GPU-ENHANCED POWER FLOW ANALYSIS

Manuel Marin

THESIS

To obtain the degree of

Doctor of Philosophy

From

UNIVERSITÉ DE PERPIGNAN VIA DOMITIA

and **UNIVERSITY COLLEGE DUBLIN**

Prepared at

École Doctorale Énergie et Environnement

and **School of Electrical and Electronic Engineering**

In the research units **DALI-LIRMM** and **ERC**

Field: **Computer Science and Electrical Engineering**

Presented by **Manuel Marin**

GPU-ENHANCED POWER FLOW ANALYSIS

Defended on December 11th, 2015, before the jury composed of

Dr David Defour	MCF HDR	Université de Perpignan	Adviser
Dr Federico Milano	Associate Professor	University College Dublin	Adviser
Dr Didier Elbaz	CR HDR	LAAS-CNRS	Reviewer
Dr Angelo Brambilla	Professor	Politecnico Milano	Reviewer
Dr Raphaël Couturier	Professor	IUT Belfort-Montbéliard	Examiner
Dr Terence O'Donnell	Senior Lecturer	University College Dublin	Examiner

Abstract

This thesis addresses the utilization of Graphics Processing Units (GPUs) to improve the Power Flow (PF) analysis of modern power systems. GPUs are powerful vector co-processors that have been very useful in the acceleration of several computational intensive applications. PF analysis is the steady-state analysis of AC power networks and is widely used for several tasks involved in system operation and planning. Currently, GPUs are challenged by applications exhibiting an irregular computational pattern, as is the case of most known methods to solve the PF analysis. At the same time, the PF analysis needs to be improved in order to cope with new requirements of efficiency and accuracy coming from the Smart Grid concept. The relevance of GPU-enhanced PF analysis is twofold. On one hand, it expands the application domain of GPUs to a new class of problems. On the other hand, it consistently increases the computational capacity available for power system operation and design. What specific features of GPUs can be used to enhance crucial aspects of PF analysis? What algorithms or models for PF analysis are best suited to the massively parallel architecture of GPUs? The present work attempts to answer such questions in two complementary ways: (i) by developing novel GPU programming strategies for available PF algorithms, and (ii) by proposing novel PF analysis methods that can exploit the numerous features present in GPU architectures. The proposed methods are implemented using state-of-the-art programming frameworks and paradigms, and tested over state-of-the-art GPU and CPU architectures. Results are discussed and compared with existing solutions from the literature of the field, leading to relevant conclusions and guidelines for future research. Specific contributions to GPU computing include: (i) a comparison of two programming paradigms, namely regularity and load-balancing, for implementing the so-called treefix operations; (ii) a study of the impact of the representation format over performance and accuracy, for fuzzy interval algebraic operations; and (iii) the utilization of architecture-specific design, as a novel strategy to improve performance scalability of scientific applications. Contributions to PF analysis include: (i) design and evaluation of a novel method for the uncertainty assessment, based on the fuzzy interval approach; and (ii) the development of an intrinsically parallel method to solve the PF analysis, which is not affected by the Amdahl's law.

Acknowledgments

I would like to express my most sincere gratitude to my advisers, David and Federico, for their continuous guidance and support. I learned a lot from them during these years, about research and life, and I am sure I will continue to do so in the future.

My thanks also go to my fellow labmates in DALI and ERC, for all the stimulating discussions and exchanges, lunches and dinners. I am very much looking forward to the next time we cross paths.

Last but not least, I would like to thank my family and friends, who were always there to give me their support and to ask me to explain my thesis, once again. For this and more I could never be grateful enough.

Contents

Nomenclature

Complex numbers	
j	The imaginary unit.
$\mathbb{x}$	A phasor.
x^*	The complex conjugate of x.
$\lvert x\rvert$	The magnitude of x.
$\mathrm{Re}(x)$	The real part of x.
$\mathrm{Im}(x)$	The imaginary part of x.
Linear algebra	
$\boldsymbol{x}$	A vector.
$\boldsymbol{A}$	A matrix.
$\boldsymbol{J_f}$	The Jacobian matrix of $\boldsymbol{f}$.
$\mathrm{Diag}(\boldsymbol{x})$	The diagonal matrix that has $\boldsymbol{x}$ as diagonal.
$\lVert\boldsymbol{x}\rVert$	The norm of $\boldsymbol{x}$.
Floating-point arithmetic	
$\triangle$	The rounding attribute towards ∞.
$\triangledown$	The rounding attribute towards $-\infty$.
$\square$	The rounding attribute towards the nearest.
ε	The relative rounding error (i.e., the machine epsilon).
η	The smallest representable (denormalized) floating-point positive number.
Interval analysis	
$[x]$	An interval.
$[\tilde{x}]$	A fuzzy interval.
$\underline{x}$	The lower bound of $[x]$.
$\overline{x}$	The upper bound of $[x]$.
$\check{x}$	The midpoint of $[x]$; alternatively, the kernel of $[\tilde{x}]$.
ρ_x	The radius of $[x]$.
$\mu(x)$	A membership function.
$[x_i]$	The ith α-cut of $[\tilde{x}]$.
$\rho_{x,i}$	The radius of $[x_i]$.
$\delta_{x,i}$	The radius increment associated to $[x_i]$.
Probability theory	
X	A random variable.
$\mathrm{Pr}(A)$	The probability of event A.
F_X	The distribution function of X.
f_X	The density function of X.
$E(X)$	The expected value of X.
$f_X \otimes f_Y$	The convolution product of f_X and f_Y.

Asynchronous power flow method

n_h	The node h.
c_{hm}	The component situated between n_h and n_m.
$\mathcal{N}$	The set of circuit nodes.
$\mathcal{C}$	The set of circuit components.
$\mathcal{I}_h$	The set of influencers of n_h.
$\mathbb{v}_h$	The voltage at n_h.
$\mathbb{u}_{hm}$	The voltage at n_h according to c_{hm}.
$\mathbb{i}_{hm}$	The current flowing from n_m to n_h.
$\Delta\mathbb{i}_h$	The current mismatch at n_h.
z_{hm}	The impedance of c_{hm}.
$\hat{\mathbb{v}}_{hm}$	The voltage source of c_{hm}.
w_{hm}	The weight within n_h of c_{hm}.
$\mathbb{s}_{hm}$	The power flowing from n_m to n_h.
$\Delta\mathbb{s}_h$	The power mismatch at n_h.
$\mathbb{s}_{L,h}$	The power injected by loads to n_h.

Abbreviations

GPU	Graphics Processing Unit.
CPU	Central Processing Unit.
HPC	High Performance Computing.
SIMT	Single Instruction Multiple Thread.
TLP	Thread-Level Parallelism.
ILP	Instruction-Level Parallelism.
SM	Streaming Multiprocessor.
API	Application Program Interface.
PF	Power Flow.
NR	Newton-Raphson.
BFS	Backward-Forward Sweep.
AC	Alternating Current.
FDPF	Fast Decoupled Power Flow.

Chapter 1

Introduction

Contents

1.1 Overview

In the last decade, Graphics Processing Units (GPUs) have gained popularity in accelerating calculations thanks to massive parallelism. Originally intended to optimize graphics rendering, they have quite naturally extended their domain of application to the most diverse fields [Pharr 2005, Nguyen 2007]. To date, several computing intensive applications have been ported to GPUs, dramatically changing the landscape of high performance computing (HPC) [Keckler 2011]. At the same time, Power Flow (PF) analysis has consolidated as a most relevant tool in power systems management and planning, as the shift towards the Smart-Grid concept takes place [Amin 2005]. By providing a snapshot of the network, PF analysis allows for several security-related tasks (e.g., the reliability assessment) to be efficiently performed [Milano 2010]. Several software packages are available that implement the different methods for PF analysis proposed in the literature [Simulator 2005, Milano 2013, Simulink 1993].

GPUs are challenged by several applications, e.g., applications that do not exhibit a regular computational pattern [Collange 2010]. This is precisely the case of most algorithms for PF analysis (e.g., the Newton-Raphson (NR) method and the Backward-Forward Sweep (BFS) [Milano 2010, Kersting 2012]). At the same time, PF analysis software need to be enhanced in order to satisfy new requirements of efficiency and accuracy [Momoh 2009, Fang 2012]. The interest of GPU-based PF analysis is twofold. On one hand, it expands the application domain of GPUs to a new class of problems, opening the possibility for similar tools to benefit from the same computing power. On the other hand, it consistently increases the computational capacity available for power system operation and design.

In the present work, the above problem is addressed in two ways. First, by taking existing algorithms for PF analysis and exploring the possibilities and issues of a GPU-based implementation. Second, by developing novel methods for PF analysis specifically designed to run on such architectures. The proposed methods are implemented using state-of-the-art programming frameworks and paradigms, and tested over recent GPU and CPU architectures. The results are discussed and compared with existing solutions from the literature of the field, leading to relevant conclusions and guidelines for future research.

The remainder of this chapter is organized as follows. Section 1.2 provides a preliminary discussion of GPUs and PF analysis. Section 1.3 exposes the main problem addressed in this thesis and its significance. The method of the investigation is stated in Section 1.4. Finally, Section 1.5 outlines the main contributions of the present research.

1.2 Context

This section introduces the state-of-the-art of GPUs and PF analysis at the time of writing this thesis. The discussion is thoroughly expanded in Chapters 2 and 3.

1.2.1 Graphics Processing Units

The development of GPUs started during the 1990s, largely motivated by the multimedia industry and the necessity of rendering graphics quickly at an affordable cost [Owens 2008, Nickolls 2010]. GPUs were originally conceived as powerful vector co-processors, capable of computing the information for thousands of pixels at a time. For this purpose, several (somewhat rudimentary) computing cores were disposed together in a configuration allowing for data-parallelism to be exploited. The Single Instruction Multiple Thread (SIMT) execution model was adopted, and a specific memory hierarchy was implemented in order to satisfy the particular needs of graphics developers. Two main manufacturers captured a large portion of the market: Nvidia and ATI. From then on, several generations of GPUs have been released, embedding more and more computing cores, as well as larger memory spaces and newer features. Various supercomputers and calculation clusters around the world adopted GPUs as part of their environment, and GPUs consolidated as a key element for high performance computing.

At the beginning, GPUs were devoted to graphics rendering. The architectural design was dictated by the specification of the programming language, e.g., DirectX and OpenGL. Programming for general applications was only possible through cleverly and carefully manipulating graphics primitives from those dedicated languages. The situation drastically changed when, by the middle of the past decade, general purpose GPU programming frameworks made their appearance. For the first time, developers were able to exploit the potential of GPUs on any application that would fit the GPU architectural model. Two of these frameworks have gained popularity: CUDA, developed by Nvidia, and OpenCL, by the Khronos Group [Cuda 2012, Stone 2010]. Thanks to these frameworks, the growth of GPU applications has been exponential. Many GPU applications nowadays achieve speed-ups of several orders of magnitudes over their serial CPU versions [Che 2008, Volkov 2008].

Before the raise of GPU technology, the performance of serial programs was automatically enhanced through increasing the frequency of CPU cores. No major effort was required from the developers: a program simply runs faster on a faster machine. However, by the end of the 1990's, the development of CPUs started hitting several walls. The first of them was the "power wall": since the power consumed by the processor grows exponentially with the frequency, it became increasingly difficult to cool high performance CPU cards. The second was the "memory wall": CPUs were evolving much faster than memories, and memory access latency prevented most applications to achieve peak CPU performance. The third was the "instruction-level parallelism (ILP) wall": the instruction *pipeline*, used by CPUs to enhance performance, was under-utilized by programs having insufficient independent instructions to execute in parallel [Asanovic 2006].

Interestingly, the stall of CPUs coincided with GPUs reaching their maturity in terms of architecture and programming. The computing power of GPUs was seen as an opportunity to keep increasing performance through the use of parallel programming. However, this time software developers were bound to play an important part, as programs designed for CPUs had to be re-written in order to run on GPUs. Developers faced a double challenge: (i) discovering the parallelism inherent in their application; and (ii) exposing that parallelism to the GPU in a special, novel programming language (such as CUDA and OpenCL).

The above did not prevent GPUs from disrupting into the world of HPC. A novel research field named General Purpose GPU programming (GPGPU) would emerge to accompany and confirm the success of GPUs [Du 2012]. GPGPU is concerned with the following question: how to write parallel code in order to extract the maximum benefit of various GPU architectural features? GPGPU has revealed several guidelines to answer such a question: guidelines to hide latency, to maximize utilization of floating-point units and to access memory more efficiently [Harris 2012]. Applications exhibiting a regular computation pattern and data structure proved to fit best the SIMT paradigm. Several examples of such successful experiences can be found in the recent literature on HPC. Currently, the challenge is to move from such "embarrassingly parallel" applications to a wider variety of tools, including, for instance, computations over graphs and sparse matrix operations.

1.2.2 PF analysis

PF analysis is the steady-state analysis of Alternating Current (AC) power systems. Given the specification of all electrical components in the network, the PF analysis determines the system operating point and is a tool for analysts and stakeholders to make better decisions and improve system security [Milano 2010].

PF analysis has been at the core of power system analysis since the 1950s. System operators have used it to anticipate and prevent the effects of critical contingencies, but also to optimize performance under a series of economical and political constraints. System planners have used it to compare expansion scenarios and select the ones that fit the best [Machowski 1997]. New applications continue to be added to the list, as the system evolves into newer and more complex configurations. Recently, the introduction of massive renewable energy generation, smart metering and load demand response is challenging system operators and planners, as part of the shift into the so-called Smart Grid concept [Momoh 2009]. The PF analysis appears to be a crucial tool in accompanying and consolidating this crucial process.

The standard PF formulation specifies loads in terms of real and reactive power consumptions; generators in terms of real power injections and rated voltages; and transmission lines in terms of susceptances and impedances. Since the power flowing through an impedance is a function of the product of voltage and current, solving the PF problem implies dealing with a highly-coupled, often ill-conditioned system of non-linear equations. In the past, a linearized model based on direct current (DC) was used to find an approximate solution. Nonetheless, this method can lead to an accuracy loss of up to 50% and is not considered in this work. On the other hand, Newton-based methods have proven to be fairly efficient and soon became the industry standard. Gauss-based methods have also been investigated, although their relevance is nowadays more academical than practical. In addition, the BFS method is largely used for the analysis of radial distribution networks.

For PF analysis to be applied to power system operation, certain standards of efficiency and accuracy need to be fulfilled. Due to computational constraints, the above did not happen until the late 1960s [Stott 1974]. At that point, key improvements in the digital processor allowed a variant of the Newton method, namely the Fast Decoupled Power Flow (FDPF), to be appropriately suited for industry. The FDPF is based on a fixed slope to move from one approximation to the next, instead of factorizing the system Jacobian matrix at each iteration. With time, numerous improvements in sparse matrix factorization made the full NR method increasingly efficient, and now several software packages implement it as the basic general purpose approach [Milano 2013].

To date, most attempts to improve performance of PF analysis have consisted in accelerating certain routines involved in the Newton method. Different libraries for sparse matrix factorization have been tried out, e.g. SuperLU, UMFPACK, KLU, among others [Li 2005, Davis 2004, Davis 2010]. Simultaneously, various forms of non-deterministic PF analysis have been proposed to provide accurate results in scenarios of high uncertainty [Alvarado 1992]. As power systems become more heterogeneous, the sources of uncertainty increase, and the numerical quality of PF solutions needs to be reassessed. A possible approach consists in providing results in terms of intervals or distributions, which allows the analysis of several possible scenarios at once. Currently, the trend is to combine all these improvements (and others to come) into a comprehensive PF analysis tool, for the enhancement of modern and future power systems [Vaccaro 2013].

1.3 Problem statement

The SIMT execution model adopted by GPUs allows the same instruction to be executed over different data by a large number of parallel threads. GPU programming is achieved through *kernel* functions, which contain the instructions to be executed by every thread within a multidimensional array. At the architectural level, threads are scheduled for execution by batches (of 32 or 64, depending on the manufacturer). All threads in the batch execute the same instruction, and memory accesses are served with full lines of cache to an entire batch. For an application to sufficiently exploit this model, all threads in a batch should follow the same execution path. Also, they should access data in the same line of cache. This means that applications exhibiting a regular computation pattern automatically achieve better performance.

The NR method for PF analysis computes successively better approximations of the zero of a non-linear function describing the system. The method moves from one point to the next one by following the tangent to the function at the current point, which is done by factorizing the system Jacobian matrix. To the date, most matrix factorization algorithms rely on some sort of reordering followed by block factorization and backward substitution. The process as a whole is characterized by strong data dependencies, making the application of the SIMT execution model extremely difficult. The underlying data structure, in turn, is highly irregular, as the power network is typically a sparse meshed graph. This makes GPU memory model difficult to exploit.

In synthesis, neither the SIMT execution model nor the GPU memory model can be properly exploited by the NR algorithm for PF analysis. However, GPUs exhibit many other features that can become a leverage for either performance or accuracy. Similarly, PF analysis can be performed by other methods, either available or newly developed. This thesis is confronted with the question of how to efficiently use GPUs to improve PF analysis in a general way. This means understanding GPU architecture in deep detail, such that novel programming guidelines can be proposed. It also means rediscovering PF analysis from the point of view of parallel computing. What specific features of GPUs can be used to enhance crucial aspects of PF analysis? What algorithms or models for PF analysis are best suited to massively parallel computing architectures? These questions have not been sufficiently explored.

The problem stated above spans two independent areas of knowledge. Accordingly, its significance is twofold. On one hand, there is the enormous benefit of expanding the applicability of GPUs to new areas. PF analysis is exactly the kind of applications that the current trend in GPU computing is trying to assess. That is to say, applications with no evident data parallelism in their work-flow, exhibiting an irregular computation pattern, with a scattered data structure, and so on. Being able to adopt PF analysis as one of GPU applications certainly opens the door for others to follow.

On the other hand, enhancing PF analysis is crucial in the current worldwide scenario of power systems. The shift towards the Smart Grid means novel technologies being introduced to the network and new services to be provided. The ability to analyze large, complex and heterogeneous networks efficiently and accurately is key for the success of that process. The above is particularly true as PF analysis is at the core of many other relevant analysis over power systems. From this point of view, any improvement either in the time of response, the numerical quality of the results or the comprehensiveness of the solution, can have a big impact on the way the whole system is conceived and dealt with.

1.4 Methodology

The very nature of the problem, i.e., how to efficiently use GPUs to enhance various aspects of PF analysis, allows it to be addressed in two complementary ways. The first consists in developing novel GPU programming strategies for available PF algorithms. The second is to propose novel PF analysis methods that can exploit the numerous features present in GPU architectures.

Each aspect of the methodology requires a deep understanding of both GPU computing and PF analysis. Accordingly, the first part of the thesis, comprising Chapters 2 and 3, is devoted to investigate the state-of-the-art in these two fields. The peculiarities of GPUs are discussed in Chapter 2. Several details of the execution model, memory hierarchy and programming frameworks are exposed. As features are described, several guidelines and strategies for GPU programming are discussed.

PF analysis is studied in Chapter 3, starting from the fundamentals of circuit theory and AC circuit analysis. After stating the PF problem, relevant solution methods are described. The above includes deterministic and non-deterministic methods, the latter oriented to deal with uncertainty. It also includes methods for fully meshed networks, i.e., with arbitrary topology, and methods for radial networks. As methods are described, relevant implementation issues are discussed.

The second part of the thesis, comprising Chapters 4, 5 ad 6, is focused on developing the field. Based on the understanding of the state-of-the-art, several research directions are proposed. These are obtained by combining one crucial aspect of each subject, i.e., GPU computing and PF analysis, which results in a particular viewpoint. The viewpoint originates a novel technique, or improves an existing technique in an original way. The proposed technique is first studied and validated from a theoretical point of view. Then, the technique is implemented and tested using state-of-the-art programming methods and computing architectures. The results are discussed and compared with results obtained using other methods from existing literature. The assessment of the proposed methods is carried out considering real-world problems. Based on the above, guidelines for further research are finally proposed. In particular, Chapter 4 studies different ways of implementing the so-called treefix operations on GPUs, for the analysis of radial power networks. Chapter 5 proposes a novel GPU fuzzy interval arithmetic implementation, and its application to uncertainty modelling in PF analysis. Chapter 6 proposes a novel fixed-point method for PF analysis which is intrinsically parallel, and discusses its implementation using asynchronous iterations on the GPU.

1.5 Summary of contributions

The present work intends to bring the following main contributions. To GPU programming:

- **Comparison of programming paradigms.** The analysis of different treefix approaches in Chapter 4 brings an answer to the problem of regularity versus load-balancing on GPUs for this kind of problem. Which approach is better for GPU programming? On one hand, regularity exploits better the SIMT execution model, but may introduce extra operations and data transfer. On the other hand, load-balancing keeps the number of operations to a minimum, but it may generate a computational pattern unsuited to SIMT.

- **Impact of the representation format.** Chapter 5 provides a comparison of different interval representation formats for fuzzy algebra. Is it possible to improve performance or accuracy of fuzzy applications by using a specific representation format? Certain peculiarities of the membership function can be exploited by the midpoint-radius format, however, most available fuzzy algebra implementations overlook that fact, relying exclusively on lower-upper.

- **Architecture-specific design.** GPU computing is a relatively recent technology. Therefore, most existing GPU-based approaches for numerical analysis and simulation rely on parallelizing already available techniques. What about using our knowledge of the GPU to design new methods, specifically suited for the implementation on such massively parallel architectures? The asynchronous PF method proposed in Chapter 6 is one of the first attempts to achieve that goal.

To PF analysis:

- **Acceleration of radial network analysis.** Chapter 4 proposes a novel vectorial implementation of the BFS algorithm, based on the study of treefix operations. How much faster can this solution be, compared to the traditional approach based on sequential tree traversals? Experimental results show that a speedup of 2 to 9 can be achieved, depending on several implementation features.

- **Uncertainty assessment.** Chapter 5 proposes a novel method for the uncertainty assessment in PF analysis. Current interval approaches are affected by the dependency problem, which causes a critical loss of accuracy in the final results. Is it possible to overcome that issue? In the proposed method, the above is achieved by using an optimization problem to compute the radius of the solution. The proposed method provides an accuracy similar to that of the Monte Carlo approach, with a much lower computational complexity.

- **Intrinsically parallel PF analysis.** Chapter 6 proposes a novel fixed-point method for PF analysis, which is intrinsically parallel. How advantageous this approach can be, compared to parallelizing existing serial approaches? Two features of the proposed method are particularly interesting: (i) performance does not depend as much on the problem size, as it does in traditional PF approaches; and (ii) performance is able to scale with the number of computing cores, automatically benefiting from improvements in the computer architecture.

1.6 List of publications

The present work has originated the following publications. In peer-reviewed international conferences:

- D. Defour and M. Marin. *Regularity Versus Load-balancing on GPU for Treefix Computations.* Procedia Computer Science, 2013 International Conference on Computational Science, vol. 18, pp. 309 – 318, 2013.

- D. Defour and M. Marin. FuzzyGPU: *A Fuzzy Arithmetic Library for GPU.* Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on, pp. 624–631, 2014.

- M. Marin, D. Defour and F. Milano. *Power flow analysis under uncertainty using symmetric fuzzy arithmetic.* IEEE PES General Meeting, Conference & Exposition, pp. 1–5, 2014.

In national journals:

- D. Defour and M. Marin. *Simulation temps réel de réseaux électriques à l'aide des architectures multicœurs.* UPVD Magazine Hors-Série recherche, no. 3, pp. 42-44, 2014.

- D. Defour and M. Marin. *Optimiser la représentation des flottants.* HPC Magazine France, vol. 4, pp. 65-70, 2013.

Under review:

- M. Marin., D. Defour and F. Milano *An efficient representation format for fuzzy intervals based on symmetric membership function.* Submitted to ACM Transactions on Mathematical Software.

References

[Alvarado 1992] F. Alvarado, Y. Hu and R. Adapa. *Uncertainty in power system modeling and computation.* Systems, Man and Cybernetics, 1992., IEEE International Conference on, pp. 754–760. IEEE, 1992. (Cited on pages 10 and 48.)

[Amin 2005] S. M. Amin and B. F. Wollenberg. *Toward a smart grid: power delivery for the 21st century.* Power and Energy Magazine, IEEE, vol. 3, no. 5, pp. 34–41, 2005. (Cited on page 7.)

[Asanovic 2006] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams*et al.* *The landscape of parallel computing research: A view from berkeley.* Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006. (Cited on page 9.)

[Che 2008] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer and K. Skadron. *A performance study of general-purpose applications on graphics processors using CUDA.* Journal of parallel and distributed computing, vol. 68, no. 10, pp. 1370–1380, 2008. (Cited on pages 9 and 19.)

[Collange 2010] S. Collange. *Design challenges of GPGPU architectures: specialized arithmetic units and exploitation of regularity.* PhD thesis, Université de Perpignan, 2010. (Cited on pages 7 and 26.)

[Cuda 2012] C. Cuda. *Programming guide.* NVIDIA Corporation, July, 2012. (Cited on pages 9, 24, 25, 27, 88 and 92.)

[Davis 2004] T. A. Davis. *Algorithm 832: UMFPACK V4. 3—an unsymmetric-pattern multifrontal method.* ACM Transactions on Mathematical Software, vol. 30, no. 2, pp. 196–199, 2004. (Cited on page 10.)

[Davis 2010] T. A. Davis and E. Palamadai Natarajan. *Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems.* ACM Transacctions on Mathematical Software, vol. 37, no. 3, pp. 36:1–36:17, 2010. (Cited on pages 10 and 124.)

[Du 2012] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson and J. Dongarra. *From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming.* Parallel Computing, vol. 38, no. 8, pp. 391–407, 2012. (Cited on page 9.)

[Fang 2012] X. Fang, S. Misra, G. Xue and D. Yang. *Smart Grid – The New and Improved Power Grid: A Survey.* IEEE Communications Surveys Tutorials, vol. 14, no. 4, pp. 944–980, 2012. (Cited on page 7.)

[Harris 2012] M. Harris. *GPGPU. org.* General Purpose Computation on Graphics Hardware, 2012. (Cited on page 9.)

[Keckler 2011] S. Keckler, W. Dally, B. Khailany, M. Garland and D. Glasco. *GPUs and the Future of Parallel Computing.* IEEE Micro, vol. 31, no. 5, pp. 7–17, 2011. (Cited on page 7.)

[Kersting 2012] W. H. Kersting. Distribution system modeling and analysis. CRC Press, 2012. (Cited on pages 7 and 37.)

[Li 2005] X. S. Li. *An overview of SuperLU: Algorithms, implementation, and user interface.* ACM Transactions on Mathematical Software, vol. 31, no. 3, pp. 302–325, 2005. (Cited on page 10.)

[Machowski 1997] J. Machowski, J. Bialek and J. Bumby. Power system dynamics and stability. Wiley, 1997. (Cited on pages 10 and 36.)

[Milano 2010] F. Milano. Power system modelling and scripting. Springer Science & Business Media, 2010. (Cited on pages 7, 10, 37, 120 and 124.)

[Milano 2013] F. Milano. *A Python-based software tool for power system analysis.* IEEE Power and Energy Society General Meeting, pp. 1–5, 2013. (Cited on pages 7, 10, 68, 98, 122 and 124.)

[Momoh 2009] J. Momoh*et al.* *Smart grid design for efficient and flexible power networks operation and control.* IEEE Power Systems Conference and Exposition, 2009, pp. 1–8, 2009. (Cited on pages 7 and 10.)

[Nguyen 2007] H. Nguyen. Gpu gems 3. Addison-Wesley Professional, 2007. (Cited on page 7.)

[Nickolls 2010] J. Nickolls and W. Dally. *The GPU Computing Era.* Micro, IEEE, vol. 30, no. 2, pp. 56–69, 2010. (Cited on page 8.)

[Owens 2008] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips. *GPU computing.* Proceedings of the IEEE, vol. 96, no. 5, pp. 879–899, 2008. (Cited on pages 8 and 19.)

[Pharr 2005] M. Pharr and R. Fernando. Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation. Addison-Wesley Professional, 2005. (Cited on page 7.)

[Simulator 2005] P. Simulator. *Version 10.0 SCOPF.* PVQV, PowerWorld Corporation, Champaign, IL, vol. 61820, 2005. (Cited on page 7.)

[Simulink 1993] M. Simulink and M. Natick. *The Mathworks*, 1993. (Cited on page 7.)

[Stone 2010] J. E. Stone, D. Gohara and G. Shi. *OpenCL: A parallel programming standard for heterogeneous computing systems.* Computing in science & engineering, vol. 12, no. 1-3, pp. 66–73, 2010. (Cited on pages 9 and 24.)

[Stott 1974] B. Stott. *Review of load-flow calculation methods.* Proceedings of the IEEE, vol. 62, no. 7, pp. 916–929, 1974. (Cited on pages 10 and 32.)

[Vaccaro 2013] A. Vaccaro, C. A. Cañizares and K. Bhattacharya. *A range arithmetic-based optimization model for Power Flow analysis under interval uncertainty.* IEEE Transactions on Power Systems, vol. 28, no. 2, pp. 1179–1186, 2013. (Cited on pages 11, 46, 49, 95, 96 and 98.)

[Volkov 2008] V. Volkov and J. W. Demmel. *Benchmarking GPUs to tune dense linear algebra.* IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11, 2008. (Cited on pages 9 and 27.)

Chapter 2

Graphics Processing Units

Contents

2.1 Introduction

This chapter describes several features of Graphics Processing Units (GPUs), focusing on architectural aspects. It does not aim to be an exhaustive review but rather provide the basis for the discussion that will follow in Chapters 4, 5 and 6.

GPUs have gained popularity lately thanks to their high computational power, attractive cost, programmability and availability. Applications ported to GPUs can achieve a substantial speed-up over their single-core versions, and this performance gain is able to scale with the number of cores [Owens 2008, Che 2008]. Writing an application for the GPU, however, requires understanding the architectural model. Specifically, programmers need to expose the data parallelism of their applications by using some dedicated programming language. In addition, they may need to tune their implementations in order to achieve maximum efficiency [Ryoo 2008]. This chapter presents the most relevant features of GPU architecture, and discusses how these features can be exploited in order to achieve maximum performance.

The remainder of the chapter is organized as follows. Section 2.2 introduces the GPU, detailing the hardware architecture, execution model, memory hierarchy and main programming frameworks. In Section 2.3, three relevant factors for GPU performance are discussed: regularity, latency hiding and instruction throughput. These factors are presented along with several strategies to leverage performance of

GPU applications. Finally, Section 2.4 draws conclusions and outlines GPU issues discussed in following chapters.

2.2 Hardware architecture

GPUs are powerful co-processors designed to exploit data parallelism. This section presents an overview of key aspects of GPU architecture, focusing particularly on Nvidia architectures.

Modern GPU architectures embed hundreds to thousands of computing cores, along with several memories, caches, and dedicated units. These elements are organized into so-called Streaming Multiprocessors (SM). Figure 2.1 presents a schematic of the Nvidia Fermi GPU architecture [Glaskowsky 2009]. The entire board is shown in Fig. 2.1(a), and Fig. 2.1(b) shows an individual SM.

At high level, the host interface communicates the GPU with the 'host' processor, which launches the task (typically, a CPU). In this context, the GPU is called the 'device'. GPU global memory allows both processors to exchange data. The thread-block scheduler distributes blocks of threads to different SMs for their execution. Additionally, the L2 cache allows data caching for redundant global memory accesses made by SMs.

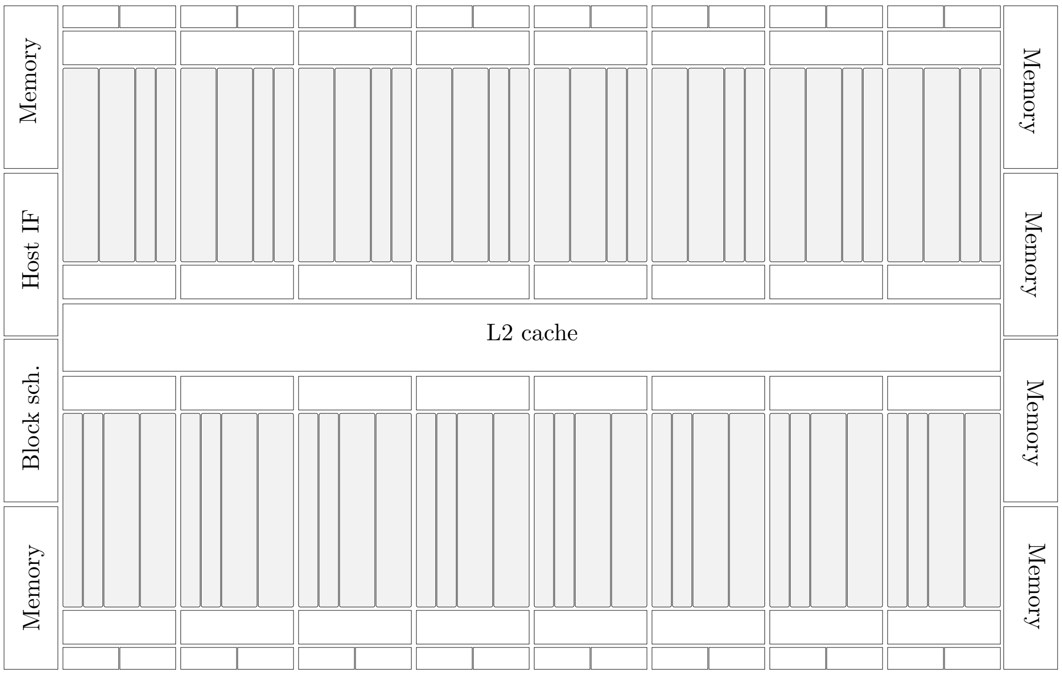
At SM level, the thread scheduler and dispatch units distribute threads to the different execution units. The latter include scalar processors (SP), special function units (SFU) and load/store units. SPs execute integer and floating-point arithmetic instructions, whereas SFUs accelerate in hardware the execution of transcendental instructions such as sine, cosine, square root and interpolation. Load/store units calculate source and destination addresses.

The SM register file provides registers for threads execution. The L1 cache allows additional data caching for global memory accesses made by different threads. Finally, shared memory allows different threads to explicitly share data.

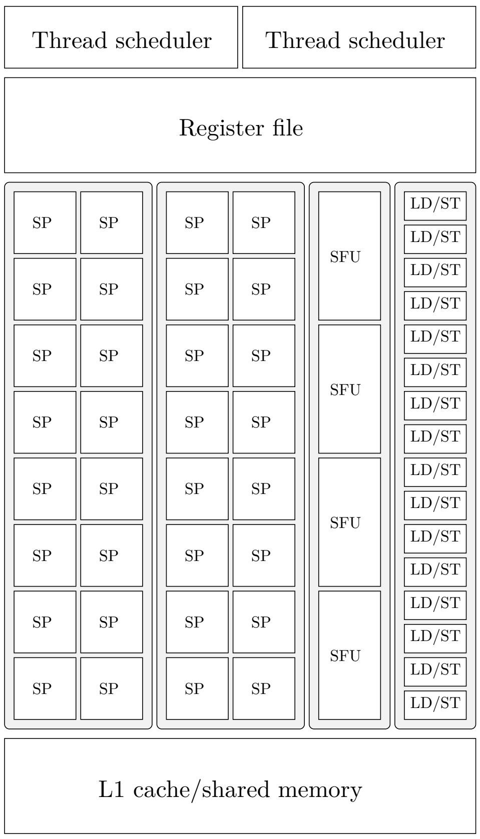
2.2.1 Execution model

GPUs rely on a Single Instruction Multiple Thread (SIMT) execution model, where a *kernel* program is launched over a grid of thread-blocks. The above is illustrated in Fig. 2.2. Grid and blocks can be up to three-dimensional. Each block has a unique identifier within the grid, and each thread has a unique identifier within its block. These two can be used together to compute a unique global identifier for each thread.

The thread-block scheduler distributes blocks of threads to the different SMs. Each block is executed on a single SM, however, each SM can have multiple blocks running concurrently. The number of blocks that can run on a given SM depends on the resources of the SMs compared to the requirements of each block. Within the SM, threads are scheduled by batches of a fixed size to the different execution units. In Nvidia architectures, 32 threads are scheduled together and called a *warp*.



(a)



(b)

Figure 2.1: Nvidia Fermi architecture: (a) Graphics processing unit. (b) Streaming multiprocessor.

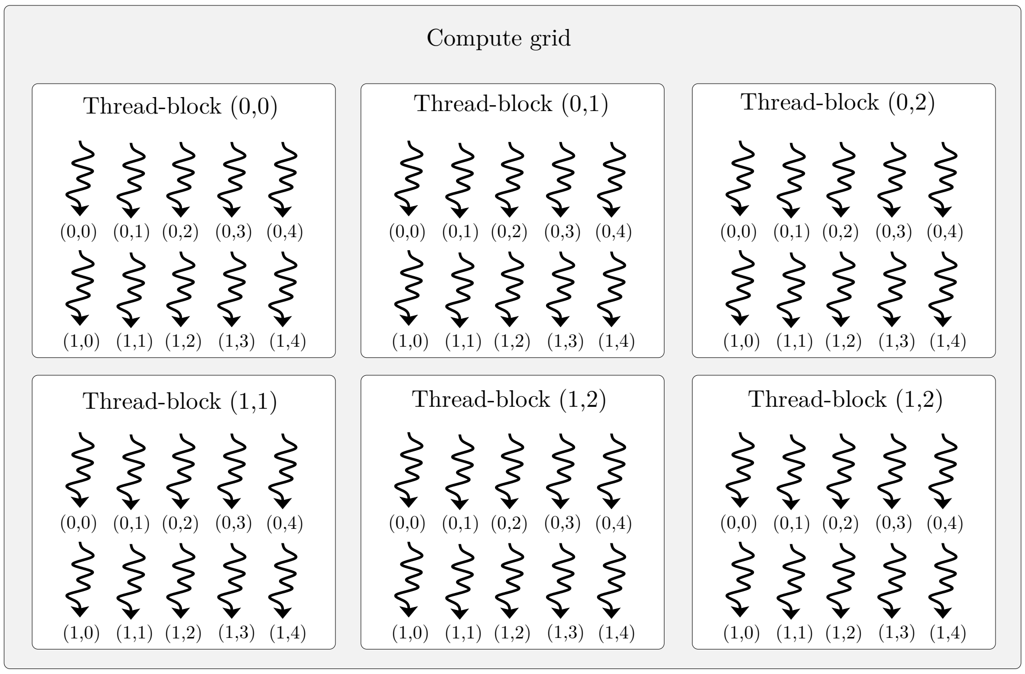


Figure 2.2: Grid of thread-blocks.

In the rest of the chapter, the term warp is used to refer to a batch of threads simultaneously executed.

For illustration purposes, consider the Nvidia Fermi architecture, where each SM has two thread schedulers and dispatch units and the execution units are organized in four sets: two sets of 16 SPs each, one set of 4 SFUs, and one set of 16 load/store units (see Fig 2.1(b)). At every instruction issue time, each scheduler selects a warp and issues one instruction from that warp to consume any two of the four sets of execution units listed above. For example, one instruction can consume both sets of 16 SPs, or one set of 16 SPs and the set of 4 SFUs, or any combination alike.

All the execution units are fully pipelined, which allows the GPU to exploit Instruction-Level Parallelism (ILP) within a single thread. This adds up to the data parallelism that is already exploited by the SIMT execution model.

In the presence of a branch, execution paths that are only taken by some of the threads in a warp are also executed by the others. Typically, all divergent paths are executed by the entire warp serially (unless all the threads happen to follow the exact same path). This generates unnecessary computation and should be avoided if possible. One solution is to sort the data so that divergent paths are taken preferable by different warps. Another is to use function templates and generate separate kernels for each execution path. For details on the latter, see [Harris 2007].

2.2.2 Memory hierarchy

GPU memory is organized in three main areas: registers, shared memory and global memory. The first two can be found inside any individual SM, i.e., *on-chip*. The third is located outside the SM, i.e., *off-chip*. These areas are distinguished in Fig. 2.1. According to their distance to the execution units, these memories are accessed at different speeds: registers and shared memory have a low latency access (few cycles), whereas global memory has a relatively high latency access (hundreds of cycles). In general, registers have a higher bandwidth than shared memory, and shared memory has a higher bandwidth than global memory. For example, in the Fermi architecture, these are 8 TB/s for registers, 1.6 TB/s for shared memory, and 192 GB/s for global memory.

Registers. Registers are allocated to each thread as needed, with a maximum depending on the architecture (e.g., 63 registers per thread in modern Nvidia architectures). The number of registers per thread decreases as the number of threads in flight increases. When a thread needs more registers than the ones available, registers are spilled onto off-chip "local" memory. Such *register spilling* should be avoided if possible, e.g., by adjusting the number of blocks and number of threads per block.

Shared memory. Shared memory can be used by the programmer to explicitly share data among threads from the same block, thus enabling cooperation. It can be seen as a scratchpad, or programmable cache. Shared memory is organized in banks. These can be accessed simultaneously by all the threads within a warp, as long as each thread accesses a different bank. If two or more threads try accessing the same bank, then several memory transactions are serially issued. Such concurrent access, also known as *bank conflict*, must be prevented in order to achieve maximum performance. Sometimes programmers add padding to the data in shared memory in order to avoid these conflicts for a given memory access pattern.

Global memory. Global memory allows host and device processors to exchange data. Typically, input data is copied from host to device prior to the kernel execution, and results are copied from device to host afterwards. Global memory is accessed by segments of a fixed size (e.g., 32, 64, or 128 bytes). When threads in a warp access data in the same segment, all the accesses are *coalesced* into a unique memory transaction. Otherwise, several transactions are issued, degrading performance. A coalesced access can be achieved, for example, by having consecutive threads accessing consecutive cells in memory.

In addition to these main memory areas, modern GPUs may provide several types of cache, e.g., L1 cache, L2 cache, constant memory and texture memory. For details about these caches, see [Glaskowsky 2009].

2.2.3 Programming frameworks

General purpose GPU programming is achieved by using a specific framework, with an Application Program Interface (API) defined for that framework. CUDA and OpenCL are the most commonly used.

These frameworks define a layer of abstraction, so that the same program can be compiled for different GPU architectures without changing the code. In this way, the parallelism in the application automatically scales with the number of computing resources in the architecture. The programmer is just responsible for exposing the parallelism by writing a kernel program. The kernel may involve tasks to be executed *independently* in parallel by blocks of threads, and smaller sub-tasks, to be executed *cooperatively* in parallel by threads within a block. The programmer specifies the number of blocks and number of threads per block. However, the specific way in which these blocks and threads are executed on the GPU is entirely up to the hardware, and the programmer does not have control over it.

The CUDA and OpenCL API extend standard programming languages such as C and C++. This allows for *heterogeneous programming*, where sections of host code are interleaved with sections of device code. Device code is written in kernel functions, which contain the instructions to be executed by a single thread. Typically, this involves retrieving of identifiers for that thread, data transfers between global and shared memory, several arithmetic operations, and synchronization instructions. Host code typically involves setting up of GPU devices, declaration of variables in host and device memory, data transfers between host and device, and kernel calls.

Despite the fact that both CUDA and OpenCL provide a very similar functionality, there are still relevant differences between them. Some of these differences and particular features are outlined below. For an extensive description of the CUDA and OpenCL frameworks, see [Cuda 2012, Stone 2010].

CUDA. Compute Unified Device Architecture (CUDA) is a parallel computing platform and API developed by Nvidia exclusively for their own manufactured GPUs. The API consists of a set of C language extensions and a runtime library.

CUDA programs are compiled with *nvcc*. Compilation is usually performed offline, by targeting a specific device or range of devices. The latter is specified by a *compute capability*, which indicates relevant features common to several GPU devices, usually attached to the generation (e.g., number of SMs, number of execution units, size of the register file, size of the L1 and L2 caches). Offline compilation is divided in three steps. First, device code is separated from host code and compiled into assembly and/or binary code by nvcc. The assembly code is tailored for a specific compute capability, whereas the binary code is already tailored for a specific device. Second, host code is modified by introducing function calls to the CUDA run-time library, needed to load and launch the kernels compiled in the precedent step. Third, the resulting host code is compiled into object code by the host compiler.

When an application loads assembly code to be executed on a specific device, the driver automatically checks if there is a previously compiled binary for that device. If not, it generates the binary at run-time, using nvcc, and caches a copy for future runs. This is called *just-in-time* compilation, and allows applications to immediately take advantage of driver upgrades. It is also a way for an application to run on devices that may not have been existed at the moment the application was originally compiled.

OpenCL. Open Computing Language (OpenCL) is an open standard for heterogeneous programming maintained by the Khronos group. Unlike CUDA, OpenCL is designed to run across heterogeneous *platforms* with devices from several manufacturers. This includes CPUs and GPUs, but also other processors such as Digital Signal Processors (DSPs) and Field-Programmable Gate Arrays (FPGAs). The Khronos group specifies the OpenCL API to be implemented by the chip manufacturers. The API is organized into a platform layer API and a run-time API. Compliant implementations are available from Apple, Intel, AMD and Nvidia, among others.

