Accelerating computational diffusion MRI using Graphics Processing Units

Moises Hernandez Fernandez

Oxford Centre for Functional MRI of the Brain (FMRIB)

Nuffield Department of Clinical Neurosciences

Hertford College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Michaelmas 2017

Acknowledgements

This DPhil was partially funded by the Human Connectome Project, WU-Minn Consortium (1U54MH091657; Principal Investigators: David Van Essen and Kamil Ugurbil) funded by the 16 NIH Institutes and Centers that support the NIH Blueprint for Neuroscience Research. The project presented in this thesis was awarded the NVIDIA 2016 GPU centre of excellence achievement, and the prize was used for partially funding the DPhil.

I have been very fortunate of having four fantastic supervisors during my DPhil: Dr. Stam Sotiropoulos, Dr. Istvan Reguly, Prof. Stephen Smith and Prof. Mike Giles. Their continuous support and guidance have been essential for making this project possible. Whenever I had problems with my research work, my supervisors were always available to me, providing solutions and proposing new ideas. I would like to thank them for all the help provided.

I am especially grateful to Stam. He has supported me from day one. I have learned many things from him, including image analysis, diffusion MRI and many other technical skills. But in my opinion, the most useful thing I learned from him is how to be a hard worker. I really appreciate his patience and the countless amount of hours that he has spent correcting my writing, including this thesis.

I am very thankful to my wife Ionela for the tremendous contribution to this project. As a mathematician, she has helped me to better understand equations. As someone with a master's degree and having experience with large documents, she has converted this thesis into Latex format. As a cook, she has provided me with very delicious meals everyday, which definitely helped for working hard. And as a wife, she has made me happy.

A special mention must go to Dr. Saad Jbabdi, Dr. Jesper Andersson and Dr. Wes Armour for reviewing my progress and offering ideas during the course of my D.Phil.

The Oxford Centre for Functional MRI of the Brain (FMRIB) provides an excellent environment for scientific research. I would like to thank everyone working in this centre, especially Susan Field and Marilyn Goulding for their support during my first months working there, the Analysis group people, and my lunch partners and friends that make FMRIB so great: Fidel, Ludo, Paul, Anderson, Zobair, Hanna, Jonathan, Ged, Sean, Matteo, Michiel, Gwen, Eugene and all those I forget to mention.

I would have not finished this project without recharging batteries by spending some leisure time with friends, playing football, running, and of course having good British beers. Thanks to my Oxford adventures partners Francesca, Link, Joe, Andrzej, Manu, Joce, Kyoko, Carina (thanks for submitting the thesis for me), Richard and Pixel's creator.

Finally, my parents Isabel and Jesus, my siblings Fuensanta, Juan Jesus and Joaquin, my cousin Carmen María, my aunt Maria, and the rest of my large family have been extremely supportive during these years. They even paid for my flight tickets to go home when I missed them, and they have been supplying Ibérico ham during all these years.

Moises

Oxford, 30 November 2017

Abstract

Diffusion magnetic resonance imaging (dMRI) allows uniquely the study of the human brain non-invasively and in vivo. Advances in dMRI offer new insight into tissue microstructure and connectivity, and the possibility of investigating the mechanisms and pathology of neurological diseases. The great potential of the technique relies on indirect inference, as modelling frameworks are necessary to map dMRI measurements to neuroanatomical features. However, this mapping can be computationally expensive, particularly given the trend of increasing dataset sizes and/or the increased complexity in biophysical modelling. Limitations on computing can restrict data exploration and even methodology development. A step forward is to take advantage of the power offered by recent parallel computing architectures, especially Graphics Processing Units (GPUs). GPUs are massive parallel processors that offer trillions of floating point operations per second, and have made possible the solution of computationally-intensive scientific problems that were intractable before. However, they are not inherently suited for all types of problems, and bespoke computational frameworks need to be developed in many cases to take advantage of their full potential.

In this thesis, we propose parallel computational frameworks for the analysis of dMRI using GPUs within different contexts. We show that GPU-based designs can offer accelerations of more than two orders of magnitude for a number of scientific computing tasks with different parallelisability requirements, ranging from biophysical modelling for tissue microstructure estimation to white matter tractography for connectome generation. We develop novel and efficient GPU-accelerated solutions, including a framework that automatically generates GPU parallel code from a user-specified biophysical model. We also present a parallel GPU framework for performing probabilistic tractography and generating whole-brain connectomes. Throughout the thesis, we discuss several strategies for parallelising scientific applications, and we show the great potential of the accelerations obtained, which change the perspective of what is computationally feasible.

Publications

Journals

- Alfaro-Almagro F., Jenkinson M., Bangerter N.K., Andersson J.L.R., Griffanti L., Douaud G., Sotiropoulos S.N., Jbabdi S., **Hernandez-Fernandez M.**, Vallee E., Vidaurre D., Webster M., McCarthy P.D.. Rorden C., Daducci A., Alexander D., Zhang H., Dragonu I., Matthews P., Miller K.L., Smith S. "Image Processing and Quality Control for the first 10,000 Brain Imaging Datasets from UK Biobank". *NeuroImage*, 166: pp.400-424. (2018).
- Madhyastha T.M.,Koh N., McAllister-Day T.K., **Hernandez Fernandez, M.**, Kelley A., Peterson D.J., Rajan S., Woelfer K., Wolf J., Grabowski T.J. "Running Neuroimaging Applications on Amazon Web Services: How, When, and at What Cost?". *Frontiers in Neuroinformatics*, 11(63). (2017)
- Sotiropoulos S.N., **Hernandez-Fernandez M.**, Vu A.T., Andersson J.L., Moeller S., Yacoub E., Lenglet C., Ugurbil K., Behrens T.E.J., Jbabdi S. "Fusion in diffusion MRI for improved fibre orientation estimation: An application to the 3T and 7T data of the Human Connectome Project". *NeuroImage*, 134: pp. 396–409. (2016).
- Donahue C.J., Sotiropoulos S.N., Jbabdi S., **Hernandez-Fernandez M.**, Behrens T.E., Dyrby T.B., Coalson T., Kennedy H., Knoblauch K., Van Essen D.C., Glasser, M. F. "Using Diffusion Tractography to Predict Cortical Connection Strength and Distance: A Quantitative Comparison with Tracers in the Monkey." *The Journal of Neuroscience: The Official Journal of the Society for Neuroscience*, 36(25): pp.6758–70. (2016)
- **Hernandez M.**, Guerrero G.D., Cecilia J.M., García J.M., Inuggi A., Jbabdi S., Behrens T.E.J., Sotiropoulos S.N. "Accelerating Fibre Orientation Estimation from Diffusion Weighted Magnetic Resonance Imaging Using GPUs". *PLoS ONE*, 8(4). (2013).
- Sotiropoulos S.N., Jbabdi S., Xu J., Andersson J.L., Moeller S., Auerbach E.J., Glasser M.F., **Hernandez M.**, Sapiro G., Jenkinson M., Feinberg D.A., Yacoub E., Lenglet C., Van Essen D.C., Ugurbil K., Behrens T.E.J. "Advances in diffusion MRI acquisition and processing in the Human Connectome Project". *NeuroImage*, 80: pp.125–43. (2013).

Conference Proceedings

- **Hernandez-Fernandez M.**, Reguly I., Jbabdi S., Giles M., Smith S., Sotiropoulos S.N. "Using GPUs to accelerate computational diffusion MRI: From microstructure estimation to tractography and connectomes". In: *International Society for Magnetic Resonance in Medicine (ISMRM) 27th Annual Meeting*, Paris (France). (2018).
- **Hernandez-Fernandez M**, Reguly I., Giles M., Smith S., Sotiropoulos S.N. "cuDIMOT: A CUDA Toolbox for Modelling the Brain Tissue Microstructure from Diffusion MRI". *GPU Technology Conference (GTC)*, Munich (Germany). (2017).
- **Hernandez-Fernandez M.**, Reguly I., Giles M., Jbabdi S., Smith S., Sotiropoulos S.N. "A fast and flexible toolbox for tracking brain connections in diffusion MRI datasets using GPUs". In: *The Organization for Human Brain Mapping (OHBM)*, Geneva (Switzerland). (2016).
- **Hernandez-Fernandez M.**, Reguly I., Giles M., Smith S., Sotiropoulos S.N. "White matter tractography and Human Brain Connections using GPUs". In *GPU Technology Conference*, San Jose (CA, US). (2016).
- Foxley S., Jbabdi S., **Hernandez Fernandez M.**, Clare S., Scott C., Ansorge O., Miller K. "A comparison of multiple acquisition strategies to overcome B1 inhomogeneities in diffusion imaging of post-mortem human brain at 7T". In: *International Society for Magnetic Resonance in Medicine (ISMRM) 24rd Annual Meeting*, Singapore. (2016).
- Foxley S., Jbabdi S., **Hernandez Fernandez M.**, Clare S., Scott, C., Ansorge O., Miller K.: "Improved tract identification of post-mortem human brain with high-resolution DTI at 7T". In: *The Organization for Human Brain Mapping (OHBM)*, Honolulu (USA). (2015).
- **Hernandez M.**, Guerrero G.D., Cecilia J.M., Garcia J.M., Inuggi A., Sotiropoulos S.N. "Accelerating fibre orientation estimation from diffusion weighted magnetic resonance imaging using GPUs". In: *International Conference on Parallel, Distributed and Network-based Processing (PDP)*, Garching (Germany). (2012).

Case studies

- NVIDIA GPU Center of Excellence Achievement Award 2016: White matter tractography and Human Brain Connections using GPUs.
- Electronic publication. A case study about the software we have developed using GPUs for analyzing brain diffusion MRI data: Imaging Software Bring the Brain into Fuller Focus, The Science and Engineering South Consortium, UK.

Software

- Tool for fitting ball & sticks model (and extensions) on GPUs: `http://users.fmrib.ox.ac.uk/~moisesf/Bedpostx_GPU`
- CUDA Diffusion modelling tool (cuDIMOT): `http://users.fmrib.ox.ac.uk/~moisesf/cudimot`
- Tool for performing probabilistic tractography on GPUs (Probtrackx_gpu): `http://users.fmrib.ox.ac.uk/~moisesf/Probtrackx_GPU`

Contents

List of Algorithms **xv**

List of Figures **xvii**

List of Tables **xxi**

Glossary **xxiii**

1 Introduction **1**

1.1 Organisation of the thesis 5

1.2 Software 8

2 GPU programming **9**

2.1 Parallel computing 10

2.1.1 Historical review and motivation 10

2.1.2 Parallel computing architectures and programming models . 12

2.1.3 Types of parallelism 16

2.2 Scientific parallel computing: typical considerations and guidelines . 17

2.3 Graphics Processing Units (GPUs) 21

2.4 The GPU architecture 22

2.4.1 General overview and CPU-GPU interconnection 22

2.4.2 The Streaming Multiprocessor 23

2.5 GPU memory hierarchy 26

2.6 The CUDA programming model 28

2.6.1 Thread organisation 28

2.6.2 Threads and memory synchronisation 29

2.7 The CUDA execution model 30

2.7.1 Warp execution 30

2.7.2 Latencies 31
2.8 Challenges in GPU programming 32
2.8.1 Exposing parallelism and increasing device occupancy 32
2.8.2 Avoiding warp divergence 34
2.8.3 Memory access patterns 35

3 Diffusion MRI and Tractography **39**
3.1 Diffusion MR Imaging 40
3.1.1 Pulsed gradient spin echo sequence 42
3.1.2 Estimation of the apparent diffusion coefficient 43
3.2 Diffusion Tensor Imaging 44
3.2.1 Fitting the diffusion tensor 45
3.2.2 Microstructure parameters derived from the diffusion tensor 47
3.2.3 Limitations of the diffusion tensor 48
3.3 Multi-compartment diffusion models 49
3.4 White matter tractography 53
3.5 Brain Connectomes 58
3.6 Computational requirements 60

4 Estimation of local diffusion MRI models on GPUs **61**
4.1 Introduction 65
4.2 Modelling diffusion signal: Fitting the ball and sticks model 67
4.3 Modelling diffusion signal on GPUs: Design and Implementation . . 70
4.3.1 Parallelising the MCMC algorithm 72
4.3.2 Parallelising the Levenberg-Marquardt algorithm 81
4.3.3 Performance limitations 86
4.3.4 Using Multiple GPUs 88
4.4 Results: Performance gains 88
4.4.1 MCMC routine evaluation 92
4.4.2 Levenberg routine evaluation 94

4.4.3 Overall evaluation 96
4.5 Validation: Comparison to CPU implementation 99
4.6 Discussion 103
4.6.1 Extension of the designs to other models 105
4.6.2 Clinical and Big Data applications 107

5 A generic GPU toolbox for model fitting and exploring brain microstructure using diffusion MRI 109

5.1 Introduction 114
5.2 Fitting microstructural diffusion MRI models 115
5.3 cuDIMOT: CUDA Diffusion Modelling Toolbox 117
5.3.1 General Design 117
5.3.2 Challenges: a generic toolbox 119
5.3.3 The user interface: a flexible toolbox 123
5.3.4 Implementation of the fitting routines 124
5.3.5 Performance and Validation 130
5.4 Exploring microstructure diffusion MRI models with cuDIMOT: Estimating fibre orientation dispersion 135
5.4.1 Modelling the dispersion of fibre orientations with a Watson distribution 136
5.4.2 Modelling the dispersion of fibre orientations with a Bingham distribution 138
5.4.3 The Ball & Rackets model 140
5.5 Comparing dispersion models and implementations 140
5.5.1 Comparing NODDI-Watson to AMICO 141
5.5.2 Comparing NODDI-Bingham implementations 146
5.5.3 Comparing NODDI-Bingham with ball & rackets 152
5.5.4 Crossing fibres and dispersion of fibre orientations 155
5.5.5 Model Selection 158
5.6 Discussion 159
Appendix 5A: NODDI-Watson implementation 166
Appendix 5B: NODDI-Bingham implementation 168

6 Probabilistic Tractography and Connectomes on GPUs 171

6.1 Introduction. Challenges of a parallel tractography design 175

6.2 Designing a GPU-accelerated probabilistic tractography framework 177

6.2.1 Framework functionality 177

6.2.2 Surfaces for performing tractography 179

6.2.3 GPU parallel design and strategies 180

6.2.4 Implementation considerations 191

6.3 Results: Anatomical Constraints in Tractography 194

6.3.1 Using pial surfaces 194

6.3.2 Advanced termination anatomical constraints 194

6.4 Results: Performance gains using the GPU implementation 196

6.4.1 Reconstructing white matter tracts 197

6.4.2 Generating dense connectomes 200

6.5 Validation: Comparison of GPU to CPU designs 201

6.6 Discussion 205

Appendix 6A: Streamline-surface mesh intersection 210

7 Conclusion 213

7.1 Summary and conclusions 213

7.2 Future perspectives 217

Bibliography 221

List of Algorithms

4.1 Levenberg-Marquardt algorithm 68
4.2 Random-walk Metropolis algorithm 69
4.3 Ball & sticks fitting process 70
4.4 Parallel version of MCMC for fitting a dMRI model 73
4.5 Parallel version of MCMC for fitting a dMRI model including a second level of parallelisation 76
4.6 Parallel version of Levenberg for fitting a dMRI model 82
4.7 Computation of partial derivatives in Levenberg algorithm 83
4.8 Parallel version of Levenberg for fitting a dMRI model including a second level of parallelisation 84
6.1 Probabilistic tractography algorithm 176

List of Figures

1.1 Methods for studying the white matter: microscopy vs. diffusion MRI 3

1.2 MRI tractography of my brain: reconstruction in 3D of some major tracts 4

2.1 Evolution of processors since 1971 11

2.2 Flynn's taxonomy 13

2.3 Distributed memory and Shared memory computing architectures . 14

2.4 Classic 5-stages pipeline for exploiting Instruction Level Parallelism 16

2.5 CPU-GPU interconnection 23

2.6 General overview of a GPU. 23

2.7 Streaming Multiprocessor in the NVIDIA Kepler microarchitecture 25

2.8 NVIDIA's Streaming Processor or CUDA core 26

2.9 GPU memory hierarchy 27

2.10 L1-cached and uncached global memory access patterns 37

2.11 Shared memory access patterns. 38

3.1 Diffusion as a Gaussian process in three dimensions 40

3.2 Diffusion process within different brain tissues 42

3.3 Pulsed gradient spin echo sequence 42

3.4 Example of diffusion tensors with their corresponding ellipsoids . . 45

3.5 Parameters derived from the diffusion tensor: principal fibre orientation, MD & FA 47

3.6 Examples of complex fibre configuration 49

3.7 Intra-axonal and extra-axonal diffusion compartments 51

3.8 Streamline tractography 54

3.9 Cones of uncertainty of the fibres orientation 56

3.10 Probabilistic tractography 57

3.11 Generation of a human connectome 59

4.1 Portions of time spent on the routines for fitting the ball & sticks model . . . 71

4.2 Parallel design for fitting dMRI models on GPUs V1 . . . 73

4.3 Distribution of the evaluation of several measurements amongst threads within a CUDA block . . . 77

4.4 Parallel design for fitting dMRI models on GPUs V2 . . . 77

4.5 Description of MCMC steps in a GPU parallel design . . . 78

4.6 Parallel design for fitting dMRI models on GPUs V3 . . . 80

4.7 Testing configurations of warps-per-voxels and voxels-per-block for running MCMC on GPUs . . . 80

4.8 Portions of time spent on the routines for fitting the ball & sticks model after parallelising MCMC . . . 81

4.9 Description of Levenberg algorithm steps in a GPU parallel design . 85

4.10 Portions of time spent on the routines for fitting the ball & sticks model after parallelising MCMC and Levenberg . . . 86

4.11 Performance reported by NVIDIA Profiler executing the MCMC kernel 87

4.12 Execution times of MCMC routine processing a low-resolution slice: single CPU core vs. single GPU . . . 93

4.13 Execution times of MCMC routine processing a high-resolution slice: single CPU core vs. single GPU . . . 94

4.14 Execution times of Levenberg routine processing a low-resolution slice: single CPU core vs. single GPU . . . 95

4.15 Execution times of Levenberg routine processing a high-resolution slice: single CPU core vs. single GPU . . . 96

4.16 Execution times fitting a dMRI model in a whole dataset: single CPU core vs. single GPU . . . 97

4.17 Execution times fitting a dMRI model in a whole dataset: CPU multi-core vs. multi-GPU . . . 98

4.18 Comparison of estimates fitting the ball & sticks model: sequential CPU vs. parallel GPU . . . 100

4.19 Distribution of standard deviations of estimated parameters fitting the ball & sticks model: sequential CPU vs. parallel GPU . . . 101

4.20 Comparison of maps with the estimated principal diffusion direction: sequential CPU vs. parallel GPU . . . 102

4.21 Comparison of maps with the uncertainty of the principal diffusion direction: sequential CPU vs. parallel GPU . . . 102

4.22 Parallel design for fitting Rubix model on GPUs . . . 107

5.1 Factors that make computational dMRI a time consuming process . 116

5.2 General design of CUDA Diffusion Modelling Toolbox 118

5.3 Example of specification of a new model in cuDIMOT 121

5.4 Example of the use of CUDA shuffle instructions 125

5.5 Parallelisation of LU decomposition across the threads within a warp 128

5.6 Transformations used in the Levenberg-Marquardt algorithm for imposing parameters bounds 130

5.7 Execution times fitting a dMRI model: sequential CPU tool vs. model-specific GPU solution vs. cuDIMOT 131

5.8 Validation of cuDIMOT comparing it with a sequential CPU tool and with a model-specific GPU solution: mean estimates 133

5.9 Validation of cuDIMOT comparing it with a sequential CPU tool and with a model-specific GPU solution: distribution of standard deviations of recorded samples 134

5.10 Example of 4 fibre orientation dispersion scenarios 136

5.11 Fitting the NODDI-Watson model: comparison of NODDI-Matlab, AMICO and cuDIMOT 143

5.12 Correlations between NODDI-Matlab and AMICO, and between NODDI-Matlab and cuDIMOT, fitting NODDI-Watson 145

5.13 Fitting the NODDI-Bingham model: comparison of NODDI-Matlab and cuDIMOT 147

5.14 Correlations between NODDI-Matlab and cuDIMOT fitting NODDI-Bingham 148

5.15 Fitting NODDI-Bingham model to synthetic data and comparing NODDI-Matlab and cuDIMOT 150

5.16 Cross-validation test with real data fitting NODDI-Bingham model: comparison of NODDI-Matlab and cuDIMOT 151

5.17 Maps with results from the ball & rackets model 153

5.18 Comparison of NODDI-Bingham and ball & rackets models being fitted to synthetic data 154

5.19 Cross-validation test with real data comparing NODDI-Bingham and ball & rackets models 155

5.20 Comparison of NODDI-Bingham model and an extension of the model with two Bingham distributions 157

5.21 Model Selection: comparison of diffusion MRI models (fibre orientation dispersion) 159

5.22 Map with significant differences between NODDI-Matlab and cuDIMOT fitting NODDI-Bingham model 163

6.1 Connectivity matrices modes offered by the GPU-accelerated tractography framework . . . 179

6.2 GPU parallel design of a probabilistic tractography framework . . . 181

6.3 Pipeline of the GPU-accelerated probabilistic tractography framework 183

6.4 Divergence in the number of propagation steps in probabilistic tractography . . . 186

6.5 Overlap of several pipelines in the GPU-accelerated tractography framework . . . 188

6.6 Execution times of the GPU tractography framework using different strategies for mitigating thread divergence . . . 190

6.7 Strategy for reducing segment-triangle intersection checks . . . 192

6.8 Example of the use of surfaces for imposing anatomical constraints . 194

6.9 Options for defining advanced termination masks in the GPU tractography framework . . . 195

6.10 Example of the use of advanced termination masks: Connectivity from Thalamus to Cortex and to Subcortical areas . . . 196

6.11 Execution times and speedup reconstructing different white matter tracts on a GPU . . . 199

6.12 Execution times and speedup reconstructing a total of 27 tracts on a GPU . . . 200

6.13 Execution times and speedup generating a dense connectome on a GPU . . . 201

6.14 Correlations and run-rerun variability of CPU-based and GPU-based probabilistic tractography frameworks . . . 203

6.15 Comparison of tracts reconstructed with CPU-based and GPU-based frameworks . . . 204

6.16 Comparison between CPU and GPU frameworks generating a dense connectome . . . 204

6.17 Reconstruction of the Acoustic Radiation with tight exclusion and termination masks . . . 208

6.18 Number of samples needed for achieving convergence in the generation of dense connectomes . . . 209

6A.1 Method for detecting streamline-surface intersections . . . 211

List of Tables

2.1 Main features of NVIDIA microarchitectures 22

2.2 Number of processing units in Streaming Multiprocessors 26

2.3 Limits in the allocation of resources on NVIDIA GPUs 33

3.1 Execution times and memory requirements of some dMRI applications 60

4.1 Major features of NVIDIA Tesla K80 GPU accelerator 89

4.2 Major features of Intel Xeon E5-2680 v3 processor 90

4.3 Accelerations achieved by a parallel design running MCMC: single CPU core vs. single GPU 93

4.4 Accelerations achieved by a parallel design running Levenberg: single CPU core vs. single GPU 95

4.5 Accelerations achieved by a parallel design running a whole model fitting application: single CPU core vs. single GPU 97

4.6 Accelerations achieved by a parallel design running a whole model fitting application: CPU multi-core vs. multi-GPU 98

5.1 Speedups obtained by cuDIMOT fitting the ball & sticks model . . 132

5.2 Major features of Intel Xeon E5-2660 v3 processor 141

5.3 Imaging protocol and parameters for generating synthetic data . . . 149

5.4 Model Selection: comparison of diffusion MRI models (fibre orientation dispersion) 159

5.5 Speedups obtained by cuDIMOT fitting several dMRI models . . . 162

6.1 Parameters for reconstructing major white matter tracts 198

Glossary

API . . . . . Application Programming Interface. Interface with a set of specifications for allowing the interaction between software applications and facilitate the use of software and hardware services.

Bandwidth . . Maximum amount of data that can be transferred per second between source and destination.

Deadlock . . . Undesirable situation where not all the threads of a concurrent system reach correctly a synchronisation barrier, getting stuck indefinitely.

FLOPS . . . . Floating point operations per second. It the most common unit to measure the performance in scientific computing.

Livelock . . . . Similar situation to deadlock where threads continue changing the states of the barriers indefinitely without progressing.

Race condition Undesirable situation in a concurrent system where threads access to some shared data at incorrect steps of an application because a lack of coordination between them.

Thread-block Group of threads in CUDA which are mapped to the same Streaming Multiprocessor (SM). It comprises one or several Warps (decided by the programmer). Threads within a block share the SM resources, can communicate between them and can be synchronised.

Throughput . Maximum amount of units that can be processed in a given amount of time.

Warp . . . . . An indivisible group of 32 threads in CUDA that execute always the same instruction in a SIMT (Single Instruction Multiple Threads) fashion.

1

Introduction

Contents

1.1 Organisation of the thesis 5
1.2 Software 8

Over the last decades, scientific computing in biomedical applications has been key for advancing medical research. Computational approaches for sequencing the human genome, for processing medical images, or for simulating biochemical processes are some of the examples. Nowadays, projects that involve large amounts of data are becoming more common: the Human Genome Project [1] or the Human Connectome Project [2–4] has generated Petabytes of data. Clearly, the trend in biomedicine is towards the use of *Big Data* for performing exploratory research, extracting information from the population, and for improving the understanding of diseases and ultimately healthcare [5, 6]. The importance and use of scientific computing frameworks for analysing this data, built upon principles from statistical modelling, artificial intelligence or machine learning, has increased dramatically in recent years [7].

The relatively inexpensive high performance computing resources that nowadays are available, allow scientists to perform computations and process this huge amount of data. Distributed and parallel computing architectures offer a very large amount

of computing power. A popular device with a massively parallel architecture (thousands of small cores) is the graphics processing unit (GPU), which has become one of the most cost-efficient component of high performance computing (HPC) systems. Mainly driven by the computer game industry, and more recently by deep learning applications [8], GPUs have evolved rapidly in the last decade, and now offer over 10 $TeraFLOPS$ (10^{13} floating operations per second) of performance [9]. Even if their full potential is not used, their suitability for scientific computing has become more and more evident. In this thesis, we illustrate the huge potential of GPUs for scientific computations on a number of problems that pose different types of challenges when designing GPU parallel solutions.

Specifically, we use GPUs within the context of computational neuroimaging. Brain mapping approaches using Magnetic Resonance Imaging (MRI) have revolutionised the study of the nervous system, the brain organisation, its structure and its connections. The mechanisms of the human brain are fascinating. With 100 billions of neurons and 100 trillions of connections between them, constantly modified and updated, the human brain is an extremely complex system [10, 11].

The long-range brain structure and connections can be studied directly by a number of techniques, including post-mortem dissections [12], histology, and chemical tracing in primate animals [13]. Magnetic resonance imaging (MRI) uniquely allows scientists to study the human brain, non-invasively and in vivo [14, 15]. Particularly, diffusion MRI (dMRI) [16] can be used for *estimating* tissue microstructure at a macroscopic scale. Diffusion MRI is an indirect method that quantifies the thermally-driven random motion (i.e. diffusion) of water molecules along different directions within brain tissue. The pattern of these motions is influenced by the structure of the tissue environment within which molecules diffuse, and thus the technique allows *inferring* information about the tissue microstructure [17]. Contrary to methods such as optical microscopy, which allows direct measurements of quantities of interest at a very small scale (see Figure 1.1), dMRI allow whole-brain explorations, which however is an indirect method and its

measurements need to be mapped to the biophysical parameters of actual interest. These biophysical parameters have proven to be useful both in neuroscience [18] and clinical applications [19–21]. Nevertheless, it is this mapping that leads to the need for *computational methods* and *modelling*.

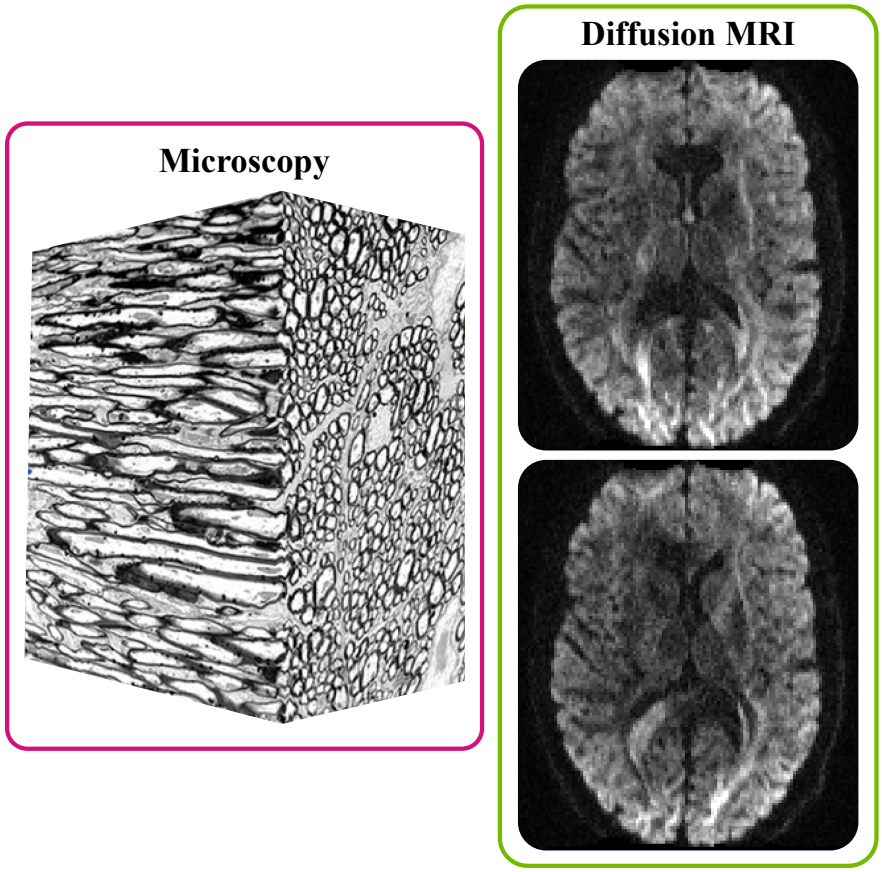

Figure 1.1: Microscopic three-dimensional reconstruction of the rat white matter (Reproduced with permission from [22]), and two diffusion weighted images of a human brain.

Apart from microstructure, diffusion MRI further permits the estimation of long-range connections in the brain. A family of algorithms known as tractography [23] are able to reconstruct in 3D the white matter tracts [24], i.e. the pathways through which information is transferred in the brain between remote areas (see Figure 1.2). The method allows the study of anatomical connections in health and disease. Brain connectivity can be affected by pathology. For instance, anatomical brain connectivity has been shown to be different in groups with Alzheimer's disease [25], multiple sclerosis [26], and Major Depression [27] compared to healthy controls. The

reconstruction of white matter pathways of a patient can also guide neurosurgeons in presurgical planning and intraoperative neuronavigation [28–30]. Similarly to tissue microstructure, brain long-range connectivity is estimated from diffusion MRI measurements through the use of computational approaches and models.

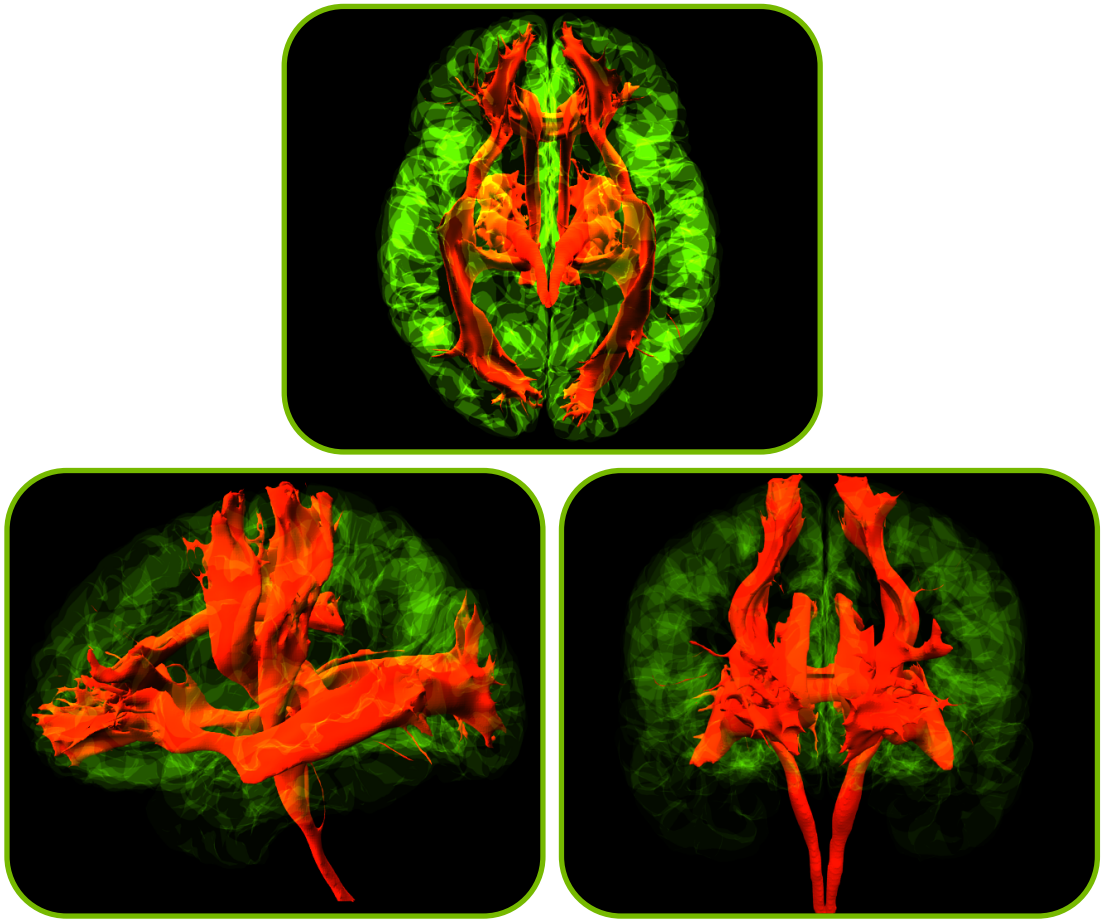

Figure 1.2: Reconstruction in 3D of some major tracts of my brain (Corticospinal tract, Superior thalamic radiation, Cingulate gyrus, Forceps minor, Posterior thalamic radiation and Inferior fronto-occipital fasciculus) using the diffusion tools developed in this thesis.

The potential of *computational dMRI methods* used for analysing dMRI data is great and it has been acknowledged by recent cornerstone studies, such as the Human Connectome Project (HCP) [2–4], the developing Human Connectome Project [31], and the UK Biobank project [32, 33]. All these large-scale projects use dMRI as one of the core imaging modalities for extracting anatomical brain connectivity maps for a large number of subjects, aiming to provide accurate

human connectomes (full brain connectivity structure), in health and disease, in development, in adulthood and in ageing.

Despite the great potential and the wide use of dMRI, the analysis typically includes computationally demanding frameworks. For instance, fitting certain biophysical models to data for studying tissue microstructure can take hours to days for a single subject. Tractography methods that estimate whole-brain anatomical connectivity may also need hours of computations even on a large computing cluster. For clinical applications the large computational times can be prohibitive for routine use. For data mining applications that process large dMRI databases, such as data from the HCP, large computational times can limit the potential exploration.

In this thesis, we propose parallel computational frameworks for the analysis of dMRI data using graphics processing units. These frameworks can achieve significant accelerations that change the perspective of what is computationally feasible and allow us to extract the most out of the latest state-of-the-art datasets. Specifically, we propose generic parallel designs for inferring locally tissue microstructure parameters, we implement these designs on GPUs achieving accelerations of more than two orders of magnitude, and we perform explorations of tissue microstructure using these implementations. We also present parallel GPU designs for performing tractography; even if these have their own challenges in terms of design and implementation and are not inherently fitted for GPUs, we show that we can still achieve excellent performance.

1.1 Organisation of the thesis

This thesis is organised in six chapters. The first two background chapters (Chapter 2 and Chapter 3) give an overview of the two main subjects of the thesis: GPU programming and diffusion MRI. The next three chapters present the contribution of the thesis, where parallel GPU frameworks are developed for

accelerating computational dMRI methods and applications are presented. The last chapter summarises the thesis.

Specifically:

Chapter 2 introduces the different types of parallel architectures and the types of parallelism that each can exploit. A guide of the best strategies for parallelising a scientific application is presented, and the GPU architecture and programming concepts are explained, including the main components of a GPU device, the memory hierarchy, the CUDA programming model and the CUDA execution model. The main challenges for designing and implementing GPU solutions are also presented, including challenges for exposing parallelism, avoiding thread divergence, and optimising GPU memory accesses.

Chapter 3 gives an overview of diffusion MRI techniques and computational methods used for analysis. The diffusion process and the pulsed gradient spin-echo sequence are introduced. The diffusion tensor imaging (DTI) framework for locally estimating the diffusion profile from dMRI is presented. Subsequently, the chapter introduces methods for tackling the limitations of DTI, including approaches for resolving complex fibre configurations, such as crossing. Finally, the chapter gives an overview of the different types of tractography algorithms, explaining their limitations and capabilities, and methods for generating brain connectomes.

Chapter 4 proposes a parallel GPU framework for fitting a specific dMRI model that estimates locally the fibre orientations, the ball & sticks model. The chapter presents an analysis of the computational routines used for fitting this dMRI model, including the Levenberg-Marquardt algorithm, and a Bayesian inference method, Markov Chain Monte Carlo (MCMC). Different designs and GPU implementations are proposed for parallelising these routines and their limitations and performance are presented. The parallel implementations are validated by comparing them to CPU implementations. Extensions of the parallel design to other dMRI models are

presented, and cases where this GPU-accelerated solution is used in Big Data applications are shown.

Chapter 5 proposes a generic toolbox for performing model fitting and non-linear optimisation (deterministic or stochastic) using GPUs. The framework is targeted at dMRI microstructure models, but in fact works for any voxel-wise MRI model. It comprises a front-end that allows the users to specify a model and a cost function using the C language. It then uses this input to automatically generate a CUDA binary that allows the user to fit the specified model via a number of fitting options, including Levenberg-Marquardt and MCMC. The different options of the framework, which make it generic and flexible, are shown, including the possibility of using priors and constraints in the model parameters and the possibility of fitting different models in *cascade*. The framework is used for exploring and augmenting a number of diffusion microstructure models. Several diffusion models that characterise fibre fanning and fibre orientation dispersion are implemented using the GPU framework, and a whole brain comparison is performed via model selection approaches. Finally, extensions of these microstructure models are proposed.

Chapter 6 proposes a GPU framework for performing probabilistic tractography, which presents considerably different challenges from the frameworks developed in previous chapters. These challenges are discussed and tackled, including high memory requirements and thread-divergence. Specifically, a solution that combines CUDA and OpenMP is presented. The chapter describes the extensive functionality incorporated in the tractography GPU framework, including the ability to handle both volumes and surfaces, options that allow large flexibility in imposing anatomical constraints, and options for generating dense connectomes (connectivity between tens of thousands of brain locations). The chapter shows the improvements obtained in tractography when using the extra functionality supported by the GPU framework. We also evaluate the number of samples that are needed per seed point to achieve convergence when generating a dense connectome. The performance achieved by the

GPU solution is shown and validation is performed comparing the GPU framework to a CPU probabilistic tractography toolbox.

Finally, **Chapter 7** summarizes the thesis, presenting conclusions, and discussing future perspectives.

1.2 Software

The GPU-accelerated solutions presented in this thesis were developed with the C\C++ [34, 35] programming language, CUDA [36] and the OpenMP [37] API. Validation of the GPU solutions were performed comparing these solutions to analysis tools included in the FMRIB's Software Library (FSL) [38]. Tools from the FSL library were used for brain extraction [39], tissue segmentation [40], and visualisation (FslView and Fsleyes image viewers were used). For visualisation we also use the Connectome Workbench tool [41]. For performing modelling simulations we used Matlab R2016a [42]. For generating statistics results we used Matlab R2016a [42] and the Python development environment Spyder 3.1.2 [43]. For editing figures we used Inkscape [44] and for writing this document we use the LaTex editor Texmaker [45]. The developed tools are (or will soon be) part of the public releases of FSL [38] and are already used in many large-scale projects (Human Connectome Project[2–4], developing Human Connectome Project [31] and the UK Biobank [32, 33]).

2

GPU programming

Contents

2.1 Parallel computing 10
2.1.1 Historical review and motivation 10
2.1.2 Parallel computing architectures and programming models 12
2.1.3 Types of parallelism 16
2.2 Scientific parallel computing: typical considerations and guidelines 17
2.3 Graphics Processing Units (GPUs) 21
2.4 The GPU architecture 22
2.4.1 General overview and CPU-GPU interconnection 22
2.4.2 The Streaming Multiprocessor 23
2.5 GPU memory hierarchy 26
2.6 The CUDA programming model 28
2.6.1 Thread organisation 28
2.6.2 Threads and memory synchronisation 29
2.7 The CUDA execution model 30
2.7.1 Warp execution 30
2.7.2 Latencies 31
2.8 Challenges in GPU programming 32
2.8.1 Exposing parallelism and increasing device occupancy . 32
2.8.2 Avoiding warp divergence 34
2.8.3 Memory access patterns 35

2.1 Parallel computing

2.1.1 Historical review and motivation

In the last five decades, processor technology and the field of computational science have evolved extremely rapidly. The integration of different techniques has allowed the development of better devices for performing complex mathematical calculations and increasing the storage capacity [46]. Over this period, both hardware and software have undergone big transformations. Nevertheless, software has always progressed at slower pace than hardware given the life cycle difference: tens of years (software) versus 2-4 years (hardware), i.e. software is not updated at the same pace than hardware. Moreover, some experts claim that "Software is getting slower more rapidly than hardware becomes faster" (Wirth's Law) [47]: due to the increase in the availability of hardware resources, software developers tend to add more functionality to their applications, and therefore complexity.

Since the late 1960s, when the first microprocessor was conceived, the number of transistors in a single microprocessor has roughly followed Moore's Law [48], doubling the number of transistors per square inch on integrated circuits every 18-24 months (see Figure 2.1). The first microprocessors had a single processing unit, or *core*, and by increasing its clock frequency and improving the computational pipelines, a higher performance was trivially obtained. This strategy has been used for many years. However, the increase in the clock frequency causes a significant rise of power consumption and chip temperature. In the early 2000s, this reached an unsustainable level. Manufacturers started to look for alternatives to continue with the improvement of performance, which led to the emergence of multiprocessors [49]; the aggregation of few low-frequency processors could reach the same performance as a high-frequency processor with less power consumption [50].

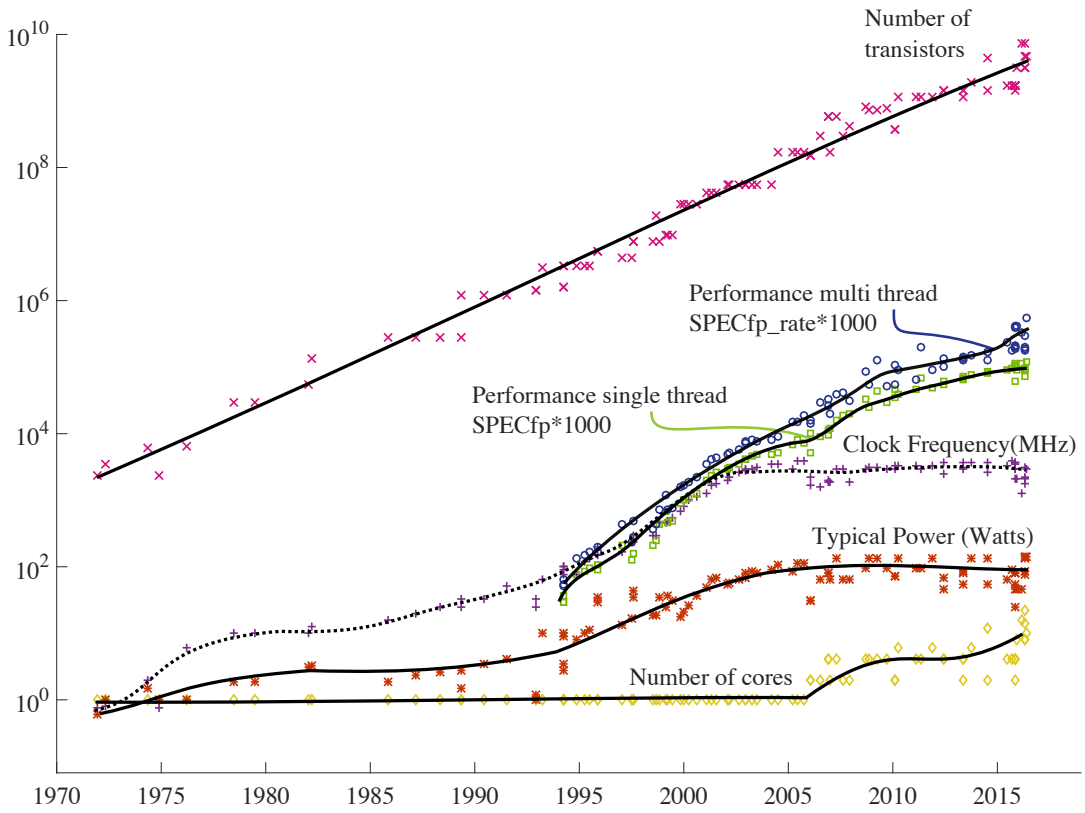

Figure 2.1: Evolution of processors since 1971. The figure shows the number of transistors, the base clock frequency (in MHz), average power (in $Watts$) and the number of cores of 99 Intel processors. It also shows the performance score obtained when running SPEC's benchmarks [51].

Over most of these years, the programming models have followed the *von Neumann* machine model. In this model the instructions are executed sequentially [52]. A sequential application may be executed faster without code modifications as a result of hardware improvements, such as the increase of microprocessors' clock frequency, the improvement of pipeline designs, the use of speculative and out-of-order instruction execution strategies or the enhancement of the Floating Point Units (FPUs) and memory hierarchies. Thus, the programming models did not evolve much. It was with the emergence of multiprocessors when the *parallel programming* models flourished, allowing the programmers to manage and coordinate several processors [53].

Computers are an indispensable tool for performing calculations in many fields, such as engineering, physics, finance, biology or medicine. Therefore, efforts have

been made to increase their performance, which directly affects common scientific tasks, such as modelling, simulations and analysis [54]. Scientific computing allows scientists to perform research that otherwise would not be possible. This often involves complex models and large amounts of data that need to be processed efficiently, normally, using High Performance Computing (HPC). Nowadays, it is not uncommon for scientific projects to generate *Tera*-to-*Petabytes* of data that need to be computationally processed in a reasonable amount of time [32, 55]. To face these *Big Data* problems, parallel computing architectures have become essential. Compared, however, to the more straightforward von Neumann model, parallel computing architectures present more challenges for programmers, and some knowledge about software and hardware is needed for using these resources appropriately.

2.1.2 Parallel computing architectures and programming models

The two most common classes of multiprocessors are the *multi-core* processors, with few cores on the same chip, and the *many-core* processors, with a large number of simpler cores [56, 57]. Using multi-core or many-core processors several instructions and/or several data streams can be computed in parallel. Flynn's taxonomy distinguishes four classes of computer architectures according to the number of instruction streams and data streams that can be computed simultaneously [58] (see Figure 2.2):

- *Single Instruction stream over Single Data stream* (**SISD**). This class represents the typical von Neumann machine model. Instructions are seen as if they were executed sequentially over scalar data. The dominant programming languages for HPC in this class of computers are Fortran, C and C++, which were designed for sequential machines (they need extensions for exploiting the benefits of parallel architectures). Interpreted languages, such as Matlab,

Python and R, are an alternative. Interpreters are easier to use for scientists, but they typically give a much lower performance than compiled languages.

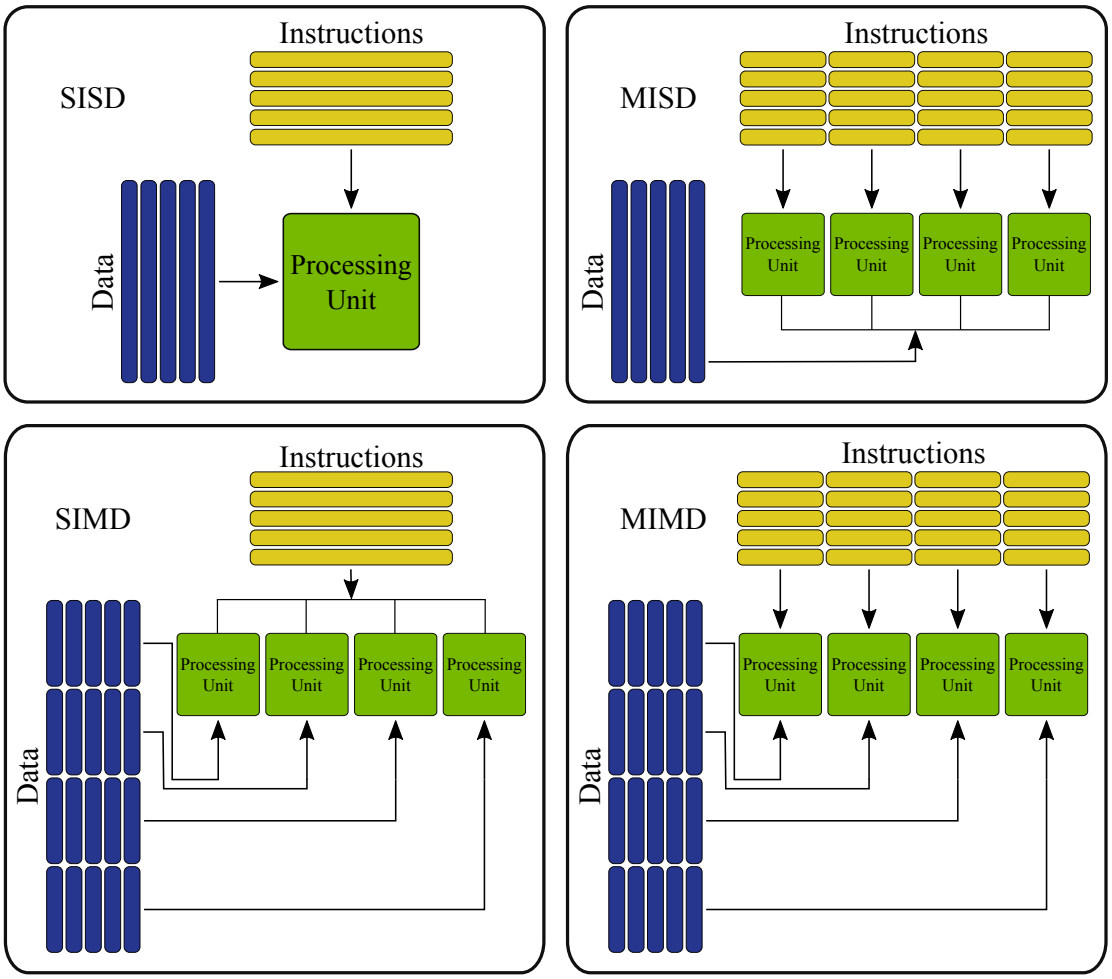

Figure 2.2: Flynn's taxonomy distinguishes four classes of computer architectures according to the number of instruction streams and data streams that can be computed simultaneously.

