples $k \in \{1, 2, \dots, K\}$, as

$$\sigma(j\omega_k)f(s) \simeq \sum_{n=1}^N \frac{\tilde{c}_n}{s - \bar{p}_n} + d + se \quad (3.3)$$

where d and e are real. Substituting (3.2) in (3.3) we get

$$\left[\sum_{n=1}^N \frac{c_n}{s - \bar{p}_n} + 1 \right] f(s) = \sum_{n=1}^N \frac{\tilde{c}_n}{s - \bar{p}_n} + d + se \quad (3.4)$$

Here, $\bar{p}_n$ is the initial guess of poles and the number of unknowns is $N_t=2N+2$ (i.e., assuming N complex poles and N complex conjugate poles; d and e to be real)

$$\left[\sum_{n=1}^N \frac{c_n}{s - \bar{p}_n} f(s) + f(s) \right] = \sum_{n=1}^N \frac{\tilde{c}_n}{s - \bar{p}_n} + d + se \quad (3.5)$$

Rearranging (3.5)

$$f(s) = \sum_{n=1}^N \frac{\tilde{c}_n}{s - \bar{p}_n} + d + se - \left[\sum_{n=1}^N \frac{c_n}{s - \bar{p}_n} f(s) \right] \quad (3.6)$$

Expanding (3.6) for k^{th} frequency point,

$$\left[\frac{\tilde{c}_1}{s_k - \bar{p}_1} + \dots + \frac{\tilde{c}_N}{s_k - \bar{p}_N} + d + s_k e \right] - \left[\frac{c_1 f(s_k)}{s_k - \bar{p}_1} + \dots + \frac{c_N f(s_k)}{s_k - \bar{p}_N} \right] = f(s_k) \quad (3.7)$$

Next, expressing (3.7) for several frequency sample data $f(s_k)$ points, where $s_k = j\omega_k$, we have,

$$\left[\frac{\tilde{c}_1}{s_1 - \bar{p}_1} + \dots + \frac{\tilde{c}_N}{s_1 - \bar{p}_N} + d + s_1 e \right] - \left[\frac{c_1 f(s_1)}{s_1 - \bar{p}_1} + \dots + \frac{c_N f(s_1)}{s_1 - \bar{p}_N} \right] = f(s_1)$$

$$\begin{aligned}
& \left[\frac{\tilde{c}_1}{s_2 - \bar{p}_1} + \dots + \frac{\tilde{c}_N}{s_2 - \bar{p}_N} + d + s_2 e \right] - \left[\frac{c_1 f(s_2)}{s_2 - \bar{p}_1} + \dots + \frac{c_N f(s_2)}{s_2 - \bar{p}_N} \right] = f(s_2) \\
& \dots \\
& \left[\frac{\tilde{c}_1}{s_k - \bar{p}_1} + \dots + \frac{\tilde{c}_N}{s_k - \bar{p}_N} + d + s_k e \right] - \left[\frac{c_1 f(s_k)}{s_k - \bar{p}_1} + \dots + \frac{c_N f(s_k)}{s_k - \bar{p}_N} \right] = f(s_k)
\end{aligned} \tag{3.8}$$

Equation (3.8) can be represented as a set of algebraic equations of the form,

$$\mathbf{A}(s)\mathbf{X}(s) = \mathbf{b}(s) \tag{3.9}$$

where $\mathbf{b}(s) = [f(s_1) f(s_2) f(s_3) \dots f(s_k)]^T$. The matrices $\mathbf{A}(s)$ and vector $\mathbf{X}(s)$ are expressed for the case of real poles or complex poles, as follows.

Case1: Real Poles

In case of real poles, writing (3.8) at several frequency points results in the following set of matrix equations

$$\underbrace{\begin{bmatrix} \frac{1}{s_1 - \bar{p}_1} & \dots & \frac{1}{s_1 - \bar{p}_N} & 1 & s_1 & \frac{-f(s_1)}{s_1 - \bar{p}_1} & \dots & \frac{-f(s_1)}{s_1 - \bar{p}_N} \\ \frac{1}{s_2 - \bar{p}_1} & \dots & \frac{1}{s_2 - \bar{p}_N} & 1 & s_2 & \frac{-f(s_2)}{s_2 - \bar{p}_1} & \dots & \frac{-f(s_2)}{s_2 - \bar{p}_N} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{1}{s_k - \bar{p}_1} & \dots & \frac{1}{s_k - \bar{p}_N} & 1 & s_k & \frac{-f(s_k)}{s_k - \bar{p}_1} & \dots & \frac{-f(s_k)}{s_k - \bar{p}_N} \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} \tilde{c}_1 \\ \vdots \\ \tilde{c}_N \\ d \\ e \\ c_1 \\ \vdots \\ c_N \end{bmatrix}}_{\mathbf{X}} = \underbrace{\begin{bmatrix} f(s_1) \\ f(s_2) \\ \vdots \\ f(s_k) \end{bmatrix}}_{\mathbf{b}} \tag{3.10}$$

For real poles, the unknowns in vector $\mathbf{X}$ are real.

Case2: Complex Poles

In the case of complex poles, since complex poles exist in pairs (i.e., with the complex conjugate poles), residues also exist in complex-conjugate pairs. Consider (3.4) with complex poles along with their conjugates, represented as

$$\bar{p}_n = \text{Re}(\bar{p}_n) + j\text{Im}(\bar{p}_n) ; \bar{p}_n^* = \text{Re}(\bar{p}_n) - j\text{Im}(\bar{p}_n) \quad (3.11)$$

where $\bar{p}_n^*$ is the complex conjugate pole and * indicates the complex conjugate operator. Therefore, residues of the scaled function $\sigma(j\omega_k)f(s)$ for complex-conjugate pairs is represented by

$$\tilde{c}_n = \text{Re}(\tilde{c}_n) + j\text{Im}(\tilde{c}_n) ; \tilde{c}_n^* = \text{Re}(\tilde{c}_n) - j\text{Im}(\tilde{c}_n) \quad (3.12)$$

and that of the scaling function by

$$c_n = \text{Re}(c_n) + j\text{Im}(c_n) ; c_n^* = \text{Re}(c_n) - j\text{Im}(c_n) \quad (3.13)$$

Using the above representation for the pole pair, (3.7) can be expressed for a single pole-pair as

$$f(s_k) = \frac{\tilde{c}_1}{s_k - \bar{p}_1} + \frac{\tilde{c}_1^*}{s_k - \bar{p}_1^*} + d + s_k e - f(s_k) \frac{c_1}{s_k - \bar{p}_1} - f(s_k) \frac{c_1^*}{s_k - \bar{p}_1^*} \quad (3.14)$$

Substituting (3.12), (3.13) in (3.14), we get

$$\begin{aligned} & \left[\frac{\text{Re}(\tilde{c}_1)}{s_k - \bar{p}_1} + \frac{\text{Re}(\tilde{c}_1)}{s_k - \bar{p}_1^*} + \frac{j\text{Im}(\tilde{c}_1)}{s_k - \bar{p}_1} - \frac{j\text{Im}(\tilde{c}_1)}{s_k - \bar{p}_1^*} + d + s_k e \right] - \\ & f(s_k) \left[\frac{\text{Re}(c_1)}{s_k - \bar{p}_1} + \frac{\text{Re}(c_1)}{s_k - \bar{p}_1^*} + \frac{j\text{Im}(c_1)}{s_k - \bar{p}_1} - \frac{j\text{Im}(c_1)}{s_k - \bar{p}_1^*} \right] = f(s_k) \quad (3.15) \end{aligned}$$

Expressing (3.15) for k different frequency points, resulting equations can be written in the form $\mathbf{A}\mathbf{X} = \mathbf{b}$. Next, both $\mathbf{A}$ and $\mathbf{b}$ are split into real and imaginary parts as

$$\begin{bmatrix} \text{Re}(\mathbf{A}) \\ \text{Im}(\mathbf{A}) \end{bmatrix} \mathbf{X} = \begin{bmatrix} \text{Re}(\mathbf{b}) \\ \text{Im}(\mathbf{b}) \end{bmatrix} \quad (3.16)$$

or

$$\hat{\mathbf{A}}\mathbf{X} = \hat{\mathbf{b}} \quad (3.17)$$

The over-determined problem of (3.17) is solved using least-square technique or QR factorization. In the case of least square method, pre-multiplying both sides of (3.19) by $\hat{\mathbf{A}}^T$, we get,

$$\hat{\mathbf{A}}^T \hat{\mathbf{A}}\mathbf{X} = \hat{\mathbf{A}}^T \hat{\mathbf{b}} \quad (3.18)$$

or

$$\mathbf{X} = (\hat{\mathbf{A}}^T \hat{\mathbf{A}})^{-1} \hat{\mathbf{A}}^T \hat{\mathbf{b}} \quad (3.19)$$

Instead of a direct inversion (which can be numerically ill-conditioned), QR factorization is often used for solution of $\mathbf{X}$ in (3.19). Writing $\hat{\mathbf{A}}$ in terms of $\mathbf{Q}$ and $\mathbf{R}$ factors, where $\mathbf{Q}$ is the orthogonal matrix and $\mathbf{R}$ is upper triangular matrix, we get,

$$\mathbf{Q}\mathbf{R}\mathbf{X} = \hat{\mathbf{b}} \quad (3.20)$$

Notify that $\mathbf{Q}^{-1} = \mathbf{Q}^T$, to evaluate $\mathbf{X}$, we have

$$\mathbf{R}\mathbf{X} = \mathbf{Q}^T \hat{\mathbf{b}} \quad (3.21)$$

$$\mathbf{R}\mathbf{X} = \tilde{\mathbf{b}} \quad (3.22)$$

where

$$\mathbf{Q}^T \hat{\mathbf{b}} = \tilde{\mathbf{b}} \quad (3.23)$$

Note that $\mathbf{R}$ is an upper triangular matrix, $\mathbf{X}$ can be solved for using back substitution in (3.22). The above steps can be used to obtain the residues of the scaling function $\sigma(s)$. Once the unknowns $c_1, \dots, c_n$ are obtained by solving (3.22), zeros of $\sigma(s)$ can be calculated efficiently as the eigenvalues of matrix $\mathbf{H}$, where $\mathbf{H}$ is given by,

$$\mathbf{H} = \mathbf{A} - \mathbf{B}\mathbf{C}^T D^{-1} \quad (3.24)$$

Here D is a real number (i.e., it takes the value 1 in case of *Basic VF* and any real value in case of *Relaxed VF*) and the entities $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ can be expressed for real poles case and complex poles case as follows.

Case 1: Real Poles

In case of real poles, $\mathbf{H}$ matrix of (3.24) can be written as,

$$\mathbf{A} = \begin{bmatrix} \bar{p}_1 & & & \\ & \ddots & & \\ & & \bar{p}_n & \\ & & & \ddots \\ & & & & \bar{p}_N \end{bmatrix}_{N \times N} \quad (3.25)$$

$$\mathbf{B} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}_{N \times 1} \quad (3.26)$$

$$\mathbf{C} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{bmatrix}_{N \times 1} \quad (3.27)$$

$$D = 1 \quad (3.38)$$

where $\mathbf{A}$ is the diagonal matrix of the starting poles, $\mathbf{B}$ is a unit column vector, $\mathbf{C}$ is the column vector of residues of $\sigma(s)$, corresponds to the starting poles computed in (3.24) of size $1 \times N$, where N is the number of poles.

Case 2: Complex Poles

In case of complex poles, $\mathbf{A}$ in (3.27) can be defined as follows,

$$\mathbf{A} = \begin{bmatrix} \psi_1 & & & \\ & \ddots & & \\ & & \psi_n & \\ & & & \ddots \\ & & & & \psi_N \end{bmatrix}_{N \times N} \quad (3.29)$$

where ψ_n is represented by

$$\psi_n = \begin{pmatrix} Re(\bar{p}_n) & Im(\bar{p}_n) \\ -Im(\bar{p}_n) & Re(\bar{p}_n) \end{pmatrix} \quad (3.30)$$

$$\mathbf{C} = \begin{pmatrix} Re(c_1) \\ Im(c_1) \\ \vdots \\ Re(c_N) \\ Im(c_N) \end{pmatrix}_{2N \times 1} \quad (3.31)$$

$$\mathbf{B} = \begin{bmatrix} 2 \\ 0 \\ \vdots \\ 2 \\ 0 \end{bmatrix}_{2N \times 1} \quad (3.32)$$

Using the above steps, the efficiently computed zeros of the scaling function $\sigma(s)$, represent the new improved poles of $f(s)$. To demonstrate this, the scaling function in (3.2) and scaled function in (3.3) are expressed in the pole-zero form as

$$\sigma(s) = \frac{\prod_{n=1}^N (s - z_n)}{\prod_{n=1}^N (s - \bar{p}_n)} \quad (3.33)$$

and

$$\sigma(j\omega_k)f(s) = \frac{\prod_{n=1}^N (s - \tilde{z}_n)}{\prod_{n=1}^N (s - \bar{p}_n)} \quad (3.34)$$

Also from (3.4), it is evident that

$$f(s) = \frac{\sigma(j\omega_k)f(s)}{\sigma(s)} \simeq \frac{\prod_{n=1}^N (s - \tilde{z}_n)}{\prod_{n=1}^N (s - z_n)} \quad (3.35)$$

From (3.34) and (3.35), it is clear that zeros of $\sigma(s)$ indeed represent the poles of $f(s)$. Note that the initial poles ($\bar{p}_n$) cancel in the division process as we started

with the same initial poles for $\sigma(s)$ and $\sigma(j\omega_k)f(s)$. Thus, by calculating zeros of $\sigma(s)$, we get a better set of poles than the initial set of poles $\overline{p_n}$, for fitting the original function $f(s)$.

Once the zeros of $\sigma(s)$ are calculated, they are used as new set of starting poles in the next iteration while solving (3.17). The next iteration provides a new set of zeros of $\sigma(s)$ which is again used in its next iteration as starting poles. The iterative process is continued until the poles converge. It is to be noted that during the iterative process of vector fitting, some of the calculated poles may become unstable (i.e., the real part of some of the poles can become positive). This problem is overcome by inverting the sign of the real part of the pole [20] and re-executing the VF iterations.

After the initial poles ($\overline{p_n}$) converge in few iterations to the dominant poles of the system, which is referred to as p_n in (3.1), the final residues for the system $\tilde{r}_n$ is determined, using an additional least square approximation of the system in (3.1).

3.2 *Selecting Initial Pole-set*

Successful application of VF requires judicious selection of initial set of poles. The following guidelines are used while selecting the initial set of poles [18]. For functions with distinct resonance peaks, starting poles are selected in complex conjugate pairs with the imaginary parts $\beta_1, \beta_2, \dots, \beta_n$, linearly distributed over the frequency range of interest $[\omega_{min} \text{ to } \omega_{max}]$. Each pole-pair is selected as follows,

$$p = -\alpha \pm j\beta \quad (3.36)$$

with

$$\alpha = \frac{\beta}{100} \quad (3.37)$$

For smooth responses, real poles are selected linearly or logarithmically spaced over the frequency of interest.

3.3 Major Computational Cost Associated with the VF Process

In VF process [35], major computational cost is with regards to the QR factorization in every pole refinement iteration. The number of floating-point operations (flops) for QR factorization of matrix $\mathbf{A}$ of size m by n (for a single port) [38], [44] is given below,

$$FL_{QR}(m, n) = \begin{cases} 2m^2n - \frac{2}{3}m^3 & \text{if } n \geq m \\ 2n^2m - \frac{2}{3}n^3 & \text{if } n < m \end{cases} \quad (3.38)$$

The computational cost involved to solve (3.22) is $FL_{QR}(m, n)$ and the cost to solve, the right hand side of (3.23) is $2mnp$ flops, where $\mathbf{Q}$ matrix has the size $m \times n$ and $\hat{\mathbf{b}}$ has the size $n \times p$. Equation (3.24) requires n_p^2 flops, where $\mathbf{R}$ matrix is of the size $n \times n$.

3.4 Challenges Associated with Pole Computing for Multiport Tabulated Sub-networks

Consider a P -port measured network and its Scattering matrix

$$\mathbf{S}(s) = \begin{bmatrix} S_{11} & S_{12} & \dots & S_{1P} \\ \vdots & \vdots & \ddots & \vdots \\ S_{P1} & S_{P2} & \dots & S_{PP} \end{bmatrix} \quad (3.41)$$

Conventionally, each element of the above scattering matrix is approximated by a rational-function. To find a common pole-set, the poles corresponding to individual elements of (3.41) are collected together. However, this approach can lead to large number of redundant poles in the macromodel resulting in an inefficient transient simulation. In addition, identifying a common multi-port pole set from pole-sets of individual elements of the admittance matrix is generally heuristic and can lead to errors. Also, when the number of ports increases then VF becomes CPU expensive.

In summary, this chapter provided an overview of the computational steps associated with the VF algorithm. It also outlined the challenges associated with multiport networks when using VF.