In OpenCL, device code is usually compiled at run-time. This is intended to ensure portability of applications among several compute platforms. As a consequence, it is up to the programmer to explicitly use the run-time API in host code in order to facilitate the compilation of device code (a process which is performed automatically in CUDA). Typically, this means calling a function to create a program object with source code from a text buffer, and another function to build that program into a binary. Then, kernel objects can be created from the program just built, and queued to launch onto a selected device.

The OpenCL run-time API also provides a function to create program objects directly from binary code, instead of using source code. This is particularly useful for applications that are designed to run several times on the same device and platform, as it can reduce the execution time. However, it is up to the programmer to purposely store a copy of the generated binary into disk, as binaries normally have the lifetime of an individual execution. Of course, this strategy is not optimal for portability [van der Sanden 2011].

2.3 Performance considerations

In order to improve performance of GPU applications, programmers may follow several guidelines. These guidelines revolve around optimizing instruction and memory usage. In order to know which strategies will be more impactful for a particular kernel, the first step is to determine the bottleneck of that kernel. There are three major bottlenecks that a kernel can encounter: instruction throughput, memory throughput and latencies [Cuda 2012].

The bottlenecks can be determined by using the CUDA profiler, which is provided by Nvidia with every modern CUDA distribution. For each kernel, the profiler calculates the ratio of arithmetic instructions to data transfers. Then, by comparing this figure with the ideal ratio of the device, one can determine whether the kernel is compute-bound, memory-bound or latency-bound. Another way is to modify the kernel by alternatively removing all arithmetic instructions and memory instructions, and compare the performance of the modified kernels with the original one.

The remainder of the section discusses three relevant factors for GPU performance: regularity, latency hiding and throughput. The discussion introduces several strategies for enhancing performance of GPU applications.

2.3.1 Regularity

In parallel computing, regularity means similarity of behaviour between threads. Regularity usually determines how well an application fits into the SIMT execution model. Therefore, it becomes one of the main aspects to consider for GPU programming [Collange 2010].

For certain computations, e.g., matrix-matrix multiplication, a parallel algorithm naturally exhibits a regular computation pattern. In some cases, the regular algorithm is also work-efficient, i.e., it performs the minimal amount of operations needed to compute the result. Whereas in other cases, additional operations have to be introduced in order to provide regularity. On the other hand, applications may use *load-balancing* whenever a regular computation pattern is hard to find. Load-balancing means equally distributing parallel tasks among available threads in execution time. This is the case, for example, of certain sparse matrix computations and graph computations where the work-flow is data-driven or even topology-driven [Nasre 2013].

Regularity can take several forms. The most basic is *instruction regularity*, where different threads execute the same instruction simultaneously. On GPUs, this is implied in the SIMT execution model and the concept of warp. Additionally, applications may also implement *control regularity* and *memory regularity*, explained below.

Control regularity means having different threads follow the same execution path. On GPUs, this implies minimizing divergence among threads in a warp caused by branching. As mentioned in Section 2.2.1, when threads in a warp take divergent paths, all these paths are executed serially by the entire warp. In such case, the instruction throughput is divided by the number of divergent paths. In CUDA, whenever a branch is likely to generate many divergent paths with a limited number of instructions each, nvcc is allowed to use *branch predication*. With branch predication, only instructions in the path taken by each thread are actually executed. Instructions from other paths are scheduled but not executed. This allows the SM to save resources, although it does not prevent serialization of paths.

Memory regularity means using a memory access pattern that minimizes the number of memory transactions. On GPUs, this comes down to avoid uncoalesced accesses to global memory, as well as bank conflicts in shared memory accesses.

As seen in Section 2.2.2, when threads in a warp read or write data in different segments of global memory, the accesses are split into several transactions. In this case, the throughput is divided by the ratio of the size of the transaction to the size of the accessed word. For example, if a 4-byte word is accessed through a 32-byte memory transaction, then the throughput is divided by 8. Accordingly, the more scattered are the addresses, the more the throughput is reduced. As also seen in Section 2.2.2, when threads in a warp access the same shared memory bank, again, several memory transactions are issued. In this case, the throughput is divided by the number of bank conflicts.

2.3.2 Latency hiding

Instruction latency is the number of cycles that a warp takes in executing a given instruction. In other words, it is the period between the instruction issue time and the completion time. On SIMT architectures, latency can be hidden by interlacing execution of independent instructions. Latency hiding is one of the main sources of GPU performance, as it allows the different execution units to remain busy most of the time [Cuda 2012].

Instruction latency is typically caused by data dependence. Thus, it depends on the memory access latency. For example, assume that an instruction needs to read a variable which is written by the previous instruction (Read After Write dependence). If this variable resides in a register, then the latency is relatively low (tens of cycles, the time of accessing the register file). Whereas if the variable resides in off-chip global memory, then the latency is much higher (hundreds of cycles).

In order to hide latency, a SM interlaces the instructions from other warps. However, the SM needs to find enough independent instructions to execute during the latency period. To determine the number of instructions needed to saturate the SM, one has to multiply latency by throughput. For example, if an instruction has a latency of 20 cycles, and the SM can issue two instructions per cycle, then at least 40 warps are needed to completely hide that 20 cycles of latency.

The CUDA programming guide recommends maximizing the *occupancy*, i.e., the number of warps in the flight, in order to hide latency and maximize utilization. However, latency can also be hidden by means of instruction-level parallelism (ILP). Indeed, as mentioned in Section 2.2.1, GPUs are able to exploit ILP in addition to data-parallelism, thanks to the instruction pipeline available in every execution unit. Following this, Volkov has demonstrated that kernels can use ILP to achieve better performance at lower occupancy [Volkov 2008].

In addition, using ILP means doing more work per thread, and thus fewer threads are required. In this way, the block size can be reduced and the number of registers per thread can be increased. Reducing the block size allows more blocks to fit into each SM. Therefore, arithmetic instructions can be overlapped with global memory accesses more efficiently. In turn, increasing the number of registers per thread allows data to be transferred at a higher bandwidth. Therefore, memory-bound kernels can perform faster. These ideas are developed in more detail in [Volkov 2010].

2.3.3 Throughput

Throughput is the number of operations performed per cycle. Latency hiding, discussed in the previous section, is a way to increase throughput at SM level. However, throughput can also be increased at instruction level.

Arithmetic throughput can be increased by using a lower precision either for integer or floating-point computations. For example, consider a numerical algorithm that computes a series of approximations to a certain variable and stops when the error is below a certain threshold. Assume that the error is decreasing geometrically, i.e., every new approximation is at least one order of magnitude more accurate than the precedent. For the first few approximations, when the error is still very large, the computations can be performed using a low precision (e.g., single). Then, only when the error approaches the threshold, the algorithm can switch to a higher precision in order to deliver the final result (e.g., double). This kind of methods are known as *mixed precision* methods [Clark 2010].

CUDA provides a set of intrinsic functions for fast mathematical computation, including division, trigonometrical functions, logarithm and exponentiation, among others. These map to fewer native instructions than their standard counterparts, resulting in faster, although less accurate computation. They can be called explicitly in device code, at the points where the programmer is willing to trade accuracy for performance.

In addition, memory throughput can be increased by avoiding memory accesses with high latency. In particular, data transfers between host and device can be reduced by running more tasks on the device, even serial ones. For example, consider a loop in host code that launches a kernel and reads back the results in order to evaluate an end condition. This requires transferring data from device to host at every iteration, whereas if the condition is evaluated on device code, then all those memory transfers are avoided.

Finally, global memory accesses from within SMs can be reduced by using shared memory. For example, assume that the previous loop can be split into several loops working on non-overlapping subsets of the data. These loops can be assigned to independent thread-blocks and performed asynchronously on shared memory. In this way, the blocks transfer their results to global memory only when the end condition is reached, instead of doing it at every iteration.

2.4 Conclusions

This chapter presents a review of GPUs and general purpose GPU programming. The hardware architecture is first described, detailing the SIMT execution model, the GPU memory hierarchy and the main frameworks for GPU computing. Next, three relevant factors for GPU performance are analyzed: regularity, latency hiding and throughput.

Following chapters resume some of the issues presented above, applied to specific problems issued from Power Flow (PF) analysis. Chapter 4 addresses the problem of regularity versus load-balancing for the implementation of the so-called *treefix* operations. Which approach is better for GPU programming? On one hand, regularity exploits better the SIMT execution model, but may introduce extra operations and data transfer. On the other hand, load-balancing keeps the number of operations to a minimum, but it may generate a computational pattern unsuited to SIMT.

Chapter 5 provides a comparison of different interval representation formats for fuzzy algebra. Is it possible to improve performance or accuracy of fuzzy interval applications by using a specific representation format? Certain peculiarities of the membership function can be exploited by the midpoint-radius format, however, most available fuzzy algebra implementations overlook that fact, relying exclusively on lower-upper.

GPU computing is a relatively recent technology. Therefore, most existing GPU-based approaches for PF analysis and simulation rely on parallelizing already available techniques. What about using our knowledge of the GPU to design new methods, specifically suited for the implementation on such massively parallel architectures? The asynchronous PF method proposed in Chapter 6 is one of the first attempts to achieve that goal.

References

[Che 2008] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer and K. Skadron. *A performance study of general-purpose applications on graphics processors using CUDA*. Journal of parallel and distributed computing, vol. 68, no. 10, pp. 1370–1380, 2008. (Cited on pages 9 and 19.)

[Clark 2010] M. A. Clark, R. Babich, K. Barros, R. C. Brower and C. Rebbi. *Solving Lattice QCD systems of equations using mixed precision solvers on GPUs*. Computer Physics Communications, vol. 181, no. 9, pp. 1517–1528, 2010. (Cited on page 28.)

[Collange 2010] S. Collange. *Design challenges of GPGPU architectures: specialized arithmetic units and exploitation of regularity*. PhD thesis, Université de Perpignan, 2010. (Cited on pages 7 and 26.)

[Cuda 2012] C. Cuda. *Programming guide*. NVIDIA Corporation, July, 2012. (Cited on pages 9, 24, 25, 27, 88 and 92.)

[Glaskowsky 2009] P. N. Glaskowsky. *NVIDIA's Fermi: the first complete GPU computing architecture*. White paper, 2009. (Cited on pages 20, 23 and 61.)

[Harris 2007] M. Harris*et al.* *Optimizing parallel reduction in CUDA*. NVIDIA Developer Technology, vol. 2, no. 4, 2007. (Cited on page 22.)

[Nasre 2013] R. Nasre, M. Burtscher and K. Pingali. *Data-driven versus topology-driven irregular computations on gpus*. IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS), pp. 463–474, 2013. (Cited on page 26.)

[Owens 2008] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips. *GPU computing*. Proceedings of the IEEE, vol. 96, no. 5, pp. 879–899, 2008. (Cited on pages 8 and 19.)

[Ryoo 2008] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton and W.-m. W. Hwu. *Program optimization space pruning for a multithreaded gpu*. Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization, pp. 195–204, 2008. (Cited on page 19.)

[Stone 2010] J. E. Stone, D. Gohara and G. Shi. *OpenCL: A parallel programming standard for heterogeneous computing systems*. Computing in science & engineering, vol. 12, no. 1-3, pp. 66–73, 2010. (Cited on pages 9 and 24.)

[van der Sanden 2011] J. van der Sanden. Evaluating the Performance and Portability of OpenCL. Master's thesis, Faculty of Electrical Engineering Eindhoven University of Technology, 2011. (Cited on page 25.)

[Volkov 2008] V. Volkov and J. W. Demmel. *Benchmarking GPUs to tune dense linear algebra.* IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11, 2008. (Cited on pages 9 and 27.)

[Volkov 2010] V. Volkov. *Better performance at lower occupancy.* Proceedings of the GPU Technology Conference, vol. 10, 2010. (Cited on page 28.)

Chapter 3

Circuit and Power Flow analysis

Contents

3.1 Introduction

This chapter outlines different models and methods for Power Flow (PF) analysis. It does not aim to be an exhaustive survey but rather an introduction to the main aspects of PF analysis that will be addressed in Chapters 4, 5 and 6.

PF analysis can be seen as a special case of circuit analysis, where the component behaviour is defined in terms of power injections. Using a power injection model leads to a set of nonlinear equations. As a consequence, the solution is commonly approached by using numerical methods. The Newton-Raphson (NR) method is considered the standard [Stott 1974].

This chapter also considers uncertainty in PF analysis, which is caused by the complex and heterogeneous nature of the power systems. Uncertainty is typically modelled using probabilistic and interval analysis, and a number of solution methods have been proposed. The problem acquires special relevance in the context of the smart-grid functionality, with the changes experimented by power systems worldwide in recent years [Wan 1993].

The remainder of the chapter is organized as follows. Section 3.2 presents the fundamentals of circuit analysis, including Alternating Current circuits in steady-state. Section 3.3 focuses on PF analysis. Conventional solution methods are classified based on network topology. Relevant approaches to deal with uncertainty are also discussed in Section 3.3. Finally, Section 3.4 draws conclusions and outlines the developments presented in Chapters 4, 5 and 6.

3.2 Classic circuit analysis

PF analysis finds its roots in circuit analysis. This consists in determining node voltages and branch currents in an electrical circuit given the description of the circuit components (sources, resistors, etc.) [Nilsson 2009].

The circuit is typically assumed to be a *lumped parameter system*, where electrical effects happen simultaneously throughout all components. This assumption is valid if the signal wavelengths are much larger than the circuit itself, and in particular if voltage and current are time-invariant. In this situation Kirchhoff's laws can be applied [Nilsson 2009]. *Kirchhoff's current law* establishes that the (signed) sum of currents injected to any node in the circuit is equal to zero. *Kirchhoff's voltage law* establishes that the voltage drop across any closed loop in the circuit is equal to zero. It is important to note that these laws are only valid when

Kirchhoff's current law. Let $i_1, i_2, \dots, i_n$ be the current intensities on n branches merging into a node in an electrical circuit. Then,

$$\sum_{k=1}^{n} i_k = 0. \tag{3.1}$$

Kirchhoff's voltage law. Let $v_1, v_2, \dots, v_n$ be the voltage drops across n branches that form a closed loop within an electrical circuit. Then,

$$\sum_{k=1}^{n} v_k = 0. \tag{3.2}$$

Kirchhoff's current law leads to the so-called *node method*. One node in the circuit is selected as *ground*, or reference node. All other voltages are measured with respect to ground. Then, Kirchhoff's current law is applied on every non-ground node in the circuit, and the resulting equation system is solved. Of course, the above needs to be completed with the particular equations that describe circuit components and relate voltage and current within the components themselves.

3.2.1 Alternating current circuits in steady-state

Let's consider the following two assumptions that characterize Alternating Current (AC) circuits in steady-state operation:

i) All sources are sinusoidal waveforms of the same frequency (AC assumption).

ii) The sources have been active long enough for all transients to fade away (steady-state assumption).

If these assumptions hold and all components are linear, then all voltages and currents in the circuit are sinusoidal waveforms of the same frequency as the sources [Nilsson 2009]. Typically, they are represented as *phasors*. For example, consider the sinusoidal waveform, $v(t) = v\cos(\omega t + \theta)$. This is written as the real part of a complex exponential, as follows.

$$v(t) = \mathrm{Re}\left(ve^{j(\omega t+\theta)}\right) = \mathrm{Re}\left(ve^{j\omega t}e^{j\theta}\right). \tag{3.3}$$

Since ω is fixed, $v(t)$ is simply noted as the phasor $\mathbb{v} = ve^{j\theta} = v(\cos\theta + j\sin\theta)$, or, in angle notation, $\mathbb{v} = v\angle\theta$.

Additionally, the concept of *impedance* is introduced to model the behaviour of electrical components under the AC regime.

Definition 3.2.1 (Impedance)**.** Let $\mathbb{v}$ be the phasor voltage across an electrical component, and $\mathbb{i}$, the phasor current through it, in an AC steady state operation regime. Then, the complex ratio,

$$z = \frac{\mathbb{v}}{\mathbb{i}}, \tag{3.4}$$

is the impedance of the component.

Note that even if z is the ratio of two phasors, it is not itself a phasor (it is not associated to any sinusoidal form). The impedance of common electrical components is obtained from the component model. In general, the resulting expression is a combination of the model parameters, the system frequency, ω, and the imaginary unit, j. For example, for the resistor, inductor and capacitor, the following linear relations are available:

$$v_R(t) = R\, i_R(t), \tag{3.5a}$$

$$i_C(t) = C\,\frac{dv_C}{dt}, \tag{3.5b}$$

$$v_L(t) = L\,\frac{di_L}{dt}, \tag{3.5c}$$

where $v_R(t)$, $v_C(t)$ and $v_L(t)$, are the voltages across the components; R, C and L, are the resistance, capacitance and inductance, respectively; $i_R(t)$, $i_C(t)$ and $i_L(t)$ are the currents through. Applying phasor differentiation to these equations, the following complex equations are obtained:

$$\mathbb{v}_R = R\,\mathbb{i}_R, \tag{3.6a}$$

$$\mathbb{v}_C = -\frac{j}{\omega C}\,\mathbb{i}_C, \tag{3.6b}$$

$$\mathbb{v}_L = j\omega L\,\mathbb{i}_L, \tag{3.6c}$$

where $\mathbb{v}_R$, $\mathbb{v}_C$ and $\mathbb{v}_L$, are the phasor voltages at component terminals; $\mathbb{i}_R$, $\mathbb{i}_C$ and $\mathbb{i}_L$ are the phasor currents through. By looking at the above equations, the expression for the impedances can be deduced. These are summarized in Table 3.1.

The remainder of the section illustrates the node method on an example AC circuit.

Table 3.1: Impedance of common linear components.

Component type	Impedance
Resistance R	R
Capacitance C	$-\frac{j}{\omega C}$
Inductance L	$j\omega L$

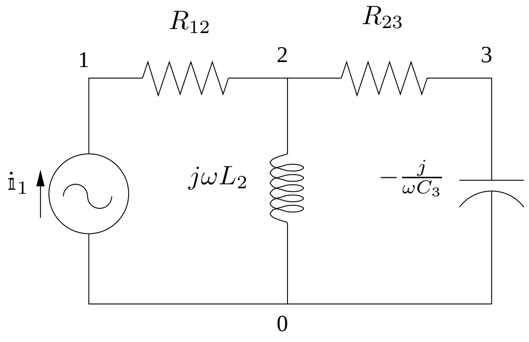


Figure 3.1: RLC circuit in stationary AC conditions.

Example 3.2.1: Node method for AC circuit analysis. Figure 3.1 shows a typical AC circuit. The current source, $\mathbb{i}_1$, the resistances R_{12} and R_{23}, inductance L_2, and capacitance C_3, are all given. The problem is to determine $\mathbb{v}_1$, $\mathbb{v}_2$ and $\mathbb{v}_3$, the phasor voltages at nodes 1, 2 and 3. Node 0 is selected as voltage reference, i.e., $\mathbb{v}_0 = 0$.

Applying Kirchhoff's current law on nodes 1, 2 and 3, the following system of complex linear equations is obtained.

$$\mathbb{i}_1 + \frac{1}{R_{12}}(\mathbb{v}_1 - \mathbb{v}_2) = 0, \tag{3.7a}$$

$$\frac{1}{R_{12}}(\mathbb{v}_2 - \mathbb{v}_1) - \frac{j}{\omega L_2}\,\mathbb{v}_2 + \frac{1}{R_{23}}(\mathbb{v}_2 - \mathbb{v}_3) = 0, \tag{3.7b}$$

$$\frac{1}{R_{23}}(\mathbb{v}_3 - \mathbb{v}_2) + j\omega C_3\,\mathbb{v}_3 = 0. \tag{3.7c}$$

In matrix form, one obtains:

$$\begin{pmatrix} \frac{1}{R_{12}} & -\frac{1}{R_{12}} & 0 \\ -\frac{1}{R_{12}} & \frac{1}{R_{12}} - \frac{j}{\omega L_2} + \frac{1}{R_{23}} & -\frac{1}{R_{23}} \\ 0 & -\frac{1}{R_{23}} & \frac{1}{R_{23}} + j\omega C_3 \end{pmatrix} \begin{pmatrix} \mathbb{v}_1 \\ \mathbb{v}_2 \\ \mathbb{v}_3 \end{pmatrix} = \begin{pmatrix} -\mathbb{i}_1 \\ 0 \\ 0 \end{pmatrix}. \tag{3.8}$$

The above system can be solved by a number of techniques (e.g., Gaussian elimination, LU factorization). The solution represents the stationary condition or steady-state of the circuit.

3.3 PF analysis

PF analysis is the steady state analysis of an AC circuit that represents the power network. The focus is on power injections at network buses, which determine the power flows across the network. The concept of *complex power* is introduced to study these power flows.

Definition 3.3.1 (Complex power)**.** Let i be the current magnitude through an electrical component in an AC circuit, and z, its impedance. Then,

$$\mathbb{s} = z\, i^2, \tag{3.9}$$

is the complex power flowing through the component.

The real part of the complex power is called *real power*, and noted p. The imaginary part is called *reactive power*, and noted q. Then, $\mathbb{s} = p + jq$. For a physical interpretation of real and reactive power, see [Machowski 1997]. Note that the complex power can also be obtained by,

$$\mathbb{s} = \mathbb{v}\, \mathbb{i}^*, \tag{3.10}$$

where $\mathbb{v}$ and $\mathbb{i}$ are the phasor voltage and current, and $(\cdot)^*$ denotes the complex conjugate.

In this thesis, the power network is assumed to be a balanced symmetrical three-phase system. Accordingly, it suffices to analyze only one of the phases using a single-phase equivalent circuit. The typical PF problem defines four types of devices: *generators*, *loads*, *transmission lines* and *transformers*. Nodes in the circuit are referred to as *buses*. In the single-phase equivalent circuit, generators and loads are always connected with one terminal to ground. Transmission lines and transformers are connected between two non-ground buses.

Generators are specified in terms of a real power injection and a voltage magnitude, whereas loads are specified in terms of a real and a reactive power consumption. Transmission lines and transformers are specified in terms of a π-circuit with constant impedance. The per-unit system (pu) is used, which allows expressing voltage magnitudes and power flows as fractions of a determined base.

The objective is to compute bus voltage magnitudes and phase angles. To this purpose, the buses in the network are classified as follows. If a generator is connected to the bus, it is called a *generator bus*. If only loads are connected, it is called a *load bus*. One of the generator buses is chosen as reference for the voltage angle, and is called the *slack bus*.

At generator buses, the real power injection and the voltage magnitude are known. This leaves the voltage phase angle and the reactive power injection to be determined. At load buses, real and reactive power injections are known, which leaves the voltage magnitude and phase angle to be determined. At the slack bus, voltage magnitude and phase are known, thus, real and reactive power injections are left to determine. The above is summarized in Table 3.2.

Table 3.2: Known and unknown variables for each type of bus in PF analysis.

Bus type	Knowns	Unknowns	Code
Generator	p, v	θ, q	PV
Load	p, q	v, θ	PQ
Slack	v, θ	p, q	-

3.3.1 Conventional solution methods

Depending on the topology of the network, different solution methods can be used. One of the main aspects that determine the suitable method is the presence (or absence) of meshes in the grid. For an extensive survey of PF analysis methods see [Milano 2010, Kersting 2012].

Meshed networks

In the general case of a network with several generators and meshes, the solution is typically approached by solving the system of *power balance equations*. These state that the power injected to each bus by generators and loads is equal to the net PF entering the bus from transmission lines. The above is expressed in the following vector equation:

$$\mathbb{s} = \mathrm{Diag}(\mathbb{v})\, \mathbb{i}^*, \tag{3.11}$$

where $\mathbb{s}$ is the vector of complex powers injected by generators and loads, $\mathbb{v}$ is the vector of phasor bus voltages, and $\mathbb{i}$ is the vector of phasor currents coming through transmission lines. $\mathrm{Diag}(\mathbb{v})$ is the diagonal matrix that has $\mathbb{v}$ as diagonal.

The vector of current injections, $\mathbb{i}$, is obtained from circuit analysis, using the following expression:

$$\mathbb{i} = \boldsymbol{Y}\mathbb{v}, \tag{3.12}$$

where $\boldsymbol{Y}$ is the *admittance matrix*, defined below.

Definition 3.3.2 (Admittance matrix)**.** Let z_{hk} be the impedance of the line connecting buses h and k in a power network. Then, the admittance matrix, $\boldsymbol{Y}$, is a symmetric matrix, defined by,

$$y_{hk} = \begin{cases} \sum\limits_m \frac{1}{z_{hm}} & \text{if } h = k, \\ -\frac{1}{z_{hk}} & \text{otherwise.} \end{cases} \tag{3.13}$$

Combining equations (3.11) and (3.12), the power balance equations are noted as follows.

$$\mathbb{s} = \mathrm{Diag}(\mathbb{v})(\boldsymbol{Y}\mathbb{v})^*, \tag{3.14}$$

And, in scalar form,

$$\mathbb{s}_h = \mathbb{v}_h \sum_{k\in\mathcal{B}} (y_{hk}\mathbb{v}_k)^*, \quad h \in \mathcal{B}, \tag{3.15}$$

where $\mathcal{B}$ is the set of buses. The above equation is split into real and imaginary parts, by writing, $\mathbb{s}_h = p_h + jq_h$, $\mathbb{v}_h = v_h e^{j\theta_h}$, and $y_{hk} = g_{hk} + jb_{hk}$. This yields the following two equations.

$$p_h = v_h \sum_{k\in\mathcal{B}} v_k(g_{hk}\cos\theta_{hk} + b_{hk}\sin\theta_{hk}), \quad h \in \mathcal{B}, \tag{3.16a}$$

$$q_h = v_h \sum_{k\in\mathcal{B}} v_k(g_{hk}\sin\theta_{hk} - b_{hk}\cos\theta_{hk}), \quad h \in \mathcal{B}, \tag{3.16b}$$

where $\theta_{hk} = \theta_h - \theta_k$.

Equation (3.16a) is written for both PV and PQ buses, whereas equation (3.16b) is only written for PQ buses. This yields a system of $2n + m - 1$ equations on $2n + m - 1$ unknowns, where n is the number of PQ buses, and m is the number of PV buses (including the slack).

Next, two approaches to the solution of the above system are presented. The first is the NR method. The second is relaxation, which includes the Jacobi and Gauss-Seidel methods.

NR method. The NR method is an iterative direct method for finding successively better approximations of the zeros of a differentiable function, $\boldsymbol{f}(\cdot) : \mathbb{R}^n \to \mathbb{R}^n$. It computes a series, $\boldsymbol{x}^{(k)}$, $k = 1, 2, \ldots$, such that,

$$\lim_{k\to\infty} \boldsymbol{f}(\boldsymbol{x}^{(k)}) = \boldsymbol{0}, \tag{3.17}$$

given a "good" initial guess, $\boldsymbol{x}^{(0)}$.

The method relies on approximating the function by its tangent at the current point. The next point is computed as the zero of this tangent. The tangent to $\boldsymbol{f}(\cdot)$ at $\boldsymbol{x}^{(k)}$ is given by,

$$\boldsymbol{y}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{x}^{(k)}) + \boldsymbol{J_f}\Big|_{\boldsymbol{x}=\boldsymbol{x}^{(k)}}(\boldsymbol{x} - \boldsymbol{x}^{(k)}). \tag{3.18}$$

where $\boldsymbol{J_f}$ is the *Jacobian matrix* of $\boldsymbol{f}(\cdot)$ (see appendix A for a definition of Jacobian matrix). Imposing, $\boldsymbol{y}(\boldsymbol{x}^{(k+1)}) = \boldsymbol{0}$ in the equation above, yields the NR iteration,

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \left(\boldsymbol{J_f}\Big|_{\boldsymbol{x}=\boldsymbol{x}^{(k)}}\right)^{-1} \boldsymbol{f}(\boldsymbol{x}^{(k)}). \tag{3.19}$$

The NR method is applied to PF analysis as follows. The buses in the network are numbered from 1 to $m+n$, such that: bus 1 is the slack bus, buses 2 to m are PV buses, and buses $m+1$ to $m+n$ are PQ buses. Then, $\boldsymbol{x}$ and $\boldsymbol{f}(\cdot)$ are chosen as,

$$\boldsymbol{x} = \begin{pmatrix} \boldsymbol{\theta} \\ \boldsymbol{v} \end{pmatrix} = \begin{pmatrix} \theta_2 \\ \vdots \\ \theta_{m+n} \\ v_{m+1} \\ \vdots \\ v_{m+n} \end{pmatrix}, \quad \boldsymbol{f}(\boldsymbol{x}) = \begin{pmatrix} \Delta \boldsymbol{p}(\boldsymbol{x}) \\ \Delta \boldsymbol{q}(\boldsymbol{x}) \end{pmatrix} = \begin{pmatrix} \Delta p_2(\boldsymbol{x}) \\ \vdots \\ \Delta p_{m+n}(\boldsymbol{x}) \\ \Delta q_{m+1}(\boldsymbol{x}) \\ \vdots \\ \Delta q_{m+n}(\boldsymbol{x}) \end{pmatrix}, \tag{3.20}$$

where,

$$\Delta p_h(\boldsymbol{x}) = -p_h + v_h \sum_{k \in \mathcal{B}} v_k (g_{hk} \cos\theta_{hk} + b_{hk} \sin\theta_{hk}), \tag{3.21a}$$

$$\Delta q_h(\boldsymbol{x}) = -q_h + v_h \sum_{k \in \mathcal{B}} v_k (g_{hk} \sin\theta_{hk} - b_{hk} \cos\theta_{hk}). \tag{3.21b}$$

Functions Δp_h and Δq_h are called *power mismatches.*

The Jacobian matrix, $\boldsymbol{J_f}$, is computed as follows.

$$\frac{\partial \Delta p_h}{\partial \theta_h} = -v_h \sum_{k \neq h} v_k \left(g_{hk} \sin\theta_{hk} - b_{hk} \cos\theta_{hk} \right), \tag{3.22a}$$

$$\frac{\partial \Delta p_h}{\partial \theta_k} = v_h v_k \left(g_{hk} \sin\theta_{hk} - b_{hk} \cos\theta_{hk} \right), \quad h \neq k, \tag{3.22b}$$

$$\frac{\partial \Delta p_h}{\partial v_h} = 2 v_h g_{hh} + \sum_{k \neq h} v_k \left(g_{hk} \cos\theta_{hk} + b_{hk} \sin\theta_{hk} \right), \tag{3.22c}$$

$$\frac{\partial \Delta p_h}{\partial v_k} = v_h \left(g_{hk} \cos\theta_{hk} + b_{hk} \sin\theta_{hk} \right), \quad h \neq k, \tag{3.22d}$$

$$\frac{\partial \Delta q_h}{\partial \theta_h} = -v_h \sum_{k \neq h} v_k \left(g_{hk} \cos\theta_{hk} + b_{hk} \sin\theta_{hk} \right), \tag{3.22e}$$

$$\frac{\partial \Delta q_h}{\partial \theta_k} = v_h v_k \left(g_{hk} \cos\theta_{hk} + b_{hk} \sin\theta_{hk} \right), \quad h \neq k, \tag{3.22f}$$

$$\frac{\partial \Delta q_h}{\partial v_h} = -2 v_h b_{hh} - \sum_{k \neq h} v_k \left(g_{hk} \sin\theta_{hk} - b_{hk} \cos\theta_{hk} \right), \tag{3.22g}$$

$$\frac{\partial \Delta q_h}{\partial v_k} = -v_h \left(g_{hk} \sin\theta_{hk} - b_{hk} \cos\theta_{hk} \right), \quad h \neq k. \tag{3.22h}$$

The NR method for PF analysis is depicted in Algorithm 3.1.

Jacobi and Gauss-Seidel methods. The Jacobi and Gauss-Seidel methods are iterative fixed-point methods for finding successively better approximations to the

Algorithm 3.1 NR algorithm for PF analysis.

Input: Real power injections at all non-slack buses, $p_2, \dots, p_{m+n}$. Reactive power injections at PQ buses, $q_{m+1}, \dots, q_{m+n}$. Voltage magnitudes at PV buses, $v_2, \dots, v_m$. Complex voltage at the slack, $|v_1|\angle\theta_1$. Admittance matrix, $\boldsymbol{Y}$. Maximum tolerance for the error, ε.

Output: Voltage phase angles at all non-slack buses, $\theta_2, \dots, \theta_{m+n}$. Voltage magnitudes at PQ buses, $v_{m+1}, \dots, v_{m+n}$.

```
1: $\boldsymbol{x} = \begin{pmatrix} \boldsymbol{\theta} \\ \boldsymbol{v} \end{pmatrix} = \begin{pmatrix} \mathbf{0} \\ \mathbf{1} \end{pmatrix}$        # Initial guess
2: repeat
3:     $\boldsymbol{f}(\boldsymbol{x}) = \begin{pmatrix} \Delta\boldsymbol{p}(\boldsymbol{x}) \\ \Delta\boldsymbol{q}(\boldsymbol{x}) \end{pmatrix}$        # Power mismatches
4:     $\boldsymbol{J_f} = \begin{pmatrix} \frac{\partial\Delta\boldsymbol{p}}{\partial\boldsymbol{\theta}} & \frac{\partial\Delta\boldsymbol{p}}{\partial|\boldsymbol{v}|} \\ \frac{\partial\Delta\boldsymbol{q}}{\partial\boldsymbol{\theta}} & \frac{\partial\Delta\boldsymbol{q}}{\partial\boldsymbol{v}} \end{pmatrix}$        # Jacobian matrix
5:     $\Delta\boldsymbol{x} = -\boldsymbol{J_f}^{-1}\boldsymbol{f}(\boldsymbol{x})$        # Increment
6:     $\boldsymbol{x}' = \boldsymbol{x} + \Delta\boldsymbol{x}$
7:     $\boldsymbol{x} = \boldsymbol{x}'$
8: until $\|\boldsymbol{f}(\boldsymbol{x})\| < \varepsilon$
```

solution of a system of linear equations,

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{3.23}$$

where $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{b} \in \mathbb{R}^n$. They compute series, $\boldsymbol{x}^{(k)}$, $k = 1, 2, \dots$, such that,

$$\lim_{k\to\infty} \boldsymbol{A}\boldsymbol{x}^{(k)} = \boldsymbol{b}, \tag{3.24}$$

given a "good" initial guess, $\boldsymbol{x}^{(0)}$.

The Jacobi method begins by decomposing matrix $\boldsymbol{A}$ into a diagonal matrix, $\boldsymbol{D}$, and a remainder, $\boldsymbol{R}$, as follows.

$$\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{R}, \tag{3.25}$$

where $\boldsymbol{D}$ is given by,

$$d_{hk} = \begin{cases} a_{hk} & \text{if } h = k, \\ 0 & \text{otherwise,} \end{cases} \tag{3.26}$$

and $\boldsymbol{R}$ is given by,

$$r_{hk} = \begin{cases} 0 & \text{if } h = k, \\ a_{hk} & \text{otherwise.} \end{cases} \tag{3.27}$$

This allows re-writing equation (3.23) as follows.

$$\boldsymbol{x} = \boldsymbol{D}^{-1}(\boldsymbol{b} - \boldsymbol{R}\boldsymbol{x}), \tag{3.28}$$

which leads to the Jacobi iteration,

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{D}^{-1}(\boldsymbol{b} - \boldsymbol{R}\boldsymbol{x}^{(k)}), \tag{3.29}$$

or, in scalar form,

$$x_h^{(k+1)} = \frac{1}{a_{hh}} \left(b_h - \sum_{m \neq h} a_{hm} x_m^{(k)} \right). \tag{3.30}$$

Coppersmith, Don; Winograd, Shmuel (1990), "Matrix multiplication via arithmetic progressions

Similarly, the Gauss-Seidel method decomposes matrix $\boldsymbol{A}$ into a lower triangular matrix, $\boldsymbol{L}$, and a strictly upper triangular matrix, $\boldsymbol{U}$, as follows.

$$\boldsymbol{A} = \boldsymbol{L} + \boldsymbol{U}, \tag{3.31}$$

where $\boldsymbol{L}$ is given by,

$$l_{hk} = \begin{cases} a_{hk} & \text{if } h \leq k, \\ 0 & \text{otherwise,} \end{cases} \tag{3.32}$$

and $\boldsymbol{U}$ is given by,

$$u_{hk} = \begin{cases} a_{hk} & \text{if } h > k, \\ 0 & \text{otherwise.} \end{cases} \tag{3.33}$$

This allows re-writing equation (3.23) as follows.

$$\boldsymbol{x} = \boldsymbol{L}^{-1}(\boldsymbol{b} - \boldsymbol{U}\boldsymbol{x}), \tag{3.34}$$

which leads to the Gauss-Seidel iteration,

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{L}^{-1}(\boldsymbol{b} - \boldsymbol{U}\boldsymbol{x}^{(k)}), \tag{3.35}$$

or, in scalar form,

$$x_h^{(k+1)} = \frac{1}{a_{hh}} \left(b_h - \sum_{m<h} a_{hm} x_m^{(k+1)} - \sum_{m>h} a_{hm} x_m^{(k)} \right). \tag{3.36}$$

The Gauss-Seidel method utilizes the most recent computed values at every update. This includes elements with lower indices from the current iteration. On the contrary, the Jacobi method only utilizes values from the past iteration.

The Jacobi and Gauss-Seidel methods are applied to PF analysis as follows. Equation (3.14) is re-written as,

$$\boldsymbol{Y}\mathbb{v} = \left(\mathrm{Diag}(\mathbb{v})^{-1}\mathbb{s}\right)^{*}. \tag{3.37}$$

Then, $\boldsymbol{A}$ and $\boldsymbol{b}$ are chosen as,

$$\boldsymbol{A} = \boldsymbol{Y}, \quad \boldsymbol{b} = \left(\mathrm{Diag}(\mathbb{v})^{-1}\mathbb{s}\right)^{*}. \tag{3.38}$$

The Jacobi iteration for PF analysis is the following.

$$\mathbb{v}_h^{(k+1)} = \frac{1}{y_{hh}}\left(\left(\frac{\mathbb{s}_h}{\mathbb{v}_h^{(k)}}\right)^{*} - \sum_{m\neq h} y_{hm}\mathbb{v}_m^{(k)}\right). \tag{3.39}$$

Whereas the Gauss-Seidel iteration is the following.

$$\mathbb{v}_{\mathbb{h}}{}^{(k+1)} = \frac{1}{y_{hh}}\left(\left(\frac{\mathbb{s}_h}{\mathbb{v}_h^{(k)}}\right)^{*} - \sum_{m>h} y_{hm}\mathbb{v}_m^{(k)} - \sum_{m<h} y_{hm}\mathbb{v}_m^{(k+1)}\right). \tag{3.40}$$

Radial networks

At the distribution level (e.g., a city) power networks are often characterized by the following two conditions.

i) There are very few *feeders*, modelled as PV buses, one of them chosen as a slack.

ii) The topology of the grid is radial.

This type of network, typically called *radial*, can be analyzed with the Backward-Forward Sweep (BFS) algorithm [Mendive 1975]. In the case of weakly-meshed networks, the BFS algorithm can be extended into a compensation-based PF method, described in [Shirmohammadi 1988].

The BFS algorithm consists of two successive steps, the *backward sweep* and the *forward sweep*, which are repeated iteratively until convergence. The algorithm starts with an initial guess for the voltage magnitude and phase at every PQ bus. These values are updated at each iteration.

The purpose of the backward sweep is to update line currents. A preliminary step computes the current injected by loads to every PQ bus, using equation (3.10). Then, line currents are obtained by applying Kirchhoff's current law on every PQ bus, starting from the line series endpoints and progressing towards the generator bus.

The purpose of the forward sweep is to update PQ bus voltages. A preliminary step computes the voltage drop across every line, using equation (3.4). Then, new

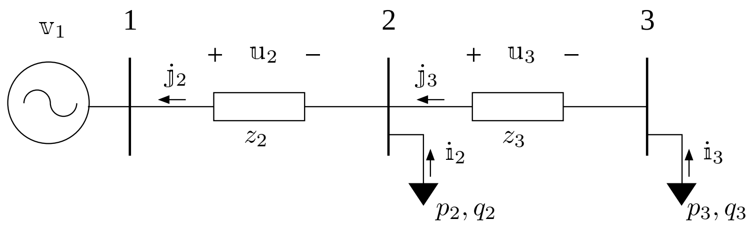


Figure 3.2: 3-bus radial network.

PQ bus voltages are obtained by applying Kirchhoff's voltage law on every line, starting from the ones connected to the root and moving progressively towards the endpoints.

The above is summarized in Algorithm 3.2. Network buses are numbered from 1 to n. Bus 1 is the feeder, or generator bus. The numbering respects the rule that buses further from the generator have higher numbers. The function parent(m) returns to each bus m the number of the parent (i.e., the bus that precedes it in the line series). Similarly, the function children(m) returns the list of children numbers (i.e., the buses that succeed it in the line series). The line connecting buses h and m, where m is further from the generator, is numbered m.

Algorithm 3.2 BFS algorithm.

Input: Vector of complex power injections at PQ buses, $\mathbb{s}$. Vector of branch impedances, $\boldsymbol{z}$. Maximum tolerance for the error, ε.

Output: Vector of complex voltages at PQ buses, $\boldsymbol{v}$.

```
1: $\mathbb{v} = \mathbf{1}\angle\mathbf{0}$                                   # Initial guess
2: repeat
3:   $\mathbb{i} = \left(\mathrm{Diag}(\mathbb{v})^{-1}\mathbb{s}\right)^*$                    # Load currents
4:   for $m$ in $n, \dots, 2$ do
5:     $h = \mathrm{parent}(m)$
6:     $\mathcal{L} = \mathrm{children}(m)$
7:     $\mathbb{j}_m = \mathbb{i}_m + \sum_{k\in\mathcal{L}} \mathbb{j}_k$                   # Backward sweep
8:   $\mathbb{u} = \mathrm{Diag}(\boldsymbol{z})\,\mathbb{j}$                          # Line drops
9:   for $m$ in $2, \dots, n$ do
10:    $h = \mathrm{parent}(m)$
11:    $\mathbb{v}_m = \mathbb{v}_h - \mathbb{u}_m$                         # Forward sweep
12: until $\|\mathbb{s} - \mathrm{Diag}(\mathbb{v})\mathbb{i}^*\| < \varepsilon$
```

The BFS algorithm is illustrated below on an example network.

Example 3.3.1: BFS algorithm. Figure 3.2 shows a 3-bus radial network. Note the similarities with the AC circuit in Fig. 3.1 in previous section.

The BFS iteration is as follows. First, load current injections are determined as follows.

$$\mathbb{i}_2 = \left(\frac{p_2 + jq_2}{\mathbb{v}_2}\right)^*, \tag{3.41a}$$

$$\mathbb{i}_3 = \left(\frac{p_3 + jq_3}{\mathbb{v}_3}\right)^*. \tag{3.41b}$$

Next, the backward sweep computes line currents in two steps, as follows.

$$\mathbb{j}_3 = \mathbb{i}_3, \tag{3.42a}$$

$$\mathbb{j}_2 = \mathbb{i}_2 + \mathbb{j}_3. \tag{3.42b}$$

Line drops are computed as follows.

$$\mathbb{u}_2 = z_2\mathbb{j}_2, \tag{3.43a}$$

$$\mathbb{u}_3 = z_3\mathbb{j}_3. \tag{3.43b}$$

Finally, the forward sweep computes new bus voltages in two steps, as follows.

$$\mathbb{v}_2 = \mathbb{v}_1 - \mathbb{u}_2, \tag{3.44a}$$

$$\mathbb{v}_3 = \mathbb{v}_2 - \mathbb{u}_3. \tag{3.44b}$$

BFS algorithm using treefix operations. The BFS algorithm can be described using the so-called treefix operations, *+rootfix* and *+leaffix*, first introduced by [Leiserson 1988].

Definition 3.3.3 (+Rootfix)**.** Given a weighted tree, the +rootfix operation returns to each vertex the sum of its ancestors' weights.

Definition 3.3.4 (+Leaffix)**.** Given a weighted tree, the +leaffix operation returns to each vertex the sum of its descendants' weights.

Figure 3.3 illustrates these operations on a 6-node tree.

Algorithm 3.3 demonstrates the BFS algorithm using treefix operations. The network is translated into a tree, T. The vertices in T are the network buses; the edges are the network lines. The function setWeights($\boldsymbol{x}$) assigns weights from vector $\boldsymbol{x}$ to the vertices of the tree instance. The function getWeights() returns a vector containing the weights of the tree instance. The functions +rootfix() and +leaffix() apply the treefix operations on the tree instance.

The use of the treefix operations is relevant when considering the parallel implementation of the BFS algorithm, as will be discussed in Chapter 4.