- *Multiple Instruction streams over Single Data stream* (**MISD**). This architecture has not been used commercially.
- *Multiple Instruction streams over Multiple Data streams* (**MIMD**). This class represents an execution model with different instructions running simultaneously over different pieces of data. We can divide it into two subclasses (see Figure 2.3):
  - Distributed memory architectures. Each processor has a private portion of memory. If one processor needs data from another portion of memory,

it needs to establish a communication channel and interchange messages. Clusters with several nodes belong to this category. These systems can easily scale by increasing the number of nodes. Message Passing Interface (MPI) [59] is the dominant programming standard used in these architectures, however, it is slow and tedious to code.

– Shared memory Architectures. In these architectures, all the processors of a system use a common memory space (Shared memory), which is also used for communication. Multi-core processors, the most common processors in desktops and laptops, belong to this category. Some many-cores systems, such as the new Knights Landing Intel Xeon Phi coprocessor (up to 72 cores) [60], also belong to this category. The scalability of these systems is limited to the number of cores of a node. Open Multi-Processing (OpenMP) [37] is the dominant programming standard interface used on these computers and it is simpler than MPI.

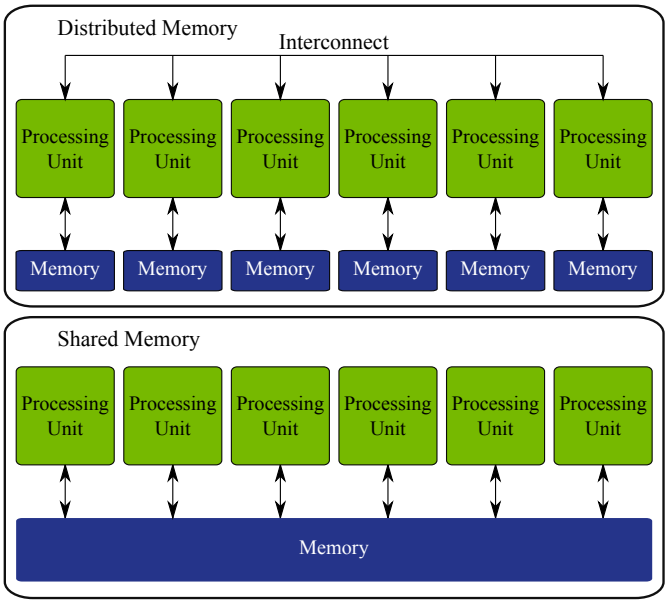

Figure 2.3: Comparison of distributed memory and Shared memory computing architectures.

- *Single Instruction stream over Multiple Data streams* **(SIMD)**. In this architecture a single instruction is executed over different data (typically a small vector or array comprising between 4-16 scalars) simultaneously. SIMD was common in the supercomputers of the 1970s [61] but nowadays is used inside

the functional units of any modern processor. Compilers may exploit this feature, for instance, when there is a loop that iterates over a vector. Graphics Processing Units (GPUs) are an epitome of this category with very large vectors or arrays; an instruction is executed by multiple threads (**Single Instruction Multiple Threads, SIMT**) over different data. Compute Unified Device Architecture (CUDA) [36], NVIDIA's proprietary, and Open Computing Language (OpenCL) [62] are common standards used for programming GPUs. These standards have a very low level of abstraction. They are challenging to use because the programmer needs to have some knowledge of the GPU architecture. Open Accelerators (OpenACC) [63] is an alternative with a higher-level of abstraction. It is quite similar to OpenMP, but the main issue is the little control that the programmer has on the software-hardware interface. Normally, the performance obtained using OpenACC is much lower than using CUDA or OpenCL [64].

Nowadays, the trend in scientific computing is towards the use of clusters with heterogeneous architectures, i.e. a mix of Flynn's taxonomy classes. Heterogeneous systems are normally composed of several nodes, each node with several multi-core processors. A node may also have several GPU's, Intel Xeon Phi coprocessors [65] or even FPGAs (Field-Programmable Gate Arrays), devices that can be programmed to change the hardware design and be used as accelerators [66]. This kind of configuration makes the scalability of the systems easier [67]. The optimal choice depends on the type of scientific application. Normally, the combined use of all these resources leads to a better performance if load balancing is efficient, however, knowledge of all these computer architectures is required. For instance, for programming a cluster with several nodes and GPUs, a few programming languages may be used, MPI + OpenMP + CUDA [68]. Alternatives like OpenCL can offer a solution for programming heterogeneous architectures using a single programming language.

2.1.3 Types of parallelism

Having in mind Flynn's Taxonomy, it is possible to exploit three main classes of parallelism in a software application:

- **Instruction Level Parallelism (ILP)**. This parallelism can be exploited inside the cores of a processor. An instruction is divided into several steps before being executed (see Figure 2.4), and at each time point, different functional units of the processor's core are used to execute different instructions. Therefore, instruction execution can be overlapped [69]. ILP is present in the cores of any modern processor and is exploited by the compilers. The requirement is that every instruction must be independent of any other instruction that is being executed. Speculative execution and out-of-order instruction execution may increase ILP. All four architectures in Flynn's taxonomy can exploit ILP.

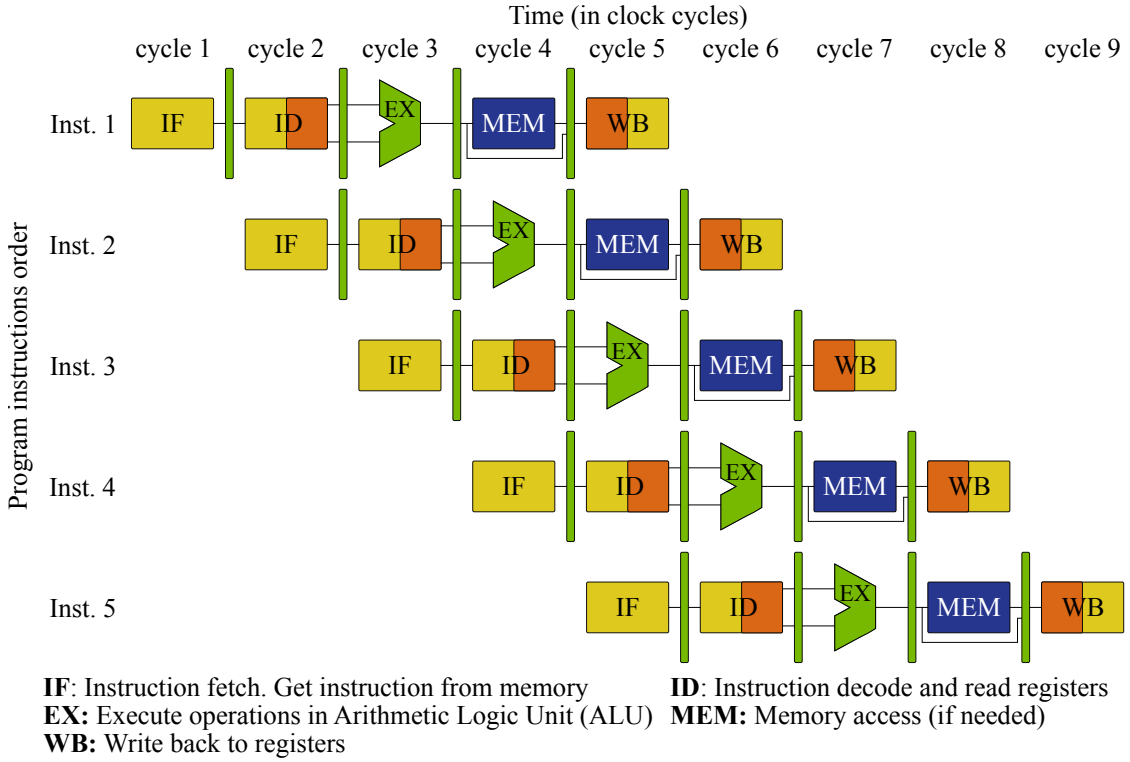

Figure 2.4: Example of Instruction Level Parallelism (ILP) with the classic 5-stages pipeline. The execution of an instruction is divided into several steps, and different instructions can be overlapped by using the different functional units of a core.

- **Data Level Parallelism (DLP)**. This parallelism is exploited when different pieces of data are processed in parallel. SIMD architectures exploit DLP, including Vector Processors, Array Processors and GPUs. These processors can execute the same instruction on all the elements of a vector (over time), an array, or different elements in memory respectively. Frequent limitations of this parallelism are the presence of scalar operations in the code, where the operation affects only to a component of the vector or the array, and divergence, where branches in the code leads to execute different instructions on the elements of a vector, which are sequentially computed.
- **Task or Thread Level Parallelism (TLP)**. At this level of parallelism, multiple threads are executed in parallel and each one can have a different set of instructions. Thus, they can perform different tasks. This parallelism is exploited by MIMD architectures, such as multi-core processors and clusters with several processing nodes.

2.2 Scientific parallel computing: typical considerations and guidelines

When designing a computationally demanding scientific application, a few typical considerations are:

1. *Is it worth speeding up my application using parallel computing?*
2. *Which type of architectures and programming languages can I use to accelerate my application?*
3. *Challenges in parallelising an application*

Let's go through these questions one by one.

Is it worth speeding up my application using parallel computing?

To answer this question the computational complexity of the application needs to be analysed. First, we need to identify which parts of the application can be

processed in parallel and which ones can only be processed sequentially. According to Amdahl's law [70], the maximum speedup that we can obtain using parallel computing compared with a sequential version is:

$$Maximum\ Speedup = \frac{1}{(1-f) + \frac{f}{C}} \tag{2.1}$$

where $0 \leq f \leq 1$ is the fraction of the application that is parallelisable, $1 - f$ is the fraction of the application that needs to be processed sequentially and C is the maximum number of parts that can be executed in parallel (or available cores/processing units). When f is small, not much improvement can be obtained using parallel computing.

Amdahl's law assumes that f is fixed regardless the size of the data of the problem. But f can increase as the size of the problem increases. Gustafson's law [71] takes this observation into consideration:

$$Maximum\ Speedup = (1-f) + C \times f \tag{2.2}$$

This law tells us two important things:

- We should start by parallelising the parts of the applications that scale worst with the size of the problem (usually the most expensive parts).
- Parallel computing is for solving *big* problems.

The maximum speedup obtained from these equations is just indicative. These laws do not take into consideration many factors such as the cost of synchronization, communication, load imbalance and many other overhead sources. However, they can give us an idea of what is the potential performance gain and guide us in the process of choosing a parallel architecture.

Which type of architectures and programming languages can I use to accelerate my application?

We can identify 3 types of patterns according to the number and complexity of the tasks in an application:

- The application is composed of a low number of complex and independent tasks ($C \in [2, 16]$ in Equation 2.2).
- The application is composed of a medium-large number of complex and independent tasks ($C > 16$ in Equation 2.2).
- The application executes the same pseudo-simple task(s) over thousands of different data instances.

Obviously, if an application is composed of a low number of simple tasks, there is limited opportunity for parallelisation. In the first two patterns of this classification the Task Level Parallelism can be exploited. If the application can be divided into few tasks, a single processor with few cores ($C \approx 8$) and OpenMP will be enough to reach the theoretical maximum speedup. If the number of tasks is larger, a cluster of multiprocessors and MPI, or some coprocessors such as the Intel Xeon Phi and OpenMP, can be used for exploiting the parallelism in the application.

When simple tasks need to be executed a very large number of times (>10,000) over different data instances, we can exploit massively the Data Level Parallelism. A cluster of multiprocessors could be used to exploit this parallelism, but there are two main issues with this approach. Firstly, the number of cores (normally a few hundred) in the cluster will limit the speedup. Secondly, multi-core processors are intended to perform complex tasks, and the sum of many processors can be economically very expensive and energy inefficient. Thus, there may be a huge waste of resources.

The best alternative for this kind of applications is the use of GPUs, which have a few thousand cores in a single chip for exploiting the Data Level Parallelism and are energy more efficient. However, there are some factors that can degrade the performance of an application running on a GPU, being the most fundamental:

- Not having enough independent tasks/threads.
- Assigning very heavy tasks to the threads.
- Irregular memory access patterns.
- Intense communication or synchronisation between threads.
- Many divergences in the code.

This thesis focuses on the use of GPUs for scientific computing, thus, these factors will be studied in the next sections of this chapter.

Challenges in parallelising an application

Parallel computing can be complicated. Parallel programmers need to deal with these aspects:

- Decompose the application into independent tasks.
- Map these independent tasks to threads and processing units.
- Thread management: creation and deletion.
- Memory management (as a shared resource between threads).
- Communication and synchronisation between threads.
- Possible errors: race conditions / deadlocks / livelocks.
- Debugging difficulties.
- Scalability, workload balance and maintainability.
- Portability and reusability.

Parallel computing is not simple, but very frequently scientific applications consume a lot of computational time. Without the use of parallel architectures it would not be possible to obtain the results in a reasonable time. As we will see in this thesis, the effort of using parallel computing can have a great reward and produce a tremendous impact on scientific data analysis.

2.3 Graphics Processing Units (GPUs)

Originally, GPUs were non-programmable devices consisting of few cores used for performing very simple graphics operations over images, such as filtering, shading, texture mapping or rasterisation. The high demand of real-time graphics has driven the advance of graphics hardware [72]. These applications generate tens of frames every second, and for each frame, million of vertices and thousand of pixels are processed. But these vertices and pixels are completely independent, and therefore they can be processed simultaneously by the different cores of a GPU.

In the early 2000s, with the release of new API libraries (Microsoft DirectX and OpengGL extensions), and the introduction of programmable shader units, the GPUs became programmable devices. The term GPGPU (General-Purpose computation on Graphics Processing Unit) was introduced [73]. Simple operations over vectors and matrices could be computed on GPUs. But these APIs were still intended for developing graphics applications and video games. The industry of computer games made a huge contribution to the increase of GPUs' performance. Gamers demand higher graphics quality all the time, which led to a competition between manufacturers and to an improvement of these devices.

It was not until 2007, with the release of CUDA (Compute Unified Device Architecture) [36] by NVIDIA, when GPUs became a fully programmable device. CUDA is a parallel computing platform and a programming model that allows the use of GPUs for general purpose computing. A few years later, in 2009, OpenCL (Open Computing Language) [62] was released. OpenCL is an open standard interface for programming heterogeneous architectures, including GPUs. With the introduction of CUDA and OpenCL, GPUs have been used in the last decade for many purposes, including High Performance Computing where industrial, economic, engineering or scientific applications have been accelerated [74, 75].

2.4 The GPU architecture

2.4.1 General overview and CPU-GPU interconnection

A GPU is a many-core device with its own memory spaces and connected to a host (CPU-based) device. It is not a standalone platform but a co-processor. It contains thousands of computing cores, although the instruction set of these cores is much simpler than those on CPUs. GPUs are capable of supporting thousands of threads running in parallel, reaching (at least theoretically) peak performances of up to ten $TeraFLOPS$ (ten trillion floating point operations per second) in single precision. Typical features and performances of five generations of NVIDIA microarchitectures are shown in Table 2.1 [9, 76–79].

GPU NVIDIA microarchitecture	Year	Max GFLOPS Single Precision*	Max GFLOPS Double Precision*	Max SMs
Tesla	2006	622.08	77.76	30
Fermi	2010	1,331.2	665.6	16
Kepler	2012	5,040	1,680	15
Maxwell	2014	6,844	213.9	24
Pascal	2016	10,609	5,304	56

Table 2.1: Main features of five GPU NVIDIA microarchitectures, including: name of microstructure, release year, maximum performance in single and double precision (*only GPUs with a single chip) and maximum number of Streaming Multiprocessors (SMs) per chip.

Typically, a GPU is connected to a host system via a Peripheral Component Interconnect express (PCIe) data bus [80], a high performance I/O bus used to interconnect peripheral devices (see Figure 2.5). This bus has been improved several times (last version is PCIe 3.0 with a peak bandwidth of 15.76 GB/s), however, it can be a bottleneck in some cases, especially in very memory intensive applications. NVIDIA has introduced a new interconnection bus in the new Pascal microarchitecture, NVLINK [9, 81], which is much faster than PCIe (the peak bandwidth is 80 GB/s) and can connect a GPU to a host, or, interconnect directly several GPUs.

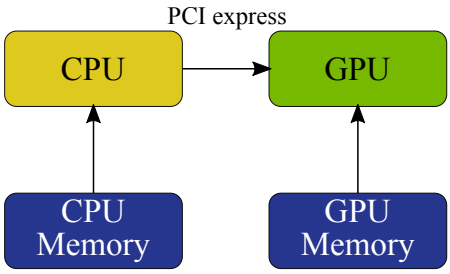

Figure 2.5: CPU-GPU interconnection via the Peripheral Component Interconnect express (PCIe) data bus.

A GPU consists of a set of SIMT (Single Instruction Multiple Threads) Streaming Multiprocessors (SMs)(see Figure 2.6). The number of SMs on a GPU depends on the microarchitecture and the chip (see Table 2.1).

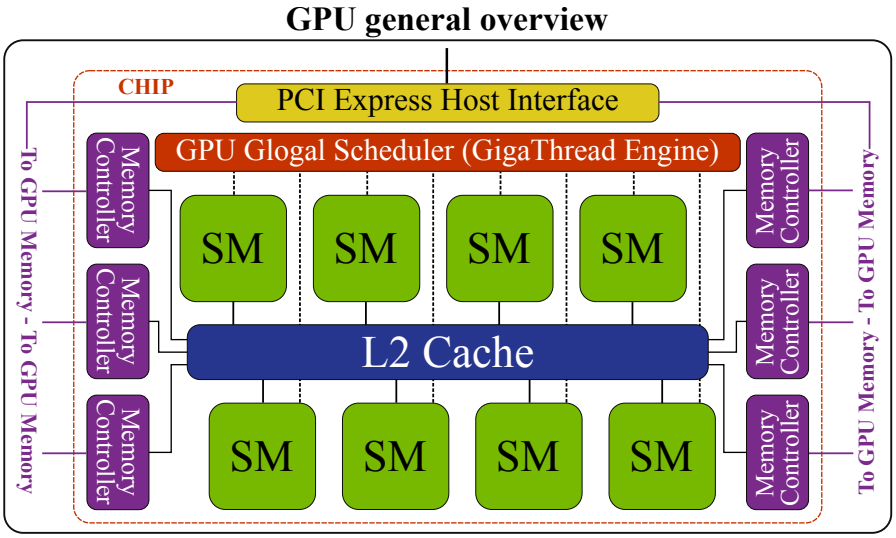

Figure 2.6: General overview of a GPU. A GPU chip is composed of a PCIe interface, a global scheduler, several Streaming Multiprocessors (SMs), L2 cache, and memory controllers.

2.4.2 The Streaming Multiprocessor

A Streaming Multiprocessor (SM) is composed of many different units (see Figure 2.7) [82]. We can classify them into three categories: processing units, memory resources and instructions dispatchers.

The most important processing units in an SM are the Streaming Processors (SPs) or CUDA cores (represented as green boxes in Figure 2.7 and shown in

detail in Figure 2.8). The SPs are responsible for most of the operations during the execution of a typical scientific application. Each of these cores has a fully pipelined integer arithmetic logic unit (ALU) and a fully pipelined floating-point unit (FPU). The use of pipelines allows these processors to exploit the Instruction Level Parallelism. They can execute one 32-bit operation per cycle. The number of cores has been increased from generation to generation (see Table 2.2), with a remarkable change from Fermi to Kepler. As the number of cores is increased, a lower clock frequency is used in these units for reducing the power consumption.

The capability of performing double-precision operations (64-bits), important for many scientific applications, has also been improved from generation to generation, especially since the introduction of dedicated double precision units in the Kepler microarchitecture (Maxwell was not intended for double precision applications). The last GPU generations offer a very high double precision performance (up to five $TeraFLOPS$ in the Pascal microarchitecture).

The load/store units fetch or save data from or to memory resources, being able to calculate the source and destination addresses of 16/32 threads per cycle. Within an SM there are also some special function units (SFUs), which have a hardware implementation of some mathematical operations, such as trigonometric functions ($\sin, \cos, \tan$, etc.). SFUs are faster than the software implementations versions of these operations, but normally they perform the operations with lower precision.

An SM also incorporates several memory spaces, warp schedulers and dispatch units. These are discussed in the following sections.

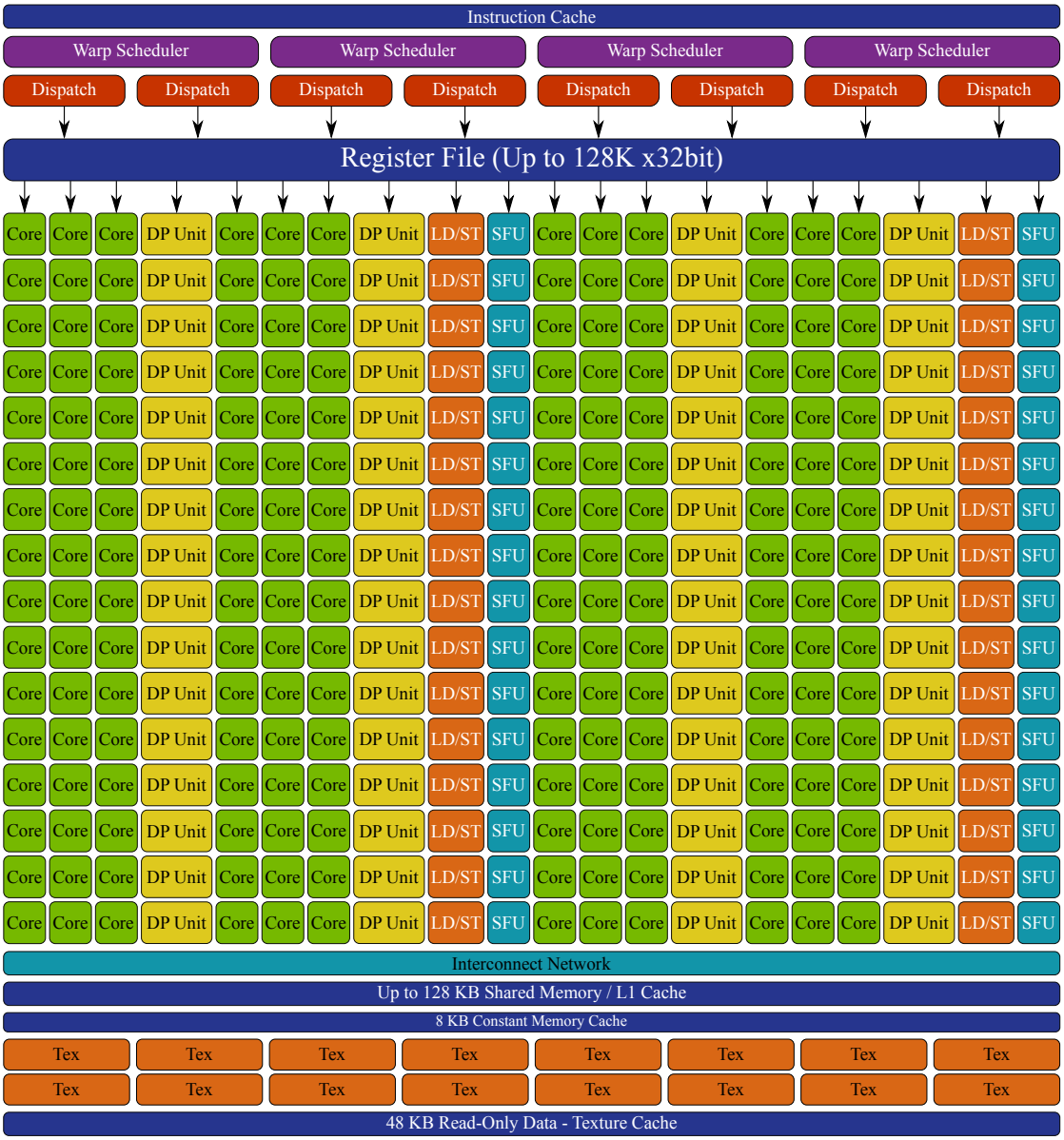

Figure 2.7: A Streaming Multiprocessor in the NVIDIA Kepler microarchitecture. **Processing units**: CUDA cores (32-bit), double precision units (64-bit, DP Unit), load/store units (LD/ST), special function units (SFU) and texture units (Tex). **Memory resources**: Instruction cache, register file, Shared memory/L1 cache, read-only/texture cache and Constant memory cache. An SM has also several warp schedulers and dispatch units to issue instructions to the processing units.

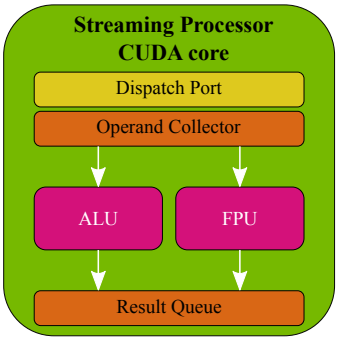

Figure 2.8: An NVIDIA's Streaming Processor (or CUDA core) with a fully pipelined integer arithmetic logic unit (ALU) and a fully pipelined floating-point unit (FPU). It can execute one 32-bit operation per cycle.

GPU NVIDIA Micro-architecture	Streaming Processors per SM	Dedicated Double Precision Units	Load/ Store Units	Special Function Units	Warp Schedulers -Dispatch Units	Texture units
Tesla	8	1	0	2	1-1	1/2
Fermi	32	0	16	4	2-2	4
Kepler	192	64	32	32	4-8	16
Maxwell	128	4	32	32	4-8	8
Pascal	64	32	16	16	2-4	4

Table 2.2: Number of processing units in the Streaming Multiprocessors of five NVIDIA microarchitectures. Note that SMs in Pascal GPUs has less processing units than in Kepler and Maxwell GPUs, however, Pascal GPUs have more SMs (see Table 2.1).

2.5 GPU memory hierarchy

The GPU's memory spaces are organised in a hierarchical way (see Figure 2.9). The lower the level in the hierarchy, the faster the memory access, but the smaller the storage space [82, 83].

At the highest level (level 3), we can find the global memory (also called device memory). This is the largest storage space on a GPU device (up to 24 *GB* in Pascal), which is shared by all the SMs. Physically, it is not located on-chip (see Figure 2.6) and the access to this memory is expensive compared to the access to any other memory space. Global memory has high latencies [84] and a low

bandwidth. In the new NVIDIA Pascal microarchitecture this memory is integrated on the chip, improving the bandwidth considerably [9].

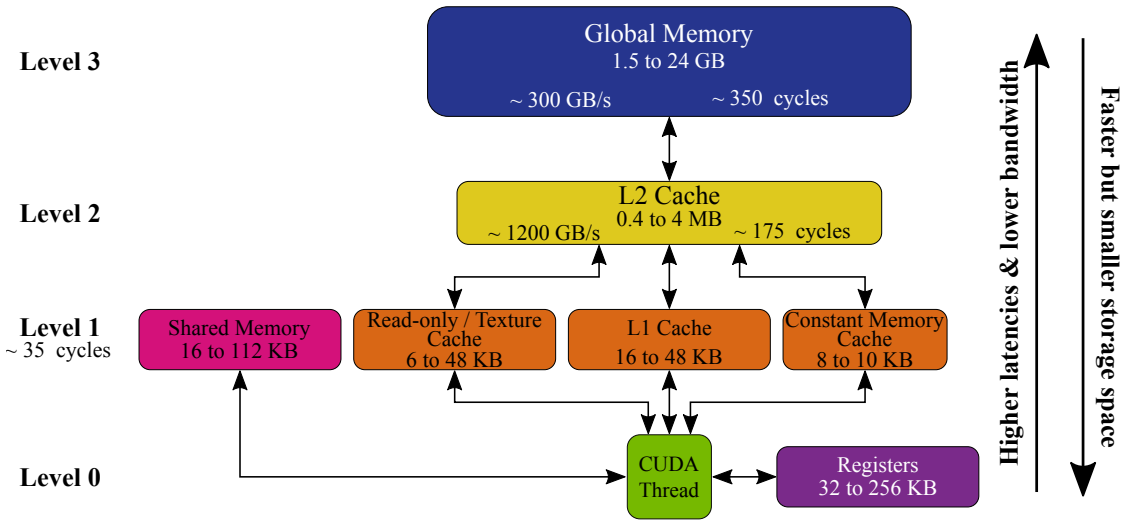

Figure 2.9: Different memory spaces and levels of the CUDA hierarchical memory system.

The non-programmable L2 cache is located at level 2. This coherent memory is located on-chip and it is shared by all the SMs. Any access to global memory goes through L2 cache (reads and writes).

At level 1 we find several cache memory spaces: L1 cache, read-only/texture cache, Constant memory cache and Shared memory. All these memory spaces are inside an SM and they are shared by the threads allocated in it. Shared memory is a programmable cache (managed by the programmer), whereas L1, normally located in the same hardware space than Shared memory, can be optionally activated for caching the read accesses to global memory. Depending on the application's requirements, the programmer can assign more space to Shared memory or to L1 cache. In general, GPU caches are smaller than CPU caches. While CPU caches are optimised for temporal and spatial locality, GPU caches are intended only for spatial locality (the cache is shared by many threads).

Finally, at the lowest level (level 0) we find the register file, which is the fastest and largest memory space inside an SM.

2.6 The CUDA programming model

The CUDA programming model [36] allows programmers to use the GPU architecture for general computing purposes. The main key of GPU programming is *multithreading*: thousands of threads are executed simultaneously for taking advantage of the GPU many-core architecture. In scientific computing it is very common to have a large amount of data that need to be identically processed. This large amount of data can be distributed among thousands of threads, and thus exploit the Data Level Parallelism. The programmer is responsible for designing this distribution process and using the GPU resources as efficiently as possible.

A CUDA application is executed from a CPU core (host) following these steps:

1. Initialise the GPU device.
2. Allocate memory in the host and in the device.
3. Transfer the data to be processed from the host to the device memory.
4. Launch multiple instances (threads) of an execution ***kernel*** on the device for processing the data.
5. Transfer the processed data from the device to the host memory.
6. Repeat 3-5 as many times as needed.
7. De-allocate host and device memory and finish.

A kernel is a function that is executed on the GPU by every thread (SIMT) launched from the host. Normally, each thread processes a different piece of data (SIMD). Depending on the algorithm requirements, the device resources, the parallelisability of the algorithm and the amount of data that need to be processed, the CUDA programmer decides how many threads are launched per kernel.

2.6.1 Thread organisation

CUDA is based on a hierarchy of abstraction layers. The whole set of threads that executes a kernel is called a *grid*. A grid is divided into groups of threads called

thread-blocks or simply *blocks*. A block is mapped to a single SM; thus, the threads that belong to the same block will share the SM's resources, including the processing units, registers, texture units, caches (excluding L2 cache) and Shared memory. The threads within a block can communicate efficiently through Shared memory and they can be synchronised if necessary. A CUDA programmer must decide the dimensions of the grid and blocks. This decision can have a high impact on the performance obtained by an application executed on a GPU (discussed in Section 2.8.1).

The threads within a block are grouped by the hardware into indivisible sets of 32 threads called *warps*. The warps are assigned simultaneously to an array of processing units and executed in a SIMT fashion (Single Instruction Multiple Threads), exploiting the Data Level Parallelism. The warp size (32 threads) imposes design constraints on the block and grid sizes, which ideally, for maximum occupancy of resources, should be a multiple-of-32 number of threads.

2.6.2 Threads and memory synchronisation

In some applications, at certain steps, it will be necessary to coordinate the execution of threads or the memory accesses. Synchronisation is an essential mechanism to guarantee the correct execution of an application and avoid race conditions.

If we want to synchronise the threads in a CUDA application, we need to set a barrier in the code. Depending on the level of synchronisation, some threads may need to wait for other ones at this barrier. This is useful for ensuring that a section of the application has completely finished before starting the next one.

In CUDA all the threads within a warp execute the same instruction. They are always at the same step of an application, thus, *synchronisation is implicit within a warp*. However, threads that do not belong to the same warp can be executing different steps of the application, and explicit synchronisation may be necessary. It is possible to define two types of barriers according to the required level of synchronisation:

- System synchronisation: The host executes this barrier, which is not very specific. It makes the host wait until all CUDA instances have finished; including any launched CUDA kernel and any memory transfers from or to GPU.
- Block synchronisation: A more specific barrier synchronises only the threads within a block. This barrier is defined inside a kernel and is executed by the CUDA threads. Note that if any thread of the block does not execute the barrier (the barrier may be inside a conditional branch with divergent paths), it may cause a deadlock.

To coordinate memory accesses, CUDA offers two options: atomic operations and memory-fences. Using atomic operations inside a kernel we can ensure that a value in memory is updated by only one thread at a time (updates are run sequentially). Memory-fences ensure that within a block, all the transfers to memory (Shared/global memory) before the fence are executed before any transfer after that fence.

2.7 The CUDA execution model

2.7.1 Warp execution

Once a CUDA kernel is launched, the GPU global scheduler (see Figure 2.6) distributes thread-blocks among the SMs. Depending on the available resources, an SM may allocate and run more than one block simultaneously. When a block is received in an SM, the memory resources needed by its threads (registers and Shared memory) are engaged. Then, the block is divided into warps.

Each SM has several warp schedulers and dispatch units (see Figure 2.7), which are in charge of issuing instructions to the processing units. A warp scheduler selects one warp for being issued at each clock cycle. The condition for a warp to be selected is that its next instruction is ready to be issued, i.e., its dependencies

have been solved and the operands are ready. When a dispatch unit issues a warp instruction, this instruction is mapped to an array of processing units (SIMT).

Note that the SMs have typically more than 32 (warp size) processing units. To use all of them simultaneously, more than one warp needs to be issued per cycle. Having more than one warp scheduler in an SM, several warps can be selected for using the processing units, and, having more than one dispatch unit (per warp scheduler), more than one instruction from a warp can be dispatched if they are independent. The purpose of this hardware strategy is to keep the SM's processing units as busy as possible, and exploit the Instruction Level Parallelism.

2.7.2 Latencies

Once an instruction is assigned to an array of processing units, it will take several cycles until the instruction is completed. The number of cycles to complete the instruction, called latency, will depend on the type of instruction and microarchitecture [82, 84]:

- Arithmetic instructions: approximately 10 cycles in Kepler for the most common operations (addition, subtraction and multiplication).
- Memory instructions: from 30 to 350 cycles depending on which memory space is accessed (see Figure 2.9).

If two instructions do not have any data dependency, the dispatch unit can issue them without any delay (ILP) and therefore hide these latencies [85]. However, it is very common to have scenarios with data dependencies between instructions, normally because one of the operands of the instruction is not ready. For instance:

Scenario 1:	**Scenario 2:**
1. $A = B \times C$	1. $A = load\_data\_from\_global\_memory$
2. $D = A \times E$	2. $B = A \times C$

In such scenarios, where the second instruction needs to wait for operand A, the warps will be held for many cycles, approximately 10 cycles in Scenario 1 and 350

cycles (if global memory access is not cached) in Scenario 2. In these cases, the only way to keep the processing units busy and *hide the latencies*, is having other warps with instructions ready to be issued, and switch between warps at execution time.

2.8 Challenges in GPU programming

Parallel programming is not straightforward and the achievement of optimal performance can be challenging. Programming GPUs can be even more complicated given the high degree of low-level control that a programmer has over the execution of an application. But this high degree of control is what can make a CUDA implementation extremely fast. In this section we describe some main strategies that should be taken into consideration when GPUs are used for scientific computing.

2.8.1 Exposing parallelism and increasing device occupancy

As mentioned in the previous section, the only way to hide the latencies is allocating several warps simultaneously in an SM, with the aim of keeping the processing units as occupied as possible. It is not a programmer decision to switch between warps at execution time (warp scheduler controls the switching), but allocating as many warps as possible in the same SM increases the chance that a switching occurs.

One way to calculate how many warps are needed for hiding latencies is using Little's Law [86] (borrowed from queuing theory):

$$Warps\ Required = Latency \times Warp\ Throughput \tag{2.3}$$

In this case, *Warp Throughput* is the number of warps that can be processed per cycle in an SM. For instance, to hide latencies of arithmetic operations (~ 10 cycles) in Kepler (192 single precision cores can process 6 warps) the number of warps required in an SM would be 60. However, in many occasions, two instructions can be issued from the same warp (ILP) reducing this number by half (30).

Nevertheless, the number of threads, warps and blocks that can be allocated in a SM is physically limited (see Table 2.3). Furthermore, the warps allocated simultaneously in an SM share resources. Among these resources, the number of registers and the amount of Shared memory can limit the number of warps that can be allocated in an SM.

GPU NVIDIA microarchitecture	MaxThreads per Block	MaxThreads per SM	MaxWarps per SM	MaxBlocks per SM
Tesla	512	768-1,024	**24-32**	**8**
Fermi	1,024	1,536	**48**	**8**
Kepler	1,024	2,048	**64**	**16**
Maxwell	1,024	2,048	**64**	**32**
Pascal	1,024	2,048	**64**	**32**

Table 2.3: Maximum number of threads that can be allocated in a block and maximum number of threads, warps and blocks that can be allocated in the SMs of NVIDIA microarchitectures.

One way to check if any of these resources is limiting the performance of a kernel, is calculating the occupancy of an SM, which we normally want to maximise. The occupancy of an SM is defined as:

$$Occupancy\_SM = \frac{Allocated\_Warps\_SM}{Maximum\_Warps\_SM} \tag{2.4}$$

Ideally, a CUDA program should consist of simple kernels. Simple kernels lead to light threads that use few registers and little Shared memory, and therefore to high occupancy. Nevertheless, if a kernel needs to have more complexity, the number of registers per thread can be manually limited for increasing the SM occupancy.

Another option for increasing the occupancy is to reconfigure the space assigned to L1-cache/Shared-memory in the CUDA microarchitectures, where this is possible (Fermi and Kepler), or to reduce Shared memory usage using alternative memory spaces such as global memory. Note that applying these techniques, the performance of an application could be worse. A high occupancy is convenient but it is not always the most important factor for achieving the best possible performance [85].

In addition to the physical and resource limitations, another factor that can affect the occupancy in an SM is the dimension of the thread blocks defined by the programmer. Threads are mapped to processing units in groups of 32 threads (warps). Thus, the number of threads per block should be a multiple of 32 to avoid wasting processing units resources.

An SM can allocate more warps with a larger number of blocks, or, with more threads per block. If small blocks are used (32, 64 or 96 threads), the maximum occupancy cannot be reached (because there is a limit in the number of blocks per SM, see Table 2.3). On the other hand, if large blocks are used, the global block scheduler will be less flexible for removing and issuing new blocks to the SMs. If workload is not balanced between threads within a block, a small block size can be also preferable, to reduce imbalances. Still, the use of larger blocks (128 or 256) is normally the best option for maximising the occupancy. For a specific application, experimentation with different block sizes is normally necessary to find which is the combination that gives the best performance. Some automated tools that perform such optimisations have been proposed [87].

2.8.2 Avoiding warp divergence

In many cases, threads within a warp may need to perform different operations (e.g. in the presence of conditional branches). In these situations a divergence happens, and the warp needs to execute the instructions of every possible branch sequentially deactivating the threads that do not need to execute the branch.

Warp divergence can cause a significant loss of parallelism, and therefore performance. They should be avoided by design, reorganising the threads and the blocks if needed. It is still possible to have branch conditions inside a kernel without causing many warp divergences. The threads should be organised in such a way that the ones that belong to the same warp always take the same branch, or at least most of the time.

2.8.3 Memory access patterns

Global memory access patterns

Global memory accesses are very costly and can degrade the performance of a CUDA application. If the application is memory intensive, most of the instructions will perform transfers to different GPU memory spaces. Hence, processing units will be idle most of the time. Typically, GPU architectures are not the best candidate for applications with high memory transfer demands. In any case, we should always prioritise the use of on-chip memory (levels 0-2 in Figure 2.9) over the use of global memory.

The accesses to global memory are the most expensive operations in a CUDA kernel given the high latencies. Each time that a warp needs to access the global memory we should try to bring to L2 cache (and L1 if activated) as much useful data as possible. The pattern of how pieces of data are requested by a warp will have a tremendous impact in the performance of an application that needs access to global memory frequently.

In a global memory access, each thread of a warp points to a memory address. The SM determines what lines/segments of memory are needed and then requested. A memory line/segment is the minimum amount of data that can be transferred. If the access is cached in L1, the size of the segment is 128 bytes, otherwise, it is 32 bytes. If the requested segments are found in L1 or L2 caches, then the segments are served with low latencies. If the data is not found in the caches, then an access to global memory is required.

We can define the efficiency of an access to global memory as:

$$Access\ Efficiency\ \% = \frac{bytes\ needed}{Total\ bytes\ requested} \tag{2.5}$$

There are 5 options depending on the lines/segments requested by the threads of a warp (Figure 2.10):

- a) All threads in a warp require the same 4 bytes of data (float or integer). In this case, only a memory segment will be requested from global memory. The access efficiency is $\frac{4}{128} = 3\%$ and $\frac{4}{32} = 12.5\%$ for L1-cached and L1-uncached accesses respectively.
- b) Each thread requires 4 different bytes of data, consecutive in memory, and all of them fall in the same segment of memory space (4 memory segments if L1-uncached access), so only that segment is requested. This is known as *coalesced* memory access, because all the data come together. The segment address is a 128 multiple (32 multiple if L1-uncached access), known as aligned access. The access efficiency is 100% for both, L1-cached and L1-uncached accesses.
- c) Same situation than (b) but the segment(s) address is not either a 32 or 128 multiple (not-aligned access), so two segments are requested (or five if L1-uncached access). The access efficiency is 50% for L1-cached accesses and 80% for L1-uncached accesses.
- d) Each thread requires 4 different bytes of data and they are not consecutive in memory, but all of them fall in the same segment(s) (coalesced access) and the segment(s) address is a 128(32) multiple (aligned access). The access efficiency is 100% for L1-cached and L1-uncached accesses.
- e) Each thread requires 4 different bytes of data and they fall in different segments (uncoalesced access). The access efficiency in the worst case is 3% and 12.5% for L1-cached and L1-uncached accesses respectively.

The programmer should organise the data structures of an application in such a way that accesses to global memory require as few segment as possible, trying to avoid the uncoalesced and non-aligned memory accesses.

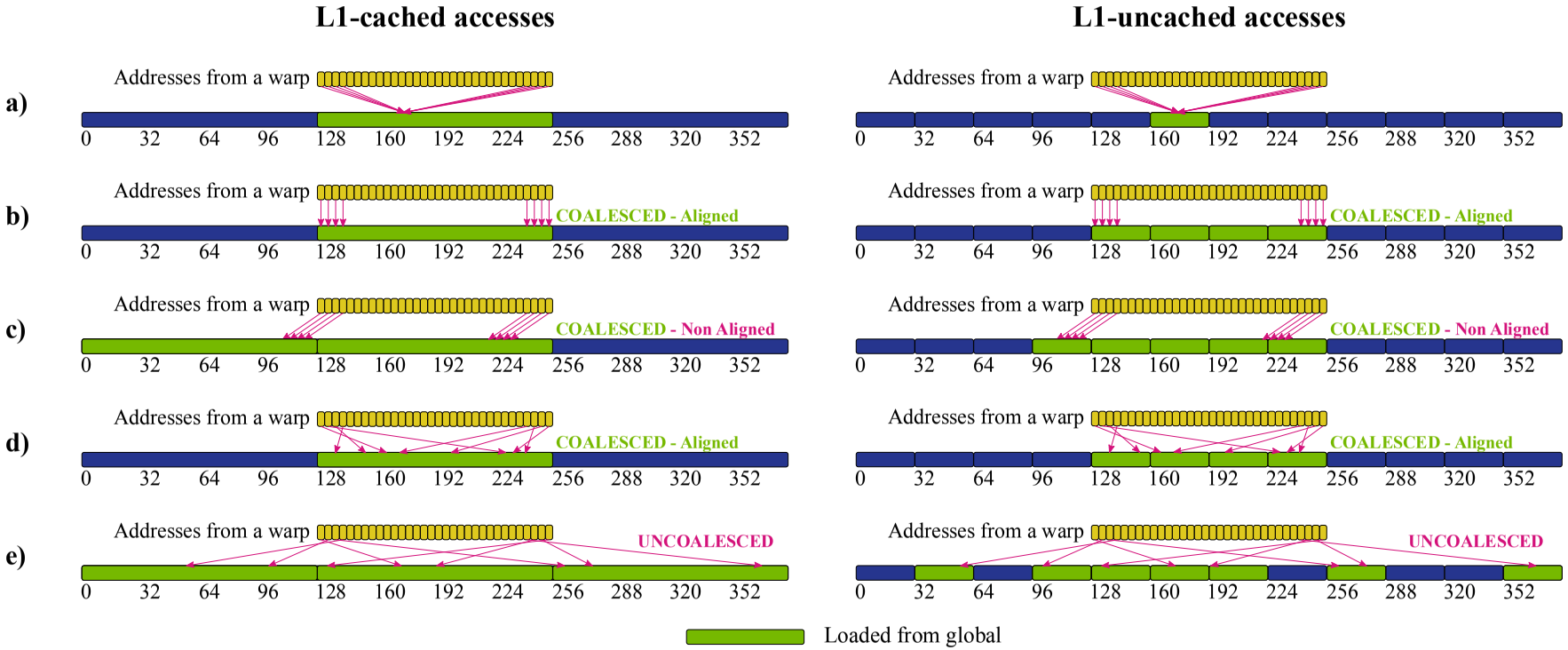

Figure 2.10: L1-cached and L1-uncached global memory access patterns.

Shared memory access patterns

When a CUDA kernel accesses Shared memory frequently, attention should be taken to the patterns in these accesses. Shared memory is divided into 32 parts or banks that can serve data simultaneously. When a warp accesses Shared memory, there are three possible access patterns (see Figure 2.11):

- a) Each thread in a warp requests data allocated in different banks. Data can be served in one transaction. Access efficiency is 100%.
- b) All threads in a warp request the same data. Only one transaction is needed although the access efficiency is low ($\frac{1}{32} = 3\%$).
- c) Several threads request different data allocated in the same bank. Accesses to the same bank are serialised. In the worst case the access efficiency will be $\frac{2}{32} = 6\%$.

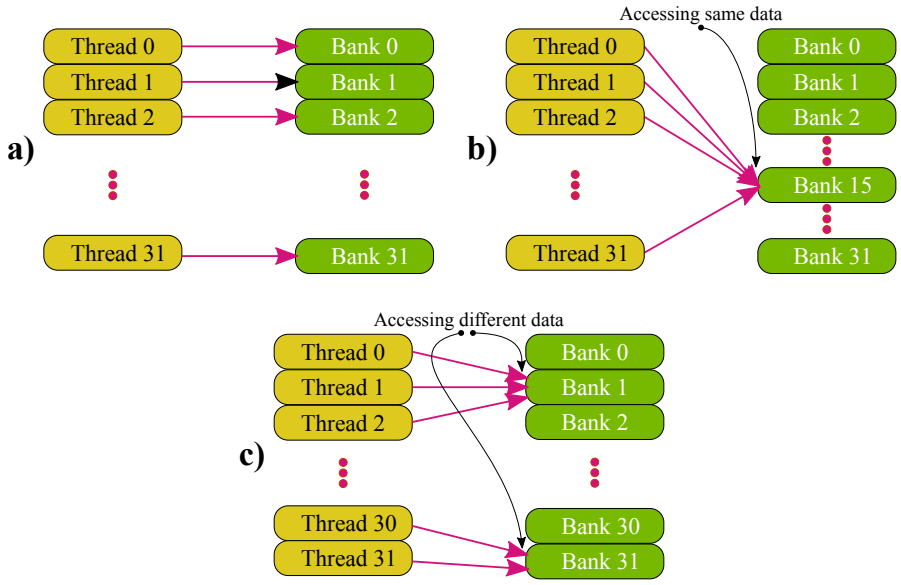

Figure 2.11: Shared memory access patterns.

Data in Shared memory should be organised so that different threads of a warp access data from different banks, trying to avoid the accesses that are serialised, although this overhead is reduced in NVIDIA Maxwell and Pascal microarchitectures [9].

3

Diffusion MRI and Tractography

Contents

3.1 Diffusion MR Imaging 40
3.1.1 Pulsed gradient spin echo sequence 42
3.1.2 Estimation of the apparent diffusion coefficient 43
3.2 Diffusion Tensor Imaging 44
3.2.1 Fitting the diffusion tensor 45
3.2.2 Microstructure parameters derived from the diffusion tensor 47
3.2.3 Limitations of the diffusion tensor 48
3.3 Multi-compartment diffusion models 49
3.4 White matter tractography 53
3.5 Brain Connectomes 58
3.6 Computational requirements 60

3.1 Diffusion MR Imaging

Almost two centuries ago, in 1827, while studying pollen grains with a microscope, the Scottish botanist Robert Brown observed that particles suspended in a fluid were always active, even in inorganic matter [88]. Further studies showed that this random transport process, or *diffusion* process, was caused by thermally-driven motion and collisions of atoms and molecules in the fluid, and it was named Brownian motion. Albert Einstein characterised this phenomenon in a barrier-free and homogeneous medium as a Gaussian process [89], where the average displacement is zero and the variance (or mean-squared displacement of the molecules $\langle z^2 \rangle$) in 3D is given by:

$$\langle z^2 \rangle = 6Dt \tag{3.1}$$

where D is called the diffusion coefficient (in mm^2/s) and t is the time in seconds observing the phenomenon (see Figure 3.1). The diffusion coefficient D is a constant that depends on the size of the molecules, temperature, pressure and microstructural features of the environment.