Chapter 4

Review of Parallel VF using CPU and GPU

In Chapter 3, a brief review of the Vector Fitting (VF) algorithm [35] which has been very widely used for system identification from multiport tabulated data (simulated or measured) of linear multiport networks was presented. If the number of poles selected is adequate, then the VF iterations typically converge in few iterations, if not, number of poles is changed and VF iterations are executed again. This problem gets aggravated in the presence of large number of ports, data requiring large number of poles and need for inclusion of large number of frequency data samples, making the VF iterations slower due to excessive computational requirements. To address these issues and to improve the efficiency of VF, parallel VF based on both the multi CPUs [38] as well as GPUs [39] have been recently reported. A brief review of the same is given in this chapter.

4.1 Overview of Formulation of Multiport VF Suitable for Parallel Computing [38]

Parallelization of the vector fitting algorithm using multiple CPUs compared to executing on a single core/CPU was proposed in [38] which reduced the computational cost significantly. Particularly, authors provided two important parallelization strategies: *None Splitting Strategy* based on computing a common pole set for the entire multiport matrix and *All Splitting Strategy* based on computing

an individual pole set for each element of the multiport matrix. These strategies are briefly reviewed in the following sections.

Consider a given set of frequency data samples, $S_{ij,k} = S_{ij}(j\omega_k)$, with $\forall k \in \{1, 2, \dots, K\}$, where K denotes the total number of frequency samples, and $\forall i, j \in \{1, 2, \dots, P\}$, where P denotes the number of ports, as an input to the VF algorithm. Given multiport data can be approximated in a rational form as a sum of partial functions represented as

$$\tilde{\mathbf{S}}(s) = \begin{bmatrix} \tilde{S}_{11} & \tilde{S}_{12} & \dots & \tilde{S}_{1P} \\ \tilde{S}_{21} & \tilde{S}_{22} & \dots & \tilde{S}_{2P} \\ \vdots & \vdots & \ddots & \vdots \\ \tilde{S}_{P1} & \tilde{S}_{P2} & \dots & \tilde{S}_{PP} \end{bmatrix} \quad (4.1)$$

with each element of the multiport data represented as

$$\tilde{S}_{ij}(s) = \sum_{n=1}^N \frac{\tilde{r}_{ij,n}}{s - p_n} + d + se \quad (4.2)$$

where, N denotes the number of poles. Evaluation of the unknown parameters in (4.2) can be formulated as a least square based nonlinear optimization problem [35], which is solved using the VF algorithm. For this purpose, an initial set of poles $\overline{p_n}$ is assumed and a scaling function $\sigma(s)$ is defined using the known poles $\overline{p_n}$ and unknown residues c_n as

$$\sigma(s) = \sum_{n=1}^N \frac{c_n}{s - \overline{p_n}} + 1 \quad (4.3)$$

Using (4.3), a scaled function is defined for all multiport responses $S_{ij,k}$ and at all

frequency samples $k \in \{1, 2, \dots, K\}$, as

$$\sigma(j\omega_k)S_{ij,k} \simeq \sum_{n=1}^N \frac{\tilde{c}_{ij,n}}{s - \bar{p}_n} + d + se \quad (4.4)$$

where d and e are real. The solution of (4.4) gives the residues c_n of the scaling function and the residues $\tilde{c}_{ij,n}$. Next, the zeros of $\sigma(s)$ can be found by computing the eigenvalues of the following system

$$z_n \leftarrow \text{eig}(\mathbf{A}_\sigma - \mathbf{B}_\sigma \mathbf{C}_\sigma \mathbf{D}_\sigma^{-1}) \quad (4.5)$$

where $\mathbf{A}_\sigma, \mathbf{B}_\sigma, \mathbf{C}_\sigma, \mathbf{D}_\sigma$ is a minimal state-space realization of $\sigma(s)$ which can be obtained using $\bar{p}_n$ and c_n using (3.25 - 3.28) for the real poles and using (3.29 - 3.32) for the complex poles. During the subsequent VF iterations, starting poles of $\sigma(s)$ are replaced with their zeros (i.e., $\bar{p}_n \leftarrow z_n$) and the process is repeated until convergence. Initial poles ($\bar{p}_n$) usually converge in few iterations to the dominant poles of the system which is referred to as p_n in (4.2).

Once the poles p_n have been found, using the form in (4.2) to determine the final residues for the multiport macromodel $\tilde{r}_{ij,n}$.

4.1.1 Parallel VF using multi CPU Environment

For the case of multiport networks (4.4) can be assembled in a matrix form [38] as

$$\begin{bmatrix} \Phi_1 & 0 & \dots & \dots & 0 & \tilde{\Phi}_1 \\ 0 & \ddots & \dots & \dots & \vdots & \vdots \\ \vdots & \vdots & \Phi_l & \dots & 0 & \tilde{\Phi}_l \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & 0 & \dots & \Phi_{P^2} & \tilde{\Phi}_{P^2} \end{bmatrix} \begin{bmatrix} \tilde{c}_1 \\ \vdots \\ \tilde{c} \\ \vdots \\ \tilde{c}_{P^2} \\ c \end{bmatrix} = \begin{bmatrix} S_1 \\ \vdots \\ S_l \\ \vdots \\ S_{P^2} \end{bmatrix} \quad (4.6)$$

where variable l spans the multiport space, $l = \{1, 2, \dots, P^2\}$ and is expressed in terms of $\forall i, j \in \{1, 2, \dots, P\}$ and P as

$$l = (i - 1)P + j \quad (4.7)$$

and

$$S_{l_{(K \times 1)}} = S_{ij} = \begin{bmatrix} S_{ij,1} & S_{ij,2} & \dots & S_{ij,K} \end{bmatrix}^T \quad (4.8)$$

$$\tilde{c}_{l_{(N+2) \times 1}} = \tilde{c}_{ij} = \begin{bmatrix} \tilde{c}_{ij,1} & \tilde{c}_{ij,2} & \dots & \tilde{c}_{ij,N} & d & e \end{bmatrix}^T \quad (4.9)$$

$$\Phi_{l_{K \times (N+2)}} = \begin{bmatrix} \hat{\Phi} & \mathbf{1}_K & \mathbf{g} \end{bmatrix} \quad (4.10)$$

$$\mathbf{1}_{K_{(K \times 1)}} = \begin{bmatrix} 1 & \dots & 1 \end{bmatrix}^T \quad (4.11)$$

$$\mathbf{g}_{(K \times 1)} = \begin{bmatrix} jw_1 & jw_2 & \dots & jw_K \end{bmatrix}^T \quad (4.12)$$

$$\hat{\Phi}_{(K \times N)} = \begin{bmatrix} \frac{1}{j\omega_1 - \bar{p}_1} & \cdots & \frac{1}{j\omega_1 - \bar{p}_N} \\ \vdots & \ddots & \vdots \\ \frac{1}{j\omega_K - \bar{p}_1} & \cdots & \frac{1}{j\omega_K - \bar{p}_N} \end{bmatrix} \quad (4.13)$$

$$\mathbf{c}_{(N \times 1)} = \begin{bmatrix} c_1 & c_2 & \cdots & c_N \end{bmatrix}^T \quad (4.14)$$

Here, $\mathbf{c}$ represents the vector containing the unknown residues of the scaling function $\sigma(s)$ and

$$\tilde{\Phi}_{l(K \times N)} = \left[-\hat{\Phi} \odot \mathbf{S}_l \right] \quad (4.15)$$

Next, Φ_l and $\tilde{\Phi}_l$ matrices are multiplied with vector α_l which contains the appropriate weight factors [42] for each $\mathbf{S}_l$ as

$$\tilde{\mathbf{A}}_{l(K \times (2N+2))} = \alpha_{l(K \times 1)} \odot \begin{bmatrix} \Phi_l & \tilde{\Phi}_l \end{bmatrix}_{(K \times (2N+2))} \quad (4.16)$$

where

$$\alpha_{l(K \times 1)} = \begin{bmatrix} \alpha_{l,1} & \alpha_{l,2} & \cdots & \alpha_{l,K} \end{bmatrix}^T \quad (4.17)$$

with individual elements of α_l computed as [42]

$$\alpha_{l,k} = 1/\sqrt{|S_{l,k}|} \quad (4.18)$$

$\forall k \in \{1, 2, \dots, K\}$, where K denotes the total number of frequency samples.

Prior to QR factorization, the $\tilde{\mathbf{A}}_l$ matrix is split into real and imaginary parts as

$$\mathbf{A}_{l_{2K \times (2N+2)}} = \begin{bmatrix} \text{real}(\tilde{\mathbf{A}}_l) \\ \text{imag}(\tilde{\mathbf{A}}_l) \end{bmatrix} \quad (4.19)$$

In [37], a computationally efficient approach to find $\mathbf{c}$ can be found, in which $\mathbf{A}_l$ matrix in each block row for $l = 1, 2, \dots, P^2$ is considered independently and subjected to a QR decomposition as follows

$$\mathbf{A}_l = \begin{bmatrix} \mathbf{Q}_l & \mathbf{R}_l \end{bmatrix} = \begin{bmatrix} \mathbf{R}_l^{11} & \mathbf{R}_l^{12} \\ 0 & \mathbf{R}_l^{22} \end{bmatrix} \quad (4.20)$$

The upper triangular matrix $\mathbf{R}_{(N \times N)}^{22}$ is collected from each of the block rows and a new compressed least square problem for evaluating $\mathbf{c}$ is formulated as

$$\begin{bmatrix} \mathbf{R}_1^{22} \\ \mathbf{R}_2^{22} \\ \vdots \\ \mathbf{R}_{P^2}^{22} \end{bmatrix}_{(NP^2 \times N)} \begin{bmatrix} \mathbf{c} \end{bmatrix}_{(N \times 1)} = \begin{bmatrix} \mathbf{Q}_1^T \mathbf{S}_1 \\ \mathbf{Q}_2^T \mathbf{S}_2 \\ \vdots \\ \mathbf{Q}_{P^2}^T \mathbf{S}_{P^2} \end{bmatrix}_{(NP^2 \times 1)} \quad (4.21)$$

4.1.2 Relaxed VF

The main idea in *relaxed VF*, which provides superior convergence performance [36], is to remove the asymptotic unitary constraint of $\sigma(s)$ in (4.3) by adding $\hat{d}$ as an unknown, where $\hat{d}$ is real. For this purpose, a modified scaling function $\hat{\sigma}(s)$ is defined as

$$\hat{\sigma}(s) = \sum_{n=1}^N \frac{c_n}{s - \bar{p}_n} + \hat{d} \quad (4.22)$$

In Relaxed VF, the least square system in (4.6) becomes

$$\begin{bmatrix} \Phi_1 & 0 & \dots & \dots & 0 & \tilde{\Phi}_1 \\ 0 & \ddots & \dots & \dots & \vdots & \vdots \\ \vdots & \vdots & \Phi_l & \dots & 0 & \tilde{\Phi}_l \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & 0 & \dots & \Phi_{P^2} & \tilde{\Phi}_{P^2} \end{bmatrix} \begin{bmatrix} \tilde{c}_1 \\ \vdots \\ \tilde{c}_l \\ \vdots \\ \tilde{c}_{P^2} \\ \hat{c} \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad (4.23)$$

where $\tilde{\Phi}_l$ and $\hat{c}$ are modified as

$$\tilde{\Phi}_{l_{K \times (N+1)}} = \begin{bmatrix} -\hat{\Phi} \odot S_l & -S_l \end{bmatrix} \quad (4.24)$$

$$\hat{c}_{(N+1) \times 1} = \begin{bmatrix} c_1 & c_2 & \dots & c_N & \hat{d} \end{bmatrix}_{(N+1) \times 1}^T \quad (4.25)$$

where $\hat{c}$ represents the vector containing the unknown residues of the scaling function $\hat{\sigma}(s)$. Next, similar to (4.16), Φ_l and $\tilde{\Phi}_l$ matrices are multiplied by α_l as

$$\tilde{A}_{l_{(K \times (2N+3))}} = \alpha_{l_{(K \times 1)}} \odot \begin{bmatrix} \Phi_l & \tilde{\Phi}_l \end{bmatrix}_{(K \times (2N+3))} \quad (4.26)$$

where

$$\alpha_l = \begin{bmatrix} \alpha_{l,1} & \alpha_{l,2} & \dots & \alpha_{l,K} \end{bmatrix}_{(K \times 1)}^T \quad (4.27)$$

with individual elements of α_l computed as [42]

$$\alpha_{l,k} = 1/\sqrt{|S_{l,k}|} \quad (4.28)$$

$\forall k \in \{1, 2, \dots, K\}$, where K denotes the total number of frequency samples.

Prior to QR factorization, the $\tilde{\mathbf{A}}_l$ matrix is split into real and imaginary parts as

$$\mathbf{A}_l = \begin{bmatrix} \text{real}(\tilde{\mathbf{A}}_l) \\ \text{imag}(\tilde{\mathbf{A}}_l) \end{bmatrix}_{2K \times (2N+3)} \quad (4.29)$$

Once the upper triangular matrix $\mathbf{R}_{l(N+1) \times (N+1)}^{22}$ from each $\mathbf{R}_l$ is collected, a modified least square problem for the computation of unknown residues of the $\hat{\sigma}(s)$ is formulated as

$$\begin{bmatrix} \mathbf{R}_1^{22} \\ \vdots \\ \mathbf{R}_{P^2}^{22} \\ \boldsymbol{\lambda}^T \end{bmatrix}_{((N+1)P^2+1) \times (N+1)} \begin{bmatrix} \hat{\mathbf{c}} \end{bmatrix}_{(N+1) \times 1} = \begin{bmatrix} 0 \\ \vdots \\ \boldsymbol{\beta} \end{bmatrix}_{((N+1)P^2+1) \times 1} \quad (4.30)$$

where, $\boldsymbol{\lambda}^T$ is a row vector containing the coefficients of the constraint that is introduced as a new equation to avoid all the vanishing trivial solution [36] as

$$\text{Re} \sum_{k=1}^K \left(\hat{d} + \sum_{n=1}^N \frac{c_n}{s - p_N} \right) = \boldsymbol{\beta} \quad (4.31)$$

Equation (4.31) enforces that the sum of the real part of $\hat{\sigma}(s)$ over the frequency samples equals some non-zero value, without fixing any of the free variables [36]. As Relaxed VF [36] converges, $\hat{\sigma}(s)$ (4.22) will approach unity at all frequencies ($\{c_n\} = 0, \hat{d} = 1$). In the implementation in [42], $\boldsymbol{\beta}$ is chosen as $\boldsymbol{\beta} = K$.

It is demonstrated in [38] that the bulk of the computational cost in VF is asso-

ciated with the QR decomposition of each A_l element. Correspondingly, parallelization strategies were developed that were suitable for multi-core/multi-CPU environment. In the next section, parallelization strategies using multi-CPU environment for various stages of the VF is discussed. Since *Relaxed VF* has better convergence properties, in the rest of the discussions, we will discuss the parallelization algorithms with respect to the same.

4.2 Parallelization Strategies for multi CPU Environment

As it is evident from the above section that as the number of ports/poles increases, system of equations that needs to be solved in every VF iteration becomes large and leads to large size system whose solution requires significant CPU and memory resources. To address this, parallelization based on *None Splitting Strategy* and *All Splitting Strategies* using multi-CPU environment [38] are discussed in the following sections.

4.2.1 None Splitting Strategy

None Splitting Strategy is based on computing a common pole set for the entire multiport matrix. In this approach, all the $P^2 S_l$ elements of the multiport matrix are considered together and a common scaling function is used for the entire S matrix.

In this parallelization strategy, the formulation of Φ_l matrix (4.10) and $\tilde{\Phi}_l$ (4.24) for each S_l is done independently in separate threads. Next, Φ_l and $\tilde{\Phi}_l$ matrices are multiplied with vector α_l which contains the appropriate weight factors [42] for each S_l . This is followed by QR factorization of the corresponding A_l matrix

in that particular thread. Next, residue ($\hat{c}$) for the scaling function $\hat{\sigma}(s)$ for the entire S matrix is computed on a single thread. At the end of the final iteration, a common set of poles for the entire macromodel is obtained. The *None Splitting Strategy* for extracting common-pole set is given as a flowchart in Figure 4.1 and the steps are summarized below.

- (a) Each of the Φ_l (4.10) and $\tilde{\Phi}_l$ (4.24) matrix pairs (i.e., $(\Phi_1, \tilde{\Phi}_1)$, $(\Phi_2, \tilde{\Phi}_2)$ $\dots$, $(\Phi_l, \tilde{\Phi}_l)$ $\dots$, $(\Phi_{P^2}, \tilde{\Phi}_{P^2})$) are formed independently in separate threads. Let ' T ' be the total number of available threads. For parallel formulation of each pair $(\Phi_l, \tilde{\Phi}_l)$, they are assigned as follows

$$\left\{ \begin{array}{c} t_1 \\ t_2 \\ \vdots \\ t_l \\ \vdots \\ T \end{array} \right\} \leftarrow \left\{ \begin{array}{c} (\Phi_1, \tilde{\Phi}_1) \\ (\Phi_2, \tilde{\Phi}_2) \\ \vdots \\ (\Phi_l, \tilde{\Phi}_l) \\ \vdots \\ (\Phi_T, \tilde{\Phi}_T) \end{array} \right\} \quad (4.32)$$

Based on the available number of threads, assignment in (4.32) is executed for parallel computation in approximately $\frac{P^2}{T}$ loops.