3.3.2 Uncertainty assessment

The methods described above can be referred to as "deterministic", since they only consider one set of input data to produce one set of results. However, it often happens in PF analysis that several sets of input data need to be considered due to at PQ buses

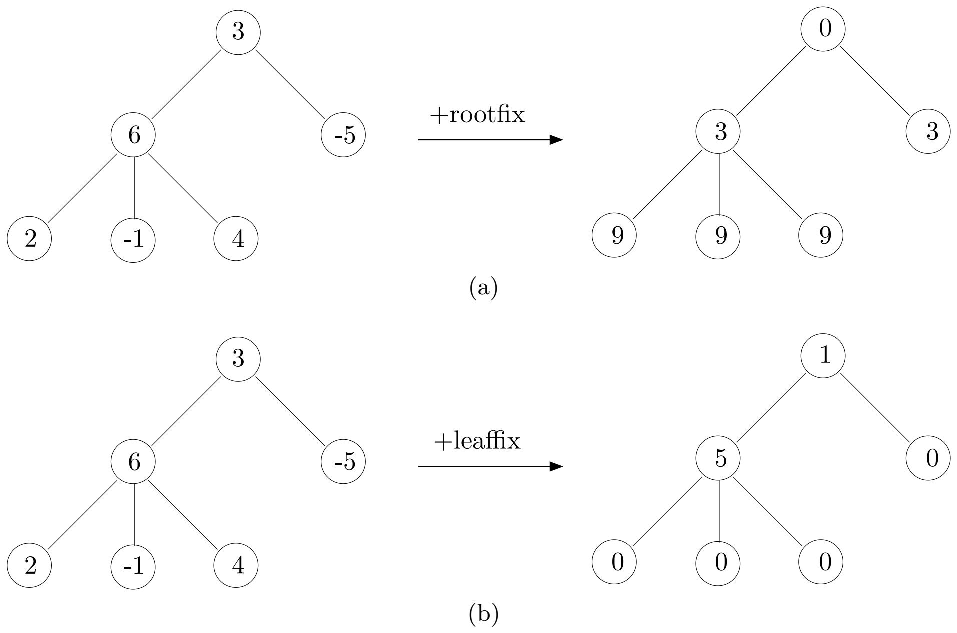


Figure 3.3: Treefix operations on a 6-node tree: (a) +Rootfix. (b) +Leaffix.

uncertainty. For example, this is the case of state estimation problems [Abur 2004]. Uncertainty in power systems takes multiple forms, e.g., imprecise demand forecast, price variability, renewable energy generation, economic growth and industry placement. Failing to account for uncertainties can lead to erroneous estimations and, therefore, improper planning and insecure operation of the system [Wan 1993].

The uncertainty assessment in PF analysis consists of finding all the possible states for a variety of input scenarios. Two modelling approaches are typically used, the *probabilistic* approach and the *interval* approach. Each of them leads to a particular family of methods. One big difference between them is that the probabilistic approach assumes that there is statistical data available, whereas the interval approach only requires a possibility distribution. However, the two methods are combined together and their results are seen as complementary of each other.

Probabilistic approach

The probabilistic approach consists in modelling the uncertainties as random variables with a given probability distribution (see Appendix A for a definition of random variable, distribution function and density function). This requires using statistical data to obtain the probability distribution of the inputs. A *probabilistic PF* model is defined by extending the PF equations to random variables. The equations are solved to obtain the distribution of the unknown variables. The solution method can be numerical, such as Monte Carlo, or analytical. For an extensive survey of probabilistic PF methods, see [Billinton 2001, Chen 2008].

Algorithm 3.3 BFS algorithm using treefix operations.

Input: Vector of complex power injections at PQ buses, $\mathbb{s}$. Vector of branch impedances, $\boldsymbol{z}$. Power network's associated tree, T. Maximum tolerance for the error, ε.

Output: Vector of complex voltages at PQ buses, $\mathbb{v}$.

```
1:  $\mathbb{v} = \mathbf{1}\angle\mathbf{0}$                                   # First guess
2:  repeat
3:      $\mathbb{i} = \left(\mathrm{Diag}(\mathbb{v})^{-1}\mathbb{s}\right)^*$          # Load currents
4:      $T$.setWeights($\mathbb{i}$)
5:      $T$.+leaffix()
6:      $\mathbb{I} = T$.getWeights()                        # Backward sweep
7:      $\mathbb{u} = \mathrm{Diag}(\boldsymbol{z})\,\mathbb{I}$                          # Line drops
8:      $T$.setWeights($-\mathbb{u}$)
9:      $T$.+rootfix()
10:     $\mathbb{v} = T$.getWeights()                         # Forward sweep
11: until $\|\mathbb{s} - \mathrm{Diag}(\mathbb{v})\mathbb{i}^*\| < \varepsilon$
```

Monte Carlo method. The Monte Carlo method is a numerical method to approximate the distribution of an unknown random variable. The method relies on the law of large numbers and sampling. The Monte Carlo consists of the following steps:

1. Create a number of scenarios by taking samples of known random variables.
2. For every scenario, compute a sample of the unknown variable using a deterministic model (e.g., simulation).
3. Aggregate the results into some relevant parameters.

Algorithm 3.4 illustrates the Monte Carlo method for probabilistic PF analysis. The algorithm computes the sample mean vector of bus voltages, $\overline{\boldsymbol{v}}$, given the distribution functions of bus power injections, $\boldsymbol{F_S}(\cdot)$. The function rng$(\cdot)$ returns a random number in the interval [0,1], and is used for sampling purposes.

Typically, a high number of samples is needed to obtain accurate results. For this reason, the method is rarely applied to real-world problems involved in the operation of the power system. It can be used for planning, or as a validation tool to test other techniques. Several examples can be found of the latter, in which the results from Monte Carlo simulations are interpreted as the "correct" ones [Vaccaro 2013, Saric 2006].

Analytical methods. These methods attempt to find analytical expressions for the distribution of the random state variables. They were first introduced by Borkowska and Allan [Borkowska 1974, Allan 1974].

In the basic formulation, nodal power injections are assumed to be mutually independent. This allows using convolution products to obtain the density of the

Algorithm 3.4 Monte Carlo method for probabilistic PF analysis.

Input: Number of samples, m. Distribution functions of bus power injections, $F_{\boldsymbol{S}}(\cdot)$. Deterministic PF equations, $\boldsymbol{s} = \boldsymbol{f}(\boldsymbol{v})$.

Output: Sample mean of bus voltages, $\check{\boldsymbol{v}}$.

```
1: for $h$ in $1, \dots, m$ do
2:     $r = \text{rng}()$
3:     $\boldsymbol{s}^{(h)} = \boldsymbol{F}_{\boldsymbol{S}}^{-1}(r)$                # Sampling
4:     $\boldsymbol{v}^{(h)} = \boldsymbol{f}^{-1}\left(\boldsymbol{s}^{(h)}\right)$    # Deterministic PF
5: $\check{\boldsymbol{v}} = \frac{1}{n}\sum_{h} \boldsymbol{v}^{(h)}$           # Expected value
```

bus voltages, once the PF equations have been linearized (see Appendix A for a definition of convolution product).

The PF equations are extended to random variables, as follows.

$$\boldsymbol{S} = \boldsymbol{f}(\boldsymbol{V}), \tag{3.45}$$

where $\boldsymbol{S}$ is the vector of random power injections and $\boldsymbol{V}$ is the vector of random bus voltages. The above equation is linearized around the estimated mean value of the bus voltages, $\widehat{\boldsymbol{V}}$. This yields the following expression.

$$\boldsymbol{V} \approx \widehat{\boldsymbol{V}} + \boldsymbol{J}_{\boldsymbol{f}}\Big|_{\boldsymbol{V}=\widehat{\boldsymbol{V}}} \left(\boldsymbol{S} - E(\boldsymbol{S})\right), \tag{3.46}$$

where $\boldsymbol{J}_{\boldsymbol{f}}$ is the Jacobian matrix of $\boldsymbol{f}(\cdot)$, and $E(\cdot)$ denotes the expected value (see Appendix A for a definition of these concepts). In scalar form, the above equation can be noted as follows.

$$V_h \approx \widehat{V}_h + \sum_k a_{hk} Z_k, \quad \forall\, h, \tag{3.47}$$

where $a_{hk} = \frac{\partial f_h}{\partial V_k}$, and $Z_k = S_k - E(S_k)$. As the nodal power injections, S_k, are mutually independent, so are the replacement variables, Z_k. Therefore, the density function of the bus voltages, X_h, can be approximated using convolution products, as follows.

$$f_{X_h}(x) \approx \left(\prod_k \frac{1}{|a_{hk}|}\right) \left(f_{Z_1}\left(\frac{x}{a_{h1}}\right) \otimes \ldots \otimes f_{Z_n}\left(\frac{x}{a_{hn}}\right)\right), \quad \forall\, h, \tag{3.48}$$

where $\otimes$ denotes the convolution product, and n is the number of buses.

The assumption of independence between the inputs, the use of a linearized model, and the difficulty of computing convolution products, are considered strong drawbacks of the technique above. There are several works which address these issues. For example, in [Da Silva 1984], linear correlation is assumed for the power inputs. In [Allan 1981], the PF equations are linearized around multiple points. In [Su 2005, Zhang 2004], the point-estimate method and the combined cumulants methods are used, instead of convolution, to fit the distribution of the state variables. The approach has also been applied to other analyses on power systems, such as Optimal Power Flow [Schellenberg 2005].

Interval approach

The interval approach consists of modelling the uncertainties as intervals without specifying a probability distribution. This allows using expert knowledge to generate intervals for the inputs, whenever statistical data is lacking. An *interval PF* model is defined by extending the PF equations to interval variables. The equations are solved to compute the interval bus voltages. The solution methods are self-validated, which ensures the reliability of the results [Alvarado 1992].

Interval Newton method. The interval Newton method is a method for bounding the zeros of a differentiable function, $\boldsymbol{f}(\cdot) : \mathbb{R}^n \to \mathbb{R}^n$. Given an initial interval guess, $[\boldsymbol{x}]^{(0)}$, the method computes a series, $[\boldsymbol{x}]^{(k)}$, $k = 1, 2, \ldots$, such that,

$$\boldsymbol{x} \in [\boldsymbol{x}]^{(0)}\,,\ \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0} \implies \boldsymbol{x} \in [\boldsymbol{x}]^{(k)} \subset [\boldsymbol{x}]^{(k-1)} \subset \ldots \subset [\boldsymbol{x}]^{(0)}\,. \tag{3.49}$$

The method consists of applying the mean value theorem with vectors in the current interval. This allows finding a new interval which contains the zeros. However, the new interval might be overlapping the current one. Therefore, both are intersected in order to compute the next interval in the series. Given the current interval, $[\boldsymbol{x}]^{(k)}$, the mean value theorem states that,

$$\boldsymbol{f}(\boldsymbol{x}) \in \boldsymbol{f}(\boldsymbol{y}) + \boldsymbol{J_f}\Big|_{\boldsymbol{x}=[\boldsymbol{x}]^{(k)}}(\boldsymbol{x} - \boldsymbol{y}), \quad \forall \boldsymbol{x}, \boldsymbol{y} \in [\boldsymbol{x}]^{(k)}\,, \tag{3.50}$$

where $\boldsymbol{J_f}$ is the Jacobian matrix of $\boldsymbol{f}(\cdot)$. Note that the Jacobian matrix is evaluated in the interval, $[\boldsymbol{x}]^{(k)}$. Thus, the result is an interval matrix. Imposing, $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0}$, and choosing $\boldsymbol{y}$ as $\check{\boldsymbol{x}}^{(k)}$ (the midpoint of $[\boldsymbol{x}]^{(k)}$), allows obtaining the following expression.

$$\boldsymbol{x} \in \check{\boldsymbol{x}}^{(k)} - \left(\boldsymbol{J_f}\Big|_{\boldsymbol{x}=[\boldsymbol{x}]^{(k)}}\right)^{-1} \boldsymbol{f}(\check{\boldsymbol{x}}^{(k)}), \tag{3.51}$$

which leads to the interval Newton iteration,

$$[\boldsymbol{x}]^{(k+1)} = [\boldsymbol{x}]^{(k)} \bigcap \left(\check{\boldsymbol{x}}^{(k)} - \left(\boldsymbol{J_f}\Big|_{\boldsymbol{x}=[\boldsymbol{x}]^{(k)}}\right)^{-1} \boldsymbol{f}(\check{\boldsymbol{x}}^{(k)})\right). \tag{3.52}$$

The interval Newton method is applied to interval PF analysis in the same way as the NR method is applied to deterministic PF analysis (see Section 3.3.1). This idea was introduced in [Wang 1992].

Note that the interval Newton iteration requires inverting an interval matrix (or, alternatively, solving a system of interval linear equations). Typical factorization techniques include interval Gaussian elimination and the Krawczyk's method [Moore 1979]. However, depending on the structure of the matrix, these methods can either not converge, take too long or simply deliver an impractical result [Barboza 2004].

In Chapter 5, a novel method for interval PF analysis is proposed which does not require factorizing an interval matrix. Another approach using range arithmetic and the mixed complementarity problem formulation has been proposed in [Vaccaro 2013].

3.4 Conclusions

This chapter provides a literature review of several aspects of circuit and PF analysis. Topics from classic circuit analysis to the uncertainty assessment are discussed. Different models and solution methods are described and some of them illustrated with examples.

Following chapters tackle some relevant aspects of the techniques presented above. Chapter 4 proposes a novel vectorial implementation of the BFS algorithm, based on the study of treefix operations. How much faster can this solution be, compared to the traditional approach based on sequential tree traversals? Experimental results show that a speedup of 2 to 9 can be achieved, depending on several implementation features.

Chapter 5 proposes a novel method for the uncertainty assessment in PF analysis. Current interval approaches are affected by the dependency problem, which causes a critical loss of accuracy in the final results. Is it possible to overcome that issue? In the proposed method, the above is achieved by using an optimization problem to compute the radius of the solution. The proposed method provides an accuracy similar to that of the Monte Carlo approach, with a much lower computational complexity.

Chapter 6 proposes a novel fixed-point method for PF analysis, which is intrinsically parallel. How advantageous this approach can be, compared to parallelizing existing serial approaches? Two features of the proposed method are particularly interesting: (i) performance does not depend as much on the problem size, as it does in traditional PF approaches; and (ii) performance is able to scale with the number of computing cores, automatically benefiting from improvements in the computer architecture.

References

[Abur 2004] A. Abur and A. G. Exposito. Power system state estimation: theory and implementation. CRC Press, 2004. (Cited on page 45.)

[Allan 1974] R. Allan, B. Borkowska and C. Grigg. *Probabilistic analysis of power flows.* Proceedings of the Institution of Electrical Engineers, vol. 121, no. 12, pp. 1551–1556, 1974. (Cited on page 46.)

[Allan 1981] R. Allan and A. L. Da Silva. *Probabilistic load flow using multilinearisations.* IEE Proceedings C (Generation, Transmission and Distribution), vol. 128, pp. 280–287. IET, 1981. (Cited on page 47.)

[Alvarado 1992] F. Alvarado, Y. Hu and R. Adapa. *Uncertainty in power system modeling and computation.* Systems, Man and Cybernetics, 1992., IEEE International Conference on, pp. 754–760. IEEE, 1992. (Cited on pages 10 and 48.)

[Barboza 2004] L. V. Barboza, G. P. Dimuro and R. H. Reiser. *Towards interval analysis of the load uncertainty in power electric systems.* Probabilistic Methods Applied to Power Systems, 2004 International Conference on, pp. 538–544. IEEE, 2004. (Cited on page 49.)

[Billinton 2001] R. Billinton, M. Fotuhi-Firuzabad and L. Bertling. *Bibliography on the application of probability methods in power system reliability evaluation 1996-1999.* IEEE Transactions on Power Systems, vol. 16, no. 4, pp. 595–602, 2001. (Cited on page 45.)

[Borkowska 1974] B. Borkowska. *Probabilistic load flow.* IEEE Transactions on Power Apparatus and Systems, vol. 93, no. 3, pp. 752–759, 1974. (Cited on page 46.)

[Chen 2008] P. Chen, Z. Chen and B. Bak-Jensen. *Probabilistic load flow: A review.* Third International Conference on Electric Utility Deregulation and Restructuring and Power Technologies, 2008. DRPT 2008., pp. 1586–1591, 2008. (Cited on page 45.)

[Da Silva 1984] A. Da Silva, V. Arienti and R. Allan. *Probabilistic load flow considering dependence between input nodal powers.* IEEE Transactions on Power Apparatus and Systems, vol. 6, no. PAS-103, pp. 1524–1530, 1984. (Cited on page 47.)

[Kersting 2012] W. H. Kersting. Distribution system modeling and analysis. CRC Press, 2012. (Cited on pages 7 and 37.)

[Leiserson 1988] C. E. Leiserson and B. M. Maggs. *Communication-efficient parallel algorithms for distributed random-access machines.* Algorithmica, vol. 3, no. 1-4, pp. 53–77, 1988. (Cited on pages 44, 54 and 55.)

[Machowski 1997] J. Machowski, J. Bialek and J. Bumby. Power system dynamics and stability. Wiley, 1997. (Cited on pages 10 and 36.)

[Mendive 1975] D. Mendive. An application of ladder network theory to the solution of three-phase radial load-flow problems. New Mexico State University, 1975. (Cited on page 42.)

[Milano 2010] F. Milano. Power system modelling and scripting. Springer Science & Business Media, 2010. (Cited on pages 7, 10, 37, 120 and 124.)

[Moore 1979] R. E. Moore. Methods and applications of interval analysis. Studies in Applied and Numerical Mathematics. Society for Industrial and Applied Mathematics, 1979. (Cited on page 49.)

[Nilsson 2009] J. W. Nilsson and S. A. Riedel. Electric circuits, vol. 8. Prentice Hall, 2009. (Cited on page 33.)

[Saric 2006] A. T. Saric and A. M. Stankovic. *Ellipsoidal approximation to uncertainty propagation in boundary power flow.* IEEE PES Power Systems Conference and Exposition, 2006, pp. 1722–1727, 2006. (Cited on page 46.)

[Schellenberg 2005] A. Schellenberg, W. Rosehart and J. Aguado. *Cumulant-based probabilistic optimal Power Flow (P-OPF) with Gaussian and gamma distributions.* IEEE Transactions on Power Systems, vol. 20, no. 2, pp. 773–781, 2005. (Cited on page 48.)

[Shirmohammadi 1988] D. Shirmohammadi, H. Hong, A. Semlyen and G. Luo. *A compensation-based Power Flow method for weakly meshed distribution and transmission networks.* Power Systems, IEEE Transactions on, vol. 3, no. 2, pp. 753–762, 1988. (Cited on page 42.)

[Stott 1974] B. Stott. *Review of load-flow calculation methods.* Proceedings of the IEEE, vol. 62, no. 7, pp. 916–929, 1974. (Cited on pages 10 and 32.)

[Su 2005] C.-L. Su. *Probabilistic load-flow computation using point estimate method.* Power Systems, IEEE Transactions on, vol. 20, no. 4, pp. 1843–1851, 2005. (Cited on page 47.)

[Vaccaro 2013] A. Vaccaro, C. A. Cañizares and K. Bhattacharya. *A range arithmetic-based optimization model for Power Flow analysis under interval uncertainty.* IEEE Transactions on Power Systems, vol. 28, no. 2, pp. 1179–1186, 2013. (Cited on pages 11, 46, 49, 95, 96 and 98.)

[Wan 1993] Y.-h. Wan and B. K. Parsons. Factors relevant to utility integration of intermittent renewable technologies. National Renewable Energy Laboratory, 1993. (Cited on pages 32 and 45.)

[Wang 1992] Z. Wang and F. Alvarado. *Interval arithmetic in Power Flow analysis.* IEEE Transactions on Power Systems, vol. 7, no. 3, pp. 1341–1349, 1992. (Cited on page 48.)

[Zhang 2004] P. Zhang and S. T. Lee. *Probabilistic load flow computation using the method of combined cumulants and Gram-Charlier expansion.* IEEE Transactions on Power Systems, vol. 19, no. 1, pp. 676–682, 2004. (Cited on page 47.)

Chapter 4

Treefix operations on the GPU

Contents

The works in this chapter have been presented at the International Conference of Computational Science (ICCS) 2013 [Defour 2013].

4.1 Introduction

This chapter discusses the implementation on GPUs of the so-called *treefix* operations. These operations can be used to describe and implement the Backward-Forward Sweep (BFS) algorithm for the analysis of radial power networks, as discussed in Section 3.3.1.

In the past, the parallel implementation of treefix operations was studied as part of higher-level algorithms on trees. Two strategies were proposed, one based on regularity and one based on load-balancing [Merrill 2012, Blelloch 1990]. The one based on regularity is able to efficiently exploit the SIMT execution model. However, it introduces extra arithmetic operations. On the other hand, the algorithm based on load-balancing is work-efficient, i.e., it performs the same amount of arithmetic operations as an optimal serial implementation. However, the load-balancing overhead can become critical on SIMT architectures.

The above unleashes a series of questions, which have not been addressed by literature. Which of the two approaches is more suitable to GPU computing? Is the cost of extra arithmetic operations in the regular algorithm compensated by efficiently exploiting the SIMT model? Is the cost of load-balancing compensated by being work-efficient? What floating-point issues are raised? This chapter attempts to answer these questions by implementing and testing both algorithms on the GPU.

The remainder of the chapter is divided as follows. Section 4.2 recalls the treefix operations and introduces the two main approaches for parallel implementation. Section 4.3 presents relevant issues regarding the implementation of such approaches on GPUs. A suite of tests comparing performance and accuracy of both implementations, and a case study regarding the analysis of a radial power network are presented in Section 4.4. Finally, Section 4.5 draws relevant conclusions.

4.2 Treefix operations

The treefix operations were first introduced by Leiserson and Maggs [Leiserson 1988] to describe certain computations over weighted trees. Two operations were defined, *+rootfix* and *+leaffix*. The term treefix was later introduced by Blelloch [Blelloch 1990]. Let us recall the definition of these operations given in Section 3.3.1.

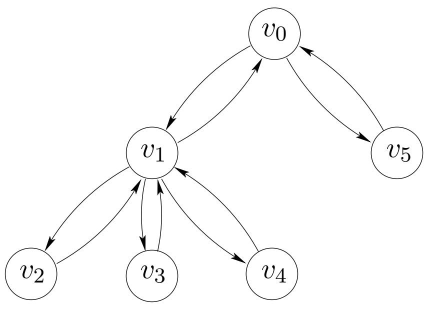
Definition 3.3.3 (+Rootfix)**.** Given a weighted tree, the +rootfix operation returns to each vertex the sum of its ancestors' weights.

Definition 3.3.4 (+Leaffix)**.** Given a weighted tree, the +leaffix operation returns to each vertex the sum of its descendants' weights.

+Rootfix and +leaffix can be performed in parallel using two approaches, described below.

Vectorial approach. The vectorial approach was studied by Blelloch [Blelloch 1990], who proposed a vectorial representation and algorithm [Blelloch 1990]. The representation is based on Euler-tour ordering and consists of two vectors of left and right parentheses. The algorithm is based on the *scan* operation and other highly efficient parallel primitives. The main advantage of this algorithm is the instruction and memory regularity, which allows it to map efficiently into the SIMT execution model of the GPU. The main issue is twofold: first, it requires to build a vector of twice the size of the tree, which means doubling the amount of memory used for data storage in comparison with the parallel approach. Second, the underlying operation needs to be associative, which is not the case of floating-point addition and therefore could result in a loss of accuracy.

Parallel approach. The parallel approach is based on the breadth-first search algorithm. Given a tree and a root-vertex, this algorithm visits all the vertices in the tree in breadth-first order starting from the root. To compute +rootfix, it suffices to add the weight of the parent to the weight of the children as the vertices are visited. The main advantage of this approach is that it is work-efficient, i.e., it performs the same amount of arithmetic operations as an optimal serial approach



$$E' = \{v_0v_1, v_1v_2, v_2v_1, v_1v_3, v_3v_1, v_1v_4, v_4v_1, v_1v_0, v_0v_5, v_5v_0\}$$

Figure 4.1: Example tree and edge set E' in Euler-tour order.

and thus does not suffer from accuracy issues. The main disadvantage is that it requires load-balancing to be implemented on a SIMT architecture, which typically causes overhead.

Two other algorithms have been developed for specific type of trees. The first one is a randomized algorithm for the balanced binary tree proposed by Leiserson and Maggs [Leiserson 1988]. It is based on the contraction technique developed by Miller and Reid [Miller 1985]. The second is for the linked list (in which each vertex has exactly one child) and is based on symmetry breaking. Both algorithms have a time complexity of $\mathcal{O}(\lg n)$, and both are easy to implement in parallel.

4.2.1 Vectorial approach

Vectorial representation

In the vectorial representation, the tree is viewed as an undirected tree $G = (V, E)$, where V is a set of n vertices and E is a set of $n - 1$ undirected edges. Each undirected edge in E is replaced by two anti-parallel edges into a new set E', which allows to trace an Euler-tour around the tree. The definition of Euler-tour is as follows.

Definition 4.2.1 (Euler-tour [Tarjan 1984])**.** Let $G' = (V, E')$ be a directed tree, where

$$v_iv_j \in E' \iff v_jv_i \in E'. \tag{4.1}$$

A closed path around the tree that visits every edge exactly once is called an Euler-tour.

The Euler-tour ordering consists of arranging the edges in E' as they are visited in an Euler-tour that starts from the root and proceeds counter-clockwise. Figure 4.1 shows an example tree and the associated set of directed edges E' after the ordering. Euler-tours can be found by using a parallel algorithm in $\mathcal{O}(\lg n)$ steps [Atallah 1984].

The set E' in Euler-tour ordering is translated using a parenthetical notation into a vector $\boldsymbol{e}$. In this vector, the element noted '${}_(i$' represents the Euler-tour reaching vertex v_i while going down, i.e., away from the root; the element noted '$i_)$', in turn, represents the Euler-tour leaving vertex v_i while going up, i.e., back to the root. The elements noted '${}_(0$' and '$0_)$', added at the beginning and end of the vector, represent fictitious edges reaching and leaving the root. For the example tree in Fig. 4.1, the vector $\boldsymbol{e}$ is the following.

$$\boldsymbol{e} = [{}_(0\ {}_(1\ {}_(2\ 2_)\ {}_(3\ 3_)\ {}_(4\ 4_)\ 1_)\ {}_(5\ 5_)\ 0_)] \tag{4.2}$$

The vectorial representation consists of two vectors, $\boldsymbol{l}$ and $\boldsymbol{r}$, holding respectively, at each position i, the positions of ${}_(i$ and $i_)$ within vector $\boldsymbol{e}$. For the example tree shown in Fig. 4.1, vector $\boldsymbol{e}$ has 12 positions, numbered from 0 to 11, and the vectors $\boldsymbol{l}$ and $\boldsymbol{r}$ are the following.

$$\boldsymbol{l} = [0\ 1\ 2\ 4\ 6\ 9] \tag{4.3}$$

$$\boldsymbol{r} = [11\ 8\ 3\ 5\ 7\ 10] \tag{4.4}$$

Vectorial algorithm

The vectorial algorithm is based on the scan operation. This operation takes a vector of data, and returns a new vector with the result of applying a binary operator over all elements up to each position [Iverson 1962].

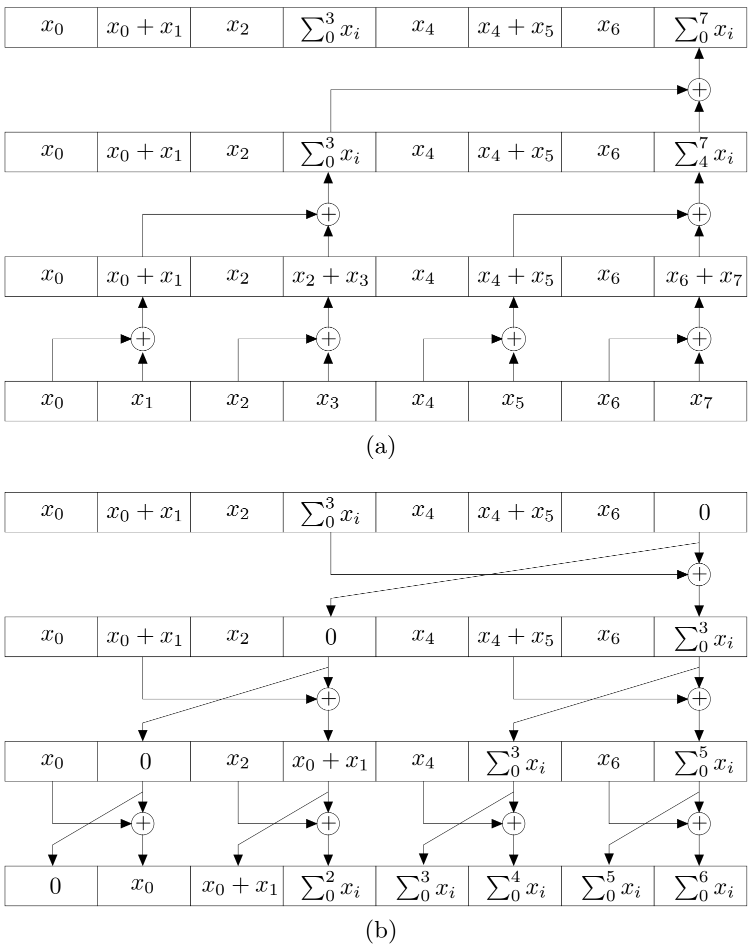
Definition 4.2.2 (Scan)**.** Let $\circ$ be a binary associative operator with identity i, and $\boldsymbol{x} = (x_0, \ldots, x_{n-1})$, a vector. Then,

$$\circ\text{scan}(\boldsymbol{x}) = (i, x_0, x_0 \circ x_1, \ldots, x_0 \circ x_1 \circ \ldots \circ x_{n-2}), \tag{4.5}$$

is called the scan operation over $\boldsymbol{x}$.

The scan operation can be efficiently implemented in parallel using a regular approach. The original algorithm was developed by Blelloch [Blelloch 1990] and relies on building a *balance binary tree* over the input vector. This algorithm is work-efficient, i.e., it has the same complexity as a sequential algorithm that simply adds the elements in the input vector, and stores the partial sums in an output vector.

The balanced binary tree is built so that each leaf corresponds to a vector case. The algorithm consists of two phases, the *up-sweep* (also known as *reduce* phase) and the *down-sweep*. The up-sweep traverses the tree from leaves to root, computing partial sums in the internal nodes. At the end, the root contains the sum of all the elements in the vector. The down-sweep traverses the tree from root to leaves, computing the result of the scan from the stored partial sums. Before starting the down-sweep, the last element of the vector is zeroed. The above is illustrated



x_0 | x_0+x_1 | x_2 | $\sum_0^3 x_i$ | x_4 | x_4+x_5 | x_6 | $\sum_0^7 x_i$
x_0 | x_0+x_1 | x_2 | $\sum_0^3 x_i$ | x_4 | x_4+x_5 | x_6 | $\sum_4^7 x_i$
x_0 | x_0+x_1 | x_2 | x_2+x_3 | x_4 | x_4+x_5 | x_6 | x_6+x_7
x_0 | x_1 | x_2 | x_3 | x_4 | x_5 | x_6 | x_7
(a)
x_0 | x_0+x_1 | x_2 | $\sum_0^3 x_i$ | x_4 | x_4+x_5 | x_6 | 0
x_0 | x_0+x_1 | x_2 | 0 | x_4 | x_4+x_5 | x_6 | $\sum_0^3 x_i$
x_0 | 0 | x_2 | x_0+x_1 | x_4 | $\sum_0^3 x_i$ | x_6 | $\sum_0^5 x_i$
0 | x_0 | x_0+x_1 | $\sum_0^2 x_i$ | $\sum_0^3 x_i$ | $\sum_0^4 x_i$ | $\sum_0^5 x_i$ | $\sum_0^6 x_i$
(b)

Figure 4.2: Work-efficient parallel scan algorithm: (a) Up-sweep. (b) Down-sweep.

in Fig. 4.2. This is a regular algorithm, as the operations that are performed in parallel are exactly the same for each level of the tree.

The up-sweep performs $n-1$ additions, whereas the down-sweep performs $n-1$ additions and $n-1$ swaps. As a consequence, the algorithm has a work complexity of $\mathcal{O}(n)$. However, as each sweep performs only one step per level of the tree, the algorithm has a *step* complexity of $\mathcal{O}(\log n)$.

Algorithm 4.1 details the procedure to perform +rootfix using the +scan operation, given the vectorial representation in form of the vectors of left and right parentheses, $\boldsymbol{l}$ and $\boldsymbol{r}$. First, the position of the left and right parentheses are read for each vertex, and the vertex's weight and its inverse are written at those positions in a working vector $\boldsymbol{h}$ of size $2n$, where n is the size of the tree. Second, an inplace parallel +scan is run over $\boldsymbol{h}$. Third, the result is gathered from the new $\boldsymbol{h}$ in the position of the left parenthesis.

The vectorial representation allows the complexity of the algorithm to be detached from the tree topology, and depend only on the tree size. However, two assumptions needs to be met by the arithmetic operation in use (the addition, in

Algorithm 4.1 Vectorial rootfix algorithm

Input: Left and right parenthesis vectors, $\boldsymbol{l}$ and $\boldsymbol{r}$; weights vector $\boldsymbol{w}$.
Output: Result vector $\boldsymbol{g}$.

```
1: for all $i \in \{0, \ldots, n-1\}$ do in parallel
2:     $l = l_i,\ r = r_i$
3:     $h_l = w_i$
4:     $h_r = -w_i$
5: Run an inplace +scan on $\boldsymbol{h}$
6: for all $i \in \{0, \ldots, n-1\}$ do in parallel
7:     $l = l_i$
8:     $g_i = h_l$
```

this case). First, that it has an inverse, so that instruction 4 can be performed. Second, that it is associative, so that the scan operation does not distort the results. Both assumptions are met by the integer addition, however, the second is not met by the floating-point addition. As a consequence, the algorithm is introducing additional sources of numerical error compared to the parallel one.

4.2.2 Parallel approach

The parallel approach is based on the breadth-first search algorithm. Given a graph G and a vertex v_0, this algorithm traverses the vertices of G in breadth-first order starting from v_0. As visited, each vertex is marked either with: (i) its distance to v_0, or (ii) its predecessor in the shortest path from v_0.

The breadth-first order of search implies visiting all the vertices in one level before passing to the next. Accordingly, the algorithm performs one iteration per level. The set of vertices to be visited during a certain iteration (i.e., the set of vertices in that level) is called the *vertex frontier*. For each iteration, the vertex frontier is determined by (i) expanding the vertices visited during the previous iteration into their neighbours, and (ii) filtering out duplicates and already visited vertices. The neighbours are given by the *adjacency lists*, defined as follows.

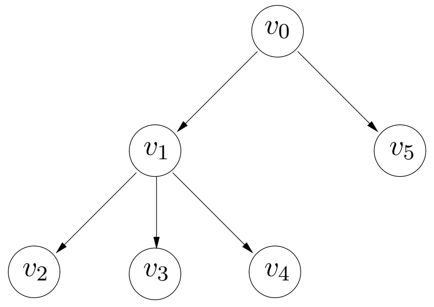
Definition 4.2.3 (Adjacency list)**.** Let $G = (V, E)$ be a directed graph, with V a set of vertices, and E a set of edges. Let v_iv_j be an edge directed from vertex v_i to v_j. Then, the set

$$\mathcal{A}_i = \{v_j \mid v_iv_j \in E\}, \tag{4.6}$$

is called the adjacency list of vertex v_i in graph G.

Note that, in the case of trees, the adjacency list correspond to the list of children of one vertex. Figure 4.3 shows a 6-vertex example tree and the corresponding adjacency lists.

Using adjacency lists for neighbours gathering allows the algorithm to be work-efficient. Indeed, the vertex frontier contains only the vertices to be visited during each iteration.



$$\mathcal{A}_0 = \{v_1, v_5\}$$
$$\mathcal{A}_1 = \{v_2, v_3, v_4\}$$
$$\mathcal{A}_2 = \mathcal{A}_3 = \mathcal{A}_4 = \mathcal{A}_5 = \emptyset$$

Figure 4.3: 6-vertex tree and adjacency lists.

Algorithm 4.2 details the procedure to perform breadth-first search by expanding vertices in parallel. The vertex frontier is managed using two queues, an *input* queue and an *output* queue. At the beginning of each iteration, the input queue contains the vertex frontier from the previous iteration. These vertices are expanded into their neighbours using the adjacency lists. As gathered, the neighbours are filtered and inserted into the output queue, which subsequently contains the vertex frontier for the current iteration. The function safeEnqueue(v_i) inserts the vertex v_i at a reserved position in the queue instance. At the end of the iteration, the output queue is copied into the input queue to be consumed by the next iteration.

Algorithm 4.2 Parallel breadth-first search algorithm.

Input: Graph $G = (V, E)$, and adjacency lists $\mathcal{A}_i$, for each vertex $v_i \in V$; root vertex v_0.

Output: Vector $\boldsymbol{d}$, holding distances from each vertex to v_0.

```
 1: for all $i\colon v_i \in V$ do in parallel
 2:     $d_i = \infty$
 3: $d_0 = 0$
 4: $inQ = \{\}$
 5: $inQ$.safeEnqueue($v_0$)
 6: while $inQ$ != $\{\}$ do
 7:     $outQ = \{\}$
 8:     for all $i\colon v_i \in inQ$ do in parallel
 9:         for all $j\colon v_j \in \mathcal{A}_i$ do
10:             if $d_j = \infty$ then
11:                 $outQ$.safeEnqueue($v_j$)
12:                 $d_j = d_i + 1$
13:     $inQ = outQ$
```

In order to perform +rootfix using the breadth-first search algorithm, the weight of the parent needs to be added to the weight of the children as the latter are visited. This is illustrated in Algorithm 4.3. The work-flow is very similar to Algorithm 4.2. However, in this case, there is no need to check for duplicates or already visited vertices before inserting gathered children into the output queue. Indeed, the tree structure ensures that every vertex only belongs to one adjacency list (the one of its parent).

Algorithm 4.3 Parallel +rootfix algorithm

```
Input: Tree $T = (V, E)$, and adjacency lists $\mathcal{A}_i$, for each $v_i \in V$; root vertex $v_0$;
    weights vector $\boldsymbol{w}$.
Output: Result vector $\boldsymbol{g}$.
 1: $g_0 = 0$
 2: $inQ = \{\}$
 3: $inQ$.safeEnqueue($v_0$)
 4: while $inQ$ != $\{\}$ do
 5:     $outQ = \{\}$
 6:     for all $i: v_i \in inQ$ do in parallel
 7:         for all $j: v_j \in \mathcal{A}_i$ do in parallel
 8:             $g_j = g_i + w_i$
 9:             $outQ$.safeEnqueue($v_j$)
10:     $inQ = outQ$
```

The amount of parallel work that this algorithm can perform depends on the tree topology. The broader the tree, the more vertices that can be visited in parallel at each iteration. In the worst case scenario, when the input tree is a linked list, the algorithm is forced to visit vertices one at a time, just as the serial version. On the other hand, the algorithm is work-efficient. That is to say, the result for every vertex is obtained exactly as the sum of the ancestors' weights. This corresponds to the minimal amount of operations needed to compute +rootfix, whatever the algorithm. As a consequence, no additional errors are introduced when using floating-point data to run the algorithm.

4.3 GPU implementation

This section describes the implementation on GPUs of the two approaches presented above.

4.3.1 Vectorial approach

Algorithm 4.1 is implemented using OpenCL. The implementation is built around three separate kernels, *seed*, *scan* and *reap*. The seed kernel performs the first parallel for (lines 1 to 4), the scan kernel performs the parallel +scan operation (line 5), and the reap kernel performs the second parallel for (lines 6 to 8). The seed and reap kernels are launched with n work-items, where n is the size of the input vectors, $\boldsymbol{l}$, $\boldsymbol{r}$ and $\boldsymbol{w}$. These work-items are packed in workgroups of maximum size, which according to our tests allows for maximizing performance. The appropriate parameter can be found via 'clGetKernelWorkGroupInfo'.

All three kernels are memory-bound, however they differ in their memory access pattern. Only the scan kernel has a regular pattern; the seed and reap kernels are forced to perform at least one uncoalesced access. The memory access strategy breaks down as follows. In the scan kernel, every work-item processes two consecutive elements in the working vector $\boldsymbol{h}$, and so all the accesses are coalesced. In the seed kernel, every work-item reads an element in vectors $\boldsymbol{l}$, $\boldsymbol{r}$ and $\boldsymbol{w}$, and writes another in vector $\boldsymbol{e}$; all three reads are coalesced, whereas the write to $\boldsymbol{h}$ is uncoalesced. In the reap kernel, every work-item reads an element in vectors $\boldsymbol{l}$ and $\boldsymbol{h}$, and writes another in vector $\boldsymbol{g}$; the read from $\boldsymbol{l}$ and the write to $\boldsymbol{g}$ are coalesced, although the read from $\boldsymbol{h}$ is uncoalesced. These uncoalesced accesses are inherent to the algorithm.

On a pre-Fermi architecture, the Nvidia profiler reports a global memory load efficiency of only about 25% for the seed and reap kernels. This is a consequence of the irregular access pattern to vector $\boldsymbol{h}$. The efficiency is improved on Fermi, due to the presence of L1 and L2 cache, reaching about 50%. See [Glaskowsky 2009] for a complete overview of the Fermi architecture.

4.3.2 Parallel approach

Algorithm 4.3 requires load-balancing in order to be efficiently implemented on GPUs. Recently, a high performance implementation has been proposed by Merril *et al* [Merrill 2012]. This implementation uses at least three different load-balancing strategies to gather neighbours from the adjacency lists. In all these strategies, one thread is first assigned to each vertex in the input queue, allowing to perform the parallel loop in line 8 of Algorithm 4.2. The loop in line 9 is performed in a distinct way by each strategy, as described below.

Serial gathering. Each thread gathers its adjacency lists individually, without cooperation from other threads. That is to say, the loop in line 9 is simply a serial one. This strategy causes serious imbalance when the adjacency lists have very different sizes.

Warp-based gathering. Each thread tries to control its warp by writing its identifier into a word shared by all the threads in the warp. Only one of these writes succeeds (the one that is performed last), and that thread is allowed to enlist the entire warp in gathering its assigned adjacency list in parallel. The process is repeated until all the threads in the warp have their corresponding vertex expanded. This strategy provides better load balance than the serial one, however, it presents two main issues. The first is that threads are left unused whenever the size of an adjacency list is not a multiple of the warp-width. The second is warp imbalance, which may happen when threads within one warp have adjacency lists much larger than threads in other warps.

Scan-based gathering. Each thread serially writes its adjacency list into a vector shared by the entire block. The offset of each list is determined by a parallel scan over an auxiliary vector which contains the lists' sizes. Afterwards, each thread in the block gathers one neighbour from the shared vector. This strategy allows threads in the same block to gather neighbours completely in parallel. This is a clear advantage when the data of those neighbours is read from global memory, as the accesses are fully coalesced. On the other hand, load imbalance may occur during the process of writing the adjacency lists into the shared vector. As this process is performed by a serial loop which spans over multiple threads, the ones with shorter lists are forced to remain idle during the last cycles.

By using these and other load-balancing techniques, the implementation by Merril *et al* achieves a graph traversal rate in excess of 3.3 billion edges per second. This rate is about five times greater than the one obtained by previous GPU implementations [Merrill 2012].

4.4 Tests and results

4.4.1 Performance

As mentioned in Section 4.2.2, when using the parallel approach to perform the +rootfix operation, one can expect the tree topology to have an impact on performance. The amount of parallelism allowed by the particular topology is given by the *average branching factor*, i.e., the ratio between the total number of vertices and the total number of levels in the tree. The larger this factor is, the better for the parallel algorithm as there are more vertices to visit in parallel, in average. On the contrary, as stated in Section 4.2.1, when using the vectorial approach the impact of the tree topology over performance should be negligible.

To validate these hypotheses, we used a group of benchmark trees with different topologies generated as follows. First, ten matrices are selected from the University of Florida Sparse Matrix Collection, corresponding to a suit of benchmarks used in the 10th DIMACS Implementation Challenge [UFS 2012, DIM 2012]. These matrices are representative of the type of networks that can be found in real-life applications in terms of topology and size. Next, the minimum spanning tree of each of these networks is computed. The resulting benchmarks are presented in Table 4.1 ordered by increasing number of vertices. Note the differences in number of levels and average branching factor in the third and forth column.

The tests consists of running the parallel and vectorial algorithms on every benchmark measuring execution time. Additionally, a purely sequential algorithm is run in order to have a mean of comparison with traditional single-core CPU approaches. Table 4.2 summarizes the computing environment used in the tests. The tests program are compiled using GCC 4.6.3, CUDA 4.2.1 and OpenCL 1.1.

The execution time is split into: (a) computation time, spent in the actual calculation, and (b) data transfer time, spent in copying data between host and

Table 4.1: Suite of benchmark trees.

Name	Nb. of vertices	Nb. of levels	Avg. branching factor
af_shell9	504,855	490	1,030
audikw1	943,695	236	3,998
ldoor	952,203	784	1,214
af_shell10	1,508,065	1,098	1,373
G3_circuit	1,585,478	705	2,248
kkt_power	2,063,494	36	57,319
nlpkkt120	3,542,400	123	28,800
cage15	5,154,859	81	63,640
nlpkkt160	8,345,600	163	51,200
nlpkkt200	16,240,000	203	80,000

device. The reason for this decoupling is that several applications, including the BFS algorithm, only need to transfer the data once per several runs of the +rootfix operation. In such case the computation times is more relevant.