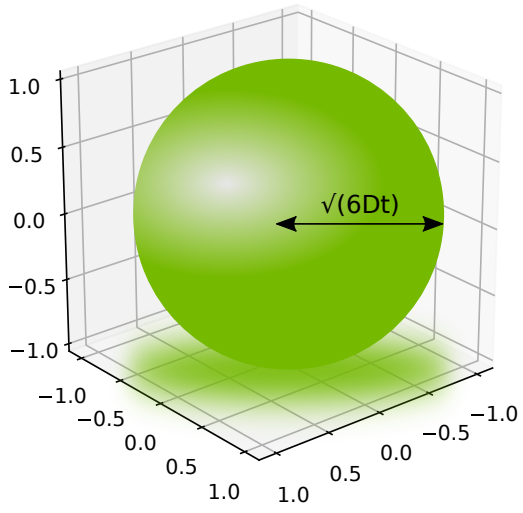

Figure 3.1: Diffusion as a Gaussian process in three dimensions. The probability of displacement is the same in any direction.

In Magnetic Resonance Imaging (MRI), the process of water diffusion within tissue can be used to introduce contrast to images, with the aim of reflecting underlying microstructural features. In such diffusion MRI (dMRI) experiments, diffusion is no longer free and does not happen within a homogeneous medium. The diffusion coefficient typically appears to be lower than the value D expected from Einstein equation. This happens because the process occurs within tissue with lots of microstructural boundaries that constrain diffusion. Furthermore, the extent to which diffusion displacements reach boundaries, and are restricted, depends on the experiment time t. For these reasons, the observed diffusion is called apparent diffusion and D is typically called the *apparent diffusion coefficient* (ADC).

Interestingly, by measuring the ADC within brain tissue, we can gain information about the tissue structure. The pattern of diffusion displacements vary throughout the different brain tissues, including white matter, which mostly comprises neuronal axons, grey matter, which contains mainly cell bodies, and the cerebrospinal fluid (CSF). Particularly, diffusion in white matter is anisotropic (see Figure 3.2); water preferably diffuses along rather than across the axons [90]. In CSF regions (and, to a slightly lesser extent in grey matter) diffusion is isotropic, i.e. there is no preference for diffusion along any particular orientation, and it is relatively unrestricted. In grey matter, diffusion is restricted (due to the presence of cell bodies and dendritic trees), but again mostly isotropic, as in general there is no coherent structure towards any particular direction. This ability to indirectly differentiate between tissue types and characterise tissue properties is key for all dMRI applications.

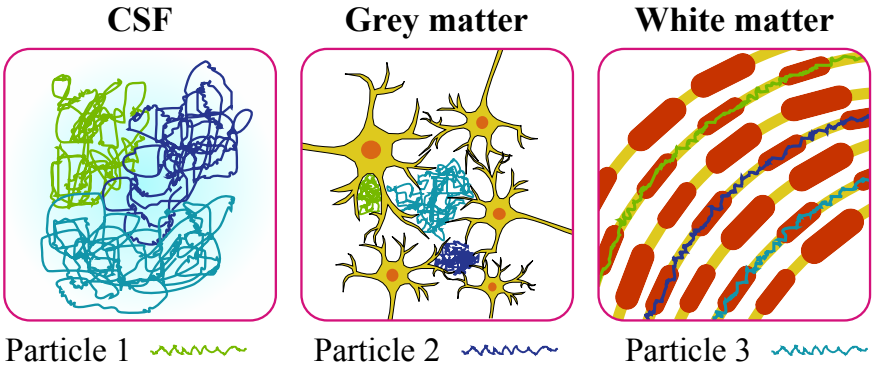

Figure 3.2: Representation of the diffusion process for three particles within different brain tissues.

3.1.1 Pulsed gradient spin echo sequence

Typically, diffusion contrast is introduced into an MRI scan using a modified spin echo sequence [91]. In a spin echo sequence, a 90° pulse is applied for rotating the longitudinal magnetisation to the transverse plane. Due to molecular interactions (spin-spin relaxation) and local magnetic field non-uniformities, spins precess at slightly different frequencies and go out of phase, leading to a reduced total magnetisation. This dephasing can be reversed by applying a 180° pulse, which leads back to building up a signal again (echo) (see Figure 3.3).

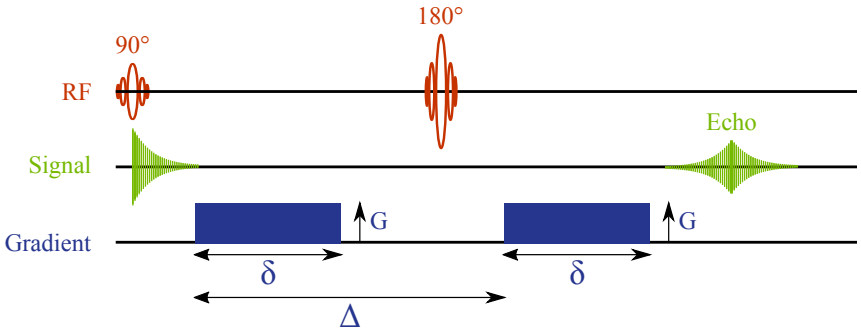

Figure 3.3: Schema of the pulsed gradient spin echo sequence, using gradients with magnitude G, duration δ, and Δ time between the application of the two gradients.

Diffusion contrast is introduced using a pulsed gradient spin echo sequence (Figure 3.3). Diffusion-sensitising magnetic field gradients are introduced to the spin echo sequence, and these introduce sensitivity to random movement of water

molecules along a particular orientation [92]. Before the 180° pulse, a first gradient is applied, making the protons to precess at different rates depending on their locations, i.e., making them to have different phase shift. A second gradient with identical amplitude and duration as the first gradient is applied after the 180° pulse. This second gradient will have an opposite polarity due to the 180° pulse. If protons have not moved from their location, they will acquire an exactly opposite phase of what they acquired from the first gradient. Therefore, the effect of the second gradient cancels completely the effect of the first gradient, and these protons will contribute to the total signal as if no gradients had been applied. However, for protons that have moved along the sensitised orientation due to diffusion, the phases acquired due to the two gradients will not be identical and will not cancel out. This will cause attenuation in the total signal contributed by these protons compared with stationary protons, or compared with the signal (spin echo) that would have been recorded without any diffusion gradient. The greater the diffusion displacements of the molecules, the higher the signal attenuation.

There are three features that affect the amount of diffusion weighting (i.e. the amount of MRI contrast introduced due to diffusion):

- Strength of the gradients: G
- Duration of the gradients: δ
- Time between the application of the two gradients: Δ

3.1.2 Estimation of the apparent diffusion coefficient

Under the assumption of Gaussian diffusion, by acquiring a signal in the absence of any diffusion-sensitising gradient, S_0, and a signal after applying a diffusion-sensitising gradient, S, it is possible to establish a relationship between these two acquired signals and estimate the ADC in each voxel of an MRI dataset [93]:

$$\frac{S}{S_0} = \exp^{(-bADC)} \tag{3.2}$$

where b, or *b-value*, represents the set of features that specify the amount of diffusion weighting:

$$b = \gamma^2 G^2 \delta^2 \left(\Delta - \frac{\delta}{3} \right) \tag{3.3}$$

and γ is the constant gyromagnetic ratio (for hydrogen protons is 42.576 $MHz/Tesla$).

However, the ADC is only estimated along the direction in which the diffusion-sensitising gradient was applied, ignoring the diffusion along the rest of directions. In scenarios where the diffusivity is anisotropic (there is a preferred diffusion orientation), such as in the white matter, the estimated diffusivity highly depends on the gradient direction, and different values are obtained when applying different gradient directions.

3.2 Diffusion Tensor Imaging

A simple and powerful framework for estimating the diffusivity in 3D is diffusion tensor imaging (DTI) [94, 95]. This framework uses a 3×3 symmetric matrix (or tensor) for characterising the diffusion displacement along different directions of a 3D Cartesian system:

$$\mathbf{D} = \begin{bmatrix} D_{xx} & D_{xy} & D_{xz} \\ D_{xy} & D_{yy} & D_{yz} \\ D_{xz} & D_{yz} & D_{zz} \end{bmatrix} \tag{3.4}$$

The diagonal elements represent the diffusivity along three orthogonal axes (the scanner's coordinate system), and the off-diagonal elements represent the correlation between diffusion in pairs of these axes. The diffusion tensor is typically diagonalised for reflecting better the local tissue coordinate system rather than the scanner's coordinates, and a 3D ellipsoid can be used for having a pictorial and more intuitive representation. The axes directions of the ellipsoid are given by the

eigenvectors e_1, e_2, e_3 of the tensor and the magnitude of each axis is a function of the corresponding eigenvalues $\lambda_1, \lambda_2, \lambda_3$ (see Figure 3.4).

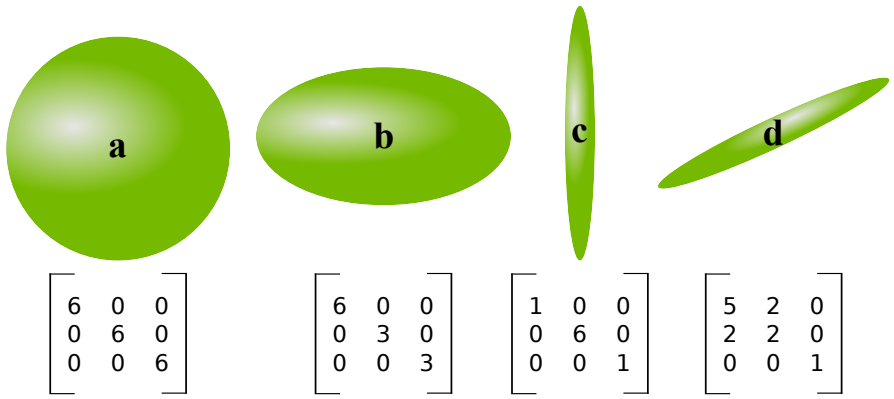

Figure 3.4: Example of diffusion tensors with their corresponding ellipsoids. The tensor in (a) is completely isotropic, and the rest of tensors are anisotropic, with higher diffusion along one of the axis. The tensor (c) and (d) are more anisotropic than (b), and the tensor (d) is not aligned with the scanner's coordinate system.

The diffusion MR signal for the diffusion tensor model is given by:

$$\begin{aligned} \frac{S}{S_0} &= \exp^{-b\mathbf{g}^T\mathbf{D}\mathbf{g}} \\ &= \exp^{\left(-bg_x^2D_{xx}-bg_y^2D_{yy}-bg_z^2D_{zz}-2bg_xg_yD_{xy}-2bg_xg_zD_{xz}-2bg_yg_zD_{yz}\right)} \end{aligned} \tag{3.5}$$

where $\mathbf{g}$ is a unit vector representing the direction of the applied diffusion-sensitising gradients.

3.2.1 Fitting the diffusion tensor

In Equation 3.5 there are six unknown elements $(D_{xx}, D_{yy}, D_{zz}, D_{xy}, D_{xz}, D_{yz})$. Thus, applying gradients in non-collinear directions and acquiring at least $M = 6$ different diffusion-weighted measurements, the system can be formulated as

a linear system:

$$\begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \\ A_5 \\ A_6 \end{bmatrix} = \begin{bmatrix} -bg_{1x}^2 & -bg_{1y}^2 & -bg_{1z}^2 & -2bg_{1x}g_{1y} & -2bg_{1x}g_{1z} & -2bg_{1y}g_{1z} \\ -bg_{2x}^2 & -bg_{2y}^2 & -bg_{2z}^2 & -2bg_{2x}g_{2y} & -2bg_{2x}g_{2z} & -2bg_{2y}g_{2z} \\ -bg_{3x}^2 & -bg_{3y}^2 & -bg_{3z}^2 & -2bg_{3x}g_{3y} & -2bg_{3x}g_{3z} & -2bg_{3y}g_{3z} \\ -bg_{4x}^2 & -bg_{4y}^2 & -bg_{4z}^2 & -2bg_{4x}g_{4y} & -2bg_{4x}g_{4z} & -2bg_{4y}g_{4z} \\ -bg_{5x}^2 & -bg_{5y}^2 & -bg_{5z}^2 & -2bg_{5x}g_{5y} & -2bg_{5x}g_{5z} & -2bg_{5y}g_{5z} \\ -bg_{6x}^2 & -bg_{6y}^2 & -bg_{6z}^2 & -2bg_{6x}g_{6y} & -2bg_{6x}g_{6z} & -2bg_{6y}g_{6z} \end{bmatrix} \begin{bmatrix} D_{xx} \\ D_{yy} \\ D_{zz} \\ D_{xy} \\ D_{xz} \\ D_{yz} \end{bmatrix}$$

$$\Rightarrow \mathbf{A} = \mathbf{WD} \tag{3.6}$$

where:

$$A_m = \log\left(\frac{S_m}{S_0}\right) \tag{3.7}$$

and S_m is the m^{th} diffusion-weighted measurement.

The diffusion tensor can easily be solved as:

$$\mathbf{D} = \mathbf{W}^{-1}\mathbf{A} \tag{3.8}$$

However, MRI introduces noise in the acquired images. Typically, $M > 6$ diffusion-weighted images are acquired for reducing the effect of noise in the estimated tensor parameters, and also for making them more rotational invariant. Thus, the $\mathbf{W}$ matrix is not square any more and the diffusion tensor is estimated using the pseudo-inverse of $\mathbf{W}$, as:

$$\mathbf{D} = \left(\mathbf{W}^T\mathbf{W}\right)^{-1}\mathbf{W}^T\mathbf{A} \tag{3.9}$$

This approach is called *ordinary least squares* (OLS) and it assumes that the variance of noise is the same for all the diffusion-weighted measurements, which is acceptable if the system were solved using directly the measurements values S_m. But the logarithm transformation applied in Equation 3.7 (used for establishing a linear relationship in the system) makes the elements of $\mathbf{A}$ with a low value to have a much higher noise variance than the elements with a high value. Thus, a *weighted*

least squares (WLS) approach can be used for estimating the tensor parameters [94]:

$$\mathbf{D} = \left(\mathbf{W}^T\boldsymbol{\Sigma}^{-1}\mathbf{W}\right)^{-1}\mathbf{W}^T\boldsymbol{\Sigma}^{-1}\mathbf{A} \tag{3.10}$$

where $\boldsymbol{\Sigma}$ represents the covariance matrix of the diffusion measurements. An alternative is to fit the tensor model directly, without logarithm transformation, to the diffusion measurements using non-linear optimisation methods, such as Levenberg–Marquardt [96–98].

3.2.2 Microstructure parameters derived from the diffusion tensor

The principal fibre orientation is normally estimated using the tensor's principal eigenvector and represented using a color-coded schema [99] (see Figure 3.5).

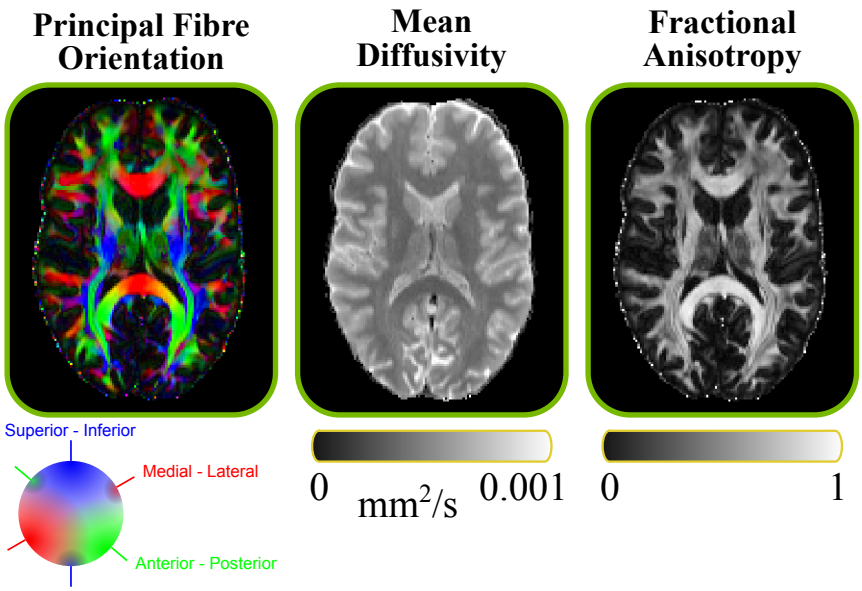

Figure 3.5: Parameters derived from the diffusion tensor, including the principal fibre orientation (colour coded in the figure), the mean diffusivity and the fractional anisotropy.

Additionally, rotationally invariant scalar indices can be derived directly from the diffusion tensor, being the most commonly used in clinical applications the mean diffusivity and the fractional anisotropy [100–102] (see Figure 3.5). The

mean diffusivity (MD) is defined as:

$$MD = \frac{\lambda_1 + \lambda_2 + \lambda_3}{3} \tag{3.11}$$

and it has been found to be useful for detecting brain ischemic lesions, where water diffusion decreases considerably [103].

Another extensively used index derived from the diffusion tensor is the fractional anisotropy (FA):

$$FA = \frac{\sqrt{3\left((\lambda_1 - MD)^2 + (\lambda_2 - MD)^2 + (\lambda_3 - MD)^2\right)}}{\sqrt{2\left(\lambda_1^2 + \lambda_2^2 + \lambda_3^2\right)}} \tag{3.12}$$

It represents the fraction of the tensor that is assigned to anisotropic diffusion, and it is a scaled version of the tensor eigenvalues variance. If $FA = 0$ the diffusion is perfectly isotropic ($\lambda_1 = \lambda_2 = \lambda_3$), and if $FA = 1$ the diffusion is infinitely anisotropic ($\lambda_1 \neq 0, \lambda_2 = \lambda_3 = 0$). Changes in FA has been shown to be associated to several neurodegenerative diseases, including Alzheimer [104] and multiple sclerosis [105].

3.2.3 Limitations of the diffusion tensor

The main limitation of the diffusion tensor is its incapability for characterising complex geometries and fibre configurations. The model assumes a single tissue compartment within a voxel. However this assumption can be problematic, for instance in white matter. The radius of an axon is in the range of a few μm whereas the voxel size is in the range of 1-3 mm. Thus, thousands of axons from possibly different fibre populations are expected to be present within a voxel, and the tensor takes a shape that represents the average of all the populations, which may not be similar to any of the present populations. Consequently, diffusion tensor is not able to represent complex fibre configurations like crossing, fanning, bending or

kissing (see Figure 3.6) [106]. In such cases, not only the principal fibre orientation is wrong, but also DTI quantitative metrics, like the FA, are biased.

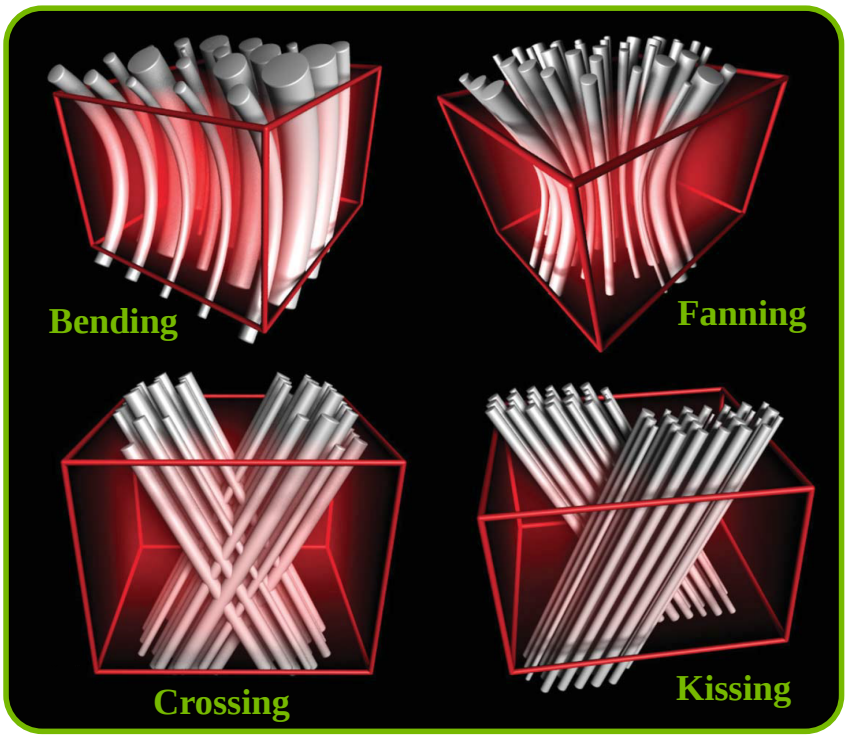

Figure 3.6: Examples of complex fibre configuration. Reproduced with permission from [106] and modified.

Some studies have reported the presence of complex fibre configurations in about 90% of the white matter voxels [107], and many approaches have been proposed for being able to characterise them, including model-based and model-free approaches. We review in the next sections the most relevant ones used in this thesis.

3.3 Multi-compartment diffusion models

A number of models that assume multiple compartments within a voxel, but also a number of model-free approaches, have been proposed as an alternative to the single compartment diffusion tensor model [106, 108–112]. Model-based approaches have the disadvantage over model-free methods of being based on assumptions, and their results may be sensitive to whether these assumptions are met. However,

they offer the advantage of estimating parameters that can be directly linked to biophysical quantities, and therefore directly interpretable. In this thesis, we will be concerned with model-based multi-compartment methods.

Multi-compartment models aim to represent more accurately tissue microstructure or to specifically represent complex fibre geometries in the white matter. A simple approach for resolving crossing patterns was proposed with the multi-tensor model [106], a direct extension of DTI. The model assumes $L \geqslant 1$ fibre populations, each one modelled with a different tensor. For each of the $\mathbf{g}_m$ gradient directions applied Equation 3.5 generalises as:

$$\frac{S_m}{S_0} = \sum_{i=1}^{L} f_i \exp^{(-b_m \mathbf{g}_m^T \mathbf{D}_i \mathbf{g}_m)} \tag{3.13}$$

where $f_i \in [0, 1]$ represents the volume fraction of the i^{th} fibre population and $\sum_{i=1}^{L} f_i = 1$.

However, in the general case, the fitting process of the multi-tensor model is problematic. Particularly, the model is not adequate for data acquired with a single *b-value*, as high correlations between model parameters makes the fitting process unstable, and the estimation of multiple unconstrained tensors $\mathbf{D}_i$ impossible [113]. Certain constrains, either in the acquisition or in the model, can be imposed to overcome these limitations. For instance, in [114] the two tensors of the model are initialised as cylindrically symmetric, i.e., $\lambda_2 = \lambda_3$.

A different geometrically-constrained multi-tensor model is the ball & sticks model [115, 116]. This model assumes a fully isotropic compartment, the ball, and one or more perfectly anisotropic compartments, the sticks. The orientations of these sticks provide the principal fibre orientations in a voxel.

When it comes to represent tissue microstructure, multi-compartment models attempt to represent different biophysical compartments within tissue, such as intra-axonal and extra-axonal space [117, 118]. Intra-axonal compartments characterise

the diffusion inside the white matter axons as an anisotropic and restricted process. Some examples of such compartments are shown in Figure 3.7, including a cylinder, a stick (a zero-radius cylinder) and a distribution of cylinders/sticks.

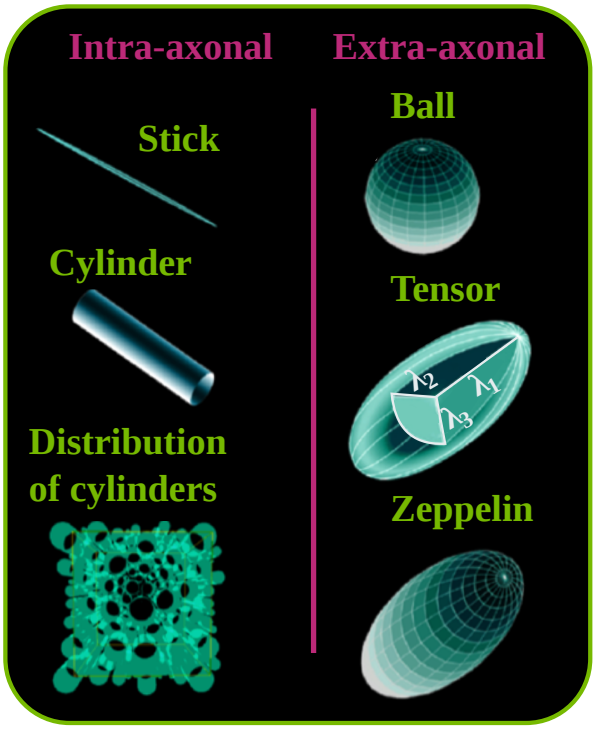

Figure 3.7: Different compartments employed in diffusion MRI models. They can be divided into two categories: intra-axonal and extra-axonal compartments. Reproduced from [118] and modified (open access).

Extra-axonal compartments characterise the diffusion outside axons as a hindered process, i.e., with a reduced diffusion coefficient compared to barrier-free diffusion. In the ball compartment the diffusion is assumed perfectly isotropic. On the other hand, in the tensor and zeppelin compartments (cylindrically-symmetric tensor) diffusion can be anisotropic.

Many models exist, with different assumptions, that follow these generic ideas. One of the first models in this family is the Composite Hindered and Restricted ModEl of Diffusion (CHARMED) [119, 120]. It includes two types of compartments, intra-axonal and extra-axonal, characterised by distributions of cylinders and tensors

respectively. The diffusion signal S is modelled as:

$$S = \sum_{i=1}^{L^{'}} f_i^{intra} S_i^{intra} + \sum_{i=1}^{L^{''}} f_i^{extra} S_i^{extra} \tag{3.14}$$

with $L^{'}$ intra-axonal compartments and $L^{''}$ extra-axonal compartments, S_i^{intra} represents the signal from the i^{th} distribution of cylinders, and S_i^{extra} the signal from the i^{th} tensor.

A number of more recent approaches have employed the cylinders compartment for inferring the axon diameter [121–123], while other models have extended these compartments for estimating more complex fibre configurations, such as fibre fanning [124, 125].

A common feature of all these multi-compartment models is that they are non-linear. Also, given the inherent noise of the dMRI data, evaluating the uncertainty of the estimates can be of interest, with approaches such as Bayesian inference or bootstrapping commonly being used. Finally, these models typically require much more data than a standard DTI scan. High Angular Resolution Diffusion-weighted Imaging (HARDI) [126], single-shell or multi-shell, is commonly used, where tens to hundreds of diffusion measurements, with different gradient directions and/or *b-values*, are acquired. For all these reasons, fitting multi-compartment diffusion MRI models can be very demanding in computational resources.

3.4 White matter tractography

Extracting fibre orientations at the voxel level, using DTI/multi-compartment models or model-free approaches, opens the way for studying long-range brain connections using tractography approaches. The human brain white matter is mainly composed of axons that are grouped into large bundles or tracts. Diffusion tractography is the only method that allows the three-dimensional reconstruction of human brain tracts non-invasively and in vivo. The method assumes that water diffusion occurs mainly along the tracts, which are aligned with the axons. Tractography methods find paths of least hindrance of water diffusion in the white matter, and these are assumed to be estimates of the underlying tracts connecting remote brain regions. Although tractography is an indirect technique, its potential is unique, as it allows a "virtual dissection" of white matter in vivo [24].

Tractography algorithms integrate voxel-wise fibre orientations for reconstructing the white matter tracts. There is a plethora of tractography algorithms (see [127] for a recent review), but the most common are *streamline-based* methods. The simplest streamline method is *deterministic tractography*. Starting from a seed point l_0, streamline algorithms generate a path that is propagated within a volume "joining" fibre orientations of adjacent voxels (see Figure 3.8a) [128]. This path, known as *streamline*, can be mathematically represented as a 3D space curve $\mathbf{u}(l)$, which is a function of the length l of the streamline from the starting point. For propagating a streamline, the fibre orientation ϵ used for calculating its next location is the tangent $t(l)$ of the curve at that length [23] (see Figure 3.8b):

$$\frac{d\mathbf{u}(l)}{dl} = t(l) = \epsilon(\mathbf{u}(l)) \tag{3.15}$$

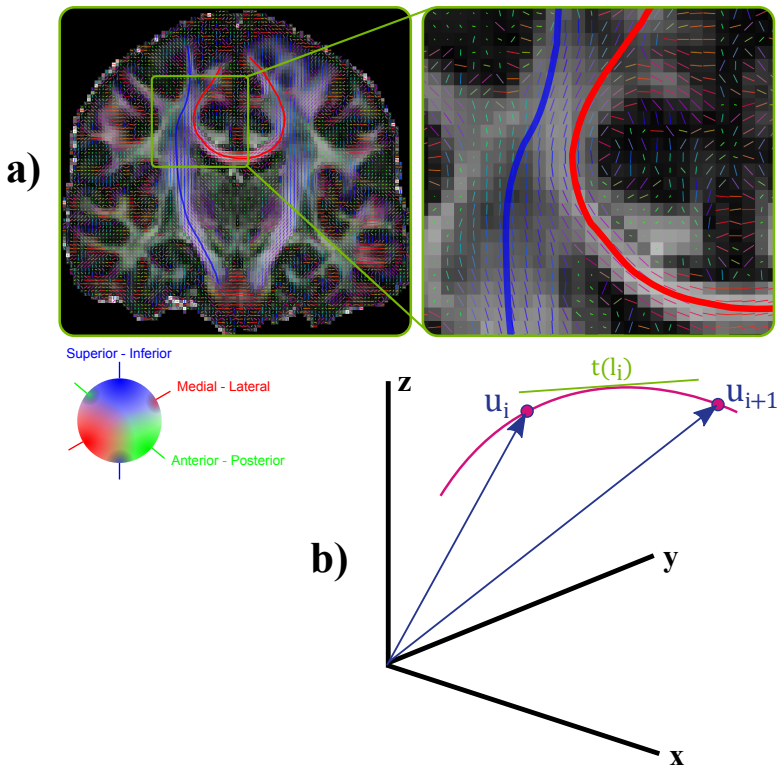

Figure 3.8: Streamline tractography. Given a map with fibre orientations in each voxel, streamline tractography reconstruct the tracts joining them as depicted in (a). In this case, the red streamline reconstructs the corpus callosum, and the blue streamline reconstructs the corticospinal tract. (b) The streamline is a 3D curve $\mathbf{u}(l)$ function of its length l, and the tangent of the streamline $t(l)$ is the fibre orientation employed at each step i.

A straightforward way to solve numerically the simple differential Equation 3.15 is Euler's method [23]. Given a step size h, the track can be propagated as:

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \epsilon(\mathbf{u}_i)h \tag{3.16}$$

Euler's method is a first-order method (considers only the current position to decide on propagation) that can lead to poor accuracy, particularly when the curve of the axons is sharp and the step size is not small enough to capture this shape. An alternative is to use a higher-order method, such as Runge-Kutta method [23].

The orientations provided by the diffusion models are discretised on a voxel-wise grid. Thus, for propagating a streamline at a specific location of the diffusion space an interpolation method is required. Different methods are used depending on the

algorithm, including nearest-neighbour [128], tri-linear interpolation [23, 129] and a probabilistic interpolation method where the orientation of a neighbour can be chosen according to a location-based criteria function [115].

An important consideration when performing tractography is the selection of seed points for starting to propagate the streamlines. A common approach is to define manually a region of interest (ROI), which location and shape depends on the study [130]. The correct definition of these seeds is critical, as it will have a direct influence on the results. An alternative is to perform whole brain connectivity starting the streamlines from a large range of locations [4], but this is more computationally expensive.

Tractography algorithms, particularly deterministic methods, use certain heuristics for deciding when to terminate a streamline. The most common ones are a brain boundary mask, curvature thresholds and anisotropy thresholds [23]. It is obvious that if a streamline crosses the brain boundary, it should stop. Additionally, under the assumption that fibre bundles change smoothly in space, if the curvature of the track is very high between two consecutive propagation steps, this is deemed unrealistic, and the algorithm terminates the streamline. Finally, if a streamline visits a voxel with a very low (fractional) anisotropy, the interpretation is that there is a very high uncertainty related to the orientation of the fibre, or it is a voxel in the grey matter or CSF. Thus, streamlines stop in these voxels.

It is this notion of uncertainty of fibre orientations that have given rise to probabilistic streamline tractography [115, 116]. Due to inherent noise in MRI acquisitions, partial volume, and modelling assumptions and errors, estimated fibre orientations are never identical between repeated acquisitions of the same subject [131] (see Figure 3.9). This orientation uncertainty is accumulated during tractography and it is reflected in the estimated streamlines. Probabilistic tractography methods consider this uncertainty and formally characterise it in the estimated tracts.

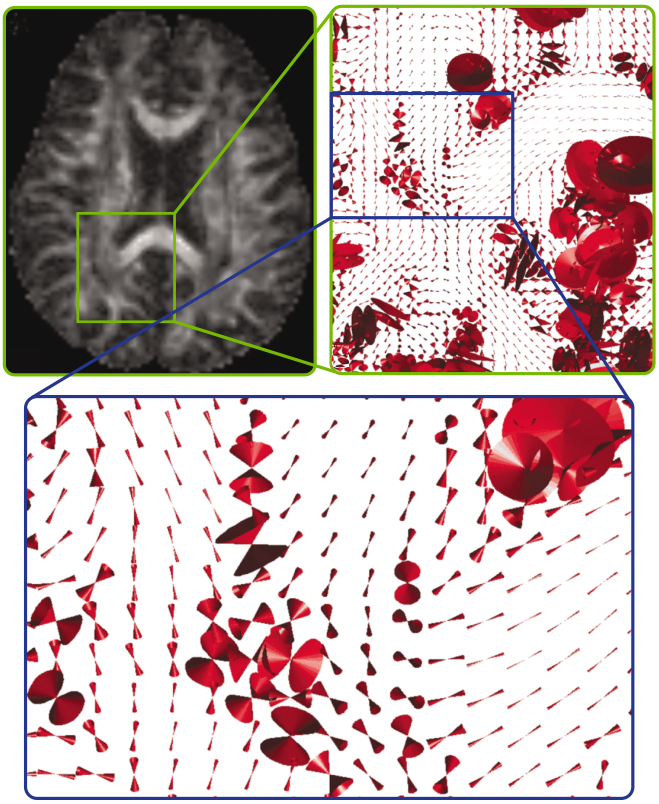

Figure 3.9: Cones of uncertainty of the fibre orientation. The figures shows a map of fractional anisotropy (top-left figure) and the 95% uncertainty cones (calculated using bootstrapping) in a region near the corpus callosum (top-right and bottom figures). In voxels with crossing fibre (green rectangle), there is a higher uncertainty. Reproduced with permission from [131] and modified.

Specifically, in probabilistic tractography a distribution of fibre orientations is assumed and used in each voxel. This distribution characterises the uncertainty of the orientation. The most common methods for obtaining these distributions are *bootstrapping* and Bayesian techniques. In bootstrapping, multiple repetitions of the diffusion dataset can be acquired for building different data combinations and estimating fibre orientations [131]. An alternative is residual bootstrapping, where model residuals are reshuffled and added to the diffusion measurements for creating different data combinations [132]. Bayesian techniques use a model, the data and potentially prior information to estimate a posterior distribution of the model parameters. From this distribution, different samples of the parameters are obtained for getting a distribution of fibre orientations [115, 116].

Streamline propagation in probabilistic tractography is different from deterministic approaches. In a particular location, a streamline progresses along a direction randomly sampled from the distribution of orientations. This stochastic component allows the propagation of many different streamlines from the same seed. Using the classic definition of probability for an experiment (number of successes over number of experiments), we can define the probability of a path starting from a seed point A to any other point or region B of the brain (see Figure 3.10):

$$P(A \rightarrow B) = \frac{\text{Number of streamlines that reached B}}{\text{Total number of streamlines propagated from A}} \tag{3.17}$$

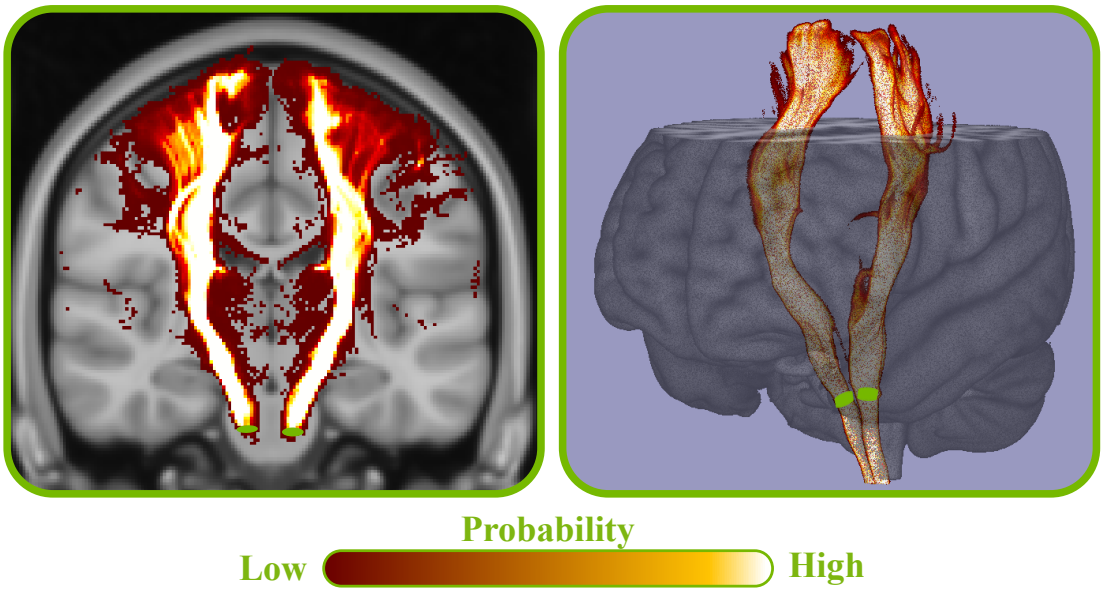

Figure 3.10: Probabilistic tractography. The green areas represent the seed points from where streamlines are propagated for reconstructing the Corticospinal tract. The figure depicts the probability of a diffusion path between these seed points and any other voxels of the brain, without applying any threshold (left), and a 3D view after applying a 0.5% threshold (right).

The method can represent the uncertainty introduced by noise and the model inaccuracies. It generates a path distribution from a seed, rather than a deterministic estimate, and it needs to propagate a large number of streamlines from the seed points to achieve convergence. For this reason, probabilistic tractography

and uncertainty estimation have a much higher computational demand than deterministic estimation.

The results from probabilistic tractography should be interpreted carefully. The probability $P(A \rightarrow B)$ does not represent the anatomical connection strength between A and B, but the confidence associated with the existence of a path between these two regions, and along which water molecules would preferentially diffuse [133]. However, the relative contrast that can be obtained in these confidence metrics has been shown to be valuable and anatomically relevant in various contexts [134]. For instance, path probabilities have been found to carry enough information to identify anatomical boundaries of functionally-distinct regions, both in the cortex [135] and in the sub-cortex [136].

3.5 Brain Connectomes

White matter tractography methods are extensively used for localising white matter tracts [137, 138] and studying the human brain connectivity. An alternative approach for studying whole-brain connectivity using tractography is the estimation of a *connectome*, a comprehensive map of the likelihood of connections between many brain regions [139, 140]. This focuses more on the pattern of connections from a region, rather than their route through white matter.

A connectome comprises a set of nodes that represent different areas of the brain, and edges between these nodes that represent their connections. Thus, for building a connectome the first step is to define the nodes, which typically are cortical or subcortical grey matter regions. The definition of these nodes can be based on architectonic atlases or parcellations from functional studies (see Figure 3.11a) [141]. The connectivity weight between each pair of nodes can be determined by performing dMRI tractography. Seed points are set at each node region, and the number of streamlines that reach the other nodes are used to determine the weights and estimate connectivity, typically summarised in a matrix format.

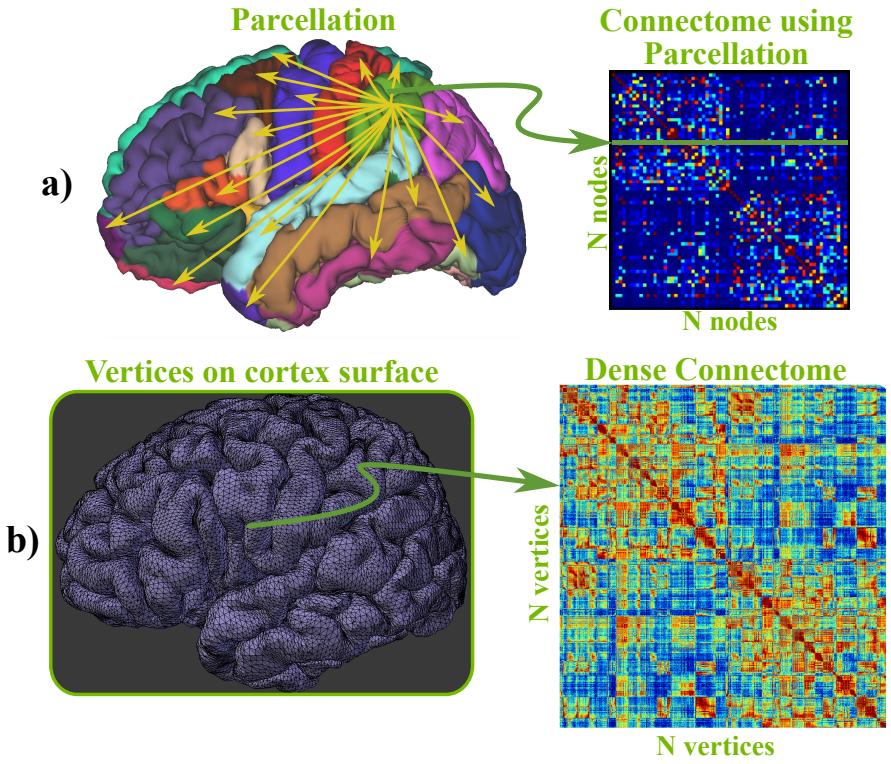

Figure 3.11: Generation of a human connectome. (a) A connectome comprises a set of nodes, which can be obtained by parcellation (using Desikan-Killiany atlas in this case [142]), and edges between them that are typically represented in a Matrix format. The weight of these edges can be obtained performing streamline tractography using each node area for defining seed points and calculating how many streamlines reach the other nodes. (b) The number of nodes in a connectome can vary. At the extreme, a dense connectome has as many nodes as points on the cortical surface, which are representative of the spatial resolution of the MRI data (for instance, in the HCP $N \sim 90{,}000$ nodes, representative of an average spacing between neighbouring nodes of ~ 2 mm).

When probabilistic tractography is used for building a connectome, a high number of streamlines (∼10,000) are propagated from each node in order to make the method converge. Moreover, a high number of nodes (>10,000) may be required in connectivity studies with high spatial resolution, such as in the Human Connectome Project (HCP) [2–4] (see Figure 3.11b). In general, computing connectomes is highly computationally expensive.

3.6 Computational requirements

Despite the great potential of all the presented approaches, the non-linear estimation involved in many levels leads to heavy computational requirements and long computational times (many hours or days, depending on the dataset size). Some examples are shown in the table below (Table 3.1).

Application (Single subject)	**Single CPU core time**	**Memory Required**
Ball & sticks (MCMC) - UK Biobank	4 days	2.5 GB
Ball & sticks (MCMC) - HCP	11 days	8 GB
NODDI-Bingham Multi-compartment model - UK Biobank	7.5 days	2.5 GB
Brain Connectome(dense) - HCP	15 days	35 GB

Table 3.1: Execution times and memory requirements of some dMRI applications processing datasets from the UK Biobank project [32, 33] and the Human Connectome Project.

Advances in high performance computing (HPC) alleviate the situation. Clusters of computers, with tens of cores, can be used to process the above tasks in parallel. However, even using a large number of cores, these applications continue being considerably time consuming, restricting the potential of the techniques and producing bottlenecks in processing pipelines [143]. This is particularly true nowadays, as the trend is for large high-resolution datasets (due to recent advances in scan time accelerations, high-field acquisitions, high gradients and multiple shells) and hundreds to thousands of subjects for Big Data imaging projects. In this thesis we propose the use of graphics processing units (GPUs), which although are not inherently suited for parallelising all types of problems, have thousands of cores to massively accelerate computations and to potentially change the perception of what is computationally feasible in the analysis of diffusion MRI.

4

Estimation of local diffusion MRI models on GPUs

Contents

4.1 Introduction 65
4.2 Modelling diffusion signal: Fitting the ball and sticks model 67
4.3 Modelling diffusion signal on GPUs: Design and Implementation 70
4.3.1 Parallelising the MCMC algorithm 72
4.3.2 Parallelising the Levenberg-Marquardt algorithm 81
4.3.3 Performance limitations 86
4.3.4 Using Multiple GPUs 88
4.4 Results: Performance gains 88
4.4.1 MCMC routine evaluation 92
4.4.2 Levenberg routine evaluation 94
4.4.3 Overall evaluation 96
4.5 Validation: Comparison to CPU implementation . . . 99
4.6 Discussion 103
4.6.1 Extension of the designs to other models 105
4.6.2 Clinical and Big Data applications 107

Overview

The analysis of diffusion MRI (dMRI) data allows inferring brain structural information and estimating microstructural tissue parameters [100, 119, 124]. Typically, a computational modelling and a signal-processing framework is used to estimate these parameters at a spatially localised level (i.e. within image volume sub-elements or voxels), taking into account several data measurements for that particular location. Such framework is normally applied *independently* to hundreds of thousands of voxels.

Because of the large number of voxels in a dMRI volume, the estimation of these parameters may require long computational times, even on a large computing cluster. For clinical applications, the large computational times can be prohibitive for routine use. In projects where data mining applications process large dMRI databases, such as the Human Connectome Project (HCP) [2–4] or the UK Biobank [32, 33], large computational times can limit the potential exploration and information extraction from the existing data.

However, since the computational modelling is applied independently to different voxels, the methods are inherently parallelisable. The large number of independent elements in the data and the fact that identical procedures need to be performed over each of these elements, make Graphics Processing Units (GPUs) a perfect candidate for processing these datasets. However, due to the heavy tasks involved in these procedures, the parallel design is highly non-trivial. We present in this chapter a GPU-based framework for parallelising the parameter estimation of voxel-wise models from dMRI data. As an example, we choose a model for estimating within-voxel tissue geometric patterns (in particular, patterns of fibre orientations). Nevertheless, the general ideas are applicable to other models, as we illustrate towards the end of the chapter. By addressing various low-level challenges in designing a GPU framework, we report accelerations of more than two orders of magnitude when comparing the parallel implementations with a commonly used sequential version.

Contributions of this chapter

- We explore the computational routines used for voxel-wise dMRI modelling, including two commonly used approaches: a non-linear optimisation routine, the Levenberg–Marquardt algorithm, and a Bayesian inference method, Markov Chain Monte Carlo (MCMC).
- Different designs and GPU implementations are proposed for parallelising these routines.
- Tests are performed to assess the performance gain obtained by the GPU implementations.
- Validation tests are presented, comparing the results from a parallel GPU implementation to a sequential CPU version.
- A software tool for estimating the fibre orientations on GPUs [144] has been developed and included in the FMRIB's Software Library (FSL) [38]. The tool is publicly released and already used by many projects and institutions worldwide.

Publications

Contributions from this chapter have appeared in the following:

- Alfaro-Almagro F., Jenkinson M., Bangerter N.K., Andersson J.L.R., Griffanti L., Douaud G., Sotiropoulos S., Jbabdi S., **Hernandez-Fernandez M.**, Vallee E., Vidaurre D., Webster M., McCarthy P.D.. Rorden C., Daducci A., Alexander D., Zhang H., Dragonu I., Matthews P., Miller K.L., Smith S. "Image Processing and Quality Control for the first 10,000 Brain Imaging Datasets from UK Biobank". *NeuroImage*. (*in press*).
- Sotiropoulos S.N., **Hernandez-Fernandez M.**, Vu A.T., Andersson J.L., Moeller S., Yacoub E., Lenglet C., Ugurbil K., Behrens T.E.J., Jbabdi S. "Fusion in diffusion MRI for improved fibre orientation estimation: An application to the 3T and 7T data of the Human Connectome Project". *NeuroImage*, 134: pp. 396–409. (2016).

- **Hernandez M.**, Guerrero G.D., Cecilia J.M., García J.M., Inuggi A., Jbabdi S., Behrens T.E.J., Sotiropoulos S.N. "Accelerating Fibre Orientation Estimation from Diffusion Weighted Magnetic Resonance Imaging Using GPUs". *PLoS ONE*, 8(4). (2013).
- Sotiropoulos S.N., Jbabdi S., Xu J., Andersson J.L., Moeller S., Auerbach E.J., Glasser M.F., **Hernandez M.**, Sapiro G., Jenkinson M., Feinberg D.A., Yacoub E., Lenglet C., Van Essen D.C., Ugurbil K., Behrens T.E.J. "Advances in diffusion MRI acquisition and processing in the Human Connectome Project". *NeuroImage*, 80: pp.125–43. (2013).
- **Hernandez M.**, Guerrero G.D., Cecilia J.M., Garcia J.M., Inuggi A., Sotiropoulos S.N. "Accelerating fibre orientation estimation from diffusion weighted magnetic resonance imaging using GPUs". In: *International Conference on Parallel, Distributed and Network-based Processing (PDP)*, Garching (Germany). (2012).

Software

- Tool for fitting ball & sticks model (and extensions) on GPUs: `http://users.fmrib.ox.ac.uk/~moisesf/Bedpostx_GPU`

4.1 Introduction

Diffusion MRI (dMRI) is used to study non-invasively and in-vivo the thermally-driven movements of water molecules within brain tissue. Contrast is introduced to MR images using special sequences that utilise the random motion of water molecules, which is affected by the presence of different microstructural barriers (axons, cell membranes, myelin sheaths, etc.). The measured signal is representative of how fast diffusion occurs along different directions and in different regions of the brain.