- (b) Once the matrices Φ_l and $\tilde{\Phi}_l$ are available, they are multiplied with vector α_l which contains the appropriate weight factors [42] for each S_l as

$$\tilde{A}_{l_{(K \times (2N+3))}} = \alpha_{l_{(K \times 1)}} \odot \left[\Phi_l \quad \tilde{\Phi}_l \right]_{(K \times (2N+3))} \quad (4.33)$$

- (c) Next, the $\tilde{\mathbf{A}}_l$ matrix is split into real and imaginary parts in the same CPU thread as

$$\mathbf{A}_l = \begin{bmatrix} \text{real}(\tilde{\mathbf{A}}_l) \\ \text{imag}(\tilde{\mathbf{A}}_l) \end{bmatrix}_{2K \times (2N+3)} \quad (4.34)$$

- (d) Once the $\mathbf{A}_l$ matrix is available, its QR factorization is done in the same CPU thread, to obtain $\mathbf{R}_l$. Note that in (4.34), all the $\mathbf{A}_l$ matrices have the same size and consequently there are no load balancing issues.
- (e) Note that only the $(N + 1) \times (N + 1)$ upper triangular matrix $\mathbf{R}_l^{22}$ are needed to evaluate the unknowns $\hat{\mathbf{c}}$ and are extracted from each respective $\mathbf{R}_l$ matrices in each respective CPU thread.
- (f) By collecting only the $\mathbf{R}_l^{22}$ matrices for each of the $\mathbf{A}_l$ s, a new compressed least square problem is formed in one of the available CPU threads, which will wait for all the threads to compute their $\mathbf{R}_l^{22}$. As a last step in a given iteration, the residue ($\hat{\mathbf{c}}$) for the scaling function $\hat{\sigma}(s)$ is computed in one of the CPU threads including the contributions from all the $\mathbf{A}_l$ s, using (4.30).

To illustrate the above parallelization strategy, consider a 4-port system with scattering matrix, $\mathbf{S}$ and let the number of available CPU threads be 4. The solution process for the *None Splitting* approach for this example is illustrated in Figure 4.1. As seen, formulation and QR decomposition of individual matrices $\mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_l \dots, \mathbf{A}_{16}$ are done parallelly in a loop using the available threads. At the end, the residue computation for the scaling function for the entire $\mathbf{S}$ matrix is done in a single CPU thread.

Figure 4.1: Graphical illustration of *None Splitting Strategy* for multiport VF

The pseudocode corresponding to the *None Splitting Strategy* is given in Algorithm 1.

Algorithm 1 : *None Splitting Strategy* for multiport VF

Input

T : number of available threads, $\forall t \in \{1, \dots, T\}$

P : number of ports

N : number of poles

K : number of frequency samples

ω_k : frequency points, $\forall k \in \{1, \dots, K\}$

$S_{l,k}$: frequency samples, $\forall i, j \in \{1, \dots, P\}$

1: $z_n \leftarrow$ starting poles

2: **repeat**

3: $p_n \leftarrow z_n$

4: $z_n \leftarrow \text{POLESREF}(T, P^2, N, K, \omega_k, S_{l,k}, p_n, I)$

5: **until** poles convergence

6: find the residues $\tilde{c}_{ij,n}$

The function POLESREF inside Algorithm 1 accepts the initial set of poles p_n and raw frequency sample data points. It outputs z_n (zeros of the scaling function which are required in calculating the relocated poles) using T number of threads in parallel. The pseudocode for the POLESREF function is given in Algorithm 2.

Algorithm 2 : Poles refinement in VF algorithm (POLESREF Function)

Input

T : number of available threads, $\forall t \in \{1, \dots, T\}$

L : number of elements to be processed

$\hat{I}$: split to be processed, $\hat{I}=(i_1, j_1), \dots, (i_L, j_L)$

N : number of poles

K : number of frequency samples

ω_k : frequency points, $\forall k \in \{1, \dots, K\}$

$S_{ij,k}$: frequency samples, $\forall i, j \in \{1, \dots, P\}$

p_n : current poles, $\forall i, j \in \{1, \dots, N\}$

function $z_n = \text{POLESREF}(T, L, N, K, \omega_k, S_{ij,k}, p_n, \hat{I})$

parallel region (only if $T > 1$)

build matrix $\hat{\Phi}$ using g and p_n

$t \leftarrow$ thread number

for $l \leftarrow \lfloor \frac{t}{T} L \rfloor$ **to** $\lfloor \frac{t+1}{T} L \rfloor$ **do**

 build matrix $\Phi_l^{(t)}, \tilde{\Phi}_l^{(t)}$ using g, p_n and $S_{l,k}$

$$A_l^{(t)} = \alpha_l^{(t)} \odot \begin{bmatrix} \Phi_l^{(t)} & \tilde{\Phi}_l^{(t)} \end{bmatrix}$$

$$A_l^{(t)} \leftarrow (\tilde{A}_l^{(t)})$$

$$R_l^{(t)} \leftarrow QR(A_l^{(t)})$$

$$V_{[(l)(N+1):l(N+1)+(N+1),:]} = R_l^{(t)}_{[N+3:2N+3, N+3:2N+3]}$$

end for

end parallel region

$$b_{[K+1,1]} = K$$

$$V_{[L(N+1),:]} \leftarrow \lambda^T$$

$$\hat{c} \leftarrow \text{argmin} ||Vx - b||$$

build state space realization $A_\sigma, B_\sigma, C_\sigma, D_\sigma$

$$z_n \leftarrow \text{eig}(A_\sigma - B_\sigma C_\sigma D_\sigma^{-1})$$

Note that, in the *None Parallelization Strategy*, the residue computation ($\hat{c}$) needs to wait until all the $\mathbf{R}_l^{22}$ matrices are computed, which results in a delay due to scheduling. This is one of the major bottlenecks of this strategy. Also while computing QR for individual elements in the last loop of VF iteration, depending on the total number of threads and the total number of matrix elements $\mathbf{A}_l$ (i.e., $l \in 1, 2, 3, \dots, P^2$), some of the threads may remain idle based on the corresponding $\frac{P^2}{T}$ value.

4.2.2 All Splitting Strategy

In the *All Splitting Strategy*, an individual pole-set is computed for the each individual S_l element in the multiport $\mathbf{S}$ matrix. In this approach, scaling function for each S_l is considered independently. In this parallelization strategy, the formulation of Φ_l matrix (4.10) and $\tilde{\Phi}_l$ (4.24), QR factorization and residue computation for the scaling function $\hat{\sigma}_l(s)$ (which is specific to the corresponding $\mathbf{A}_l$) for each S_l is done independently in separate threads. At the end of the final iteration, a individual set of poles for each of the $\mathbf{S}$ matrix element (S_l) is obtained. In this parallelization strategy, scaling function for each S_l is considered independently. The *All Splitting Strategy* for extracting individual-pole set is given as a flowchart in Figure 4.2 and the steps are summarized below.

- (a) Each of the Φ_l (4.10) and $\tilde{\Phi}_l$ (4.24) matrix pairs (i.e., $(\Phi_1, \tilde{\Phi}_1)$, $(\Phi_2, \tilde{\Phi}_2)$ $\dots$, $(\Phi_l, \tilde{\Phi}_l)$ $\dots$, $(\Phi_{P^2}, \tilde{\Phi}_{P^2})$) are formed independently in separate threads. Let 'T' be the total number of available threads. For parallel formulation of each pair $(\Phi_l, \tilde{\Phi}_l)$, they are assigned as in (4.32).
- (b) Once the matrices Φ_l and $\tilde{\Phi}_l$ are available, they are multiplied with vec-

tor α_l which contains the appropriate weight factors [42] for each S_l as in (4.33). Next, $\tilde{A}_l$ matrix is split into real and imaginary parts as in (4.34) to obtain A_l .

- (c) Once the A_l matrix is available, its QR factorization is done in the same thread, to obtain R_l .
- (d) After extracting R_l^{22} , the residues corresponding to the scaling function $\hat{o}_l(s)$ (specific to that S_l element) is computed in the same thread using

$$\begin{bmatrix} R_l^{22} \\ \lambda_l^T \end{bmatrix}_{(N+1 \times N+1)} \hat{c}_{l(N+1 \times 1)} = \begin{bmatrix} 0 \\ \beta_l \end{bmatrix}_{(N+1 \times 1)} \quad (4.35)$$

Similarly, the above steps are used for the computation of residues for all the S_l s, independently using T number of CPUs.

- (e) Based on the available number of threads, steps (a), (b), (c) and (d) are executed for computing the residues of rest of the $\hat{o}_l(s)$ in approximately $\frac{p^2}{T}$ loops.

To illustrate the above parallelization strategy, consider a 4-port system with scattering matrix, S and let the number of available threads be 4. The solution process for the *All Splitting* approach for this example is illustrated in Figure 4.2. As seen, formulation, QR and residue evaluation of individual matrices $A_1, A_2, \dots, A_l, \dots, A_{16}$ are done parallelly in a loop using the available threads.

Figure 4.2: Graphical illustration of *All Splitting Strategy* for multiport VF

A pseudocode corresponding to the *All Splitting Strategy* is given in Algorithm 3.

Algorithm 3 : *All Splitting Strategy* for multiport VF

Input

T : number of available threads, $\forall t \in \{1, \dots, T\}$

P : number of ports

N : number of poles

K : number of frequency samples

ω_k : frequency points, $\forall k \in \{1, \dots, K\}$

$S_{ij,k}$: frequency samples, $\forall i, j \in \{1, \dots, P\}$

1: **parallel region**

2: $t \leftarrow$ thread number

3: **for** $\mu \leftarrow \lceil \frac{t}{T} P^2 \rceil$ **to** $\lceil \frac{t+1}{T} P^2 \rceil$ **do**

4: $I_\mu \leftarrow (i, j), \mu \leftarrow i + (P - 1)j$

5: $z_n^{(t)} \leftarrow$ starting poles

6: **repeat**

7: $p_n^{(t)} \leftarrow z_n^{(t)}$

8: $z_n^{(t)} \leftarrow \text{POLESREF}(1, 1, N, K, \omega_k, S_{ij,k}, p_n^{(t)}, I_\mu)$

9: **until** poles convergence

10: find the residues $\tilde{c}_{ij,n}$

11: **end for**

12: **end parallel region**

A challenge in this parallelization strategy is the memory drain when large number of threads are executed in parallel. In this case, few threads wait for the memory till it is released. This overhead is typically negligible considering the overall runtime [38]. However, this can be overcome by imposing implicit barriers that may block the execution of the thread until memory is released.

4.3 Parallelization Strategies for mixed multi CPU + GPU Environment

In the previous section, a brief review of parallel vector fitting [38] and its parallelization strategies with emphasis on multi-CPU environment was presented. In this section, the emerging platform of GPU is considered and several parallelization strategies based on *None Splitting Strategy* and *All Splitting Strategy* that are suitable for GPU platforms are reviewed [39].

4.3.1 Approach-1: Formulation of A_l in CPU followed by immediate QR factorization and residue evaluation in GPU for None Splitting Strategy

In this parallelization strategy, the formulation of Φ_l matrix (4.10) and $\tilde{\Phi}_l$ (4.24) for each S_l is done independently in separate threads. Next, Φ_l and $\tilde{\Phi}_l$ matrices are multiplied with vector α_l which contains the appropriate weight factors [42] for each S_l . This is followed by QR factorization in GPU to obtain R_l (i.e., although each Φ_l matrix (4.10) and $\tilde{\Phi}_l$ (4.24)) are formulated parallelly, they are sent to GPU in a serial way). This is followed by the residue evaluation ($\hat{c}$) for the scaling function $\hat{\sigma}(s)$ for the entire S matrix on a single CPU (i.e., *None Splitting*). At the end of the final iteration, a common set of poles for the entire macromodel is obtained. The steps are summarized below.

- (a) Each of the Φ_l (4.10) and $\tilde{\Phi}_l$ (4.24) matrix pairs (i.e., $(\Phi_1, \tilde{\Phi}_1)$, $(\Phi_2, \tilde{\Phi}_2)$ $\dots$, $(\Phi_l, \tilde{\Phi}_l)$ $\dots$, $(\Phi_{P^2}, \tilde{\Phi}_{P^2})$) are formed independently in separate threads. Let ' T ' be the total number of available threads. For parallel formulation of each pair $(\Phi_l, \tilde{\Phi}_l)$, they are assigned as in (4.32).
- (b) Once the matrices Φ_l and $\tilde{\Phi}_l$ are available, they are multiplied with vec-

tor α_l which contains the appropriate weight factors [42] for each S_l as in (4.27). Subsequently, $\tilde{A}_l$ matrix is split into real and imaginary parts as in (4.29).

- (c) Once the A_l matrix is available, it is sent to the GPU immediately for QR factorization to obtain R_l . Although each A_l matrix is formulated parallelly, it is sent to GPU in a serial way. While the QR factorization of a single A_l matrix of a particular CPU is being computed, the remaining CPU threads need to wait until the QR computation of that particular A_l matrix is completed in the GPU.
- (d) Next, the R_l^{22} matrices are collected for each of the A_l s in the same GPU one at a time and the residues ($\hat{c}$) for the scaling function $\hat{\sigma}(s)$ is computed in that GPU itself, including the contributions of all A_l s, using (4.30).

The pseudocode corresponding to the *GPU based Splitting (Approach-1)* is given in Algorithm 4.

Algorithm 4 : *GPU based Splitting (Approach-1)* for multiport VF

Input
 T : number of available threads, $t \in 1, 2, \dots, T$
 N : number of poles

 K : number of frequency samples

 ω_k : frequency points, $\forall k \in \{1, \dots, K\}$
 $S_{ij,k}$: frequency samples, $\forall i, j \in \{1, \dots, P\}$
 p_n : current poles, $\forall i, j \in 1, \dots, N \ \omega_k, S_{ij,k}, p_n, \hat{I}$

1: **parallel region** (only if $T > 1$)

2: $t \leftarrow$ thread number

3: **for** $l \leftarrow \lceil \frac{t}{T} P^2 \rceil + 1$ **to** $\lceil \frac{t+1}{T} P^2 \rceil$ **do**

4: build matrix $\Phi_l^{(t)}, \tilde{\Phi}_l^{(t)}$ using g, p_n and $S_{l,k}$

5: $A_l^{(t)} = \alpha_l^{(t)} \odot \begin{bmatrix} \Phi_l^{(t)} & \tilde{\Phi}_l^{(t)} \end{bmatrix}$

6: $A_l^{(t)} \leftarrow (\tilde{A}_l^{(t)})$

7: A is sent to GPU for QR factorization

8: $V_{[(l(N+1)):l(N+1)+(N+1),:]} = R_l^{(t)}_{[N+3:2N+3, N+3:2N+3]}$

9: **end for**

10: **end parallel region**

11: $b_{[K+1,1]} = K$

12: $V_{[L(N+1),:]} \leftarrow \lambda^T$

13: $\hat{c} \leftarrow \operatorname{argmin} ||Vx - b||$

14: build state space realization $A_\sigma, B_\sigma, C_\sigma, D_\sigma$

15: $z_n \leftarrow \operatorname{eig}(A_\sigma - B_\sigma C_\sigma D_\sigma^{-1})$

In this *Approach-1*, once the A_l is formed, each thread waits in queue to perform QR factorization. This adds to delay in parallel pipeline scheduling. Also, the data transfer time from CPU to GPU for the transfer of P^2 A_l s is excessive. To overcome the above challenges, *Approach-2* was presented and is reviewed in the next section.

4.3.2 Approach-2: Transfer of block of A_l s from CPU to GPU and parallel QR factorization in CPUs and GPU for None Splitting Strategy (single scaling function)

In this parallelization strategy, the formulation of Φ_l matrix (4.10) and $\tilde{\Phi}_l$ (4.24) for each S_l is done independently in separate CPU threads. Next, Φ_l and $\tilde{\Phi}_l$ matrices are multiplied with vector α_l which contains the appropriate weight factors [42] for each S_l . Subsequently, A_l s are grouped into two blocks, with n_g number of A_l s for QR factorization in GPU and n_c number of A_l s for QR factorization in $(T-1)$ CPU threads. The numbers n_g and n_c are decided in a way that both the $(T-1)$ CPU threads and GPU are kept busy in the parallel loop of a PR iteration. As soon as n_g number of A_l s are parallelly formulated in threads, they are sent at once to the GPU for QR factorization. The remaining threads $(T-1)$ continue to parallelly formulate the remaining n_c A_l s and also their QR factorization. The R_l^{22} matrices are then collected from both the GPU and CPUs parallelly, in the CPU DRAM memory associated with one of the threads (refer to Figure 2.2). This is followed by the residue ($\hat{c}$) evaluation for the scaling function $\hat{\sigma}(s)$ in the same CPU. At the end of the final PR iteration, a common set of poles for the entire macromodel is obtained. The steps involved in Approach-2 for extracting common-pole set are summarized below.