Table 4.2: Test environment.

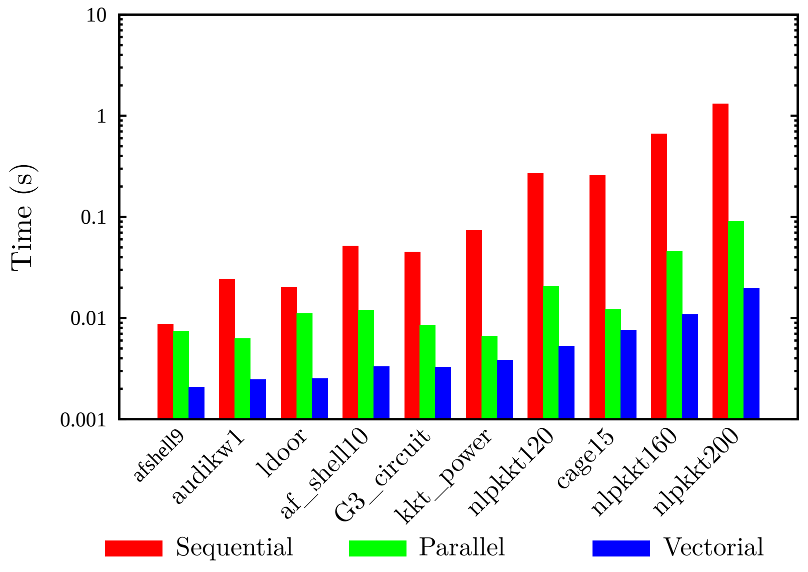
Device	Compute capability	Number of cores
Xeon E645	N/A	12 (1 used)
GeForce GTX 670	2.0	1,344

Figure 4.4(a) shows the related performance of sequential, parallel and vectorial +rootfix over the set of benchmarks ordered by increasing number of vertices. The computation time of both the sequential and parallel implementations exhibit an irregular pattern in which, sometimes, a larger tree is treated in lesser time than a smaller one. In the case of the vectorial implementation, the computation time is constantly increasing with the tree size.

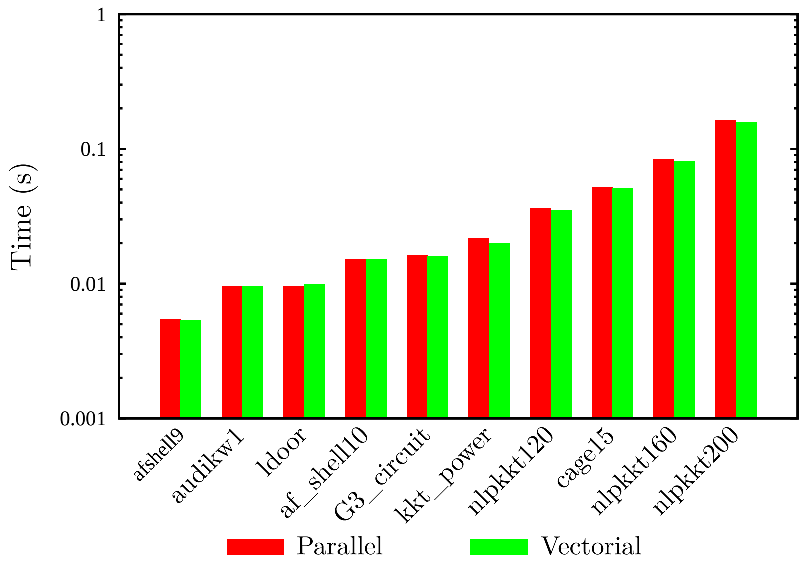
The data transfer time is almost the same for both the parallel and vectorial implementations, as shown in Fig. 4.4(b). This is due to the fact that both implementations transfer the same amount of data. From host to device, the parallel implementation transfers two arrays of sizes $n - 1$ and $n + 1$, containing the adjacency lists in compressed sparse row (CSR) form. The vectorial implementation transfers the vectors of left and right parentheses, both of size n. From device to host, both implementations transfer the result in one array of size n.

Figure 4.5 shows the speedup of parallel and vectorial +rootfix over sequential +rootfix, considering only computation time. The acceleration is quite substantial, especially in the case of our vectorial implementation where it reaches a speed-up greater than 60 on the two largest benchmarks.

Figure 4.6 shows the vertex density of two benchmarks, namely ‘nlpkkt120’ and



(a)



(b)

Figure 4.4: Execution time (s) of sequential, parallel and vectorial +rootfix on GTX 670. The benchmarks are ordered by increasing number of vertices. (a) Computation time. (b) Data transfer time.

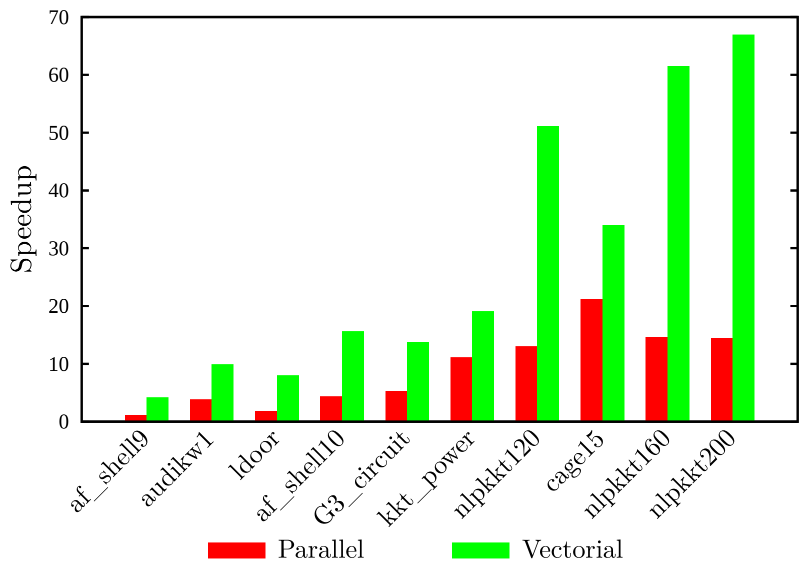


Figure 4.5: Speedup of parallel and vectorial +rootfix over sequential +rootfix on GTX 670.

'cage15'. Note that the latter has fewer levels even though it has more vertices; this explains why it is processed faster by the parallel implementation (see Fig. 4.4). This validates the hypothesis about the topology (via the average branching factor) having an impact on performance in the parallel implementation.

This effect can be further seen in Figures 4.7(a) and 4.7(b). Here, two extreme cases of topology are considered: (a) caterpillar, where every vertex has exactly one child, and (b) star, where the root has $n-1$ children. In the caterpillar the average branching factor takes its minimum value (1), whereas in the star it takes its maximum (n). Next, random trees of 2^{15} to 2^{24} vertices are generated having these special topologies, and the different algorithms are run on them. The results show that the parallel implementation, as expected, performs poorly on the caterpillar type of tree as the algorithm becomes serialized. In this case, there is even a slow-down over the single-core CPU sequential implementation, caused by the parallelism overhead. The vectorial implementation, in turn, is unaffected by the tree topology and outperforms the parallel implementation in both scenarios: on the caterpillar tree by a factor of 5,000, and, more surprisingly, on the star by a factor of 5. This demonstrates that even in the best-case scenario for the parallel implementation, i.e., when the input tree has the maximum average branching factor, our vectorial approach is still faster compared to the state-of-the-art of the load-balancing solution. This demonstrates how important regularity is.

4.4.2 Accuracy

Section 4.2 stated that the vectorial algorithm requires the arithmetic operation involved in the calculations to be associative, in order to provide the same results

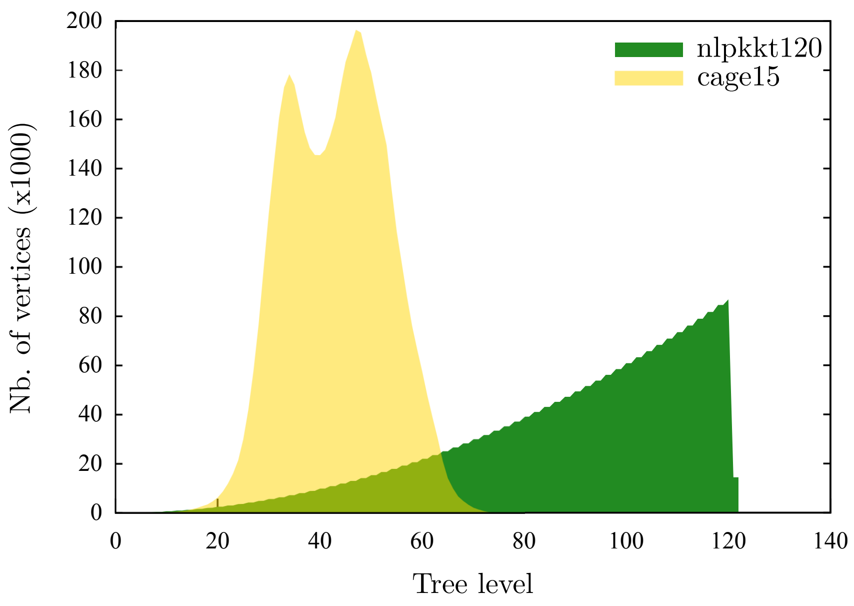


Figure 4.6: Vertex density of the ‘nlpkkt120’ and ‘cage15’ benchmarks.

as the parallel one. As floating-point addition is not associative, one can expect the vectorial algorithm to be less accurate than the parallel algorithm when dealing with floating-point data.

This section studies the numerical quality of the parallel and vectorial +rootfix and +leaffix algorithms for ill-conditioned problems. The conditioning of a problem (independently of the algorithm used to solve it) is given by the *condition number*. In the case of the addition of n floating-point numbers, $p_0, \ldots, p_{n-1}$, the condition number is determined as follows.

$$\text{cond}\left(\sum p_i\right) = \frac{\sum |p_i|}{|\sum p_i|}, \tag{4.7}$$

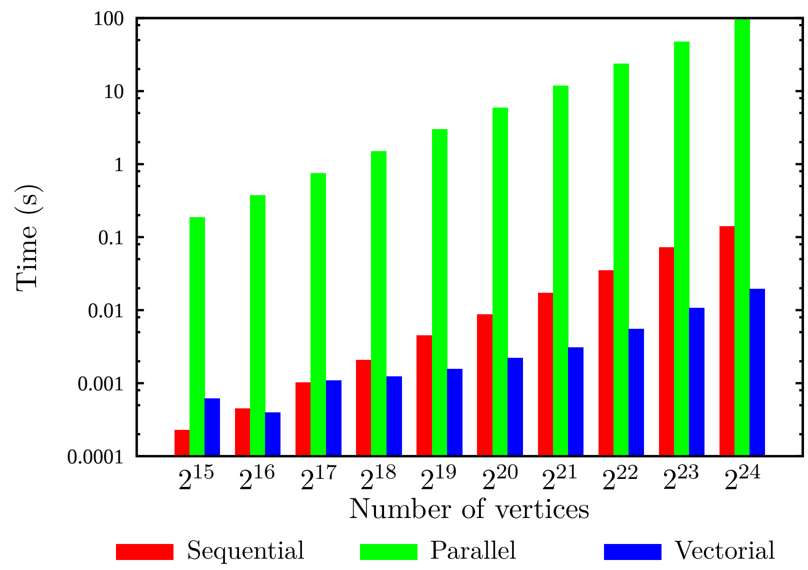
where $|\cdot|$ is the absolute value.

The condition number is proportional to the degree of cancellation in $\sum p_i$ (see [Higham 2002] for a definition of cancellation). Based on this fact, Ogita *et al.* developed an algorithm to generate vectors having any arbitrary condition number, as follows. Half of the vector is first generated randomly within a large exponent range; then, the other half is generated such that some cancellation occurs, depending on the desired condition number [Ogita 2005].

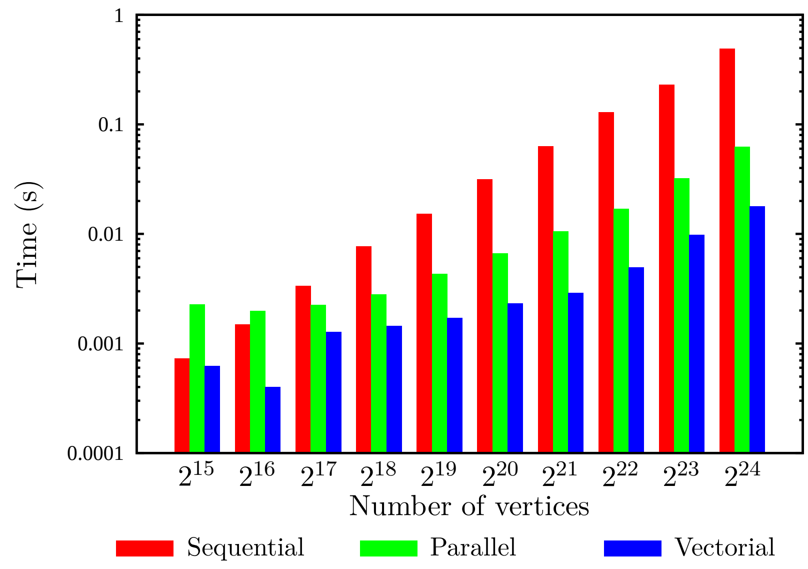
The accuracy of an algorithm is given by the *relative error*. If s is the *exact* result of a computation and $\hat{s}$ is the *approximate* result provided by the algorithm, then the relative error is measured as follows.

$$\text{err}(\hat{s}) = \frac{|s - \hat{s}|}{|s|}. \tag{4.8}$$

In the case of +rootifx and +leaffix the result is a set of n sums, one for each



(a)



(b)

Figure 4.7: Execution time (s) of sequential, parallel and vectorial +rootfix with extreme topologies on GTX 670: (a) Caterpillar topology. (b) Star topology.

vertex in the tree, so there are n relative errors and n condition numbers which can be measured. In order to keep a one-to-one relation, we consider the maximum relative error, and the condition number corresponding to the longest sum to be performed. A suit of 100 random trees of 10,000 vertices each are generated, where each vertex had between 2 and 5 children. For each tree, the longest path from root to leaf is computed, associated with the longest sum in +rootfix; then, vectors of random weights are generated having condition numbers between 10 and 10^{10}, to assign to the vertices on that path. These vectors are generated using the algorithm developed by Ogita *et al.* [Ogita 2005]. Finally, the parallel and vectorial +rootfix and +leaffix implementations are run over the suite of trees, using double precision to compute an accurate approximation and single precision to compute the approximate result. In this way, the numerical behaviour of the algorithms is captured for a large range of condition numbers.

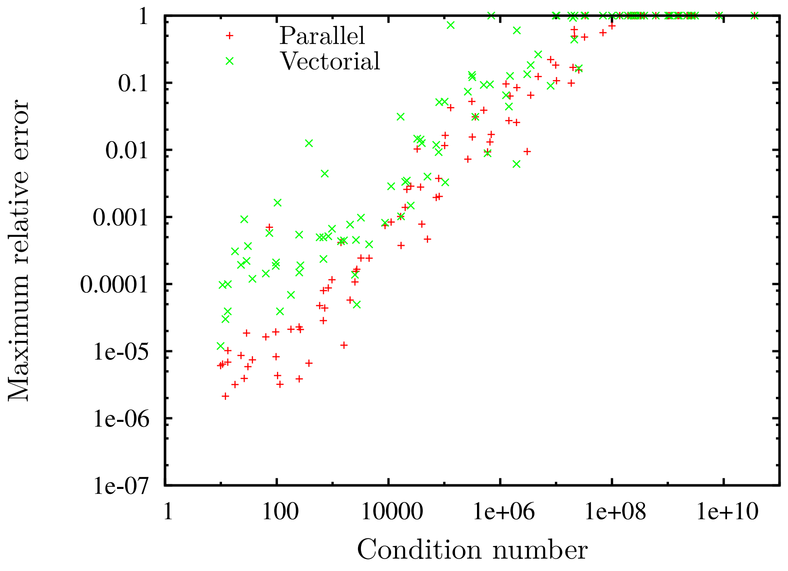
Figure 4.8 shows the maximum relative error as a function of the condition number for the +rootfix and +leaffix parallel and vectorial algorithms. Note that both parallel and vectorial have similar numerical behaviour. The large dispersion of points for condition number lower than 10^4 may come from the difficulties in generating vectors with such characteristics. Regardless, the parallel version seems better than the vectorial one. It seems that in this case, our vectorial implementation is loosing an extra two bits of accuracy in average compared to the parallel one.

In the parallel algorithm, the topology of the tree is affecting the computation scheme and therefore the error. For example, on a caterpillar topology, where every vertex has exactly one child, the parallel algorithm requires a recursive sum of n values with $n - 1$ partial sums. Whereas on a star topology, where the root has $n-1$ children, it requires n sums of two values, i.e., no partial sums. The latter is a much better scenario, as the error of a sum in floating-point is proportional to the magnitudes of the partial sums [Higham 2002]. A large average branching factor is thus beneficial for the parallel approach.

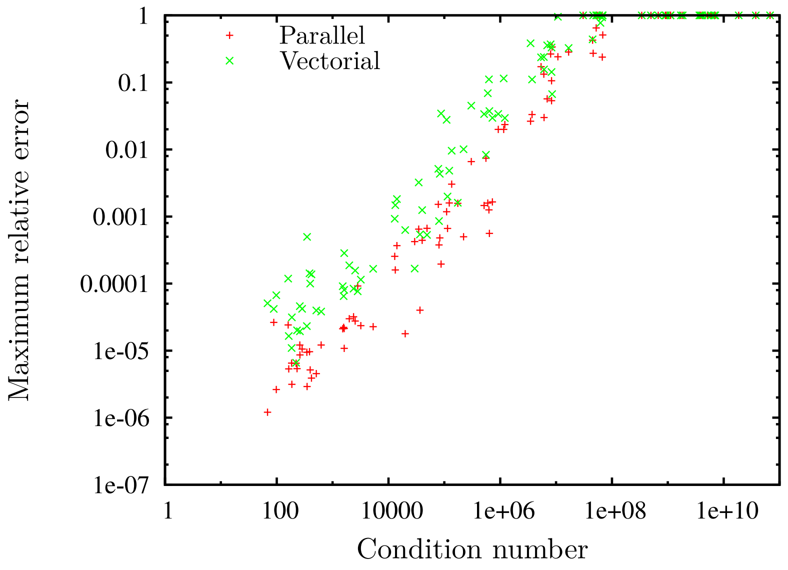
4.4.3 Case study

This section applies the vectorial approach introduced above to the implementation of the BFS algorithm. As seen in Section 3.3.1, the BFS algorithm is used for radial Power Flow (PF) analysis and can be implemented using treefix operations. Using the OpenCL implementation presented in Section 4.3, we developed a vectorial BFS routine and integrated it within the Python-based software Dome [Milano 2013].

The implementation is written in C and provided as an extension to Dome. The vectorial BFS routine can be called from python code and used to analyze any radial network. Dome also provides a serial BFS routine, based on standard tree traversals, which is used in this section for comparison purposes. A series of tests are run to measure the speed-up that the vectorial version can achieve over the serial one. The aim is to determine whether a vectorial BFS implementation can be used in real applications to accelerate computations and reduce the time of



(a)



(b)

Figure 4.8: Related accuracy of parallel and vectorial treefix algorithms: (a) +Rootfix operation. (b) +Leaffix operation.

response.

To this purpose, benchmark networks from 2,500 to 20,000 buses are generated using a routine from Dome for random radial network generation. The routine uses a random number generator and attempts to reproduce typical conditions from distribution networks in the real-world. This kind of networks are characterized by having very few feeders (typically, only one). Also, the grid does not have meshes. The benchmark networks are generated using a base voltage of 15 kV and a base power of 100 MVA. At least half of the buses, selected at random, have a load attached to them. This load is chosen randomly between 0 and 30 kVA, with a power factor between 0.85 and 1.

The random radial network generator distinguishes certain buses as 'roots'. These are the buses where forks occur. The feeder bus is one of these roots, and it has the majority of the forks (as many as the total number of buses divided by 50). For other roots, the number of forks is chosen randomly between 2 and 5. Each fork marks the beginning of a 'row', i.e., a series of buses connected in cascade where no fork takes place. The number of buses in a row is chosen randomly between 5 and 20. The last bus in each row is flagged as a root and originates new forks.

Table 4.3 specifies some of the details above for the generated benchmarks. The number of forks grows linearly with the number of buses. This means that the average branching factor of all four benchmarks is very similar.

Table 4.3: Details of the randomly generated benchmarks.

Number of buses	2,500	5,000	10,000	20,000
Loads	1,942	2,802	6,800	18,583
Lines	2,499	4,999	9,999	19,999
Forks	183	382	752	1,490

The benchmarks are analyzed with both BFS implementations and the execution times are measured. The vectorial BFS execution time is split in order to reflect the time that the program spend in each task. These are: (i) Euler-tour ordering, (ii) OpenCL platform and context setup, (iii) data transfer between host and device and (iv) actual PF calculations. The test environment is described in Section 4.4.1 and summarized in Table 4.2.

Table 4.4 shows the execution time of serial and vectorial BFS on the four benchmarks. Note that, in the latter case, the times are largely dominated by the OpenCL platform and context configuration, which actually causes a slowdown over the serial implementation. As a consequence, to take fully advantage of the vectorial BFS extension, an implementation must be able to re-use a previously configured OpenCL context for the current platform. This is the case, for example, of a real-time simulation which remains online while performing a series of PF analyses. As observed in the last row of Table 4.4, the speedup excluding the OpenCL setup is quite considerable, ranging from 2.7 to about 9 and increasing with the network size.

Table 4.4: Execution time (ms) of vectorial BFS on the GTX 680 GPU and comparison with sequential BFS.

Number of buses	2,500	5,000	10,000	20,000
Sequential time	5.31	20.68	27.58	77.02
Vectorial time	91.83	80.52	91.83	97.44
Euler-tour ordering	0.10	0.21	0.49	1.33
OpenCL setup	89.87	76.63	87.05	89.02
Data transfer	0.32	0.48	1.08	1.53
PF	1.54	3.19	3.21	5.56
Speedup	0.05	0.25	0.30	0.79
Speedup (excluding OpenCL setup)	2.7	5.32	5.76	9.14

It is interesting that the PF is not so relevant in the balance of all tasks. For example, in the 20,000 buses network, the PF calculations only account for about two thirds of the computation time excluding the OpenCL setup. This means that reducing the time spent in either Euler-tour ordering or data transfer can have a relevant impact on performance. The latter cannot be manipulated by software as it depends on specific features of the communication bus between host and device, and the respective memories. The Euler-tour ordering, nonetheless, can be accelerated by using the parallel algorithm provided by [Atallah 1984]. This algorithm has a step complexity of $\mathcal{O}(\log n)$, lower than the $\mathcal{O}(n)$ complexity of the serial algorithm used in the current implementation. This improvement is planned as future work.

4.5 Conclusions

This chapter addresses the implementation of the BFS algorithm on GPUs. The above requires studying the so-called treefix operations, which are the basic building blocks of BFS.

The implementation of treefix operations on GPUs is studied by comparing two approaches. A parallel approach, that minimizes the number of operations and intermediate storage by using load-balancing, and a vectorial approach, that involves a higher amount of operations but exhibits a regular computation pattern. The two approaches are compared in terms of performance and accuracy on GPU architectures. The accuracy tests show that the vectorial approach introduces a higher rounding error, as it performs more floating-point operations to compute the result. On the other hand, the performance tests show that the vectorial approach is always faster, leading to a speedup of 5 to 5,000. Furthermore, its performance is not affected by the tree topology.

Based on the above, the BFS algorithm is implemented using the vectorial approach. The proposed implementation is tested against a standard sequential implementation provided by the software Dome. A speedup of about 2 to 9 is achieved, if the overhead due to OpenCL configuration is not considered.

References

[Atallah 1984] M. Atallah and U. Vishkin. *Finding Euler tours in parallel.* Journal of Computer and System Sciences, vol. 29, no. 3, pp. 330–337, 1984. (Cited on pages 55 and 71.)

[Blelloch 1990] G. E. Blelloch. Vector models for data-parallel computing. MIT Press, Cambridge, MA, USA, 1990. (Cited on pages 53, 54 and 56.)

[Defour 2013] D. Defour and M. Marin. *Regularity Versus Load-balancing on {GPU} for Treefix Computations.* Procedia Computer Science, 2013 International Conference on Computational Science, vol. 18, pp. 309 – 318, 2013. (Cited on page 53.)

[DIM 2012] *10th DIMACS Implementation Challenge.* http://www.cc.gatech.edu/dimacs10/index.shtml, 2012. (Cited on page 62.)

[Glaskowsky 2009] P. N. Glaskowsky. *NVIDIA's Fermi: the first complete GPU computing architecture.* White paper, 2009. (Cited on pages 20, 23 and 61.)

[Higham 2002] N. J. Higham. Accuracy and stability of numerical algorithms. Siam, 2002. (Cited on pages 66, 68, 77, 84 and 139.)

[Iverson 1962] K. E. Iverson. *A programming language.* Proceedings of the May 1-3, 1962, Spring Joint Computer Conference, pp. 345–351. ACM, 1962. (Cited on page 56.)

[Leiserson 1988] C. E. Leiserson and B. M. Maggs. *Communication-efficient parallel algorithms for distributed random-access machines.* Algorithmica, vol. 3, no. 1-4, pp. 53–77, 1988. (Cited on pages 44, 54 and 55.)

[Merrill 2012] D. Merrill, M. Garland and A. Grimshaw. *Scalable GPU graph traversal.* ACM Special Interest Group on Programming Languages Notices, vol. 47, pp. 117–128. ACM, 2012. (Cited on pages 53, 61 and 62.)

[Milano 2013] F. Milano. *A Python-based software tool for power system analysis.* IEEE Power and Energy Society General Meeting, pp. 1–5, 2013. (Cited on pages 7, 10, 68, 98, 122 and 124.)

[Miller 1985] G. L. Miller and J. H. Reif. *Parallel Tree Contraction and Its Application.* 26th Symposium on Foundations of Computer Science, pp. 478–489, Portland, Oregon, 1985. IEEE. (Cited on page 55.)

[Ogita 2005] T. Ogita, S. M. Rump and S. Oishi. *Accurate Sum and Dot Product.* SIAM J. Sci. Comput., vol. 26, no. 6, pp. 1955–1988, 2005. (Cited on pages 66 and 68.)

[Tarjan 1984] R. E. Tarjan and U. Vishkin. *Finding Biconnected Componemts And Computing Tree Functions In Logarithmic Parallel Time.* Proceedings of the 25th Annual Symposium on Foundations of Computer Science, 1984, pp. 12–20, 1984. (Cited on page 55.)

[UFS 2012] *The University of Florida Sparse Matrix Collection.* http://www.cise.ufl.edu/research/sparse/matrices/, 2012. (Cited on page 62.)

Chapter 5

Fuzzy arithmetic on the GPU

Contents

Elements of this chapter were presented at the Euromicro International Conference on Parallel and Distributed Processing (PDP) 2014, receiving the Nvidia Best Paper Award for GPU applications [Defour 2014], and at the IEEE Power & Energy Society General Meeting (PES-GM) 2014, where they have been selected for the Best Paper Session [Marin 2014]. A paper with most recent results has been submitted to ACM Transactions on Mathematical Software and is currently under review. Another paper focusing on Power Flow (PF) analysis under uncertainty is currently in preparation, to be submitted to IEEE Transactions on Power Systems.

5.1 Introduction

This chapter discusses the implementation on GPUs of fuzzy interval arithmetic. Also, it proposes a novel methodology for the assessment of uncertainty in PF analysis, using the fuzzy interval approach.

Interval arithmetic and fuzzy interval arithmetic are tools for the uncertainty assessment used in several fields [Buisson 2003, Bondia 2006, Chen 2006, Cortes-Carmona 2010].

In the classic approach to interval arithmetic, a region is defined to which the value of a variable is expected to belong. This allows modelling uncertainty in numerical data if the information about the bounds is available, e.g., "the speed is between 20 and 50 km/h". However, if the information is more ambiguous and imprecise, e.g., "the speed is about 30 km/h", then the classic approach fails and the fuzzy interval approach can be more appropriate [Siler 2005]. Fuzzy intervals are characterized by a *membership function*, which assigns a degree of membership to each number in the real domain.The α-cut approach to fuzzy interval arithmetic consists of splitting the fuzzy intervals into sets of regular intervals by degree of membership [Dubois 1978]. In this way, only a finite number of levels are considered. Regarding the representation format, several options such as lower-upper, midpoint-radius or even lower-diameter can be used [Moore 1966, Neumaier 1990]. Most fuzzy arithmetic applications rely on interval libraries which implement the lower-upper format [Brönnimann 2006, Revol 2001, Goualard 2006, Anile 1995, Kolarovic 2013, Gagolewski 2012, Trutschnig 2013]. However, a comparative study of different representation formats has not been provided. Is it possible to enhance performance and/or accuracy of fuzzy interval arithmetic by using a specific representation format?

This chapter addresses the above question by studying different formats from both a theoretical and an empirical point of view. As a result of this study, two novel fuzzy interval implementations are proposed. The first is based on the midpoint-radius format that reduces by two the memory requirements and number of operations compared to the traditional lower-upper approach. The second is based on the midpoint-increment format that improves accuracy compared to the previous one, while maintaining or even reducing the number of operations. Next, the midpoint-radius format is used to proposed a novel methodology for PF analysis under uncertainty.

The remainder of the chapter is organized as follows: in Section 5.2, the mathematical framework underlying fuzzy interval arithmetic is presented. Section 5.4 describes the proposed representation formats for symmetric fuzzy intervals; details the implementation of these formats into a fuzzy arithmetic library; and evaluates performance of the library on the GPU. The proposed method for the assessment of uncertainty in PF analysis is presented in Section 5.5. Finally, Section 5.6 draws conclusions and outlines future work.

5.2 Mathematical background

In order to support the discussion of following sections, the basic concepts of interval arithmetic and α-cut fuzzy arithmetic are outlined next.

5.2.1 Interval arithmetic

Intervals are defined as convex sets of real numbers. It is assumed that, given a real number and an interval, the real number either belongs to the interval or not.

In this way, each element in the interval is considered equally possible (and each element outside the interval, equally impossible).

Interval representation

Intervals can be represented in several formats, e.g., through lower and upper bounds, midpoint and radius, or lower bound and diameter. This chapter considers the first two alternatives, as follows.

Definition 5.2.1 (Lower-upper representation)**.** Let $[x]$ be an interval defined by

$$[x] = \{x \in \mathbb{R} \colon \underline{x} \leq x \leq \overline{x}\},$$

where $\underline{x}, \overline{x} \in \mathbb{R}, \underline{x} \leq \overline{x}$. Then $[x]$ is noted $[\underline{x}, \overline{x}]$.

Definition 5.2.2 (Midpoint-radius representation)**.** Let $[x]$ be an interval defined by

$$[x] = \{x \in \mathbb{R} \colon |x - \check{x}| \leq \rho_x\},$$

where $\check{x}, \rho_x \in \mathbb{R}, \rho_x \geq 0$. Then $[x]$ is noted $\langle \check{x}, \rho_x \rangle$.

Interval operations and rounding

This subsection defines interval operations based on the definitions given above. The basic property of *inclusion isotonocity* is assumed, as follows.

Definition 5.2.3 (Inclusion isotonicity)**.** Let $\circ$ be a basic arithmetic operation, i.e., $\circ \in \{+, -, \cdot, /\}$, $[x]$ and $[y]$ intervals. If

$$x \circ y \subseteq [x] \circ [y]\,, \ \forall\, x \in [x]\,, \ \forall\, y \in [y]\,,$$

then $\circ$ is said to be inclusion isotone.

Inclusion isotonicity ensures that no possible values are "left behind" when performing interval operations. To respect this property, interval arithmetic implementations have to cope with floating-point rounding errors.

The IEEE-754 Standard for floating-point computation establishes that the following three rounding attributes must be available: rounding upwards (towards infinity), rounding downwards (towards minus infinity), and rounding to nearest [461 2008]. The IEEE-754 Standard also defines the relative rounding error and the smallest representable (unnormalized) floating-point number, the latter being associated with the underflow error [Rump 1999, Higham 2002].

In the lower-upper representation, isotonicity implies rounding downwards when computing the lower bounds, and upwards when computing the upper bounds [Dubois 1978]. The interval operations for the lower-upper representation are defined below.

Definition 5.2.4 (Lower-upper interval operations)**.** Let $[x] = [\underline{x}, \overline{x}]$ and $[y] = [\underline{y}, \overline{y}]$. Let $\bigtriangledown(\cdot)$ and $\bigtriangleup(\cdot)$ be the rounding attributes towards minus infinity and plus infinity, respectively. Then

$$\begin{aligned}
[x] + [y] &= [\bigtriangledown(\underline{x} + \underline{y}), \bigtriangleup(\overline{x} + \overline{y})], \\
[x] - [y] &= [\bigtriangledown(\underline{x} - \overline{y}), \bigtriangleup(\overline{x} - \underline{y})], \\
[x] \cdot [y] &= [\bigtriangledown(\min(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y})), \bigtriangleup(\max(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}))], \\
\frac{1}{[y]} &= \left[\bigtriangledown\left(\frac{1}{\overline{y}}\right), \bigtriangleup\left(\frac{1}{\underline{y}}\right)\right], \quad 0 \notin [y] \,.
\end{aligned}$$

In the case of the midpoint-radius representation, in addition to adequate rounding, isotonocity requires adding to the radius the error due to midpoint rounding [Rump 1999]. The interval operations for the midpoint-radius representation are defined below.

Definition 5.2.5 (Midpoint-radius interval operations)**.** Let $[x] = \langle \check{x}, \rho_x \rangle$ and $[y] = \langle \check{y}, \rho_y \rangle$. Let $\square(\cdot)$ and $\bigtriangleup(\cdot)$ be the rounding attributes to nearest and towards plus infinity, respectively. Let ε be the relative rounding error, $\varepsilon' = \frac{1}{2}\varepsilon$ and η be the smallest representable (unnormalized) floating-point positive number. Then

$$[x] \pm [y] = \langle z, \bigtriangleup(\varepsilon'|z| + \rho_x + \rho_y) \rangle, \; z = \square(\check{x} \pm \check{y}), \tag{5.2a}$$

$$[x] \cdot [y] = \langle z, \bigtriangleup(\eta + \varepsilon'|z| + (|\check{x}| + \rho_x)\rho_y + |\check{y}|\rho_x) \rangle, \; z = \square(\check{x}\check{y}), \tag{5.2b}$$

$$\frac{1}{[y]} = \left\langle z, \bigtriangleup\left(\eta + \varepsilon'|z| + \frac{-\rho_y}{|\check{y}|(\rho_y - |\check{y}|)}\right)\right\rangle, \; z = \square\left(\frac{1}{\check{y}}\right), \; 0 \notin [y] \,. \tag{5.2c}$$

It is worth noting that both equations (5.2b) and (5.2c) introduce a bounded overestimation over the lower-upper format definition. In the case of the multiplication, the overestimation is bounded by a factor of 1.5 [Rump 1999].

The impact of rounding on performance depends on the computing architecture. For example, for most CPU architectures, the rounding attribute is a processor ‘mode’, and thus changes in rounding mode cause the entire instruction pipeline to be flushed.

The dependency problem

The so-called dependency problem causes overestimation in interval computations where the same variable occurs more than once. Since all the instances of the same variable are taken independently by the rules of interval arithmetic, the radius of the result might be expanded to unrealistic values. For example, consider the function, $f(x) = x^2 - 1$, where $x \in [-1, 1]$. Then, $f(x) \in [-1, 0]$. However, using interval arithmetic, $f([-1, 1]) = [-1, 1] \cdot [-1, 1] - 1 = [-2, 0]$.

This can become a major issue to the application of interval arithmetic to real-world problems, especially to non-linear ones. Accordingly, specific measures have to be taken to preserve the relevance of the results.

5.2.2 Fuzzy interval arithmetic

Fuzzy intervals are defined as fuzzy sets of real numbers. Given a number and a fuzzy interval, the number belongs to the fuzzy interval with a certain degree of membership. Elements with a higher degree are considered more possible than elements with lower degrees. This is the main feature that distinguish fuzzy intervals from regular intervals.

Fuzzy interval representation

Fuzzy intervals are characterized by a membership function. This chapter considers continuous, strictly monotonic membership functions. The definition of membership function is as follows.

Definition 5.2.6 (Membership function)**.** Let $\mu(\cdot) : \mathbb{R} \to [0,1]$ be a continuous function. Let $\underline{x}, \check{x}, \overline{x} \in \mathbb{R}$, such that $\underline{x} \leq \check{x} \leq \overline{x}$, and:

$$\begin{aligned} \mu(\check{x}) &= 1, \\ \mu(x_1) < \mu(x_2),\ \forall\, x_1, x_2 &\in [\underline{x}, \check{x}],\ x_1 < x_2, \\ \mu(x_1) > \mu(x_2),\ \forall\, x_1, x_2 &\in [\check{x}, \overline{x}],\ x_1 < x_2, \\ \mu(x) = 0,\ \forall\, x &\notin [\underline{x}, \overline{x}]. \end{aligned}$$

Then, $\mu(\cdot)$ is called a membership function.

In other words, the membership function is strictly increasing up until reaching the point $\mu(\check{x})$, and then strictly decreasing. The formal definition of fuzzy interval is given below.

Definition 5.2.7 (Fuzzy interval)**.** Let $\mu(\cdot)$ be a membership function. Then, the set

$$[\tilde{x}] = \{(x, \mu(x)) \colon x \in \mathbb{R}\},$$

is called a fuzzy interval.

According to Definition 5.2.7, a fuzzy interval is a set of ordered pairs, each composed of a real number and a degree of membership, the latter given by a membership function. Figure 5.1 shows an example of fuzzy interval.

Fuzzy interval operations: the α-cut approach

The α-cut approach to fuzzy arithmetic consists of splitting the fuzzy intervals into sets of regular intervals characterized by given membership degree [Dubois 1978]. These intervals are called α-cuts. In this chapter, only a finite number of uncertainty levels is considered, N. Each uncertainty level, or α-level, is assigned a membership degree, $\alpha_i \in [0,1]$, such that $\alpha_1 > \ldots > \alpha_N$. For each level i, the corresponding

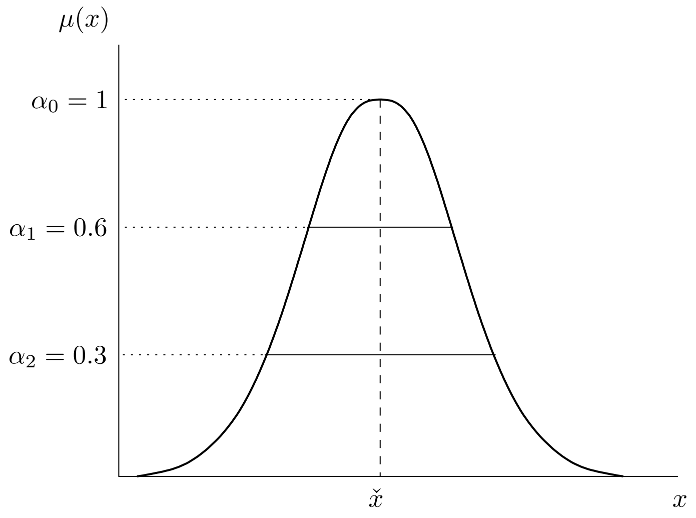


Figure 5.1: Fuzzy interval, membership function and α-cuts.

α-cut contains all the real numbers having membership degree of at least α_i. The formal definition of α-cut is the following:

Definition 5.2.8 (α-cut)**.** Let $[\tilde{x}]$ be a fuzzy interval with membership function $\mu(\cdot)$. Let $N \in \mathbb{N}$, $i \in \{1, \ldots, N\}$, and $\alpha_i \in [0, 1]$. Then

$$[x_i] = \{x \in \mathbb{R} \colon \mu(x) \geq \alpha_i\},$$

is called an α-cut (of level i) of the fuzzy interval $[\tilde{x}]$.

The assumptions made on the membership function (see Definition 5.2.6) allow the α-cuts to become well-defined intervals, as seen in Fig. 5.1. Once the α-cuts are obtained for each operand, the α-cut concept defined below may be applied.

Definition 5.2.9 (α-cut concept)**.** Let $\circ \in \{+, -, \cdot, /\}$, and $[\tilde{x}], [\tilde{y}], [\tilde{z}]$, fuzzy intervals, such that $[\tilde{z}] = [\tilde{x}] \circ [\tilde{y}]$. Let $[x_i], [y_i], [z_i]$, be the α-cuts of level i of each. Then

$$[z_i] = [x_i] \circ [y_i] \, .$$

The above definition states that to perform a fuzzy operation, the α-cuts of the result are to be computed via the corresponding interval operation between α-cuts of the operands.

5.3 Proposed fuzzy arithmetic implementation

This section presents two novel approaches to implement fuzzy interval arithmetic based on the midpoint-radius format.

5.3.1 Midpoint-radius format

The shape of the membership function is a crucial aspect in fuzzy modelling [Boukezzoula 2014]. This section considers a membership function symmetric with respect to a vertical axis, which is of particular relevance in several fuzzy arithmetic applications [Chang 1995, Mendel 2006]. For symmetric membership functions, all α-cuts are centered on the kernel, which leads to relevant properties that are exploited by the proposed implementation. The definition of a symmetric fuzzy interval is given below.

Definition 5.3.1 (Symmetric fuzzy interval)**.** Let $[\tilde{x}]$ be a fuzzy interval with membership function $\mu(\cdot)$ and kernel $\check{x}$. If $\mu(\cdot)$ is symmetric around $\check{x}$, i.e.,

$$\mu(\check{x} - x) = \mu(\check{x} + x), \ \forall\, x \in \mathbb{R},$$

then $[\tilde{x}]$ is called a symmetric fuzzy interval.

Although symmetry is immaterial for the lower-upper interval representation, it is relevant for the midpoint-radius representation, as it is stated by the following proposition.

Proposition 5.3.1. *Let $[\tilde{x}]$ be a symmetric fuzzy interval with kernel $\check{x}$. Let $N \in \mathbb{N}$, $i \in \{1, \ldots, N\}$, and $[x_i] = \langle \check{x}_i, \rho_{x,i} \rangle$, an α-cut. Then,*

$$[\tilde{x}] \textit{ is symmetric} \iff \check{x}_i = \check{x}, \ \forall\, i \in \{1, \ldots, N\}.$$

Proof. See Appendix B. □

The above result can be graphically seen in Fig. 5.1, where the fuzzy interval is symmetric. Note how all the α-cuts are centered on the kernel. The property of symmetry is also preserved by basic fuzzy arithmetic operations, as stated by the following theorem.

Theorem 5.3.1. *Let $[\tilde{x}]$ and $[\tilde{y}]$ be fuzzy intervals, $\circ \in \{+, -, \cdot, /\}$. If $[\tilde{x}]$ and $[\tilde{y}]$ are symmetric, then $[\tilde{z}] = [\tilde{x}] \circ [\tilde{y}]$ is also symmetric.*

Proof. See Appendix B. □

Proposition 5.3.1 and Theorem 5.3.1 are applied next to propose an efficient representation format of symmetric fuzzy intervals. Fuzzy intervals are stored with two elements: (i) the kernel (or midpoint, common to all α-cuts) and (ii) a set of radii (one per α-cut). Fuzzy operations are also solved in two steps: (i) the computation of the kernel, and (ii) the computation of the individual radius of each α-cut. Algorithm 5.1 illustrates how to perform fuzzy multiplications. The parameters ε and η correspond to the relative rounding error and the smallest unnormalized floating-point number, respectively (see Definition 5.2.5). As shown in Section 5.4.1, this format allows us to save both bandwidth and execution time when performing fuzzy operations on symmetric fuzzy intervals.

Algorithm 5.1 Symmetric fuzzy multiplication in the midpoint-radius format.

```
Require: Symmetric fuzzy operands $[\tilde{x}]$ and $[\tilde{y}]$.
Ensure: Symmetric fuzzy result $[\tilde{z}] = [\tilde{x}] \cdot [\tilde{y}]$.
1: kernel: $\check{z} = \square(\check{x}\check{y})$
2: for $i$ in $1, \ldots, N$ do
3:     radius: $\rho_{z,i} = \triangle(\eta + \frac{1}{2}\varepsilon|\check{z}| + (|\check{x}| + \rho_{x,i})\rho_{y,i} + |\check{y}|\rho_{x,i})$
```

5.3.2 Midpoint-increment format

Another relevant property of fuzzy intervals is that every α-cut is fully contained within any lower level α-cut [Fortin 2008]. This is an immediate consequence of the membership function being monotonic around the kernel.