The aim of dMRI analysis is to use the diffusion measurements in order to gain information about tissue microstructure. For instance, diffusion is more anisotropic in white matter than in grey matter or CSF [102, 145], meaning that water moves preferentially along certain directions. The most popular framework to extract this information is diffusion tensor imaging (DTI) [94, 95], a voxel-wise model that uses a 3×3 symmetric matrix to characterise the diffusion process. The diffusion tensor model can be estimated using linear regression, and thus, requires very few computational resources [146].

However, the diffusion tensor model has major limitations. A voxel may comprise different tissue types. For instance, a fraction of the voxel may be occupied by white mater and another fraction by cells or CSF. This partial volume effect cannot be characterised by the diffusion tensor model [147]. Furthermore, a white matter voxel contains hundred of thousands of fibres that may be arranged in complex configurations, such as crossing, kissing or bending arrangements, and once again, this cannot be characterised correctly by this model [23, 148–151]. Several dMRI models with multiple compartments have been proposed to improve the characterisation of the diffusion process [115, 116, 121, 123–125, 152–157]. As an example, we pick here one of the most commonly used, the ball & sticks model [115, 116, 157], which solves the fibre-crossing estimation problem.

Multi-compartment dMRI models are commonly non-linear functions of the signal, and thus non-linear optimisation and Non-Linear Least Square (NLLS)

algorithms are typically used for fitting the model to the diffusion-weighted measurements [158, 159]. These algorithms use iterative optimisation procedures for finding a global solution, but their main disadvantage is the high computational time required, especially when many parameters are estimated in the model.

The presence of noise in the acquired dMRI data introduces uncertainty in the estimated parameters. An alternative for representing this uncertainty is to use a Bayesian inference framework to perform the model fitting and obtain a distribution of values for each parameter [160]. But once again, such inference methods are computationally demanding.

In this chapter we present research on the parallelisability of non-linear optimisation and Bayesian methods for fitting voxel-wise diffusion MRI models. In particular, we explore two methods, Levenberg-Marquardt [96–98] and Markov Chain Monte Carlo (MCMC) [161, 162]. We use a specific diffusion MRI model, the ball & sticks model, and we follow the same fitting procedure as the FMRIB's Software Library (FSL) [38, 116].

We propose parallel designs and GPU implementations for these routines, and we perform tests to assess the performance gains. Finally, we validate the GPU implementations comparing their results to the results obtained from a sequential CPU version.

4.2 Modelling diffusion signal: Fitting the ball and sticks model

The ball & sticks model explains the signal in each voxel of the brain using a multi-compartment representation. It assumes a fully isotropic compartment (the ball) and one or more perfectly anisotropic compartments (L sticks). The orientations of these sticks provide the principal fibre orientations in a voxel. Equation 4.1 shows the signal model when each of the $m = 1 : M$ gradient directions $\mathbf{g}_m$ is applied:

$$S_m = S_0 \left[\left(1 - \sum_{i=1}^{L} f_i \right) \exp(-b_m d) + \sum_{i=1}^{L} f_i \exp(-b_m d (\mathbf{g}_m^T \mathbf{v}_i)^2) \right] \tag{4.1}$$

where S_0 is the baseline signal without any diffusion-sensitising gradient applied, d is the mean diffusivity, b_m is the *b-value* and depends on the magnitude and duration of the m^{th} diffusion sensitising gradient and $f_i \in [0, 1]$ and $\mathbf{v}_i$ describe the volume fraction and orientation of the i^{th} stick, with:

$$\mathbf{v_i} = \begin{bmatrix} \sin(\theta_i)\cos(\phi_i) & \sin(\theta_i)\sin(\phi_i) & \cos(\theta_i) \end{bmatrix}^T \tag{4.2}$$

where $\theta_i \in [0, \pi]$ and $\phi_i \in [0, 2\pi]$.

Using non-linear optimisation or Bayesian approaches, this non-linear diffusion MRI model is fitted independently in hundreds of thousands of different voxels, which can require substantial computation time.

When fitting a non-linear model, NLLS methods are typically used. Levenberg-Marquardt (LM) [96–98] has become a popular algorithm for these optimisation problems [163]. The algorithm is a good compromise between finding a global solution and converging relatively fast.

Assuming a model $S(\mathbf{\Theta})$ with a set of parameters $\mathbf{\Theta}$ and a dataset Y containing M measurements, the main steps of LM are:

Algorithm 4.1 Pseudocode of the Levenberg-Marquardt algorithm. In our particular case the convergence criteria is met when $\lambda > 1e^{20}$ and the maximum number of iterations is set to 200.

Initialise model parameters $\mathbf{\Theta}$
$f(\mathbf{\Theta}) = \sum_{m=1}^{M}(Y_m - S_m(\mathbf{\Theta}))^2$
$Cost = f(\mathbf{\Theta})$
while !(convergence_criteria) && !(max_iterations) **do**
    $\mathbf{r} = gradient(f(\mathbf{\Theta}))$
    $\mathbf{J} = Jacobian(f(\mathbf{\Theta}))$
    $\mathbf{Hessian}_{approx} = \mathbf{J} \times \mathbf{J}^T$
    $\mathbf{H} = \mathbf{Hessian}_{approx} + \lambda \times diag(\mathbf{Hessian}_{approx})$
    $\mathbf{\Theta}_{new} = \mathbf{\Theta} - \mathbf{r} \times \mathbf{H}^{-1}$
    $Cost_{new} = f(\mathbf{\Theta}_{new})$
    if $Cost_{new} < Cost$ **then**
        $\mathbf{\Theta} = \mathbf{\Theta}_{new}$
        $Cost = Cost_{new}$
        $\lambda = \frac{\lambda}{10}$
    else
        $\lambda = \lambda \times 10$
    end if
end while

Certain steps of this algorithm are computationally expensive, especially when the dMRI data contains hundreds of diffusion-weighted measurements M or many parameters $\mathbf{\Theta}$ are estimated in the model. The most expensive steps are the calculation of the Jacobian matrix $\mathbf{J}$ and the gradient vector $\mathbf{r}$, where the partial derivatives need to be evaluated at each data measurement. The cost function $f(\mathbf{\Theta})$ used here is defined as the sum of squared differences between measurements and model predictions:

$$f(\mathbf{\Theta}) = Sum\_squares = \sum_{m=1}^{M}(Y_m - S_m(\mathbf{\Theta}))^2 \tag{4.3}$$

The evaluation of this function is relatively expensive during LM. The predicted signal from the model S_m needs to be computed for each data measurement. Finally, the equation $\mathbf{\Theta}_{new} = \mathbf{\Theta} - \mathbf{r} \times \mathbf{H}^{-1}$ is solved using LU decomposition with Gaussian elimination and pivoting, and it may be expensive when the model has a large number of parameters.

LM will provide a single point estimate, for the parameters $\mathbf{\Theta}$, that minimises the cost function. An alternative way for fitting the model is to use a Bayesian inference framework. Bayes theorem allows us to estimate the posterior distribution of the model parameters given the data, $P(\mathbf{\Theta}|Y)$ instead of a single point estimate. The posterior is proportional to the product of the model likelihood $P(Y|\mathbf{\Theta})$ and the prior distribution of model parameters $P(\mathbf{\Theta})$. One can solve this optimisation problem using a family of iterative sampling methods called Markov Chain Monte Carlo (MCMC). In particular, we use the random-walk Metropolis algorithm [161, 164], with independent sampling for each parameter, adaptive parameter proposals and non-informative priors.

The steps of Random walk Metropolis algorithm are:

Algorithm 4.2 Pseudocode describing the random-walk Metropolis algorithm.

Initialise model parameters $\mathbf{\Theta}_{1...P}$
Define $Q_{1...P}$ *proposal distributions*
$Posterior(\mathbf{\Theta}) = Likelihood(\mathbf{\Theta}) \times Prior(\mathbf{\Theta})$
for *iteration* $= 1$ to I **do**
    for *parameter* $p = 1$ to P **do**
        Draw sample from Q_p *and propose* Θ'_p
        Calculate $Prior(\mathbf{\Theta}')$
        Compute $Likelihood(\mathbf{\Theta}')$
        $Posterior(\mathbf{\Theta}') = Likelihood(\mathbf{\Theta}') \times Prior(\mathbf{\Theta}')$
        $\alpha = \min\left[1, \frac{Posterior(\mathbf{\Theta}')}{Posterior(\mathbf{\Theta})}\right]$
        Draw random sample from uniform $U(0, 1)$
        if *random* $< \alpha$ **then**
            accept sample
            $\mathbf{\Theta} = \mathbf{\Theta}'$
            $Posterior(\mathbf{\Theta}) = Posterior(\mathbf{\Theta}')$
        else
            reject sample
        end if
    end for
    Every I' *iterations: adapt*$(Q_{1...P})$
end for

The most expensive step in this algorithm is the computation of the Likelihood. If we assume zero-mean Gaussian noise with variance σ^2, the Likelihood becomes a

function of the sum of squared differences between measurements and model predictions:

$$Likelihood(\mathbf{\Theta}) = (2\pi\sigma^2)^{-\frac{M}{2}} \exp\left(\frac{-1}{2\sigma^2}\sum_{m=1}^{M}(Y_m - S_m(\mathbf{\Theta}))^2\right) \tag{4.4}$$

The predicted signal from the model $S_m(\mathbf{\Theta})$ needs to be computed for each data measurement using the proposed parameters in each iteration.

For fitting the ball & sticks model to the data we use the same routines as the FSL software library [38, 116]:

Algorithm 4.3 Pseudocode describing the fitting process of the ball & sticks model.

```
1: for voxel = 1 to N do
2:     Linear fit diffusion tensor ()
3:     Levenberg ball & sticks ()
4:     MCMC ball & sticks ()
5: end for
```

Linear regression and diffusion tensor model are used to initialise some of the parameters, since they are common to both, the diffusion tensor model and the ball & sticks model. Next, a first estimation of the model is performed using Levenberg algorithm (Marquardt modification is not used), and the output provides a starting point to the MCMC algorithm, where finally, a distribution for each of the model parameters is estimated.

4.3 Modelling diffusion signal on GPUs: Design and Implementation

As we discussed in detail in Section 2.2, the first consideration towards a parallel design for a computationally demanding application is to assess its parallelisability, and whether it is limited by various factors, such as heavy tasks, code divergences, memory access patterns or intense communication between threads.

We performed this assessment by finding what portions of the application are computationally more expensive and how parallelisable they are. We measured the

times of the different routines used in the sequential version of the application: a linear regression routine fitting the diffusion tensor model, the Levenberg algorithm fitting the ball & sticks model, and the MCMC algorithm fitting the ball & sticks model. Figure 4.1 shows the portions of time spent on these routines.

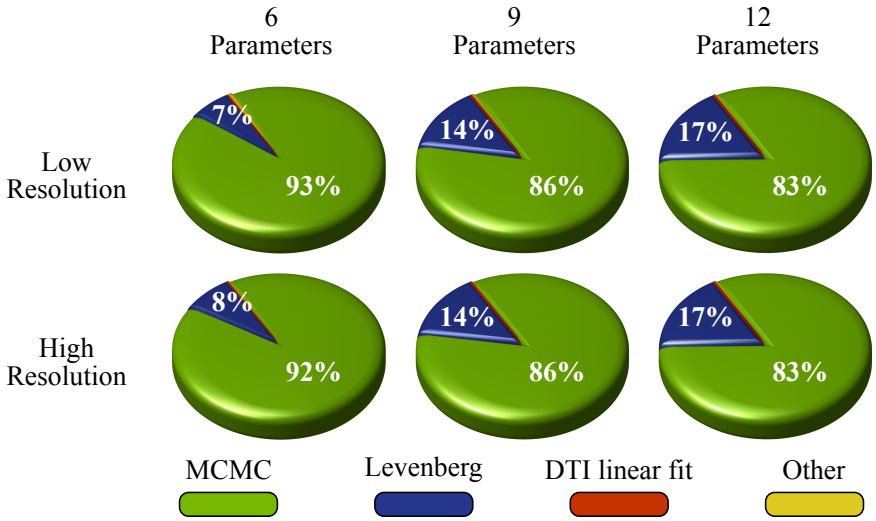

Figure 4.1: Portions of time spent on three different routines during the fitting process of the ball & sticks model: MCMC algorithm (green), Levenberg algorithm (blue) and linear fitting of diffusion tensor model (red). The portion of time spent on preparing the data for these routines and reading/writing to disk is represented in yellow (Other). The first row shows the time portions fitting a low resolution dataset (voxel size $2.5 \times 2.5 \times 2.5\ mm^3$) and the second row shows the time portions fitting a high resolution dataset (voxel size $1.35 \times 1.35 \times 1.35\ mm^3$). The three columns show different numbers of parameters, 6, 9 and 12 being estimated by the model (in the ball & sticks model the number of parameters depends on the number of anisotropic compartments or sticks L). The number of iterations in MCMC is fixed to 2,250 (default value in FSL).

As expected, the MCMC algorithm occupies most of the application time. The portion of time spent on the Levenberg algorithm grows as the problem gets more complicated, when increasing the number of model parameters.

Since the time spent on the MCMC algorithm represents on average 87% of the application, we first focus on the parallelisation of this algorithm. According to Equation 2.2 the potential speedup that we could obtain parallelising this routine is:

$$Maximum\ Speedup = 0.13 + C \times 0.87 \tag{4.5}$$

If we have a large number of independent tasks and we assign them to different processing units C, we could potentially obtain a substantial performance improvement. Typically, dMRI datasets contains hundreds of thousands of voxels, and the same fitting routines need to be executed independently over all of them (SIMD).

This factor and the large number of processing units C available on a GPU, make these devices a high potential candidate for parallelising these algorithms. Moreover, a typical dMRI dataset is not very large (few hundred *megabytes*), and thus, the entirely dataset can be allocated in the GPU global memory.

In the following sections we explored three designs for exploiting the parallelism of two fitting routines with GPUs. First, we focus on the MCMC algorithm, given that it is the most expensive routine, and then we present GPU-parallel solutions for the Levenberg-Marquardt algorithm.

4.3.1 Parallelising the MCMC algorithm

Parallel design Version 1

A straightforward approach towards a parallel design is motivated by the independent nature of the fitting process across voxels in Algorithm 4.3. The execution of MCMC can be parallelised creating several threads of execution, each one processing the MCMC routine in a different voxel and on a different processing unit or core (see Algorithm 4.4).

Algorithm 4.4 Pseudocode of a parallel version of MCMC algorithm for fitting a dMRI model. The instances of the MCMC algorithm for fitting the model in V voxels are distributed amongst V threads. Each thread uses an identifier number ($thread_{ID}$) for selecting a different voxel.

```
1: for voxel = 1 to V do
2:     Linear fit diffusion tensor ()
3:     Levenberg ball & sticks ()
4: end for
5: Run V threads in parallel
6: Parallel section {
7:     voxel = voxels[thread_ID]
8:     MCMC ball & sticks (voxel)
9: }
```

We built a CUDA implementation of this parallel design creating thousands of threads, which execute the MCMC algorithm (kernel). Figure 4.2 depicts the parallel CUDA implementation. The GPU resources, including cores and memory spaces, are shared amongst all the threads. Synchronisation is not needed and there are no divergences, as all the threads execute the same number of MCMC iterations.

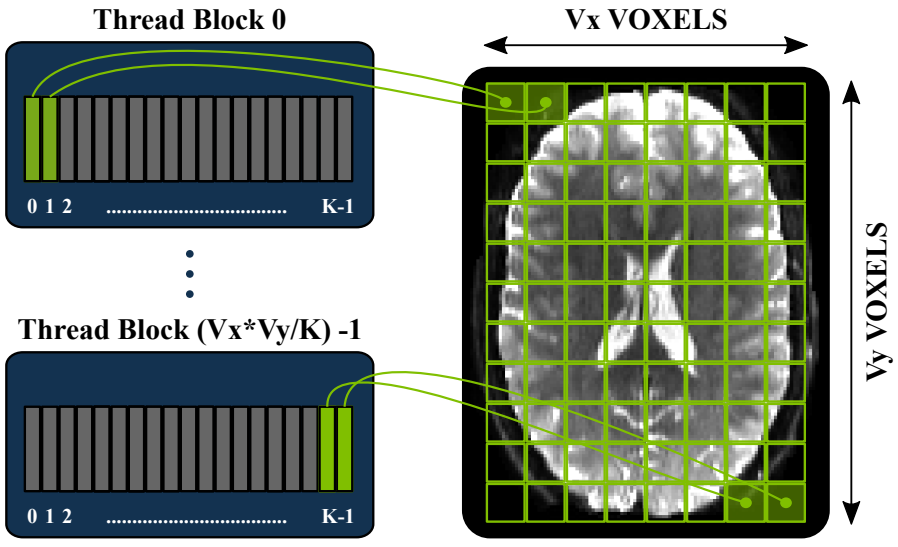

Figure 4.2: A parallel design for fitting dMRI models on a GPU (parallel design Version 1). The fitting process of V voxels is distributed amongst V CUDA threads, which are grouped into blocks of K threads.

In this parallel design, each thread will need to store the entire information for fitting the model in a voxel, including:

- Diffusion data measurements and associated measurement points (gradients intensities and directions).
- Model parameters (value of parameters, priors, proposal distributions, etc.).
- Posterior distribution estimates and Likelihood values.

This data need to be accessed and updated in each MCMC iteration, which can be very costly if it is stored in global memory. Ideally we want to occupy the low-levels of the GPU memory hierarchy, such as the Shared memory and the registers, which have low latencies. However, with such a naive voxel-based parallel design, the amount of data needed does not fit in these limited memory spaces, and thus, the threads will need to access the global memory in each iteration. This leads to high latencies (the warps will be held for many cycles), and the threads will spend most of the time waiting for the data to be served from global memory, instead of using the computational resources.

The only way to hide these latencies and avoid a waste of computational resources is by exposing high enough parallelism for having a high Streaming Multiprocessor (SM) occupancy, i.e. having many warps running on the SM simultaneously for switching between them (see Section 2.8.1). However, in this parallel design each thread needs to cope with many computational operations involved in the fitting process, which entails heavy tasks. Assigning heavy tasks to the threads will make them to consume a lot of registers. Moreover, large amount of Shared memory is used per block, as each thread stores private information corresponding to a different voxel. We only obtain 22% of occupancy in the SMs.

If we analyse the parallel MCMC routine, the most expensive operation is the computation of the Likelihood. It needs to be computed in each iteration and for each proposed parameter (step 8 in Algorithm 4.2 and Equation 4.4). Such an operation is expensive for a CUDA implementation for three reasons:

- Given the complexity of some dMRI models, the computation of the model predicted signal $S(\mathbf{\Theta})$ might be expensive.

- Because of the low-level memory limitations on GPUs, and the large amount of measurements M typically used in dMRI, these data measurements need to be located in global memory, which involves high latencies. Moreover, with this design, every time a warp access to global memory, 32 different segments are requested (see Section 2.8.3). Each thread of a warp is requesting access to measurements of different voxels, and measurements from different voxels are not in consecutive memory locations, causing highly inefficient uncoalesced memory accesses (Figure 2.10).
- A single thread needs to iterate over all the M diffusion-weighted measurements and thus, evaluate many signal predictions $S(\mathbf{\Theta})_m$ (see Equation 4.4).

These considerations and evaluations led to a second parallel design, explained below.

Parallel design Version 2

The uncoalesced memory accesses problem can be solved by rearranging the data in memory or by modifying the access patterns. Also, it would be desirable to deal with the other limitations by reducing the thread tasks complexity and the amount of Shared memory consumed per block. We propose an alternative parallel design for the MCMC routine with light-weight threads running simple tasks and using few computational resources.

In this design, the MCMC routine of each voxel is performed by a group of threads (rather than by a single thread as before). A second level of parallelisation occurs within the computation of the likelihood function in each voxel. The computation of the model predicted signal for the m^{th} measurement, and subsequently of the squared residuals $(Y_m - S(\mathbf{\Theta})_m)^2$, is independent across measurements, and it can be distributed amongst different threads as shown in Algorithm 4.5 and Figure 4.3. Now, a thread needs to compute the model predicted signal for only one or a few measurements.

Algorithm 4.5 Pseudocode of a parallel version of MCMC algorithm for fitting a dMRI model in V voxels. The computation of V voxels is distributed amongst V groups of threads, each one with W warps ($32W$ threads). Each thread T within a group collaborate for calculating the Likelihood. The computation of the squared residuals for M measurements is distributed amongst the threads of a group and Shared memory is used for reducing the results. There are some steps of the algorithm that are computed by only one thread of the group (T_0). Synchronisation and communication between the threads within a group is necessary. *mod* is the modulo operation.

```
1:  Run V × 32W threads in parallel
2:  Parallel section {
3:      voxel = voxels[thread_ID / 32W]
4:      T_ID = mod(thread_ID, 32W)
5:      for iteration = 1 to I do
6:          for parameter = 1 to P do
7:              if T_0 then
8:                  MCMC_Pre_Likelihood_steps ()
9:              end if
10:             synchronise ()
11:             transfer updated parameters Θ^voxel from T_0
12:             for (measurement m = T_ID; m < M; m+ = 32W) do
13:                 calculate Residuals_m(voxel, Θ^voxel)
14:             end for
15:             synchronise ()
16:             rr = reduction(Residuals) in Shared memory
17:             if T_0 then
18:                 Compute_Likelihood(rr)
19:                 MCMC_Post_Likelihood_steps ()
20:             end if
21:         end for
22:     end for
23: }
```

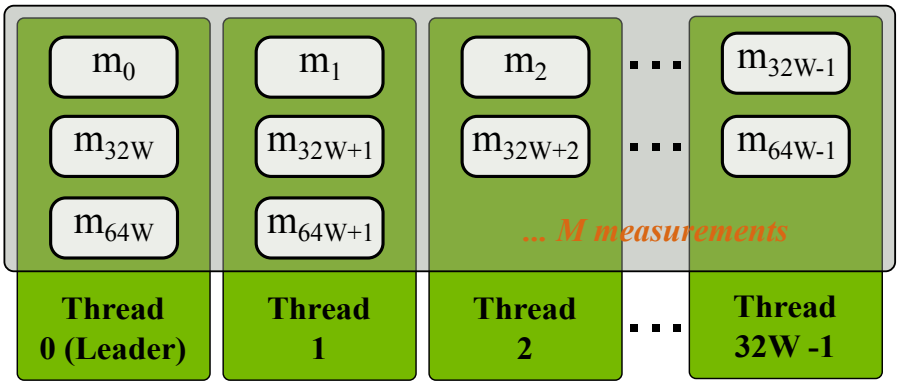

Figure 4.3: Description of how the computation of the squared residuals for $m_0 \cdots m_{M-1}$ measurements is distributed amongst the $32W$ threads of a block with W warps in a case with $M > 32W$.

We implement this design on GPUs, assigning each group of threads to a CUDA block of $32W$ threads, and W warps per block (see Figure 4.4).

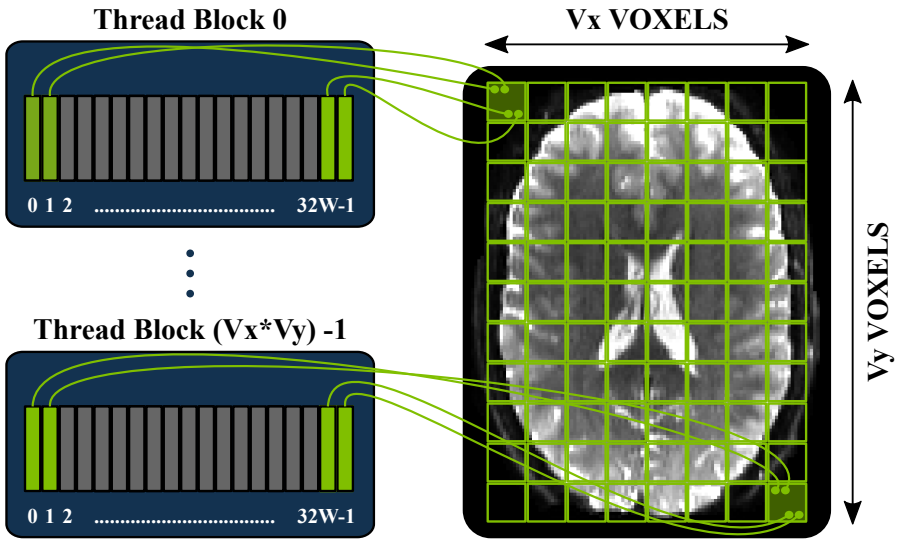

Figure 4.4: A parallel design for fitting dMRI models on a GPU (parallel design Version 2). The fitting process of each of the V voxels is assigned to a block of $32W$ threads. The threads within a block collaborate for within-voxel computations.

There are certain steps of the algorithm that are performed by only one thread (T_0) of each block, while the other threads are waiting at a barrier. The threads within a group need to be explicitly synchronised and establish a communication, for reducing the squared residuals for the final likelihood calculation, and for obtaining new proposals from T_0 (only one thread per block draw samples). Figure 4.5 describes these steps. Shared memory is used for communication.

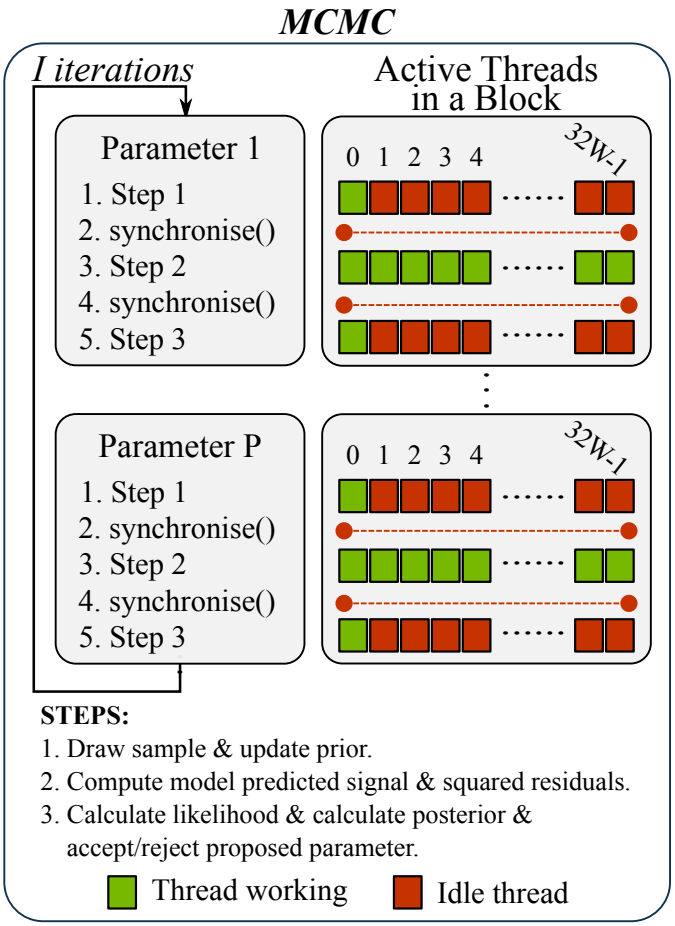

Figure 4.5: In each of the I MCMC iterations, and for each of the P model parameters, there are some steps (before and after the computation of the squared residuals) where only one thread of a block with $32W$ threads is active. The rest of the threads are idle waiting at a synchronisation barrier.

This design solves most of the previous limitations:

- It does not involve heavy tasks. Now threads are very light and they consume less computational resources; registers and Shared memory. Now a group of threads will need the same amount of Shared memory as a single thread in the previous design. With this design we reach 100% of occupancy when using $W = 4$ warps per block.
- The most expensive operation in the MCMC algorithm, the computation of the likelihood, is now processed by a group of threads instead of by a single thread.
- Various data are still located in global memory, but now the memory accesses are coalesced. When a warp accesses this data, only 4 segments are requested

($L1$-uncached), because the pieces of data that the threads need (data from the same voxel) are in consecutive memory locations. The efficiency of memory accesses increases significantly.

However, divergence, synchronisation and communication between threads are introduced with this design.

Parallel design Version 3

In the previous design, when the number of measurements M is not close to a multiple of the number of threads per block $32W$, GPU resources are wasted (see Figure 4.3). In such case, the workload is not balanced across threads, and some of them are idle at some steps. Even worse, if $M < 32W$ there are threads in the block that do not execute anything at all during the application, however, they consume resources. A simple solution would be to use the smallest possible block size ($W = 1$). However, this creates a trade-off. Setting a small block size leads to the problem that more blocks are needed to achieve maximum occupancy. But the number of blocks that can be allocated to an SM is limited (16), and thus, a low occupancy would be achieved instead.

The solution that we propose here is to use a small number W of warps for computing the fitting process of a single voxel, and increase the number of threads per block by assigning several voxels to the same block, instead of just one (see Figure 4.6). A block will then have $32W \times B$ threads, where $B \geq 1$ is the number of voxels assigned to each block.

We perform a test to verify that this solution gives a better performance and to identify the optimal W and B. Figure 4.7 shows the execution time of some of these combinations, including the configuration used in the parallel design Version 2 ($W = 4,\ B = 1$, i.e. one block with 128 threads fitting the model in just one voxel). Any configuration with $B > 1$ reduces the execution time, especially when the data does not have many measurements. We found that the best configuration was $W = 1$ and $B = 8$, i.e., using 256 threads per block, and each warp fitting the

model for a different voxel.

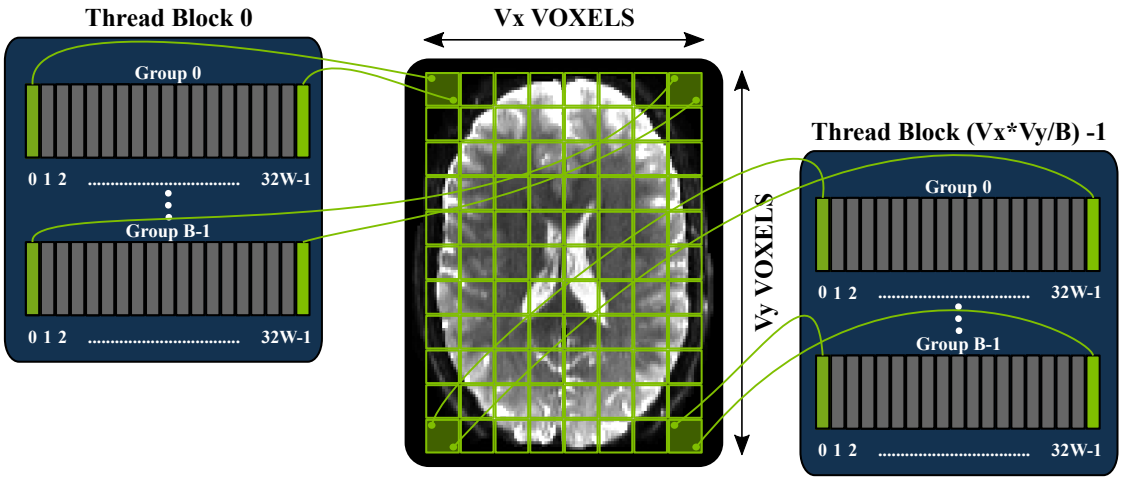

Figure 4.6: A parallel design for fitting dMRI models on a GPU (parallel design Version 3). The V voxels of a dataset are divided into groups of B (voxels per block), and the fitting process of each of these groups is assigned to different CUDA blocks. Inside a block, one or few warps W(warps per voxel) collaborate for within-voxel computations.

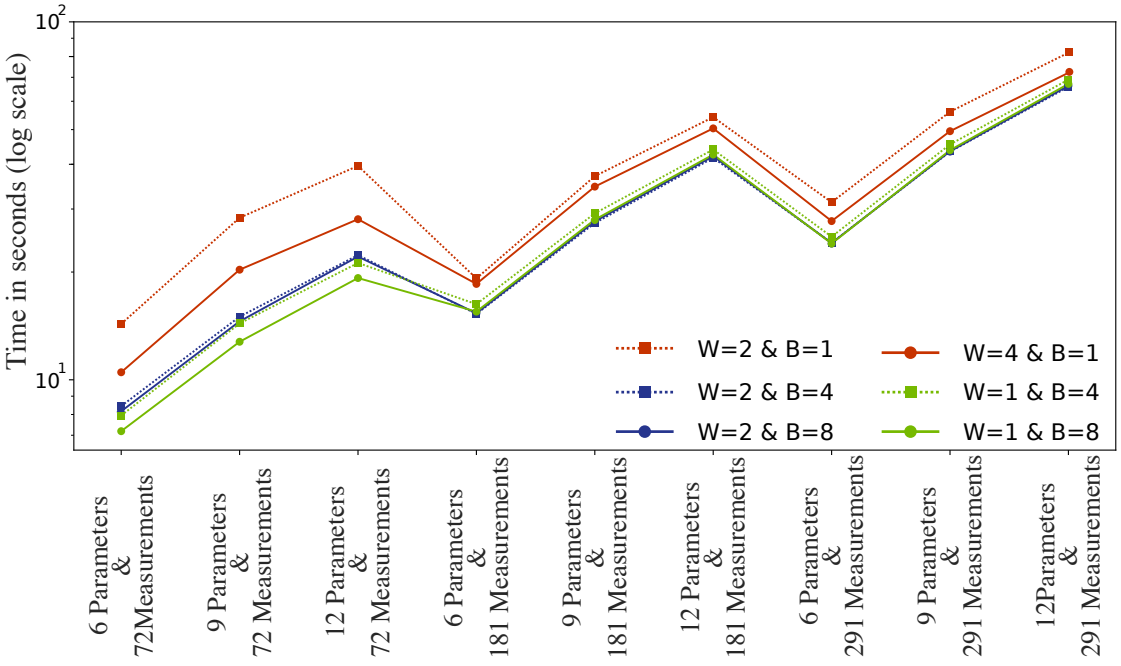

Figure 4.7: Execution times of the parallel CUDA implementation of MCMC routine depicted in Figure 4.6 processing a slice from a high resolution dataset ($1.35 \times 1.35 \times 1.35\ mm^3$ voxels), and testing different configurations of W warps per voxels and B voxels per block. Different number of measurements (72/181/291) and model parameters (6/9/12) were tested.

4.3.2 Parallelising the Levenberg-Marquardt algorithm

After parallelising the most expensive routine of our application, the MCMC algorithm, we measure again the computation time needed by each routine. As shown in Figure 4.8, the parallelised MCMC is no longer a bottleneck.

On average, 94% of the time is now spent on initialising the MCMC using the Levenberg algorithm. According to Equation 2.2, the potential speedup that we can obtain parallelising this routine is:

$$Maximum\ Speedup = 0.06 + C \times 0.94 \tag{4.6}$$

Once again, the fact that this routine can be divided into a large number of independent tasks, makes GPUs, with a large number of computational units, a perfect platform for implementing a parallel solution.

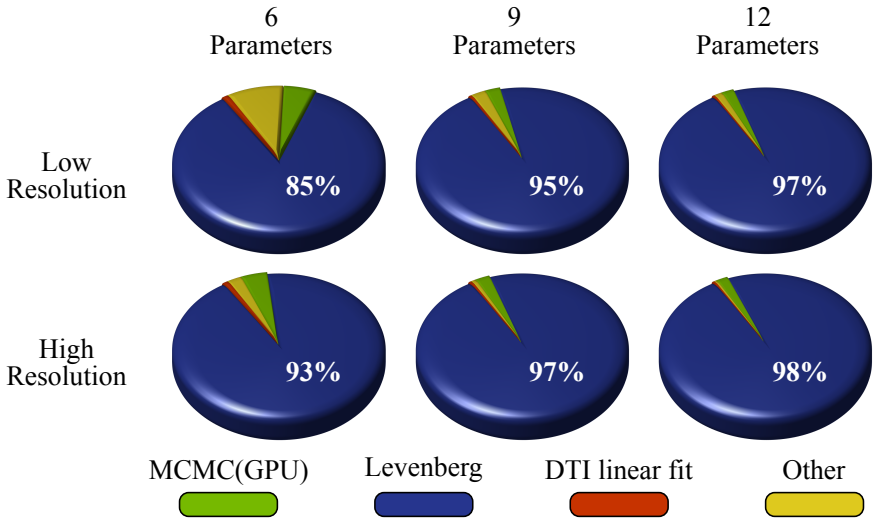

Figure 4.8: Time spent on three routines for fitting the ball & sticks model (as Figure 4.1) running a parallel version of the MCMC algorithm on a GPU.

Here, we use the first two designs proposed in the previous section (parallel design Version 1 and parallel design Version 2) for implementing GPU-accelerated solutions for the Levenberg algorithm.

As in the MCMC routine, the fitting process is completely independent across voxels. We use the parallel design Version 1 (Figure 4.2) for implementing a first GPU solution. The fitting process of different voxels can be distributed amongst CUDA threads. Algorithm 4.6 describes this implementation.

Algorithm 4.6 Pseudocode describing a parallel version of Levenberg algorithm for fitting a dMRI model. The fitting process in V voxels is distributed amongst V threads.

```
1: for voxel = 1 to N do
2:     Linear fit diffusion tensor ()
3: end for
4: Run V threads in parallel
5: Parallel section {
6:     voxel = voxels[thread_ID]
7:     Levenberg ball & sticks (voxel)
8: }
9: Parallelised MCMC
```

Similar to the MCMC implementation, this simple design has limitations. Each CUDA thread needs to perform heavy tasks, executing an instance of the Algorithm 4.1, and it will need to store a lot of data, including:

- Diffusion data measurements and associated measurement points (gradients intensities and directions)
- Model parameters
- Gradient vector (size P)
- Jacobian Matrix (size $M \times P$)
- Approximated Hessian matrix (size $P \times P$)
- New proposed parameters (size P)

where P is the number of parameters and M the number of measurements.

All this data cannot be allocated at the low-levels of the GPU memory hierarchy, and thus some is allocated in global memory, leading to high latencies. Moreover, the large amount of Shared memory required per block and the large number of registers consumed (manually limited to 64 registers per thread in this case) by the heavy threads of a block, constraints the occupancy of the SMs to 50%.

Additionally, there are a lot of uncoalesced memory accesses to global memory, because the measurements of different threads (assigned to different voxels) are not in consecutive memory locations.

Instead, we can use the parallel design Version 2 (Figure 4.4) of the previous section to tackle these limitations. In this design the fitting process of each voxel is assigned to a thread block, threads within a block compute certain steps in parallel, and they get synchronised to obtain the final result. Specifically, the computation of the squared residuals, $(Y_m - S_m(\mathbf{\Theta}))^2$ required for the calculation of the cost function (Equation 4.3), is distributed as evenly as possible amongst the threads within a block.

Besides that, the computation of the partial derivatives, required for the calculation of the gradient $\mathbf{r}$ and the Jacobian matrix $\mathbf{J}$, is also distributed amongst the threads within a block. This distribution is more complicated. The algorithm iterates over each data measurement and over each model parameter for evaluating the partial derivatives (see Algorithm 4.7). The iterations over the model parameters perform different instructions on different data (MIMD), as the derivatives may be different for each parameter Θ_p. Thus, the parallelisation of this internal loop is not suitable for SIMD architectures, such as GPUs, where all the threads execute the same instructions. However, the iterations over the measurements m_i perform the same instruction over different data (SIMD), and this computation is distributed amongst the threads within a block, performing the loop over the parameters sequentially. The final parallel implementation is described in Algorithm 4.8.

Algorithm 4.7 Computation of the partial derivatives in Levenberg algorithm. The partial derivative respect each model parameter Θ_p needs to be evaluated at each data measurement m_i.

```
1: for measurement m_i do
2:     for each model parameter Θ_p do
3:         Compute partial derivative (m_i) respect (Θ_p)
4:     end for
5:     Calculate gradient (Partial derivatives)
6:     Calculate Jacobian (Partial derivatives)
7: end for
```

Algorithm 4.8 Pseudocode of a parallel version of Levenberg algorithm for fitting a dMRI model in V voxels. The computation of the V voxels is distributed amongst V groups, each one with W warps. Each thread T within group collaborates in the computation of the residuals and partial derivatives for M measurements. Shared memory is used for reducing the residuals. The partial derivatives respect the model parameters $\mathbf{\Theta}$ are computed sequentially by each thread. There are some steps of the algorithm that are computed by only one thread of the group (T_0). Synchronisation and communication between the threads of a group is necessary. *mod* is the modulo operation.

1: *Run* $V \times 32W$ *threads* **in parallel**
2: ***Parallel section*** **{**
3: $voxel = voxels[thread_{ID} \ / \ 32W]$
4: $T_{ID} = mod(thread_{ID}, 32W)$
5: **for** *iteration* $= 1$ to I **do**
6: **for** $(measurement\ m = T_{ID};\ m < M;\ m+ = 32W)$ **do**
7: **for** *Parameter* $p = 0$ to P **do**
8: $Partial\_derivative(m_i)\ respect(\Theta_p^{voxel})$
9: **end for**
10: **end for**
11: *synchronise ()*
12: *transfer Partial_derivatives to* T_0
13: **if** T_0 **then**
14: $\mathbf{r}$ = *Calculate gradient (Partial_derivatives)*
15: $\mathbf{J}$ = *Calculate Jacobian (Partial_derivatives)*
16: $\mathbf{H}$ = *Calculate Hessian approximation (*$\mathbf{J}$*)*
17: **LU** $solver(\mathbf{r}, \mathbf{H})$ & $Calculate(\mathbf{\Theta}_{new}^{voxel})$
18: **end if**
19: *synchronise ()*
20: **for** $(measurement\ m = T_{ID};\ m < M;\ m+ = 32W)$ **do**
21: $calculate\ Residuals_m(voxel, \mathbf{\Theta}_{new}^{voxel})$
22: **end for**
23: *synchronise ()*
24: $rr = reduction(Residuals)$
25: **if** T_0 **then**
26: $Calculate\ cost\ function(rr)$
27: $accept\ /\ Reject(\mathbf{\Theta}_{new}^{voxel})$
28: *adapt* λ
29: **end if**
30: *synchronise ()*
31: *transfer* $\mathbf{\Theta}_{new}^{voxel}$ *from* T_0
32: **end for**
33: **}**

Some steps of the algorithm are performed by only one thread, T_0. The threads will need to be explicitly synchronised and they use Shared memory for reducing the squared residuals, transferring the partial derivatives to T_0, and copying the new proposed parameters from T_0. Figure 4.9 describes these steps.

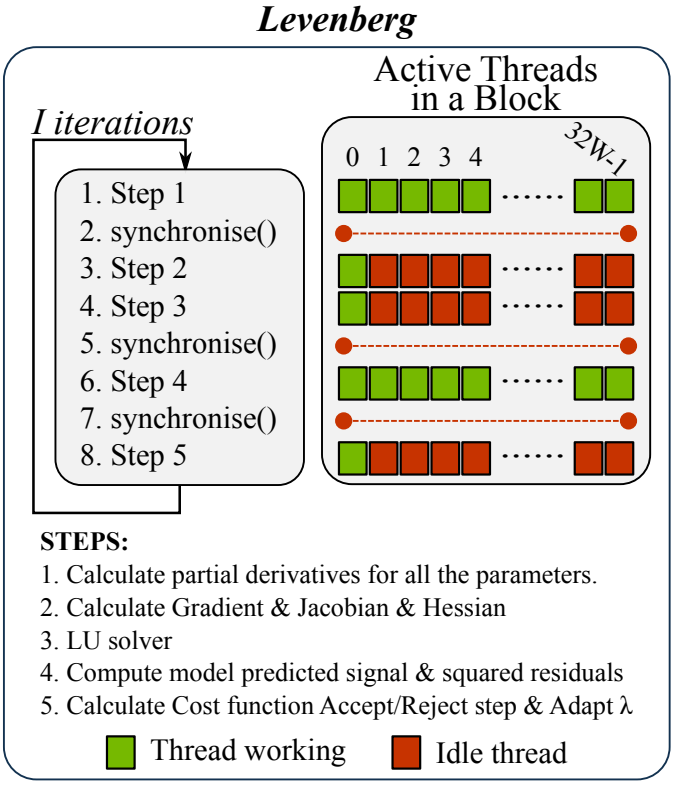

Figure 4.9: In each of the I Levenberg algorithm iterations, some steps $(2, 3$ and $5)$ are performed by only one thread of a block with $32W$ threads. The rest of the threads are idle waiting at a synchronisation barrier.

Using this design, even if synchronisation and communication is needed between threads, and divergence is introduced, we tackle most of the previous limitations:

- Threads are lighter, which means less GPU resources consumed per thread. There is more available Shared memory per thread, so more data can be stored there. The gradient, the Jacobian Matrix, the Hessian Matrix, and the proposed parameters are now stored in Shared memory. Even increasing the use of Shared memory, a 100% of SM occupancy is reached when $W = 4$ warps are allocated per block.

- The most expensive steps of Levenberg algorithm (computation of partial derivatives and residuals) are now processed by a group of threads instead of by a single thread.
- The global memory accesses to the data measurements are now coalesced.

4.3.3 Performance limitations

Once the most computationally expensive routines have been parallelised, we find that the application is spending a large portion of time on data handling, reading and writing to disk (Figure 4.10). These tasks cannot be parallelised and the figure suggests that we are approaching the maximum reachable performance.

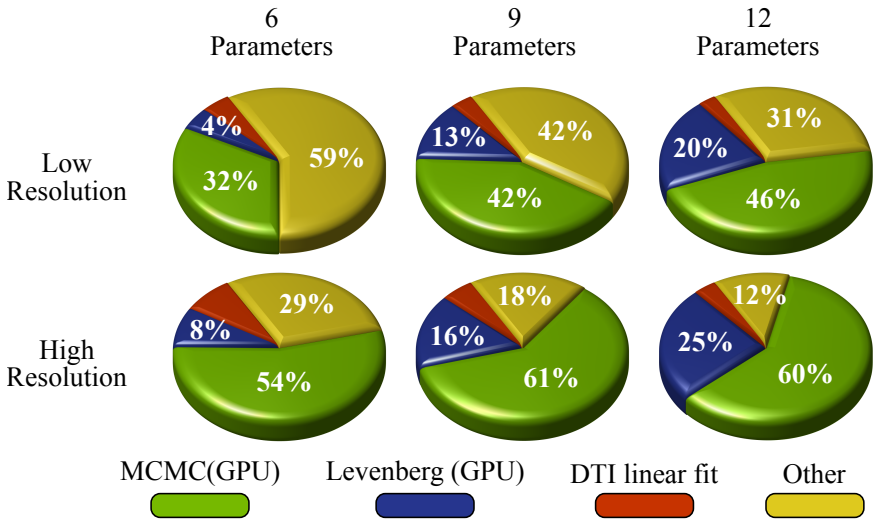

Figure 4.10: Time spent on three routines for fitting the ball & sticks model (as Figure 4.1) running parallel versions of the MCMC and Levenberg algorithms on a GPU.

MCMC is again the routine where the application spends most of the time. Analysing the MCMC kernel with the NVIDIA profiler [165], we find that it is saturating the global memory bandwidth (see Figure 4.11). Many accesses to this memory are required for reading the data measurements every time that the likelihood needs to be computed. Although this data is not located in the lower levels of the GPU memory hierarchy, the accesses are particularly efficient, as shown

in the figure. This is a highly desirable feature in the design (i.e. if we have to use global memory, the memory accesses need to be as efficient as possible).

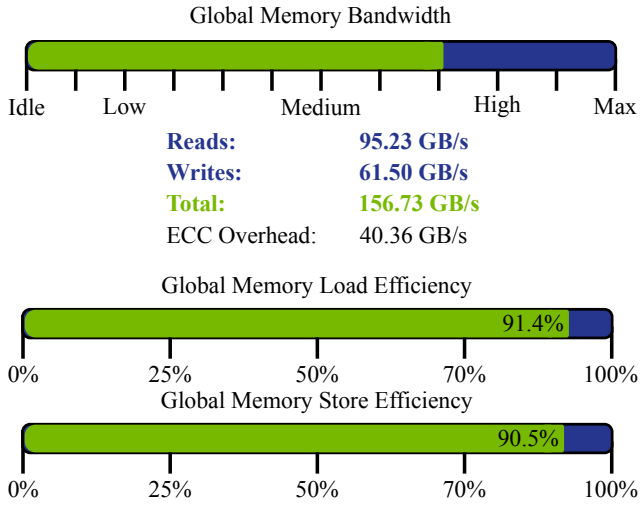

Figure 4.11: Performance reported by NVIDIA Profiler executing the MCMC kernel on a single NVIDIA K80 GPU. The application is using very efficiently a large percentage of the global memory bandwidth. Table 4.1 shows the main features of this GPU.

Another limitation comes from the divergences introduced by the second and third parallel designs. There are some steps in the MCMC and Levenberg routines performed by only one thread of a group (see Figure 4.5 and Figure 4.9). Some computational resources are wasted because the idle threads are consuming resources, even if they are not performing any computation. This is more notable during the LU decomposition step in the Levenberg routine, which is performed by only one thread of a group. We are proposing a better solution for this step in the next chapter.

Finally, the synchronisation barriers and the communication between threads required in the second and third parallel designs also contribute to the limitation of the application performance. However, despite the need of these extra actions, these two designs achieve considerable performance improvements.

4.3.4 Using Multiple GPUs

A higher-level of parallelism can be used to enhance even more the performance of the application, using very large groups of voxels and a multi-GPU system. We can divide a single dataset into groups of voxels and assign each group to a different GPU. The different GPUs do not need to communicate because the groups of voxels are totally independent, apart from the final step of writing the results to volumetric files. This can also be applied to a CPU multi-core processor or to a cluster of processing nodes.

4.4 Results: Performance gains

Hardware - Software Features

In order to test our parallel designs we used an Intel host system with NVIDIA GPUs. The system is comprised of 2 Intel Xeon E5-2680 v3 2.50 GHz processors, each one with 12 cores (24 CPU cores in total and 48 threads), and it has a total of 384 GB (24×16 GB) RDIMM memory. The system has 2 NVIDIA K80 accelerators (Error Correcting Codes ECC enabled), connected each one to a different processor via PCI express v3. Each NVIDIA K80 accelerator has 2 GPUs. In total, the system has 4 GPUs. The server runs a Linux Operating system with the Centos release 6.8 distribution. We compiled the tested application using CUDA 7.5 (V.7.5.17) and gcc 4.4.7 compilers.