- (a) Each of the Φ_l (4.10) and $\tilde{\Phi}_l$ (4.24) matrix pairs (i.e., $(\Phi_1, \tilde{\Phi}_1)$, $(\Phi_2, \tilde{\Phi}_2)$ $\dots$, $(\Phi_l, \tilde{\Phi}_l)$ $\dots$, $(\Phi_{P^2}, \tilde{\Phi}_{P^2})$) are formed independently in separate threads, followed by multiplication by vector α_l which contains the appropriate weight factors [42] for each S_l as in (4.27). Subsequently, $\tilde{A}_l$ matrix is split into real and imaginary parts as in (4.29) in the same thread.

- (b) The above steps are repeated until the first n_g number of $\mathbf{A}_l$ elements are computed for which approximately $\frac{n_g}{T}$ loops are required.
- (c) Once n_g number of $\mathbf{A}_l$ s are obtained, they are stored as $\mathbf{A}_{GPU} \leftarrow \mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_{n_g}$ and sent at once to a GPU for parallel QR factorization to obtain $\mathbf{R}_l$ s for each of the $\mathbf{A}_l$ s. Then, the respective $\mathbf{R}_l^{22}$ for all $\mathbf{A}_l$ matrices are collected and then sent to a single CPU thread at once.
- (d) While the *step (b)* is in progress, remaining $(T-1)$ threads continue with parallel formulation and QR factorization of the remaining n_c $\mathbf{A}_l$ s and the respective $\mathbf{R}_l^{22}$ matrices are stored in the CPU DRAM associated with one of the threads.

The *step (b)* and *step (c)* are executed in parallel until the $\mathbf{R}_l$ s of all the $\mathbf{A}_l$ matrices are obtained (n_g and n_c are selected such that threads and GPU are kept busy throughout the parallel loop).

- (e) Once the $\mathbf{R}_l^{22}$ matrices are collected for each of $\mathbf{A}_l$ s from the GPU and the threads in one of the available threads, the residue vector ($\hat{c}$) for the scaling function $\hat{\sigma}(s)$ is computed in that same thread itself, using (4.30).

The pseudocode corresponding to the *Approach-2* is given in Algorithm 5.

Algorithm 5 : *GPU based Splitting (Approach-2) for multiport VF*

Input

T : number of available threads, $t \in 1, 2, \dots, T$

P : number of ports

N : number of poles

K : number of frequency samples

ω_k : frequency points, $\forall k \in \{1, \dots, K\}$

$S_{ij,k}$: frequency samples, $\forall i, j \in \{1, \dots, P\}$

p_n : current poles, $\forall i, j \in \{1, \dots, N\}$

- 1: **parallel region** (only if $T > 1$)
 - 2: build matrix $\hat{\Phi}$ using g and p_n
 - 3: $t \leftarrow$ thread number
 - 4: $d \leftarrow \text{gpuDeviceCount}$
 - 5: **for** $l \leftarrow \lfloor \frac{t-1}{T} n_g \rfloor + 1$ **to** $\lfloor \frac{t}{T} n_g \rfloor$ **do**
 - 6: $index1 \leftarrow n_g - \frac{t-1}{T} n_g + 1$
 - 7: $\mathbf{A}_{GPU}[:, :, index1] = \mathbf{A}$
 - 8: **end for**
 - 9: **if** $t \leq d$ **then**
 - 10: $\mathbf{A}_{GPU}$ is sent to GPU
 - 11: $\mathbf{R}_l^{22} = \mathbf{R}_{l[N+3:2N+2, N+3:2N+2, :]}$
 - 12: **else**
 - 13: $n_c \leftarrow L - n_g$
 - 14: **for** $l \leftarrow \lfloor \frac{t-1}{T-1} n_c \rfloor + 1$ **to** $\lfloor \frac{t}{T-1} n_c \rfloor$ **do**
 - 15: $(\mathbf{Q}^{(t)}, \mathbf{R}^{(t)}) \leftarrow QR(\mathbf{A}_l^{(t)})$
 - 16: $index2 \leftarrow n_c - \frac{t-1}{T-1} n_c + 1$
 - 17: $\mathbf{R}_{l[:, :, index2]}^{22} = \mathbf{R}_{l[N+3:2N+2, N+3:2N+2]}^{(t)}$
 - 18: **end for**
 - 19: **end if**
 - 20: **end parallel region**
 - 21: $\mathbf{V}_{[(l)(N+1):l(N+1)+(N+1), :]} = \mathbf{R}_l^{22}$
 - 22: $\mathbf{b}_{[K+1, 1]} = \mathbf{K}$
 - 23: $\mathbf{V}_{[L(N+1), :]} \leftarrow \boldsymbol{\lambda}^T$
 - 24: $\hat{\mathbf{c}} \leftarrow \text{argmin} ||\mathbf{V}\mathbf{x} - \mathbf{b}||$
 - 25: build state space realization $\mathbf{A}_\sigma, \mathbf{B}_\sigma, \mathbf{C}_\sigma, \mathbf{D}_\sigma$
 - 26: $z_n \leftarrow \text{eig}(\mathbf{A}_\sigma - \mathbf{B}_\sigma \mathbf{C}_\sigma \mathbf{D}_\sigma^{-1})$
-

4.3.3 Approach-3: Transfer of block of A_l s from CPUs to GPU, parallel QR factorization and residue evaluation in CPUs as well as in GPU for All Splitting Strategy (individual scaling functions)

This approach is based on *All Splitting Strategy* using mixed CPU + GPU environment, where, at the end of converged PR iteration, each S_l element of the multiport S matrix will have its own individual pole-set.

In this parallelization strategy, similar to *Approach-2*, the A_l s are grouped into two blocks, with n_g number of A_l s for QR factorization as well as corresponding residue evaluation in GPU and n_c number of A_l s for QR factorization as well as corresponding residue evaluation in CPUs. Here, each element S_l will have its own scaling function, resulting in its own set of poles. Therefore, once the R_l^{22} matrices are computed in GPU, corresponding residue $\hat{c}_l$ for the scaling function $\hat{\sigma}_l(s)$ is computed (S_l vectors needed for residue evaluation is sent to GPU in the beginning itself). Parallely, remaining $(T-1)$ threads compute the residues $\hat{c}_l$ of the remaining n_c number of scaling function $\hat{\sigma}_l(s)$. At the end of the converged PR iteration, each S_l element will have its own set of poles. The steps involved in *Approach-3* for extracting individual-pole set are summarized below.

- (a) Each of the Φ_l (4.10) and $\tilde{\Phi}_l$ (4.24) matrix pairs (i.e., $(\Phi_1, \tilde{\Phi}_1)$, $(\Phi_2, \tilde{\Phi}_2)$ $\dots$, $(\Phi_l, \tilde{\Phi}_l)$ $\dots$, $(\Phi_{P^2}, \tilde{\Phi}_{P^2})$) are formed independently in separate threads, followed by multiplication by vector α_l which contains the appropriate weight factors [42] for each S_l as in (4.27). Subsequently, $\tilde{A}_l$ matrix is split into real and imaginary parts as in (4.29) in the same thread.
- (b) The above steps are repeated until the first n_g number of A_l s are computed

for which approximately $\frac{n_g}{T}$ loops are required.

- (c) Once n_g number of A_l s are obtained, they are stored as $A_{GPU} \leftarrow A_1, A_2, \dots, A_{n_g}$ and it is sent along with $b_{GPU} \leftarrow S_1, S_2, \dots, S_{n_g}$, at once to the GPU. GPU performs parallel QR factorization to obtain R_l for each of the A_l s. Subsequently, R_l^{22} for each of the n_g A_l s are extracted and residues ($\hat{c}_l$) are computed for the corresponding scaling function $\hat{\sigma}_l(s)$ (which is specific to that S_{ij} element) in the GPU itself.
- (d) While *step (b)* is being processed, remaining threads ($T-I$) continue with parallel formulation and QR factorization of the remaining n_c A_l s and the resulting R_l^{22} matrices are collected. Next, the residues of the corresponding scaling function $\hat{\sigma}_l(s)$ (which is specific to that S_{ij} element) are also computed in the respective threads.

Since the steps (d) and *step (e)* are executed in parallel in both the ($T-I$) threads and GPU, they are kept busy throughout the parallel loop. The pseudocode corresponding to the *Approach-3* is given in Algorithm 5.

Algorithm 6 : *GPU based Splitting (Approach-3)* for multiport VF

Input

P : number of ports
 N : number of poles
 K : number of frequency samples
 ω_k : frequency points, $\forall k \in \{1, \dots, K\}$
 $S_{ij,k}$: frequency samples, $\forall i, j \in \{1, \dots, P\}$
 p_n : current poles, $\forall i, j \in \{1, \dots, N\}$

- 1: **parallel region** (only if $T > 1$)
- 2: build matrix $\hat{\Phi}$ using g and p_n
- 3: $t \leftarrow$ thread number
- 4: $d \leftarrow \text{gpuDeviceCount}$
- 5: **for** $l \leftarrow \lceil \frac{t-1}{T} n_g \rceil + 1$ **to** $\lceil \frac{t}{T} n_g \rceil$ **do**
- 6: $index1 \leftarrow n_g - \frac{t-1}{T} n_g + 1$
- 7: build matrix A using g_k, p_n and $S_{ijl,k}$
- 8: build vector b using $S_{l,k}$
- 9: $A_{GPU}[:, :, index1] = A$
- 10: $b_{GPU}[:, :, index1] = b$
- 11: **end for**
- 12: **if** $t \leq d$ **then**
- 13: A_{GPU}, b_{GPU} is sent to GPU
- 14: $\hat{c} \leftarrow b_{GPU} (dgels)$
- 15: **else**
- 16: $n_c \leftarrow L - n_g$
- 17: **for** $l \leftarrow \lceil \frac{t-1}{T-1} n_c \rceil + 1$ **to** $\lceil \frac{t}{T-1} n_c \rceil$ **do**
- 18: $(Q^{(t)}, R^{(t)}) \leftarrow QR(A_l^{(t)})$
- 19: $index2 \leftarrow n_c - \frac{t-1}{T-1} n_c + 1$
- 20: $R_{l[(:, :, index2)]}^{22} = R_{l[N+3:2N+2, N+3:2N+2]}^{(t)}$
- 21: find $\hat{c}_l^{(t)}$
- 22: **end for**
- 23: **end if**
- 24: **end parallel region**
- 25: build state space realization $A_\sigma, B_\sigma, C_\sigma, D_\sigma$
- 26: $z_n \leftarrow \text{eig}(A_\sigma - B_\sigma C_\sigma D_\sigma^{-1})$

This chapter reviewed the recent work in the literature for parallelization of VF for relaxed VF suitable for multi core CPUs as well as mixed CPU + GPU environment. Detailed discussions were given for the associated formulation as well as solution for algorithms.

Chapter 5

Development of GPU based Vector Fitting (GVF) with None Splitting Strategy

In Chapter 4, a brief review of parallel vector fitting [38] - [39] and its splitting strategies for high-order rational-function approximation of multiport tabulated data using multi-core or distributed CPU and mixed multi CPU + GPU environment was presented. However, in VF, it is to be noted that, to arrive at the proper number of converged poles, large number of VF iterations are often required making it a computationally expensive process despite all the advances in the field. This problem gets further aggravated in the presence of large number of ports, data requiring large number of poles and need for inclusion of large number of frequency samples.

To address the above problem, in this chapter, novel contributions are made to further advance the applicability of VF to exploit the emerging massively parallel graphical processing Units (GPUs) so as to significantly reduce the computational time. Necessary parallelization strategies that are suitable for GPU platforms are developed. In developing the GVF based *None Splitting Strategy*, emphasis is placed on minimizing the major cost associated with the VF algorithm, which is the QR factorization of the individual $\mathbf{A}_l$ elements, and also on other minor costs like $\mathbf{A}$ -formulation, $\mathbf{R}_l^{22}$ extraction and scaling function residue computations in the GPU itself. Hence, formulation of the $\mathbf{A}_l$ elements were planned on

the GPU along with QR factorization and scaling function residue computations. Comparison is made between *MAGMA* [40] and *cuBLAS* libraries [41] for the computational cost of one pole refinement iteration associated with the VF algorithm. Also, the developed approach provides comparison in terms of whether to use *LAPACK* library (i.e., CPU library) [45] or *MAGMA* library [40] (i.e., GPU library) for residue computation of the scaling function.

During the detailed presentation of the proposed parallelization algorithms, discussions are given with respect to the Relaxed VF case (since it provides better convergence) using both the *All Splitting* and *None Splitting* based approaches. In this chapter, parallelization based on computing a common pole set for the entire matrix using *Relaxed VF* is presented.

5.1 GVF based on None Splitting Strategy

In this section, parallelization of the VF using the *None Splitting strategy* (wherein a common pole set is extracted for the entire matrix) that are suitable for GPU platforms are developed. Parallelization strategy is focused on minimizing all the computational costs: involved in formulation (4.10) and (4.24), splitting $\tilde{\mathbf{A}}_l$ into real and imaginary parts (4.29), QR factorization of $\mathbf{A}_l$ (4.20) and $\mathbf{R}_l^{22}$ extraction (4.20). Developed strategy is focused on parallelizing these tasks on independent P^2 thread blocks in GPU. The proposed GVF algorithm for extracting common-pole set is given as a flowchart in Figure 5.1 and the steps are summarized below.

- (a) Once the matrix $\mathbf{S}_l$ (4.8), $\hat{\Phi}$ (4.10), $\mathbf{g}$ (4.12) and α_l (4.17) are available for the ports $l = 1, 2, \dots, P^2$, they are bulk transferred from the CPU DRAM to the GPUs global memory.

Figure 5.1: Flow diagram for the proposed *GVF* (for *None Splitting*) for one pole refinement iteration

- (b) Assign $\text{GPU} \leftarrow \hat{\Phi}$ (4.13). Note that after the first iteration, in the subsequent iterations, *step (a)* is skipped and only *step (b)* is used.
- (c) Next, Φ_l (4.10) and $\tilde{\Phi}_l$ (4.24) are formulated and stored in GPU global memory using P^2 blocks and K threads.
- (d) Φ_l and $\tilde{\Phi}_l$ are then assembled in a block row-wise and are split into real and imaginary parts as in (4.29) to obtain all the A_l s.
- (e) Next, each A_l is sent to *batch QR function* which computes just the R_l matrices of the corresponding A_l matrices in parallel, in GPU.
- (f) Next, parallel extraction of R_l^{22} from R_l matrices is done in the GPUs global memory using P^2 blocks and $N+1$ threads.
- (g) Only the R_l^{22} matrices for the ports $l = 1, 2, \dots, P^2$ are transferred from GPU global memory to a single CPU.
- (h) Once the R_l^{22} for all the ports are available, residue vector $\hat{c}$ of the scaling function $\hat{\sigma}(s)$ is computed using (4.30), using a single CPU.
- (i) Using the form in (3.24), the zeros of the scaling function are computed in the same CPU, which serves as the refined poles for starting the next iteration.

In the above flow, for the *step (e)* (i.e., *the batch QR function*), applicability and efficiency of *MAGMA* [40] and *cuBLAS* [41] libraries are investigated. In case of availability of large GPU global memory, after the first iteration, steps (g) and (h) and formulation of $\hat{\Phi}$ can be done in GPU itself for every subsequent iteration

until the VF iterations converge. Details and the results of the investigation are given in the next section.

5.2 *Performance Comparison of the Proposed GVF with None Splitting Strategy on a Single GPU + CPU Platform versus PVF [38] on multiple CPUs*

Performance comparison of the proposed GVF when executed on a single GPU + CPU platform versus the PVF [38] when executed on multiple CPUs is given. For this purpose, two relatively large examples are considered under various scenarios. Discussions are given with respect to one full pole refinement iteration of the VF algorithm. Example 1 represents a medium size problem with 60 ports and 70 poles while the Example 2 represents a relatively large problem for VF with 120 ports and 80 poles.

Numerical results reported in this section are based on execution using the Compute Canada platform (a Canada wide national computing grid) [48] that uses GPU (Tesla V100-SXM2-16GB) which consists of 5120 cores, 16-GB RAM, 6MB of L2 cache memory and operates at 1530MHz clock frequency and uses a CPU (Intel Xeon 6148 Skylake) operating at 2.4 GHz in a set up with 20 allocated cores and 20 GB/core for the experiment.

Example 1: In this example, a multiport network with 60-port (P), 1000 frequency samples (K) and 70 poles (N) (including complex poles and their conjugate poles) is considered and performance of *GVF with None Splitting Strategy* is investigated for one pole refinement iteration. Such a scenario leads to each $\mathbf{A}_l$ in (4.29) with a matrix size of (2000×143) where $\hat{\Phi}$ in (4.13) is a matrix of size (1000×70) .

In the first part of this experiment, while using the GVF, QR decomposition is done using the *MAGMA* [40] library. Particularly, the routine *magma_dgeqrf_batched* is used. The proposed GVF needed only **1.08 sec** for one pole refinement iteration. Results are compared against the PVF [38] on varying number of CPUs (1 to 20). Corresponding time taken by the PVF [38] is given in Table 5.1 in column 3. As can be seen from Table 5.1, the proposed GVF yields a speedup of **23.35** against a single core CPU, **10.94** against two cores, **2.20** against 12 cores and **1.46** against 20 cores.