Proposition 5.3.2. *Let $[\tilde{x}]$ be a fuzzy interval, $N \in \mathbb{N}$, $i, j \in \{1, \ldots, N\}$, and $[x_i], [x_j]$, α-cuts. Then*

$$\alpha_i > \alpha_j \implies [x_i] \subset [x_j] .$$

Proof. See Appendix B. □

The result above is applied next to propose a representation format of symmetric fuzzy intervals. The fuzzy intervals are stored with two elements: (i) the kernel (or midpoint, common to all α-cuts), and (ii) a set of radius increments (one per α-cut). This format differs from the previous one (Section 5.3.1) as only the increments are stored here. The increments correspond to the difference between the radii of two consecutive α-cuts, as defined below.

Definition 5.3.2 (Radius increments)**.** Let $[\tilde{x}]$ be a fuzzy interval, $N \in \mathbb{N}$ and $i \in \{1, \ldots, N\}$. Let $\{\alpha_1, \ldots, \alpha_N\} \in [0, 1]$, such that $\alpha_1 > \ldots > \alpha_N$. Let $[x_1], \ldots, [x_N]$, be α-cuts and $\rho_{x,1}, \ldots, \rho_{x,N}$, their respective radii. The numbers, $\delta_{x,1}, \ldots, \delta_{x,N}$, defined as:

$$\delta_{x,i} = \begin{cases} \rho_{x,i} - \rho_{x,i-1} & \text{if } 2 \leq i \leq N, \\ \rho_{x,1} & \text{if } i = 1, \end{cases}$$

are called radius increments.

From the above definition, it is transparent that any radius of level i can be computed as a sum of increments:

$$\rho_{x,i} = \sum_{k=1}^{i} \delta_{x,k}.$$

The purpose of this format is to increase the accuracy of fuzzy computations. As the increments are, by definition, smaller than the radii, the rounding errors originated from the use of floating-point arithmetic are also smaller. This concept is further developed in Section 5.3.3.

Algorithms 5.2, 5.3 and 5.4 show how to compute the addition, subtraction, multiplication and inversion on this format. Note that the number of floating-point operations needed by this format is not increased compared to midpoint-radius. In fact, addition and subtraction require one fewer operation per α-cut than the midpoint-radius. Multiplication and inversion require the same amount of operations per α-cut, provided that the α-cuts are treated sequentially (such that certain computations associated to a given α-level can be re-used in the following level).

Algorithm 5.2 Symmetric fuzzy addition and subtraction in the midpoint-increment format.

```
Require: Symmetric fuzzy operands $[\tilde{x}]$ and $[\tilde{y}]$.
Ensure: Symmetric fuzzy result $[\tilde{z}] = [\tilde{x}] \pm [\tilde{y}]$.
1: kernel: $\check{z} = \square(\check{x} \pm \check{y})$
2: first increment: $\delta_{z,1} = \triangle(\frac{1}{2}\varepsilon|\check{z}| + \delta_{x,1} + \delta_{y,1})$
3: for $i$ in $2, \ldots, N$ do
4:     increment: $\delta_{z,i} = \triangle(\delta_{x,i} + \delta_{y,i})$
```

Algorithm 5.3 Symmetric fuzzy multiplication in the midpoint-increment format.

```
Require: Symmetric fuzzy operands $[\tilde{x}]$ and $[\tilde{y}]$.
Ensure: Symmetric fuzzy result $[\tilde{z}] = [\tilde{x}] \cdot [\tilde{y}]$.
1: kernel: $\check{z} = \square(\check{x}\check{y})$
2: accumulator in $[\tilde{x}]$: $t_{x,1} = \triangle(|\check{x}| + \delta_{x,i})$
3: accumulator in $[\tilde{y}]$: $t_{y,1} = \triangle(|\check{y}|)$
4: first inc.: $\delta_{z,1} = \triangle(\eta + \frac{1}{2}\varepsilon|\check{z}| + t_{x,1}\delta_{y,1} + t_{y,1}\delta_{x,1})$
5: for $i$ in $2, \ldots, N$ do
6:     accumulator in $[\tilde{x}]$: $t_{x,i} = \triangle(t_{x,i-1} + \delta_{x,i})$
7:     accumulator in $[\tilde{y}]$: $t_{y,i} = \triangle(t_{y,i-1} + \delta_{y,i})$
8:     increment: $\delta_{z,i} = \triangle(\eta + \frac{1}{2}\varepsilon|\check{z}| + t_{x,i}\delta_{y,i} + t_{y,i}\delta_{x,i})$
```

Algorithm 5.4 Symmetric fuzzy inversion in the midpoint-increment format.

```
Require: Symmetric fuzzy operand $[\tilde{y}]$.
Ensure: Symmetric fuzzy result $[\tilde{z}] = 1/[\tilde{y}]$.
1: kernel: $\check{z} = \square(1/\check{y})$
2: accumulator: $t_{y,1} = |\check{y}|$
3: first increment: $\delta_{z,1} = \triangle(\eta + \frac{1}{2}\varepsilon|\check{z}| + (-\delta_{y,1})/(t_{y,1}(\delta_{y,1} - t_{y,1}))$
4: for $i$ in $2, \ldots, N$ do
5:     accumulator: $t_{y,i} = \triangle(t_{y,i-1} - \delta_{y,i-1})$
6:     increment: $\delta_{z,i} = \triangle(\eta + \frac{1}{2}\varepsilon|\check{z}| + (-\delta_{y,i})/(t_{y,i}(\delta_{y,i} - t_{y,i}))$
```

In Algorithms 5.2, 5.3 and 5.4, the kernel's rounding error is added only to the first increment, so one operation per α-cut is saved. In the case of the multiplication and inversion, i has to be increased sequentially. Variables $t_{(\cdot)}$ are introduced to serve as accumulators. In the multiplication, these accumulators store the maximum (absolute) value of two α-cuts, one in each operand, one level apart from each other. Only one operation is needed to update the accumulator at each α-level. Interestingly, these accumulators allow for a quite transparent calculation of the increment. The same amount of operations are needed as in computing the full radius using the midpoint-radius format.

5.3.3 Error analysis

This section compares the accuracy of radius computations in both proposed formats. The main operation involved in this kind of computation is the rounded floating-point addition. The following result, which concerns the error associated with floating-point summation algorithms, is useful for the analysis that is discussed later in this section.

Theorem 5.3.2 (Bound for the absolute error in summation algorithms [Higham 2002]). *Let $x_1, \ldots, x_n \in \mathbb{R}$ and $S_n = \sum_{i=1}^{n} x_i$. Let $\hat{S}_n$ be an approximation to S_n computed by a summation algorithm, which performs exactly $n-1$ floating-point additions. Let $\hat{T}_1, \ldots, \hat{T}_{n-1}$, be the $n-1$ partial sums computed by such algorithm. Then,*

$$E_n := \left|S_n - \hat{S}_n\right| \leq \varepsilon \sum_{i=1}^{n-1} \left|\hat{T}_i\right|,$$

where ε is the relative rounding error.

Proof. See [Higham 2002]. □

Hence, the absolute error introduced by a floating-point summation algorithm is no greater than the relative rounding error, multiplied by the sum of magnitudes of all the intermediate sums.

Theorem A.2.1 is used to prove that the midpoint-increment format improves the accuracy of basic fuzzy computations, compared to midpoint-radius. The result presented below is a direct consequence of the increments being smaller in magnitude than the radii.

Theorem 5.3.3. *Let $\circ \in \{+, -, \cdot, /\}$ and $[\tilde{x}], [\tilde{y}], [\tilde{z}]$, be fuzzy intervals, such that $[\tilde{z}] = [\tilde{x}] \circ [\tilde{y}]$. Let $[z_i]$ be the α-cut of level i of $[\tilde{z}]$, and $\rho_{z,i}$, the radius of $[z_i]$. Let $E^{(rad)}(\cdot)$ and $E^{(inc)}(\cdot)$ return the maximum absolute error in the midpoint-radius and midpoint-increment formats, respectively. Then,*

$$E^{(inc)}(\rho_{z,i}) < E^{(rad)}(\rho_{z,i}), \quad \forall\, i \geq 2.$$

Proof. See Appendix B. □

Theorem 5.3.3 allows concluding that the gain in accuracy is driven ultimately by the fact that the radius at any α-level is always greater than the one at the previous level. This, in turn, is a consequence of the monotonicity of the membership function. In the midpoint-increment format, the smallest radius is computed first, and then the others by simply adding the increments. In other words, large values are broken down into smaller ones, prior to perform fuzzy computations. As these computations are composed mainly of floating-point additions, this strategy reduces rounding error in absolute value.

It can be shown that the result holds for any summation algorithm chosen to compute the increments and the radii, as long as it is the same for both. However, this proof is omitted for sake of brevity. The accuracy gain can also be quantified. It depends on several factors, e.g., the considered summation algorithm (i.e., the order of the operations); the ratio of increment to radius at different α-levels; the "precision" of the intervals (i.e., the ratio of midpoint to radius); and the relative rounding error ε.

5.4 Implementation on the GPU

We implemented the proposed midpoint-radius and the traditional lower-upper format in CUDA and C++. The library of fuzzy arithmetic operations is available at `https://github.com/mmarin-upvd/fuzzy-gpu/`. The implementation of the midpoint-increment format is planned as future work.

The library is callable from either CPU and GPU code, effectively adapting itself to the underlying architecture. The code is composed of a series of wrappers. At the user-end, one finds the two fuzzy classes. The lower-upper fuzzy class relies on a lower-upper interval class, which serves the purpose of holding the α-cuts. The lower-upper interval class, in turn, relies on a rounded arithmetic class for operating between the α-cuts, according to the rules of interval arithmetic. The midpoint-radius fuzzy class is linked directly to the rounded-arithmetic class, as the α-cuts are managed as a whole and not individually, making use of the result from Theorem 5.3.1 in the previous section. The rounded arithmetic class is a wrapper of C and CUDA compiler *intrinsics*, directly associated to specific machine instructions.

The fuzzy template classes are parametrized by the number of α-cuts N and the data type T. In the lower-upper class, N and T specify the size and type of the array of intervals which corresponds to the different α-cuts. In the midpoint-radius class, N and T are the size and type of the array of radii, and T is also the type of the scalar midpoint common to all the α-cuts.

Basic arithmetic operators are overloaded to work on both fuzzy classes. The loop that sweeps over the set of α-cuts (see Algorithms 5.1, 5.2, 5.3 and 5.4) is not spread among multiple threads in GPU code, since most applications in real life do not involve more than three to four degrees of uncertainty [Buisson 2003, Bondia 2006, Chen 2006, Cortes-Carmona 2010]. This number is way too low to conveniently exploit thread level parallelism (TLP). However, TLP may be exploited in vector operations involving fuzzy intervals, through kernels that assigns each element of a fuzzy array to a different thread.

Performance of basic operations over complex data type such as fuzzy intervals are impacted by numerous factors. Following paragraphs discuss and compare different representation formats regarding number of instructions, memory requirements and Instruction-Level Parallelism (ILP).

Number of instructions

Table 5.1 shows the number of instructions, such as basic operations with different rounding, minimum, maximum and absolute value, required for the addition, multiplication and inversion of different data types, including fuzzy intervals. In fuzzy data types, N represents the number of α-cuts. Note that the addition of fuzzy intervals in the lower-upper format require only 3 operations less than in midpoint-radius. However, the multiplication requires 2.8 times more operations per α-cut. In the case of a full division, which corresponds to one inversion plus one multiplication, the lower-upper fuzzy requires $16N$ operations, whereas the midpoint-radius requires $11 + 10N$, i.e., lower-upper requires 1.6 more operations per α-cut. Therefore, just by comparing the number of instructions, it can be anticipated that the midpoint-radius format shall bring a speed-up over lower-upper.

Table 5.1: Number of instructions per operation, for different data types.

Data type	Addition	Multiplication	Inversion
Scalar	1	1	1
Lower-upper interval	2	14	2
Midpoint-radius interval	5	11	10
Lower-upper fuzzy	$2N$	$14N$	$2N$
Midpoint-radius fuzzy	$3+2N$	$6+5N$	$5+5N$
Midpoint-increment fuzzy	$4+N$	$6+5N$	$5+5N$

N: number of α-cuts.

Memory usage

Table 5.2 shows the memory space required to store different data types. The units have been normalized to the size of one scalar. In the fuzzy case, once again, N represents the number of α-cuts. The lower-upper fuzzy requires double the space than the midpoint-radius and midpoint-increment fuzzy. As the bandwidth requirement of both the latter is half the one of the former, an application that needs to access memory at a high rate will definitely benefit from that property.

Note that the memory footprint of all these representation formats can be further reduced by lowering the type used to store the radius or the increment, in the case where the α-cuts are not too wide. The impact is solely on the width of the α-cuts which can be considered and not on the dynamic of numbers. For example, the midpoint can be stored in double precision and the radius in single precision, without impacting too much on the accuracy. This again is an advantage for the proposed formats, as the lower bound is an array of scalars, whereas the midpoint is only one scalar.

Table 5.2: Memory requirements of different data types.

Data type	Memory usage
Scalar	1
Lower-upper interval	2
Midpoint-radius interval	2
Lower-upper fuzzy	$2N$
Midpoint-radius fuzzy	$1 + N$
Midpoint-increment fuzzy	$1 + N$

N: number of α-cuts.

Instruction-Level Parallelism

Ideal ILP, defined as the ratio of the number of instructions to the number of levels in the dependency tree [Goossens 2013], is a good measure of how a given sequence of instructions could be handled on today's but also future architectures. The higher the ideal ILP is, the higher the amount of instructions that can be pipelined during the execution of a given application. However, a bigger ideal ILP requires a larger amount of hardware resources.

Table 5.3 shows the ideal ILP in the addition, multiplication and inversion of different data types. The number of α-cuts in lower-upper and midpoint-radius fuzzy data types does not affect the size of the dependency tree, as each α-cut is processed independently from all others. The same is valid for the addition in the midpoint-increment format. However it does not hold in the multiplication and inversion, because of dependencies between computations belonging to different α-cuts. Accordingly, the size of the dependency tree depends in this case on N, the number of α-levels.

In general, lower-upper fuzzy exhibits more ideal ILP than midpoint-radius and midpoint-increment in either the addition, the multiplication and the inversion. However, ILP is exploited differently depending on the GPU generation as well as the precision in use (single or double). For example, GPUs with CUDA capability 3.0 can schedule up to 2 independent instructions for a given warp scheduler. The impact of ILP on example applications is also studied in the next section.

5.4.1 Tests and results

This section evaluates the performance of the proposed fuzzy arithmetic library on GPUs. According to the CUDA Programming Guide, GPU performance is mainly

Table 5.3: ILP per arithmetical operation, for different data types.

Data type	Addition	Multiplication	Inversion
Scalar	1	1	1
Lower-upper interval	2	$\frac{14}{3}$	2
Midpoint-radius interval	$\frac{5}{4}$	$\frac{11}{5}$	$\frac{9}{2}$
Lower-upper fuzzy	$2N$	$\frac{11}{5}N$	$2N$
Midpoint-radius fuzzy	$\frac{3}{4}+\frac{1}{2}N$	$\frac{6}{5}+N$	$1+\frac{4}{5}N$
Midpoint-increment fuzzy	$1+\frac{1}{4}N$	$\frac{6+5N}{3+N}$	$\frac{5+5N}{3+N}$

N: number of α-cuts.

driven by the ratio of computing intensity to memory usage [Cuda 2012]. With this regard, two types of applications are considered: compute-bound application, where arithmetic instructions are dominant over memory accesses; and memory-bound application, where the opposite is true. For an example of the former, see [Beliakov 2015]. For an example of the latter, see [Davis 2012]. The performance of the implemented fuzzy arithmetic library is tested in these two extreme cases, obtaining a general idea of its behaviour.

The test environment is summarized in Table 5.4. The tests programs are compiled using GCC 4.8.2 and CUDA 6.0. The different tests scenarios are generated by varying significant parameters in the application over an appropriate range. The number of α-cuts varies from 1 to 24; the lower-upper and midpoint-radius fuzzy representation formats are alternatively selected; single and double precision floating-point numbers are chosen as the underlying data-type.

Table 5.4: Test environment

Device	Compute capability	Number of cores
Xeon X560	N/A	12 (1 used)
GeForce GTX 480	2.0	480
GeForce GTX 680	2.0	1,536

Compute-bound test: AXPY kernel

Figure 5.2 shows a kernel that computes the n-th element of the series $x_{k+1} = ax_k + b$, where all the values are fuzzy intervals. The threads read the value from an input array, perform the computation through an iterative loop and write the result back to an output array. As the number of iterations, n, is incremented, the number of floating-point instructions is incremented as well. Whereas the number

```
#include "fuzzy_lib.h"

template<class T, int N>
__global__ void axpy(int n,
fuzzy<T, N> * input,
fuzzy<T, N> b,
fuzzy<T, N> * output)
{
  int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
  fuzzy<T, N> a, c = 0;
  a = input[thread_id];
  for (int i = 0; i < n; i++)
  c = a * c + b;
  output[thread_id] = c;
}
```

Figure 5.2: AXPY kernel, compute-bound test.

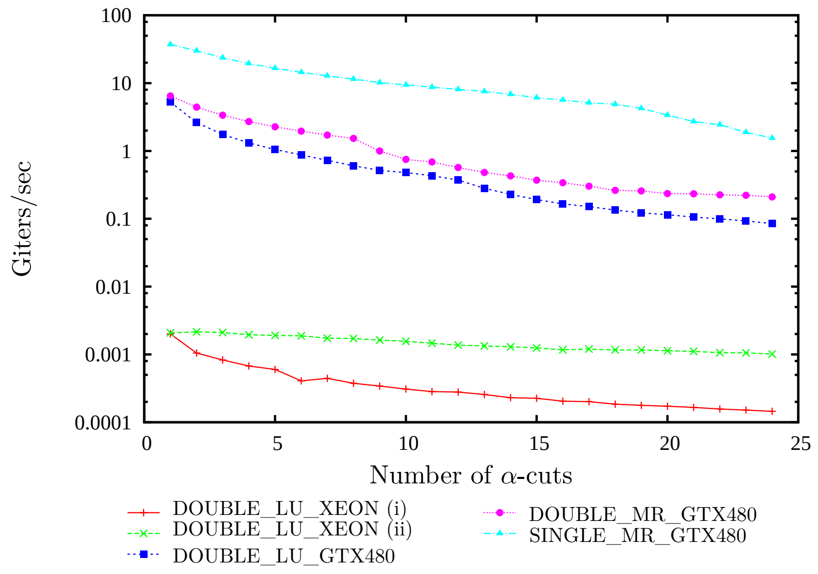
of accesses to global memory remains constant, i.e., one load for the starting value and one store for the final result. In this way, the ratio of floating-point instructions to global memory accesses can be arbitrarily incremented.

The kernel's execution time is measured along with other relevant performance indicators, for the different scenarios generated. The kernel arguments are tuned for different runs in order to achieve maximum occupancy on the GPU architecture. The results are plotted in Figs. 5.3 and 5.4. Figure 5.3(a) shows performance achieved by different configurations on various CPU and GPU architectures, in terms of number of iterations per second as a function of the number of α-cuts. The Xeon CPU is running a single threaded version of the kernel using (i) the proposed library, and (ii) the Java library in [Kolarovic 2013]. The latter is a straightforward implementation of the α-cut approach, however, it does not use correct rounding to certify the results. Therefore, the computational burden is a little lower than in the proposed library.

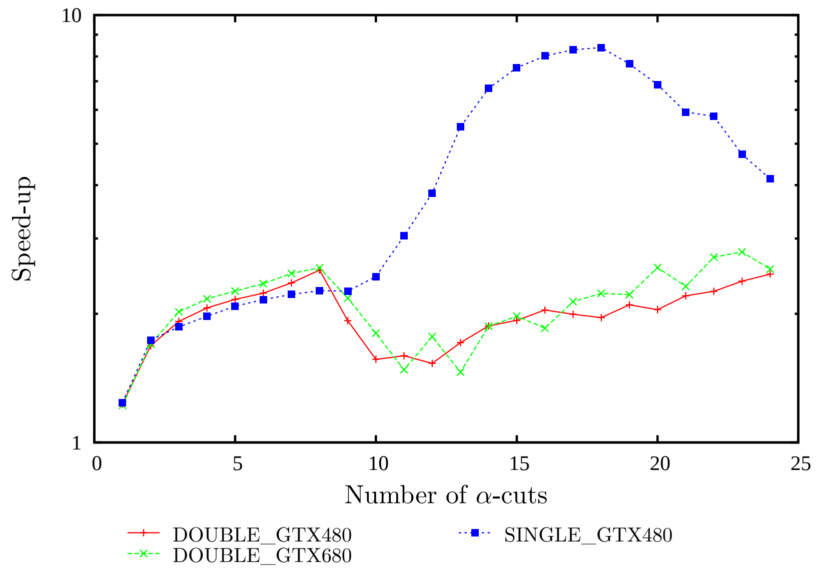
The differences in performance come from heterogeneous sources. First, there is a pure architectural gain of about three orders of magnitude on passing from the CPU to the GPU. Second, within the GPU, there is an algorithmic gain on choosing midpoint-radius over lower-upper, although this remains in the same order of magnitude. Finally, there is a gain of one order of magnitude by considering single versus double precision format in fuzzy calculations.

The most interesting of these gains is the one driven by the algorithm, as it reveals the differences between the lower-upper and the midpoint-radius representation formats. Figure 5.3(b) shows the speed-up achieved by the midpoint-radius over the lower-upper using different precision on two GPU architectures. The behaviour observed is the result of different factors. Some of them pertain exclusively to the implementation, the others are introduced by CUDA and the hardware.

The pure effect of the implementation is observed between 1 and 8 α-cuts. In this range, the curve follows a growing pattern which is consistent with the theoretical analysis of the number of operations presented in Section 5.4. According to Table 5.1, the ratio between the number of operations per cycle of the AXPY

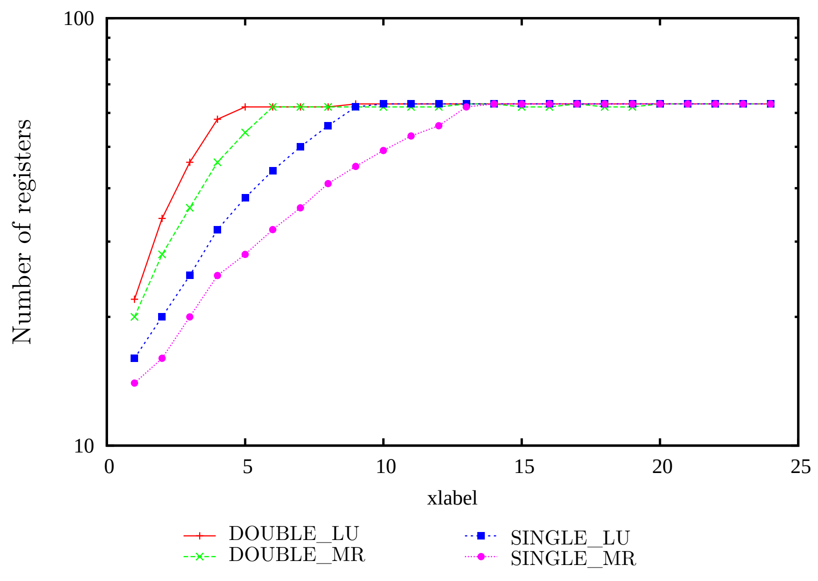


(a)

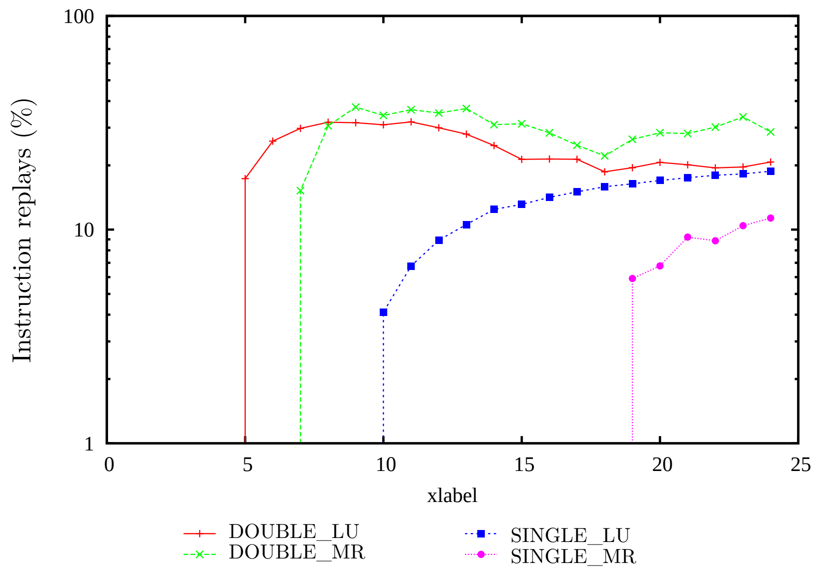


(b)

Figure 5.3: Results of the compute-bound test: (a) Performance comparison of different representation formats. (b) Speed-up achieved by midpoint-radius over lower-upper.



(a)



(b)

Figure 5.4: Results of the compute-bound test: (a) Registers used in each representation format. (b) Instruction replays due to local memory accesses in each format.

loop required by each format is:

$$r_c(N) = \frac{16N}{7 + 7N}, \tag{5.4}$$

where N is the number of α-cuts. Note that the speed-up curve in Fig. 5.3b follows approximately equation (5.4) between 1 and 8 to 9 α-cuts. Moreover, such behaviour does not depend on the architecture or the precision. This result validates the theoretical analysis and illustrates the performance of the library when the architecture is not saturated.

The effects of CUDA and the hardware are observed from 9 to 10 α-cuts. In this area, register spilling and local memory accesses with high latency become relevant. It is worth noticing that CUDA devices of compute capability 2.0 and 3.0 may allocate up to 63 registers of 32-bits per thread on a kernel execution. Double precision values are stored in two consecutive registers. If this limit is binding, the kernel uses local memory to allocate extra data. Figure 5.4(a) shows the number of registers used by the AXPY kernel in each format. Figure 5.4(b) shows the percentage of replayed instructions due to local memory accesses.

When using the lower-upper format and single precision values, the 63 registers limit is reached at about 9 α-cuts. Local memory transactions start at 10 α-cuts. Figure 5.3(b) shows an increase of the speed-up in single precision for 10 α-cuts, as the midpoint-radius format is not suffering from register spilling yet. The size of the midpoint-radius fuzzy being about half of the lower-upper, spilling only starts at double the α-cuts, i.e., 18. For more than 18 α-cuts, the speed-up curve in single precision moves back to the ideal shape that can be explained purely by the ratio of number of operations.

In double precision, the scenario is slightly more complex. Both lower-upper and midpoint-radius fuzzy start spilling registers relatively early, between 6 and 8 α-cuts. Figure 5.3(b) shows that there is a slight drop in the speed-up in double precision, at about 9 α-cuts. The performance of the midpoint-radius format decreases in this area. This is due to spilling the midpoint of one of the fuzzy intervals involved in the calculation. Note that midpoint-radius fuzzy arithmetic proceeds by computing the midpoint first and then uses the result to calculate all the radii. If one midpoint is spilled to off-chip local memory, there might not be enough independent arithmetic instructions to hide the latency of accessing that value. This effect cannot be appreciated in single precision as register spilling does not have a relevant impact despite the high number of α-cuts considered. This proves that performance is mainly driven by the amount of memory required to store data. Temporal dependency among data has a limited impact.

Next, the GPU L1 cache/shared memory configuration is changed in order to see the impact of different memory allocations over register spilling. According to the CUDA programming guide [Cuda 2012], the two following configurations are possible for GPUs with compute capability 2.0 or higher: (i) 48 KB of L1 cache and 16 KB of shared memory, and (ii) 16 KB of L1 cache and 48 KB of shared

```
#include "fuzzy_lib.h"
#include <thrust/sort.h>

int main(){
  thrust::device_vector<fuzzy<double, 4> > d_a(M);
  thrust::device_vector<unsigned int> k(M);
  ...
  thrust::sort_by_key(k.begin(), k.end(), d_a.begin());
}
```

Figure 5.5: THRUST's sort by keys, memory-bound test.

memory, totalling 64 KB in both cases. In the proposed experimental study, option (i) increases the number of α-cuts for which register spilling is still not an issue. On the other hand, option (ii) mitigates the negative effect of register spilling whenever it appears.

It is worth noting that real-world applications typically do not involve more than 3 to 4 α-cuts. In consequence, register spilling, which only appears at 8 α-cuts, should not be a problem in the majority of cases.

Memory-bound test: sort by keys

Figure 5.5 shows a THRUST [Hoberock 2010] program that sorts a vector of fuzzy intervals on the device. This example shows how easily the proposed fuzzy type is integrated to other GPU libraries, as all the arithmetic operators are overloaded.

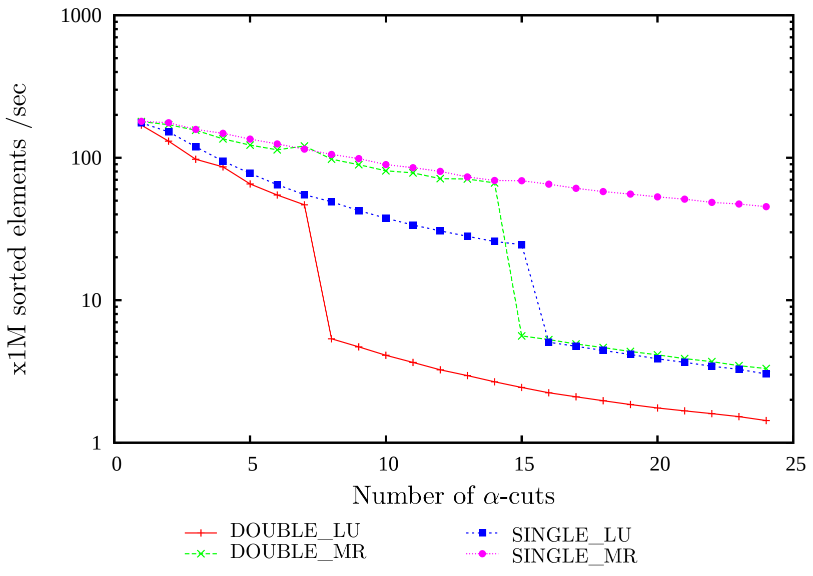
The vector is sorted by keys, which are integers of 32 bits. The sorting algorithm used by THRUST in this case is radix sort. The THRUST kernels read the fuzzy array from global memory, sort it on the device, and then write the sorted array back to global memory. The sorting process performs one step per key bit, i.e., 32 steps in this case. At each step, each element in the fuzzy array is read and copied into a new position. This is a typical case of memory-bound application.

GPU sorting time is measured when varying different key parameters, as in the previous experiment. Results for the GTX 480 are plotted in Fig. 5.6. Figure 5.6(a) shows performance of different representation formats in terms of millions of sorted elements per second. Figure 5.6(b) shows the speed-up of midpoint-radius over lower-upper. The time spent in transferring data between host and device is not considered in the experiment.

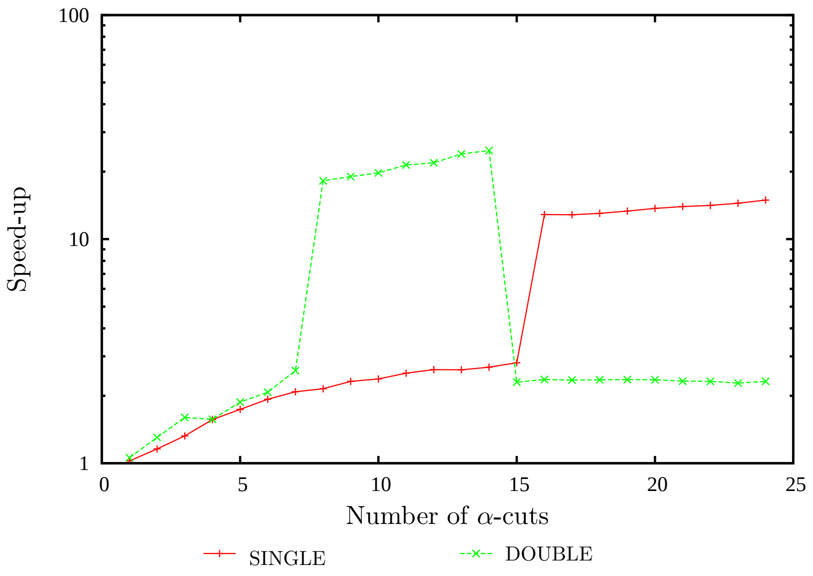
As in the compute-bound test, the results are a combination of purely algorithmic factors and CUDA-related factors. The purely algorithmic behaviour can be observed before 8 and afer 14 α-cuts in double precision, and after 15 α-cuts in single precision. In this zone, the speed-up curves follow approximately the ratio between the memory requirements by each format. This can be obtained from Table 5.2:

$$r_m(N) = \frac{2N}{1+N}, \tag{5.5}$$

where N is the number of α-cuts. Outside that area, the effects of CUDA and the



(a)



(b)

Figure 5.6: Results of the memory-bound test: (a) Performance comparison of different representation formats. (b) Speed-up achieved by midpoint-radius over lower-upper.

hardware become more relevant, causing the speed-up to fluctuate. THRUST's sort uses shared memory to optimize the sorting process. When the values being treated by threads within one Streaming Multiprocessor do not longer fit in shared memory, the application starts using global memory with a direct impact over performance. As stated above, lower-upper needs twice the amount of memory than midpoint-radius. As a consequence, it saturates shared memory at half the number of α-cuts. When both architectures saturate shared memory, the speed-up curve goes back to the normal behaviour, explained by the ratio of memory usage (equation (5.5)). Note that single precision midpoint-radius does not saturate shared memory in this experiment and keeps a high performance all along the considered range.

5.5 Proposed method for fuzzy PF analysis

This section presents a novel approach to evaluate the impact of uncertainty on PF analysis, using symmetric fuzzy intervals discussed above. As discussed in Section 3.3.2, uncertainty has been traditionally addressed by means of two approaches: (i) probabilistic and (ii) interval-based. The probabilistic approach, which usually takes the form of Monte Carlo simulations, is able to provide increasingly accurate results by enlarging the sample size. However, the computational cost increases as well. On the other hand, the interval-based approach does not require as much computation as Monte Carlo, although the radius of the interval result tends to grow with the size of the problem.

The purpose of this section is to present a technique which is not as expensive as Monte Carlo, but more accurate than previous interval-based approaches. With this aim, midpoint-radius fuzzy interval algebra is applied to the classic PF equations. Then, the problem is split into two consecutive parts: (i) the midpoint is computed using crisp PF analysis, and (ii) the radius is obtained by solving a minimization problem, especially designed to prevent overestimation of the interval result.

Recently, another approach has been proposed in [Vaccaro 2013]. This approach is based on range arithmetic and a mixed complementarity problem formulation of PF analysis. The interval PF solution is obtained by solving two nonlinear optimization problems, one for the lower bound and one for the upper bound.

The remainder of this section describes the proposed technique, and presents a case study including comparisons with the Monte Carlo approach.

5.5.1 Method description

The proposed methodology consists in using the midpoint-radius format to represent the fuzzy intervals, which allows the PF equations to be solved separately for the midpoint and the radius.

For clarity, let us recall the crisp PF equations (3.14) discussed in Chapter 3, as follows:

$$\mathbb{s} = \mathrm{Diag}(\mathbb{v})(\boldsymbol{Y}\mathbb{v})^{*}, \tag{5.6}$$

where $\mathbb{s}$ is the vector of complex power injections, $\mathbb{v}$ is the vector of complex voltages and $\boldsymbol{Y}$ is the admittance matrix. Diag($\mathbb{v}$) is the diagonal matrix that has $\mathbb{v}$ as diagonal and $(\cdot)^*$ denotes the complex conjugate. Let us also recall the following definitions:

$$\mathbb{s} = \boldsymbol{p} + j\boldsymbol{q}, \tag{5.7}$$
$$\mathbb{v} = \boldsymbol{v}\angle\boldsymbol{\theta}, \tag{5.8}$$

where $\boldsymbol{p}$ is the vector of real power injections, $\boldsymbol{q}$ is the vector of reactive power injections, $\boldsymbol{v}$ is the vector of voltage magnitudes and $\boldsymbol{\theta}$ is the vector of voltage phase angles.

Equation (5.6) is translated in interval form as follows:

$$[\mathbb{s}] = \mathrm{Diag}([\mathbb{v}])(\boldsymbol{Y}\,[\mathbb{v}])^*, \tag{5.9}$$

where $[\mathbb{s}]$ and $[\mathbb{v}]$ are now complex interval vectors. The method assumes that there is no variability associated with line parameters. Therefore, the admittance matrix $\boldsymbol{Y}$ is crisp. This is a common assumption in similar approaches (e.g., [Vaccaro 2013])

Using the midpoint-radius format, $[\mathbb{s}]$ and $[\mathbb{v}]$ are expressed in the following form:

$$[\mathbb{s}] = \langle \check{\mathbb{s}}, \boldsymbol{\rho_s} \rangle, \tag{5.10}$$
$$[\mathbb{v}] = \langle \check{\mathbb{v}}, \boldsymbol{\rho_v} \rangle, \tag{5.11}$$

where

$$\check{\mathbb{s}} = \check{\boldsymbol{p}} + j\check{\boldsymbol{q}}, \tag{5.12}$$
$$\check{\mathbb{v}} = \check{\boldsymbol{v}}\angle\check{\boldsymbol{\theta}}, \tag{5.13}$$

and $\boldsymbol{\rho_s}, \boldsymbol{\rho_v} \in \mathbb{R}$.

The above means that each element in $[\mathbb{s}]$ and $[\mathbb{v}]$ defines a *disc* in the complex plane. In this way the power radius, $\boldsymbol{\rho_s}$, simultaneously defines the variation for both the real and reactive power. Similarly, the voltage radius, $\boldsymbol{\rho_v}$, simultaneously defines the variation for both the voltage magnitude and phase angle (see [Rump 1999] for a complete description of complex intervals in the midpoint-radius format).

By applying the rules of interval arithmetic introduced above (see Definition 5.2.5), equation (5.9) is split in midpoint and radius parts. This yields the following system of equations:

$$\check{\mathbb{s}} = \mathrm{Diag}(\check{\mathbb{v}})(\boldsymbol{Y}\check{\mathbb{v}})^*, \tag{5.14a}$$
$$\boldsymbol{\rho_s} = \mathrm{Diag}(|\check{\mathbb{v}}| + \boldsymbol{\rho_v})|\boldsymbol{Y}|\boldsymbol{\rho_v} + \mathrm{Diag}(\boldsymbol{\rho_v})|\boldsymbol{Y}\check{\mathbb{v}}|. \tag{5.14b}$$

The knowns and unknowns of this system are summarized in Table 5.5.

Note that equation (5.14a) represents a crisp PF problem with the midpoints

Table 5.5: Known and unknown variables for each type of bus in the proposed fuzzy PF analysis method.

Bus type	Knowns	Unknowns
Generator	$\check{p}, \check{v}, \rho_s, \rho_v$	$\check{\theta}, \check{q}$
Load	$\check{p}, \check{q}, \rho_s$	$\check{v}, \check{\theta}, \rho_v$
Slack	$\check{v}, \check{\theta}, \rho_v$	$\check{p}, \check{q}, \rho_s$

of all variables. Solving this problem, e.g., by using the Newton-Raphson (NR) method, yields $\check{\mathbf{v}}$, the midpoint of the interval solution (see Chapter 3 for a description of the NR method). The value of $\check{\mathbf{v}}$ obtained from the crisp PF solution can be replaced in equation (5.14b). Then, the Newton method can be used to obtain the value of $\boldsymbol{\rho_v}$ that satisfies (5.14b). The main drawback of this technique is that it can lead to a value of $\boldsymbol{\rho_v}$ with some negative components. To avoid this, the equality in (5.14b) is relaxed by allowing the calculated power radius to be greater than the specified value. This yields the following inequality:

$$\boldsymbol{\rho_s} \leq \mathrm{Diag}(|\check{\mathbf{v}}| + \boldsymbol{\rho_v})|\boldsymbol{Y}|\boldsymbol{\rho_v} + \mathrm{Diag}(\boldsymbol{\rho_v})|\boldsymbol{Y}\check{\mathbf{v}}|. \tag{5.15}$$

The above inequality represents a sufficient condition for the voltage radius $\boldsymbol{\rho_v}$ to be a solution of the PF problem specified by the power radius $\boldsymbol{\rho_s}$ (given the precomputed voltage midpoint, $\check{\mathbf{v}}$). This condition redefines the power balance at each bus in terms of *interval* injections. The interval power injected by transmission lines must at least *contain* the interval power injected by generators and loads.

Clearly, any sufficiently large radius will satisfy the above condition. Hence, to obtain the smallest possible radius the following minimization problem is proposed:

$$\begin{aligned}
&\text{minimize} && \|\boldsymbol{\rho_v}\|_p && (5.16a)\\
&\text{subject to} && \mathrm{Diag}(|\check{\mathbf{v}}| + \boldsymbol{\rho_v})|\boldsymbol{Y}|\boldsymbol{\rho_v} + \mathrm{Diag}(\boldsymbol{\rho_v})|\boldsymbol{Y}\check{\mathbf{v}}| - \boldsymbol{\rho_s} \geq 0, && (5.16b)\\
& && \boldsymbol{\rho_v} \geq \mathbf{0}, && (5.16c)
\end{aligned}$$

where $\|\cdot\|_p$ is the p-norm (see the definition of p-norm in Appendix A). The choice of the norm can be based on a specific need, e.g., reducing the complexity or increasing the accuracy of the result. Section 5.5.2 presents a case study using the 2-norm, i.e., the sum of squared vector components.

Complexity analysis. This paragraph studies the computational complexity of the proposed method as well as of other methods from literature.

The complexity of the Monte Carlo method depends on the technique used for solving each instance of PF analysis, namely NR. In turn, the complexity of NR is given by the cost of LU factorizing the Jacobian matrix. As the cost of LU factorization is proportional to the cost of multiplying two matrices, there exists a $\mathcal{O}(n^{2.376})$ algorithm based on the Coppersmith–Winograd algorithm [Coppersmith 1987]. Therefore, the complexity of Monte Carlo simulations using the NR method is of $\mathcal{O}(m \cdot n^{2.376})$, where m is the number of samples.

The complexity of the interval approach is that of the Classic Interval Newton method, which requires solving an interval linear system at each iteration. Using the algorithm described in [Rump 1992], the above can be done in $5n^3$ operations, i.e., $\mathcal{O}(n^3)$.

Finally, the complexity of the method proposed in [Vaccaro 2013] and the complexity of the method proposed in this thesis are the same. Both methods are dominated by solving a nonlinear optimization problem. Using the interior point method, the above can be done in $\mathcal{O}(n^3)$ steps [Den Hertog 1992].

Table 5.6 summarizes the discussions above. Observe that in the Monte Carlo method, m is generally much larger than n, which leads to a complexity much higher than in the other three approaches.

Table 5.6: Computational complexity of methods for PF analysis under uncertainty.

Method	Complexity
Monte Carlo	$\mathcal{O}(m \cdot n^{2.376})$
Interval Newton	$\mathcal{O}(n^3)$
Range Arithmetic	$\mathcal{O}(n^3)$
Proposed method	$\mathcal{O}(n^3)$

5.5.2 Case study

This subsection studies the accuracy of the proposed method by comparing with the Monte Carlo approach. The objective is to determine the feasibility of the method in real-world applications. With this aim, the proposed method is implemented in C++ using the OPT++ library, a package for nonlinear optimization based on an interior point method [Meza 2007]. The implementation consists of two routines: the first is a standard PF solver based on the NR method, and is used to obtain the midpoint. The second uses OPT++ to solve the optimization problem defined by (5.16), thus determining the radius. The objective function in (5.16a) is chosen as the sum of squared vector components (i.e., the square of the Euclidean norm). A Newton nonlinear interior-point method provided by OPT++ is chosen as the optimization method [Potra 2000]. This method needs to compute the gradient and Hessian matrix of the nonlinear constraints. To alleviate the programming burden, the latter is computed using finite differences in place of the analytical expression. Since the network used in the study is relatively small, the above is not dramatic from the point of view of performance.

The Monte Carlo approach is implemented using the Python-based software Dome [Milano 2013]. Dome is a tool for power system analysis that provides the interface needed to perform Monte Carlo simulations. The input file containing the network specification is modified in order to introduce a random scaling on certain parameters. This allows the power injections and consumptions to oscillate within the desired bounds. Then, an arbitrary number of PF analyses can be launched over this random network, thus obtaining the result of the Monte Carlo experiment. One additional feature is that the program is able to run in parallel on multiple computing cores, reducing the total simulation time.

Both implementations are tested over a fuzzified version of the IEEE 57 bus test case system. This system represents a part of the American Electric Power System as it was in the early 1960's. It has 7 generators, 42 loads and 80 transmission lines. Figure 5.7 shows a one-line diagram. The original (non-fuzzified) data can be found at `http://www.ee.washington.edu/research/pstca/`. The fuzzification consists in assuming a percentage variation over generator bus voltages and power injections and consumptions. For generator bus voltages, the variation considered is $\pm 1\%$. For power injections and consumptions, three cases are considered: (a) $\pm 10\%$, (b) $\pm 20\%$ and (c) $\pm 50\%$ variation. This idea is translated to the Monte Carlo simulations by assuming uniformly distributed random variables within the respective bounds. Then, 5,000 runs of the PF analysis are launched using the Dome implementation.