Experiments were performed on the system, with the sequential version of the algorithm running on a single CPU core (and running a single thread) and the parallel versions on a single GPU. Multi-GPU and multi-core experiments were also performed using all the available CPU cores (24 and running 48 threads), and all the available GPUs (4) in the system.

Major features, including peak performance, of the GPU accelerators and the CPU host are summarized in Table 4.1 and Table 4.2 [78, 166].

Element	Feature	NVIDIA Tesla K80 accelerator
Processing Units	Number of GPUs per accelerator	2 (Tesla GK210)
	Number of SMs per GPU	13
	Single precision cores per SM	192
	Double precision cores per SM	64
	Special Function Units per SM	32
	Load/Store Units per SM	32
	Texture Units per SM	16
	Core clock	560 MHz
Memory	DRAM - Main memory per GPU	12 GB
	L2 Cache per GPU	1,536 KB
	Shared memory per SM	80/96/112 KB
	L1 cache per SM	16/32/48 KB
	Read Only Cache per SM	48 KB
	Constant memory cache per SM	10 KB
	Register File per SM	512 KB
	Memory clock	2,500 MHz
	Bandwidth per GPU	240 GB/s
Schedulers/ Dispatchers	Warp Schedulers per SM	4
	Instruction dispatch units	8
Processing Power	Peak single precision	$\mathbf{2 \times 4.365}$ ***TFLOPS***
	Peak double precision	$\mathbf{2 \times 1.45}$ ***TFLOPS***
Power Consumption		300 $watts$
Price (January 2017)		$\sim$ £4,960

Table 4.1: Major features of NVIDIA Tesla K80 GPU accelerator.

Element	Feature	Intel Xeon E5-2680 v3
Processing Units	Number of Cores (per processor)	12
	Number of Threads (per processor)	24
	Clock frequency	2.50 GHz
Memory	DRAM - Main memory (system)	384 GB
	L1 Cache (per processor)	30 MB
Processing Power per processor	Peak single precision	**960 *GFLOPS***
	Peak double precision	**480 *GFLOPS***
Processing Power per single core	Peak single precision	**80 *GFLOPS***
	Peak double precision	**40 *GFLOPS***
Power Consumption		120 $watts$
Price (January 2017)		$\sim$ £1,760

Table 4.2: Major features of Intel Xeon E5-2680 v3 processor.

Diffusion–weighted MRI data

The GPU design has been used so far to process thousands of datasets. Here, we illustrate performance gains on various subsets of some exemplar data with both high and low spatial resolutions. Data were acquired in a $3T$ Siemens Magnetom Prisma research imaging system and diffusion-weighting was introduced using single-shot EPI. For the high resolution dataset, an in-plane resolution of 1.35×1.35 mm^2 and 1.35 mm slice thickness were used ($TR = 5.59$ s, $TE = 94.6$ ms, 32-channel coil, 6/8 partial Fourier). 134 slices were acquired in total and diffusion weighting was applied in $M = 291$ evenly spaced directions, with 21 directions $b = 0$ s/mm^2, 90 directions $b = 1{,}000$ s/mm^2, 90 directions $b = 2{,}000$ s/mm^2 and 90 directions $b = 3{,}000$ s/mm^2. (HCP-style acquisition as in [4]).

For the low resolution dataset, a voxel size of $2.5 \times 2.5 \times 2.5$ mm^3 was used ($TR = 2.5$ s, $TE = 75$ ms, 32-channel coil, 6/8 partial Fourier) and 72 slices were acquired in total. Diffusion weighting was applied in $M = 291$ evenly spaced directions, with 21 directions $b = 0$ s/mm^2, 90 directions $b = 1{,}000$ s/mm^2, 90 directions $b = 2{,}000$ s/mm^2 and 90 directions $b = 3{,}000$ s/mm^2.

A multiband factor of 3 was employed in all cases [167, 168].

A set of sub-sampled datasets was generated from these acquisitions for performance exploration purposes. Datasets with 36, 72, 108, 145, 181, 218 and 254 diffusion-weighted measurements were generated by removing evenly spaced directions from the original 291 diffusion-weighted measurements. We used the above datasets to assess the computational performance of the parallel GPU implementations.

Performance of parallel GPU implementations

We present the performance results and the accelerations achieved by the GPU implementations under different aspects:

- Comparison of the parallel implementations of the MCMC and Levenberg routines to their sequential versions. The routines are executed on a single GPU and on a single CPU core respectively.
- Overall comparison of the whole model fitting application. A sequential version is executed on a single CPU core and the best parallel implementation is executed on a single GPU.
- Comparison of a multi-GPU to a CPU multi-core system running the whole model fitting application.
- Scalability of performance when varying the spatial resolution of the data. We use a low resolution dataset (voxels size $2.5 \times 2.5 \times 2.5\ mm^3$) and a high resolution dataset (voxels size $1.35 \times 1.35 \times 1.35\ mm^3$).
- Scalability when varying the angular resolution of the data. We use datasets with 36, 72, 108, 145, 181, 218, 254 and 291 diffusion-weighted measurements.
- Scalability when increasing the number P of estimated model parameters. In the ball & sticks model we use 1, 2 and 3 anisotropic compartments or sticks, which requires the estimation of 6, 9 and 12 parameters respectively.

The three parallel versions designs are used for the evaluations of the MCMC and Levenberg (only two parallel versions) routines:

- GPU v1 (parallel design Version 1): Each thread fits the model in a single voxel. (Figure 4.2).

- GPU v2 (parallel design Version 2): A block of threads fits the model in a single voxel and threads within a block collaborate for within-voxel computations (Figure 4.4).
- GPU v3 (parallel design Version 3): A block of threads fits the model in eight voxels. Inside a block, each warp fit the model in a single voxel, and threads within a warp collaborate for within-voxel computations (Figure 4.6).

4.4.1 MCMC routine evaluation

Figure 4.12 and Figure 4.13 show the execution times of the MCMC routine processing a low and a high resolution dataset respectively. It can be seen that the GPU versions achieve substantial reductions in the execution time.

The first parallel version provides accelerations that are higher than an order of magnitude (in the range of 27× to 42×), while versions 2 and 3 provide accelerations of more than two orders of magnitude (in the range of 70× to 280×). On average, version 2 improves upon version 1 by a factor of 5.42, and version 3 improves upon version 2 by a factor of 1.4. The performance gains of version 3 are more remarkable when the datasets have fewer diffusion measurements (e.g. version 3 is 2.2× faster than version 2 in processing datasets with 36 measurements).

The accelerations are maximised (up to 280×) for high resolution datasets with a large number of diffusion measurements and few model parameters, and are lower, yet substantial (in the range of 155× to 167×), for datasets with a small number of diffusion measurements. The angular resolution seems to affect more the performance gains than the spatial resolution. Since the fitting process of a voxel is assigned to a warp, a few thousand voxels will be enough to achieve the maximum occupancy of a GPU at this level of parallelisation, which is true for any typical dMRI dataset. However, a dataset with a few tens of diffusion measurements, which is common in clinical dMRI datasets (because acquisition time restrictions), will not take all the advantages of the second level of parallelisation in our design. The more

measurements in the data, the more expensive the computation of the likelihood, and the more beneficial is the second level of parallelisation.

Table 4.3 reports the acceleration achieved by the best parallel design of MCMC routine (GPU v3).

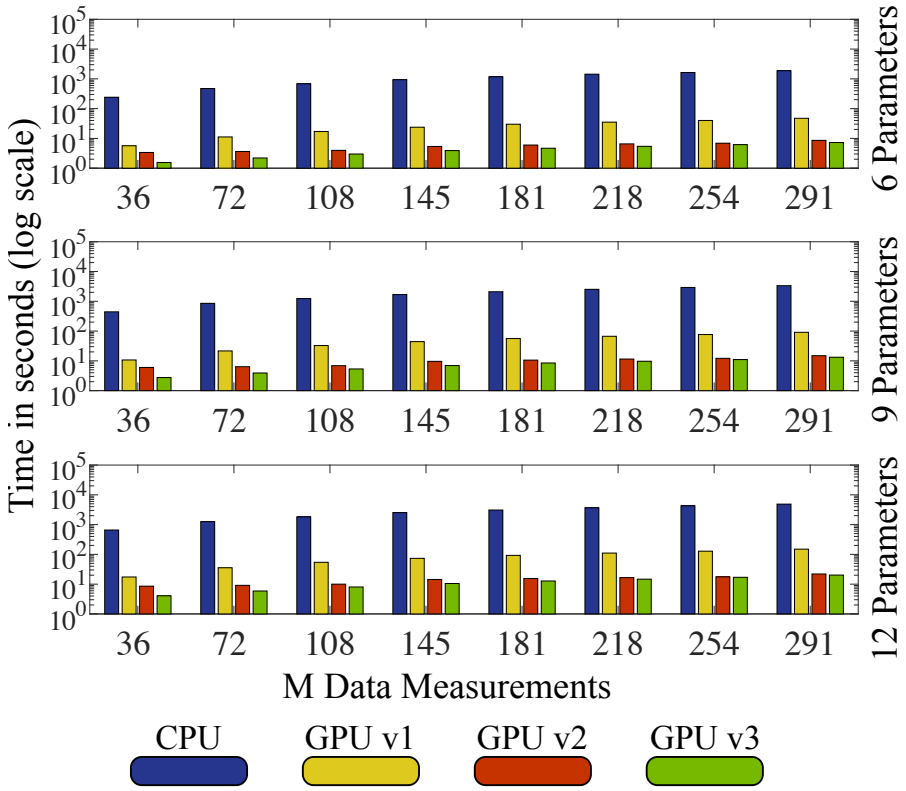

Figure 4.12: Execution times in logarithmic scale of a single CPU core and a single GPU running the MCMC routine and processing a slice with 3,088 voxels from a low resolution dataset. The CPU runs a sequential version of the routine (blue) whereas the GPU runs three different parallel versions (yellow, red and green). Datasets with different number of diffusion-weighted measurements (36/72/108/145/181/218/254/291) were computed varying the number of estimated parameters (6/9/12).

Measurements	36	72	108	145	181	218	254	291
Low-res-6 parameters	155×	214×	228×	240×	250×	264×	265×	257×
Low-res-9 parameters	161×	218×	231×	242×	247×	260×	264×	253×
Low-res-12 parameters	161×	213×	226×	240×	241×	249×	251×	241×
High-res-6 parameters	159×	216×	237×	249×	257×	269×	**280**×	266×
High-res-9 parameters	167×	220×	233×	244×	248×	263×	268×	264×
High-res-12 parameters	165×	220×	232×	246×	242×	249×	255×	244×

Table 4.3: Accelerations achieved by the best parallel design of MCMC routine, GPU v3, on a single GPU, compared to the sequential version executed on a single CPU core.

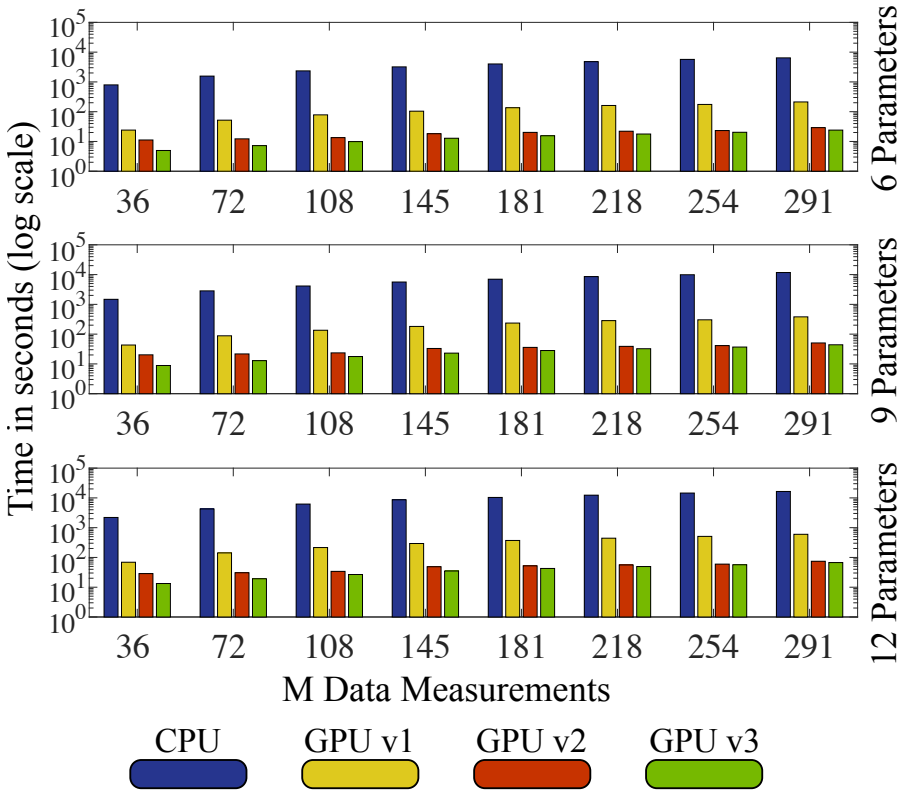

Figure 4.13: As Figure 4.12 but processing a slice with 10,522 voxels from a high resolution dataset.

4.4.2 Levenberg routine evaluation

Similarly, Figure 4.14 and Figure 4.15 show the execution times of the Levenberg routine processing a low and a high resolution dataset respectively. The GPU versions reduce the execution time considerably, offering accelerations of more than two orders of magnitude.

Version 1 improves upon the sequential version by a factor of 60 on average (in the range of 20× to 110×), while version 2 improves upon version 1 by a factor of 3.25 on average (in the range of 0.6× to 11×). Version 2 is slower than 1 only when processing a low resolution dataset with 36 diffusion measurements. In all other cases, version 2 obtained a better performance (speedup factor $> 1.0\times$).

As in the MCMC routine, the angular resolution affects more the performance gains than the spatial resolution or the number of model parameters.

The accelerations are maximised (up to 261×) for datasets with many voxels, and many diffusion measurements. Table 4.4 reports the acceleration achieved by the best parallel design of Levenberg routine (GPU v2).

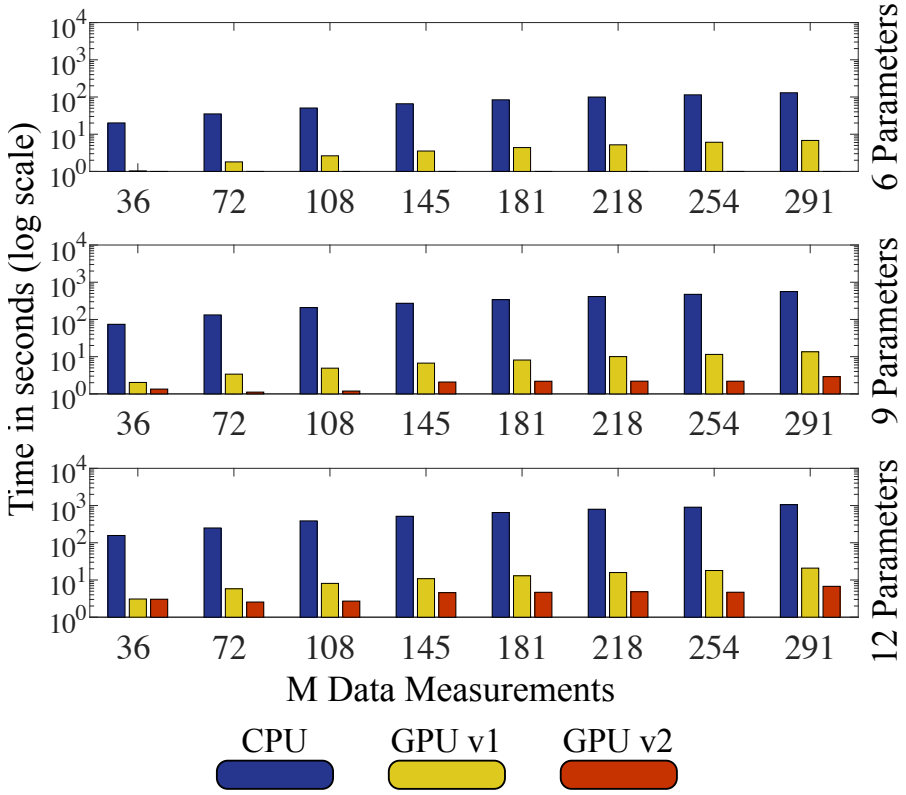

Figure 4.14: Execution times in logarithmic scale of a single CPU core and a single GPU running the Levenberg routine and processing a slice with 3,088 voxels from a low resolution dataset. The CPU runs a sequential version of the routine (blue) whereas the GPU runs two different parallel versions (yellow and red). Datasets with different number of diffusion-weighted measurements (36/72/108/145/181/218/254/291) were computed varying the number of estimated parameters (6/9/12).

Measurements	**36**	**72**	**108**	**145**	**181**	**218**	**254**	**291**
Low-res-6 parameters	41×	108×	110×	122×	148×	177×	205×	139×
Low-res-9 parameters	55×	118×	174×	130×	155×	187×	215×	193×
Low-res-12 parameters	51×	97×	142×	112×	138×	165×	192×	156×
High-res-6 parameters	63×	114×	197×	141×	177×	206×	241×	203×
High-res-9 parameters	68×	136×	192×	154×	191×	226×	**261**×	216×
High-res-12 parameters	56×	106×	159×	123×	149×	177×	204×	166×

Table 4.4: Accelerations achieved by the best parallel design of Levenberg routine, GPU v2, on a single GPU, compared to the sequential version executed on a single CPU core.

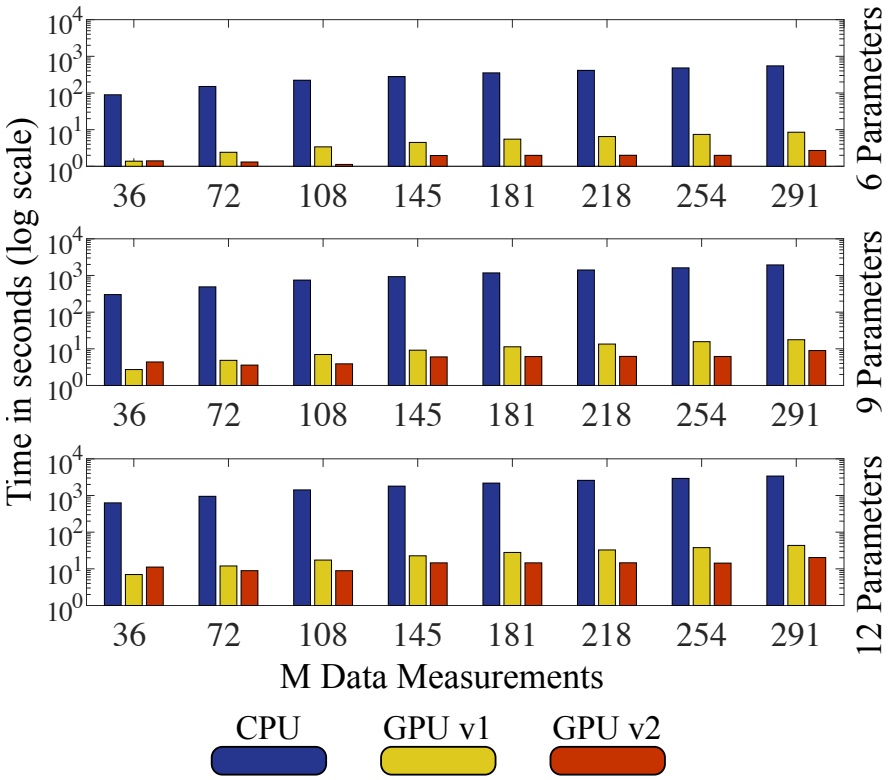

Figure 4.15: As Figure 4.14 but processing a slice with 10,522 voxels from a high resolution dataset.

4.4.3 Overall evaluation

Figure 4.16 shows the execution times of the whole fitting process processing a low and a high resolution dataset. The best parallel GPU version is compared to the sequential CPU version. Accelerations in the range of 92× and 238× are achieved by the parallel GPU version (see Table 4.5). As shown before, the highest accelerations are obtained processing datasets with a large number of diffusion measurements, as these allow to exploit better the within-voxel parallelism in the designs of both algorithms.

Figure 4.17 shows the execution time of the same scenario as before but dividing the dataset into 48 parts and processing each one on a different CPU thread, while using a multi-GPU version that divides the dataset into 4 parts and processes each one on a different GPU. Table 4.6 shows the accelerations achieved by the multi-GPU version compared to the CPU multi-core version. The multi-GPU system

achieves accelerations in the range of 17× and 42× compared to the multi-core system. The application shows steady scalability.

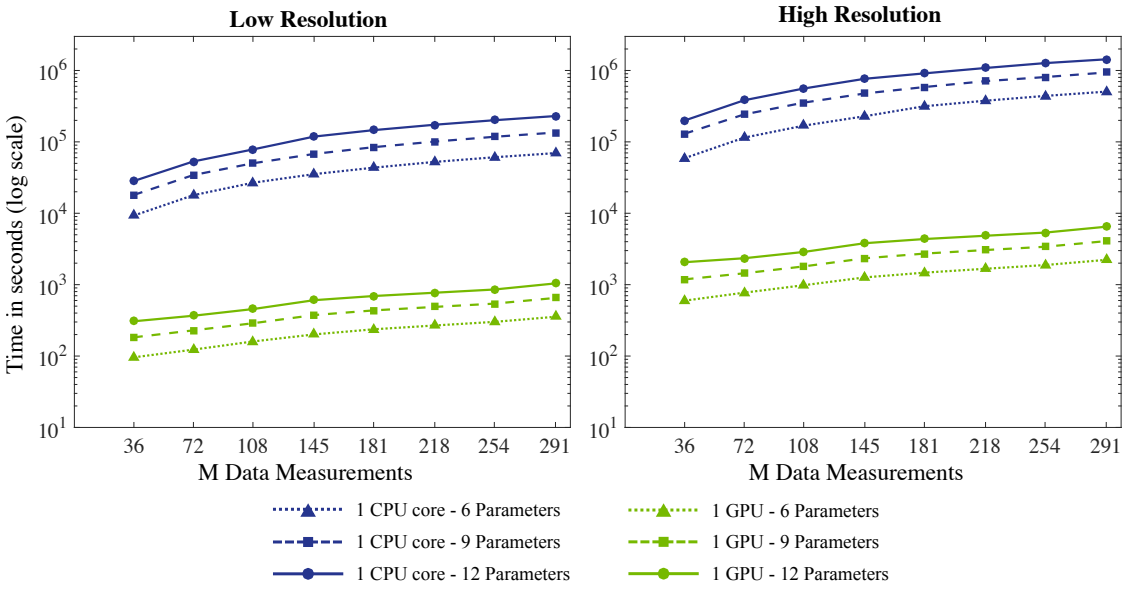

Figure 4.16: Execution times in logarithmic scale of a single CPU core and a single GPU fitting a dMRI model in a whole dataset. The application fits the model in a low resolution dataset with 106,910 voxels and in a high resolution dataset with 684,203 voxels. The CPU runs a sequential version of the routine (blue) whereas the GPU runs a parallel version (green). Different number of diffusion measurements (36/72/108/145/181/218/254/291) and number of model parameters (6/9/12) were tested.

Measurements	36	72	108	145	181	218	254	291
Low-res-6 parameters	96×	145×	167×	175×	183×	195×	202×	196×
Low-res-9 parameters	98×	149×	174×	180×	193×	206×	217×	207×
Low-res-12 parameters	92×	144×	172×	195×	210×	225×	235×	220×
High-res-6 parameters	99×	148×	171×	180×	214×	225×	233×	227×
High-res-9 parameters	109×	168×	196×	203×	215×	232×	**238**×	230×
High-res-12 parameters	96×	163×	194×	201×	209×	223×	236×	221×

Table 4.5: Accelerations obtained by the best parallel designs (MCMC GPU v3 and Levenberg GPU v2) running the whole application for fitting a dMRI model on a single GPU compared to a sequential version executed on a single CPU core.

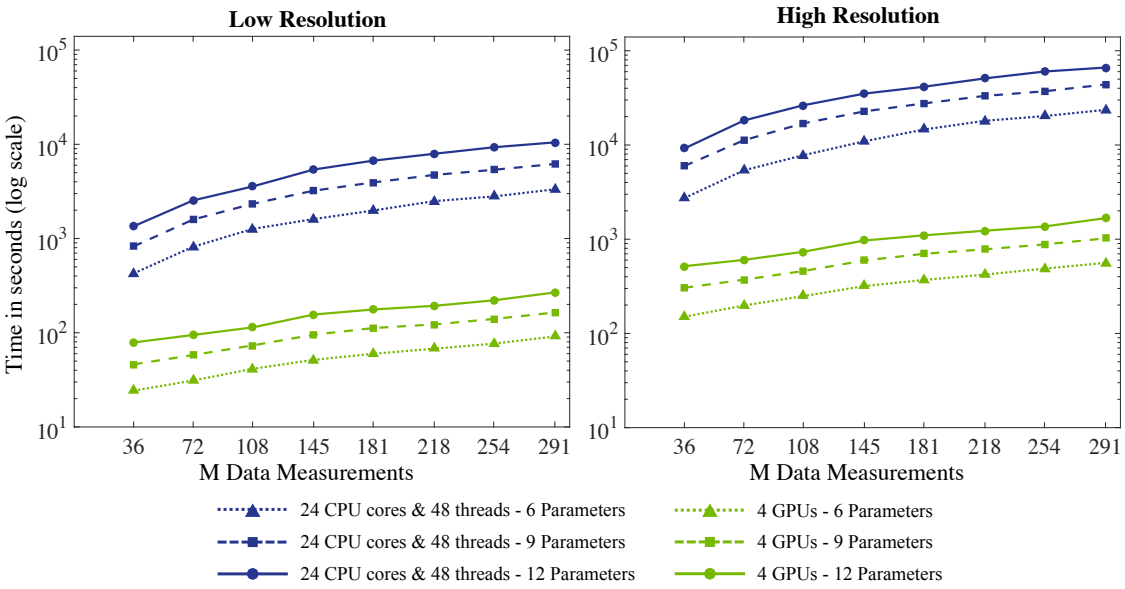

Figure 4.17: As Figure 4.16 but dividing the datasets into 48 parts and using 24 CPU cores and 48 threads (blue), and dividing the datasets into 4 parts and using 4 GPUs (green).

Measurements	36	72	108	145	181	218	254	291
Low-res-6 parameters	17×	26×	30×	31×	33×	36×	36×	36×
Low-res-9 parameters	17×	27×	31×	33×	35×	38×	38×	37×
Low-res-12 parameters	18×	26×	31×	34×	37×	41×	42×	39×
High-res-6 parameters	18×	27×	31×	34×	39×	42×	41×	42×
High-res-9 parameters	19×	30×	36×	38×	39×	42×	42×	42×
High-res-12 parameters	17×	30×	35×	36×	37×	41×	42×	39×

Table 4.6: Accelerations obtained by the best parallel designs (MCMC GPU v3 and Levenberg GPU v2) fitting a dMRI model on 4 GPU compared with a version using 24 CPU cores and 48 threads.

Finally, we made a price-performance comparison between the multi-CPU and multi-GPU configurations, i.e. assessed the relative performance gains of the GPU designs per unit cost. Table 4.1 and Table 4.2 show the prices of the CPU processors and the GPUs used for the tests. The two NVIDIA GPU accelerators have a total cost of £9,920 and the two Intel processors have a total cost of £3,520. The GPU accelerators cost 2.8 times as much as the CPU processors, but the average speedup factor processing the low resolution dataset is 32.5×, and processing the high resolution dataset 35.5×. Thus, in terms of price-performance ratio, the multi-GPU configuration is 11.6 and 12.6 times better than the CPU multi-core system. If we consider the power consumption, this gain gets even bigger.

4.5 Validation: Comparison to CPU implementation

We performed various tests in order to validate the GPU implementation against the sequential code. These comparisons are not straightforward because of the stochastic nature of the MCMC, which induces run-rerun variability.

We performed a first comparison test where we artificially introduced the same "randomness" in the fitting process. Specifically, we removed run-rerun variability by pre-generating and fixing a large set of random numbers, which were used for both the CPU and GPU versions. We then ran the estimation using both versions and choose the exact same random numbers in the stochastic components of the algorithm. We found differences only due to precision, caused by the different rounding modes between CPU and GPU [169].

In a second test, we allowed the stochastic components to operate as normal. Even in the case of having run-rerun variability in the MCMC, we expect that the distributions of parameter estimates across many runs converge, and we also expect to obtain similar results from CPU and GPU implementations. Therefore, we ran the GPU and CPU versions 1,000 times using the same data and obtained the distributions of estimated parameter values. Figure 4.18 shows representative examples from a number of voxels in the brain: in the corpus callosum, centrum semiovale and grey matter. For each case, two different fibres were estimated using the ball & sticks model, and the mean of the respective posterior distribution for every model parameter was recorded across the 1,000 runs. A distribution of these mean values across the 1,000 repeats was then compared between the CPU and GPU designs. Figure 4.18 shows the results for several model parameters: the baseline signal S_0, the diffusivity d, the volume fraction f_1 , f_2 of the first and second fibre and the angles that define the orientation of the first fibre th_1, ph_1.

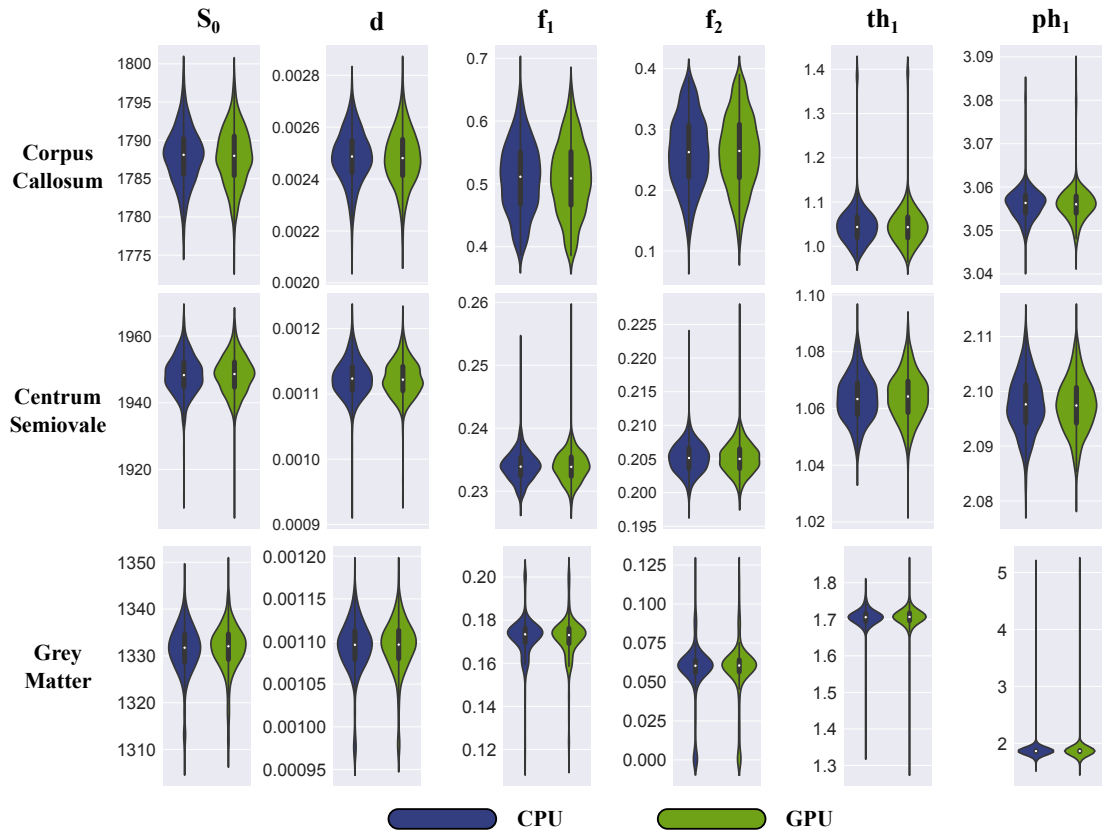

Figure 4.18: Comparison between CPU and GPU implementations of estimates fitting the ball & sticks model, including the baseline signal S_0, the diffusivity d, the volume fraction f_1, f_2 of the first and second fibre, and the angles that define the orientation of the first fibre th_1, ph_1, in different brain areas: a corpus callosum voxel, a centrum semiovale voxel and a grey matter voxel. Each implementation was ran 1,000 times on the same data and for each repeat the mean of the posterior distribution of the estimated parameters was recorded. A distribution of these means across all 1,000 repeats is shown. For each repeat, a burn-in period of 1,000 iterations and a thinning period of 25 samples was used in the MCMC. For obtaining the mean of th_1, ph_1 in each repeat, we calculate the mean dyadic vector from the angle samples, and we convert it from Cartesian to spherical coordinates.

The width of the distributions of the MCMC estimates was also compared. Figure 4.19 shows a distribution of the standard deviations, comparing the CPU with the GPU implementation, across the 1,000 repeats for the model parameters S_0, d, f_1, f_2 and the uncertainty of the first fibre orientation.

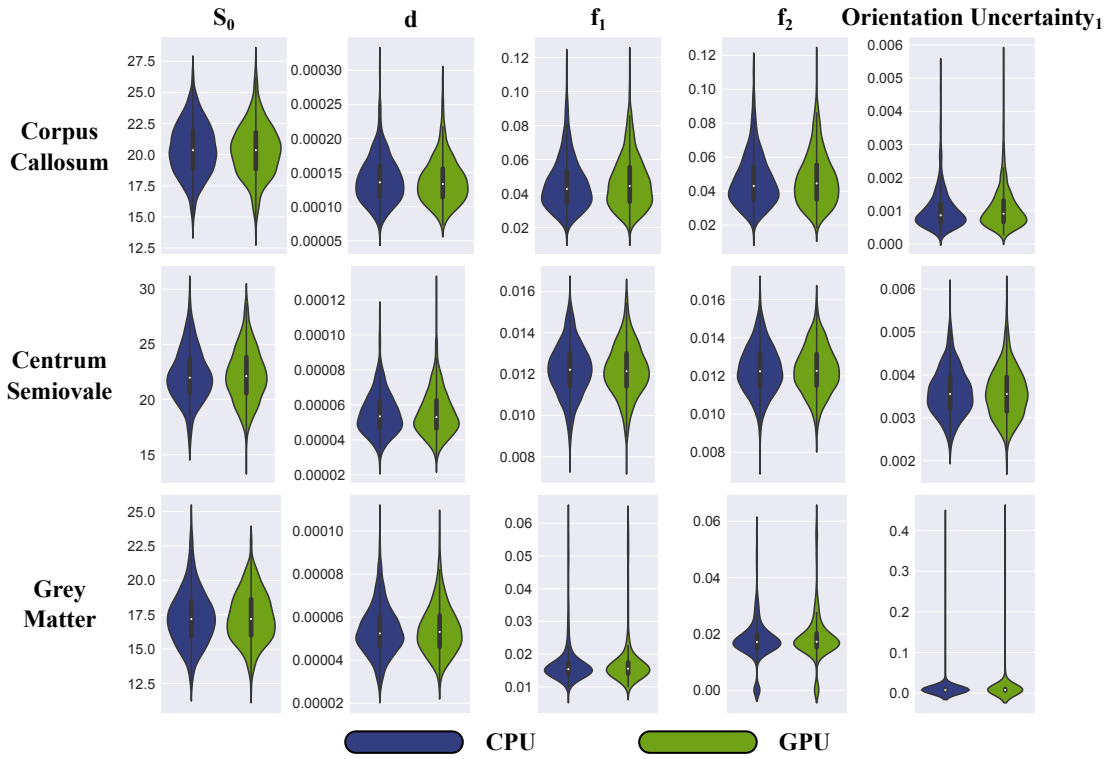

Figure 4.19: Distribution of the standard deviations of the recorded samples across 1,000 repeats, fitting the ball & sticks model in CPU and GPU implementations, for the baseline signal S_0, the diffusivity d, and the volume fraction of the first and second fibres f_1, f_2. The last column shows the distribution of the uncertainty when estimating the first fibre orientation across 1,000 repeats, which was obtained in each repeat from the dispersion of the mean dyadic vector. Voxels from different brain areas are shown: a corpus callosum voxel, a centrum semiovale voxel and a grey matter voxel.

We can see that the GPU implementation returns on average the same parameter estimates, and with the same uncertainty, as the CPU version. Figure 4.20 shows a map of the estimated first fibre orientation in each voxel for both, the CPU and the GPU implementations, and Figure 4.21 shows their uncertainties. These implementations return almost identical estimates.

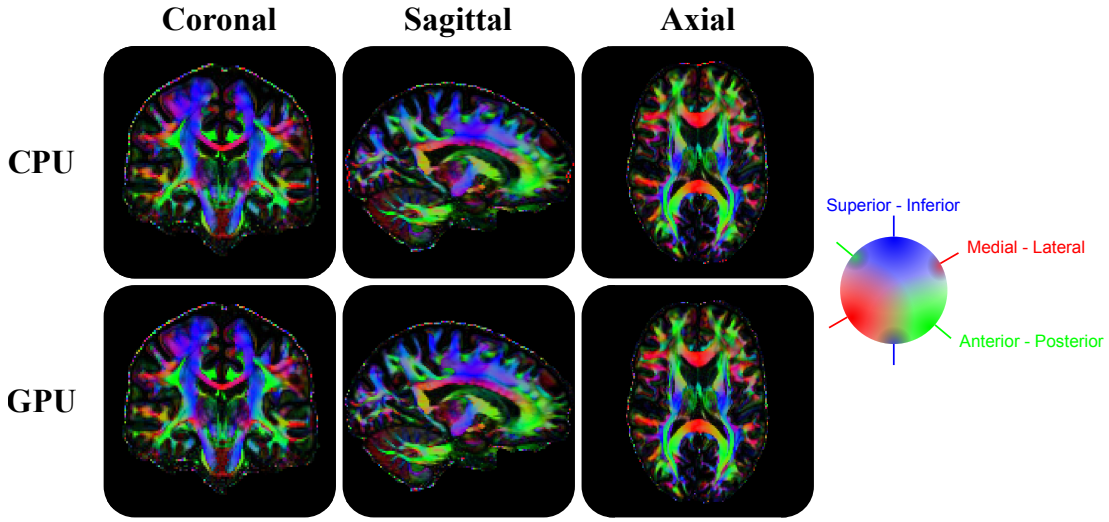

Figure 4.20: Coronal, sagittal and axial views of a map with the estimated mean principal diffusion direction in each voxel. Estimations are obtained using CPU and GPU implementations. The images are color-coded by orientation: medial-lateral corresponds to red, superior-inferior to blue and anterior-posterior to green.

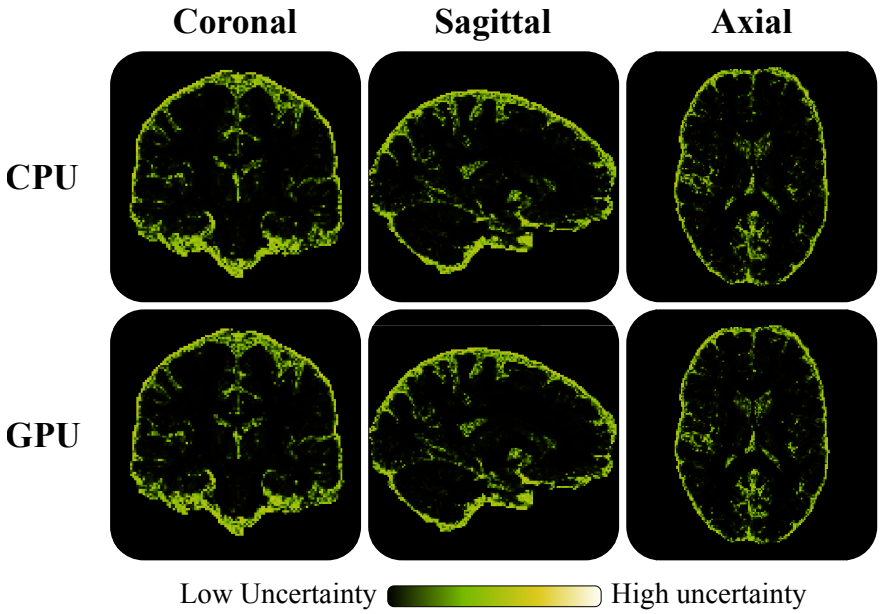

Figure 4.21: Coronal, sagittal and axial views of a map with the uncertainty of the estimated principal diffusion direction in each voxel. Uncertainties are showed for the estimation using CPU and GPU implementations.

4.6 Discussion

The analysis of dMRI data can require long computational times when non-linear models are used to characterise tissue microstructure or complex fibre configurations in the brain. We have analysed the fitting process of the ball & sticks model [115, 116, 157], a commonly-used dMRI model that estimates the main fibre orientations in each voxel. We use the fitting process used in Bedpostx, a tool included in FMRIB's Software Library (FSL) [38, 116] which performs Bayesian inference on the model parameters (using random-walk Metropolis MCMC), initialised by a non-linear optimisation routine (Levenberg-Marquardt). This process is applied independently to thousands of voxels and, as we saw, is well suited for a GPU implementation. It can be divided into thousands of independent tasks.

We showed the potential of using GPUs for scientific computing, for the first time, within the context of diffusion MRI biophysical modelling. We have proposed three parallel designs for accelerating both the MCMC and the Levenberg routines.

The first proposed design distributes the fitting process of different voxels amongst CUDA threads. This straightforward GPU design parallelises the application according to the way data is acquired and organised (i.e. voxel-wise). The design has also been proposed by other authors [170, 171] for fitting non-linear dMRI models, achieving accelerations up to 39× comparing a single GPU to a single CPU core. We achieved slightly higher accelerations (up to 44×), when we applied this design.

We showed the limitations of this design, which constrain the performance of the GPU implementation, and we have proposed two more designs where the fitting process of each voxel is assigned to a group of threads, instead of only one thread. A second level of parallelisation is used for distributing the most computationally demanding tasks of the MCMC and the Levenberg routines amongst the threads of a group. The light-weight threads, the coalesced global memory accesses and the high occupancy in the SMs that is accomplished, make these designs achieve accelerations up to 238× on a single GPU compared to a single CPU core. Despite

the fact that this design is limited by the GPU global memory bandwidth, and by divergences in the kernels, it can achieve roughly half an order of magnitude better performance than the previous design.

Our GPU design shows very steady scalability in terms of number of model parameters and spatial resolution. The speedup is lower, yet substantial (near 100x), when a dataset has few diffusion measurements, because the within-voxel parallelism is not completely exploited. The application shows the same scalability trend when a multi-GPU system is used. We have shown that for this application, a multi-GPU system is more efficient than a CPU multi-core system in terms of price-performance ratio (12.6 times more efficient).

The GPU solution has been designed for processing datasets with any number of voxels and measurements. If the memory required for storing the data of all the voxels exceeds the device memory, the framework divides the data into several subsets (according to the amount of device memory available), and these subsets are processed one after the other. However, before being processed, a subset needs to be copied into the device memory. This can create a performance penalty if the device memory capacity is small (1 or 2 GB) because a large number of CPU to GPU transfers is required, and these transfers are expensive (using normally the PCIe interconnection bus). This is however not a problem in the new NVIDIA architectures, where global memory space is larger (up to 24 GB in Pascal architecture) and a new CPU-GPU interconnection bus is incorporated (NVLINK).

The designs shown in this thesis have been optimised for a specific type of GPUs, NVIDIA Tesla K80. Although the framework works for any NVIDIA GPU (with double precision capability), the optimal number of warps per voxel (W), and voxels per block (B) may be different for other types of GPUs. Furthermore, the amount of measurements in the dataset can affect the optimal configuration. Figure 4.7 shows a comparison of the configurations of W and B for the NVIDIA Tesla K80, and for different number of data measurements. We have tried the same experiment on other NVIDIA Kepler GPUs (Tesla K40 and Tesla K20) obtaining

similar results (results not shown). The framework may have a different optimal configuration for other NVIDIA architectures, such as Pascal. Options for auto-tuning and auto-compilation [87] could be integrated with the framework, but the evaluation of these is left for future work. Nevertheless, accelerations of more than two orders of magnitude have been reported by many other users of our toolboxes using different NVIDIA architectures.

We validated the parallel implementations in various ways by comparing the results obtained from the GPU with the results obtained from the sequential version executed on a CPU. We verified that the GPU and the CPU designs gave almost identical estimates.

The parallel GPU implementation has been used to develop a software tool for estimating the fibre orientations on GPUs [144], which has been included in the FMRIB's Software Library (FSL) [38].

The designs proposed here are not particularly tied to CUDA. They could be implemented with OpenCL [62]. They are not tied to GPUs either, and its main components could be implemented using different parallel platforms. Nevertheless, since the designs assume a large amount of threads, they are more suitable for many-cores architectures, such as GPUs. The second level of parallelism added to our designs is specific to GPU solutions, where a relative small group of threads collaborate performing simple tasks.

4.6.1 Extension of the designs to other models

The GPU-accelerated application discussed previously implements the ball & sticks model, however, the parallel designs do not assume any specific model. They can be applied for implementing any other non-linear voxel-wise model on GPUs. In fact, we have implemented several diffusion models following the same approach:

- An extension of the ball & sticks model [157] that uses a Gamma distribution for modelling the diffusivity d. This model improves considerably the estimation when applied to multi-shell diffusion data.
- Ball and zeppelins [117, 172]. This model replaces the sticks with cylindrically-symmetric diffusion tensors. The GPU tool was especially useful for developing and validating this model, which was not available before in the FSL library. The accelerations offered by the GPU tool allowed testing this model extensively with large datasets, such as the HCP data.

In order to implement these two models, several modifications were needed in the CUDA code implementation. First and foremost, we had to modify:

– The model predicted signal function needed for calculating the residuals.
– The partial derivative functions, needed in the Levenberg algorithm.

But we also had to modify the number of parameters stored in Shared memory, the arguments passed to the kernels, many data pointers, and some other minor implementation details that restricts the flexibility of these implementations to be adapted to other models. In the next chapter we propose a generic toolbox that automatically deals with all these implementation details and offers a flexible user interface for specifying a model.

- A special case is the Rubix model [172, 173] where two datasets with different spatial resolutions (low and high resolution) are used during the fitting process. Each voxel of the low resolution dataset is fused with several voxels of the high resolution dataset, and a non-linear multi-voxel model is fitted. We adapted the parallel design Version 2 for this purpose (see Figure 4.22). We still assign the fitting process of each low resolution voxel to a block of threads, but now, the block of threads will iterate over the high resolution voxels that are fused with it, fitting them one by one. The threads of a block collaborate in the most computationally demanding tasks for fitting all the voxels, both low and high resolution, as in the parallel design Version 2.

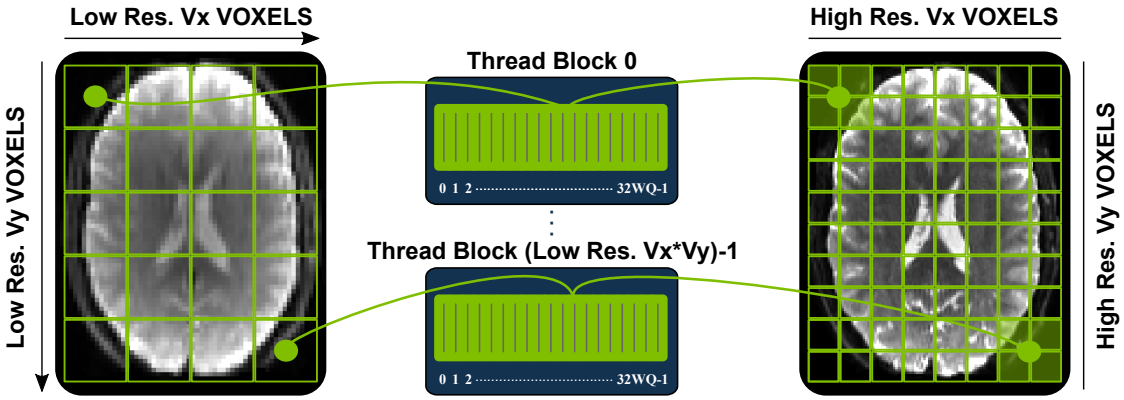

Figure 4.22: Adjustment of the parallel design Version 2 for fitting the Rubix model. The fitting process of each low resolution voxel, and all the high resolution voxels fused to it, is assigned to a block with Q threads.

4.6.2 Clinical and Big Data applications

The designs proposed here achieve massive accelerations in applications that extract tissue microstructural information from dMRI, such as the fibre orientations. These accelerations can be tremendously beneficial. Two direct advantages of using the parallel implementations are:

- The response time for analysing a single dataset is dramatically reduced from several hours/days to few minutes using a single GPU. Close to real-time processing could make these methods more appealing for clinical practice.
- Big databases arise more and more often from large consortiums and cornerstone projects worldwide. Hundreds or even thousands of datasets need to be processed. The throughput of the parallel designs using a single or a multi-GPU system is higher than a CPU multi-core system. Very large recent studies such as the Human Connectome Project (HCP) [2–4], (data from 1,200 adults), the Developing Human Connectome Project (dHCP) (data from 1,000 babies) and the UK Biobank [32, 33] project (data from 100,000 adults) are using (or have used) our designs for processing these datasets on GPU clusters. For instance, a 10-GPU cluster has been built for processing the most computationally expensive tasks of the UK Biobank pipeline. The cluster allows fitting the ball & sticks model to 415 datasets per day. Running

the same tasks with a cluster of 100 CPU cores, only 25 datasets can be processed per day. Moreover, to obtain a similar throughput as the 10-GPU cluster, more than 1,600 CPU cores would be necessary.