Table 5.1: Performance comparison of the proposed *GVF* using *MAGMA library* versus PVF based on multi CPU approach (for *None Splitting*)

Example	PVF [34] (multi CPU)		Proposed GVF time using MAGMA library (sec)	Speedup
	# CPU cores used	Time (sec)		
#ports = 60 #poles = 70 #fpoints = 1000 #size of $\hat{\Phi}=1000 \times 70$ Total # of $\phi_l=3600$	1	25.22	1.08	23.35
	2	11.82		10.94
	4	6.01		5.56
	6	4.24		3.93
	8	3.20		2.96
	12	2.38		2.20
	16	1.83		1.69
	20	1.58		1.46

In the following sections, further experimental cost analysis of each task associated with the *GVF* based on *None Splitting Strategy* is given.

5.2.1 Data (S , g and α) transfer cost from CPU to GPU: step (a)

In this section, transfer time required for transferring the data (S , g and α) from CPU to GPU is given. Bulk of this cost corresponds to the cost associated with transferring the data (S) of multiport matrix. Reported times correspond to the

previously defined system with 60 ports (P), 1000 frequency samples (K) and 70 poles (N).

Note that, as seen in Figure 5.1, in *step (a)*, only for the first PR iteration, data corresponding to S , g and α need to be transferred from CPU to GPU. The transfer time required for these data from CPU to GPU is **20.19 msec**. This step is not required in subsequent iterations. Hence subsequent iterations will be slightly faster compared to the first iteration.

5.2.2 Formulation and transfer of $\hat{\Phi}$: *step (b)*

In the first as well as subsequent iterations, $\hat{\Phi}$ (in *step (b)*) is formulated and is transferred from CPU to GPU. The cost involved in formulating $\hat{\Phi}$ in CPU is **1.62 msec** for a matrix size of (1000×70) . The transfer time associated with the $\hat{\Phi}$ from CPU to GPU is only **0.35 msec**. This formulation and data transfer cost for $\hat{\Phi}$ associated with the subsequent VF iterations is minimal considering the overall cost of a pole refinement iteration. Since $\hat{\Phi}$ is a relatively small matrix of size $(K \times N)$ and the transfer cost for the transfer of $\hat{\Phi}$ is very minimal, it was opted to formulate $\hat{\Phi}$ in CPU itself instead of GPU.

5.2.3 Formulation of A_l : *step (c), step (d)*

After the transfer of $\hat{\Phi}$ from CPU to GPU, matrix A_l needs to be formulated for each S_l element (i.e., the computations involved in (4.10), (4.24), (4.26) and (4.29)). For this purpose, a customized GPU batch function is developed that uses P^2 blocks and K threads to formulate all P^2 A_l matrices (with $P^2 = 3600$) parallelly across multiple threads and blocks.

With regards to the example under consideration (system with 60 ports), $P^2 =$

3600 number of A_l s are formulated in GPU. Total time taken for formulation of all P^2 A_l matrices is only **0.04 msec** in GPU.

5.2.4 QR factorization time in GPU using MAGMA [40] library: step (e)

This step constitutes the bulk of the cost associated with the VF iterations. Once all A_l matrices are formulated, each A_l is sent to *batch QR function* (i.e., *step (e)*). For this step, in the first part of this experiment, the *MAGMA library* [40] is used. Particularly, the routine *magma_dgeqrf_batched* is used. Corresponding QR factorization time (with *step (e)*) is **543.09 msec**.

5.2.5 R_l^{22} extraction: step (f)

After the QR factorization in GPU in *step (f)*, R_l^{22} (a sub-matrix of size $(N + 1) \times (N + 1)$) is extracted from the R_l matrix (of size $(2K) \times (2N + 3)$) for each S_l element. Since, R_l^{22} needs to be extracted for each S_l element, R_l^{22} extraction is done in GPU itself instead of using multi-core CPUs or a single CPU, so as to exploit higher parallelism and avoid the excessive data transfer time for the transfer of larger matrix R_l from the GPU to CPU (for all the P^2 elements). For this *step (f)*, a customized GPU algorithm is developed, which uses P^2 blocks and $(N+1)$ threads to extract all P^2 R_l^{22} matrices parallelly across multiple threads and blocks. Total time taken to extract all ($P^2 = 3600$) R_l^{22} matrices is only **0.02 msec** in GPU.

5.2.6 Residue evaluation of the scaling function: step (h)

After *step (f)* (i.e., extraction of all R_l^{22} s), two different methods are considered for evaluation of residue $\hat{c}$ of the scaling function $\hat{\sigma}(s)$.

Approach (a): In the first approach, all ($P^2 = 3600$) R_l^{22} matrices are transferred

from GPU to CPU. Next, the residue vector $\hat{c}$ of the scaling function $\hat{\sigma}(s)$ is computed using (4.30) in a single CPU. The total transfer cost involved in transferring all P^2 R_t^{22} elements from GPU to CPU is **55.96 msec**. Total time taken for computation of residue ($\hat{c}$) of the scaling function $\hat{\sigma}(s)$ using (4.30) in a single CPU is **460.02 msec**. Hence this approach needed a total time of $55.96 \text{ msec} + 460.02 \text{ msec} = 515.98 \text{ msec}$ for the computation of $\hat{c}$. This constitutes the second major cost in the VF PR iterations.

Approach (b): In the second approach, all ($P^2 = 3600$) R_t^{22} matrices are retained in the GPU itself and the residue computation $\hat{c}$ of the scaling function $\hat{\sigma}(s)$ is done in the GPU itself. Particularly, the *MAGMA* routine *magma_dgels* is used for this purpose. The cost involved in computing the residues $\hat{c}$ of the scaling function $\hat{\sigma}(s)$ in GPU is **562.502 msec**. After this step, the unknowns $\hat{c}_1, \dots, \hat{c}_n$ are transferred from GPU to CPU. The transfer costs associated with this is only **0.03 msec**. Hence, the total time taken for calculating and transferring the $\hat{c}$ to the CPU is **562.532 msec** using *Approach (b)*. Comparing *Approach (a)*: **515.98 msec** and *Approach (b)*: **562.532 msec**, it can be seen that, *Approach (a)* performs computationally better.

5.2.7 Computation of zeros of the scaling function $\hat{\sigma}(s)$: step (i)

Once the unknowns $\hat{c}_1, \dots, \hat{c}_n$ are obtained by solving (4.30), as a last step in the PR iterations, zeros of $\hat{\sigma}(s)$ are calculated in the CPU using the form in (3.24). Total time taken for zeros calculation is only **3.6 msec**.

Table 5.2 provides the itemized computational times for one pole refinement iteration using *MAGMA library* while using both *Approach (a)* and *Approach (b)*.

Table 5.2: Itemized cost for the proposed *GVF* using *MAGMA* library (for *None Splitting*)

Example	Steps	Cost of individual steps in GVF (Using <i>MAGMA</i> [36] library)	Time (msec) (Approach (a))	Time (msec) (Approach (b))
#ports = 60 #poles = 70 #fpoints = 1000 #size of $\hat{\Phi}=1000 \times 70$ Total # of $\Phi_l=3600$	(a)	Transfer of $\mathcal{S}, \mathbf{g}, \alpha$ from CPU to GPU	20.19	20.19
	(b)	Formulation of $\hat{\Phi}$ in CPU and transfer to GPU	1.97	1.97
	(c), (d)	Formulation of $\mathbf{A}$ in GPU	0.04	0.04
	(e)	QR factorization in GPU (<i>MAGMA</i>)	543.09	543.09
	(f)	$\mathbf{R}_l^{22}$ extraction in GPU	0.02	0.02
	(g)	Transfer back $\mathbf{R}_l^{22}$ from GPU to CPU	56.00	-
	(h)	Residue $\hat{\mathbf{c}}$ calculation in CPU (<i>LAPACK</i>)	460.02	-
	(h1)	Residue $\hat{\mathbf{c}}$ calculation in GPU (<i>MAGMA</i>)	-	562.50
	(h2)	Transfer back $\hat{\mathbf{c}}$ from GPU to CPU	-	0.03
	(i)	Zeros computation in CPU	3.60	3.60
		Total Time Taken	1084.9	1131.44

The computational time reported in Table 5.1 for *GVF* is by considering *Approach (a)* for solving residues $\hat{\mathbf{c}}$ of the scaling function $\hat{\sigma}(s)$. Table 5.3 shows the performance comparison when *Approach (b)* is used for the *step (h1) and step (h2)* (see Table 5.2) in a pole refinement iteration.

Table 5.3: Performance comparison of the proposed *GVF* (using *MAGMA* library and *Approach (b)*) versus PVF based on *Multi CPU* approach (for *None Splitting*)

Example	PVF [34] (multi CPU)		Proposed GVF time using MAGMA library (sec)	Speedup
	# CPU cores used	Time (sec)		
#ports = 60 #poles = 70 #fpoints = 1000 #size of $\hat{\Phi}=1000 \times 70$ Total # of $\Phi_l=3600$	1	25.22	1.13	22.31
	2	11.82		10.46
	4	6.01		5.32
	6	4.24		3.75
	8	3.20		2.83
	12	2.38		2.11
	16	1.83		1.62
	20	1.58		1.40

As can be seen from Table 5.1 and Table 5.3, using *Approach (a)* for residue ($\hat{c}$) evaluation of the scaling function $\hat{\sigma}(s)$ yields superior performance.

5.3 QR factorization time in GPU using cuBLAS [41] library: step (e)

In the second part of the experiment, while using the GVF, QR decomposition for the *step (e)* is done using the *cuBLAS* [41] library. Particularly, the routine *cublasDgeqrfBatched* is used for QR decomposition. The total cost in this case for computing the QR for all ($P^2 = 3600$) A_l elements is **3800.25 msec** which is much higher than **543.09 msec** taken by *MAGMA* routine. In this case, GVF needed a total time of **4.34** seconds for one pole refinement iteration.

It is to be noted that, the higher computational efficiency by the *MAGMA* routines is mainly due to the fact that they have been designed to well adopt to the modern heterogeneous architectures of multi-core + multi-GPU systems, focussing on achieving massive parallelism, minimizing the CPU-GPU communication time, and optimal software solutions that combine the strengths of different algorithms within a single framework.

Table 5.4 provides the itemized cost of one pole refinement iteration using *cuBLAS* while using both *Approach (a)* and *Approach (b)*.

Table 5.4: Itemized cost for the proposed *GVF* using *cuBLAS* library (for *None Splitting*)

Example	Steps	Cost of individual steps in GVF (Using <i>cuBLAS</i> [37] library)	Time (msec) (Approach (a))	Time (msec) (Approach (b))
#ports = 60 #poles = 70 #fpoints = 1000 #size of $\hat{\Phi} = 1000 \times 70$ Total # of $\phi_l = 3600$	(a)	Transfer of $\mathbf{S}, \mathbf{g}, \alpha$ from CPU to GPU	20.19	20.19
	(b)	Formulation of $\hat{\Phi}$ in CPU and transfer to GPU	1.97	1.97
	(c), (d)	Formulation of $\mathbf{A}$ in GPU	0.04	0.04
	(e)	QR factorization in GPU (<i>cuBLAS</i>)	3800.25	3800.25
	(f)	$\mathbf{R}_l^{22}$ extraction in GPU	0.02	0.02
	(g)	Transfer back $\mathbf{R}_l^{22}$ from GPU to CPU	56.00	-
	(h)	Residue $\hat{\mathbf{c}}$ calculation in CPU (<i>LAPACK</i>)	460.02	-
	(h1)	Residue $\hat{\mathbf{c}}$ calculation in GPU (<i>MAGMA</i>)	-	562.50
	(h2)	Transfer back $\hat{\mathbf{c}}$ from GPU to CPU	-	0.03
	(i)	Zeros computation in CPU	3.60	3.60
		Total Time Taken	4342.09	4388.60

Table 5.5: Performance comparison of the proposed *GVF* (using *cuBLAS* library) versus PVF [38] based *Multi CPU approach* (for *None Splitting*)

Example	PVF [34] (multi CPU)		Proposed GVF time using <i>cuBLAS</i> library (sec)		Speedup	
	# CPU cores used	Time (sec)	Approach (a)	Approach (b)	Approach (a) v/s PVF	Approach (b) v/s PVF
#ports = 60 #poles = 70 #fpoints = 1000 #size of $\hat{\Phi} = 1000 \times 70$ Total # of $\phi_l = 3600$	1	25.22	4.34	4.38	5.81	5.75
	2	11.82			2.72	2.69
	4	6.01			1.38	1.37
	6	4.24			0.98	0.97
	8	3.20			0.74	0.73
	12	2.38			0.55	0.54
	16	1.83			0.42	0.42
	20	1.58			0.36	0.36

Results based on using *cuBLAS* library are also compared against the PVF on varying number of CPUs (1 to 20) in Table 5.5 in column 3 and column 4 for both the *approaches (a)* and *(b)*. As can be seen from Table 5.5, the proposed GVF yields a speedup of **5.81** against a single core CPU, **2.72** against two cores and **1.38** against 4 cores when using *Approach (a)* and a speedup of **5.75** against a single core CPU, **2.69** against two cores and **1.37** against 4 cores when using

Approach (b).

As can be seen from Table 5.1 and Table 5.5, using *MAGMA* library brings better performance compared to using *cuBLAS* library for the VF iterations.

CPU time comparison of the proposed *GVF* using *Approach (a)* with *cuBLAS* and *MAGMA* libraries versus PVF based on *Multi CPU approach* is given as bar graphs in Figure 5.2 and the corresponding summary of the speed up using various scenarios are given as bar graphs in Figure 5.3.

Figure 5.2: CPU time comparison of the proposed *GVF* using *Approach (a)* with *cuBLAS* and *MAGMA* libraries versus PVF based on *Multi CPU approach* (for *None Splitting*)

Figure 5.3: Speed up comparison of the proposed *GVF* using *Approach (a)* with *cuBLAS* and *MAGMA* libraries versus PVF (*for None Splitting*)

Figure 5.4 shows the CPU time comparison of the proposed *GVF* using *Approach (b)* for the *step (h)* with *cuBLAS* and *MAGMA* libraries versus PVF based on *Multi CPU approach* is given as bar graphs and the corresponding summary of the speed up using various scenarios are given as bar graphs in Figure 5.5.

Figure 5.4: CPU time comparison of the proposed *GVF* using *Approach (b)* with *cuBLAS* and *MAGMA* libraries versus *PVF* based on *Multi CPU approach* (for *None Splitting*)

Figure 5.5: Speed up comparison of the proposed *GVF* using *Approach (b)* with *cuBLAS* and *MAGMA* libraries versus *PVF* (for *None Splitting*)

Figures 5.6 provides a sample accuracy comparison between the converged *VF* model versus the original model using the proposed *GVF* using *Approach (a)* for

Example 1.

Figure 5.6: Accuracy comparison of sample response for Example 1

Example 2: The second example corresponds to a relatively large multiport tab-

ulated S-parameter data of a 120-port (P) interconnect network. Performance evaluation is done for the case of 80 poles (including complex 40 poles and 40 complex conjugate poles) and for 1000 frequency samples is considered and a discussion comparing the *GVF* based on *None Splitting Strategy* with a single GPU versus just the multiple CPUs for one pole refinement iteration is given. Such a scenario leads to each $\mathbf{A}_l$ with a matrix size in (4.29) of (2000×163) where $\hat{\Phi}$ in (4.13) is a matrix of size (1000×80) .

Table 5.6 provides the itemized computational times for one pole refinement iteration using *MAGMA library* while using both *Approach (a)* and *Approach (b)*.

Table 5.6: Itemized cost for the proposed *GVF* using *MAGMA* library (for *None Splitting*)

Example	Steps	Cost of individual steps in GVF (Using <i>MAGMA</i> [36] library)	Time (msec) (Approach (a))	Time (msec) (Approach (b))
#ports = 120 #poles = 80 #fpoints = 1000 #size of $\hat{\Phi}=1000 \times 80$ Total # of $\phi_l=14400$	(a)	Transfer of $\mathbf{S}, \mathbf{g}, \boldsymbol{\alpha}$ from CPU to GPU	74.80	74.80
	(b)	Formulation of $\hat{\Phi}$ in CPU and transfer to GPU	2.18	2.18
	(c), (d)	Formulation of $\mathbf{A}$ in GPU	0.07	0.07
	(e)	QR factorisation in GPU (<i>MAGMA</i>)	2567.20	2567.20
	(f)	$\mathbf{R}_l^{22}$ extraction in GPU	0.05	0.05
	(g)	Transfer back $\mathbf{R}_l^{22}$ from GPU to CPU	290.79	-
	(h)	Residue $\hat{\mathbf{c}}$ calculation in CPU (<i>LAPACK</i>)	2724.25	-
	(h1)	Residue $\hat{\mathbf{c}}$ calculation in GPU (<i>MAGMA</i>)	-	3416.62
	(h2)	Transfer back $\hat{\mathbf{c}}$ from GPU to CPU	-	0.05
	(i)	Zeros computation in CPU	6.47	6.47
		Total Time Taken	5665.81	6067.44

Similar to Example 1, in the first part of the experiment, while using the *GVF*, QR decomposition is done using the *MAGMA* [40] library. As can be seen from Table 5.6, the *GVF* needed only **5.66 sec** for one PR iteration.