The execution time of the proposed method implemented in C++ is about 10 seconds in average, whereas the execution time of the Monte Carlo approach implemented with Dome is larger than 1 minute.

Results are shown in Fig. 5.8. The bars represent the bounds obtained for the bus voltages in the Monte Carlo approach. Bullets and crosses represent the bounds obtained with the proposed method. It can be seen that both methods yield very similar results in case (a). In cases (b) and (c), however, as the considered variation increases, some discrepancies appear. In case (c), the proposed method reports a very high variation on the voltage at one of the buses, namely bus 7. This variation is almost three times that of the Monte Carlo approach. At other buses, in turn, the variation obtained by Monte Carlo is much higher, up to three times that of the proposed method. Apparently, the large interval at bus 7 is absorbing much of the variability considered in the inputs, allowing other buses to keep a narrow interval. That is to say, the constraint in (5.16b) gets satisfied by the large voltage radius at bus 7, without the other radii needing to increase too much.

As mentioned above, the current implementation uses the 2-norm in the definition of the objective function. However, it is possible that using a norm of a 'higher rank', such as the ∞-norm, yields better results in scenarios of high variation. The ∞-norm corresponds to the maximum absolute value of vector components. Thus, using this norm will penalize solutions where one of the radius is much larger than the others, as in case (c) above. As a consequence, the resulting interval voltage will have a more homogeneous profile, similar to the one seen in the Monte Carlo approach. Unfortunately, the implementation of the ∞-norm poses some difficulties, as the interior point optimization method requires to evaluate the derivative of the objective function. Since the ∞-norm is not differentiable, the solution needs to be

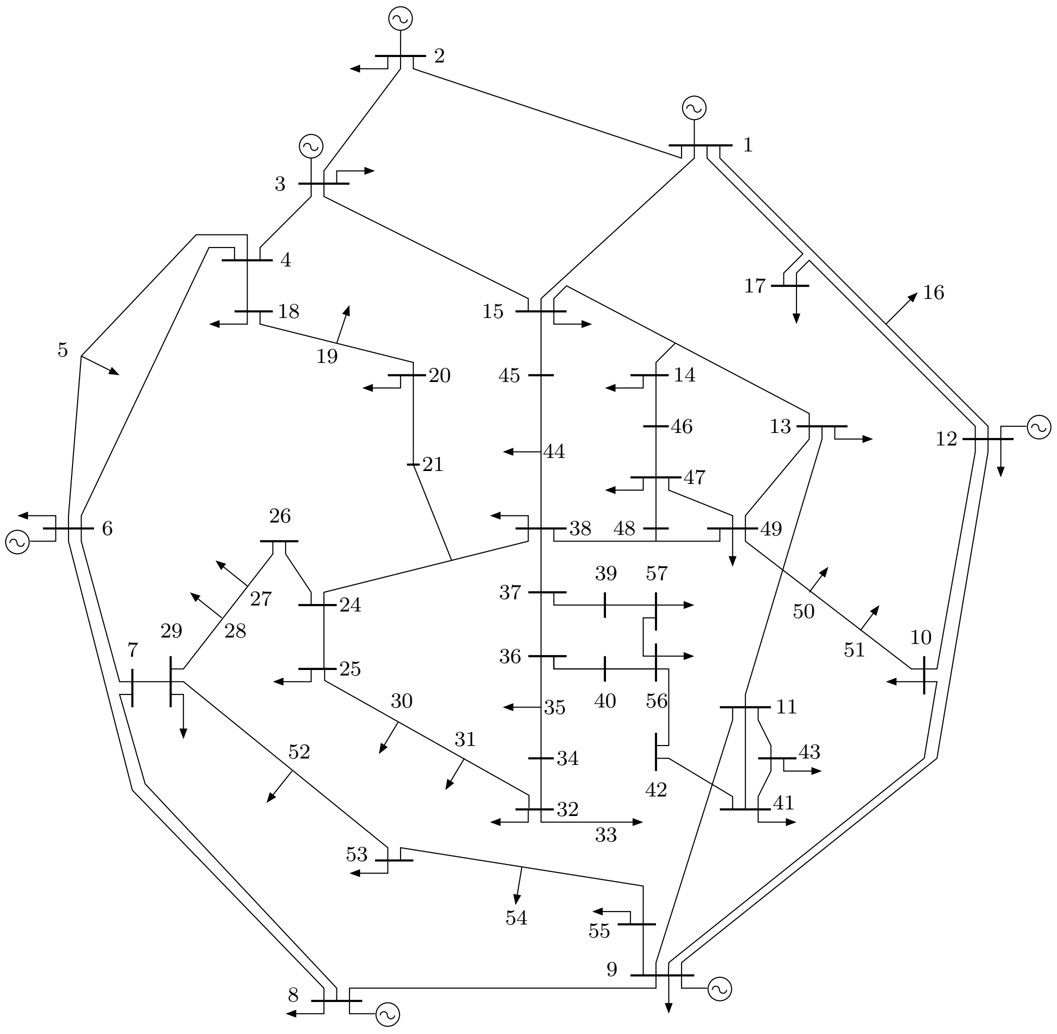


Figure 5.7: One-line diagram of the IEEE 57 bus test case system.

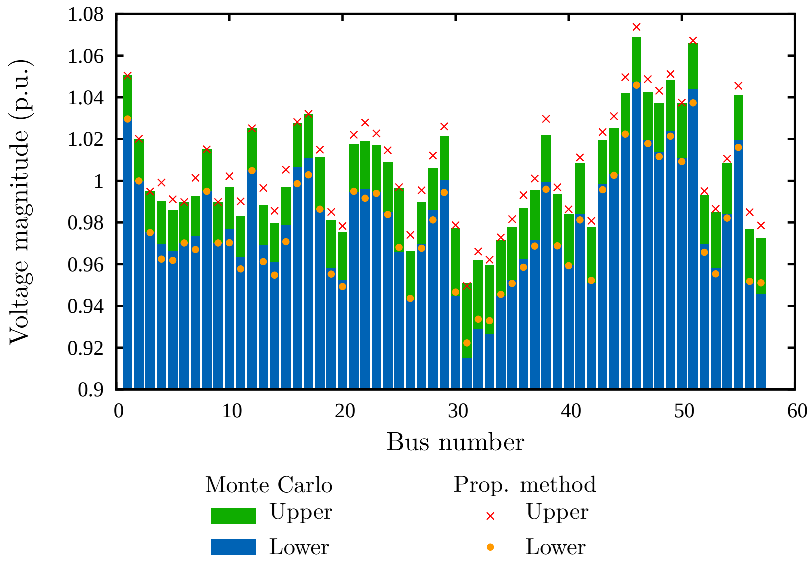
approached by other optimization methods, probably more expensive in terms of computation. Of course, a trade-off between accuracy and performance is expected for this kind of numerical applications. An investigation of this trade-off is planned as future work.

5.6 Conclusions

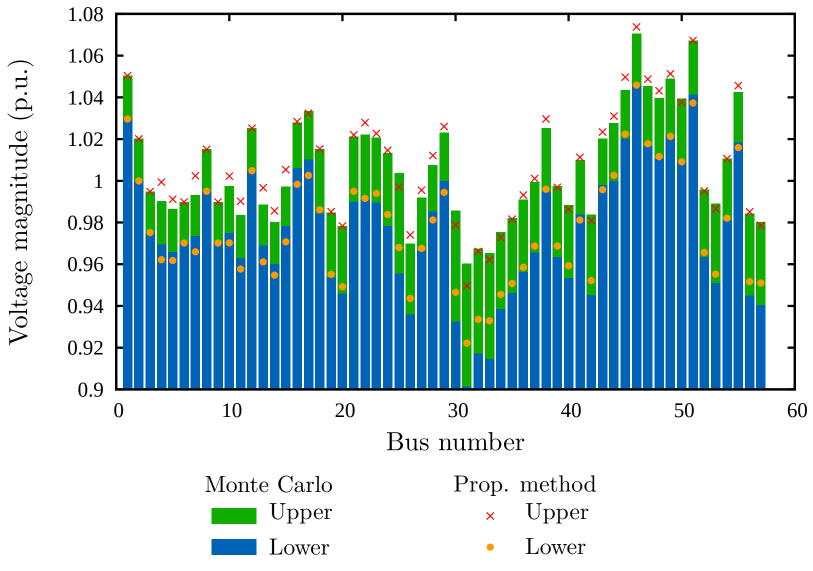
This chapter proposes and compares different representation formats for fuzzy intervals and their implementation on GPUs. The conventional lower-upper approach is compared to the midpoint-radius approach, which is particularly suited to model symmetric membership functions. A novel representation format is proposed based on the midpoint-radius approach. The proposed format allows halving both the memory requirements and the number of operations compared to the lower-upper format. The midpoint-increment format is also explored in this paper. This format improves the accuracy of the midpoint-radius format while using the same number of operations. With this aim, a library of fuzzy interval arithmetic operations, written in CUDA and C++, is described and evaluated using compute-bound and memory-bound benchmarks.

In addition, the midpoint-radius interval approach is used for the assessment of uncertainty in PF analysis. A novel technique is proposed, which separates the computation of the midpoint from the computation of the radius. The proposed technique proves to yield relatively accurate result at an affordable computational cost.

Future works will include the expansion of the fuzzy arithmetic library with more operations and conversion between formats. In case of non-symmetric membership function, the use of a symmetric envelope will be explored. Also, regarding the proposed method for the uncertainty assessment in PF analysis, the trade-off between complexity and accuracy will be investigated.

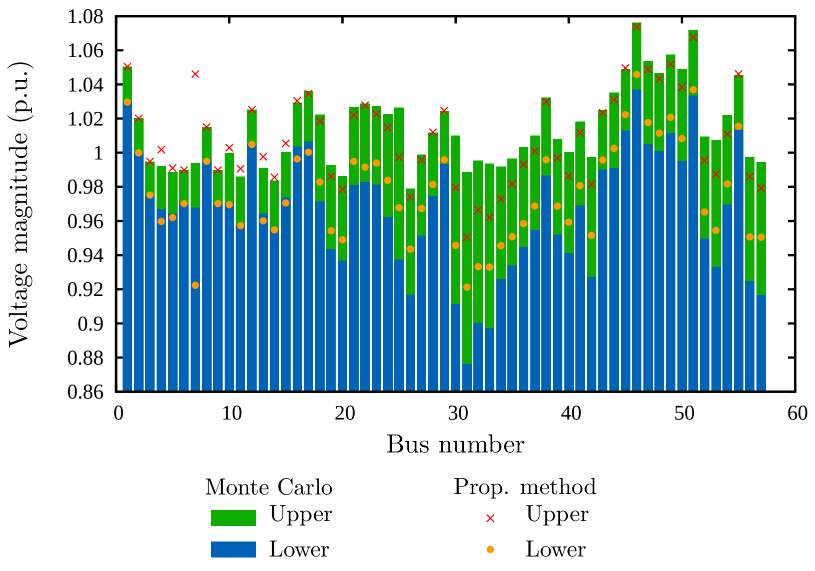


(a)



(b)

Figure 5.8: Bus voltage magnitude intervals for different variations in load power consumption and generator power injection: (a) ±10% variation. (b) ±20% variation.



(c)

Figure 5.8: Bus voltage magnitude intervals for different variations in load power consumption and generator power injection: (c) ±50% variation.

References

[461 2008] *IEEE Standard for Floating-Point Arithmetic.* IEEE Std 754-2008, pp. 1–70, 2008. (Cited on page 77.)

[Anile 1995] A. Anile, S. Deodato and G. Privitera. *Implementing fuzzy arithmetic.* Fuzzy Sets and Systems, vol. 72, no. 2, pp. 239–250, 1995. (Cited on page 76.)

[Beliakov 2015] G. Beliakov and Y. Matiyasevich. *A parallel algorithm for calculation of determinants and minors using arbitrary precision arithmetic.* BIT Numerical Mathematics, pp. 1–18, 2015. (Cited on page 88.)

[Bondia 2006] J. Bondia, A. Sala, J. Pico and M. Sainz. *Controller Design Under Fuzzy Pole-Placement Specifications: An Interval Arithmetic Approach.* IEEE Transactions on Fuzzy Systems, vol. 14, no. 6, pp. 822–836, 2006. (Cited on pages 75 and 85.)

[Boukezzoula 2014] R. Boukezzoula, S. Galichet, L. Foulloy and M. Elmasry. *Extended gradual interval (EGI) arithmetic and its application to gradual weighted averages.* Fuzzy Sets and Systems, vol. 257, pp. 67–84, 2014. (Cited on page 81.)

[Brönnimann 2006] H. Brönnimann, G. Melquiond and S. Pion. *The design of the Boost interval arithmetic library.* Theoretical Computer Science, vol. 351, no. 1, pp. 111–118, 2006. (Cited on page 76.)

[Buisson 2003] J.-C. Buisson and A. Garel. *Balancing meals using fuzzy arithmetic and heuristic search algorithms.* IEEE Transactions on Fuzzy Systems, vol. 11, no. 1, pp. 68–78, 2003. (Cited on pages 75 and 85.)

[Chang 1995] C.-H. Chang and Y.-C. Wu. *The genetic algorithm based tuning method for symmetric membership functions of fuzzy logic control systems.* International IEEE/IAS Conference on Industrial Automation and Control: Emerging Technologies, 1995., pp. 421–428. IEEE, 1995. (Cited on page 81.)

[Chen 2006] C.-T. Chen, C.-T. Lin and S.-F. Huang. *A fuzzy approach for supplier evaluation and selection in supply chain management.* International journal of production economics, vol. 102, no. 2, pp. 289–301, 2006. (Cited on pages 75 and 85.)

[Coppersmith 1987] D. Coppersmith and S. Winograd. *Matrix multiplication via arithmetic progressions.* Proceedings of the nineteenth annual ACM symposium on Theory of computing, pp. 1–6. ACM, 1987. (Cited on page 98.)

[Cortes-Carmona 2010] M. Cortes-Carmona, R. Palma-Behnke and G. Jimenez-Estevez. *Fuzzy Arithmetic for the DC Load Flow.* IEEE Transactions on Power Systems, vol. 25, no. 1, pp. 206–214, 2010. (Cited on pages 75 and 85.)

[Cuda 2012] C. Cuda. *Programming guide.* NVIDIA Corporation, July, 2012. (Cited on pages 9, 24, 25, 27, 88 and 92.)

[Davis 2012] J. D. Davis and E. S. Chung. *SpMV: A Memory-Bound Application on the GPU Stuck Between a Rock and a Hard Place.* Technical Report MSR-TR-2012-95, 2012. (Cited on page 88.)

[Defour 2014] D. Defour and M. Marin. *FuzzyGPU: A Fuzzy Arithmetic Library for GPU.* Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on, pp. 624–631, 2014. (Cited on page 75.)

[Den Hertog 1992] D. Den Hertog. *Interior point approach to linear, quadratic and convex programming: algorithms and complexity.* PhD thesis, TU Delft, Delft University of Technology, 1992. (Cited on page 98.)

[Dubois 1978] D. Dubois and H. Prade. *Operations on fuzzy numbers.* International Journal of systems science, vol. 9, no. 6, pp. 613–626, 1978. (Cited on pages 76, 77 and 79.)

[Fortin 2008] J. Fortin, D. Dubois and H. Fargier. *Gradual Numbers and Their Application to Fuzzy Interval Analysis.* IEEE Transactions on Fuzzy Systems, vol. 16, no. 2, pp. 388–402, 2008. (Cited on page 82.)

[Gagolewski 2012] M. Gagolewski. *FuzzyNumbers Package: Tools to deal with fuzzy numbers in R.* http://www. ibspan. waw. pl/g̃agolews/FuzzyNumbers, 2012. (Cited on page 76.)

[Goossens 2013] B. Goossens and D. Parello. *Limits of instruction-level parallelism capture.* Procedia Computer Science, vol. 18, pp. 1664–1673, 2013. (Cited on page 87.)

[Goualard 2006] F. Goualard. *Gaol 3.1.1: Not just another interval arithmetic library.* Laboratoire d'Informatique de Nantes-Atlantique, 2006. (Cited on page 76.)

[Higham 2002] N. J. Higham. Accuracy and stability of numerical algorithms. Siam, 2002. (Cited on pages 66, 68, 77, 84 and 139.)

[Hoberock 2010] J. Hoberock and N. Bell. *Thrust: A Parallel Template Library, Version 1.7.0.* http://thrust.github.io/, 2010. (Cited on page 93.)

[Kolarovic 2013] N. Kolarovic. *Fuzzy numbers and basic fuzzy arithmetics (+, -, *, /, 1/x) implementation written in Java.* Online at http://http://sourceforge.net/projects/fuzzyarith/, 2013. (Cited on pages 76 and 89.)

[Marin 2014] M. Marin, D. Defour and F. Milano. *Power flow analysis under uncertainty using symmetric fuzzy arithmetic.* IEEE PES General Meeting, Conference & Exposition, pp. 1–5, 2014. (Cited on page 75.)

[Mendel 2006] J. Mendel and H. Wu. *Type-2 Fuzzistics for Symmetric Interval Type-2 Fuzzy Sets: Part 1, Forward Problems.* IEEE Transactions on Fuzzy Systems, vol. 14, no. 6, pp. 781–792, 2006. (Cited on page 81.)

[Meza 2007] J. Meza, R. Oliva, P. Hough and P. Williams. *OPT++: An object-oriented toolkit for nonlinear optimization.* ACM Transactions on Mathematical Software, vol. 33, no. 2, page 12, 2007. (Cited on page 98.)

[Milano 2013] F. Milano. *A Python-based software tool for power system analysis.* IEEE Power and Energy Society General Meeting, pp. 1–5, 2013. (Cited on pages 7, 10, 68, 98, 122 and 124.)

[Moore 1966] R. E. Moore. Interval analysis, vol. 4. Prentice-Hall Englewood Cliffs, 1966. (Cited on page 76.)

[Neumaier 1990] A. Neumaier. Interval methods for systems of equations, vol. 37. Cambridge university press, 1990. (Cited on pages 76 and 139.)

[Potra 2000] F. A. Potra and S. J. Wright. *Interior-point methods.* Journal of Computational and Applied Mathematics, vol. 124, no. 1 - 2, pp. 281 – 302, 2000. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations. (Cited on page 98.)

[Revol 2001] N. Revol and F. Rouillier. *The MPFI library.* http://mpfi.gforge.inria.fr/, 2001. (Cited on page 76.)

[Rump 1992] S. M. Rump. *On the solution of interval linear systems.* Computing, vol. 47, no. 3-4, pp. 337–353, 1992. (Cited on page 98.)

[Rump 1999] S. M. Rump. *Fast and parallel interval arithmetic.* BIT Numerical Mathematics, vol. 39, no. 3, pp. 534–554, 1999. (Cited on pages 77, 78 and 96.)

[Siler 2005] W. Siler and J. J. Buckley. Fuzzy expert systems and fuzzy reasoning. John Wiley & Sons, 2005. (Cited on page 76.)

[Trutschnig 2013] W. Trutschnig, M. Lubiano and J. Lastra. *SAFD — An R Package for Statistical Analysis of Fuzzy Data.* In C. Borgelt, M. Á. Gil, J. M. Sousa and M. Verleysen, editeurs, Towards Advanced Data Analysis by Combining Soft Computing and Statistics, vol. 285 of *Studies in Fuzziness and Soft Computing*, pp. 107–118. Springer Berlin Heidelberg, 2013. (Cited on page 76.)

[Vaccaro 2013] A. Vaccaro, C. A. Cañizares and K. Bhattacharya. *A range arithmetic-based optimization model for Power Flow analysis under interval uncertainty.* IEEE Transactions on Power Systems, vol. 28, no. 2, pp. 1179–1186, 2013. (Cited on pages 11, 46, 49, 95, 96 and 98.)

Chapter 6

Asynchronous Power Flow analysis

Contents

Elements of this chapter have been published as a research report in HAL open archives [Marin 2015]. A paper with most recent results is currently in preparation, to be submitted to IEEE Transactions on Power Systems.

6.1 Introduction

This chapter describes an intrinsically parallel method for Alternating Current (AC) circuit analysis and Power Flow (PF) analysis. As discussed in Chapter 3, these analyses are typically based on a vectorial approach and direct solvers. In the past, most implementations of such methods were designed to run on single-core processors. This allowed performance to scale with the processor frequency. More recently, as single-core processors started reaching their peak frequency, parallel implementations were investigated [Gupta 1997, Honkala 2001, Tu 2002, Garcia 2010].

The above methods, however, are intrinsically serial. That is to say, some parts of the algorithm need to wait for others to complete in order to start. In consequence, there is always a portion of the algorithm that remains serial in every parallel implementation. According to the Amdahl's law, this remaining serial code limits the performance of the parallel program even if the number of parallel cores grows to infinity [Amdahl 1967]. For performance to keep growing with the number of cores, the algorithm must be intrinsically parallel. The use of analog emulation has been investigated in [Nagel 2013]. This consists in mapping the real power system onto a CMOS integrated circuit for its simulation. However, no software-based intrinsically parallel method for circuit or PF analysis has been proposed in the current literature.

The approach presented in this chapter is inspired from the physical system, which, in fact, is intrinsically parallel: each component in the physical system is continuously evolving, without explicit synchronization with others. At some point, an equilibrium point naturally arises. In the proposed method, each element in the network is performing a fixed-point iteration. If all these converge, the current state is the solution for the network. The above is implemented using the concept of *team algorithms*, presented in [Talukdar 1983], where different algorithms are combined into the solution of a common problem. A similar idea is presented in [Tsitsiklis 1984] under the name of *agreement algorithm*.

An important antecedent of the proposed approach is the work in [Bertsekas 1987], where the authors study the use of asynchronous iterations in the network flow problem. Indeed, the network flow problem is representative of the situation that arises in power flow analysis when the standard formulation is utilized. In [El-Baz 1996], a novel computational model is introduced which allows *flexible communication* between program threads, helping to accelerate convergence of a network flow algorithm.

The remainder of the chapter is organized as follows. Section 6.2 describes several alternatives to implement fixed-point iterations in parallel, using different synchronization schemes. In Section 6.3, the proposed intrinsically parallel method for AC circuit analysis is presented and convergence is studied. The method is extended to PF analysis in Section 6.4. Section 6.5 describes the implementation and evaluation of the method on a set of benchmarks and comparison with existing approaches. Finally, Section 6.6 draws conclusions and outlines future work.

6.2 Fixed-point iterations on a parallel computer

This section introduces the concept of fixed-point iterations and discusses the implementation of such iterations on a parallel machine. Fixed-point iterations are iterative processes that lead to a fixed-point of a function, i.e., a point that is mapped to itself.

Definition 6.2.1 (Fixed-point and fixed-point iteration)**.** Let $\boldsymbol{f}\colon \mathbb{R}^n \to \mathbb{R}^n$ and $\boldsymbol{x} \in \mathbb{R}^n$. Then, $\boldsymbol{x}$ is a fixed-point of $\boldsymbol{f}(\cdot)$ if and only if $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x})$. Furthermore, let $\boldsymbol{x}^{(0)} \in \mathbb{R}^n$. Then, the iteration given by:

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{f}(\boldsymbol{x}^{(k)}), \quad k = 0, 1, \dots, \tag{6.1}$$

is called a fixed-point iteration on $\boldsymbol{f}(\cdot)$.

Examples of fixed-point iterations are the Jacobi and Gauss-Seidel methods discussed in Section 3.3.1. Fixed-point iterations can be implemented in parallel by assigning each element of the iterated vector to an individual thread. Each thread updates its respective element and communicates its result to others. A synchronization point may be defined, where the process has to make sure that the results computed by each thread are visible to others.

6.2.1 Synchronous iterations

The usual setup of a fixed-point iteration, as presented by Definition 6.2.1, corresponds to a *synchronous* iteration in the context of parallel computing. This means that the vector is updated as a block, i.e., all the elements are updated before the process moves as a whole into the next step. The above requires to place a synchronization point at the end of each iteration. As a consequence, threads that finish their updates earlier are forced to wait for the others to catch up. The reason for threads having different processing times are various. Some of them may be intrinsic to the algorithm, e.g., irregular memory access patterns, load imbalance. Some others may be related to the architecture, e.g., differences in processor frequency, communication network constraints [Frommer 2000].

6.2.2 Asynchronous iterations

An alternative to the synchronous approach is the so-called *asynchronous* iteration, in which individual threads do not wait for others and simply go ahead with their work. This means that the vector is updated by chunks, i.e., at every asynchronous iteration only a subset of the vector elements are updated. No synchronization point is explicitly added to the program. As a consequence, some of the updates may be computed with data from several iterations back.

An asynchronous iteration is defined in terms of an *update function* and a *delay function*. The update function, noted $\mathcal{U}(\cdot)$, receives the iteration counter $k \in \mathbb{N}$, and returns a subset of $\{1, \dots, n\}$ indicating the list of threads that update their elements at iteration k. The delay function, noted $d(\cdot)$, receives the indices $i, j \in \{1, \dots, n\}$ and the iteration counter $k \in \mathbb{N}$, and returns the delay of thread j with respect to thread i at iteration k. This delay represents the number of iterations performed since the value of x_j was last made visible to i.

Definition 6.2.2 (Asynchronous iteration)**.** Let $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{f} \colon \mathbb{R}^n \to \mathbb{R}^n$. For $k \in \mathbb{N}, i, j \in \{1, \dots, n\}$, let $\mathcal{U}(k) \subseteq \{1, \dots, n\}$ and $d(i, j, k) \in \mathbb{N}_0$, such that:

$$d(i,j,k) \geq 0, \qquad \forall i, j, k, \tag{6.2a}$$

$$\lim_{k\to\infty} d(i,j,k) < \infty, \qquad \forall i, j, \tag{6.2b}$$

$$|\{k \colon i \in \mathcal{U}(k)\}| = \infty, \qquad \forall i. \tag{6.2c}$$

Finally, let $\boldsymbol{x}^0 \in \mathbb{R}^n$. Then, the iteration defined by:

$$x_i^{(k+1)} = \begin{cases} f_i(x_1^{(k-d(i,1,k))}, \dots, x_n^{(k-d(i,n,k))}) & \text{if } i \in \mathcal{U}(k), \\ x_i^{(k)} & \text{if } i \notin \mathcal{U}(k), \end{cases} \tag{6.3}$$

is called an asynchronous iteration on $\boldsymbol{f}(\cdot)$, with update function $\mathcal{U}(\cdot)$ and delay function $d(\cdot)$.

The assumptions in (6.2a), (6.2b) and (6.2c) ensure that the process is well defined. Specifically, the assumption in (6.2a) states that only values computed in previous iterations are used in any update. The one in (6.2b) states that newer values of the elements are eventually used. Finally, the assumption in (6.2c) states that no element ceases to be updated during the course of the iteration. In practical terms, if the updates are made visible as soon as they are computed, then the third assumption implies the second. This is the case, for example, of an application that uses GPU global memory to store the data. Note that a synchronous iteration is a special case of asynchronous iteration with $\mathcal{U}(k) = \{1, \dots, n\}$ and $d(i, j, k) = 0$, for all i, j, k.

Due to the peculiarities introduced above, asynchronous iterations converge differently from synchronous ones. The following result considers the case of a linear application.

Theorem 6.2.1 (Necessary and sufficient condition for convergence of linear asynchronous iterations [Chazan 1969])**.** *Let $\boldsymbol{f} \colon \mathbb{R}^n \to \mathbb{R}^n$, given by,*

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{L}\boldsymbol{x} + \boldsymbol{b}, \tag{6.4}$$

where $\boldsymbol{L} \in \mathbb{R}^{n\times n}$, and $\boldsymbol{b} \in \mathbb{R}^n$. Let $|\boldsymbol{L}|$ denote the matrix of absolute values of the entries of $\boldsymbol{L}$, and $\rho(|\boldsymbol{L}|)$ its spectral radius (see Appendix A for a definition of spectral radius). If

$$\rho(|\boldsymbol{L}|) < 1, \tag{6.5}$$

then any asynchronous iteration on $\boldsymbol{f}(\cdot)$ converges to $\boldsymbol{x}^*$, the unique fixed-point of $\boldsymbol{f}(\cdot)$, regardless of the selection of $\mathcal{U}(\cdot)$, $d(\cdot)$ and $\boldsymbol{x}^{(0)}$. Furthermore, if $\rho(|\boldsymbol{L}|) \geq 1$, then there exists $\mathcal{U}(\cdot)$, $d(\cdot)$ and $\boldsymbol{x}^{(0)}$ such that the associated asynchronous iteration does not converge to $\boldsymbol{x}^*$.

Proof. See [Chazan 1969].

6.2.3 Partially asynchronous iterations

In some circumstances, convergence of asynchronous iterations can be facilitated by making threads synchronize every once in a while.

Definition 6.2.3 (Partially asynchronous iteration)**.** Consider Definition 6.2.2 of an asynchronous iteration. Replace the assumptions in (6.2b) and (6.2c) by the following:

$$\exists\, \bar{d} \in \mathbb{N} \colon d(i,j,k) \leq \bar{d}, \qquad \forall i,j,k, \tag{6.6a}$$

$$\exists\, \bar{s} \in \mathbb{N} \colon i \in \bigcup_{s=1}^{\bar{s}} \mathcal{U}(k+s), \qquad \forall i,k. \tag{6.6b}$$

$$d(i,i,k) = 0, \qquad \forall i,k, \tag{6.6c}$$

Then, the iteration given by equation (6.3) is now termed a partially asynchronous iteration.

The assumption in (6.6a) establishes that not only newer values of the vector elements are eventually used, but each of these values is used before $\bar{d}$ iterations have passed from their calculation. The assumption in (6.6b), in turn, states that each element is updated at least once in every $\bar{s}$ consecutive iterations. Finally, the assumption in (6.6c) establishes that every element is updated using its own last calculated value. In practical terms, the first two assumptions can be met by performing a synchronization step every l iterations, where l is the minimum of $\bar{d}$ and $\bar{s}$. The third assumption can be met by having each element assigned to only one thread.

As mentioned above, partially asynchronous iterations converge 'more easily' than totally asynchronous ones. The following result establishes a convergence criteria in the linear case, with softer assumptions than the ones in Theorem 6.2.1.

Theorem 6.2.2 (Sufficient condition for convergence of linear partially asynchronous iterations [Tseng 1990])**.** *Let $\boldsymbol{f} \colon \mathbb{R}^n \to \mathbb{R}^n$, given by,*

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{L}\boldsymbol{x} + \boldsymbol{b}, \tag{6.7}$$

where $\boldsymbol{L} \in \mathbb{R}^{n\times n}$, and $\boldsymbol{b} \in \mathbb{R}^n$. Let $\boldsymbol{g} \colon \mathbb{R}^n \to \mathbb{R}^n$, given by:

$$\boldsymbol{g}(\boldsymbol{x}) = (1-\alpha)\boldsymbol{x} + \alpha \boldsymbol{f}(x), \tag{6.8}$$

where $\alpha \in (0,1)$.

If $\boldsymbol{L} = (l_{ij})$ is irreducible (see Appendix A for a definition of irreducibility) and

$$\sum_{j=1}^{n} |l_{ij}| \leq 1, \quad \forall i, \tag{6.9}$$

then any partially *asynchronous iteration on $\boldsymbol{g}(\cdot)$ converges to $\boldsymbol{x}^*$, fixed-point of $\boldsymbol{f}(\cdot)$.*

Proof. See [Tseng 1990].

In [Lubachevsky 1986], the authors presented a special case in which convergence occurs for $\alpha = 1$, at a linear (geometric) rate. They also determined a lower bound for the average convergence rate (per iteration) in the long-term, τ, as follows:

$$\tau \geq (1 - \sigma^r)^{1/r}, \tag{6.10}$$

where

$$r = 1 + \bar{d} + (n-1)(\bar{s} + \bar{d}), \tag{6.11a}$$

$$\sigma = \min_{i,j}{}^{+} \left(\frac{x_i^* l_{ij}}{x_j^*} \right), \tag{6.11b}$$

and $\min^{+}(\cdot)$ refers to the minimum of the positive elements. This means that the error is scaled at most by a factor of $(1 - \sigma^r)^{1/r}$ in any average iteration. As σ is always lower than 1, the convergence rate approaches 1 for increasing values of n, $\bar{d}$ and $\bar{s}$. That is to say, the convergence becomes sub-linear.

For illustration purposes, consider that $\bar{d} = \bar{s} = 0$ (synchronous iteration). In this case, from equations (6.10) and (6.11a), $r = 1$ and $\tau = 1 - \sigma$. In other words, the convergence rate is unaffected by the problem size n and only depends on σ.

Now consider that $\bar{d} = \bar{s} = 1$. This corresponds to a partially asynchronous iteration with a synchronization step every other iteration. In this case, $r = 2n$, which means that convergence slows down as the problem grows. Figure 6.1 shows the lower bound on the convergence rate as a function of the problem size n. The different curves represent different values of σ. Observe that, unless σ is very high, convergence becomes sub-linear very quickly.

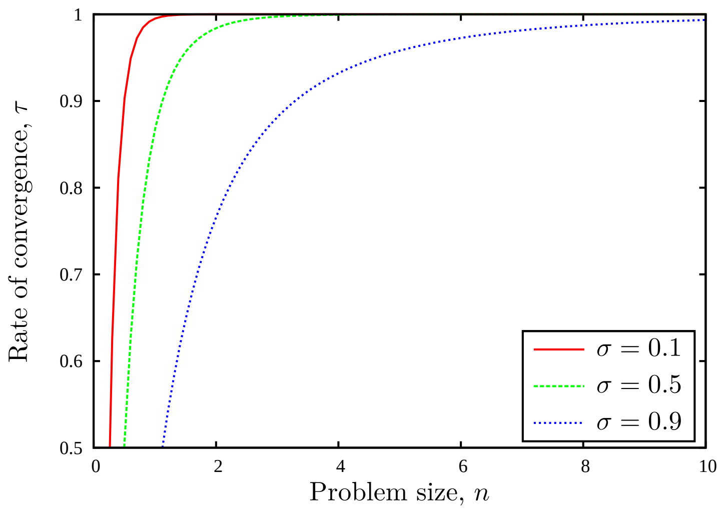


Figure 6.1: Bound on the convergence rate as a function of the problem size, for different values of the parameter σ.

This might be a strong reason to prefer the synchronous approach over the partially asynchronous. However, the partially asynchronous approach can be a valid alternative in some cases, e.g., if the costs of synchronization are relatively high, or when the number of iterations required to converge is relatively low.

6.3 Proposed method for AC circuit analysis

This section presents a novel technique for AC circuit analysis based on partially asynchronous iterations. The proposed method is intrinsically parallel, and attempts to emulate the real behaviour of the physical system. The method is designed to be implemented on Single Instruction Multiple Thread (SIMT) architectures such as the GPU. As seen in Section 2.2.1, the SIMT execution model allows the same instruction to be executed simultaneously by a large number of parallel cores over different data. The proposed method is based on a "universal component" model that allows for an efficient utilization of the above feature.

The method is presented through a top-down approach. The network model is described first, specifying how the different parts communicate and interact. Next, the models of the different elements that compose the AC system are specified in their atomic behaviour. Finally, the convergence properties of the proposed method are studied.

6.3.1 Network model

The proposed methodology distinguishes two fundamental units in the circuit: components and nodes. The components are the electrical devices, and the nodes are the points of interconnection. For sake of simplicity and also to facilitate illustration of the method, the following three assumptions are made:

1. Only dipole components, i.e., components connected to two nodes, are present.
2. There is no more than one component connected between any two nodes.
3. Every node is connected to at least two components.

The circuit is *well defined* if there are components connecting all the nodes in a closed path. Node h in the circuit is noted n_h. The component connected between n_h and n_m is noted c_{hm}. The set of all nodes in the circuit is noted $\mathcal{N}$, and the set of components, $\mathcal{C}$.

In addition, a set of *influencers*, $\mathcal{I}_h \subset \mathcal{N}$, is defined for each non-ground node $n_h \in \mathcal{N}$. This set contains all the nodes separated from n_h by exactly one component. These are also known as the *fanin* nodes [Newton 1984].

For illustration purposes, consider the circuit in Fig. 6.2. In this case, $\mathcal{N} = \{n_0, n_1, n_2, n_3\}$ and $\mathcal{C} = \{c_{01}, c_{12}, c_{02}, c_{23}, c_{03}\}$. The circuit is well defined, since c_{01}, c_{12}, c_{23} and c_{03} connect all the nodes in a closed path. In addition, $\mathcal{I}_1 = \{n_0, n_2\}$, $\mathcal{I}_2 = \{n_0, n_1, n_3\}$, and $\mathcal{I}_3 = \{n_0, n_2\}$.

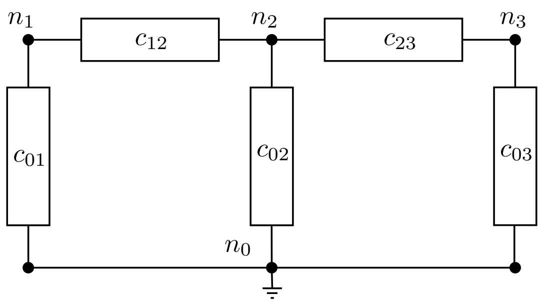
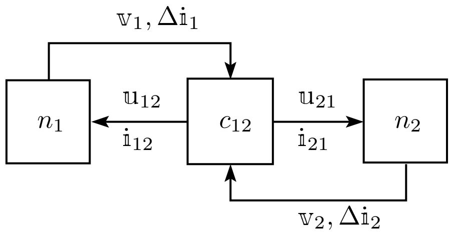


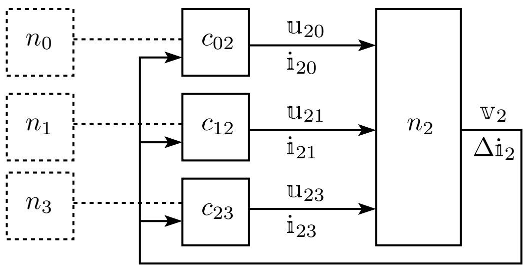
Figure 6.2: Example 4-node circuit.

In the proposed methodology, $\mathbb{v}_h$ denotes the voltage at node n_h, and $\mathbb{u}_{hm}$ denotes the voltage at node n_h 'according' to component c_{hm}. The two are different during the course of the iteration, and become equal at convergence. In addition, $\mathbb{i}_{hm}$ denotes the current injected to n_h by component c_{hm}, and $\Delta\mathbb{i}_h$ denotes the current mismatch at n_h.

At every iteration k, a component c_{hm} obtains $\mathbb{v}_h^{(k)}$ and $\Delta\mathbb{i}_h^{(k)}$ from node n_h, and returns $\mathbb{u}_{hm}^{(k)}$ and $\mathbb{i}_{hm}^{(k)}$ to it. A similar exchange occurs with node n_m at the other end of the component. Next, at the same iteration k, node n_h receives $\mathbb{u}_{hj}^{(k)}$ and $\mathbb{i}_{hj}^{(k)}$ from each component c_{hj} connected to it, and returns $\mathbb{v}_h^{(k+1)}$ and $\Delta\mathbb{i}_h^{(k+1)}$ to all of them. Then, the process moves into a next iteration. Figure 6.3 illustrates the above for component c_{12} and node n_2 in the example 4-node circuit.



(a)



(b)

Figure 6.3: Network model: (a) Component; and (b) Node.

6.3.2 Component and node models

Component model

For the purposes of the proposed method, only two types of component are required: the voltage source and the impedance. Although these are conceptually different, they are both modelled as a voltage source in series with an impedance. The source and the impedance assume the appropriate values to represent one and the other. The use of a universal model improves instruction regularity of the GPU implementation, as discussed in Section 6.5.

The voltage source of component c_{hm}, measured from n_h to n_m, is noted $\hat{\mathbb{v}}_{hm}$. The impedance is noted z_{hm}. Observe that $\hat{\mathbb{v}}_{hm} = -\hat{\mathbb{v}}_{mh}$ and $z_{hm} = z_{mh}$. The values of $\hat{\mathbb{v}}_{hm}$ and z_{hm} for each component are detailed below.

Voltage source. The voltage source is modelled as an independent source $\mathbb{V}$ of maximum current $\mathbb{I}$. Accordingly, $\hat{\mathbb{v}}_{hm}$ is set to $\mathbb{V}$. The value of z_{hm}, in turn, is chosen so that the voltage drop across its terminals is negligible, i.e., lower than a positive, small real number ε:

$$0 \leq z_{hm}\mathbb{I} < \varepsilon. \tag{6.12}$$

This is achieved by setting $z_{hm} = \frac{\varepsilon}{2\mathbb{I}}$.

Impedance. The impedance is simply modelled as a constant impedance Z. Accordingly, $\hat{\mathbb{v}}_{hm}$ is set to zero and z_{hm} is set to Z.

Table 6.1: Voltage source and resistance for each component.

Component	z_{hm}	$\hat{\mathbb{v}}_{hm}$
Voltage source	$\frac{\varepsilon}{2\mathbb{I}}$	$\mathbb{V}$
Impedance	Z	0

The component iteration is defined as follows:

$$\mathbb{u}_{hm}^{(k)} = \mathbb{v}_h^{(k)} + z_{hm}\Delta\mathbb{i}_h^{(k)}, \tag{6.13a}$$

$$\mathbb{i}_{hm}^{(k)} = \frac{\mathbb{v}_m^{(k)} - \mathbb{v}_h^{(k)} - \hat{\mathbb{v}}_{hm}}{z_{hm}}. \tag{6.13b}$$

These equations are obtained by applying the Ohm Law on component c_{hm}, and the Kirchhoff Current Law on node n_h (see Appendix B for a detailed deduction). Note that if the current mismatch at n_h is equal to zero (i.e., $\Delta\mathbb{i}_h^{(k)} = 0$), then $\mathbb{u}_{hm}^{(k)} = \mathbb{v}_h^{(k)}$. That is to say, the voltage according to component c_{hm} is the same as the one according to node n_h itself. This corresponds to a condition of convergence at component level.

Node model

The node is simply modelled as a point where different components exchange data and consolidate their results. The node iteration is defined as follows:

$$\mathbb{v}_h^{(k+1)} = \sum_{m \in \mathcal{I}_h} w_{hm} \mathbb{u}_{hm}^{(k)}, \tag{6.14a}$$

$$\Delta \mathbb{i}_h^{(k+1)} = \sum_{m \in \mathcal{I}_h} \mathbb{i}_{hm}^{(k)}, \tag{6.14b}$$

where w_{hm} is defined as the weight of component c_{hm} within node n_h. These weights satisfy the following equation:

$$\sum_{m \in I_h} w_{hm} = 1. \tag{6.15}$$

In other words, the voltage at each node is updated with the weighted sum of the voltages computed by each component connected to it. Note that if convergence is attained at component level (i.e., $\mathbb{u}_{hm}^{(k)} = \mathbb{v}_h^{(k)}, \forall m \in \mathcal{I}_h$), then $\mathbb{v}_h^{(k+1)} = \mathbb{v}_h^{(k)}$. This happens when all the components agree on the value of the node voltage, and corresponds to a condition of convergence at node level.

The next subsection shows how these weights can be chosen in order to guarantee convergence of the proposed iteration.

6.3.3 Convergence study

This section investigates the convergence properties of the proposed iterative method for two different synchronization schemes.

Let n_0 be the ground-node, and $\mathcal{N}'$ the set of all nodes in the circuit minus the ground. That is to say, $\mathcal{N}' = \mathcal{N} \setminus \{n_0\}$. Also, let $\mathbb{v} \in \mathbb{C}^{n-1}$ be the vector of voltages at all non-ground nodes. The iteration function, $\mathbb{f} : \mathbb{C}^{n-1} \to \mathbb{C}^{n-1}$, is given by:

$$\mathbb{f}_h(\mathbb{v}) = \mathbb{v}_h + \sum_{m \in \mathcal{I}_h} \sum_{j \in \mathcal{I}_h} w_{hm} z_{hm} \frac{\mathbb{v}_j - \mathbb{v}_h - \hat{\mathbb{v}}_{hj}}{z_{hj}}, \quad h \in \mathcal{N}'. \tag{6.16}$$

This expression is obtained by combining equations (6.13a), (6.13b), (6.14a), (6.14b) and (6.15). Equation (6.16) is linear, thus, it can be written as follows:

$$\mathbb{f}(\mathbb{v}) = \boldsymbol{L}\mathbb{v} + \mathbb{b}, \tag{6.17}$$

where $\boldsymbol{L}$ and $\mathbb{b}$ are given by:

$$\begin{aligned}
l_{hh} &= 1 - \sum_{m\in\mathcal{I}_h}\sum_{j\in\mathcal{I}_h} \frac{w_{hm}z_{hm}}{z_{hj}}, \\
l_{hj} &= \begin{cases} \sum_{m\in\mathcal{I}_h} \frac{w_{hm}z_{hm}}{z_{hj}} & \text{if } j \in \mathcal{I}_h, \\ 0 & \text{otherwise.} \end{cases} \\
\mathbb{b}_h &= -\sum_{m\in\mathcal{I}_h}\sum_{j\in\mathcal{I}_h} \frac{w_{hm}z_{hm}\hat{\mathbb{v}}_{hj}}{z_{hj}}.
\end{aligned} \tag{6.18}$$

The following theorem establishes that a totally asynchronous iteration on $\boldsymbol{f}(\cdot)$, as given by (6.17), is not guaranteed to converge.