5

A generic GPU toolbox for model fitting and exploring brain microstructure using diffusion MRI

Contents

5.1 Introduction 114
5.2 Fitting microstructural diffusion MRI models 115
5.3 cuDIMOT: CUDA Diffusion Modelling Toolbox 117
5.3.1 General Design 117
5.3.2 Challenges: a generic toolbox 119
5.3.3 The user interface: a flexible toolbox 123
5.3.4 Implementation of the fitting routines 124
5.3.5 Performance and Validation 130
5.4 Exploring microstructure diffusion MRI models with cuDIMOT: Estimating fibre orientation dispersion . . 135
5.4.1 Modelling the dispersion of fibre orientations with a Watson distribution 136
5.4.2 Modelling the dispersion of fibre orientations with a Bingham distribution 138
5.4.3 The Ball & Rackets model 140
5.5 Comparing dispersion models and implementations . . 140
5.5.1 Comparing NODDI-Watson to AMICO 141
5.5.2 Comparing NODDI-Bingham implementations 146
5.5.3 Comparing NODDI-Bingham with ball & rackets 152
5.5.4 Crossing fibres and dispersion of fibre orientations . . . 155
5.5.5 Model Selection 158
5.6 Discussion 159
Appendix 5A: NODDI-Watson implementation 166
Appendix 5B: NODDI-Bingham implementation 168

Overview

Despite recent attempts to increase spatial resolution in diffusion MRI (e.g.[4, 174, 175]) voxel sizes remain relatively large, in the order of 1-10 mm^3 . As a result, the measured signal reflects the diffusion of water molecules within tissue sub-volumes that comprise tens of thousands of micro-components (e.g. axons, glia, cell bodies). Simple diffusion MRI (dMRI) models, such as the diffusion tensor model, represent average features of the tissue structure using single compartment models. They assume a single diffusion Gaussian process within a voxel level and ignore the complexity of the white matter microstructure. Several biophysical models have been proposed for improving the representation of this complexity, using more than one compartment in each voxel to represent different microstructural features [121, 123–125, 152–156].

These multi-compartment dMRI models aim to improve the specificity of the model parameters in explaining the diffusion process, and can gain new insight into the human brain microstructure. However, these are typically non-linear models, and therefore involve complex computational frameworks. Moreover, they typically require larger than average datasets (multiple *b-values* or high angular resolution), which can be very time consuming, for simply estimating model parameters, and more importantly, for driving model exploration and development.

In this chapter we propose a generic toolbox for designing, implementing and fitting MRI models[1] on GPUs. We extend the ideas presented in the previous chapter and provide a flexible front-end that allows users to define new models. The toolbox then automatically generates parallel CUDA code that can be executed on GPUs, and allows choosing between different optimisation routines for fitting the model to MRI data volumes. One of the key features is the provided flexibility when choosing the optimisation routines (deterministic or stochastic estimation,

[1]Notice that even if the presentation here is oriented around diffusion MRI models, the toolbox can be used for fitting any (voxel-wise) model on GPUs. Even if the developed front-end assumes MRI volumes for input/output of data and estimates, the principles are generic and could be extended to any different I/O handling routine.

a number of cost/likelihood functions, and prior distributions/constraints). We describe the challenges associated with such a generic design and provide the solutions. The toolbox offers accelerations of more than two orders of magnitude in the fitting processes of dMRI models.

Subsequently, we use the toolbox to explore diffusion models for characterising brain microstructure, specifically fibre orientation dispersion. We implement some recently published dispersion models, which are computationally expensive to estimate when using CPUs (on average a day of computations for a single subject), and we explore comparatively their performance under different scenarios. We generate Monte Carlo simulations, but also compare performance in real data using cross-validation. Finally, we propose some extensions of these models, including crossings of dispersing white matter compartments, and we perform model selection to identify regions in the brain where such a model would be beneficial.

Contributions of this chapter

- Development of a generic toolbox for fitting non-linear MRI models on GPUs: CUDA Diffusion Modelling Toolbox (cuDIMOT).
- Validation and performance tests of the toolbox are presented.
- Exploration of diffusion fibre orientation dispersion models. Implementation, comparison and model selection study using cuDIMOT.
- Extension of fibre orientation dispersion models for including crossing fibres configurations.

Publications

Contributions from this chapter have appeared in the following:

- **Hernandez-Fernandez M.**, Reguly I., Jbabdi S., Giles M., Smith S., Sotiropoulos S.N. "Using GPUs to accelerate computational diffusion MRI: From microstructure estimation to tractography and connectomes". In: *International Society for Magnetic Resonance in Medicine (ISMRM) 27th Annual Meeting*, Paris (France). (2018).
- **Hernandez-Fernandez M**, Reguly I., Giles M., Smith S., Sotiropoulos S.N. "cuDIMOT: A CUDA Toolbox for Modelling the Brain Tissue Microstructure from Diffusion MRI". *GPU Technology Conference (GTC)*, Munich (Germany). (2017).
- Foxley S., Jbabdi S., **Hernandez Fernandez M.**, Clare S., Scott C., Ansorge O., Miller K. "A comparison of multiple acquisition strategies to overcome B1 inhomogeneities in diffusion imaging of post-mortem human brain at 7T". In: *International Society for Magnetic Resonance in Medicine (ISMRM) 24rd Annual Meeting*, Singapore. (2016).
- Foxley S., Jbabdi S., **Hernandez Fernandez M.**, Clare S., Scott, C., Ansorge O., Miller K.: "Improved tract identification of post-mortem human brain with high-resolution DTI at 7T". In: *The Organization for Human Brain Mapping (OHBM)*, Honolulu (USA). (2015).

Software

- CUDA Diffusion modelling tool (cuDIMOT): http://users.fmrib.ox.ac.uk/~moisesf/cudimot

5.1 Introduction

The white matter is composed of several cellular components, including axons, myelin sheaths and neuroglial cells [176, 177]. Information about features of this tissue microstructure can be gained by modelling the diffusion process using biophysical parameters. However, the water diffusion process is different in each of these tissue components, and simplistic voxel-wise average models, such as the diffusion tensor, cannot characterise the underlying heterogeneity. Furthermore, the resolution of dMRI is very low (mm) compared to the cellular processes ($microns$). Thus, the group of axons giving rise to a signal in a voxel, may have complex organisation patterns that are not possible to represent with single compartment models [106, 126].

For these reasons, several approaches have been proposed where the diffusion signal attenuation is represented as a mixture of signals obtained from multiple compartments [121, 123–125, 152–156]. The use of phenomenological multi-compartment models allows in theory to differentiate between tissue structural components. Meaningful biophysical parameters that are *specific* to each compartment can be used for modelling the different diffusion processes and gain new insight into the brain microstructure. Along these lines, efforts have focused on inferring the axon diameter [121, 123], and characterising complex axons patterns such as crossing fibres [116, 157], and recently fanning fibres [124, 125, 178].

Nevertheless, the fitting process of these non-linear models is in general computationally expensive. Typically, larger datasets (higher angular resolution and/or multiple *b-values*) are required, and the models are applied to relatively large groups (tens to hundreds of subjects), which can make the estimation of tissue microstructure from dMRI a very time consuming process. In the previous chapter, we showed how the fitting process of one multi-compartment model, the ball & sticks [116], has been massively accelerated using GPUs. We developed a CUDA

toolbox [144] that achieves accelerations of two orders of magnitude comparing a GPU to a sequential CPU implementation.

However, this toolbox was specific to a dMRI model. Although the parallel design can be extended to other models, it is not straightforward from a user's perspective to develop such an extension and implement it on GPUs. In this chapter we propose a generic CUDA toolbox that provides a model-independent parallel implementation of non-linear optimisation routines for fitting voxel-wise MRI models on GPUs. The tool accepts as an input a model description, and it automatically generates CUDA code and binaries for the estimation of model parameters from MRI data in various ways (deterministic or stochastic estimation). This allows the user to be exempted from performing the parallelisation and CUDA coding, and to focus only on the model design. We perform validation tests of the toolbox and we measure its performance. Notice that even if the presentation here is oriented around dMRI models, the toolbox can be used for fitting any voxel-wise MRI model on GPUs.

Subsequently, we demonstrate the efficiency of the parallel toolbox using it for exploring and augmenting a number of microstructure models. Specifically, we focus on diffusion models that characterise fibre fanning and fibre orientation dispersion, implementing with the GPU toolbox some recently developed models. We perform a comparison study and finally we propose some extensions of these models.

5.2 Fitting microstructural diffusion MRI models

As we have already explained in the previous chapter, dMRI multi-compartment models are commonly non-linear functions of the signal, and Non-Linear Least Square (NLLS) algorithms or Bayesian inference methods can be used for the parameter estimation.

These model-fitting routines are iterative and computationally expensive. Moreover, the complexity of a dMRI model is proportional to the number of features that it attempts to infer, and typically, the more complex the model, the higher the computational cost. Some of the functions included in the models may not have an analytical solution, and numerical approximations are required, making even more expensive the process.

Furthermore, usually, these models are fitted to tens or even hundreds of diffusion measurements in hundreds of thousands of different voxels per subject. In large-scale projects, such as the Human Connectome Project (HCP) [2–4] or the UK Biobank [32, 33], thousands of subjects may be included. Therefore, the fitting process of dMRI models can be a very time consuming process, both at a single-subject level and also at a group level, as Figure 5.1 illustrates.

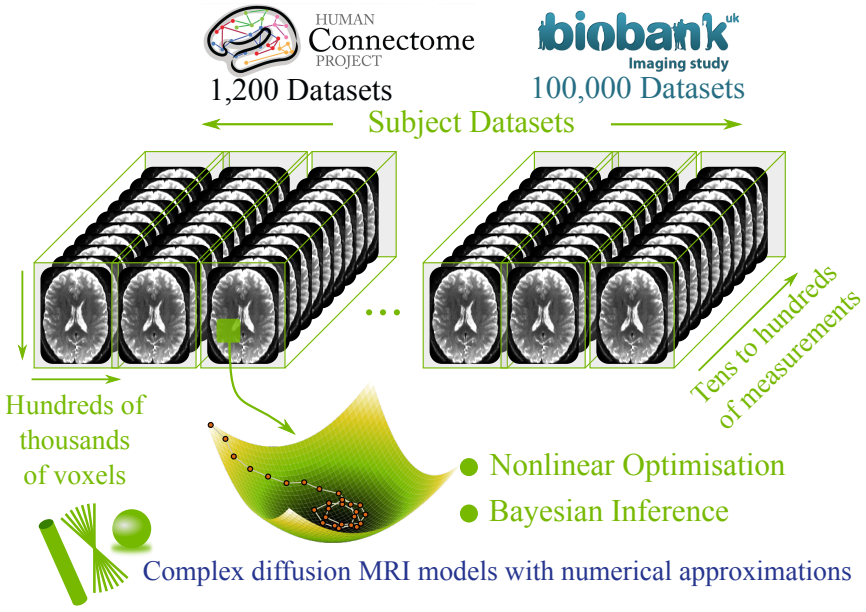

Figure 5.1: Depiction of the main factors that make the analysis of diffusion MRI data a time consuming process.

Here, we consider some typical routines used for model fitting, including:

- Greedy optimisation using Grid-Search.
- Non-linear optimisation using Levenberg-Marquardt (LM).
- Bayesian inference using Markov Chain Monte Carlo (MCMC).

These fitting routines form the backbone of our generic GPU modelling toolbox. LM and MCMC routines have already been described in Chapter 4. The Grid-Search is a simple greedy routine that in practice is mostly used for initialisation. A common problem in non-linear optimisation routines is the difficulty of finding the global optimal solution. If the search space is highly multidimensional and contains several local minima, the initialisation of the model parameters will have an important role for avoiding these local minima. The Grid-Search method defines a grid discretising the space of possible values of the model parameters. The method evaluates a cost function at each element of the grid, and selects the set of parameter values that provide the best solution (i.e. the minimum cost). Because of the nature of the greedy explorations, Grid-Search can be computationally very expensive, particularly for high dimensional problems. For instance, if we define a grid with 5 possible values per parameter, for a model with 5 parameters the method will need to evaluate 3,125 combinations, but if the model has 10 parameters, the number of combinations grows to 9,765,625. Fortunately, as in the case of LM and MCMC, the method is independent across voxels, and thus inherently parallelisable.

5.3 cuDIMOT: CUDA Diffusion Modelling Toolbox

5.3.1 General Design

In the previous chapter, we showed how GPUs can be used for accelerating dMRI model fitting. We have proposed some parallel designs that can be used for implementing non-linear optimisation and Bayesian routines on GPUs. However, the implementations were tied to a specific family of models, mirroring the implementation of the bedpostx tool in FSL [38, 116], and allowing maximum performance for these particular tasks.

Here, we propose a generic toolbox for model fitting using GPUs. The aim is to provide an interface for defining any model using C-like header files. The translation

to parallel CUDA code occurs automatically without the user needing to have any knowledge of parallel computing or GPUs. The CUDA Diffusion Modelling Toolbox (cuDIMOT) offers flexibility and control over both defining a model and choosing features of the fitting routine. Figure 5.2 depicts the toolbox design.

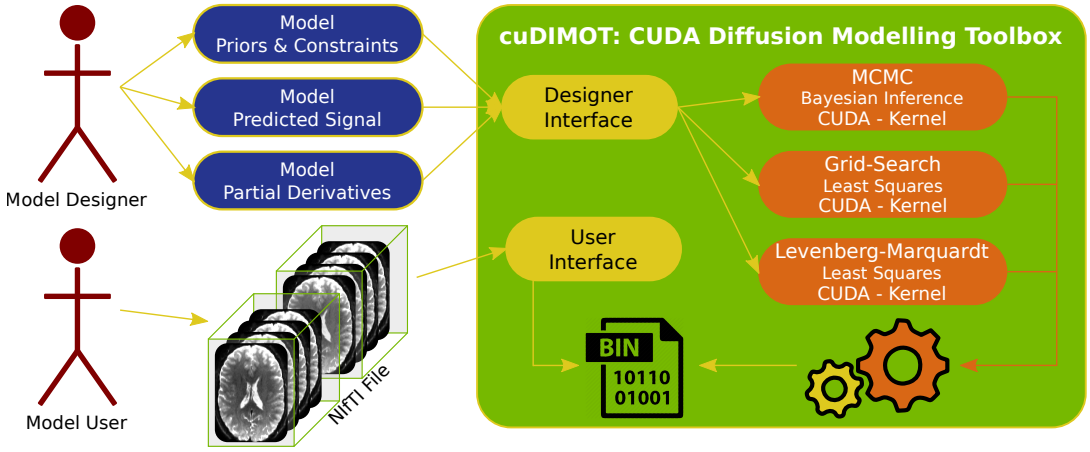

Figure 5.2: General design of CUDA Diffusion Modelling Toolbox (cuDIMOT). Two types of users interact with the toolbox through interfaces, a model designer and a model user. The model designer must provide the model specification, whereas the model user interacts with the toolbox for fitting the model to a dataset. The toolbox provides CUDA kernels that implement several fitting routines. These kernels are combined with the model specification at compilation time for generating a GPU executable application.

A model designer provides the specification of a model through a C-like interface, including information about the parameters and the cost function (and if applicable, its derivatives with respect to model parameters) that will be used in the fitting routines. The model designer can choose from a range of options for imposing prior knowledge for the parameters (bounded constraints in the deterministic case or distributions in the stochastic case, including shrinkage priors), and can also decide the range of values for Grid-Search. This specification is automatically processed, integrated into CUDA kernels that implement the fitting routines, and compiled for generating a GPU executable application.

Once the executable application has been generated, a model user can execute it through a different interface that provides some flexibility and offers different fitting options. cuDIMOT allows the user to choose from non-linear deterministic

estimation to stochastic Bayesian estimation (or a combination of them), and between a range of noise models for the MCMC case.

5.3.2 Challenges: a generic toolbox

In order to achieve an abstraction of the fitting routines, valid for any voxel-wise MRI model, the routines are implemented in a generic way excluding the functions that are model-dependent. Each fitting routine is implemented in a different CUDA kernel, which are executed by thousands of threads, and each thread processes different data (from different voxels or measurements).

The management of threads and data distribution are challenging aspects of a CUDA program, which we automate to avoid the model designer having to deal with. The toolbox implements the different kernels, deals with the arduous task of distributing the data among thousands of threads, uses the GPU memory spaces efficiently, and even distributes the computation among multi-GPUs if requested.

There are two facts to consider in the design of the toolbox:

- The specification of the model is known only at compilation time.
- The dimensions of the dataset to be processed are not known until execution time.

The specification of the model includes:

- The number of parameters of the model.
- The model predicted signal that gives rise to the cost/likelihood function.
- Its partial derivatives with respect to the parameters, used in optimisation routines that evaluate the gradient, such as the Levenberg-Marquardt algorithm.

In most of the cases, a kernel function performs the same process over each model parameter (for instance in the MCMC functions), thus, an iterative approach can be used in the implementation of the kernels. Memory needs to be allocated on

the GPU for these parameters and for certain data structures whose size depends on the number of parameters. This number is known at compilation time, and thus the number of iterations and the amount of memory to allocate for the parameters can be provided to the compiler.

The key aspect of the toolbox is the integration of the model predicted signal within the kernels. The kernels assume that there is a function $f(\mathbf{\Theta})$ already defined, that, given an array with the value of the model parameters $\mathbf{\Theta}$, returns a single value with the model predicted signal. This function needs to be defined at compilation time by the model designer.

For interacting with the toolbox using the designer interface, and define $f(\mathbf{\Theta})$, we decided to use external functions implemented outside the main routines. cuDIMOT is implemented using the CUDA C API, where the syntax for specifying the kernels and its functions is similar to C sequential functions. $f(\mathbf{\Theta})$ is part of the CUDA kernels (integrated at compilation time), and thus, it can be implemented as a C function.

The design of the toolbox follows the same approach for integrating the function that implements the partial derivatives. In this case, given an array with the value of the model parameters $\mathbf{\Theta}$, the function defined by the designer must return an array with several values, one per model parameter, which correspond to each partial derivative. Figure 5.3 shows an example of how a model is specified in the interface of the toolbox.

Once the specification of the model has been provided to the toolbox, a GPU executable file is generated. But the toolbox still needs to deal with the distribution of the MRI dataset among the GPU threads. The dimensions of the dataset are not known until execution time, and therefore this distribution needs to be programmed dynamically. When the kernels are executed, thousands of threads are created dynamically. Internally, each thread has an ID, which is used to calculate, on the

fly, the address of the data to process by each thread. cuDIMOT allocates the MRI dataset on the GPU memory at execution time.

Model Predicted Signal $f(\theta)=\theta_1*exp(-\theta_2*x)$ and Partial Derivatives

```
MACRO T Predicted_Signal (int npar, T* P,  T* CFP, T* FixP){
    return P[0]*exp(-P[1]*CFP[0]);
}
MACRO void Partial_Derivatives (int npar, T* P,  T* CFP, T* FixP, T* derivatives){
    derivatives[0]=exp(-P[1]*CFP[0]);
    derivatives[1]=-P[0]*CFP[0]*exp(-P[1]*CFP[0]);
}
```

Parameters Bounds and Priors

```
bounds[0] = (80,120)
bounds[1] = (,1.5)
prior[0] = Gaussian(100,10)
prior[1] = ARD(1)
```

Figure 5.3: Example of the specification of a simple model with 2 parameters in cuDIMOT. The model predicted signal returns a single value and the function that defines the partial derivatives store in an array the results for each parameter. The macro includes the necessary keywords for declaring a device function in CUDA, P contains the parameters $\mathbf{\Theta}$, and CFP and FixP contain the measurement points (CFP contains x in this case). Prior information and constraints can be imposed on the model parameters.

Bounds, priors, and constraints

Prior information or constraints on the parameters of a model can be integrated into the fitting process using the model designer interface (as in Figure 5.3), where a simple syntax is used for enumerating the type and the value of the priors. A C++ *parser* was implemented to read and store the list of these values, which are checked at runtime during the fitting routines.

The bounds can be defined using a lower and/or an upper limit. The priors can be any of the following types:

- Gaussian probability distribution. The mean and the standard deviation of the distribution must be specified.
- Gamma probability distribution. The shape and the scale of the distribution must be specified.

- *Automatic Relevance Determination* (ARD) prior [179]. A model designer may not be sure about the relevance of a parameter, or, this relevance may depend on the data. Using an ARD prior, if a certain feature of the model is not relevant, it will be forced to zero (i.e. shrinkage prior).
- Angle uniformly distributed on a sphere. This type of prior can be used for parameters that describe an orientation. If the surface of a sphere is used for representing the orientation probabilities, this prior ensures that every area of the sphere has the same prior probability.

Note that using bounds with both, lower and upper limits, the user defines a prior that follows a uniform probability distribution.

Occasionally, it is required to define constraints that depend on the relation of two or more parameters (for instance, $a < b$). These constraints can be specified using external functions and the C syntax, similar to the specification of the model predicted signal function.

I/O and measurement points

Diffusion-weighted MRI measurements are acquired using several imaging settings that define the measurement points. Some of them are present in any dMRI acquisition sequence, such as diffusion-sensitising gradient strengths and associated directions. However, other dMRI sequences may use some additional settings. For instance, diffusion-weighted steady-state free precession (DW-SSFP) [180] adds four more imaging parameters: flip angle (α), repetition time (TR), and longitudinal and transverse relaxation times ($T1$ and $T2$). These measurements points may be common to all voxels or not (CFP or common fixed parameters and FixP or fixed parameters in Figure 5.3 respectively).

In cuDIMOT, the number, type, and dimensions of the imaging settings that define these measurement points are specified at compilation time by the model designer. Using a simple syntax, a list with all the information is passed to the toolbox through the designer interface. This information is parsed by

cuDIMOT and used at execution time, when the model user must provide maps with these parameters.

This generic interface allows the users to combine data from dMRI with data from other modalities, such as relaxometry [181], and develop more complex models [182, 183], or, even use cuDIMOT in different modalities where non-linear optimisation is required.

Cascaded model fitting

High-dimensional models are difficult to fit. They may not find the optimal global solution unless an adequate parameter initialisation is used. Grid-Search can be used for initialising the model parameters, however, the method cannot test all the possible combinations of values and can be inefficient.

A common strategy for fitting complex models consists in estimating a simpler model first, with similar parameters, and afterwards, the estimated parameters can be used for initialising some parameters of the complex model. In cuDIMOT the fitting processes of different models can be cascaded, i.e., use the output of fitting a model to initialise the fitting of another model. 3D volumes can be used for specifying the initialisation value of the model parameters in every voxel.

5.3.3 The user interface: a flexible toolbox

After a model has been defined, a user does not need to deal with the details of the fitting routines implementation. Still, the toolbox allows them flexibility in controlling a number of options, including:

- Choosing fitting routines: Grid-Search, Levenberg-Marquardt or MCMC. A combination of them is possible, using the output of one method to initialise the next one.
- Selecting number of iterations in Levenberg-Marquardt.

- Selecting number of iterations in MCMC (burn-in, total, sample thinning interval).
- Use Gaussian or Rician noise modelling in MCMC.
- Choose parameters of the model to be kept fixed during the fitting process.

The toolbox can optionally generate interesting statistics maps for model selection, such as:

- *BIC*: Bayesian information criteria [184].
- *AIC*: Akaike information criteria [185].
- Return the model predicted signal.

A debugging mechanism is also implemented for reporting information during the execution of the fitting routines. When this mechanism is activated, cuDIMOT displays the most important information of the fitting routines in each iteration, such as the value of the model parameters.

5.3.4 Implementation of the fitting routines

Performance aspects

The implementation of the fitting routines in cuDIMOT follows the parallel design Version 3 proposed in the previous chapter for MCMC (Figure 4.6). Here, we use this design for implementing several routines, the MCMC, the LM and the Grid-Search. Specifically, the design divides each dataset into groups with a few voxels (eight in general and two in LM). The fitting process of each group is assigned to a CUDA block, and each warp of the block fits the model in one voxel. In a second level of parallelism, the threads within a warp collaborate for computing the most expensive steps of the fitting routines.

Despite the general ideas being the same, we have improved the design in various aspects; this is to counterbalance any performance losses due to the higher flexibility of the toolbox, compared with the more specific designs in the previous chapter.

A first improvement is the utilisation of *shuffle instructions*. The threads that cooperate for computing the fitting process in a voxel need to communicate between them frequently. In the previous CUDA implementation the Shared memory was used for communication, however, in cuDIMOT we have improved the design by using CUDA shuffle instructions.

CUDA shuffle instructions allow threads to read the registers of other threads in the same warp. This communication mechanism is faster than Shared memory, and it does not require synchronisation, as it is implicit within the threads of a warp. Figure 5.4 illustrates how shuffle instructions are used in cuDIMOT for calculating the sum of thread registers. Each thread sums one of its registers with the register of another thread. After running a few iterations and selecting the correct threads from where to read a register, all the threads will end with the sum of all the registers. This mechanism is very efficient for reduction operations, and we use it for computing the sum of the squared residuals in all the fitting routines (cost/likelihood functions).

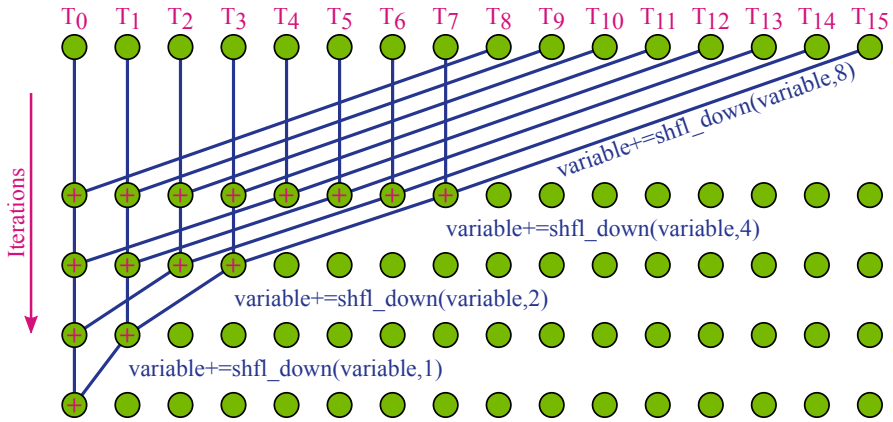

Figure 5.4: Example of the use of shuffle instructions for summing the values of the registers from $16T$ threads. The shuffle instruction receives as arguments the name of the variable to read and an offset for choosing the thread from where to read it. The figure shows how the thread T_0 gets the sum of all the registers after 4 iterations. Actually, all the 16 threads get the same sum value after the 4 iterations, because the operation is circular, i.e., T_{15} will read the values from threads T_7, T_3, T_1 and T_0.

Another implementation strategy to improve the performance of cuDIMOT is the use of Constant memory for storing the bounds and prior distributions information.

This is a not writable memory (excepts for initialisation) that resides in global memory (Level 3 in Figure 2.9), and thus accesses involve high latencies. But differently to global memory, accesses to this memory are cached in the Constant memory cache (Level 1 in Figure 2.9), so consecutive reads to the same data, like in our case, reduce the memory traffic [82]. We also use the Constant memory for storing the values to evaluate in the Grid-Search routine, i.e., the grid.

Levenberg-Marquardt implementation

The implementation of this routine includes some complex functions, such as the computation of the gradient, the Hessian matrix, the cost function, and a LU decomposition solver. The cost function calls an external function that, given the current value of the model parameters, must return the model predicted signal. The methods for computing the gradient and the Hessian matrix also call an external function that must return a list with the partial derivatives of the model parameters. In each voxel, these methods are parallelised using the threads within a warp. Each thread calls the external functions with different values as arguments for evaluating the function at a different data measurement. Shuffle instructions are used for transferring the partial derivatives to a main thread of the warp, for forming the final gradient vector and Hessian matrix.

We performed a test to identify the optimal number of voxels assigned to a block, as we did in Chapter 4 for MCMC (Figure 4.7). We set the number of warps per voxel $W = 1$, because the method uses shuffle instructions and they only work correctly on the level of warps. We found $B = 2$ to be optimal (fewer than in the MCMC case). The reason is that in the LM algorithm the number of iterations is not fixed, and the method stops when a convergence criterion is met. Thus, it may happen that warps belonging to the same block execute different number of iterations. Having fewer warps per block, but more blocks in a SM, the GPU has more flexibility for switching between blocks at execution time. However, the GPU can only reach 50% of the occupancy because of the limitation on the maximum number of blocks per SM (16 in Kepler microarchitecture).

LU decomposition: The LU decomposition is used for solving the system:

$$\mathbf{H}\boldsymbol{\Theta}_{new} = \boldsymbol{\Theta} - \mathbf{r} \tag{5.1}$$

and calculating the proposed parameters $\boldsymbol{\Theta}_{new}$ in every iteration of the Levenberg-Marquardt algorithm (see Algorithm 4.1), where $\mathbf{r}$ is the gradient and $\mathbf{H}$ the approximated Hessian matrix. Using LU decomposition in the matrix $\mathbf{H}$:

$$\mathbf{LU}\boldsymbol{\Theta}_{new} = \boldsymbol{\Theta} - \mathbf{r} \tag{5.2}$$

cuDIMOT solve this system using $P + 1$ threads, where P is the number of columns of $\mathbf{H}$ (and number of model parameters). Figure 5.5 depicts the parallelisation of the method, which uses Gaussian elimination with pivoting and solves the system as:

$$\mathbf{U}\boldsymbol{\Theta}_{new} = \mathbf{L}^{-1}(\boldsymbol{\Theta} - \mathbf{r}) \tag{5.3}$$

Firstly, P forward steps are made for calculating $\mathbf{U}$, producing ones at the diagonal of the matrix and zeros below the diagonal. At each step, a diagonal element is set as the pivot, and each thread computes the elements of a different column. Shuffle instructions are used for broadcasting the pivot. Secondly, P backward steps are made for producing zeros above the diagonal. Again, each thread computes the elements of a different column. The method produces the results $\boldsymbol{\Theta}_{new}$ at the rightmost column.

It is easy to see that the workload is not balanced across threads. They compute different number of elements at each step of the method. Also, normally the number of columns of the matrix is lower that the number of threads of a warp, and thus there are idle threads. Nevertheless, we show in the next section that this implementation is more efficient than solving the system with only one thread of the warp, as was done in the previous chapter.

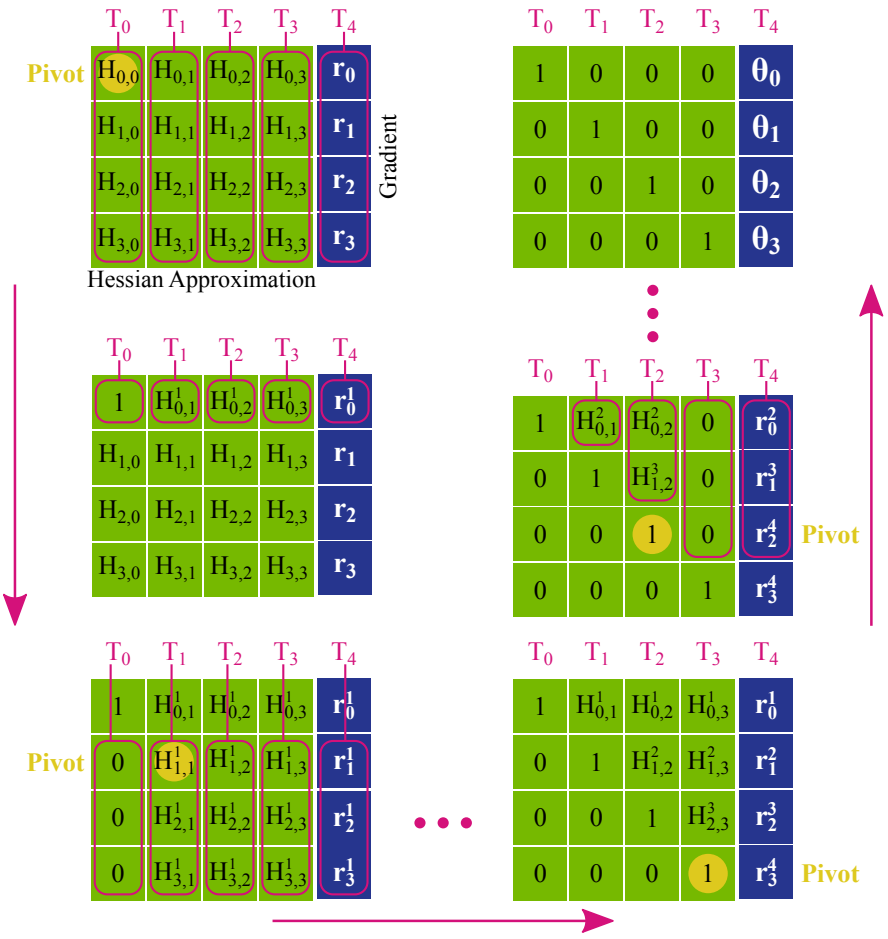

Figure 5.5: Parallelisation of the LU decomposition across the threads within a warp. The LU decomposition is computed using Gaussian elimination with pivoting. The method produces ones at the diagonal elements of the matrix and zeros above and below the diagonal. The results are produced at the rightmost column. Each thread T computes the elements of a column.

Functionality aspects

An additional feature added to the MCMC routine implementation is the option of using a Rician noise model [186], instead of a Gaussian noise model. The likelihood becomes:

$$Likelihood(\mathbf{\Theta}) = \prod_{m=1}^{M} \frac{Y_m}{\sigma^2} \exp\left(\frac{-(Y_m^2 + S_m(\mathbf{\Theta})^2)}{2\sigma^2}\right) I_0\left(\frac{S_m(\mathbf{\Theta})^* Y_m}{\sigma^2}\right) \tag{5.4}$$

where $S_m(\mathbf{\Theta})$ is the model predicted signal for the m^{th} measurement and a set of parameters $\mathbf{\Theta}$, Y_m is m^{th} measurement of the dataset, M is the number of

measurements, I_0 is the modified Bessel function of the first kind with order zero, and σ^2 is the variance of the noise distribution.

In this model, the squared residuals do not need to be calculated, but the model predicted signal and the Bessel function need to be computed for every measurement m. Thus, the parallel design remains the same, and the computation is distributed among the threads of a warp.

Finally, the BIC and AIC indices are implemented as:

$$BIC = \log(M)P - 2\log(Likelihood) \tag{5.5}$$

$$AIC = 2P - 2\log(Likelihood) \tag{5.6}$$

where P is the number of model parameters.

Numerical differentiation: It is possible that the partial derivatives of the model cannot be computed analytically, or they are too complex to be computed analytically. cuDIMOT offers the option of calculating automatically the derivative of a parameter Θ_p using numerical differentiation. Internally this is implemented as:

$$Derivative = \frac{S(\Theta_p + 0.5\ step\_size) - S(\Theta_p - 0.5\ step\_size)}{step\_size} \tag{5.7}$$

where $step\_size$ is very small ($1e^{-5} \times \Theta_p$). However, using numerical differentiation the method can take more time to converge.

Bounds and Parameter Transformations: In the LM, the parameter bounds are implemented using transformations. There are three types of reparametrisations applied, depending on the kind of bound.

- If a single lower limit is used:

$$\Theta_p = \exp(\Theta'_p) + lower_{bound} \tag{5.8}$$

- If a single upper limit is used:

$$\Theta_p = upper_{bound} - \exp(\Theta'_p) \tag{5.9}$$

- If both, lower and upper limits are used:

$$\Theta_p = \frac{upper_{bound} - lower_{bound}}{2} \arctan(\Theta'_p)\frac{2}{\pi} + \frac{lower_{bound} - upper_{bound}}{2} \tag{5.10}$$

where Θ_p is the original parameter and Θ'_p is the transformed one. Figure 5.6 shows the functions for these transformations.

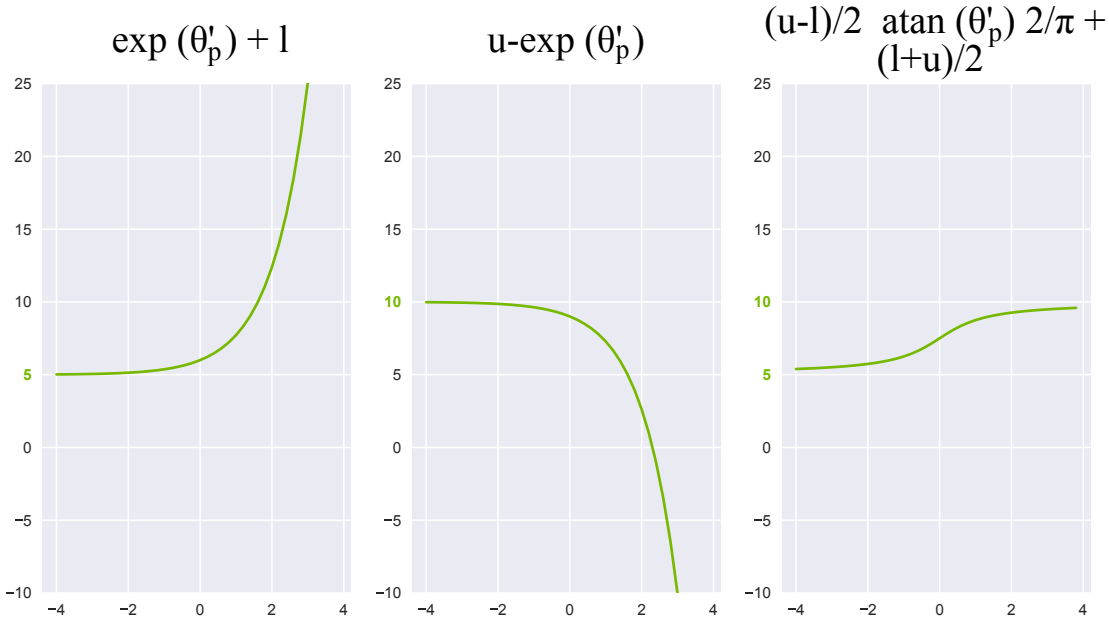

Figure 5.6: Functions of the transformations used in the Levenberg-Marquardt algorithm for imposing parameters bounds. The first column represents a bound with a lower limit $l = 5$, the middle column a bound with an upper limit $u = 10$ and the right column use a lower and an upper limit $l = 5$ & $u = 10$.

5.3.5 Performance and Validation

In this section we evaluate the performance of cuDIMOT when fitting the ball & sticks model. We compare it with the model-specific GPU tool developed in the previous chapter and with a CPU sequential tool. We also validate the toolbox comparing the results from the three different tools. For performing

these tests, we use the same datasets and the same hardware as in the previous chapter (described in Section 4.4).

Figure 5.7a and Figure 5.7b show the execution times of the Levenberg and the MCMC routines fitting a high-resolution slice. In all the cases cuDIMOT was faster than the model-specific GPU tool running Levenberg and slower running MCMC. The improvements in the implementation of the LM routine, including the parallelisation of the LU solver and the use of shuffle instructions, make this kernel more efficient. On average, running the Levenberg algorithm, cuDIMOT is 390 times faster than the sequential CPU implementation and 2.3 times faster than the model-specific GPU tool.

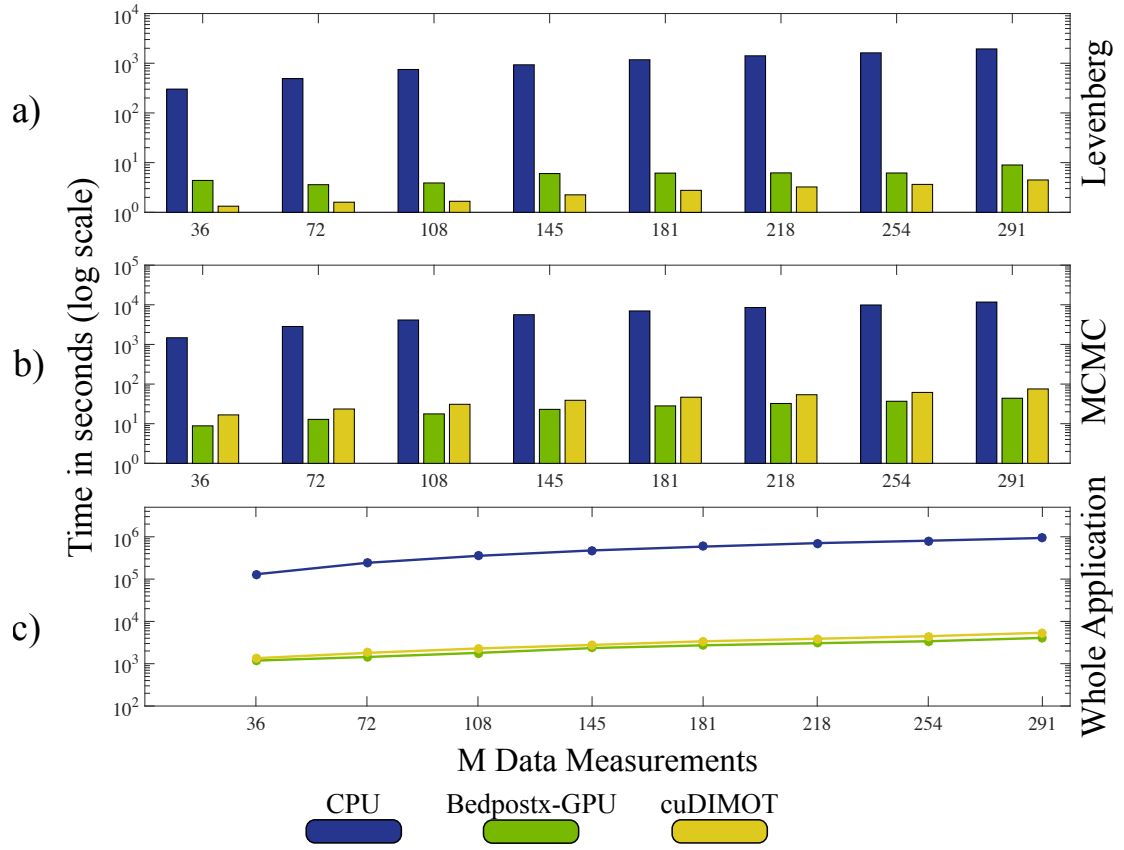

Figure 5.7: Execution times of three different tools fitting the ball & sticks model. Blue colour is used for reporting the execution times of a sequential CPU tool, green for a model-specific GPU tool and yellow for cuDIMOT. Times processing a high resolution slice and running the Levenberg algorithm are shown in a), and running MCMC in b). Execution times processing a high-resolution dataset and running the whole fitting application (Levenberg & MCMC) are reported in c). Datasets with different number of diffusion-weighted measurements (36/72/108/145/181/218/254/291) were computed.

On the other hand, the price for making MCMC generic to any model is the introduction of some extra book-keeping tasks at execution time, for instance, updating and checking priors and constraints for every model parameter, which adds a relatively small penalty to computation times. On average, running the MCMC algorithm, cuDIMOT is 139 times faster than the sequential CPU implementation and 1.73 times slower than the model-specific GPU tool.

Figure 5.7c shows the execution times of the whole model fitting application (Levenberg algorithm followed by MCMC) for the three tools processing a high-resolution dataset. On average, cuDIMOT was 158 times faster than the sequential CPU version and 1.25 times slower than the model-specific GPU tool. Table 5.1 reports the speedups obtained running Levenberg, MCMC, and the whole application.

M Data measurements	**36**	**72**	**108**	**145**	**181**	**218**	**254**	**291**
Levenberg vs.CPU	227×	306×	448×	413×	424×	436×	443×	433×
Levenberg vs.GPU-specific	3.29×	2.25×	2.33×	2.67×	2.22×	1.92×	1.69×	2.01×
MCMC vs.CPU	88×	120×	133×	144×	150×	158×	160×	155×
MCMC vs.GPU-specific	0.52×	0.54×	0.57×	0.59×	0.60×	0.60×	0.59×	0.58×
Total vs.CPU	96×	134×	154×	171×	173×	**182**×	180×	173×
Total vs.GPU-specific	0.87×	0.79×	0.78×	0.84×	0.80×	0.78×	0.76×	0.75×

Table 5.1: Speedups and slowdowns obtained by cuDIMOT compared with a CPU sequential version and with a model-specific GPU tool fitting the ball & sticks model. Speedups and slowdowns are reported for the Levenberg and the MCMC algorithms processing a high resolution slice, and for the whole model fitting application processing a high resolution dataset. Datasets with different number of diffusion-weighted measurements were computed.

In order to validate the toolbox, we perform similar tests as in the previous chapter (Figure 4.18 and Figure 4.19). We fit the ball & sticks model with cuDIMOT, using two sticks, and we compare the results with the results obtained using a sequential CPU version and a model-specific GPU tool. Figure 5.8 compares the distributions of the estimated parameters in three representative voxels after 1,000

runs. The results for the main model parameters are showed, including the baseline signal S_0, the diffusivity d, the volume fraction f_1 & f_2 of the first & second fibres, and the angles that define the orientation of the first fibre th_1, ph_1. The standard deviations of the MCMC samples were also recorded, and their distributions are shown in Figure 5.9 for the model parameters S_0, d, f_1, f_2. The figure also shows the distribution of uncertainties of the first fibre orientation. On average, the three implementations give the same results.

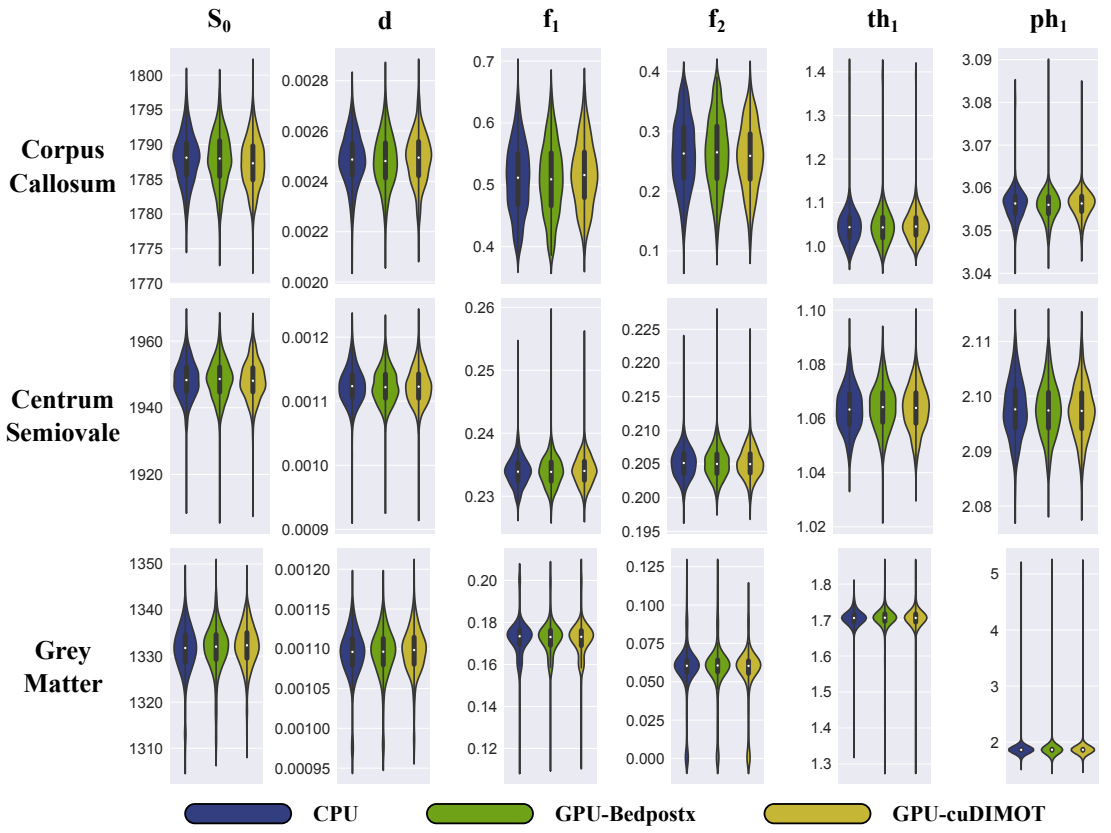

Figure 5.8: Comparison of the estimated parameters obtained from a sequential CPU version, a model-specific GPU tool (Bedpostx_gpu) and cuDIMOT fitting the ball & sticks model. Parameters shown are: the baseline signal S_0, the diffusivity d, the volume fraction f_1, f_2 of the first and second fibre and the angles that define the orientation of the first fibre th_1, ph_1. Results in three different brain areas are shown: a corpus callosum voxel, a centrum semiovale voxel and a grey matter voxel. Each design was ran 1,000 times on the same data and for each repeat the mean of the posterior distribution of the respective parameter was recorded. A distribution of these means across all 1,000 repeats is shown. For each repeat, a burn-in period of 1,000 iterations and a thinning period of 25 samples was used in the MCMC. For obtaining the mean of th_1, ph_1 in each repeat we calculate the mean dyadic vector from the angle samples and we convert it from Cartesian to spherical coordinates.