Table 5.7: Performance comparison of the proposed GVF (using *MAGMA library*) versus PVF based on *Multi CPU approach* (for *None Splitting*)

Example	PVF [34] (multi CPU)		Proposed GVF time using <i>MAGMA</i> library (sec)		Speedup	
	# CPU cores used	Time (sec)	<i>Approach (a)</i>	<i>Approach (b)</i>	<i>Approach (a)</i> v/s PVF	<i>Approach (b)</i> v/s PVF
#ports = 120 #poles = 80 #fpoints = 1000 #size of $\hat{\Phi}=1000 \times 80$ Total # of $\Phi_l=14400$	1	106.75	5.67	6.07	18.84	17.58
	2	56.89			10.03	9.37
	4	28.87			5.09	4.75
	6	19.67			3.46	3.24
	8	15.38			2.71	2.53
	12	10.95			1.93	1.80
	16	8.62			1.52	1.42
	20	7.37			1.29	1.21

Results are compared against the PVF [38] on varying number of CPUs (1 to 20). Corresponding time taken by the PVF [38] is given in Table 5.7 in column 3 while columns 4 and 5 shows the corresponding time for *approaches (a)* and *(b)*, respectively. As can be seen from Table 5.7, the proposed GVF yields a speedup of **18.84** against a single core CPU, **10.03** against two cores, **1.93** against 12 cores and **1.29** against 20 cores when *Approach (a)* is used. When *Approach (b)* is used, the speed up reduces to **17.58** against a single core CPU, **9.37** against two cores, **1.80** against 12 cores and **1.21** against 20 cores are achieved. As can be seen from Table 5.7, using *Approach (a)* for residue ($\hat{c}$) evaluation of the scaling function $\hat{\sigma}(s)$ yields superior performance.

In the second part of the experiment, while using the GVF, QR decomposition for the *step (e)* is done using the *cuBLAS* [41] library. The total cost in this case for computing the QR for all ($P^2 = 3600$) A_l elements is **20.90 sec** which is much higher than **2567.20 msec** taken by *MAGMA* routine. In this case, GVF needed a total time of **24.01 sec** for one pole refinement iteration.

Table 5.8 provides the itemized computational times for one pole refinement iteration using *cuBLAS library* while using both *Approach (a)* and *Approach (b)*.

Table 5.8: Itemized cost for the proposed *GVF* using *cuBLAS* library (for *None Splitting*)

Example	Steps	Cost of individual steps in GVF (Using <i>cuBLAS</i> [37] library)	Time (msec) (Approach (a))	Time (msec) (Approach (b))
#ports = 120 #poles = 80 #fpoints = 1000 #size of $\hat{\Phi}=1000 \times 80$ Total # of $\Phi_l=14400$	(a)	Transfer of $\mathcal{S}, \mathbf{g}, \alpha$ from CPU to GPU	74.80	74.80
	(b)	Formulation of $\hat{\Phi}$ in CPU and transfer to GPU	2.18	2.18
	(c), (d)	Formulation of $\mathbf{A}$ in GPU	0.07	0.07
	(e)	QR factorisation in GPU (<i>cuBLAS</i>)	20909.54	20909.54
	(f)	$\mathbf{R}_l^{22}$ extraction in GPU	0.05	0.05
	(g)	Transfer back $\mathbf{R}_l^{22}$ from GPU to CPU	290.79	-
	(h)	Residue $\hat{\mathbf{c}}$ calculation in CPU (<i>LAPACK</i>)	2724.25	-
	(h1)	Residue $\hat{\mathbf{c}}$ calculation in GPU (<i>MAGMA</i>)	-	3416.62
	(h2)	Transfer back $\hat{\mathbf{c}}$ from GPU to CPU	-	0.05
	(i)	Zeros computation in CPU	6.47	6.47
		Total Time Taken	24008.15	24409.78

Table 5.9: Performance comparison of the proposed *GVF* (using *cuBLAS library*) versus PVF based on *Multi CPU approach* (for *None Splitting*)

Example	PVF [34] (multi CPU)		Proposed GVF time using <i>cuBLAS</i> library (sec)		Speedup	
	# CPU cores used	Time (sec)	Approach (a)	Approach (b)	Approach (a) v/s PVF	Approach (b) v/s PVF
#ports = 120 #poles = 80 #fpoints = 1000 #size of $\hat{\Phi}=1000 \times 80$ Total # of $\Phi_l=14400$	1	106.75	24.01	24.41	4.45	4.37
	2	56.89			2.37	2.33
	4	28.87			1.20	1.18
	6	19.67			0.82	0.81
	8	15.38			0.64	0.63
	12	10.95			0.46	0.45
	16	8.62			0.36	0.35
	20	7.37			0.31	0.30

Results based on using *cuBLAS* library are also compared against the PVF on varying number of CPUs (1 to 20) in Table 5.9 in column 3, while columns 4 and

5 shows the corresponding time for both the *approaches (a)* and *(b)*. As can be seen from Table 5.9, the proposed GVF yields a speedup of **4.45** against a single core CPU, **2.37** against two cores and **1.20** against 4 cores when *Approach (a)* is used. When *Approach (b)* is used, the speed up reduces to **4.37** against a single core CPU, **2.33** against two cores and **1.18** against 4 cores are achieved. As can be seen from Table 5.7 and Table 5.9, using *MAGMA* library brings significantly better performance compared to using *cuBLAS* library for the VF iterations.

Figure 5.7 shows the CPU time comparison of the proposed *GVF* using *Approach (a)* for the *step (h)* with *cuBLAS* and *MAGMA* libraries versus PVF based on *Multi CPU approach* is given as bar graphs and the corresponding summary of the speed up using various scenarios are given as bar graphs in Figure 5.8.

Figure 5.7: CPU time comparison of the proposed *GVF* using *Approach (a)* with *cuBLAS* and *MAGMA* libraries versus PVF based on *Multi CPU approach (for None Splitting)*

Figure 5.8: Speed up comparison of the proposed *GVF* using *Approach (a)* with *cuBLAS* and *MAGMA* libraries versus *PVF (for None Splitting)*

Figure 5.9 shows the CPU time comparison of the proposed *GVF* using *Approach (b)* for the *step (h)* with *cuBLAS* and *MAGMA* libraries versus *PVF* based on *Multi CPU approach* is given as bar graphs and the corresponding summary of the speed up using various scenarios are given as bar graphs in Figure 5.10.

Figure 5.9: CPU time comparison of the proposed *GVF* using *Approach (b)* with *cuBLAS* and *MAGMA* libraries versus PVF based on *Multi CPU approach* (for *None Splitting*)

Figure 5.10: Speed up comparison of the proposed *GVF* using *Approach (b)* with *cuBLAS* and *MAGMA* libraries versus PVF (for *None Splitting*)

Figure 5.11 provides a sample accuracy comparison between the converged VF model versus the original model using the proposed *GVF* using *Approach (a)* for Example 2.

Figure 5.11: Accuracy comparison of sample response for Example 2

5.4 Discussion on computational aspects and efficiency

From the numerical results, it is evident that the proposed *GVF* yields significantly better results for systems with large number of ports and poles, compared to using *PVF* [38] (for the experiments conducted using up to 20 cores). It is also evident that the *MAGMA* [40] library better results compared to *cuBLAS* [41] library. Bulk of the cost in the proposed *GVF* is in the QR decomposition of the A_l elements and computation of the residues of the scaling function. It is to be noted that, the computational cost in the last step which is evaluation of zeros of the scaling function (via the eigenvalue of the system formed by its residues) is trivial in the case of *None Splitting* Strategy. However, this step becomes significantly expensive when using the *All Splitting* Strategy as will be seen in the next chapter.

To summarize, this chapter introduced the *GVF* based on *None Splitting* Strategy for parallelization of *VF* for computing a common pole set for the entire matrix. It also provided performance comparisons against *PVF* using multi-core CPUs. Also, efficiency comparison is given between *cuBLAS* [41] and *MAGMA* [40] libraries for the same.

Chapter 6

Development of GPU based Vector Fitting (GVF) with All Splitting Strategy

In Chapter 5, development of the proposed *GVF* for *None Splitting Strategy* to compute a common pole-set for the entire S matrix was presented. It is evident from Chapter 5 that even in the presence of large number of ports, data with large number of frequency samples and large number of poles, *GVF* based *None Splitting Strategy* performed better than existing approaches in the literature [38] - [39] based on multi CPUs. Also, the developed approach provided comparison in terms to use of *LAPACK* library (i.e., CPU library) [45] or *MAGMA* library [40] (i.e., GPU library) for residue computation of the scaling function. In addition to that, experimental cost analysis of each task involved in *GVF* based on *None Splitting Strategy* was also given.

In this chapter, development of the *GVF* for *All Splitting Strategy*, wherein each S_l element will have its own individual pole set is given. Emphasis is placed on minimizing the costs of all the blocks associated in pole refinement iterations in VF. During the detailed presentation of the proposed parallelization algorithms for the *All Splitting Strategy*, discussions are given with respect to the *Relaxed VF* case (since it provides better convergence).

6.1 GVF based on All Splitting Strategy

In this section, parallelization of the VF using the *All Splitting Strategy* (wherein an individual pole set is extracted for each S_l element) that are suitable for GPU platforms is given. Parallelization strategy is focused on minimizing all the computational costs: involved in formulation of $\hat{\Phi}$ in (4.13), formulation of $\mathbf{A}$ in (4.10), (4.24) and (4.26), splitting $\tilde{\mathbf{A}}_l$ into real and imaginary parts (4.29) and all other steps leading to residue evaluation of the scaling function (4.35). Developed strategy is focused on parallelizing these tasks on independent P^2 thread blocks in GPU.

In the proposed approach, formulation of both $\hat{\Phi}$ and $\mathbf{A}$ as well as the steps leading to computation of the residue of the scaling function are planned using the GPU. Comparison is made between *MAGMA* [40] and *cuBLAS* libraries [41] for the computational cost of one pole refinement iteration associated with the VF algorithm. Also, the developed approach provides comparison in terms of whether to use *cuBLAS* library [41] or *MAGMA* library [40] for residue computation of individual scaling functions. In the following, detailed steps of the proposed algorithm is given.

- (a) Given data S_l (4.8), $\mathbf{g}$ (4.12) and the weighting vectors α_l (4.27) $\forall l = \{1, 2, \dots, P^2\}$ are bulk transferred from the CPU DRAM to the GPUs global memory.
- (b) Assign $\text{GPU} \leftarrow p_{n,l} \forall n \in \{1, 2, \dots, N\}, \forall l = 1, 2, \dots, P^2$. Note that, in the first iteration, the poles $p_{n,l} \forall n \in \{1, 2, \dots, N\}$ of size $(N \times 1)$ is same for all the ports and are transferred from CPU DRAM to GPU global memory.

Figure 6.1: Flow diagram for the proposed GVF with All Splitting

In every subsequent iterations, improved poles $p_{n,l} \forall n \in \{1, 2, \dots, N\}, \forall l = 1, 2, \dots, P^2$ for all the ports are transferred from CPU DRAM to GPU global memory (note that, since each S_l element has its own scaling function, at the end of each pole refinement iteration, each S -element will have its own individual set of new improved poles).

- (c) Next, $\hat{\Phi}_l$ (4.13) is formulated in parallel in GPU using P^2 blocks and K threads and stored in GPU global memory .
- (d) Φ_l (4.10) and $\tilde{\Phi}_l$ (4.24) matrices are formulated and stored in GPU global memory using P^2 blocks and K threads.
- (e) Φ_l and $\tilde{\Phi}_l$ are then assembled in a block row-wise and are split into real and imaginary parts as in (4.29) to obtain all the A_l s.
- (f) Next, vector λ_l^T (4.31) is computed in GPU and is appended to form each of the A_l s as follows

$$\mathbf{A}_{l(2K+1) \times (2N+3)} = \begin{bmatrix} \Phi_l & \tilde{\Phi}_l \\ 0 & \lambda_l^T \end{bmatrix}_{(2K+1) \times (2N+3)} \quad (6.1)$$

Note that adding λ_l^T in each of the A_l matrix provides the scaling function residues ($\hat{c}$) similar to when adding λ_l^T in each R_l^{22} matrix as in (4.35). Also b_l s are formulated for each S -element as

$$\mathbf{b}_{l(2K+1) \times 1} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ K \end{bmatrix}_{(2K+1) \times 1} \quad (6.2)$$

Note that, the system of equations represented by

$$\begin{bmatrix} \mathbf{A}_l \end{bmatrix}_{(2K+1) \times (2N+3)} \begin{bmatrix} \mathbf{c}_l \end{bmatrix}_{((2N+3) \times 1)} = \begin{bmatrix} \mathbf{b}_l \end{bmatrix}_{(2K+1) \times (1)} \quad (6.3)$$

is an over-determined system which needs to be solved for obtaining the residues $\hat{c}_l$ of the individual scaling function $\hat{\sigma}_l(s)$. Note that there are P^2 such $\mathbf{A}_l$ elements, which need to be solved in parallel for obtaining higher computational efficiency. Solution of this system using the available solvers in GPU platforms faced significant challenges. What is needed for efficient solution of (6.3) is an efficient batched least square solver (using QR), which was not available in *MAGMA*. Hence alternate batched function in *MAGMA* are used to arrive at the solution as in the following steps. These steps are designed to benefit from the available solver *magma_dgesv_batched* for linear set of equations which supports batched option only for square system matrices (whereas we have in $\mathbf{A}_l$, a rectangular matrix).

- (g) To address the above issue, in this step, normal equations are formulated for least square optimization. Each $\mathbf{A}_l$ is sent in a batch to *magma_dgemm_batched* which computes $\mathbf{A}_l^T \times \mathbf{A}_l$. Also, each $\mathbf{A}_l$ along with $\mathbf{b}_l$ is sent to *magma_dgemm_vbatched* to compute $\mathbf{A}_l^T \times \mathbf{b}_l$, to obtain the set of equations represented by

$$\begin{bmatrix} \mathbf{A}_l^T \times \mathbf{A}_l \end{bmatrix}_{(2N+3) \times (2N+3)} \begin{bmatrix} \hat{\mathbf{c}}_l \end{bmatrix}_{(2N+3)} = \begin{bmatrix} \mathbf{A}_l^T \times \mathbf{b}_l \end{bmatrix}_{(2N+3) \times 1} \quad (6.3)$$

Since the resulting $[\mathbf{A}_l^T \times \mathbf{A}_l]$ is a square matrix, (6.3) can be solved using *magma_dgesv_batched* routine, which is done in the next step.

- (h) Next, the $A_l^T A_l$ matrices and $A_l^T b_l$ vectors are sent *magma_dgesv_batched* routine that does the LU factorization, followed by forward and backward substitution to compute residues $\hat{c}_l$ of the corresponding scaling function $\hat{o}_l(s)$ (which is specific to that S_l element).
- (i) Residues $\hat{c}_l$ of matrices for the ports $l = 1, 2, \dots, P^2$ are collected and bulk transferred from GPU global memory to a single CPU.
- (j) Using the form in (3.24), the zeros $\hat{z}_{n,l} \forall n \in \{1, 2, \dots, N\}, \forall l = 1, 2, \dots, P^2$ of the scaling function $\hat{o}_l(s)$ are computed using parallel CPU threads (T). Based on the available number of CPU threads, zeros are computed in approximately $\frac{P^2}{T}$ loops. These zeros serve as the refined poles for starting the next iteration.

NOTE: As will be seen in the Numerical Results in the next section, *step (j)* forms the bulk of the CPU cost in a GPU based VF implementation with *All Splitting*. This is mainly because of the lack of the availability of a non-symmetric matrix eigenvalue finding routine in both *MAGMA* and *cuBLAS* libraries. Also, it is to be noted that this eigenvalue computation process needs to be repeated P^2 times resulting in overall excessive time. On the contrary, the cost of this step is trivial in the case of *None Splitting* as it needs to be done only once (not P^2 times).

In the above flow, for the *step (g)*, *step (h)*, applicability and efficiency of *MAGMA* [40] and *cuBLAS* [41] libraries are investigated. Details and results of the investigation are given in next section.

6.2 Performance comparison of the proposed GVF with All Splitting Strategy on a Single GPU + CPUs Platform versus PVF [38] on multiple CPUs

Performance comparison of the proposed GVF on a single GPU + CPU platform versus the PVF [38] when executed on multiple CPUs is given. For this purpose, two large examples as in the previous chapter are considered under various scenarios. Discussions are given with respect to one full pole refinement iteration of the VF algorithm.

Example 1: In this experiment, same example as in the previous chapter, a multipoint network with 60-port (P), 1000 frequency samples (K) and 70 poles (N) (including complex poles and their conjugate poles) is considered. Such a scenario leads to each A_l in (4.29) with a matrix size of (2000×143) where $\hat{\Phi}$ in (4.13) is a matrix of size (1000×70) . Performance of GVF based on All Splitting Strategy is investigated in a VF pole refinement iteration.