Theorem 6.3.1. *Let $\mathbb{f}(\cdot)$ be defined by (6.17). Then, there exists an update function $\mathcal{U}(\cdot)$, a delay function $d(\cdot)$ and an initial guess $\mathbb{v}^{(0)}$, such that the associated asynchronous iteration on $\mathbb{f}(\cdot)$ does not converge to a fixed-point.*

Proof. See Appendix B.

The above result implies that to ensure convergence of the proposed iteration, further assumptions are needed. The following theorem establishes a sufficient condition for convergence based on partially asynchronous iterations.

Theorem 6.3.2. *Let $\mathbf{g}(\cdot)$ be defined by:*

$$\mathbf{g}(\mathbb{v}) = (1-\alpha)\mathbb{v} + \alpha\mathbb{f}(\mathbb{v}), \tag{6.19}$$

where $0 < \alpha < 1$ and $\mathbb{f}(\cdot)$ is given by (6.16). Let the weights, $w_{hm}, m \in I_h$, satisfy the following:

$$1 - \sum_{m\in\mathcal{I}_h}\sum_{j\in\mathcal{I}_h} \frac{w_{hm}z_{hm}}{z_{hj}} \geq 0, \quad \forall h. \tag{6.20}$$

Then, any partially asynchronous iteration over $\mathbf{g}(\cdot)$ converges to $\mathbb{v}^*$, fixed-point of $\mathbb{f}(\cdot)$.

Proof. See Appendix B.

6.4 Extension of the method to PF analysis

As discussed in Section 3.3, the main difference between solving an AC circuit and the PF analysis lies in the set of input data. Circuit analysis uses voltages and currents to characterize electrical components, whereas PF analysis is based on power injections and consumptions. Another difference comes from the topology. In circuit analysis, the ground node is explicitly defined, so that components are always connected between two nodes. In PF analysis, on the other hand, the ground is not represented. This allows components with one terminal to ground to appear as connected to only one node.

The proposed method for AC circuit analysis is extended to PF analysis by making slight modifications to the network and component models. These are described below.

6.4.1 Network model

The network model is modified by substituting current mismatches, $\Delta\mathbb{i}$, with complex power mismatches, $\Delta\mathbb{s}$. Similarly, current injections, $\mathbb{i}$, are replaced by complex power injections, $\mathbb{s}$. Therefore, instead of n_h communicating $\Delta\mathbb{i}_h$ to its attached components, it communicates $\Delta\mathbb{s}_h$. Also, instead of c_{hm} sending $\mathbb{i}_{hm}$ and $\mathbb{i}_{mh}$ to its terminal nodes, it sends $\mathbb{s}_{hm}$ and $\mathbb{s}_{mh}$. The following equations are utilized to make the proper adjustments:

$$\Delta\mathbb{i} = \left(\frac{\Delta\mathbb{s}}{\mathbb{v}}\right)^*, \tag{6.21}$$

$$\mathbb{i} = \left(\frac{\mathbb{s}}{\mathbb{v}}\right)^*, \tag{6.22}$$

where $(\cdot)^*$ indicates the complex conjugate.

To better reflect the power system topology, the ground node is taken out. Also, certain components are allowed to be connected to only one node, as described in the following subsection.

6.4.2 Component and node model

Component model

The proposed method for PF analysis considers three types of component: generator, branch and shunt. These components are described below.

Generator. It is connected to only one node, say n_h. The generator is there to ensure that the reactive power mismatch at the node is equal to zero. In this sense, the model assumes that generators are able to provide as much reactive power as needed in order to maintain the voltage. The slack generator also does the same for real power. Accordingly, the iteration is simply the following: for non-slack generators,

$$\mathrm{Im}(\mathbb{s}_h^{(k)}) = 0. \tag{6.23}$$

And for the slack generator,

$$\mathbb{s}_h^{(k)} = 0. \tag{6.24}$$

Branch. The branch is connected between two non-ground nodes, say n_h and n_m. It consists of a constant impedance, z_{hm}. The branch iteration is defined as follows:

$$\mathbb{u}_{hm}^{(k)} = \mathbb{v}_h^{(k)} + z_{hm}\left(\frac{\Delta \mathbb{s}_h^{(k)}}{\mathbb{v}_h^{(k)}}\right)^*, \tag{6.25a}$$

$$\mathbb{s}_{hm}^{(k)} = \mathbb{v}_h^{(k)}\left(\frac{\mathbb{v}_m^{(k)} - \mathbb{v}_h^{(k)}}{z_{hm}}\right)^*. \tag{6.25b}$$

The above expressions are obtained by combining equations (6.13a), (6.13b) and (6.21).

Shunt. As the generator, the shunt is connected to only one node, say n_h. It represents an impedance to ground, noted z_{h0} (the subscript '0' indicates the ground). The shunt iteration is similar to the branch one. The main difference is that $\mathbb{v}_m$ in equation (6.25b) is replaced by zero (the ground voltage). The iteration is the following:

$$\mathbb{u}_{h0}^{(k)} = \mathbb{v}_h^{(k)} + z_{h0}\left(\frac{\Delta \mathbb{s}_h^{(k)}}{\mathbb{v}_h^{(k)}}\right)^*, \tag{6.26a}$$

$$\mathbb{s}_{h0}^{(k)} = -\frac{|\mathbb{v}_h^{(k)}|^2}{(z_{hm})^*}, \tag{6.26b}$$

where $|\cdot|$ indicates the absolute value.

Node model

The node model is simply extended by replacing equation (6.14b) with:

$$\Delta \mathbb{s}_h^{(k+1)} = \sum_{m \in \mathcal{I}_h} \mathbb{s}_{hm}^{(k)}. \tag{6.27}$$

Also, a new iteration is added which allows PQ loads to become constant impedances, if the voltage at the node is outside specific limits. This is a technique used in traditional PF analysis [Milano 2010]. In the proposed method, the power injected to n_h by loads is noted $\mathbb{s}_{L,h}$. If the voltage is outside the limits, this variable is scaled by a factor depending on the current voltage and the violated limit. Then, the power mismatch at the node is adjusted to reflect the change in the load. In conclusion, the iteration is composed of two parts. The first updates the power injected by loads, as follows:

$$\mathbb{s}_{L,h}^{(k+1)} = \begin{cases} \left(\frac{v_h^{(k)}}{v_h^{min}}\right)^2 \mathbb{s}_{L,h}^{(k)} & \text{if } v_h^{(k)} < v_h^{min}, \\ \left(\frac{v_h^{(k)}}{v_h^{max}}\right)^2 \mathbb{s}_{L,h}^{(k)} & \text{if } v_h^{(k)} > v_h^{max}, \\ \mathbb{s}_{L,h}^{(k)} & \text{otherwise,} \end{cases} \tag{6.28}$$

where v_h^{min} and v_h^{max} are the voltage limits at n_h. Then, the second part updates the power mismatch, as follows:

$$\Delta \mathbb{s}_h^{(k+1)} = \Delta \mathbb{s}_h^{(k)} - \mathbb{s}_{L,h}^{(k)} + \mathbb{s}_{L,h}^{(k+1)}. \tag{6.29}$$

The use of the above iteration reduces instruction regularity, as it needs to be performed only by nodes. However, it may be required for convergence of the method, which makes it affordable from an overall point of view.

6.5 Implementation and evaluation

In this section, the proposed method for PF analysis is implemented in CUDA C++ and tested on the Nvidia Kepler GPU architecture [kep 2012]. As any GPU implementation, this one requires to take into account the peculiarities of GPU architecture, as described below. The proposed method is also evaluated on a series of benchmarks, with the aim of exploring convergence, performance and scalability issues.

6.5.1 Implementation

As explained in Section 2.3.1, regularity is one of the most powerful leverages for GPU performance. The above is demonstrated for the case of treefix computations in Chapter 4. Algorithm 6.1 presents a regular implementation of the method proposed in Section 6.4. The algorithm assigns one parallel thread to each component in the system. In instructions 4 and 5, each thread calculates the voltage at its component's terminals and updates the appropriate coordinates on vector $\mathbb{v}$. Next, in instructions 6 and 7, the thread calculates the power injections into each terminal and updates the appropriate coordinates on vector $\Delta\mathbb{s}$. The process is repeated after synchronizing all threads, until convergence is reached.

Algorithm 6.1 Intrinsically parallel algorithm for PF analysis.

Input: Vector of initial power mismatches, $\Delta\mathbb{s}$; component model parameters; tolerance for the error, ε.

Output: Vector of node voltages, $\mathbb{v}$.

```
1: $\mathbb{v} = \mathbf{1}\angle\mathbf{0}$                                # Initial guess
2: for all $h, m$: $c_{hm} \in \mathcal{C}$ in parallel do
3:   repeat
4:     calculate $\mathbb{u}_{hm}$ and $\mathbb{u}_{mh}$
5:     update $\mathbb{v}_h$ and $\mathbb{v}_m$
6:     calculate $\mathbb{s}_{hm}$ and $\mathbb{s}_{mh}$
7:     update $\Delta\mathbb{s}_h$ and $\Delta\mathbb{s}_m$
8:     synchronize threads
9:   until $|\Delta\mathbb{s}_h|, |\Delta\mathbb{s}_m| < \varepsilon$
```

By assigning threads to components, the amount of work per thread is almost the same, as components are always connected to either one or two nodes. If threads were assigned to nodes instead, then some threads would perform much more work than others, as nodes can be connected to any number of components.

The proposed implementation revolves around two classes, namely *Component* and *Node*, which provide all the necessary variables and routines to efficiently perform the iterations described in Section 6.4. As discussed in Section 2.2.2, coalesced global memory accesses can drastically improve GPU performance by reducing the number of memory transactions. In the proposed implementation, the above is achieved by using a 'structure of arrays' in the definition of the aforementioned classes.

As explained in Section 2.2.2 as well, GPU performance can be improved by using shared memory instead of global memory for recurrent memory operations. This is due to the fact that shared memory has a much lower latency (few cycles compared to hundreds of cycles). However, shared memory is only accessible to threads within one thread-block, whereas global memory is accessible to all threads. Accordingly, using shared memory in the proposed method allows the delay between threads to grow larger, which affects convergence as discussed in Section 6.3.3. In the proposed implementation, global memory is preferred over shared memory for sake of simplicity. An investigation of the trade-off between memory efficiency and convergence of the method is planned as future work. In particular, the framework of asynchronous iterations with *flexible communication*, as introduced in [Miellou 1998], seems useful for carrying out such study.

6.5.2 Evaluation

The implementation is evaluated using randomly generated benchmarks from 1,024 to 8,192 nodes. The benchmarks are obtained using the random radial network generator provided by Dome [Milano 2013]. The specifications of this random network generator are thoroughly detailed in Section 4.4.3.

Table 6.2 summarizes the hardware used in the tests. All source code is compiled using G++ 4.8.2 and CUDA 6.5.

Table 6.2: Test environment

Device	Clock rate (GHz)	Number of cores
Xeon E5645 CPU	2.4	12 (1 used)
Tesla K40c GPU	0.7	2,880
GeForce GTX 680 GPU	1.06	1,536

Convergence

As discussed in Section 6.2.3, partially asynchronous fixed-point iterations typically exhibit a sub-linear rate of convergence. This means that reducing the convergence tolerance below a small threshold may require a large number of iterations. This

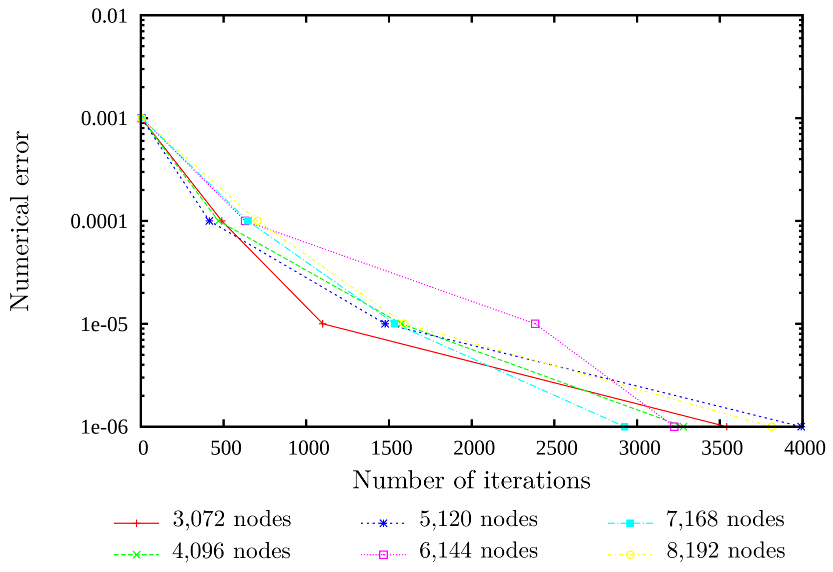


Figure 6.4: Numerical error as a function of the number of iterations on the K40c GPU.

paragraph investigates the relation between the convergence error and number of iterations for the proposed method using the K40c GPU.

Figure 6.4 shows the mismatch (on a logarithmic scale) as a function of the maximum number of iterations performed by any component. The different curves represent different problem sizes. Note that the behaviour is very similar on all 6 benchmarks, suggesting that convergence is independent from the problem size.

The convergence rate for this experiment is changing with the accuracy achieved. For example, reducing the error from 10^{-3} to 10^{-4} requires about 555 iterations in average. This corresponds to a convergence rate of 0.995 (i.e., each iteration scales the error by a factor of 0.995). Whereas reducing the error from 10^{-5} to 10^{-6} requires about 1,850 iterations. This represents a convergence rate of 0.998. That is to say, convergence is better when the error is higher. The farther the method is from the exact solution, the faster it approaches that solution. This is the opposite of the Newton-Raphson (NR) method where the convergence rate is improving at each iteration. From this point of view, it may be interesting to combine both approaches into a hybrid solution that switches methods depending on the error. For example, the proposed method can be used to obtain a first solution fast, and the NR method can be used to improve that solution afterwards. This idea will be investigated in future works.

It is worth noting that this experiment validates the theoretical study of Section 6.2.3, which shows that the convergence rate of this kind of methods approaches 1 in the long term.

Performance

This paragraph compares performance of the proposed approach with serial NR, the latter being provided by Dome [Milano 2013]. The Dome's implementation uses KLU, a highly optimized library for sparse matrix factorization, particularly suited to deal with matrices from circuit analysis [Davis 2010]. The proposed method is run on the Tesla K40c GPU, and the NR method is run on the Xeon E5645 CPU.

Figure 6.5 shows the execution time of both methods (on a logarithmic scale) as a function of the problem size, n. The NR execution time grows exponentially with the problem size, which is consistent with precedent results [Milano 2010]. In the proposed method, in turn, the execution time seems less affected by the size, and exhibits a more flat profile. At about 3,000 nodes, there is a turning point in which the proposed method becomes faster than NR.

It is worth noting that the tolerance for the error in this experiment is only of 10^{-4}. For lower tolerances, the execution time is likely to increase more in the proposed method than in NR (as the former converges sub-linearly, whereas the latter converges quadratically). Thus, as the tolerance decreases, the turning point should move to the right of the plot. The final position of the point will depend on other factors as well, e.g., quality of both implementations, network topology, computer architectures. However, given the differences in the slope observed in Fig. 6.5, it is expected to see a turning point in most comparable scenarios.

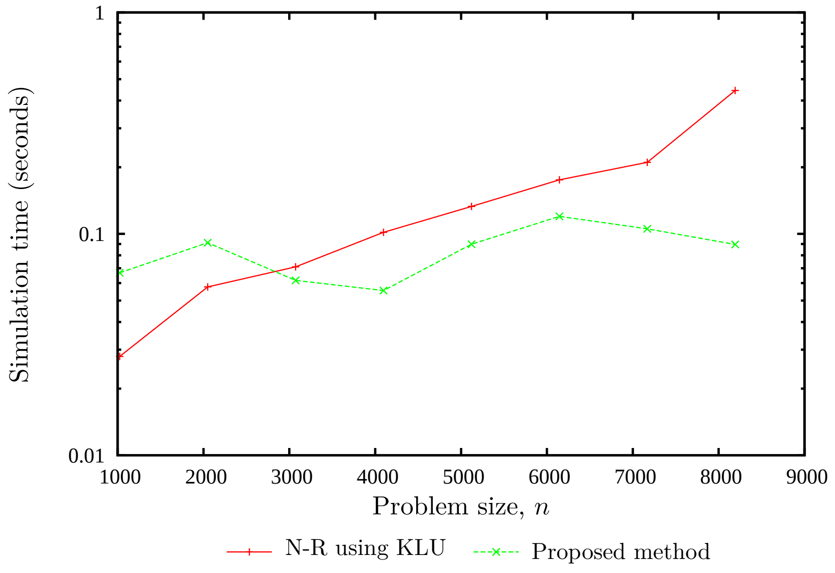


Figure 6.5: Related performance of N-R and the proposed method (accuracy: 10^{-4}).

Scalability

As mentioned in Section 6.1, performance of intrinsically parallel methods should increase with the number of computing cores. The above is confirmed by Fig. 6.6, which shows the execution time of the proposed method on two Nvidia Kepler GPU architectures as a function of the problem size. The Tesla K40c GPU, embedding 2,880 computing cores, allows for a performance gain of about 1.3 over the GTX 680, which has 1,536 cores. The ratio between the number of cores is of 1.8, which means that the proposed method almost achieve strong scalability.

It is noteworthy that both architectures are running binaries issued from the same source code. That is to say, the program is not adjusted to fit either architecture in particular, e.g., by using grid and block sizes that maximize occupancy in one or another. Different gains could be observed depending on such details.

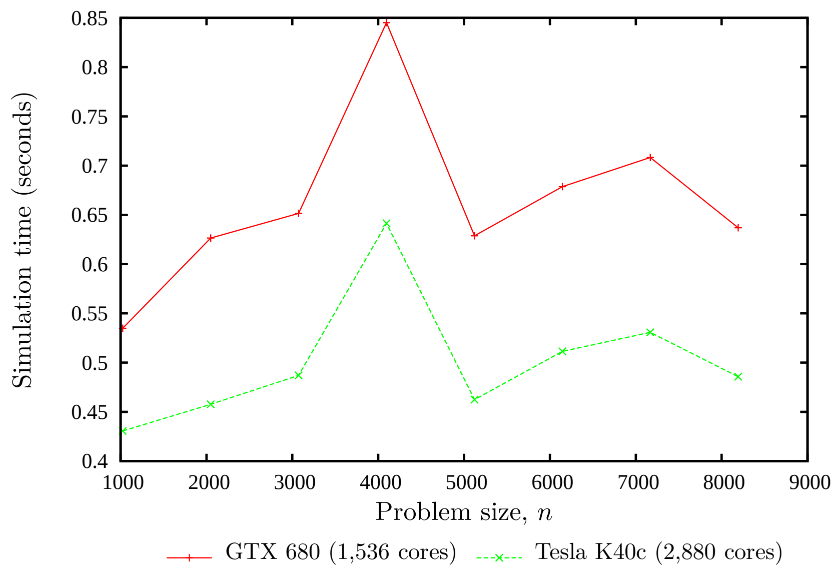


Figure 6.6: Performance of the proposed method on two GPU architectures.

6.6 Conclusions

This chapter presents a partially asynchronous fixed-point method for AC circuit analysis and PF analysis. The method is designed to be implemented on SIMT architectures such as the GPU. A theoretical result is provided which shows that the method is certain to converge, although convergence is generally sub-linear. The proposed method is implemented in CUDA and tested on different GPU architectures, using a set of radial randomly generated benchmarks. Simulation results show that: (i) the convergence rate of the proposed method tends to be better at an early stage, which makes it a good complement of the traditional NR method; (ii) beyond a certain problem size, which depends on several parameters including architectural and implementation features, the proposed method becomes faster than the traditional NR approach; (iii) the proposed method scales well with the number of computing cores in the parallel architecture, allowing the application to automatically take advantage of newer GPU architectures.

Future works will consider the improvement of the proposed CUDA implementation with aims at reducing simulation time. In particular, the use of shared memory will be investigated. The above requires addressing convergence of the method in the case where an increasing number of iterations are performed ‘asynchronously’, i.e., without communicating the results to all threads. Also, the method will be evaluated on a larger range of networks with various sizes and topologies, and compared with other solution techniques such as the Backward-Forward Sweep. Finally, the adaptation of additional types of loads and generators will be considered (e.g., renewable energy generators, storage units, electric vehicle chargers, and others).

References

[Amdahl 1967] G. M. Amdahl. *Validity of the single processor approach to achieving large scale computing capabilities.* Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, pp. 483–485. ACM, 1967. (Cited on page 109.)

[Bertsekas 1987] D. P. Bertsekas and D. El Baz. *Distributed asynchronous relaxation methods for convex network flow problems.* SIAM Journal on Control and Optimization, vol. 25, no. 1, pp. 74–85, 1987. (Cited on page 109.)

[Chazan 1969] D. Chazan and W. Miranker. *Chaotic relaxation.* Linear algebra and its applications, vol. 2, no. 2, pp. 199–222, 1969. (Cited on page 111.)

[Davis 2010] T. A. Davis and E. Palamadai Natarajan. *Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems.* ACM Transacctions on Mathematical Software, vol. 37, no. 3, pp. 36:1–36:17, 2010. (Cited on pages 10 and 124.)

[El-Baz 1996] D. El-Baz, P. Spiteri, J.-C. Miellou and D. Gazen. *Asynchronous iterative algorithms with flexible communication for nonlinear network flow problems.* Journal of Parallel and Distributed Computing, vol. 38, pp. 1–15, 1996. (Cited on page 109.)

[Frommer 2000] A. Frommer and D. B. Szyld. *On asynchronous iterations.* Journal of Computational and Applied Mathematics, vol. 123, no. 1–2, pp. 201 – 216, 2000. Numerical Analysis 2000. Vol. III: Linear Algebra. (Cited on page 110.)

[Garcia 2010] N. Garcia. *Parallel Power Flow solutions using a biconjugate gradient algorithm and a Newton method: A GPU-based approach.* IEEE Power and Energy Society General Meeting, pp. 1–4, 2010. (Cited on page 109.)

[Gupta 1997] A. Gupta, G. Karypis and V. Kumar. *Highly scalable parallel algorithms for sparse matrix factorization.* IEEE Transactions on Parallel and Distributed Systems, vol. 8, no. 5, pp. 502–520, 1997. (Cited on page 109.)

[Honkala 2001] M. Honkala, J. Roos and M. Valtonen. *New multilevel Newton-Raphson method for parallel circuit simulation.* Proceedings of European Conference on Circuit Theory and Design, vol. 1, pp. 113–116, 2001. (Cited on page 109.)

[kep 2012] *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, 2012. (Cited on page 121.)

[Lubachevsky 1986] B. Lubachevsky and D. Mitra. *A chaotic asynchronous algorithm for computing the fixed-point of a nonnegative matrix of unit spectral radius.* Journal of the ACM, vol. 33, no. 1, pp. 130–150, 1986. (Cited on page 113.)

[Marin 2015] M. Marin, D. Defour and F. Milano. Linear circuit analysis based on parallel asynchronous fixed-point method. 2015. (Cited on page 108.)

[Miellou 1998] J. Miellou, D. El Baz and P. Spiteri. *A new class of asynchronous iterative algorithms with order intervals.* Mathematics of Computation of the American Mathematical Society, vol. 67, no. 221, pp. 237–255, 1998. (Cited on page 122.)

[Milano 2010] F. Milano. Power system modelling and scripting. Springer Science & Business Media, 2010. (Cited on pages 7, 10, 37, 120 and 124.)

[Milano 2013] F. Milano. *A Python-based software tool for power system analysis.* IEEE Power and Energy Society General Meeting, pp. 1–5, 2013. (Cited on pages 7, 10, 68, 98, 122 and 124.)

[Nagel 2013] I. Nagel. *Analog microelectronic emulation for dynamic power system computation.* PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2013. (Cited on page 109.)

[Newton 1984] A. R. Newton and A. L. Sangiovanni-Vincentelli. *Relaxation-based electrical simulation.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 3, no. 4, pp. 308–331, 1984. (Cited on pages 114 and 147.)

[Talukdar 1983] S. Talukdar, S. S. Pyo and R. Mehrotra. Designing algorithms and assignments for distributed processing. Electric Power Research Institute, 1983. (Cited on page 109.)

[Tseng 1990] P. Tseng, D. P. Bertsekas and J. N. Tsitsiklis. *Partially asynchronous, parallel algorithms for network flow and other problems.* SIAM Journal on Control and Optimization, vol. 28, no. 3, pp. 678–710, 1990. (Cited on pages 112 and 113.)

[Tsitsiklis 1984] J. N. Tsitsiklis. *Problems in Decentralized Decision making and Computation.* Technical report, DTIC Document, 1984. (Cited on page 109.)

[Tu 2002] F. Tu and A. J. Flueck. *A message-passing distributed-memory Newton-GMRES parallel Power Flow algorithm.* IEEE Power Engineering Society Summer Meeting, vol. 3, pp. 1477–1482, 2002. (Cited on page 109.)

Chapter 7
Conclusions

Contents

7.1 Overview

This thesis addresses the application of Graphics Processing Units (GPUs) to the Power Flow (PF) analysis of modern power systems. The aim is twofold: (i) to develop a new understanding of GPUs in the context of challenging problems, where typical GPU performance leverages are not straightforward to implement; and (ii), to enhance PF analysis for its application to the Smart Grid concept, with new standards for performance and accuracy.

GPUs are generally utilized in the field of High Performance Computing (HPC) to accelerate several computational intensive applications, some of them involved in PF analysis. At the same time, accelerating PF analysis was recognized as a fundamental challenge for the efficient operation of modern power systems, and GPU computing was identified as an efficient tool for accelerating computations. However, several questions regarding specific aspects of a GPU-based solution for PF analysis remained unanswered. The present investigation attempts to answer some of these questions. The questions addressed in this thesis are: which available methods for PF analysis are best suited to GPU computing? What kind of representation formats are most adapted to both power system modelling and GPU implementation? How to apply the knowledge of various GPU features to develop novel PF algorithms?

Answering these questions requires a balanced understanding of both GPU computing and PF analysis. One is incomplete without the other. Accordingly, Chapters 2 and 3 of this thesis revisit the state-of-the-art of each field. In particular, Chapter 2 introduces GPU architecture by reviewing the execution model, memory hierarchy and main programming frameworks. Chapter 2 also presents relevant aspects of GPU programming such as regularity and latency hiding. In turn, Chapter 3 introduces PF analysis through a review of the classic circuit analysis problem. Then, main methods for PF analysis are classified according to network topology. Additionally, some fundamental approaches to handle uncertainty in PF analysis are described.

Chapters 4, 5 and 6 present original research regarding the application of GPUs to PF analysis. Each chapter explores a different idea to enhance the PF analysis, through a specific set of GPU-based programming strategies. Chapter 4 is devoted to the so-called 'treefix' operations and their implementation on GPUs. Treefix operations are useful primitives defined over a tree structure, that can be used to implement the BFS algorithm for the analysis of radial power networks, among other applications. Treefix operations have been traditionally performed by utilizing mainly two different approaches. The first is a vectorial approach, which exhibits a highly regular pattern but doubles the amount of arithmetic operations and memory required. The second is a parallel approach, that performs the minimal amount of arithmetic operations but requires load-balancing to handle the irregularity of the tree structure. Both approaches are well-suited for GPU computing. However, no comparative study of their behaviour on modern GPU architectures has been presented before this thesis. Moreover, no implementation of the BFS algorithm using GPU-based treefix computations has been provided. Chapter 4 provides such comparison of treefix approaches and the correspondent BFS implementation.

Chapter 5 explores the implementation on GPUs of fuzzy interval algebra. Fuzzy intervals are mathematical entities used for modelling uncertainty in numerical analyses, such as PF analysis. Fuzzy interval analysis extends classic interval analysis to multiple levels of uncertainty through the concepts of membership function and α-cuts. The α-cuts, which represents different levels of uncertainty in the model, can be represented using classic interval representation formats such as lower-upper and midpoint-radius. Depending on the membership function, different representation formats may have a specific impact over performance. However, most available fuzzy arithmetic implementations are based on classic interval libraries that only provide the lower-upper format. Chapter 5 expands this vision by implementing both formats into a library of fuzzy interval arithmetic, callable from either CPU and GPU architectures. Subsequently, the impact of the hardware and representation format over performance is measured for benchmark fuzzy interval applications.

Chapter 5 addresses uncertainty modelling in PF analysis. In the literature, the above has been done by two methods. The first is the probabilistic Monte Carlo approach, i.e., running several PF instances using samples of random variables obtained from statistical data. Afterwards, the results are aggregated into some relevant figures. The second is the interval approach, i.e., using interval analysis to solve the PF equations with interval data. Both approaches have severe drawbacks. The former is very inefficient, as it requires thousands of PF analyses to provide accurate results. The latter tends to be very inaccurate, as interval computations tend to overestimate results in long computations such as PF analysis. Chapter 5 proposes a novel interval approach based on midpoint-radius fuzzy intervals. The method is based on decoupling the calculation of midpoint and radius, which allows reducing complexity and increasing accuracy of numerical results.

Finally, Chapter 6 discusses the use of asynchronous iterations in a novel fixed-point method for PF analysis. The asynchronous iteration model allows several threads to carry out a computation with no rigid synchronization schemes. In this way, performance scales with the number of computing cores offered by the architecture. However, most available PF methods, including Newton-Raphson (NR), are not well suited to the asynchronous model. The above is due to strong data dependencies, which result in several synchronization points. Chapter 6 proposes a novel fixed-point method to fix this problem and take full advantage of the asynchronous framework.

The remainder of the chapter is organized as follows. Section 7.2 summarizes the empirical findings of this thesis. Section 7.3 discusses the theoretical implications of such findings. Limitations of the study are presented in Section 7.4, and prospects for further research are outlined in Section 7.5. Finally, Section 7.6 concludes the chapter and the thesis.

7.2 Empirical findings

This section summarizes the empirical findings obtained in this thesis. These are chapter-specific, as presented below.

7.2.1 Treefix operations on the GPU

A comparison of vectorial and parallel treefix algorithms discussed in the previous section is provided. The study is based on measuring performance and accuracy of two GPU implementations, one for each approach. The performance of each implementation is evaluated using a series of relevant benchmark trees. Results show that the vectorial approach, which exploits regularity to enhance GPU performance, is always faster than the parallel approach based on load-balancing. It is also shown that performance in the vectorial approach is exclusively driven by the size of the tree. The larger the tree, the longer the algorithm takes to complete, independently of any other factor, which is an important property for performance predictability. Whereas in the parallel approach, performance is also affected by the tree topology. Broader trees take less time to sweep as there are more vertices for the algorithm to visit in parallel.

Accuracy is evaluated using a random topology and several sets of input data, with different condition numbers. The condition number is used to determine the numerical difficulty to provide accurate results in floating-point computations. Results show that the maximum relative error among all vertices tends to be lower in the parallel algorithm. In comparison, the vectorial algorithm loses up to two bits of accuracy as it generally needs more floating-point operations to compute the result.

A novel implementation of the BFS algorithmm, based on the vectorial approach, is compared with the sequential BFS routine provided by the software Dome. Both implementations of BFS are tested using a series of randomly generated benchmark trees, measuring the speedup of the vectorial version over the sequential one. The speedup ranges from 2 to 9, depending on two main factors: (i) the size of the network, and (ii) whether a new OpenCL context needs to be created, or a previously created one can be reused.

7.2.2 Fuzzy arithmetic on the GPU

A library of fuzzy interval arithmetic in lower-upper and midpoint-radius formats is written in CUDA and made available online. Using this library, performance of both formats is compared. To this purpose, two benchmarks application are designed: one compute-bound and one memory-bound. The compute-bound is a classic ‘axpy’ loop, and the memory-bound benchmark is a radix sort. Midpoint-radius outperforms lower-upper by a factor of two to twenty in both cases, which is consistent with the observation made in the previous section.

The proposed method for fuzzy PF analysis is compared with the Monte Carlo approach. Both methods are run over a fuzzified version of the IEEE-57 bus test case system, assuming a 10% variation over the central value in all power injections and consumptions. A total of 5,000 uniformly distributed samples are obtained to implement the Monte Carlo approach. The results of both methods are very similar, whereas the proposed technique has a much lower computational cost.

7.2.3 Asynchronous PF analysis

The proposed fixed-point method for PF analysis is implemented in CUDA. The implementation is tested on modern GPU architectures using a series of randomly generated radial networks, from about 2,000 to 8,000 nodes. The aim is to study convergence, accuracy, performance and scalability of the proposed approach.

A first series of tests show that the convergence rate decreases as the method approaches the exact solution. In other words, it is increasingly harder for the method to reduce the approximation error. However, during the early iterations the error is reduced quite fast, which means that a first (although not too accurate) approximation is easy to compute.

A second experiment shows that the number of iterations does not vary too much with the number of nodes in the power network. This suggests that performance of the proposed method is independent from the problem size. As a consequence, there is a turning point where the proposed approach becomes faster than traditional

PF methods, such as NR, which require increasing amounts of time to analyze increasingly larger networks.

Finally, a third experiment provides a measurement of the scalability of the solution. The proposed approach is tested over two different GPUs of the same architecture (Nvidia Kepler), measuring execution time. The application runs faster on this architecture when more cores are used. The above demonstrates that performance scales with the number of threads, and thus the proposed method can benefit for architectural improvements without major changes in the algorithm.

7.3 Theoretical implications

This section synthesizes the theoretical implications of the findings discussed above.

7.3.1 For GPU computing

Regularity versus load-balancing

Chapter 4 compares the efficacy of regularity and load-balancing in the solution of the treefix problem on GPUs. We show that regularity is always a better choice for performance, independently of the tree size and topology. This is an interesting result, considering that the regular algorithm actually requires more arithmetic operations than the one using load-balancing. The GPU appears to compensate the cost of extra arithmetic operations and data transfer by exploiting the SIMT execution model. The GPU rewards regularity highly, as discussed in Section 2.3.1.

The above illustrates an interesting fact: despite the natural intuition, GPU applications do not require to be work-efficient for achieving the best performance. The cost of extra operations is affordable and even meaningless, provided that they enhance regularity of the solution. In this way, applications can be designed with regularity in mind, rather than trying to achieve work-efficiency. This level of design is only reachable by understanding both the computer architecture and the considered algorithm. Dedicating most of the efforts to only one of the two aspects is clearly an unreliable strategy.

Impact of the representation format

Chapter 5 compares lower-upper and midpoint-radius formats in symmetric fuzzy interval operations performed on the GPU. The results show that midpoint-radius outperforms lower-upper by a factor of 2 to 20, in both compute-bound and memory-bound applications. It is interesting that such performance gain is only achieved by using a dedicated representation format. No changes to the fuzzy interval model are introduced. The representation format has a considerable impact over performance of symmetric fuzzy interval operations. The above has a strong implication for design: even if different representation formats yield the same result, one of them may be more adapted to the scenario. In the case of symmetric fuzzy interval algebra, this is true for the midpoint-radius format. The choice of the representation format can enhance several aspects of the implementation, beyond the final result.

Architecture-specific design

Chapter 6 proposes a novel PF method based on asynchronous fixed-point iterations. The method is implemented in CUDA and tested on state-of-the-art GPU architectures. The results show several interesting properties:

- the method is efficient at reducing the error during the first iterations, quickly providing a first approximation of the solution;
- performance of the method is not too dependant on the problem size, unlike traditional approaches such as NR;
- performance scales well with the number of computing cores available in the architecture, almost reaching strong scalability.

All these good properties are a natural consequence of the algorithm design. The proposed approach is conceived with the architecture in mind, knowing that the framework of asynchronous iterations is able to efficiently exploit the SIMT execution model of GPUs. In this way, the application could immediately benefit of architectural improvements expected from future GPUs. For example, as observed in the empirical tests, the increase in the number of computing cores allows larger networks to be analyzed without increasing the simulation time. A similar situation was observed in the past with serial computing, where improvements in CPU frequency immediately implied that single-core applications would run faster. The proposed approach achieves that kind of behaviour, now in the domain of parallel computing.

7.3.2 For Power System Analysis

Accelerated radial network analysis

Chapter 4 presents a novel implementation of the BFS algorithm which relies on the vectorial treefix approach. A cases study shows a speed-up of 2 to 9 achieved by the proposed implementation over the traditional sequential approach. The speed-up is higher on larger networks, i.e., the impact of the vectorial approach is more appreciable as the network grows. The speed-up is also higher if the application reuses an already available OpenCL context, as the platform and context configuration process can be skipped for that run.

The above illustrates the advantages and limitations of using GPU computing for this type of analyses. On one hand, there is the natural acceleration coming from the use of parallel computing on a dedicated architecture. On the other hand, there are the issues related to the practical implementation, e.g., platform configuration, data transfers, which ultimately reduce the overall gain. Applications that implement the parallel or vectorial approach must be aware of such opportunities and issues, in order to exploit the former while trying to overcome the latter.

Fuzzy PF solutions

Chapter 5 proposes a novel fuzzy PF analysis method based on symmetric fuzzy intervals and the midpoint-radius format. The accuracy of the proposed method is similar to that of the Monte Carlo approach, whereas its computational complexity is much lower. This improves previous attempts in the area of interval PF analysis, where the radius of the result tends to explode. It is shown that interval computations can be useful if some measures are applied to prevent radius overestimation.

The proposed technique provides a novel criterion of validity for the solution of the interval PF analysis. This condition establishes that a valid solution is one that considers all the variability specified in the inputs, even if it includes non-possible scenarios (overestimation of the result). The best solution is the one that includes the least amount of non-possible scenarios. In the proposed method, this solution is obtained by solving an optimization problem, where the objective function is a measure of the radius length and the constraints are the validity criterion expressed above.

Intrinsically parallel simulation

Chapter 6 proposes a software-based intrinsically parallel PF simulator. This solution overcomes the serialization problem that affects most existing PF analysis methods (such as NR) according to the Amdahl's law. The use of intrinsically parallel methods appears as a promising technique, as computer architectures evolve towards massive parallelism. The fact that most physical systems, particularly power systems, are themselves 'intrinsically parallel' makes the situation even more favourable. Several physical phenomena can be modelled using very simple mathematical expressions, such as the Kirchhoff's circuit laws. The problem comes from the aggregation of all these expressions into systems of equations for their centralized solution. Using intrinsically parallel methods bypasses this difficulty and recovers the simplicity and originality of the physical phenomenon. The method proposed in Chapter 6 admits the inclusion of almost any electrical device, with the only requirement that its behaviour can be modelled using voltages and power flows.

7.4 Limitations of the study

In Chapter 4, the performance comparison of different treefix approaches does not consider the time in obtaining the tree representation for each algorithm. In the case of the vectorial algorithm, this corresponds to the vector tree representation, computed using Euler-tour ordering. In the case of the parallel algorithm, it corresponds to the adjacency lists. The rationale behind this decision is that most treefix-based applications do not need to compute the input tree representation very often. Indeed, these applications rely on sweeping the input tree several times, typically in an iterative scheme. That is to say, the tree representation is computed as part of the algorithm setup, and then re-used several times for each instance of treefix. Therefore, the time in computing the representation is not critical.

In Chapter 5, the midpoint-radius format is used to implement symmetric fuzzy intervals. In the case of interval multiplication, this format introduces an overestimation of up to 1.5 over lower-upper. The overestimation depends on the precision of the interval, i.e., the ratio of the radius to the midpoint. The lower this ratio, the less the overestimation. In PF analysis, where typically the variability is small with respect to the central value, the above should not be a big issue.

Finally, in Chapter 6, the proposed asynchronous method for PF analysis is tested using radial networks of about 2,000 to about 8,000 nodes. This is acceptable to have an idea of the method behavior. However, more testing needs to be done in order to fully validate the proposed approach.

7.5 Recommendations for future research

Based on the results of the present research, some future work directions are outlined next:

- The accuracy analysis of Chapter 4 shows that the vectorial treefix algorithm loses up to 2 bits of accuracy with respect to the parallel one. This is due to extra operations needed to ensure regularity, which introduce additional rounding errors. In fact, computing the result for a given vertex requires as many operations as there are steps in the Euler-tour up to that vertex. Thus, using a different Euler-tour ordering (e.g., swapping branches according to the size of the sub-tree) may have a positive impact on the representation and the accuracy.

- Chapter 4 proposes a novel implementation of the BFS method using a vectorial treefix approach. Results show that the speedup achieved over the sequential version is significant, especially if the same OpenCL context can be reused in multiple runs. The above can be exploited by any application that utilizes the BFS algorithm as building block. Two examples are particularly relevant: (i) admission control, which requires to run several instances of PF analysis over slightly different networks, and (ii) the compensation-based method for weakly meshed networks, which splits the network into several trees and applies BFS over each one.

- Chapter 5 shows that the midpoint-radius format achieves a speed-up of 2 to 20 over traditional lower-upper. However, the above requires fuzzy intervals to be symmetric, which may not be the case in some scenarios. To overcome this, one alternative is to use symmetric envelopes of asymmetric membership functions. This would allow the midpoint-radius approach to be applicable at the cost of losing accuracy in some of the α-cuts.

- Chapter 5 proposes a fuzzy PF analysis method that compares well in terms of accuracy with the reputed Monte Carlo approach. However, as the uncertainty grows, the proposed method has more trouble in finding the correct interval bounds. The above can be explained by the choice of the 2-norm in measuring the length of the radius vector. A norm of a 'higher rank', such as the ∞-norm, could improve this situation at the expense of increasing the complexity of the algorithm.

- The method proposed in Chapter 5 illustrates the advantages of using the interval approach for PF analysis. The technique can be extended to other analyses which may benefit from the interval approach as well, e.g., the state-estimation problem, Optimal Power Flow, or even time domain integration.

- Chapter 6 proposes a novel approach for PF analysis based on asynchronous fixed-point iterations. The approach is well suited to parallel implementation over SIMT architectures such as the GPU. This is a promising technique given the current trend in GPU architectures design, where the number of computing cores increases with each new generation. The proposed solution can be complemented with models of additional electrical components. Also, the implementation can be optimized by using shared memory for local iterations within thread-blocks.

- The PF simulator proposed in Chapter 6 relies on classic circuit theory results. The above means that a circuit simulator could also be implemented using the proposed intrinsically parallel approach. Therefore, the approach could be extended to circuit and dynamic analysis, probably preserving most of its interesting properties.

7.6 Final remarks

This thesis proposes a number of approaches to improve different aspects of PF analysis using specific GPU programming strategies. The proposed methods are tested on modern GPU architectures and compared with state-of-the-art PF solutions. The results allow for a novel understanding of various GPU features. They also show what kind of improvements can be made over PF analysis in the context of novel requirements from the Smart Grid concept.

Appendix A

Mathematical reminders

This appendix recalls a series of classic mathematical definitions and theorems utilized in the body of the thesis. The definitions and theorems are presented in no particular order.

A.1 Definitions

Definition A.1.1 (Jacobian matrix)**.** Let $x \in \mathbb{R}^n$, and $\boldsymbol{f}(x)\colon \mathbb{R}^n \to \mathbb{R}^m$, be a differentiable function. Then, the matrix given by,

$$\boldsymbol{J_f} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}, \tag{A.1}$$

is called Jacobian matrix of $\boldsymbol{f}(x)$.

Definition A.1.2 (Random variable)**.** Let Ω be the set of possible outcomes of an experiment. Then, the function, $X\colon \Omega \to \mathbb{R}$, which maps every element in Ω into a real number, is called random variable.

Definition A.1.3 (Distribution and density functions)**.** Let X be a continuous, real valued random variable. Let $\Pr(\cdot)$ be a measure of probability. Then, the function,

$$F_X(x) = \Pr\left(X \leq x\right), \tag{A.2}$$

is called distribution function of X. The function,

$$f_X(x) = \frac{dF_X}{dx}, \tag{A.3}$$

is called probability density function.

Definition A.1.4 (Convolution product)**.** Let $f(\cdot)\colon \mathbb{R} \to \mathbb{R}$ and $g(\cdot)\colon \mathbb{R} \to \mathbb{R}$, be two differentiable functions. Then,

$$(f \otimes g)(x) = \int_{-\infty}^{\infty} f(y)g(x-y)dy, \tag{A.4}$$

is called the convolution product of $f(\cdot)$ and $g(\cdot)$.

Definition A.1.5 (p-norm)**.** Let $\boldsymbol{x} = (x_0, \dots, x_{n-1}) \in \mathbb{R}^n$ and $p \geq 1$. Then,

$$\|\boldsymbol{x}\|_p = (|x_0|^p + \dots + |x_{n-1}|^p)^{\frac{1}{p}} \tag{A.5}$$

is called the p-norm of $\boldsymbol{x}$.