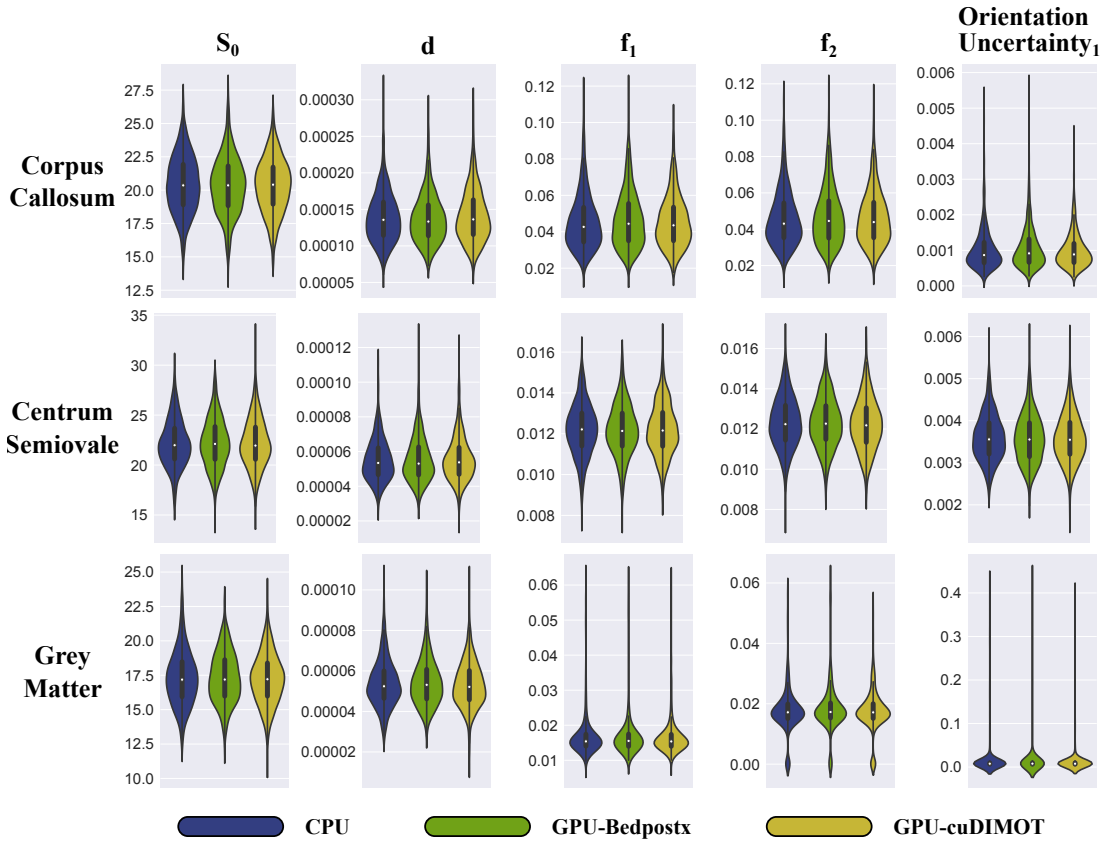

Figure 5.9: Distribution of the standard deviations of the recorded samples, across 1,000 repeats, for the baseline signal S_0, the diffusivity d, and the volume fraction of the first and second fibres f_1, f_2. The last column shows the distribution of the uncertainty when estimating the first fibre orientation across 1,000 repeats, which in each repeat was obtained from the dispersion of the mean dyadic vector. Voxels from different brain areas are shown: a corpus callosum voxel, a centrum semiovale voxel and a grey matter voxel. In each plot the standard deviation distribution is compared between CPU, GPU-Bedpostx and GPU-cuDIMOT implementations.

5.4 Exploring microstructure diffusion MRI models with cuDIMOT: Estimating fibre orientation dispersion

Many diffusion MRI models have been proposed for improving the characterisation of complex fibre configurations, most of them focusing on crossing fibre configurations [116, 157]. Recently, other approaches have focused on fibre fanning patterns [124, 125, 178], where fibre orientation dispersion indices can be locally inferred from the model. These dispersion indices may serve as markers for pathology, and have been validated with data observed from the microscopy [187].

In histological data, the presence of fanning patterns near cortical areas has been shown [188], and it has also been argued that a large portion ($> 40\%$) of the brain's parenchyma is better modelled when dispersion dMRI models are used for describing the diffusion signal [189]. Tractography techniques can also been improved when the dispersion of fibre orientations is considered in the models [190].

In this section we use cuDIMOT for exploring these fibre orientation dispersion models, which are typically very expensive to fit as the model predicted signals are expensive to evaluate.

First, we use the toolbox for implementing three extensively used dispersion models. The first one uses a Watson distribution [191] for representing the fibre orientation dispersion. This is an antipodal and cylindrically symmetric distribution that is equivalent to a Gaussian distribution on the sphere, i.e., a symmetric distribution around a main fibre orientation on a sphere. The two scenarios presented at the top row of Figure 5.10 can be represented by this distribution. The other two models use a Bingham distribution [191], [192]. This distribution is similar to the Watson distribution with the difference that it can be cylindrically asymmetric. The four scenarios in Figure 5.10 can be represented with this distribution.

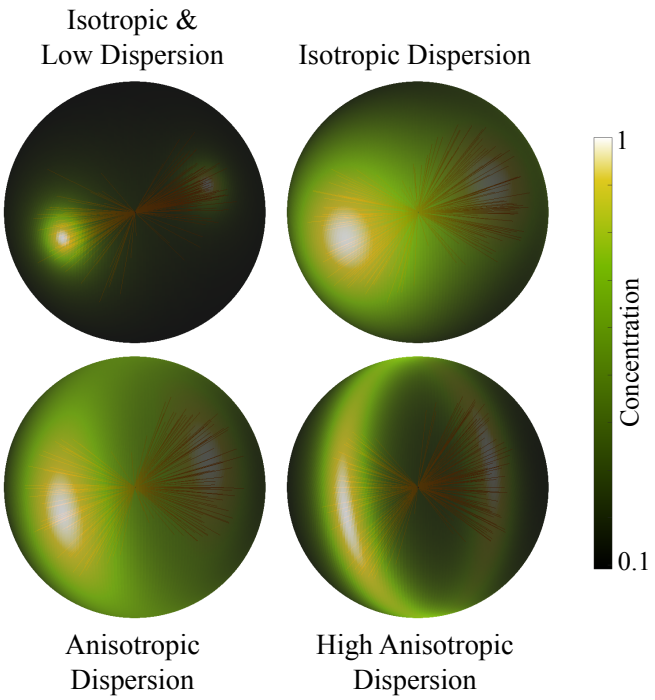

Figure 5.10: Example of 4 fibre orientation dispersion scenarios. In the first row, two scenarios with isotropic dispersion are represented, one with very low dispersion (left), and another one with higher dispersion (right). The second row shows a scenario with moderately anisotropic dispersion (left), and a scenario with high anisotropic dispersion (right).

5.4.1 Modelling the dispersion of fibre orientations with a Watson distribution

We start the exploration of the fibre orientation dispersion by considering some dMRI models proposed in the literature. The first model that we consider is the Neurite Orientation Dispersion and Density Imaging (NODDI) model [124, 193]. This model assumes that the signal comes from three different compartments:

- Isotropic compartment: represented by an isotropic 3D Gaussian diffusion process.
- Intra-cellular compartment: restricted diffusion represented by a distribution of sticks with certain orientation and radius zero.
- Extra-cellular compartment: hindered diffusion represented by a zeppelin (a cylindrically symmetric tensor) parallel to the mean orientation of the sticks.

Equation 5.11 shows the model signal S_m for each of the $m = 1 : M$ gradient directions applied:

$$S_m = S_0 \left[f_{iso} S_m^{iso} + (1 - f_{iso})(f_{intra} S_m^{intra} + (1 - f_{intra}) S_m^{extra}) \right] \tag{5.11}$$

S_0 is the baseline signal without any diffusion-sensitising gradient applied; $f_{iso} \in [0, 1]$ is the fraction of the isotropic compartment, and $f_{intra} \in [0, 1]$ is the fraction of the intra-cellular compartment relative to the aggregate fraction of the intra-cellular and extra-cellular compartments. S_m^{iso}, S_m^{intra} and S_m^{extra} are the signals from the isotropic, intra-cellular and extra-cellular compartments respectively.

The isotropic compartment is modelled as 3D Gaussian diffusion:

$$S_m^{iso} = \exp(-b_m d_{iso}) \tag{5.12}$$

where b_m (*b-value*) depends on the magnitude and duration of the m^{th} diffusion sensitizing gradient and d_{iso} is the diffusivity within this compartment, which is fixed to $3.0 \times 10^{-3}\ mm^2 s^{-1}$.

The signal from the intra-cellular compartment is modelled as:

$$S_m^{intra} = \int_{\mathbf{S}^2} H(n) \exp\left(-b_m d_{par} (\mathbf{g}_m^T n)^2\right) dn \tag{5.13}$$

where $\mathbf{g}_m$ indicates the direction of the m^{th} diffusion gradient, d_{par} is the diffusivity parallel to the sticks and the zeppelin, fixed to $1.7 \times 10^{-3}\ mm^2 s^{-1}$, and $H(n)$ is a fibre orientation distribution function (fODF) that returns the probability of finding a stick along the orientation n. The fODF is modelled with a Watson distribution:

$$H(n) = {}_1F_1\left(\frac{1}{2}; \frac{3}{2}; \kappa\right)^{-1} \exp(\kappa(\mathbf{v}n)^2) \tag{5.14}$$

${}_1F_1$ is the confluent hypergeometric function of the first kind, with a scalar parameter κ, that specifies the concentration of fibre orientations (the lower this value the

higher the dispersion), and $\mathbf{v}$ is the mean orientation of the sticks.

From now on, we name this model NODDI-Watson to differentiate it from a similar model that uses a Bingham distribution, which we name NODDI-Bingham.

The signal from the extra-cellular compartment is modelled as:

$$S_m^{extra} = \exp(-b_m \mathbf{g}_m^T \mathbf{D}_{ec} \mathbf{g}_m) \tag{5.15}$$

where $\mathbf{D}_{ec}$ is the signal of a zeppelin (see Appendix 5A for details).

The NODDI-Watson model has five free parameters: f_{iso}, f_{intra} , κ , $\theta \in [0, \pi]$ and $\phi \in [0, 2\pi]$, where θ and ϕ define $\mathbf{v}$ as:

$$\mathbf{v} = \begin{bmatrix} \sin(\theta)\cos(\phi) & \sin(\theta)\sin(\phi) & \cos(\theta) \end{bmatrix}^T \tag{5.16}$$

The concentration parameter κ can be transformed and expressed as the orientation dispersion index ($OD \in [0, 1]$):

$$OD = \frac{2}{\pi} \arctan\left(\frac{1}{\kappa}\right) \tag{5.17}$$

5.4.2 Modelling the dispersion of fibre orientations with a Bingham distribution

The NODDI-Watson model has a key limitation: it assumes that the fibre orientation dispersion is cylindrically symmetric. An alternative is the NODDI-Bingham model [178], which is able to characterise anisotropic dispersion (cylindrically asymmetric) using a Bingham distribution for describing the fibre orientations.

The NODDI-Bingham model assumes the same compartments as NODDI-Watson (Equation 5.11). The fODF $H(n)$ is given by a Bingham distribution and

the intra-cellular compartment signal is modelled as:

$$S_m^{intra} = \frac{{}_1F_1\left(\frac{1}{2};\frac{3}{2};\mathbf{B} - b_m d_{par}\mathbf{g}_m\mathbf{g}_m^T\right)}{{}_1F_1\left(\frac{1}{2};\frac{3}{2};\mathbf{B}\right)} \tag{5.18}$$

where $\mathbf{B}$ is a 3×3 symmetric matrix defined in terms of two concentrations parameters κ_1, κ_2 and a rotation matrix $\mathbf{R}$ (defined in Appendix 5B) :

$$\mathbf{B} = \mathbf{R}^T \begin{bmatrix} -\kappa_1 & 0 & 0 \\ 0 & -\kappa_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{R} \tag{5.19}$$

The rotation matrix $\mathbf{R}$ encodes two orientations: the mean orientation of the fibres defined with two angles θ and ϕ, and the rotation of the main dispersion direction ψ on a plane, which is orthogonal to the mean orientation of the fibres.

When $\kappa_1 = \kappa_2$ the dispersion is isotropic (scenarios at the top row in Figure 5.10). When $\kappa_1 > \kappa_2$ anisotropic dispersion occurs (second row in Figure 5.10).

The signal for the extra-cellular compartment is similar to NODDI-Watson (Equation 5.15), but in this case the signal from the zeppelin takes into account the possible anisotropic dispersion. Numerical differentiation is used for calculating the extra-cellular compartment signal (see Appendix 5B).

The model has seven free parameters: f_{iso}, f_{intra}, κ_1, κ_2, θ, ϕ and ψ.

The orientation dispersion index, $OD \in [0, 1]$, is defined in this case as:

$$OD = \frac{2}{\pi}\arctan\left(\left|\sqrt{\left(\frac{1}{\kappa_2}\right)\left(\frac{1}{\kappa_1}\right)}\right|\right) \tag{5.20}$$

And a new index, a factor of anisotropic dispersion $DA \in [0, 1]$, is given by:

$$DA = \frac{2}{\pi}\arctan\left(\frac{\kappa_1 - \kappa_2}{\kappa_2}\right) \tag{5.21}$$

5.4.3 The Ball & Rackets model

The ball & rackets model [125] is an alternative to NODDI-Bingham. This model also uses a Bingham distribution for describing the fibre orientations. It assumes only two compartments, an isotropic compartment and an anisotropic compartment. The model signal is given by:

$$S_m = S_0 \left[f_{iso} S_m^{iso} + (1 - f_{iso}) S_m^{aniso} \right] \tag{5.22}$$

where S_m^{iso} is given by Equation 5.12 and S_m^{aniso} is similar to S_m^{intra} in Equation 5.13, assuming $d_{iso} = d_{par}$. Contrary to NODDI models, the ball & rackets model estimates this diffusivity parameter, making the assumption that a single parameter within a voxel can characterise the diffusivity. In that respect the model is more phenomenological than NODDI. It tries to estimate the anisotropic compartment and its fibre orientation dispersion, and lets an isotropic compartment to model out all other contributions to the signal. The model has seven free parameters: $f_{iso},\ d,\ \kappa_1,\ \kappa_2,\ \theta,\ \phi$ and ψ.

5.5 Comparing dispersion models and implementations

Hardware Features

For the GPU experiments described below we use the same hardware and software as described in Section 4.4. For performing the CPU experiments we use 72 different CPU cores from a cluster with tens of nodes. Each node is comprised of 2 Intel Xeon E5-2680 v3 2.50 GHz processors, each one with 10 cores (20 CPU cores per node), and it has a total of 320 GB RDIMM memory. Major features of the CPU nodes are summarized in Table 5.2.

Element	Feature	Intel Xeon E5-2660 v3
Processing Units	Number of Cores (per processor)	10
	Clock frequency	2.60 GHz
Memory	DRAM - Main Memory (system)	320 GB
	L1 Cache (per processor)	25 MB
Processing Power per processor	Peak single precision	832 $GFlops$
	Peak double precision	416 $GFlops$
Processing Power per single core	Peak single precision	83.2 $GFlops$
	Peak double precision	41.6 $GFlops$
Power Consumption		105 $watts$
Price (January 2017)		$\sim$ £1,500

Table 5.2: Major features of Intel Xeon E5-2660 v3 processor

Diffusion–weighted MRI data

We used data from the UK Biobank project database [32]. Diffusion-weighting data were acquired using an EPI-based spin-echo pulse sequence in a 3T Siemens Skyra system. A voxel size of $2.0\times2.0\times2.0\ mm^3$ was used ($TR = 3.6\ s$, $TE = 92\ ms$, 32-channel coil, 6/8 partial Fourier) and 72 slices were acquired in total. Diffusion weighting was applied in $M = 105$ evenly spaced directions, with 5 directions $b = 0$ s/mm^2, 50 directions b = 1,000 s/mm^2 and 50 directions b = 2,000 s/mm^2. A multiband factor of 3 was employed [167, 168].

A T1 structural image (1 mm isotropic) of the same subject was used for creating a White & Grey matter mask, which was non-linear registered to the space of the diffusion dataset [194], and applied to the map of the estimated parameters before showing the results in the following experiments. For creating this mask, a brain extraction tool [39] and a tissue segmentation tool [40] were used.

5.5.1 Comparing NODDI-Watson to AMICO

The NODDI toolbox [195] implements the NODDI-Watson model in Matlab. The toolbox can parallelise the fitting process distributing the computation among several CPU cores. It assigns the fitting process for each voxel to a different CPU thread. However, a dMRI dataset contains hundreds of thousands of voxels, and

common computer clusters (in neuroscience centres) have tens to hundreds of cores. Moreover, the numerical approximations used to compute the model signal (e.g. for the confluent hypergeometric function) are computationally very expensive. For a typical dMRI dataset, the NODDI toolbox takes 40 hours to fit the model.

We implemented NODDI-Watson with cuDIMOT using the designer interface. Equation 5.13 and Equation 5.15 are implemented as in [193] (see Appendix 5A for details). We only provide the derivatives for f_{iso}, f_{intra}. Thus, cuDIMOT uses numerical differentiation to evaluate the partial derivatives of the rest of the parameters. In our cuDIMOT implementation we use the same optimisation steps as in the Matlab NODDI toolbox. Firstly we fit the diffusion tensor model for obtaining the mean of principal fibre orientation (θ and ϕ). Secondly we run a Grid-Search algorithm testing different combination of values for the parameters f_{iso}, f_{intra} and κ. Thirdly, we run Levenberg-Marquardt algorithm fitting only f_{iso} and f_{intra} , and fixing the rest of parameters. Finally, we run Levenberg-Marquardt fitting all the model parameters. The Matlab toolbox uses an active set algorithm [196] instead of Levenberg-Marquardt.

A different framework, Accelerated Microstructure Imaging via Convex Optimization (AMICO), has been proposed for accelerating the model-fitting process of NODDI-Watson [197]. The problem is reformulated as a linear system via convex optimisation. AMICO achieves accelerations of two orders of magnitude compared with the NODDI Matlab toolbox using the same hardware (a single CPU core). However, the algorithm does a discrete search in the multidimensional space of the problem, which may lead to failures and biases in the search of the global solution.

We fit the NODDI-Watson model to a UK Biobank dataset using these three approaches, the Matlab toolbox, AMICO and cuDIMOT. In Figure 5.11 we show maps with the estimated parameters, the used computational resources and execution times, and we compare the estimates obtained from the three tools.

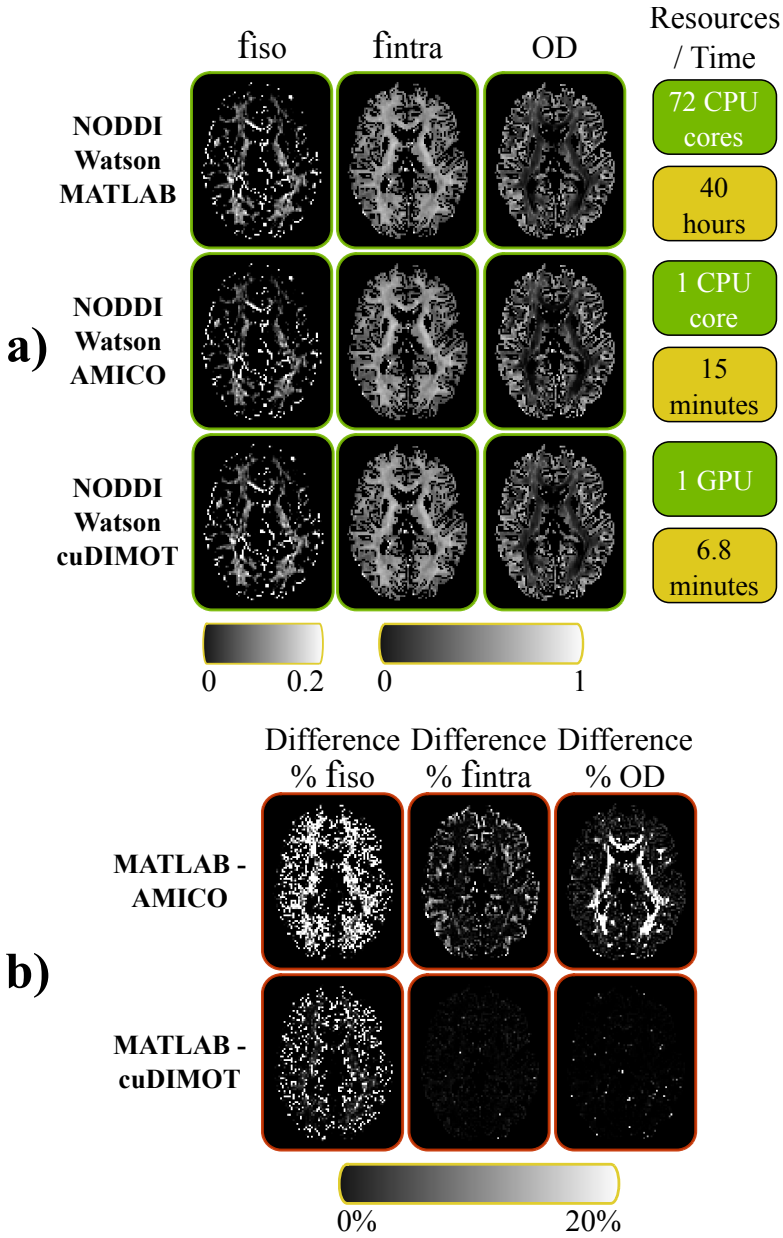

Figure 5.11: Comparison of three different tools fitting the NODDI-Watson model. (a) The results from each tool are presented in different rows. The first 3 columns show the map of the estimates for the parameters f_{iso}, f_{intra} and the index OD . The 4^{th} column shows the employed computational resources and the execution times. (b) Differences, in percentage, of the estimated values between the Matlab toolbox and the other approaches.

For running the Matlab tool, the dataset was divided into 72 axial slices, and each one was processed on a different CPU core. Both tools cuDIMOT and AMICO achieved accelerations of more than two orders of magnitude (cuDIMOT 352× and AMICO 160×) using a single NVIDIA K80 GPU and a single CPU core respectively. cuDIMOT was 2.2 times faster than AMICO.

For comparing the estimates, we subtract from the Matlab tool results the maps of the other two tools, and we calculate the percentage with respect to the former one. Figure 5.11b shows higher differences between the Matlab tool and AMICO, than between the Matlab tool and cuDIMOT. The differences between the Matlab tool and cuDIMOT are insignificant, except for the parameter f_{iso} in certain parts of the white matter. However, the values of f_{iso} are very low in the white matter, and the absolute difference between the Matlab tool and cuDIMOT after the map subtraction are very small (~ 0.003). The absolute differences between the Matlab tool and AMICO in the same area are more significant (~ 0.03).

In Figure 5.12 we show scatterplots of the estimated values throughout the brain. The correlations between the Matlab tool and AMICO, and between the Matlab tool and cuDIMOT are presented. cuDIMOT results are highly correlated to the results from the Matlab tool. We only find differences in few voxels where OD takes high values ($OD > 0.85$, high concentration) in the grey matter, which actually correspond with the voxels with higher differences between the Matlab toolbox and cuDIMOT in the estimation of OD in Figure 5.11b (at the bottom right hand corner). AMICO results are highly correlated as well, but less than cuDIMOT, and it is easy to see in the figure some discretisation artefacts in the estimates f_{intra} and OD.

These results indicate that cuDIMOT achieves very similar performance to the NODDI Matlab toolbox, and compared with AMICO, cuDIMOT is faster and obtains better results.

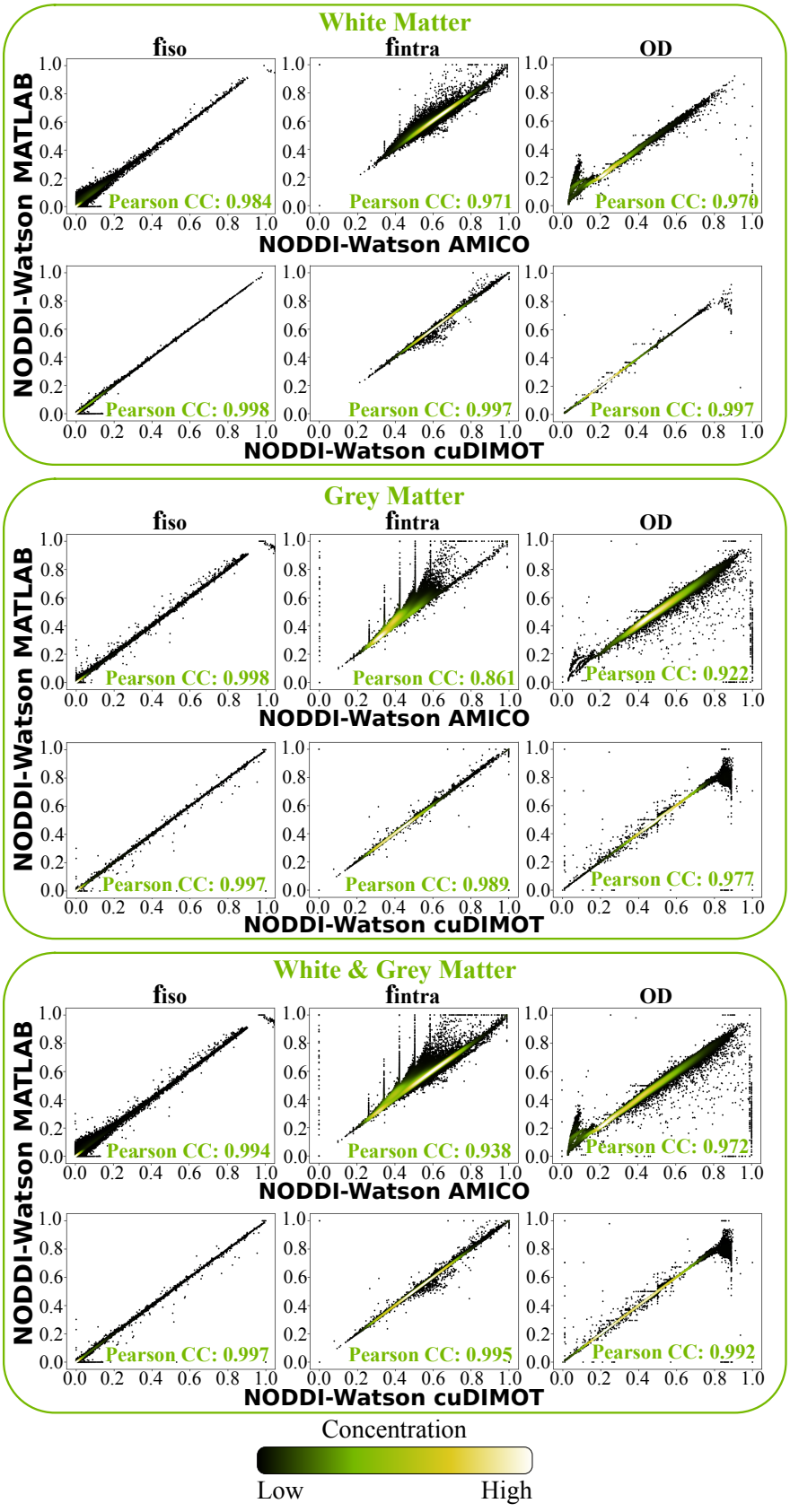

Figure 5.12: Correlations between the results from a Matlab tool and AMICO/cuDIMOT fitting the NODDI-Watson model in the white matter, grey matter and the combination of white & grey matter.

5.5.2 Comparing NODDI-Bingham implementations

The NODDI-Bingham extension has been implemented in the same Matlab toolbox as NODDI-Watson [195]. We implemented NODDI-Bingham using cuDIMOT in a similar manner as the previous model (only provided the analytic derivatives for the f_{iso} and f_{intra} parameters). For implementing the confluent hypergeometric function ${}_1F_1$ (in Equation 5.18) of a matrix argument, we use the approximation described in [198].

We compare the estimates obtained with cuDIMOT and the Matlab toolbox. For running the Matlab tool, we used again 72 CPU cores, and for running cuDIMOT, a single GPU device using the same optimisation steps as in the NODDI-Watson (diffusion tensor fitting, Grid-Search, and Levenberg-Marquardt twice). cuDIMOT was 7 times faster than the Matlab toolbox[2] (Figure 5.13)

For validating the cuDIMOT implementation, we compare the results from cuDIMOT with the ones obtained from the Matlab tool. In Figure 5.13 we show maps of the parameter estimates. We obtain similar results from both tools, however, we appreciate distinctions in the maps showing the percentage of difference between estimates (bottom row).

[2]Notice the almost sixfold difference between fitting NODDI-Bingham vs. NODDI-Watson using the Matlab toolbox, despite the higher complexity of NODDI-Bingham. This is due to the inefficient approximation of the confluent hypergeometric function of a scalar argument used in the NODDI-Watson implementation. For this reason, the comparisons in performance gains with cuDIMOT are more meaningful in the NODDI-Bingham case.

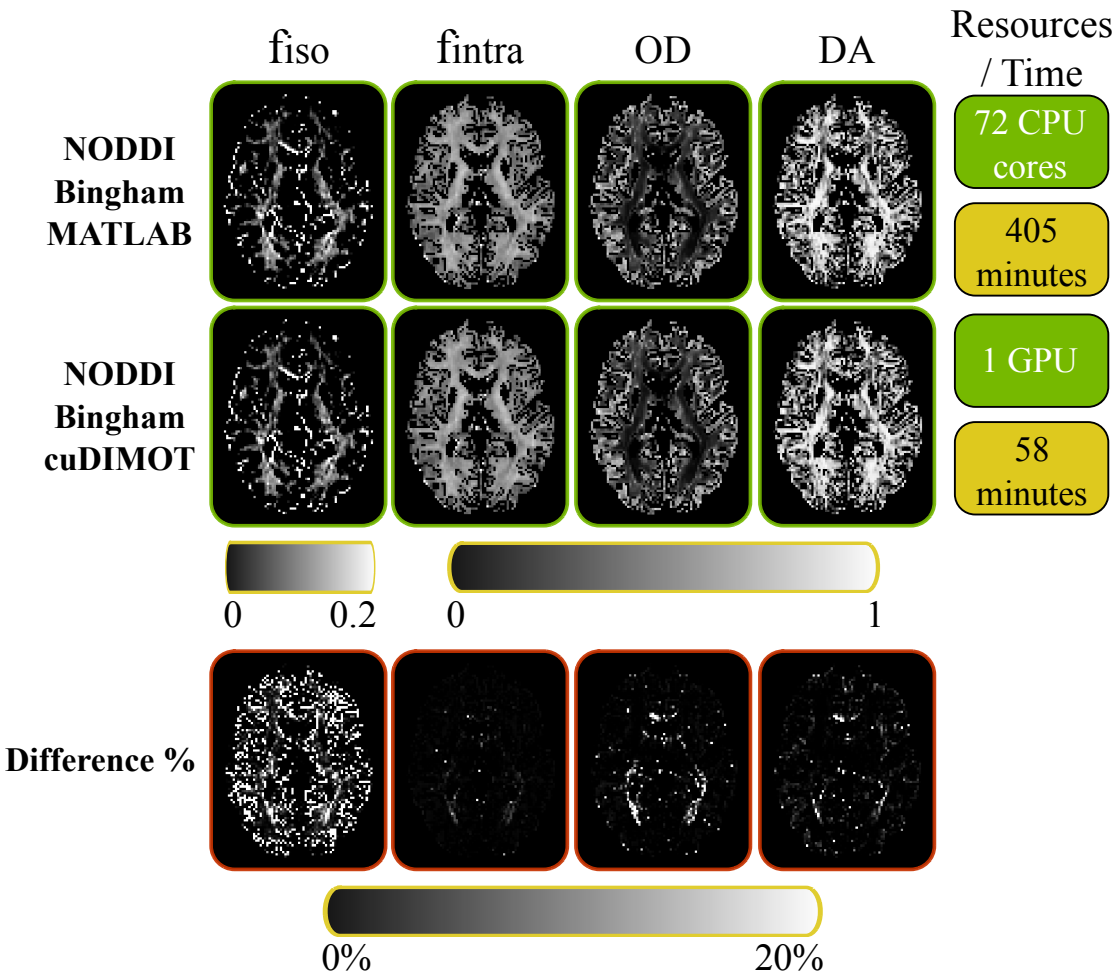

Figure 5.13: Comparison of a Matlab tool and cuDIMOT fitting the NODDI-Bingham model. The first 2 rows show the map of the estimates for the parameters f_{iso}, f_{intra}, the indices OD and DA, the used computational resources and execution times for each tool. The bottom row shows the differences, in percentage, of the estimated parameters between the Matlab tool and cuDIMOT.

In order to explore the significance of these differences, and investigate the source of these, we performed several tests. First, we examine the correlations between the results from both tools. Figure 5.14 shows scatterplots of the parameter values estimated using both methods throughout the brain. In all the parameters, the correlation coefficient was higher than 0.984 in the white matter, and 0.977 in the grey matter.

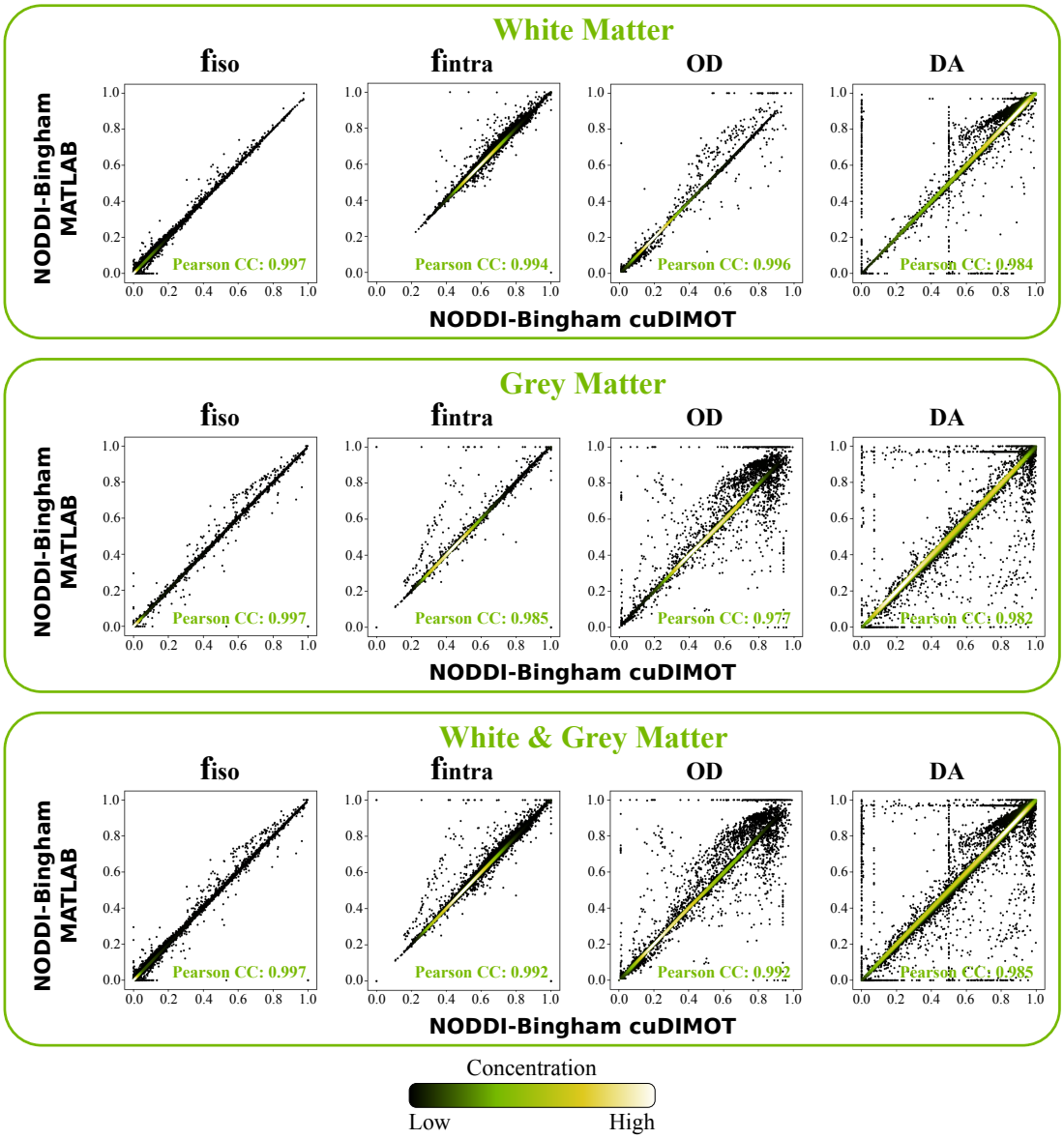

Figure 5.14: Correlations between the results from a Matlab tool and cuDIMOT fitting the NODDI-Bingham model in the white matter, grey matter and the combination of white & grey matter.

Afterwards, we perform a test using synthetic data. We synthesised dMRI data for creating four types of fibre orientation dispersion scenarios, which correspond to the ones depicted in Figure 5.10:

- A scenario with low and isotropic dispersion: $\kappa_1 = 40$, $\kappa_2 = 39$
- A scenario with isotropic dispersion: $\kappa_1 = 2.1$, $\kappa_2 = 1.8$
- A scenario with moderate anisotropic dispersion: $\kappa_1 = 5.5$, $\kappa_2 = 1.5$

- A scenario with high anisotropic dispersion: $\kappa_1 = 25$, $\kappa_2 = 0.9$

We used the model proposed in [193] (as implemented in the NODDI toolbox) to generate synthetic data, using a Bingham distribution of orientations. This is effectively a model similar to NODDI-Bingham, but assumes non-zero axon radii. Using this model and a multi-shell imaging protocol, we created synthetic data with 250 uniform distributed fibre orientations, 100 Rician noise realisations and 2 different axon radii, i.e., a total of 50,000 realisations per scenario. A summary of the imaging protocol and parameter values is given in Table 5.3.

Setting	**Number/Value**	**Parameter**	**Value**
Number of Shells	2 (711 & 2,855)	S_0	1,000
$b = 0\ s/mm^2$	9	f_{iso}, f_{intra}	0, 0.8
$b = 711\ s/mm^2$	30	d_{iso} d_{par}	$3.0 \times 10^{-3}\ mm^2s^{-1}$, $1.7 \times 10^{-3}\ mm^2s^{-1}$
$b = 2{,}855\ s/mm^2$	60	OD, DA	$40, 39; 2.1, 1.8;$ $5.5, 1.5; 25, 0.9$
SNR	20	Axon radii	1 μm & 2 μm
Rician Noise realization	100	Fibre Orientation	250 (uniform)
		ψ	$\frac{\pi}{3}$

Table 5.3: Summary of the imaging protocol and parameter values used for generating synthetic data.

We then used both the Matlab toolbox and cuDIMOT for fitting the NODDI-Bingham model to the synthetic data. Figure 5.15 shows the distribution of some parameter estimates and the ground-truth values used for generating the data (dashed fuchsia line). Very similar distributions are obtained from both tools, whereas it is clear that isotropic dispersions are more challenging to estimate (as noise tends to increase the estimated anisotropy). While low concentration values are estimated correctly, as reported in [178], high concentration values ($\kappa > 16$) are more difficult to estimate, which were not reported in [178].

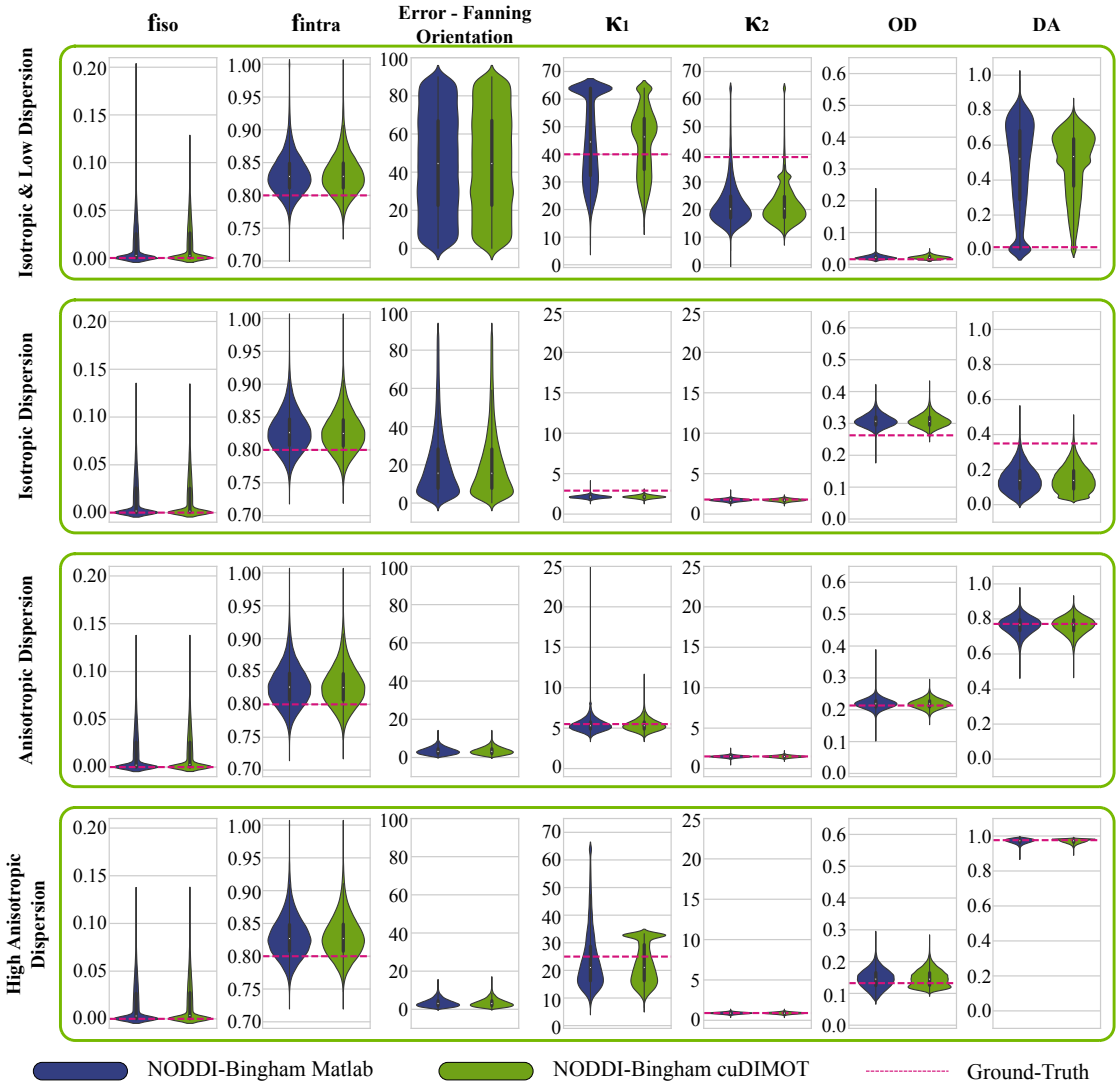

Figure 5.15: Distribution of the estimated parameter values, and the fanning orientation errors, from NODDI-Matlab tool and cuDIMOT fitting NODDI-Bingham model to synthetic data. Four different fibre orientation dispersion scenarios were considered for generating the synthetic data. The dashed fuchsia line represents the value used to synthesise the data.

Finally, we performed a cross-validation test to check which implementation performs better on data. For this test, we used again data from the UK Biobank project, which comprises 100 diffusion-weighting measurements from two different b-shells (50 from each b-shell), and 5 b_0 measurements. We created subsets with 85 diffusion-weighting measurements in total, 40 from each b-shell, and the 5 b_0 measurements. From the possible combinations $(\binom{50}{40}\binom{50}{40})$ for creating subsets with these features, we generate randomly 500. We use each subset to estimate the

model parameters, and then predict the non-used 20 measurements. We calculate a map with the error of the model, where the error for each test is given by:

$$Error_{CV} = \frac{1}{20} \sum_{m=0}^{19} \left(Y_m - S_m(\mathbf{\Theta})\right)^2 \tag{5.23}$$

Y_m is the m^{th} data measurement used for model evaluation, and S_m is the m^{th} predicted signal from the model given the estimated parameters θ.

Figure 5.16 shows the maps with the mean of $Error_{CV}$ values across the 500 tests obtained with the Matlab tool and cuDIMOT. We cannot identify significant differences between the models. Figure 5.16 also shows a map indicating in what voxels each model performs better (i.e., error is lower) than the other. The percentage of voxels where the Matlab tool, or cuDIMOT, performs better is almost 50%.

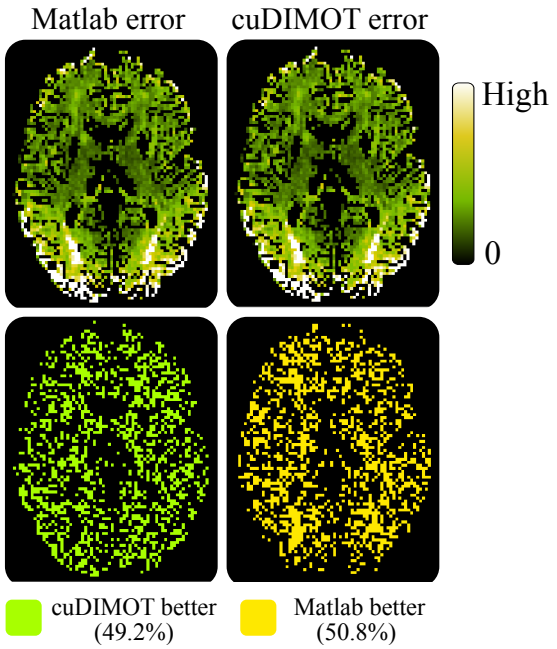

Figure 5.16: Maps with errors from a cross-validation test using a Matlab tool and cuDIMOT fitting the NODDI-Bingham model. The top row shows the mean of 500 cross-validation tests. The bottom row shows maps representing where one model is better than the other.

After performing these tests, we conclude that there are not significant differences between the Matlab tool and cuDIMOT implementations of the NODDI-Bingham model. We believe that the source of the differences in the Figure 5.13 come from:

- Using a different approximation of the hypergeometric function. In cuDIMOT we use a Saddlepoint approximation [198] and in the Matlab toolbox the function is approximated as in [199].
- Different non-linear optimisation method. We use Levenberg-Marquardt whereas the Matlab toolbox uses the active set algorithm included in the *fmincon* function [196].

5.5.3 Comparing NODDI-Bingham with ball & rackets

The ball & rackets model was also implemented with cuDIMOT and compared against the NODDI-Bingham. We used the same approximation [198] as before for approximating the confluent hypergeometric function of a matrix argument, and numerical differentiation for calculating the derivatives (except for f_{iso}) in Levenberg-Marquardt. We further tried Bayesian inference, for which a Gaussian distribution was used for defining a very informative prior for the parameter d (in mm^2/s):

$$P(d) = N(0.002, 0.0001) \tag{5.24}$$

The choice of these values is based on the NODDI model diffusivity values, which are fixed to 0.0017 mm^2/s and 0.003 mm^2/s for the parallel and isotropic diffusivity respectively.

We fit the model to the UK Biobank dataset running several optimisation routines. First, we fit the diffusion tensor model for obtaining the mean of the fibre orientations. Subsequently, we run the Grid-Search algorithm for initialising f_{iso}, κ_1 and κ_2. Then we run Levenberg-Marquardt algorithm fixing θ and ϕ. In the last two steps, we fit all the parameters, first running again the Levenberg-Marquardt algorithm, and finally the MCMC algorithm. Maps with the mean of some of the parameter estimates fitting the ball & rackets are shown in Figure 5.17.

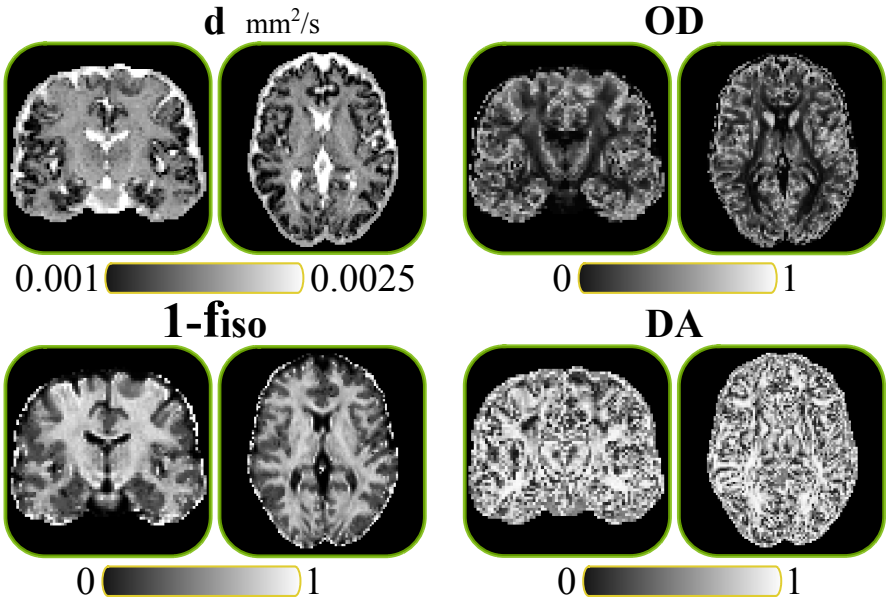

Figure 5.17: Maps with the mean of some of the parameter estimates fitting the ball & rackets model.

Subsequently, we compared ball & rackets to NODDI-Bingham, but to be consistent in the comparison, we rerun the fitting process of NODDI-Bingham adding the MCMC routine at the end of the process, (which seems to help NODDI-Bingham as well for the cases of isotropic dispersion, see Figure 5.15 and Figure 5.18).

Figure 5.18 shows comparisons in the estimates of the two models using synthetic data. Ball & rackets model is fitted with and without using MCMC routine. There are no significant differences between the two models when using MCMC, but it seems that NODDI-Bingham has fewer outliers and performs in general slightly better than ball & rackets, which tends to exhibit slightly higher bias in the estimated parameters. The differences are more significant when MCMC routine is not used for fitting the model. These results are somewhat expected, given that the data have been synthesised with a model that is effectively a generalisation of NODDI-Bingham. It is however reassuring that the two models' predictions agree reasonably well. On the other hand, ball & rackets seems to make a reasonable estimation of the diffusivity parameter d, which is fixed in NODDI-Bingham. In fact, the estimated diffusivity seems to be closer to the intra-axonal diffusivity, and these scenarios were simulated with a high intra-axonal volume fraction (0.8).