In the first part of this experiment, while using the GVF, *step (g)* and *step (h)* are done using MAGMA library. For the *step (g)*, *magma_dgemm_batched* and *magma_dgemm_vbatched* are used. For the *step (h)*, *magma_dgesv_batched* is used.

Table 6.1: Performance comparison of the proposed *GVF* using *MAGMA* library versus PVF based on *Multi CPU* approach (for *All Splitting*)

Example	PVF [34] (multi CPU)		Proposed GVF time using <i>MAGMA</i> library (sec)	Speedup
	# CPU cores used	Time (sec)		
#ports = 60 #poles = 70 #fpoints = 1000 #size of $\hat{\Phi}=1000 \times 70$ Total # of $\phi_l=3600$	1	42.00	5.81	7.23
	2	17.84	2.67	6.69
	4	9.06	1.40	6.48
	6	6.10	1.04	5.88
	8	4.62	0.85	5.45
	12	3.20	0.71	4.52
	16	2.53	0.66	3.85
	20	2.27	0.61	3.73

Results are compared against the PVF [38] on varying number of CPUs (1 to 20). Corresponding time taken by the PVF [38] is given in Table 6.1 in column 3. As can be seen from Table 6.1, the proposed GVF yields a speedup of **7.23** against a single core CPU, **6.69** against two cores, **5.45** against 8 cores, **4.52** against 12 cores and **3.73** against 20 cores

In the following sections, further experimental cost analysis of each task associated with the *GVF* based on *All Splitting Strategy* is given.

6.2.1 Data (S , g and α) transfer cost from CPU to GPU: step (a)

In this section, transfer time required for transferring data (S , g and α) from CPU to GPU is given. Bulk of this cost corresponds to the cost associated with transferring the data (S) of the given multiport matrix.

Note that, as seen in Figure 6.1, in *step (a)* for the first PR iteration, data corresponding to S , g and α need to be transferred from CPU to GPU. The transfer

time required for these data from CPU to GPU is **20.19 msec**. This step is not required in the subsequent iterations.

6.2.2 *Transfer of poles $p_{n,l}$: step (b)*

In the first as well as subsequent iterations, $p_{n,l}$ (in *step (b)*) is transferred from CPU to GPU. Note that each S_l element will have its own pole set. The cost involved in transferring all $p_{n,l} \forall n \in \{1, 2, \dots, N\}, \forall l = 1, 2, \dots, P^2$ from CPU to GPU in subsequent iterations is only **1.52 msec**. This data transfer cost associated with the subsequent VF iterations is minimal considering the overall cost of a pole refinement iteration.

6.2.3 *Formulation of $\hat{\Phi}$: step (c)*

Next, matrix $\hat{\Phi}_l$ (i.e., the computation involved in (4.13)) is formulated for each S_l element. For this purpose, a customized GPU batch function is developed that uses P^2 blocks and K threads to formulate all P^2 $\hat{\Phi}_l$ matrices (with $P^2 = 3600$) parallely across multiple threads and blocks. Total time taken for formulation of all $P^2 = 3600$ $\hat{\Phi}_l$ matrices is only **0.06 msec** in GPU.

6.2.4 *Formulation of A_l and b_l : step (d), step (e) and step (f)*

Once $\hat{\Phi}_l$ s are formulated, matrix A_l (i.e., the computations involved in (4.10), (4.24), (4.26), (4.29) and (6.1)) needs to be formulated for each S_l element. For this purpose, a customized GPU batch function is developed that uses P^2 blocks and K threads to formulate all P^2 A_l matrices and b_l vectors (with $P^2 = 3600$) parallely across multiple threads and blocks. Total time taken for formulation of

all P^2 A_l matrices and b_l vectors is only **0.04 msec** in GPU.

6.2.5 Computation time of $A_l^T \times A_l$, $A_l^T \times b_l$ using MAGMA [40] library: step (g)

Once all A_l matrices are formulated, each A_l is sent to *magma_dgemm_batched* and *magma_dgemm_bbatched* routines that compute $A_l^T \times A_l$ and $A_l^T \times b_l$ for each S_l element, respectively. The total time taken for computing $A_l^T \times A_l$ is **124.28 msec** and $A_l^T \times b_l$ is **10.87 msec**. Hence this step needs a total time of $124.28 \text{ msec} + 10.87 \text{ msec} = \mathbf{135.15 \text{ msec}}$ (for all P^2 elements).

6.2.6 LU decomposition, forward/backward substitution time in GPU using MAGMA [40] library: step (h)

Once the $A_l^T \times A_l$, $A_l^T \times b_l$ are computed, they are sent to *magma_dgesv_batched* routine. This routine computes the L and U factors for each $A_l^T \times A_l$ followed by forward/backward substitution to compute the residues $\hat{c}_l$ of the scaling function $\hat{\sigma}_l(s)$. The total time involved for residue computation of all individual scaling functions (i.e., *step (g)*) is only **17.32 msec**.

6.2.7 Data transfer cost from GPU to CPU: step (i)

After *step (h)*, the individual residues $\hat{c}_l$ of the scaling function $\hat{\sigma}_l(s)$ for each S_l element are transferred from GPU to CPU. The transfer costs associated with this is only **3.45 msec**.

NOTE: All the steps from *step (a)* to *step (i)* needed a total time of **177.73 msec**

for computation in GPU and related data transfers. However, as will be noted the next step adds comparatively significant additional cost, computation of zeros.

6.2.8 Computation of zeros of the scaling function $\hat{\sigma}_l(s)$: step (h)

Once the individual residues $\hat{c}_1, \dots, \hat{c}_n$ for all the ports $l = 1, 2, \dots, P^2$ are obtained, zeros ($\hat{z}_{n,l}$) of $\hat{\sigma}_l(s)$ are calculated in the multi-core CPUs using the form in (3.24). Based on the available number of threads, zeros are computed in approximately $\frac{P^2}{T}$ loops. Table 6.2 below provides the time taken for zeros computation using varying number of CPUs (1 to 20).

Table 6.2: Computational time for zeros computation in CPUs (for *All Splitting*)

Example	Zeros calculation (multi CPU)	
	# CPU cores used	Time (msec)
#ports = 60 #poles = 70 #fpoints = 1000 #size of $\hat{\Phi} = 1000 \times 70$ Total # of $\Phi_l = 3600$	1	5630
	2	2490
	4	1220
	6	860
	8	670
	12	530
	16	480
	20	430

NOTE: It was investigated using both the *MAGMA* and *cuBLAS* libraries for the zeros computation in GPU. However, the cost came out to be significantly higher than the one reported in Table 6.2. Hence it was decided to use the CPUs for zeros computation.

Table 6.3 provides the itemized computational times for one pole refinement iteration for the proposed *GVF* for *All Splitting Strategy* using *MAGMA* library.

Table 6.3: Itemized cost for the proposed *GVF* using *MAGMA* library (for *All Splitting*)

Example	Steps	Cost of individual steps in GVF (Using <i>MAGMA</i> [36] library)	Time (msec)
#ports = 60 #poles = 70 #fpoints = 1000 #size of $\hat{\Phi}=1000 \times 70$ Total # of $\phi_l=3600$	(a)	Transfer of $\mathbf{S}, \mathbf{g}, \boldsymbol{\alpha}$ from CPU to GPU	20.19
	(b)	Transfer of p_n from CPU to GPU	1.52
	(c)	Formulation of $\hat{\Phi}$ in GPU	0.06
	(d), (e), (f)	Formulation of $\mathbf{A}_l, \mathbf{b}_l$ in GPU	0.04
	(g)	Computation of $\mathbf{A}_l^T \mathbf{A}_l$ and $\mathbf{A}_l^T \mathbf{b}_l$ (<i>MAGMA</i>)	135.15
	(h)	Residue evaluation $\hat{\mathbf{c}}_l$ (<i>MAGMA</i>)	17.32
	(i)	Transfer back $\hat{\mathbf{c}}_l$ from GPU to CPU	3.45
		Total time taken in GPU and related data transfer	177.73
	(j)	Zeros computation in T CPUs	Refer Table 6.2

6.3 Using *cuBLAS* [41] library for the step (g) and step (h)

In the second part of the experiment, while using *GVF*, steps (g) and (h) are done using *cuBLAS* [41] library. After the step (f), once each $\mathbf{A}_l$ and $\mathbf{b}_l$ are formulated, they are sent to *dgels_batched* routine of *cuBLAS* library. This routine computes solution ($\hat{\mathbf{c}}_l$, the residues of the scaling function $\hat{\sigma}_l(s)$) based on a least square formulation and solution via QR decomposition and backward substitution.

The flowchart corresponding to the proposed *GVF* extracting individual pole-set using *cuBLAS* library is given in Figure 6.2

Figure 6.2: Flow diagram for the proposed GVF (for All Splitting)

The total cost in this case for computing QR factorization and individual residues $\hat{\mathbf{c}}_l$ of the scaling function $\hat{\sigma}_l(s)$ is **5829.10 msec**. In this case, *GVF* needs a total time of **5847.35 msec** excluding zeros calculation. Whereas when using *MAGMA* library *GVF* needs a total time of **177.73 msec** excluding zeros calculation for one pole refinement iteration. Therefore, using *MAGMA* [40] library brings better performance than using *cuBLAS* library.

Table 6.4 provide the itemized cost of one pole refinement iteration using *cuBLAS* library.

Table 6.4: Itemized cost for the proposed *GVF* using *cuBLAS* library (for *All Splitting*)

Example	Steps	Cost of individual steps in GVF (Using <i>cuBLAS</i> [37] library)	Time (msec)
#ports = 60 #poles = 70 #fpoints = 1000 #size of $\hat{\Phi}=1000 \times 70$ Total # of $\phi_l=3600$	(a)	Transfer of $\mathbf{S}, \mathbf{g}, \boldsymbol{\alpha}$ from CPU to GPU	20.19
	(b)	Transfer of $\mathbf{p}_n$ from CPU to GPU	1.52
	(c)	Formulation of $\hat{\Phi}$ in GPU	0.06
	(d), (e), (f)	Formulation of $\mathbf{A}_l, \mathbf{b}_l$ in GPU	0.04
	(g), (h)	<i>QR factorization and residue $\hat{\mathbf{c}}_l$ calculation in GPU (cuBLAS)</i>	5829.10
	(i)	Transfer back $\hat{\mathbf{c}}_l$ from GPU to CPU	23.44
		Total time taken in GPU and related data transfer	5847.35
	(j)	Zeros computation in <i>T</i> CPUs	<i>Refer Table 6.2</i>

Table 6.5: Performance comparison of the proposed *GVF* using *cuBLAS* library versus PVF based on *Multi CPU approach* (for *All Splitting*)

Example	PVF [34] (multi CPU)		Proposed GVF time using <i>cuBLAS</i> library (sec)	Speedup
	# CPU cores used	Time (sec)		
#ports = 60 #poles = 70 #fpoints = 1000 #size of $\hat{\Phi}=1000 \times 70$ Total # of $\phi_l=3600$	1	42.00	11.50	3.65
	2	17.84	8.36	2.13
	4	9.06	7.09	1.27
	6	6.10	6.73	0.90
	8	4.62	6.51	0.70
	12	3.20	6.40	0.50
	16	2.53	6.35	0.37
	20	2.27	6.30	0.36

Results are compared against the PVF [38] on varying number of CPUs (1 to 20). Corresponding time taken by the PVF [38] is given in Table 6.5 in column 3. As can be seen from Table 6.5, the proposed GVF yields a speedup of **3.65** against a single core CPU, **2.13** against two cores, **1.27** against 4 cores. As seen from Table 6.5 and Table 6.1, using *MAGMA* [40] library brings better performance compared to *cuBLAS* library for the VF iterations.

CPU time comparison of the proposed *GVF* using *cuBLAS* library versus PVF based on *Multi CPU approach* is given as bar graphs in Figure 6.3 and the corresponding summary of the speed up are given as bar graphs in Figure 6.4.

Figure 6.3: CPU time comparison of the proposed *GVF* using *cuBLAS* and *MAGMA* libraries versus PVF based on *Multi CPU approach* (for All Splitting)

Figure 6.4: Speed up comparison of the proposed *GVF* using *cuBLAS* and *MAGMA* libraries versus PVF (*for All Splitting*)

Figures 6.5 provides a sample accuracy comparison between the converged VF model versus the original model using the proposed *GVF* using *MAGMA* library for Example 1.

Figure 6.5: Accuracy comparison of sample response for Example 1

Example 2: In the second example, same as in Chapter 5, corresponds to a relatively large multiport tabulated S-parameter data of a 120-port (P) interconnect network. Performance evaluation is done for the case of 80 poles (including complex 40 poles and 40 complex conjugate poles) and for 1000 frequency samples is considered. Such a scenario leads to each $\mathbf{A}_l$ with a matrix size in (4.29) of (2000×163) where $\hat{\Phi}$ in (4.13) is a matrix of size (1000×80) .

Similar to Example 1, a discussion comparing the *GVF* based on *All Splitting Strategy* with a single GPU versus just the multiple CPUs for one pole refinement iteration is given. In the first part of the experiment, while using the *GVF*, *step (g)* and *step (h)* are done using *MAGMA* library.

Table 6.6 provides the itemized computational times for one pole refinement while using *MAGMA* [40] library. Corresponding zeros computation time is given in Table 6.7.

Table 6.6: Itemized cost for the proposed *GVF* using *MAGMA* library (for *All Splitting*)

Example	Steps	Cost of individual steps in GVF (Using <i>MAGMA</i> [36] library)	Time (msec)
#ports = 120 #poles = 80 #fpoints = 1000 #size of $\hat{\Phi}=1000 \times 80$ Total # of $\phi_l=14400$	(a)	Transfer of $\mathbf{S}, \mathbf{g}, \boldsymbol{\alpha}$ from CPU to GPU	74.80
	(b)	Transfer of p_n from CPU to GPU	4.55
	(c)	Formulation of $\hat{\Phi}$ in GPU	0.12
	(d), (e), (f)	Formulation of $\mathbf{A}_l, \mathbf{b}_l$ in GPU	0.07
	(g)	Computation of $\mathbf{A}_l^T \mathbf{A}_l$ and $\mathbf{A}_l^T \mathbf{b}_l$ (<i>MAGMA</i>)	687.89
	(h)	Residue evaluation $\hat{\mathbf{c}}_l$ (<i>MAGMA</i>)	87.69
	(i)	Transfer back $\hat{\mathbf{c}}_l$ from GPU to CPU	7.4
		Total time taken in GPU and related data transfer	862.52
	(j)	Zeros computation in T CPUs	Refer Table 6.7

Table 6.7: Computational time for zeros computation in CPUs (for *All Splitting*)

Example	Zeros calculation (multi CPU)	
	# CPU cores used	Time (msec)
#ports = 120 #poles = 80 #fpoints = 1000 #size of $\hat{\Phi}=1000 \times 80$ Total # of $\Phi_l=14400$	1	42000
	2	20350
	4	10240
	6	7860
	8	5170
	12	3570
	16	2950
	20	2830

Table 6.8: Performance comparison of the proposed *GVF* using *MAGMA* library versus PVF based on *Multi CPU approach* (for *All Splitting*)

Example	PVF [34] (multi CPU)		Proposed GVF time using <i>MAGMA</i> library (sec)	Speedup
	# CPU cores used	Time (sec)		
#ports = 120 #poles = 80 #fpoints = 1000 #size of $\hat{\Phi}=1000 \times 80$ Total # of $\Phi_l=14400$	1	193.90	42.86	4.52
	2	95.45	21.21	4.50
	4	48.40	11.10	4.36
	6	36.80	8.72	4.22
	8	25.03	6.03	4.15
	12	17.71	4.43	4.00
	16	14.42	3.81	3.78
	20	12.23	3.69	3.31

Results are compared against the PVF [38] on varying number of CPUs (1 to 20). Corresponding time taken by the PVF [38] is given in Table 6.8 in column 3 and the proposed GVF yields a speedup of **4.52** against a single core CPU, **4.15** against 8 cores, **4.00** against 12 cores, **3.78** against 16 cores and **3.31** against 20 cores.

In the second part of the experiment, *step (g)* and *step (h)* are done using *cuBLAS* library. Table 6.9 provides the itemized computational times for one pole refine-

ment while using *cuBLAS* [41] library.