Definition A.1.6 (Spectral radius)**.** Let $\boldsymbol{A} \in \mathbb{C}^{n \times n}$, with eigenvalues $\lambda_i, 1 \leq i \leq n$. Then,

$$\rho(\boldsymbol{A}) = \max_{1 \leq i \leq n} |\lambda_i|, \tag{A.6}$$

is called the spectral radius of $\boldsymbol{A}$.

Definition A.1.7 (Reducible matrix)**.** Let $\boldsymbol{A} \in \mathbb{C}^{n \times n}$. If there exists $\boldsymbol{P} \in \mathbb{C}^{n \times n}$, such that $\boldsymbol{P}^T \boldsymbol{A} \boldsymbol{P}$ is of the form:

$$\begin{bmatrix} \boldsymbol{A}_1 & \boldsymbol{A}_{12} \\ \boldsymbol{0} & \boldsymbol{A}_2 \end{bmatrix},$$

where $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ are square matrices of size at least 1, then $\boldsymbol{A}$ is called reducible.

Definition A.1.8 (Digraph of a matrix)**.** Let $\boldsymbol{A} \in \mathbb{C}^{n \times n}$. Let $D = (V, A)$ be a directed graph, such that $v_i v_j \in A$ if and only if $a_{ij} \neq 0$. Then $\boldsymbol{D}$ is called the digraph of $\boldsymbol{A}$, and noted $D = D(\boldsymbol{A})$.

Definition A.1.9 (Strongly connected)**.** Let $D = (V, E)$, be a directed graph, and $v_i, v_j \in V$. If there are directed walks from v_i to v_j and from v_j to v_i, then D is strongly connected.

A.2 Theorems

Theorem A.2.1 (Bound for the absolute error in summation algorithms)**.** *Let $x_1, \dots, x_n \in \mathbb{R}$ and $S_n = \sum_{i=1}^n x_i$. Let $\hat{S}_n$ be an approximation to S_n computed by a summation algorithm, which performs exactly $n-1$ floating-point additions. Let $\hat{T}_1, \dots, \hat{T}_{n-1}$, be the $n-1$ partial sums computed by such algorithm. Then,*

$$E_n := \left|S_n - \hat{S}_n\right| \leq \varepsilon \sum_{i=1}^{n-1} \left|\hat{T}_i\right|, \tag{A.7}$$

where ε is the relative rounding error.

Proof. See [Higham 2002].

Theorem A.2.2 (Necessary and sufficient condition for inclusion of intervals.)**.** *Let $[x] = \langle \check{x}, \rho_x \rangle$ and $[y] = \langle \check{y}, \rho_y \rangle$. Then,*

$$[x] \subset [y] \iff |\check{y} - \check{x}| < \rho_y - \rho_x. \tag{A.8}$$

Proof. See [Neumaier 1990].

Theorem A.2.3 (Bound for the spectral radius)**.** *Let $\boldsymbol{A} \in \mathbb{C}^{n\times n}$, and let $|\boldsymbol{A}|$ be the matrix of absolute values of the entries of $\boldsymbol{A}$. Then $\rho(\boldsymbol{A}) \leq \rho(|\boldsymbol{A}|)$.*

Proof. See [Horn 2012].

Theorem A.2.4 (Irreducibility and matrix digraph)**.** *Let $\boldsymbol{A} \in \mathbb{C}^{n\times n}$. Then $\boldsymbol{A}$ is irreducible if and only if $\boldsymbol{D}(\boldsymbol{A})$ is strongly connected.*

Proof. See [Horn 2012].

References

[Higham 2002] N. J. Higham. Accuracy and stability of numerical algorithms. Siam, 2002. (Cited on pages 66, 68, 77, 84 and 139.)

[Horn 2012] R. A. Horn and C. R. Johnson. Matrix analysis. Cambridge University Press, 2012. (Cited on page 140.)

[Neumaier 1990] A. Neumaier. Interval methods for systems of equations, vol. 37. Cambridge university press, 1990. (Cited on pages 76 and 139.)

Appendix B

Proof of theorems and propositions

This appendix provides the proofs of novel propositions and theorems presented in the body of the thesis.

Proposition 5.3.1. *Let $[\tilde{x}]$ be a symmetric fuzzy interval with kernel $\check{x}$. Let $N \in \mathbb{N}$, $i \in \{1, \ldots, N\}$, and $[x_i] = \langle \check{x}_i, \rho_{x,i} \rangle$, an α-cut. Then,*

$$[\tilde{x}] \text{ is symmetric} \iff \check{x}_i = \check{x},\ \forall\, i \in \{1, \ldots, N\}.$$

Proof. Let $\mu(\cdot)$ be the membership function of $[\tilde{x}]$. The reciprocal is proven, i.e.,

$$[\tilde{x}] \text{ is non-symmetric} \iff \exists\, j \in \{1, \ldots, N\},\ \check{x}_j \neq \check{x}. \tag{B.1}$$

i) ($\Longrightarrow$)
If $[\tilde{x}]$ is non-symmetric, then there is $x_0 \in \mathbb{R}$ such that,

$$\mu(\check{x} - x_0) \neq \mu(\check{x} + x_0). \tag{B.2}$$

Let $j \in \{1, \ldots, N\}$ and $[x_j] = [\underline{x}_j, \overline{x}_j]$ be an α-cut, such that,

$$\underline{x}_j = \check{x} - x_0. \tag{B.3}$$

Let $\check{x}_j$ be the midpoint of $[x_j]$ and assume,

$$\check{x}_j = \check{x}. \tag{B.4}$$

Then,

$$\overline{x}_j = 2\check{x}_j - \underline{x}_j = 2\check{x} - \underline{x}_j = \check{x} + x_0, \tag{B.5}$$

where the first equality is due to the definition of midpoint, the second to the assumption in (B.4), and the third to (B.3). Now, by definition of α-cut,

$$\mu(\underline{x}_j) = \mu(\overline{x}_j), \tag{B.6}$$

and replacing equations (B.3) and (B.5) above,

$$\mu(\check{x} - x_0) = \mu(\check{x} + x_0), \tag{B.7}$$

which contradicts equation (B.2). Hence, the assumption in (B.4) must be false.

ii) ($\Longleftarrow$)
Let $[x_j] = [\underline{x}_j, \overline{x}_j]$, and $x_0 \in \mathbb{R}$, such that,

$$\underline{x}_j = \check{x} - x_0. \tag{B.8}$$

If $\check{x}_j \neq \check{x}$, then,

$$\overline{x}_j = 2\check{x}_j - \underline{x}_j \neq 2\check{x} - \underline{x}_j = \check{x} + x_0, \tag{B.9}$$

where the first equality is due to the definition of midpoint, and the last to equation (B.8). Now, by construction, both $\overline{x}_j$ and $\check{x} + x_0$ are in the interval $[\check{x}, \overline{x}_j]$, where μ is strictly monotonous. Then,

$$\mu(\overline{x}_j) \neq \mu(\check{x} + x_0). \tag{B.10}$$

And, by definition of α-cut,

$$\mu(\underline{x}_j) = \mu(\overline{x}_j). \tag{B.11}$$

Replacing this in equation (B.10),

$$\mu(\underline{x}_j) \neq \mu(\check{x} + x_0), \tag{B.12}$$

and replacing equation (B.8) above,

$$\mu(\check{x} - x_0) \neq \mu(\check{x} + x_0). \tag{B.13}$$

Hence, $[\tilde{x}]$ is non-symmetric.

□

Theorem 5.3.1. *Let $[\tilde{x}]$ and $[\tilde{y}]$ be fuzzy intervals, $\circ \in \{+, -, \cdot, /\}$. If $[\tilde{x}]$ and $[\tilde{y}]$ are symmetric, then $[\tilde{z}] = [\tilde{x}] \circ [\tilde{y}]$ is also symmetric.*

Proof. Let $i \in \{1, \ldots, N\}$, be an uncertainty level. Recalling the α-cut concept,

$$[z_i] = [x_i] \circ [y_i]\,. \tag{B.14}$$

If $[\tilde{x}]$ and $[\tilde{y}]$ are symmetric, then, by Proposition 5.3.1,

$$\check{x}_i = \check{x}, \quad \check{y}_i = \check{y}, \quad \forall\, i. \tag{B.15}$$

Without loss of generality, let $\alpha_1 = 1$. Then, the α-cuts of level 1 are crisp, i.e., composed of a single point. The kernel of $[\tilde{z}]$ can be expressed as the midpoint of such α-cut, i.e.,

$$\check{z} = \check{z}_1. \tag{B.16}$$

If $\circ \in \{+, -, \cdot\}$, then,

$$\check{z}_i = \square(\check{x}_i \circ \check{y}_i) = \square(\check{x}_1 \circ \check{y}_1) = \check{z}_1 = \check{z}, \quad \forall\, i. \tag{B.17}$$

Hence, $[\tilde{z}]$ is symmetric. If $\circ$ is the inversion, a similar calculation yields the result. $\square$

Proposition 5.3.2. *Let $[\tilde{x}]$ be a fuzzy interval, $N \in \mathbb{N}$, $i, j \in \{1, \ldots, N\}$, and $[x_i], [x_j]$, α-cuts. Then*

$$\alpha_i > \alpha_j \implies [x_i] \subset [x_j].$$

Proof. Let $[x_i] = \langle \check{x}_i, \rho_{x,i} \rangle$ and $[y_i] = \langle \check{y}_i, \rho_{y,i} \rangle$. Let $\mu(\cdot)$ be the membership function of $[\tilde{x}]$, and $\check{x}$, its kernel. By definition of α-cut,

$$\mu(\check{x}_i - \rho_{x,i}) = \mu(\check{x}_i + \rho_{x,i}) = \alpha_i, \tag{B.18a}$$

$$\mu(\check{x}_j - \rho_{x,j}) = \mu(\check{x}_j + \rho_{x,j}) = \alpha_j. \tag{B.18b}$$

From above, if $\alpha_i > \alpha_j$, then,

$$\mu(\check{x}_i - \rho_{x,i}) > \mu(\check{x}_j - \rho_{x,j}), \tag{B.19a}$$

$$\mu(\check{x}_i + \rho_{x,i}) > \mu(\check{x}_j + \rho_{x,j}). \tag{B.19b}$$

Now, by construction, both $\check{x}_i - \rho_{x,i}$ and $\check{x}_j - \rho_{x,j}$ are lower than $\check{x}$, where μ is strictly increasing. Similarly, both $\check{x}_i + \rho_{x,i}$ and $\check{x}_j + \rho_{x,j}$ are higher than $\check{x}$, where μ is strictly decreasing. Then,

$$\check{x}_i - \rho_{x,i} > \check{x}_j - \rho_{x,j}, \tag{B.20a}$$

$$\check{x}_i + \rho_{x,i} < \check{x}_j + \rho_{x,j}. \tag{B.20b}$$

Combining these two,

$$-\left(\rho_{x,j} - \rho_{x,i}\right) < \check{x}_j - \check{x}_i < \rho_{x,j} - \rho_{x,i}. \tag{B.21}$$

And, more synthetically,

$$|\check{x}_j - \check{x}_i| < \rho_{x,j} - \rho_{x,i}. \tag{B.22}$$

Then, from Theorem A.2.2,

$$[x_i] \subset [x_j]. \tag{B.23}$$

$\square$

Theorem 5.3.3. *Let $\circ \in \{+, -, \cdot, /\}$ and $[\tilde{x}], [\tilde{y}], [\tilde{z}]$, be fuzzy intervals, such that $[\tilde{z}] = [\tilde{x}] \circ [\tilde{y}]$. Let $[z_i]$ be the α-cut of level i of $[\tilde{z}]$, and $\rho_{z,i}$, the radius of $[z_i]$. Let $E^{(rad)}(\cdot)$ and $E^{(inc)}(\cdot)$ return the maximum absolute error in the midpoint-radius and midpoint-increment formats, respectively. Then,*

$$E^{(inc)}(\rho_{z,i}) < E^{(rad)}(\rho_{z,i}), \quad \forall\, i \geq 2.$$

Proof. The result is proven for the addition and subtraction. The other two cases (multiplication and inversion) can be obtained by a similar reasoning.

The proof proceeds by induction.

i) (Assume $i = 2$.)
In the midpoint-increment representation,

$$\rho_{z,2} = \delta_{z,1} + \delta_{z,2}. \tag{B.24}$$

Applying the error function,

$$E^{(inc)}(\rho_{z,2}) = E^{(inc)}(\delta_{z,1}) + E^{(inc)}(\delta_{z,2}). \tag{B.25}$$

Now, according to Algorithm 5.2,

$$\delta_{z,1} = \triangle\left(\frac{1}{2}\varepsilon\,|\check{z}| + \delta_{x,1} + \delta_{y,1}\right). \tag{B.26}$$

Without loss of generality, assume that the above is computed with the following summation algorithm,

$$\begin{aligned}\hat{T}_1 &= \triangle\left(\frac{1}{2}\varepsilon|\check{z}| + \delta_{x,1}\right),\\ \hat{T}_2 &= \triangle\left(\hat{T}_1 + \delta_{y,1}\right).\end{aligned} \tag{B.27}$$

By Theorem A.2.1,

$$E^{(inc)}(\delta_{z,1}) = \varepsilon\cdot\left(\left|\hat{T}_1\right| + \left|\hat{T}_2\right|\right) = \varepsilon\cdot(2\delta_{x,1} + \delta_{y,1}) + \varepsilon^2|\check{z}|. \tag{B.28}$$

Similarly,

$$E^{(inc)}(\delta_{z,2}) = \varepsilon\cdot(\delta_{x,2} + \delta_{y,2}). \tag{B.29}$$

Replacing (B.28) and (B.29) in (B.25),

$$E^{(inc)}(\rho_{z,2}) = \varepsilon(\delta_{x,1} + \rho_{x,2} + \rho_{y,2}) + \varepsilon^2|\check{z}|. \tag{B.30}$$

Following an analogous procedure,

$$E^{(rad)}(\rho_{z,2}) = \varepsilon(\delta_{x,1} + \delta_{x,2} + \rho_{x,2} + \rho_{y,2}) + \varepsilon^2|\check{z}|. \tag{B.31}$$

And, since $\delta_{x,2} > 0$,

$$E^{(inc)}(\rho_{z,2}) < E^{(rad)}(\rho_{z,2}). \tag{B.32}$$

ii) (Assume $E^{(inc)}(\rho_{z,i}) < E^{(rad)}(\rho_{z,i}),\ \forall\, i \geq 2$.)
In the midpoint-increment representation,

$$\rho_{z,i+1} = \rho_{z,i} + \delta_{z,i+1}. \tag{B.33}$$

Applying the error function,

$$E^{(inc)}(\rho_{z,i+1}) = E^{(inc)}(\rho_{z,i}) + E^{(inc)}(\delta_{z,i+1}). \tag{B.34}$$

By the induction hypothesis,

$$E^{(inc)}(\rho_{z,i+1}) < E^{(rad)}(\rho_{z,i}) + E^{(inc)}(\delta_{z,i+1}). \tag{B.35}$$

By Theorem A.2.1,

$$E^{(rad)}(\rho_{z,i}) = \varepsilon \cdot (2\rho_{x,i} + \rho_{y,i}) + \varepsilon^2|\check{z}|, \tag{B.36a}$$

$$E^{(inc)}(\delta_{z,i+1}) = \varepsilon \cdot (\delta_{x,i+1} + \delta_{y,i+1}). \tag{B.36b}$$

Replacing in (B.35),

$$E^{(inc)}(\rho_{z,i+1}) < \varepsilon \cdot (\rho_{x,i} + \rho_{x,i+1} + \rho_{y,i+1}) + \varepsilon^2|\check{z}|. \tag{B.37}$$

Again, by Theorem A.2.1,

$$E^{(rad)}(\rho_{z,i+1}) = \varepsilon \cdot (2\rho_{x,i+1} + \rho_{y,i+1}) + \varepsilon^2|\check{z}| \tag{B.38}$$

And, since $\rho_{x,i} < \rho_{x,i+1}$,

$$E^{(inc)}(\rho_{z,i+1}) < E^{(rad)}(\rho_{z,i+1}). \tag{B.39}$$

□

Theorem 6.3.1. *Let $\mathbb{f}(\cdot)$ be defined by (6.17). Then, there exists an update function $\mathcal{U}(\cdot)$, a delay function $d(\cdot)$ and an initial guess $\mathbb{v}^{(0)}$, such that the associated asynchronous iteration on $\mathbb{f}(\cdot)$ does not converge to a fixed-point.*

Proof. From Theorem 6.2.1, it suffices to prove that $\rho(|\boldsymbol{L}|) \geq 1$, where $\boldsymbol{L}$ is given by (6.18). Since the sum of the elements in any row of $\boldsymbol{L}$ is equal to 1, then 1 is an eigenvalue of $\boldsymbol{L}$ and $\rho(\boldsymbol{L}) \geq 1$. Thus, from Theorem A.2.3 in Appendix A, $\rho(|\boldsymbol{L}|) \geq 1$. □

Theorem 6.3.2. *Let $\mathbf{g}(\cdot)$ be defined by:*

$$\mathbf{g}(\mathbb{v}) = (1-\alpha)\mathbb{v} + \alpha\mathbb{f}(\mathbb{v}), \tag{6.19}$$

where $0 < \alpha < 1$ and $\mathbb{f}(\cdot)$ is given by (6.16). Let the weights, $w_{hm}, m \in I_h$, satisfy the following:

$$1 - \sum_{m\in\mathcal{I}_h}\sum_{j\in\mathcal{I}_h} \frac{w_{hm}z_{hm}}{z_{hj}} \geq 0, \quad \forall h. \tag{6.20}$$

Then, any partially asynchronous iteration over $\mathbf{g}(\cdot)$ converges to $\mathbb{v}^*$, fixed-point of $\mathbb{f}(\cdot)$.

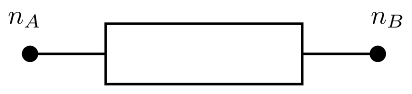
Proof. From Theorem 6.2.2, it suffices to prove that $\boldsymbol{L}$ given by (6.18) is irreducible and $\sum_{m=1}^{n} |l_{hm}| \leq 1$. Assuming (6.20), then it follows that $l_{hh} \geq 0$. Since $l_{hm} \geq 0$ by definition,

$$\sum_{m=1}^{n} |l_{hm}| = \sum_{m=1}^{n} l_{hm} = 1, \quad \forall h.$$

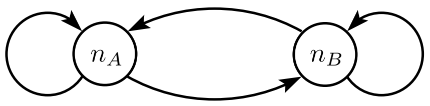
Now, consider the digraph $\boldsymbol{D}$ associated to matrix $\boldsymbol{L}$, which can be obtained from the circuit diagram in the following three steps: (i) for each node in the circuit, place a vertex in $\boldsymbol{D}$; (ii) for each component in the circuit, place a pair of anti-parallel arcs connecting the two terminal vertices in $\boldsymbol{D}$; (iii) place an arc from each vertex to itself. Figure B.1 illustrates this transformation process.

If the circuit is well defined, then $\boldsymbol{D}$ is strongly connected. And from Theorem A.2.4 in Appendix A, $\boldsymbol{L}$ is irreducible.

$\square$



(a) Circuit diagram.



(b) Digraph of matrix $\boldsymbol{L}$.

Figure B.1: Equivalence between the circuit diagram and the digraph of the application matrix $\boldsymbol{L}$, or *dependency graph* [Newton 1984].

References

[Newton 1984] A. R. Newton and A. L. Sangiovanni-Vincentelli. *Relaxation-based electrical simulation.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 3, no. 4, pp. 308–331, 1984. (Cited on pages 114 and 147.)

References

[461 2008] *IEEE Standard for Floating-Point Arithmetic.* IEEE Std 754-2008, pp. 1–70, 2008. (Cited on page 77.)

[Abur 2004] A. Abur and A. G. Exposito. Power system state estimation: theory and implementation. CRC Press, 2004. (Cited on page 45.)

[Allan 1974] R. Allan, B. Borkowska and C. Grigg. *Probabilistic analysis of power flows.* Proceedings of the Institution of Electrical Engineers, vol. 121, no. 12, pp. 1551–1556, 1974. (Cited on page 46.)

[Allan 1981] R. Allan and A. L. Da Silva. *Probabilistic load flow using multilinearisations.* IEE Proceedings C (Generation, Transmission and Distribution), vol. 128, pp. 280–287. IET, 1981. (Cited on page 47.)

[Alvarado 1992] F. Alvarado, Y. Hu and R. Adapa. *Uncertainty in power system modeling and computation.* Systems, Man and Cybernetics, 1992., IEEE International Conference on, pp. 754–760. IEEE, 1992. (Cited on pages 10 and 48.)

[Amdahl 1967] G. M. Amdahl. *Validity of the single processor approach to achieving large scale computing capabilities.* Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, pp. 483–485. ACM, 1967. (Cited on page 109.)

[Amin 2005] S. M. Amin and B. F. Wollenberg. *Toward a smart grid: power delivery for the 21st century.* Power and Energy Magazine, IEEE, vol. 3, no. 5, pp. 34–41, 2005. (Cited on page 7.)

[Anile 1995] A. Anile, S. Deodato and G. Privitera. *Implementing fuzzy arithmetic.* Fuzzy Sets and Systems, vol. 72, no. 2, pp. 239–250, 1995. (Cited on page 76.)

[Asanovic 2006] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams*et al.* *The landscape of parallel computing research: A view from berkeley.* Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006. (Cited on page 9.)

[Atallah 1984] M. Atallah and U. Vishkin. *Finding Euler tours in parallel.* Journal of Computer and System Sciences, vol. 29, no. 3, pp. 330–337, 1984. (Cited on pages 55 and 71.)

[Barboza 2004] L. V. Barboza, G. P. Dimuro and R. H. Reiser. *Towards interval analysis of the load uncertainty in power electric systems.* Probabilistic Methods Applied to Power Systems, 2004 International Conference on, pp. 538–544. IEEE, 2004. (Cited on page 49.)

[Beliakov 2015] G. Beliakov and Y. Matiyasevich. *A parallel algorithm for calculation of determinants and minors using arbitrary precision arithmetic.* BIT Numerical Mathematics, pp. 1–18, 2015. (Cited on page 88.)

[Bertsekas 1987] D. P. Bertsekas and D. El Baz. *Distributed asynchronous relaxation methods for convex network flow problems.* SIAM Journal on Control and Optimization, vol. 25, no. 1, pp. 74–85, 1987. (Cited on page 109.)

[Billinton 2001] R. Billinton, M. Fotuhi-Firuzabad and L. Bertling. *Bibliography on the application of probability methods in power system reliability evaluation 1996-1999.* IEEE Transactions on Power Systems, vol. 16, no. 4, pp. 595–602, 2001. (Cited on page 45.)

[Blelloch 1990] G. E. Blelloch. Vector models for data-parallel computing. MIT Press, Cambridge, MA, USA, 1990. (Cited on pages 53, 54 and 56.)

[Bondia 2006] J. Bondia, A. Sala, J. Pico and M. Sainz. *Controller Design Under Fuzzy Pole-Placement Specifications: An Interval Arithmetic Approach.* IEEE Transactions on Fuzzy Systems, vol. 14, no. 6, pp. 822–836, 2006. (Cited on pages 75 and 85.)

[Borkowska 1974] B. Borkowska. *Probabilistic load flow.* IEEE Transactions on Power Apparatus and Systems, vol. 93, no. 3, pp. 752–759, 1974. (Cited on page 46.)

[Boukezzoula 2014] R. Boukezzoula, S. Galichet, L. Foulloy and M. Elmasry. *Extended gradual interval (EGI) arithmetic and its application to gradual weighted averages.* Fuzzy Sets and Systems, vol. 257, pp. 67–84, 2014. (Cited on page 81.)

[Brönnimann 2006] H. Brönnimann, G. Melquiond and S. Pion. *The design of the Boost interval arithmetic library.* Theoretical Computer Science, vol. 351, no. 1, pp. 111–118, 2006. (Cited on page 76.)

[Buisson 2003] J.-C. Buisson and A. Garel. *Balancing meals using fuzzy arithmetic and heuristic search algorithms.* IEEE Transactions on Fuzzy Systems, vol. 11, no. 1, pp. 68–78, 2003. (Cited on pages 75 and 85.)

[Chang 1995] C.-H. Chang and Y.-C. Wu. *The genetic algorithm based tuning method for symmetric membership functions of fuzzy logic control systems.* International IEEE/IAS Conference on Industrial Automation and Control: Emerging Technologies, 1995., pp. 421–428. IEEE, 1995. (Cited on page 81.)

[Chazan 1969] D. Chazan and W. Miranker. *Chaotic relaxation.* Linear algebra and its applications, vol. 2, no. 2, pp. 199–222, 1969. (Cited on page 111.)

[Che 2008] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer and K. Skadron. *A performance study of general-purpose applications on graphics processors using CUDA.* Journal of parallel and distributed computing, vol. 68, no. 10, pp. 1370–1380, 2008. (Cited on pages 9 and 19.)

[Chen 2006] C.-T. Chen, C.-T. Lin and S.-F. Huang. *A fuzzy approach for supplier evaluation and selection in supply chain management.* International journal of production economics, vol. 102, no. 2, pp. 289–301, 2006. (Cited on pages 75 and 85.)

[Chen 2008] P. Chen, Z. Chen and B. Bak-Jensen. *Probabilistic load flow: A review.* Third International Conference on Electric Utility Deregulation and Restructuring and Power Technologies, 2008. DRPT 2008., pp. 1586–1591, 2008. (Cited on page 45.)

[Clark 2010] M. A. Clark, R. Babich, K. Barros, R. C. Brower and C. Rebbi. *Solving Lattice QCD systems of equations using mixed precision solvers on GPUs.* Computer Physics Communications, vol. 181, no. 9, pp. 1517–1528, 2010. (Cited on page 28.)

[Collange 2010] S. Collange. *Design challenges of GPGPU architectures: specialized arithmetic units and exploitation of regularity.* PhD thesis, Université de Perpignan, 2010. (Cited on pages 7 and 26.)

[Coppersmith 1987] D. Coppersmith and S. Winograd. *Matrix multiplication via arithmetic progressions.* Proceedings of the nineteenth annual ACM symposium on Theory of computing, pp. 1–6. ACM, 1987. (Cited on page 98.)

[Cortes-Carmona 2010] M. Cortes-Carmona, R. Palma-Behnke and G. Jimenez-Estevez. *Fuzzy Arithmetic for the DC Load Flow.* IEEE Transactions on Power Systems, vol. 25, no. 1, pp. 206–214, 2010. (Cited on pages 75 and 85.)

[Cuda 2012] C. Cuda. *Programming guide.* NVIDIA Corporation, July, 2012. (Cited on pages 9, 24, 25, 27, 88 and 92.)

[Da Silva 1984] A. Da Silva, V. Arienti and R. Allan. *Probabilistic load flow considering dependence between input nodal powers.* IEEE Transactions on Power Apparatus and Systems, vol. 6, no. PAS-103, pp. 1524–1530, 1984. (Cited on page 47.)

[Davis 2004] T. A. Davis. *Algorithm 832: UMFPACK V4. 3—an unsymmetric-pattern multifrontal method.* ACM Transactions on Mathematical Software, vol. 30, no. 2, pp. 196–199, 2004. (Cited on page 10.)

[Davis 2010] T. A. Davis and E. Palamadai Natarajan. *Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems.* ACM Transacctions on Mathematical Software, vol. 37, no. 3, pp. 36:1–36:17, 2010. (Cited on pages 10 and 124.)

[Davis 2012] J. D. Davis and E. S. Chung. *SpMV: A Memory-Bound Application on the GPU Stuck Between a Rock and a Hard Place.* Technical Report MSR-TR-2012-95, 2012. (Cited on page 88.)

[Defour 2013] D. Defour and M. Marin. *Regularity Versus Load-balancing on {GPU} for Treefix Computations.* Procedia Computer Science, 2013 International Conference on Computational Science, vol. 18, pp. 309 – 318, 2013. (Cited on page 53.)

[Defour 2014] D. Defour and M. Marin. *FuzzyGPU: A Fuzzy Arithmetic Library for GPU.* Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on, pp. 624–631, 2014. (Cited on page 75.)

[Den Hertog 1992] D. Den Hertog. *Interior point approach to linear, quadratic and convex programming: algorithms and complexity.* PhD thesis, TU Delft, Delft University of Technology, 1992. (Cited on page 98.)

[DIM 2012] *10th DIMACS Implementation Challenge.* http://www.cc.gatech.edu/dimacs10/index.shtml, 2012. (Cited on page 62.)

[Du 2012] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson and J. Dongarra. *From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming.* Parallel Computing, vol. 38, no. 8, pp. 391–407, 2012. (Cited on page 9.)

[Dubois 1978] D. Dubois and H. Prade. *Operations on fuzzy numbers.* International Journal of systems science, vol. 9, no. 6, pp. 613–626, 1978. (Cited on pages 76, 77 and 79.)

[El-Baz 1996] D. El-Baz, P. Spiteri, J.-C. Miellou and D. Gazen. *Asynchronous iterative algorithms with flexible communication for nonlinear network flow problems.* Journal of Parallel and Distributed Computing, vol. 38, pp. 1–15, 1996. (Cited on page 109.)

[Fang 2012] X. Fang, S. Misra, G. Xue and D. Yang. *Smart Grid – The New and Improved Power Grid: A Survey.* IEEE Communications Surveys Tutorials, vol. 14, no. 4, pp. 944–980, 2012. (Cited on page 7.)

[Fortin 2008] J. Fortin, D. Dubois and H. Fargier. *Gradual Numbers and Their Application to Fuzzy Interval Analysis.* IEEE Transactions on Fuzzy Systems, vol. 16, no. 2, pp. 388–402, 2008. (Cited on page 82.)

[Frommer 2000] A. Frommer and D. B. Szyld. *On asynchronous iterations.* Journal of Computational and Applied Mathematics, vol. 123, no. 1–2, pp. 201 – 216, 2000. Numerical Analysis 2000. Vol. III: Linear Algebra. (Cited on page 110.)

[Gagolewski 2012] M. Gagolewski. *FuzzyNumbers Package: Tools to deal with fuzzy numbers in R.* http://www. ibspan. waw. pl/g̃agolews/FuzzyNumbers, 2012. (Cited on page 76.)

[Garcia 2010] N. Garcia. *Parallel Power Flow solutions using a biconjugate gradient algorithm and a Newton method: A GPU-based approach.* IEEE Power and Energy Society General Meeting, pp. 1–4, 2010. (Cited on page 109.)

[Glaskowsky 2009] P. N. Glaskowsky. *NVIDIA's Fermi: the first complete GPU computing architecture.* White paper, 2009. (Cited on pages 20, 23 and 61.)

[Goossens 2013] B. Goossens and D. Parello. *Limits of instruction-level parallelism capture.* Procedia Computer Science, vol. 18, pp. 1664–1673, 2013. (Cited on page 87.)

[Goualard 2006] F. Goualard. *Gaol 3.1.1: Not just another interval arithmetic library.* Laboratoire d'Informatique de Nantes-Atlantique, 2006. (Cited on page 76.)

[Gupta 1997] A. Gupta, G. Karypis and V. Kumar. *Highly scalable parallel algorithms for sparse matrix factorization.* IEEE Transactions on Parallel and Distributed Systems, vol. 8, no. 5, pp. 502–520, 1997. (Cited on page 109.)

[Harris 2007] M. Harris*et al. Optimizing parallel reduction in CUDA.* NVIDIA Developer Technology, vol. 2, no. 4, 2007. (Cited on page 22.)

[Harris 2012] M. Harris. *GPGPU. org.* General Purpose Computation on Graphics Hardware, 2012. (Cited on page 9.)

[Higham 2002] N. J. Higham. Accuracy and stability of numerical algorithms. Siam, 2002. (Cited on pages 66, 68, 77, 84 and 139.)

[Hoberock 2010] J. Hoberock and N. Bell. *Thrust: A Parallel Template Library, Version 1.7.0.* http://thrust.github.io/, 2010. (Cited on page 93.)

[Honkala 2001] M. Honkala, J. Roos and M. Valtonen. *New multilevel Newton-Raphson method for parallel circuit simulation.* Proceedings of European Conference on Circuit Theory and Design, vol. 1, pp. 113–116, 2001. (Cited on page 109.)

[Horn 2012] R. A. Horn and C. R. Johnson. Matrix analysis. Cambridge University Press, 2012. (Cited on page 140.)

[Iverson 1962] K. E. Iverson. *A programming language.* Proceedings of the May 1-3, 1962, Spring Joint Computer Conference, pp. 345–351. ACM, 1962. (Cited on page 56.)

[Keckler 2011] S. Keckler, W. Dally, B. Khailany, M. Garland and D. Glasco. *GPUs and the Future of Parallel Computing.* IEEE Micro, vol. 31, no. 5, pp. 7–17, 2011. (Cited on page 7.)

[kep 2012] *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, 2012. (Cited on page 121.)

[Kersting 2012] W. H. Kersting. Distribution system modeling and analysis. CRC Press, 2012. (Cited on pages 7 and 37.)

[Kolarovic 2013] N. Kolarovic. *Fuzzy numbers and basic fuzzy arithmetics (+, -, *, /, 1/x) implementation written in Java.* Online at http://http://sourceforge.net/projects/fuzzyarith/, 2013. (Cited on pages 76 and 89.)

[Leiserson 1988] C. E. Leiserson and B. M. Maggs. *Communication-efficient parallel algorithms for distributed random-access machines.* Algorithmica, vol. 3, no. 1-4, pp. 53–77, 1988. (Cited on pages 44, 54 and 55.)

[Li 2005] X. S. Li. *An overview of SuperLU: Algorithms, implementation, and user interface.* ACM Transactions on Mathematical Software, vol. 31, no. 3, pp. 302–325, 2005. (Cited on page 10.)

[Lubachevsky 1986] B. Lubachevsky and D. Mitra. *A chaotic asynchronous algorithm for computing the fixed-point of a nonnegative matrix of unit spectral radius.* Journal of the ACM, vol. 33, no. 1, pp. 130–150, 1986. (Cited on page 113.)

[Machowski 1997] J. Machowski, J. Bialek and J. Bumby. Power system dynamics and stability. Wiley, 1997. (Cited on pages 10 and 36.)

[Marin 2014] M. Marin, D. Defour and F. Milano. *Power flow analysis under uncertainty using symmetric fuzzy arithmetic.* IEEE PES General Meeting, Conference & Exposition, pp. 1–5, 2014. (Cited on page 75.)

[Marin 2015] M. Marin, D. Defour and F. Milano. Linear circuit analysis based on parallel asynchronous fixed-point method. 2015. (Cited on page 108.)

[Mendel 2006] J. Mendel and H. Wu. *Type-2 Fuzzistics for Symmetric Interval Type-2 Fuzzy Sets: Part 1, Forward Problems.* IEEE Transactions on Fuzzy Systems, vol. 14, no. 6, pp. 781–792, 2006. (Cited on page 81.)

[Mendive 1975] D. Mendive. An application of ladder network theory to the solution of three-phase radial load-flow problems. New Mexico State University, 1975. (Cited on page 42.)

[Merrill 2012] D. Merrill, M. Garland and A. Grimshaw. *Scalable GPU graph traversal.* ACM Special Interest Group on Programming Languages Notices, vol. 47, pp. 117–128. ACM, 2012. (Cited on pages 53, 61 and 62.)

[Meza 2007] J. Meza, R. Oliva, P. Hough and P. Williams. *OPT++: An object-oriented toolkit for nonlinear optimization.* ACM Transactions on Mathematical Software, vol. 33, no. 2, page 12, 2007. (Cited on page 98.)

[Miellou 1998] J. Miellou, D. El Baz and P. Spiteri. *A new class of asynchronous iterative algorithms with order intervals.* Mathematics of Computation of the American Mathematical Society, vol. 67, no. 221, pp. 237–255, 1998. (Cited on page 122.)

[Milano 2010] F. Milano. Power system modelling and scripting. Springer Science & Business Media, 2010. (Cited on pages 7, 10, 37, 120 and 124.)

[Milano 2013] F. Milano. *A Python-based software tool for power system analysis.* IEEE Power and Energy Society General Meeting, pp. 1–5, 2013. (Cited on pages 7, 10, 68, 98, 122 and 124.)

[Miller 1985] G. L. Miller and J. H. Reif. *Parallel Tree Contraction and Its Application.* 26th Symposium on Foundations of Computer Science, pp. 478–489, Portland, Oregon, 1985. IEEE. (Cited on page 55.)

[Momoh 2009] J. Momoh*et al.* *Smart grid design for efficient and flexible power networks operation and control.* IEEE Power Systems Conference and Exposition, 2009, pp. 1–8, 2009. (Cited on pages 7 and 10.)

[Moore 1966] R. E. Moore. Interval analysis, vol. 4. Prentice-Hall Englewood Cliffs, 1966. (Cited on page 76.)

[Moore 1979] R. E. Moore. Methods and applications of interval analysis. Studies in Applied and Numerical Mathematics. Society for Industrial and Applied Mathematics, 1979. (Cited on page 49.)

[Nagel 2013] I. Nagel. *Analog microelectronic emulation for dynamic power system computation.* PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2013. (Cited on page 109.)

[Nasre 2013] R. Nasre, M. Burtscher and K. Pingali. *Data-driven versus topology-driven irregular computations on gpus.* IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS), pp. 463–474, 2013. (Cited on page 26.)

[Neumaier 1990] A. Neumaier. Interval methods for systems of equations, vol. 37. Cambridge university press, 1990. (Cited on pages 76 and 139.)

[Newton 1984] A. R. Newton and A. L. Sangiovanni-Vincentelli. *Relaxation-based electrical simulation.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 3, no. 4, pp. 308–331, 1984. (Cited on pages 114 and 147.)

[Nguyen 2007] H. Nguyen. Gpu gems 3. Addison-Wesley Professional, 2007. (Cited on page 7.)

[Nickolls 2010] J. Nickolls and W. Dally. *The GPU Computing Era.* Micro, IEEE, vol. 30, no. 2, pp. 56–69, 2010. (Cited on page 8.)

[Nilsson 2009] J. W. Nilsson and S. A. Riedel. Electric circuits, vol. 8. Prentice Hall, 2009. (Cited on page 33.)

[Ogita 2005] T. Ogita, S. M. Rump and S. Oishi. *Accurate Sum and Dot Product.* SIAM J. Sci. Comput., vol. 26, no. 6, pp. 1955–1988, 2005. (Cited on pages 66 and 68.)

[Owens 2008] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips. *GPU computing.* Proceedings of the IEEE, vol. 96, no. 5, pp. 879–899, 2008. (Cited on pages 8 and 19.)

[Pharr 2005] M. Pharr and R. Fernando. Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation. Addison-Wesley Professional, 2005. (Cited on page 7.)

[Potra 2000] F. A. Potra and S. J. Wright. *Interior-point methods.* Journal of Computational and Applied Mathematics, vol. 124, no. 1 - 2, pp. 281 – 302, 2000. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations. (Cited on page 98.)

[Revol 2001] N. Revol and F. Rouillier. *The MPFI library.* http://mpfi.gforge.inria.fr/, 2001. (Cited on page 76.)

[Rump 1992] S. M. Rump. *On the solution of interval linear systems.* Computing, vol. 47, no. 3-4, pp. 337–353, 1992. (Cited on page 98.)

[Rump 1999] S. M. Rump. *Fast and parallel interval arithmetic.* BIT Numerical Mathematics, vol. 39, no. 3, pp. 534–554, 1999. (Cited on pages 77, 78 and 96.)

[Ryoo 2008] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton and W.-m. W. Hwu. *Program optimization space pruning for a multithreaded gpu.* Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization, pp. 195–204, 2008. (Cited on page 19.)

[Saric 2006] A. T. Saric and A. M. Stankovic. *Ellipsoidal approximation to uncertainty propagation in boundary power flow.* IEEE PES Power Systems Conference and Exposition, 2006, pp. 1722–1727, 2006. (Cited on page 46.)

[Schellenberg 2005] A. Schellenberg, W. Rosehart and J. Aguado. *Cumulant-based probabilistic optimal Power Flow (P-OPF) with Gaussian and gamma distributions.* IEEE Transactions on Power Systems, vol. 20, no. 2, pp. 773–781, 2005. (Cited on page 48.)

[Shirmohammadi 1988] D. Shirmohammadi, H. Hong, A. Semlyen and G. Luo. *A compensation-based Power Flow method for weakly meshed distribution and transmission networks.* Power Systems, IEEE Transactions on, vol. 3, no. 2, pp. 753–762, 1988. (Cited on page 42.)

[Siler 2005] W. Siler and J. J. Buckley. Fuzzy expert systems and fuzzy reasoning. John Wiley & Sons, 2005. (Cited on page 76.)

[Simulator 2005] P. Simulator. *Version 10.0 SCOPF.* PVQV, PowerWorld Corporation, Champaign, IL, vol. 61820, 2005. (Cited on page 7.)

[Simulink 1993] M. Simulink and M. Natick. *The Mathworks*, 1993. (Cited on page 7.)

[Stone 2010] J. E. Stone, D. Gohara and G. Shi. *OpenCL: A parallel programming standard for heterogeneous computing systems.* Computing in science & engineering, vol. 12, no. 1-3, pp. 66–73, 2010. (Cited on pages 9 and 24.)

[Stott 1974] B. Stott. *Review of load-flow calculation methods.* Proceedings of the IEEE, vol. 62, no. 7, pp. 916–929, 1974. (Cited on pages 10 and 32.)

[Su 2005] C.-L. Su. *Probabilistic load-flow computation using point estimate method.* Power Systems, IEEE Transactions on, vol. 20, no. 4, pp. 1843–1851, 2005. (Cited on page 47.)

[Talukdar 1983] S. Talukdar, S. S. Pyo and R. Mehrotra. Designing algorithms and assignments for distributed processing. Electric Power Research Institute, 1983. (Cited on page 109.)

[Tarjan 1984] R. E. Tarjan and U. Vishkin. *Finding Biconnected Componemts And Computing Tree Functions In Logarithmic Parallel Time.* Proceedings of the 25th Annual Symposium on Foundations of Computer Science, 1984, pp. 12–20, 1984. (Cited on page 55.)

[Trutschnig 2013] W. Trutschnig, M. Lubiano and J. Lastra. *SAFD — An R Package for Statistical Analysis of Fuzzy Data.* In C. Borgelt, M. Á. Gil, J. M. Sousa and M. Verleysen, editeurs, Towards Advanced Data Analysis by Combining Soft Computing and Statistics, vol. 285 of *Studies in Fuzziness and Soft Computing*, pp. 107–118. Springer Berlin Heidelberg, 2013. (Cited on page 76.)

[Tseng 1990] P. Tseng, D. P. Bertsekas and J. N. Tsitsiklis. *Partially asynchronous, parallel algorithms for network flow and other problems.* SIAM Journal on Control and Optimization, vol. 28, no. 3, pp. 678–710, 1990. (Cited on pages 112 and 113.)

[Tsitsiklis 1984] J. N. Tsitsiklis. *Problems in Decentralized Decision making and Computation.* Technical report, DTIC Document, 1984. (Cited on page 109.)

[Tu 2002] F. Tu and A. J. Flueck. *A message-passing distributed-memory Newton-GMRES parallel Power Flow algorithm.* IEEE Power Engineering Society Summer Meeting, vol. 3, pp. 1477–1482, 2002. (Cited on page 109.)

[UFS 2012] *The University of Florida Sparse Matrix Collection.* http://www.cise.ufl.edu/research/sparse/matrices/, 2012. (Cited on page 62.)

[Vaccaro 2013] A. Vaccaro, C. A. Cañizares and K. Bhattacharya. *A range arithmetic-based optimization model for Power Flow analysis under interval uncertainty.* IEEE Transactions on Power Systems, vol. 28, no. 2, pp. 1179–1186, 2013. (Cited on pages 11, 46, 49, 95, 96 and 98.)

[van der Sanden 2011] J. van der Sanden. Evaluating the Performance and Portability of OpenCL. Master's thesis, Faculty of Electrical Engineering Eindhoven University of Technology, 2011. (Cited on page 25.)

[Volkov 2008] V. Volkov and J. W. Demmel. *Benchmarking GPUs to tune dense linear algebra.* IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11, 2008. (Cited on pages 9 and 27.)

[Volkov 2010] V. Volkov. *Better performance at lower occupancy.* Proceedings of the GPU Technology Conference, vol. 10, 2010. (Cited on page 28.)

[Wan 1993] Y.-h. Wan and B. K. Parsons. Factors relevant to utility integration of intermittent renewable technologies. National Renewable Energy Laboratory, 1993. (Cited on pages 32 and 45.)

[Wang 1992] Z. Wang and F. Alvarado. *Interval arithmetic in Power Flow analysis.* IEEE Transactions on Power Systems, vol. 7, no. 3, pp. 1341–1349, 1992. (Cited on page 48.)

[Zhang 2004] P. Zhang and S. T. Lee. *Probabilistic load flow computation using the method of combined cumulants and Gram-Charlier expansion.* IEEE Transactions on Power Systems, vol. 19, no. 1, pp. 676–682, 2004. (Cited on page 47.)