Table 6.9: Itemized cost for the proposed *GVF* using *cuBLAS* library (for *All Splitting*)

Example	Steps	Cost of individual steps in GVF (Using <i>cuBLAS</i> [37] library)	Time (msec)
#ports = 120 #poles = 80 #fpoints = 1000 #size of $\hat{\Phi}=1000 \times 80$ Total # of $\Phi_l=14400$	(a)	Transfer of $\mathbf{S}, \mathbf{g}, \boldsymbol{\alpha}$ from CPU to GPU	74.80
	(b)	Transfer of p_n from CPU to GPU	4.55
	(c)	Formulation of $\hat{\Phi}$ in GPU	0.12
	(d), (e), (f)	Formulation of $\mathbf{A}$ in GPU	0.07
	(g), (h)	<i>QR factorization and residue $\hat{\mathbf{c}}_l$ calculation in GPU (<i>cuBLAS</i>)</i>	30174.02
	(i)	Transfer back $\hat{\mathbf{c}}_l$ from GPU to CPU	90.25
		Total time taken in GPU and related data transfer	30343.81
	(j)	Zeros computation in T CPUs	<i>Refer Table 6.6</i>

Table 6.10: Performance comparison of the proposed *GVF* with *cuBLAS* library versus PVF based on *Multi CPU approach* (for *All Splitting*)

Example	PVF [34] (multi CPU)		Proposed GVF time using <i>cuBLAS</i> library (sec)	Speedup
	# CPU cores used	Time (sec)		
#ports = 120 #poles = 80 #fpoints = 1000 #size of $\hat{\Phi}=1000 \times 80$ Total # of $\Phi_l=14400$	1	193.90	72.34	2.68
	2	95.45	50.69	1.88
	4	48.40	40.58	1.19
	6	36.80	38.20	0.96
	8	25.03	35.51	0.70
	12	17.71	33.91	0.52
	16	14.42	33.29	0.43
	20	12.23	33.17	0.37

As seen, the proposed GVF yields a speedup of **2.68** against a single core CPU, **1.88** against 2 cores and **1.19** against 4 cores. As can be seen from Table 6.8 and Table 6.10, using *MAGMA* library brings better performance than *cuBLAS* library for the proposed *GVF* based on *All Splitting*.

CPU time comparison of the proposed *GVF* using *MAGMA* and *cuBLAS* libraries

versus PVF based on *Multi CPU approach* is given as bar graphs in Figure 6.6 and the corresponding speedup is given as bar graphs in Figure 6.7.

Figure 6.6: CPU time comparison of the proposed *GVF* versus PVF based on *Multi CPU approach (for All Splitting)*

Figure 6.7: Speed up comparison of the proposed *GVF* using *cuBLAS* and *MAGMA* libraries versus PVF (*for All Splitting*)

Note that in all the examples presented, the CPU speedup is reported with respect to one pole refinement iteration. The performance enhancement is achieved by parallelizing the computational tasks in each pole refinement iteration. It is to be noted that after the first iteration, for subsequent iterations, only the first step of transferring the multiport data from CPU to GPU can be skipped and all other steps would be needed; Hence except for this cost, when considering subsequent iterations, the CPU cost can be expected in a similar range.

Figures 6.8 provides an accuracy comparison between the converged VF model versus the original model for sample S-parameter elements. As seen, results are in good agreement.

Figure 6.8: Accuracy comparison of sample response for Example 2

6.4 Discussion on computational aspects and efficiency

As is obvious from Examples 1 and 2 in this chapter, the proposed *GVF* with *All Splitting* while using *MAGMA* library performs significantly better compared to the case when using *cuBLAS* library. Availability of a batched QR decomposition based linear least square solver in *MAGMA* library could have provided an alternate approach to the normal equations (6.3) used in this chapter. However, this option is currently not available in *MAGMA* library. What is obvious in the case of *All Splitting* case is that, bulk of the computational cost in a pole refinement iteration is in the last *step* (*j*) for computing the zeros of the scaling function (whereas this cost is trivial in the case of *None Splitting* Strategy). This is because there is no batched eigenvalue solver for non-symmetric matrices currently available in *MAGMA* and *cuBLAS* libraries. Due to this reason, for GPU platforms, overall computational efficiency of the *None Splitting* Strategy is better when working with large number of ports and poles compared to *All Splitting* case.

To conclude, this chapter presented the computational steps for parallelization of VF based on *All Splitting Strategy* to compute individual pole set for each *S* element. Performance comparisons are made between *MAGMA* [40] and *cuBLAS* [41] libraries. It is observed that *MAGMA* library [40] provides better speed-up compared to *cuBLAS* libraries. Also, it is observed that *All Splitting Strategy* suffers in the case of many ports compared to *None Splitting Strategy* during the computation of zeros of the scaling function due to the lack of availability of batched eigen value solver for non-symmetric matrices (both in *MAGMA* [40] and *cuBLAS* [41] libraries).

Chapter 7

Conclusion and Future work

This thesis introduced advancement of the VF algorithm to exploit the emerging massively parallel computing platforms of GPUs. In this thesis, two important parallelization strategies for GPU platforms were discussed based on both the *None Splitting* and *All Splitting* based approaches.

- a) GVF based on *None Splitting Strategy* was proposed to provide enhanced parallelization for computing a common pole set for the entire multiport S matrix. It minimized all the computational costs associated with the VF for formulation, QR factorization and computation of scaling function residues.
- b) GVF based on *All Splitting Strategy* is developed to provide enhanced parallelization for computing an individual pole set for each S element of the multiport S matrix. Similar to the above strategy, this strategy minimized the computational costs associated with all the blocks in a pole refinement iteration of VF (i.e., formulation, QR factorization and computation of residues of the scaling function).
- c) Also, a detailed cost analysis of each of the steps in VF iterations in both the strategies were performed. A comparison was provided between *MAGMA* and *cuBLAS* libraries for the computational costs associated in the VF algorithm. Results indicate that using *MAGMA* libraries yield far superior results compared to using *cuBLAS* library.

As seen from the performance results, proposed *GVF* method better computational performance compared to VF based on single CPU or multi-CPU based existing parallel approaches depending on the number of cores used.

7.1 *Future work*

This thesis presented parallelization strategies based on the existing GPU architectures and CUDA programming options. It is to be noted that both of these are rapidly advancing and changing, implying that the proposed algorithms can be further advanced or modified to fully exploit the future advances in GPU architectures and programming. This thesis explored the possible options for each of the computational tasks in VF. It explored both state-of-the-art libraries *MAGMA* [40] and *cuBLAS* [41] for this purpose. Particularly for the last step of computing the zeros as eigenvalues based on a system of matrices formed using the residues of the scaling function, they are calculated in CPUs (instead of GPU) due to the lack of batched eigenvalue routine for non-symmetric matrices in both *MAGMA* [40] and *cuBLAS* [41] libraries. In a future work, this could be addressed to obtain even better efficiency using GPU platforms.

References

- [1] H. B. Bakoglu, *Circuits, Interconnections and Packaging for VLSI*, Addison-Wesley, Reading MA, 1990.
- [2] W. M. M. Dai (Guest Editor), "Special issue on simulation, modeling, and electrical design of high-speed and high-density interconnects," *IEEE Transactions on Circuits and Systems*, vol. 39, no. 11, pp. 857-82, November 1992.
- [3] M. Nakhla, R. Achar and R. Khazaka, *Modeling and Simulation of High-Speed VLSI Interconnects*, CHAPTER IV: *Circuits and Systems In the Information Age*, New Jersey: IEEE Publishers, pp. 187-215, 1997.
- [4] A. Deustsch, "Electrical characteristics of interconnections for high-performance systems," *Proceedings of the IEEE*, vol.86, no. 2, pp. 315-355, February 1998.
- [5] C. Paul, *Analysis of Multiconductor Transmission Lines*, New York: John Wiley and Sons Inc., 1994.
- [6] ITRS, "*ITRS Report on Interconnect Technology, 2009 Edition*," 2009. [Online]. Available: www.itrs2.net/itrs-reports.html
- [7] ITRS, "*ITRS Report on Interconnect Technology, 2012 Edition*," 2012. [Online]. Available: www.itrs2.net/itrs-reports.html
- [8] ITRS, "*ITRS Report on Interconnect Technology, 2013 Edition*," 2013. [Online]. Available: <http://www.itrs2.net/itrs-reports.html>

- [9] IRDS, "*International Road-map for Devices and Systems, 2017 Edition*, 2017.
[Online]. Available <https://irds.ieee.org/images/files/pdf/2017/2017IRDS.pdf>
- [10] HSPICE Applications Manual [Online].
- [11] R. Khazaka, J. Poltz, M. Nakhla, and Q. J. Zhang, "A fast method for the simulation of lossy interconnects with frequency dependent parameters," *Proceedings IEEE Multi-Chip Module Conference*, Santa Cruz, CA, pp. 95-98, February 1996.
- [12] S. D. Corey and A. T. Yang, "Interconnect characterization using time-domain reflectometry", *IEEE Transactions on Microwave Theory and Techniques*, Issue: 9, pp. 2155-2156, September 1995.
- [13] W. T. Beyene, and J. E. Schutt-Aaine, "Accurate frequency-domain modelling and simulation of high-speed packaging interconnects", *IEEE Transactions Microwave Theory Technology*, pp. 1941-1947, October 1997.
- [14] J. E. Schutt-Aaine and R. Mittra, "Scattering parameter transient analysis of transmission lines loaded with nonlinear terminations," *IEEE Transactions on Microwave Theory Technology*, vol. 36, pp. 529-536, 1988.
- [15] A. R. Djordjevic and T. K. Sarkar, "Analysis of lossy transmission lines with arbitrary nonlinear terminal networks", *IEEE Transactions on Microwave Theory and Techniques*, vol. 34, no. 6, pp. 660-666, June 1986.
- [16] D. Winklestein, M. B. Steer, and R. Pomeerleau, "Simulation of arbitrary transmission line networks with nonlinear terminations," *IEEE Transactions on Circuit Systems*, vol. 38, pp. 418-422, April 1991.

- [17] R. Achar and M. Nakla, "Efficient transient Simulation of embedded sub-networks characterized by S-parameters in the presence of nonlinear elements," *IEEE Transactions on Microwave Theory and Techniques*, vol. MTT-46, pp. 2356-2363, December 1998.
- [18] M. Elzinga, K. L. Virga and J. L. Prince, "Pole-residue formulation for transient simulation of high-frequency interconnects using householder LS curve-fitting techniques," *IEEE Transactions on Advanced Packaging*, vol. 25, pp. 142-147, May 2000.
- [19] M. Elzinga, K. L. Virga and J. L. Prince, "Improved Global Rational Approximation Macromodeling Algorithm for Networks Characterized by Frequency-Sampled Data," *IEEE Transactions on Microwave Theory and Techniques*, vol. 48, no. 9, September 2000.
- [20] W. T. Beyene and J. E. Schutt-Aine, "Efficient Transient Simulation of High-Speed Interconnects Characterized by Sampled Data," *IEEE Transactions on Components, Packaging, and Manufacturing Technology, Part B*, vol. 21, no. 1, pp. 105-114, February, 1998.
- [21] W. T. Beyene and J. E. Schutt-Aine, "Interconnect Simulation Using Order Reduction and Scattering Parameters," *IEEE Electronic Components and Technology Conference*, pp. 627-631, 1998.
- [22] C. P. Coelho, J. R. Philips, and L. M. Silveria, "Robust Rational Function Approximation Algorithm for Model Generation." *Proceedings of 36th Design Automation Conference*", PP. 207-212, New Orleans, June 1999.

- [23] D. Saraswat, R. Achar and M. Nakhla, " A Fast Algorithm and Practical Considerations for Passive Macromodeling of Measured/Simulated Data," *IEEE 11th Topical Meeting on Electrical Performance of Electronic Packaging*, Monterey, CA, pp. 297-300, October 2002.
- [24] A. Lin and E. S. Kuh, "Transient simulations of lossy interconnects based on the recursive convolution formulation," *IEEE Transactions on Circuits and Systems*, vol. 39, no. 11, pp. 879-892, November 1992.
- [25] L. M. Silveria, I. M. Elfadel, J. K. White, M. Chilukuri and K. S. Kundert, "An efficient approach to transmission line simulation using measured or tabulated S-parameter data," in *Proceedings ACM/IEEE Electronic Components and Technology Conference*, pp. 634-639, June 1994.
- [26] K. J. Kerns and A. T. Yang, "Stable and efficient reduction of large, multi-port RC networks by pole analysis via congruence transformations," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 280-285, June 1995.
- [27] K. J. Kerns and A. T. Yang, "Preservation of passivity during RLC network reduction via split congruence transformations," *IEEE Transactions on Computer-Aided Design*, vol. 17, pp. 582-591, July 1998.
- [28] A. Odabasioglu, M. Celik and L. T. Pillage, "PRIMA: Passive Reduced-Order Interconnect Macromodeling Algorithm:, *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pp. 58-65, November 1997.

- [29] I. M. Elfadel and D. D. Ling, "A block rational Arnoldi algorithm for multiport passive model-order reduction of multiport RLC networks," *1997 Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pp. 66-71, November 1997.
- [30] Q. Yu, J. M. L. Wang and E. S. Kuh, "Multipoint multiport algorithm for passive reduced-order model of interconnect networks, " *ISCAS '98. Proceedings of the 1998 IEEE International Symposium on Circuits and Systems*, vol. 6, pp. 74-77, June 1998.
- [31] R. Achar and M. Nakhla, "Addressing passivity issues in CFH", *Proceedings 2nd IEEE International Workshop on Signal Propagation on Interconnects*, pp. 33-34, May 1998, Hannover, Germany.
- [32] R. Achar, P. Gunnupudi, M. Nakhla and E. Chiprout, "Passive Interconnect Reduction Algorithm for Distributed/Measured Networks", *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, pp. 287-301, April 2000.
- [33] S. Min and M. Swaminathan, "Efficient construction of two port passive macromodels for resonant networks," in *IEEE 10th Topical Meeting Electrical Performance of Electronic Packaging and Systems (EPEPS)*, Cambridge, MA, October 2001, pp. 229-232.
- [34] M. Elzinga, K. L. Virga and J. L. Prince, "Improved Global Rational Approximation Macromodeling Algorithm for Networks Characterized by Frequency-Sampled Data," *IEEE Transactions on Microwave Theory and Techniques*, vol. 48, no. 9, September 2000.

- [35] B. Gustavsen and A. Semlyen, "Rational Approximation of Frequency Domain Responses by Vector Fitting," *IEEE Transactions on Power Delivery*, vol. 14, no. 3, pp. 1052-1061, July 1999.
- [36] B. Gustavsen, "Improving the pole relocating properties of vector fitting," *IEEE Transactions on Power Delivery*, vol. 21, no. 3, pp. 1587-1592, August 2006.
- [37] D. Deschrijver, M. Mrozowski, T. Dhaene and D. De Zutter, "Macromodeling of Multiport Systems Using a Fast Implementation of the Vector Fitting Method," *IEEE Microwave and Wireless Components Letters* vol. 18, no. 6, June 2008.
- [38] A. Chinae and S. Grivet-Talocia, "On the Parallelization of Vector Fitting Algorithms", *IEEE Transactions on Components, Packaging and Manufacturing Technology*, vol. 1, no. 11, November 2011.
- [39] N. K. Elumalai, "Parallelization of Vector Fitting Algorithm for GPU platforms", MAsc. thesis, Carleton University, Ottawa, May 2019.
- [40] *MAGMA* <https://icl.cs.utk.edu/magma/>
- [41] *cuBLAS* <https://docs.nvidia.com/cuda/cublas/index.html>
- [42] *Vector Fitting* <https://www.sintef.no/projectweb/vectfit/>
- [43] *OpenMP* <https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf>
- [44] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd edition, Baltimore, MD: Johns Hopkins University Press, 1996.

- [45] *LAPACK* <http://www.netlib.org/lapack/>
- [46] K. H. Jin, M. T. McCann, E. Froustey, and M. Unser, “Deep convolutional neural network for inverse problems in imaging,” *IEEE Transactions on Image Processing*, vol. 26, no. 9, pp. 4509–4522, September 2017.
- [47] S. Hussain, R. C. P. Silva, and D. A. Lowther, “Implementation of iron loss model on graphic processing units,” *IEEE Transactions on Magnetics*, vol. 52, no. 3, pp. 1–4, March 2017.
- [48] Y. Liang, X. Xing, and Y. Li, “A GPU-based large-scale Monte Carlo simulation method for systems with long-range interactions,” *Computational Physics*, vol. 338, pp. 252–268, June 2017.
- [49] W.-K. Lee, R. C.-W. Phan, G.-S. Poh, and B.-M. Goi, “SearchaStore: Fast and secure searchable cloud services,” *Cluster Computing*, vol. 21, pp. 1189–1202, June 2018.
- [50] X.-X. Liu, H. Yu, and S. X.-D. Tan, “A GPU-accelerated parallel shooting algorithm for analysis of radio frequency and microwave integrated circuits,” *IEEE Transactions on Very Large Scale Integrated (VLSI) Systems*, vol. 23, no. 3, pp. 480–492, March 2015.
- [51] B. D. de Vos, J. M. Wolterink, P. A. de Jong, T. Leiner, M. A. Viergever, and I. Isgum, “ConvNet-based localization of anatomical structures in 3-D medical images,” *IEEE Transactions on Medical Imaging*, vol. 36, no. 7, pp. 1470–1481, July 2017.
- [52] *Compute Canada*. Accessed: April 2019. [Online]. Available: <https://www.computecanada.ca/home/>

- [53] N. K. Elumalai, S. Ganeshan, R. Achar, "GVF based Vector Fitting", *IEEE International conference on Electrical Performance of Electronic Packaging and Systems (EPEPS - 2019)*, Montreal, October 2019 (accepted for publication).