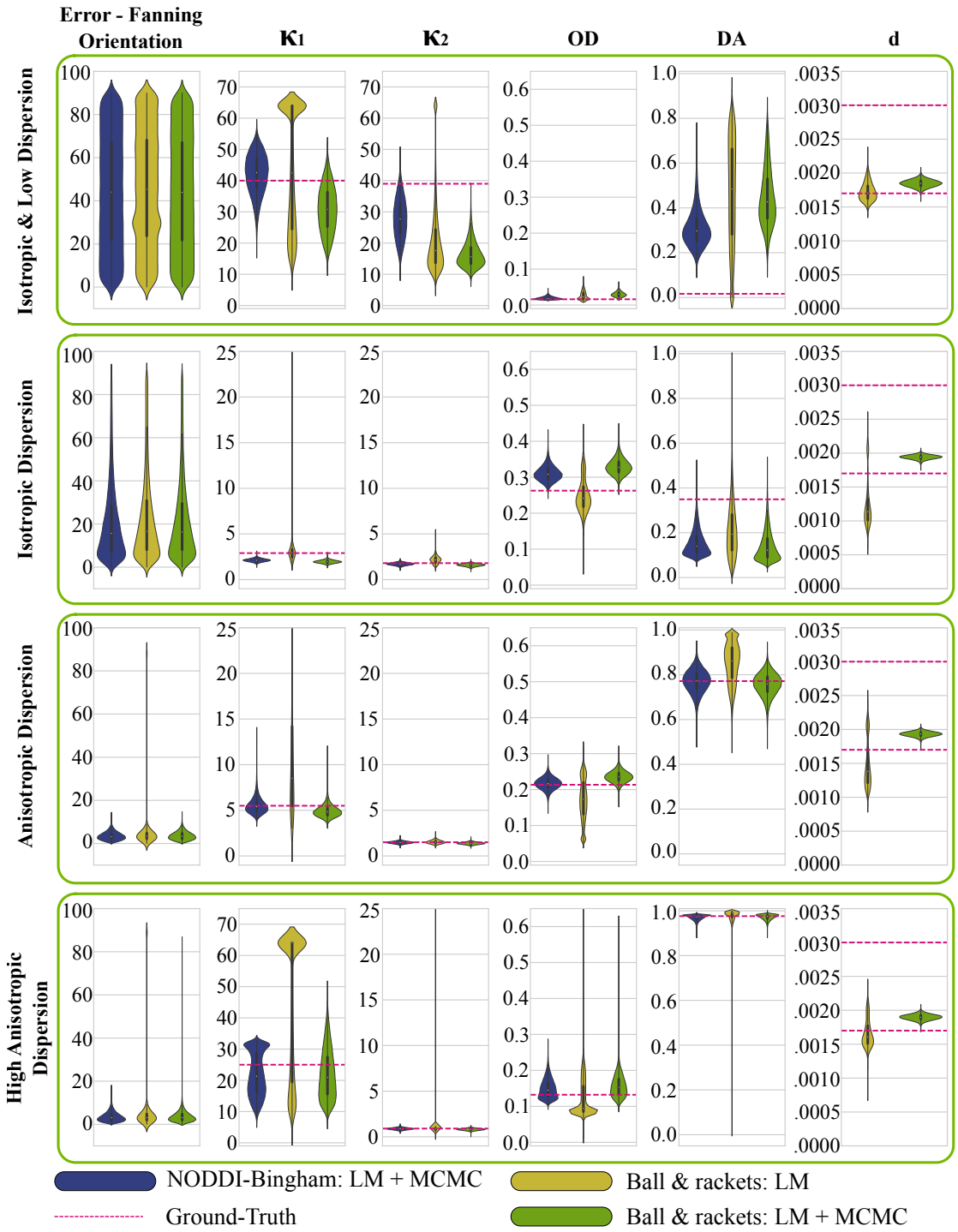

Figure 5.18: Comparison of NODDI-Bingham and ball & rackets models being fitted to synthetic data with four different fibre orientation dispersion scenarios. Both models were fitted using cuDIMOT: NODDI-Bingham running the MCMC routine in the last step, and ball & rackets with and without running MCMC routine. The dashed fuchsia line represents the value used to synthesise the data. Diffusivity (d) is fixed in NODDI model.

We also compared the performance of both models when they are fitted to real data. We calculated the BIC, and we do the same cross-validation test than before, obtaining the mean of the error of 500 cross-validation runs. Figure 5.19 shows the results. We found that NODDI-Bingham is better than ball & rackets in 80% of the voxels according to the BIC, and in 87% of the voxels according to the cross-validation error. It seems that ball & rackets underestimates the concentration parameters, i.e., overestimates the OD, especially in scenarios where the concentration is high. This can be due to the inability of a single diffusivity compartment (as assumed in the ball & rackets model) to capture the non-monoexponential decay of the signal at high *b-values*. In the context of the ball & sticks model, this leads to overestimation of crossings [157], which here, similarly, can lead to overestimate the dispersion.

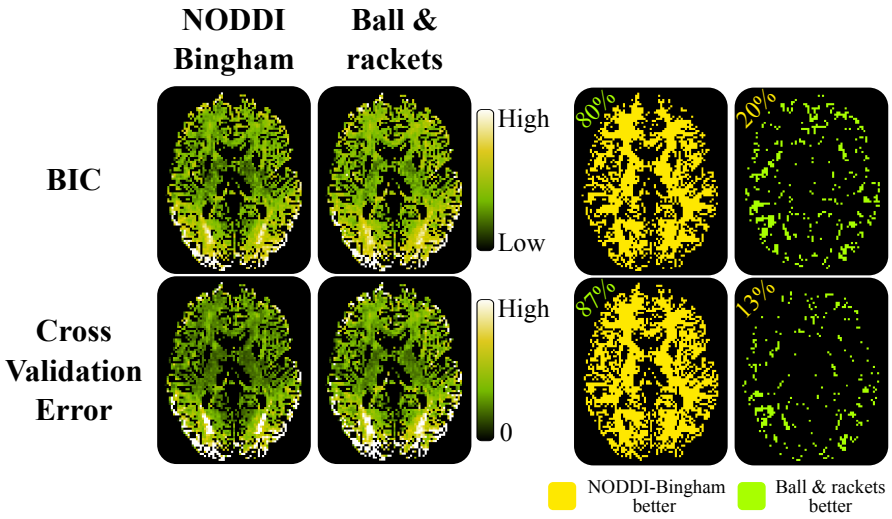

Figure 5.19: Comparison of the BIC and cross-validation error obtained when fitting NODDI-Bingham and ball & rackets model to a UK Biobank dataset. The fitting process for both models were performed with cuDIMOT and including MCMC.

5.5.4 Crossing fibres and dispersion of fibre orientations

NODDI and ball & rackets models are able to characterise the dispersion of a group of fibres dispersed around a mean orientation in each voxel. However, they ignore the fact that many regions of the white matter contain crossing fibre patterns [107]. We extend the NODDI-Bingham model to a model with two different Bingham

distributions, for representing two fibre populations. We use a similar approach to [200] but using Bingham distributions, instead of Watson distributions, and without fixing the fraction of the signal that is represented by each one (fixed to 50% in their approach), i.e., we estimate this parameter. The model signal is given by:

$$S_m = S_0 \Big[f_{iso} S_m^{iso} + (1 - f_{iso})(1 - f_{fan2})(f_{intra1} S_m^{intra1} + (1 - f_{intra1}) S_m^{extra1}) \\ + (1 - f_{iso})(f_{fan2})(f_{intra2} S_m^{intra2} + (1 - f_{intra2}) S_m^{extra2}) \Big] \tag{5.25}$$

S_m^{iso}, S_m^{intra1}, S_m^{intra2}, S_m^{extra1} and S_m^{extra2} are defined as in NODDI-Bingham.

The model has a total of 14 free parameters:

- Compartments fraction: f_{iso}, f_{fan2}, f_{intra1}, f_{intra2}
- First fibre distribution: κ_{1_1}, κ_{1_2}, θ_1, ϕ_1, ψ_1
- Second fibre distribution: κ_{2_1}, κ_{2_2}, θ_2, ϕ_2, ψ_2

We implement this model with cuDIMOT. We name it NODDI-2-Binghams. We fit it to the UK Biobank dataset, we calculate the BIC, and we compare it to the BIC obtained with NODDI-Bingham. MCMC was used in the pipeline for fitting both models. Figure 5.20 shows the results. In most of the voxels NODDI-Bingham performs better, however in some areas where the presence of crossing fibres is well known, such as the corona radiata, NODDI-2-Binghams performs better. The model seems to estimate correctly the two orientations in areas where it is better supported than NODDI-Bingham.

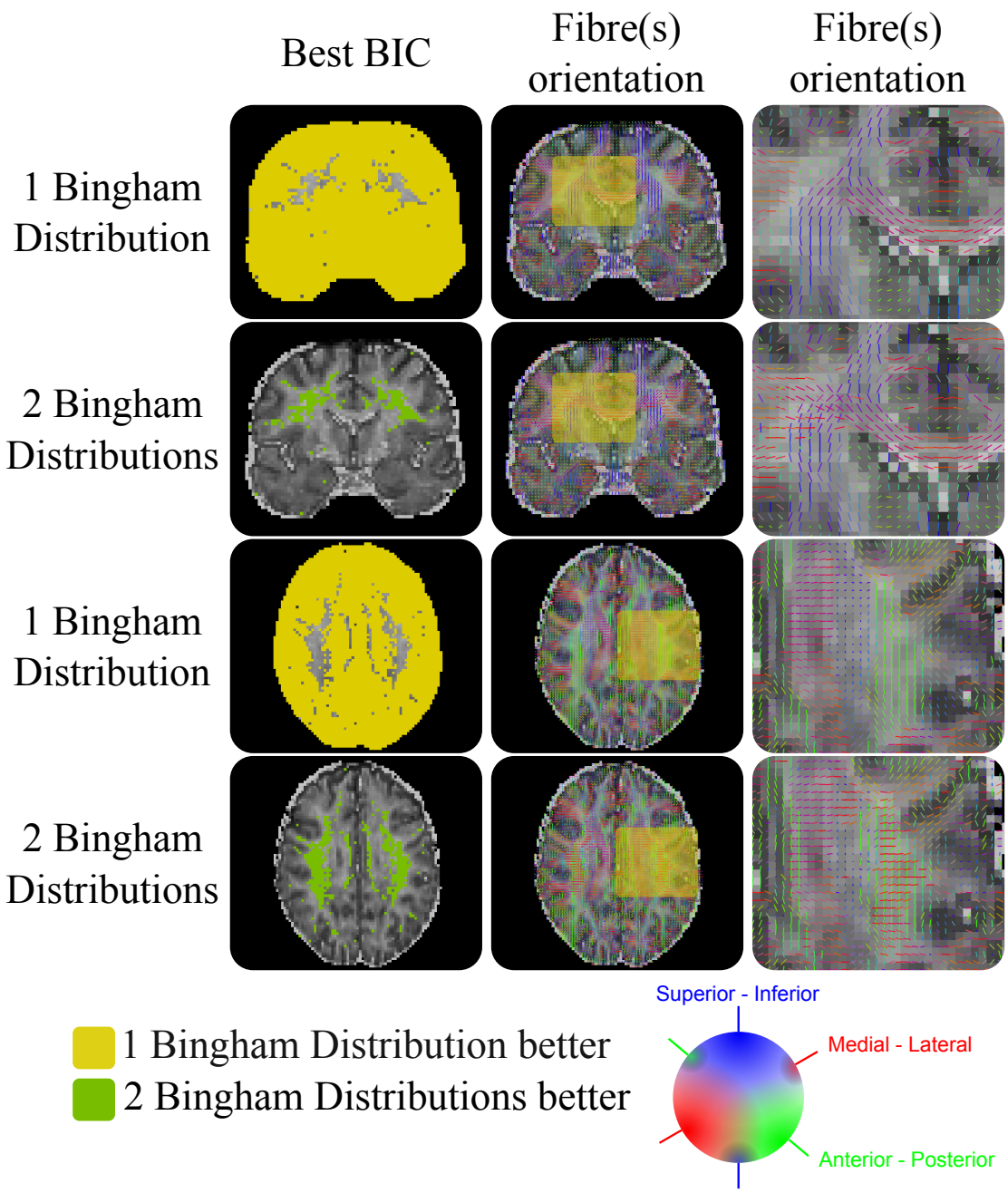

Figure 5.20: Comparison of NODDI-Bingham (with a single Bingham distribution) and NODDI-2-Binghams (with two Bingham distributions) models. The maps of the first column indicate where each model gets a better BIC (lower value) than the other. The maps of the second column show the estimated mean fibre(s) orientation of the models. The maps of the third column show enlarge areas of the previous column.

5.5.5 Model Selection

Finally, we performed a comparison of several models implemented with cuDIMOT. We use the BIC index that cuDIMOT offers, for comparing the performance of six models being fitted to the whole brain of the UK Biobank dataset. The models included in this test are:

- Ball & 1 stick (with gamma-distribution for the diffusivity [157])
- Ball & 2 sticks (with gamma-distribution for the diffusivity)
- NODDI-Watson
- Ball & racket
- NODDI-Bingham
- NODDI-2-Binghams

In all the cases we run an initialisation routine (Grid-Search or the output of the fitting process of other model), we run Levenberg-Marquardt and MCMC. cuDIMOT calculates the BIC from the mean of the parameter estimates.

We first classify the six models into two groups, one group with the models that do not characterise the dispersion of fibre orientations, which include the ball & stick(s) models, and another group with the models that characterise the dispersion. The second row in Figure 5.21 shows a colour-coded map indicating in what voxels each group gets a better BIC (lower). Using dispersion models the diffusion signal is better explained and the obtained BIC is lower (better) in the majority of brain regions.

The last row of Figure 5.21 compares the four considered dispersion models. Table 5.4 shows the percentage of voxels where each model (or class of models) gets the lowest BIC. The dominant model that gets a lower BIC is NODDI-Bingham.

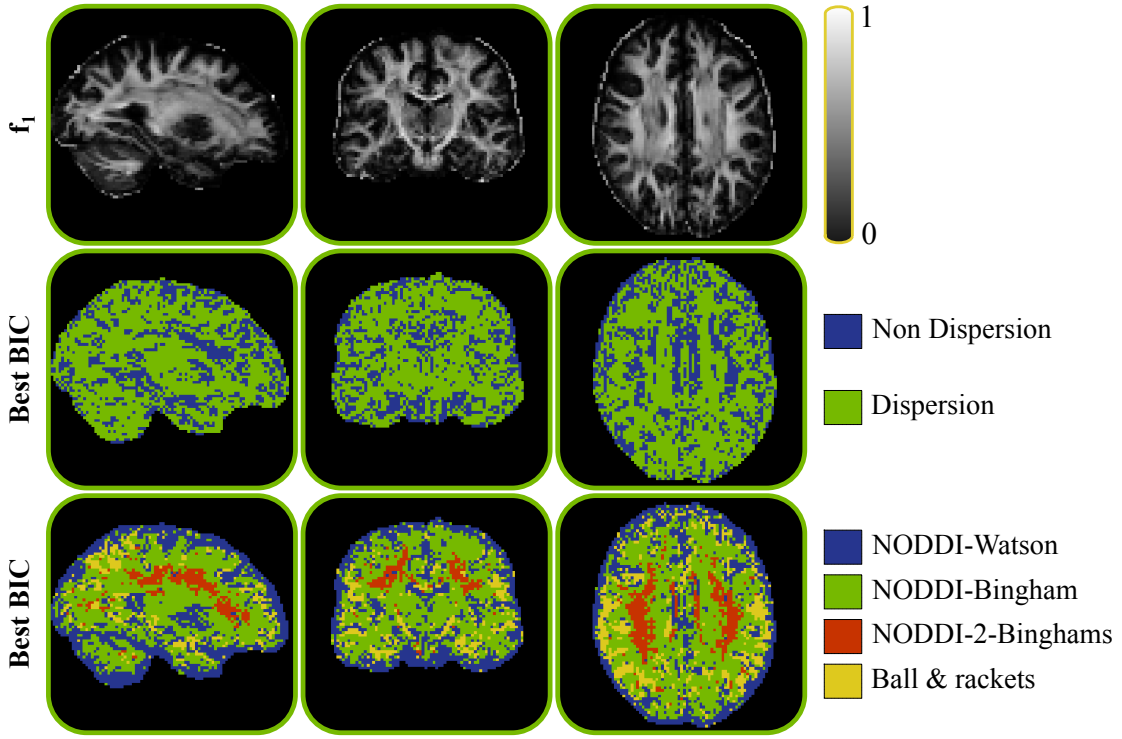

Figure 5.21: Model performance comparison. The first row shows a map for reference with the estimated fraction of the principal fibre in the ball & 2 sticks model. The second and third rows show color-coded maps indicating in what locations a model or a group of models get the best BIC.

	Non Dispersion	Dispersion		
% lowest BIC	33%	66%		
	NODDI-Watson	NODDI-Bingham	NODDI-2 Binghams	Ball & rackets
% lowest BIC	24%	55%	5%	15%

Table 5.4: Percentage of voxels where certain dMRI model(s) performs better than the rest, in terms of the BIC.

5.6 Discussion

We developed a generic toolbox, for designing and fitting non-linear diffusion models on GPUs, cuDIMOT (CUDA Diffusion Modelling Toolbox). The toolbox offers a friendly and flexible interface for implementing new models easily. The users only need to specify the number and type of parameters in the model, and a function that defines the model predicted signal using the C language. Additionally, the

partial derivatives can be provided for non-linear optimisation methods, and prior knowledge can be imposed to the parameters. Given this specification, cuDIMOT automatically generates parallel CUDA code and a binary file that can be executed on NVIDIA GPUs, which includes several routines for fitting the new model.

One of the key features of this toolbox is the provided flexibility in choosing the optimisation routines. Various model-fitting approaches are available, including Grid-Search, non-linear Levenberg-Marquardt optimisation and Bayesian inference using MCMC. Two noise models are available for the MCMC routine, Gaussian and Rician noise models. In the future, the toolbox can be extended for including other fitting routines. Additionally, several model fitting processes can be cascaded with cuDIMOT, and the toolbox can optionally generate model selection maps, such as BIC and AIC.

A toolbox with the same purpose as cuDIMOT, has been recently presented [201]. However, it does not include the option for performing Bayesian inference, which is a desired feature for fitting MRI models given the uncertainty introduced by the presence of noise in MRI data. Moreover, the previous work does not include the option to add constraints and priors directly. Finally, its design assumes a combination of model compartments, which can restrict the flexibility for defining the model predicted signal. cuDIMOT does not assume any form in the model predicted signal, and any mathematical expression can be used in the model design interface.

Our toolbox, similar to the solutions proposed in the previous chapter, has been designed for processing datasets with any number of voxels and measurements, iterating over groups of voxels in case that a GPU has not enough memory for allocating the whole dataset, and having a performance penalty if the device memory capacity is small because of the large number of CPU-GPU transfers required. Additionally, in cuDIMOT there is a limitation on the number of parameters of a model. In the Levenberg-Marquardt routine the number of model parameters is limited to 31. The cause of this limitation is in the implementation of the LU solver (see Figure 5.5), where each thread of a warp processes a column of the

matrix for solving the system. For a model with P parameters, $P + 1$ threads are required, and a warp has 32 threads. In the MCMC routine also exists a limitation on the number of model parameters. The framework stores in Shared memory the parameters and some associated information (priors, number of proposals accepted/rejected, and standard deviation of the proposals distribution). Thus, a model is limited to a maximum number of around 300 parameters, (this number depends on the size of Shared memory of the specific GPU and on the precision used to store the parameters, single or double).

We validated cuDIMOT fitting the ball & sticks model and comparing the results with the ones obtained from a sequential tool, and from a GPU-tool developed for that specific model [144]. The three tools gave almost identical distribution of estimates after repeating many times the fitting process.

On average, our toolbox achieves accelerations of two orders of magnitude compared with commonly used tools for fitting dMRI models running on a single CPU core. Nowadays, it is very common to have computing clusters available for processing MRI datasets. We report in Table 5.5 the speedups obtained by cuDIMOT fitting several models and using a single NVIDIA K80 GPU, compared with the commonly used tools running on 72 CPU cores. A UK Biobank dataset was used for this experiment. Compared to a C++ tool [38] cuDIMOT is on average 3.67 times faster.

	Ball 1-stick [38, 116]	Ball 2-sticks [38, 116]	Ball 1-stick Gamma [38, 157]	Ball 2-sticks Gamma [38, 157]	NODDI Watson [124, 195]	NODDI Bingham [178, 195]
Common Tools 72 CPU cores	720 s	1,380 s	1,260 s	2,520 s	2,400 m	405 m
cuDIMOT single NVIDIA K80 GPU	187 s	423 s	324 s	679 s	6.8 m	58 m
Speedup	3.85×	3.26×	3.88×	3.7×	352×	6.98×

Table 5.5: Speedups obtained by cuDIMOT, fitting several dMRI models to a dataset from the UK Biobank on a single K80 NVIDIA GPU, compared with the commonly used tools that implement these models and executed on a computing cluster with 72 CPU cores.

We took advantage of the flexibility of cuDIMOT and we used it to explore diffusion MRI models that characterise the fibre orientation dispersion. We considered models that use Watson and Bingham distributions for representing such dispersion.

First, we considered the NODDI-Watson model [124]. The toolbox that was available for fitting this model was implemented in Matlab [195], which turned out to be computationally very inefficient. It took more than a day and a half to process a UK Biobank dataset using 72 CPU cores. An alternative to this tool was proposed by [197], where the fitting process was accelerated by reformulating the non-linear problem as a linearised system. Although massive accelerations are obtained using this tool, we have reported remarkable differences in the results compared with the Matlab toolbox. We implemented with cuDIMOT the NODDI-Watson model. We reported very similar results to the Matlab toolbox, finding differences only in few voxels in the grey matter where the dispersion index OD takes high values (see Figure 5.12). In our cuDIMOT implementation we approximate the Dawson's integral as in [205]. The Matlab toolbox probably uses a different approximation (source code not available), and it seems that the tool has difficulties to represent scenarios where the concentration parameter is very low ($\kappa < 0.2$). We believe that the differences come from the different approximations. We will explore in future

work the effect of different approximations. Nevertheless, using cuDIMOT for fitting the NODDI-Watson model we obtained massive accelerations using a single GPU (NVIDIA K80). It took less than 7 minutes to process a UK Biobank dataset.

We also considered the NODDI-Bingham model [178], which is similar to NODDI-Watson but it uses a Bingham distribution for representing the fibre orientation dispersion. The model is implemented in the same Matlab toolbox as NODDI-Watson. The fitting process of this model is less, but still very time consuming; it took more than 6 hours to process a UK Biobank dataset using 72 CPU cores. We implemented the model with cuDIMOT and we compared the performance and the results to the Matlab toolbox. cuDIMOT processed the same dataset in 58 minutes using a single GPU (NVIDIA K80). In this case we also observed some differences in the results. We performed several tests to discard significant differences between both tools, including correlation between results, synthetic data experiments and cross-validation. We could not find any significant difference, and we believe that the causes of the minor differences, come again from the different approximations, and from the optimisations methods used for fitting the model. In the results from the correlation test, we found some voxels where one of the toolbox returns a very low DA (near zero) but not the other (see Figure 5.14, white matter row). We found that these voxels represent a very low proportion of the whole dataset, 0.2%, and they are at the interface between white matter and CSF, as shown in Figure 5.22

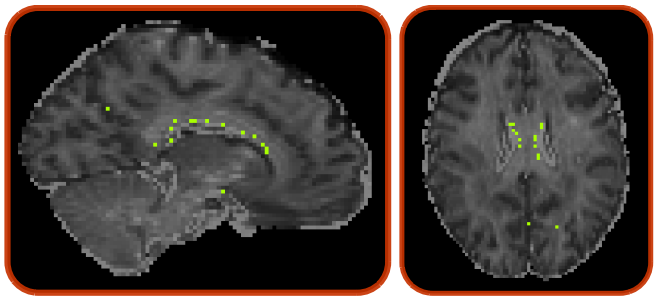

Figure 5.22: Map with the voxels where the obtained DA, fitting the NODDI-Bingham model, is very low in Matlab toolbox or cuDIMOT but not in the other implementation.

Finally, we implemented the ball & rackets model [125] with cuDIMOT, which is an alternative model to NODDI-Bingham that also uses a Bingham distribution for representing the fibre orientation dispersion. It has the advantage of estimating the diffusivity, which was a fixed parameter in the NODDI models. We fit the ball & rackets and NODDI-Bingham models to synthetic and real data and we compare their performance. For the real data experiments we performed a cross-validation test fitting a UK Biobank dataset. The errors obtained using both models were very similar, however, in most of the voxels, NODDI-Bingham performed better. This can be due to the inability of a single diffusivity compartment (as used in the ball & racket model) to capture the signal behaviour for multi-shell and high *b-values* data. An extension of ball & rackets similar to [157], with a distribution of diffusivities may improve the model and reduce any biases in the estimated parameters.

All the previous models can represent only one distribution of fibre orientations. We proposed an extension of the NODDI-Bingham that duplicates its intra and extra cellular compartments. The dispersion is represented by two Bingham distributions, which potentially can characterise crossing fibres configurations. Using the BIC, we compared the model to NODDI-Bingham and we found that in crossing regions, such as the corona radiata, the signal is better explained when 2 Bingham distributions are used, but the model seems to perform much worse in the rest of the areas. Other models that combine dispersion and crossing fibres have been recently proposed, using a Watson distribution [197, 200, 202]. The main limitations is the high-dimensionality of the model, which make the fitting process difficult and extremely time consuming, thus, these approaches reformulate the problem for simplifying the system, or they use less demanding optimisation routines. Here we use the typical optimisation routines for fitting the model, i.e., Grid-Search, Levenberg-Marquardt and MCMC. It took 6 hours for processing a UK Biobank dataset using 8 NVIDIA GPUs (mix of K20, K40 and K80 GPUs). Even if this computation time is very high using GPUs, it illustrates the high computational needs for performing such an exploration, which would be infeasible in practice using only CPUs.

We performed a comparison, using the BIC, between models characterising the fibre orientation dispersion and not. The models that characterise the dispersion performed better in 65% of the voxels. Comparing only the dispersion models (the ones considered here), the NODDI-Bingham model performs better than the rest in 55% of the voxels, followed by NODDI-Watson that performs better in 24% of the voxels. These results coincide with the ones reported in [189, 203] and suggest that an automatic model selection algorithm would be very beneficial. But this involves fitting different models to the data, which may be very time consuming and prohibitive. Here, we have shown how GPUs can be used to accelerate these fitting processes and obtain the results in reasonable times, making plausible the idea of implementing an automatic model selection algorithm.

Finally, we want to remark again, that although cuDIMOT was originally conceived for fitting diffusion MRI models (thus the name), it can be used for fitting any MRI voxel-wise model on GPUs. The toolbox performs the model fitting process independently for each voxel. Thus, approaches where neighbourhood information is used cannot be directly implemented in cuDIMOT, unless up/down-sampling methods are used in conjunction with the toolbox cascade functionality. For instance, a model can be fitted to a low resolution dataset, and its voxel-wise estimates can be up-sampled and used for initialising the model fitting to a high resolution dataset.

We believe that cuDIMOT is going to be very useful in the development and improvement of new diffusion MRI models, which may explain the complexity of the diffusion process, extract useful biophysical parameters and contribute to the development of new biomarkers.

Appendix 5A

NODDI-Watson implementation

The intracellular signal S_m^{intra} in Equation 5.13 does not have an analytic expression and it is numerically approximated using the implementation presented in [193]. A spherical harmonic approximation of the Watson distribution is used. The integral can be analytically expressed as (Equation 10 in [193]):

$$S_m^{intra} = \sum_{l=0}^{\infty} f_{l0}^c(\kappa)\sqrt{\frac{2l+1}{4\pi}}P_l(\mathbf{g_m v})Q_l(d_{par}) \tag{5A.1}$$

where the summation is truncated to the 12^{th} degree ($l = 12$). f_{l0}^c is the spherical harmonic coefficient with degree l given analytically as a function of κ in [193] and provided in the Matlab NODDI toolbox [195]. P_l is the Legendre polynomial and Q_l is given by (Equation $A.5$ in [193] with $L_\perp = 0$):

$$Q_l(d) = \frac{1}{2}C_l(b_m d) \tag{5A.2}$$

with C_l as in Equation 12 in [204] (replacing $2n$ with l and x with bd).

In the extra-cellular signal S_m^{extra} (Equation 5.15), $\mathbf{D}_{ec}$ is given by (Equation 7 in [193]):

$$\mathbf{D}_{ec} = (d'_{par} - d'_{perp})\mathbf{v}\mathbf{v}^T + d'_{perp}I \tag{5A.3}$$

where I is the identity tensor and d'_{par} and d'_{perp} are given by (Equation 5 and Equation 6 in [124]):

$$d'_{par} = d_{par} - d_{par}f_{intra}(1 - \tau_1) \tag{5A.4}$$

$$d'_{perp} = d_{par} - d_{par}f_{intra}\left(\frac{1+\tau_1}{2}\right) \tag{5A.5}$$

assuming a tortuosity model [152] where the diffusivity perpendicular to the zeppelin d_{perp} is defined as:

$$d_{perp} = d_{par}(1 - f_{intra}) \tag{5A.6}$$

and τ_1 is a function of the concentration parameter κ and is given by (Equation 8 in [124]):

$$\tau_1 = -\frac{1}{2\kappa} + \frac{1}{2F\left(\sqrt{\kappa}\right)\sqrt{\kappa}} \tag{5A.7}$$

where F is the Dawson's integral , which we approximate as in [205].

Appendix 5B

NODDI-Bingham implementation

The rotation matrix $\mathbf{R}$ included in Equation 5.19 is defined as in [125]:

$$\mathbf{R} = \mathbf{R}_\psi \mathbf{R}_\theta \mathbf{R}_\phi \tag{5B.1}$$

θ and ϕ defines the mean orientation of the fibres and ψ the rotation of dispersion on a plane that is orthogonal to the mean orientation of the fibres:

$$\mathbf{R}_\psi = \begin{bmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5B.2}$$

$$\mathbf{R}_\theta = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \tag{5B.3}$$

$$\mathbf{R}_\phi = \begin{bmatrix} \cos(\phi) & \sin(\phi) & 0 \\ -\sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5B.4}$$

$\theta \in [0, \pi],\ \phi \in [0, 2\pi],\ \psi \in [0, \pi]$.

The extra-cellular signal S_m^{extra} is given by (as Equation 5.15):

$$S_m^{extra} = \exp(-b_m \mathbf{g}_m^T \mathbf{D}_{ec} \mathbf{g}_m) \tag{5B.5}$$

$\mathbf{D}_{ec}$ is defined as Equation 13 in [178]:

$$\mathbf{D}_{ec} = \begin{pmatrix} \mu_1 & \mu_2 & \mu_3 \end{pmatrix} \begin{pmatrix} d_{\mu_1} & 0 & 0 \\ 0 & d_{\mu_2} & 0 \\ 0 & 0 & d_{\mu_3} \end{pmatrix} \tag{5B.6}$$

μ_1 represents the mean orientation of the fibres, and μ_2 and μ_3 the directions orthogonal to μ_1. The diffusivity along each direction d_{μ_n} is given by (Equations

14-16 in [178]):

$$d_{\mu_1} = d_{perp}(d_{par} - d_{perp})\frac{\partial_1 F_1\left(\frac{1}{2};\frac{3}{2};\mathbf{B}\right)}{\partial\kappa_1} \tag{5B.7}$$

$$d_{\mu_2} = d_{perp}(d_{par} - d_{perp})\frac{\partial_1 F_1\left(\frac{1}{2};\frac{3}{2};\mathbf{B}\right)}{\partial\beta} \tag{5B.8}$$

With the transformation $\beta = \kappa_1 - \kappa_2$

$$\beta = \kappa_1 - \kappa_2 \tag{5B.9}$$

$$d_{\mu_3} = (d_{par} + 2d_{perp}) - d_{\mu_1} - d_{\mu_2} \tag{5B.10}$$

The model assumes the same tortuosity model [152] as NODDI-Watson (Equation 5A.6).

6

Probabilistic Tractography and Connectomes on GPUs

Contents

6.1 Introduction. Challenges of a parallel tractography design . . . 175
6.2 Designing a GPU-accelerated probabilistic tractography framework . . . 177
6.2.1 Framework functionality . . . 177
6.2.2 Surfaces for performing tractography . . . 179
6.2.3 GPU parallel design and strategies . . . 180
6.2.4 Implementation considerations . . . 191
6.3 Results: Anatomical Constraints in Tractography . . . 194
6.3.1 Using pial surfaces . . . 194
6.3.2 Advanced termination anatomical constraints . . . 194
6.4 Results: Performance gains using the GPU implementation . . . 196
6.4.1 Reconstructing white matter tracts . . . 197
6.4.2 Generating dense connectomes . . . 200
6.5 Validation: Comparison of GPU to CPU designs . . . 201
6.6 Discussion . . . 205
Appendix 6A: Streamline-surface mesh intersection . . . 210

Overview

Tractography methods have great potential for mapping brain connections. They allow reconstructing human white matter pathways from dMRI, non-invasively and in vivo. Probabilistic tractography methods are able to represent the uncertainty introduced by the presence of noise in dMRI data, and by the model inaccuracies.

However, tractography methods can be computationally very demanding. In particular, probabilistic approaches for mapping whole-brain connectomes [139] typically need to propagate millions of streamlines for achieving convergence, while complex protocols may be used when anatomical constraints are imposed. Moreover, recent efforts for mapping the human brain in health and disease require tractography for analysing hundreds [2–4] to tens of thousands of subjects [32, 33], increasing even more the computational demands.

Typically the propagation of each streamline is independent, and therefore, the method can be potentially parallelised. Given the high number of independent tasks, GPUs seem to be a good candidate for implementing an efficient parallel solution. However, contrary to the applications we have considered so far, there is a number of factors that make the GPU implementation of tractography methods non-trivial and challenging. The required data for propagating each streamline is not known in advance, as their paths are reconstructed dynamically on the fly. This makes the allocation of GPU resources difficult, and therefore, the a-priori assessment of the parallelisability of the application challenging. Moreover, the streamlines propagation is likely to be asynchronous, which induces divergence. And furthermore, the methods have typically high memory requirements and include heavy tasks, particularly for large datasets and whole-brain explorations, making even less straightforward the design of an efficient GPU-accelerated solution (ideally composed of small tasks).

In this chapter, we present a parallel GPU framework that performs probabilistic streamline tractography and propose solutions for the above challenges. The framework reduces computation times by more than two orders of magnitude. Apart from the considerable speed-ups, the framework also supports extra functionality compared with typical tractography approaches. This includes the ability to handle both volumes and surfaces in a tractography protocol, and options that allow large flexibility in imposing anatomical constraints.

Contributions of this chapter

- Design and development of a fast and flexible probabilistic tractography framework on GPUs.
- Functionality for generating dense connectomes and for defining surfaces-based constraints is included in the framework.
- We propose different parallel strategies and solutions for the challenges linked to a GPU-based probabilistic tractography design.
- We show the improvements obtained in tractography when using the extra functionality supported by the framework.
- We assess the performance gains and we validate the new design against a CPU implementation.

Publications

Contributions from this chapter have appeared in the following:

- Madhyastha T.M.,Koh N., McAllister-Day T.K., **Hernandez Fernandez, M.**, Kelley A., Peterson D.J., Rajan S., Woelfer K., Wolf J., Grabowski T.J. "Running Neuroimaging Applications on Amazon Web Services: How, When, and at What Cost?". *Frontiers in Neuroinformatics*, 11(63). (2017)
- **Hernandez-Fernandez M.**, Reguly I., Giles M., Jbabdi S., Smith S., Sotiropoulos S. N. "A fast and flexible toolbox for tracking brain connections in diffusion MRI datasets using GPUs". In: *The Organization for Human Brain Mapping (OHBM)*, Geneva (Switzerland). (2016).
- **Hernandez-Fernandez M.**, Reguly I., Giles M., Smith S., Sotiropoulos S.N. "White matter tractography and Human Brain Connections using GPUs". In *GPU Technology Conference*, San Jose (CA, US). (2016).
- Donahue C.J., Sotiropoulos S.N., Jbabdi S., **Hernandez-Fernandez M.**, Behrens T.E., Dyrby T.B., Coalson T., Kennedy H., Knoblauch K., Van Essen D.C., Glasser, M. F. "Using Diffusion Tractography to Predict Cortical Connection Strength and Distance: A Quantitative Comparison with Tracers in the Monkey." *The Journal of Neuroscience: The Official Journal of the Society for Neuroscience*, 36(25): pp.6758–70.(2016)

Case studies

- NVIDIA GPU Center of Excellence Achievement Award 2016: White matter tractography and Human Brain Connections using GPUs.
- Electronic publication. A case study about the software we have developed using GPUs for analyzing brain diffusion MRI data: Imaging Software Bring the Brain into Fuller Focus, The Science and Engineering South Consortium, UK.

Software

- Tool for performing probabilistic tractography on GPUs (Probtrackx_gpu): `http://users.fmrib.ox.ac.uk/~moisesf/Probtrackx_GPU`

6.1 Introduction. Challenges of a parallel tractography design

Tractography is a family of algorithms for estimating the human white matter pathways, non-invasively and in vivo. The aim of this mapping is to understand the underlying anatomical and structural organisation of the white matter for gaining new insight into brain mechanisms and function.

Tractography algorithms make the assumption that, within a bundle of axons, diffusion occurs mainly along the major axis of that bundle. A map with the preferred diffusion orientations in each voxel of the brain is used as an estimate for the underlying axonal orientations. Typically, these algorithms integrate the orientations of locally adjacent voxels for reconstructing long white matter pathways.

This method for localising and *"measuring"* the white matter pathways is an indirect method that relies on diffusion MRI measurements, which contain noise, and on models that estimate locally the fibre orientations as average orientations of thousands of axons contained inside a voxel. Moreover, the algorithms need to make decisions in the integration process, and may introduce even more uncertainty [133].

Probabilistic tractography algorithms can represent the uncertainty of these methods. The algorithms run thousands of samples in a Monte Carlo (i.e. independent) fashion, selecting a fibre orientation from a distribution at each local voxel. Having enough samples, the algorithms converge to a solution, where a map with the spatial distribution of a pathway arising from a region is generated.

Despite the benefits of probabilistic tractography, the algorithms can be computationally very demanding. Thousands of seed points may be used for studying different brain connections, and thousands of samples from each seed point may be required by the algorithm to converge. Algorithm 6.1 summarises the main steps of a probabilistic tractography algorithm. Moreover, recent tractography applications, such as the generation of dense connectomes [139], and the estimation

of cortico-cortical connectivity [206, 207], add substantial computational complexity. For instance, a connectome from a single subject using high-resolution data from the Human Connectome Project [2–4] can take many days, up to weeks, to be computed using a non-parallel design.

Algorithm 6.1 Pseudocode of a probabilistic tractography algorithm. Millions of streamlines may be propagated in the method. The streamlines select at propagation step a fibre orientation from a distribution. These orientations are defined by two angles, θ and φ that represent the spherical polar coordinates.

```
1: for seed = 1 to D do
2:     for sample = 1 to F do
3:         Initialise streamline location at seed coordinate
4:         while stop_criterion is not met do
5:             Select random sample (θ, φ) from orientation distribution
6:             Propagate certain distance along (θ, φ)
7:         end while
8:     end for
9: end for
```

Here, we study the parallelisation of probabilistic tractography algorithms using GPUs. The integration of non-local information makes this approach completely different to the voxel-wise fitting problems presented in the previous chapters, and introduces new challenges that need novel treatment and specialised solutions. Contrary to the model fitting applications, where several routines are included, probabilistic tractography applications normally include a core routine, the propagation of millions of streamlines. This propagation process is independent across streamlines, and thus, in principle, its computation can be parallelised creating several execution threads. Given the high amount of independent threads, the application seems very suitable for being parallelised on GPUs, however a number of factors make an application like probabilistic tractography far from ideal for implementing on GPUs. These factors include:

- Unpredictable behaviour of the threads that leads to divergences and uncoalesced memory accesses.
- Heavy tasks performed by the threads.
- Very high memory requirements.

In this chapter we show how we address these challenges and present a GPU-accelerated probabilistic tractography framework that, nevertheless, achieves extremely high speed-ups compared with CPU designs. Furthermore, the framework incorporates extensive functionality, including the possibility of imposing anatomical constraints using either volumes (NIFTI files) or surfaces (GIFTI files [208]), and generating dense connectomes. We show the benefits of using this extended functionality and we report the performance gains of the parallel probabilistic tractography framework. Finally, we validate the parallel application performing several tests and comparing the results to the ones obtained with a CPU-based design.

6.2 Designing a GPU-accelerated probabilistic tractography framework

The framework presented here is based on *probtrackx*, a tractography tool included in the FSL library [116, 209]. We extend the functionality of this application and we propose a GPU-accelerated solution.

6.2.1 Framework functionality

To fully characterise the design requirements, we first need to consider the supported functionality of our tractography algorithm. Our application includes the basic functionality for performing probabilistic tractography, including for instance options to set:

- The number of streamlines propagated from each seed point, i.e., the number of samples.
- The step length when tracking streamlines.
- The maximum number of steps of the streamlines.
- A curvature threshold angle, for avoiding unrealistic sharp pathways.
- Tracts loops detection.
- Modified Euler's method (order two Runge-Kutta method [23]).

The framework uses two different coordinate spaces, diffusion space and an input/output space. Diffusion space is where the propagation of the streamlines takes place. Input/output space, commonly MRI standard space, is the space where the user specifies the tractography protocol (seed masks, anatomical constraints, etc). The application generates the results in the input/output space and can perform linear or non-linear transformations for converting coordinates from one space to another.

The user can specify the seed points using volume masks of voxels (i.e. NIFTI images), or using surfaces (GIFTI files [208]). Anatomical constraints can also be defined with volume masks and surfaces. Different kinds of constraints are allowed:

- Termination masks. If a streamline visits a voxel/vertex of these masks the algorithm stops the streamline.
- Exclusion mask. If a streamline visits this mask the streamline is discarded.
- Inclusions masks. The algorithm discards the streamlines that do not visit these masks.
- Classification masks that can be used for studying individual connectivity between regions.

Connectome generation is also inherently supported [139]. Three options are available (see Figure 6.1) [206, 207]:

- Connectivity matrix between all the seed points and all the other seed points. A typical use is for studying the connectivity from all grey matter to all grey matter [143].
- Connectivity matrix between all the points of two different masks, which are independent of the seed points mask. A typical use is for studying the connectivity between grey matter regions, when seeding from all white matter.
- Connectivity matrix between all the seed points and all the points specified in a different mask. A typical example is to use the whole brain as a target mask for studying the connectivity profile of the grey matter in a specific seed, and use it for connectivity-based classification [135].

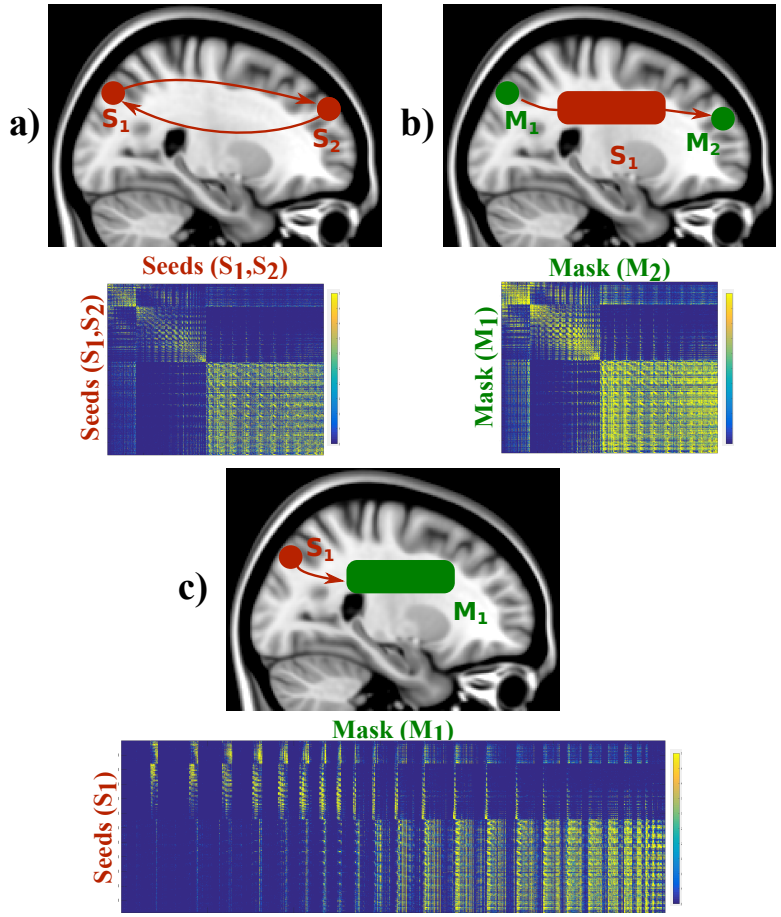

Figure 6.1: Connectivity matrices modes offered by the GPU-accelerated tractography framework. The framework can generate connectivity matrices from a) all seed to all seed points, b) all points in a mask to all points in another mask seeding from an independent region, or c) all seed points to all points in a different mask.

The masks used to specify the elements of the matrices can be defined with volumes and/or surfaces.

6.2.2 Surfaces for performing tractography

We include in our parallel framework the possibility of using surfaces for defining seed points and regions of interest (ROI). We implement the GIFTI format [208], according to which surfaces are defined by meshes of triangles, and each triangle is composed of three vertices. Three spatial coordinates define each vertex in a 3D space.

Surface vertices can be used for defining seed points, and mesh triangles can be used for defining stopping/constraint masks. If the latter, a streamline needs to be checked upon crossing the surface meshes. In our framework, we implement the method described in [210] (ray-plane intersection) for checking if the segments of a streamline intersects a triangle. We present this method in detail in Appendix 6A.

Surfaces have a main advantage over the use of volumes of voxels. A ROI specified with surfaces can capture an irregular shape in a more compact way, while voxels need to follow a regular grid and cannot easily follow irregularities, such as cortical convolutions, unless the grid has a really high resolution. Therefore, using surfaces, brain structures and interfaces with complex shapes can be defined more accurately at the same resolution as volumes. This is particularly important for dense connectome generation, as compact descriptions of regions and shapes can substantially reduce memory and storage requirements.

6.2.3 GPU parallel design and strategies

The streamline propagation process in this tractography algorithm is completely independent across streamlines, and thus it can be parallelised. The main idea behind the implementation is simple, we create as many CUDA threads as the number of streamlines. In fact, for the majority of cases, we need to propagate each streamline towards both directions indicated by the fibre orientation at a seed location. Thus, for each streamline we create two CUDA threads, having a total of $2 \times D \times F$ threads, where D is the total number of seeds points and F is the number of streamlines per seed (see Figure 6.2). Nevertheless, there are complexities that make the implementation of this idea challenging, and reduce efficiency if not addressed, as we see in the following sections.

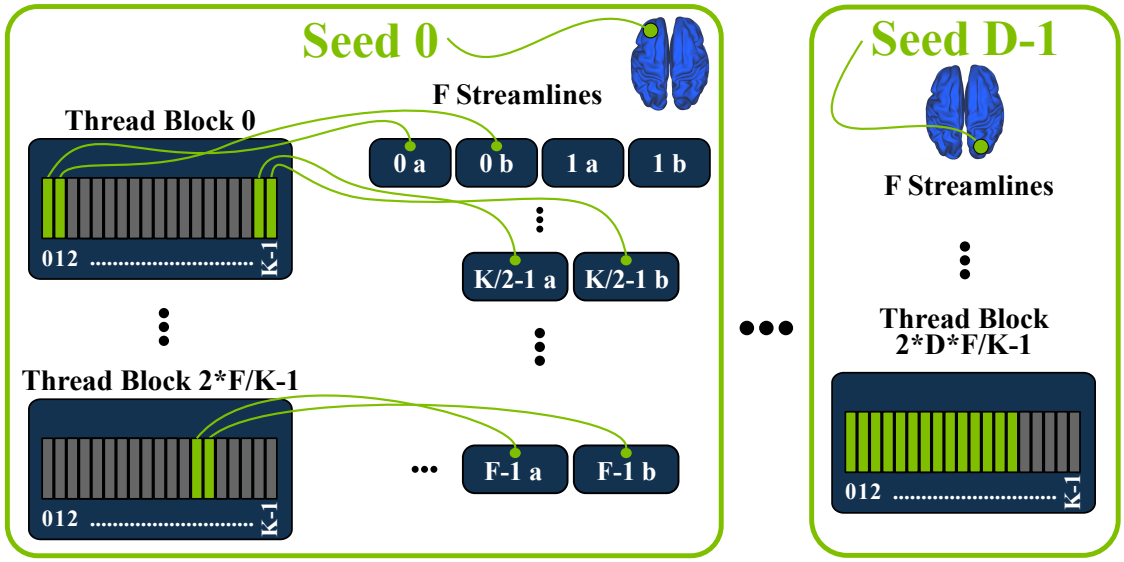

Figure 6.2: GPU parallel design of a probabilistic tractography framework. For each of the D seeds and for each of the F streamlines per seed, we create two CUDA threads (a and b), which are distributed amongst blocks of K threads.

Heavy Tasks

A first consideration in our GPU parallel design is the high complexity of some of the routines included for implementing the offered functionality. For instance, the use of surfaces involves the execution of an intersection detection algorithm, while the streamline propagation includes interpolation and space transformation routines. Having a single CUDA kernel for performing all these tasks leads to substantially heavy threads, which consume a lot of computational resources and consequently cause low occupancy in a SM.

Thus, we split the application into multiple tasks that are implemented in different CUDA kernels. Specifically, we define the following:

- A kernel that propagates the streamlines and checks the basic termination criteria, including a curvature threshold, an anisotropic diffusivity threshold, loop detection and out of brain detection.
- Several kernels, for checking the anatomical constraints masks. The application implements a different CUDA kernel for each type of anatomical constraint, i.e., a kernel for checking the stop masks, a kernel for checking the exclusion

masks, a kernel for checking the inclusion masks and a kernel for checking the classification masks.

- A kernel for updating the path distribution map. It uses the streamlines coordinates for updating each visited voxel in a unique and shared volume of voxels located in the GPU global memory.
- A kernel for generating a dense connectome. This optional kernel checks if any of the elements of the volumes and surfaces of the connectivity matrix is visited. It checks all the streamlines coordinates. The connectivity matrix can be very large ($\sim 30\ GB$ in the case of using HCP greyordinates [143]) and cannot be allocated on the GPU memory. Thus, each thread of this kernel stores in a vector only the ID of the visited matrix elements. The vectors are transferred from the GPU to the host, which finally, executes a routine for updating a general common matrix allocated in the host memory.

For executing all these kernels, we create two threads per streamline (streamlines are propagated towards two directions). The pipeline of the execution is depicted in Figure 6.3, where each kernel runs one after the other. The locations (coordinates) of the streamlines, obtained in the propagation kernel, need to be stored for being used in the other kernels, which increases the memory requirements (3 floats per coordinate). However, this is not particularly problematic compared to the heavy task that the framework would have to perform using a single kernel. With the proposed design we achieve much lighter CUDA threads, relieving pressure on the number of used registers per thread.

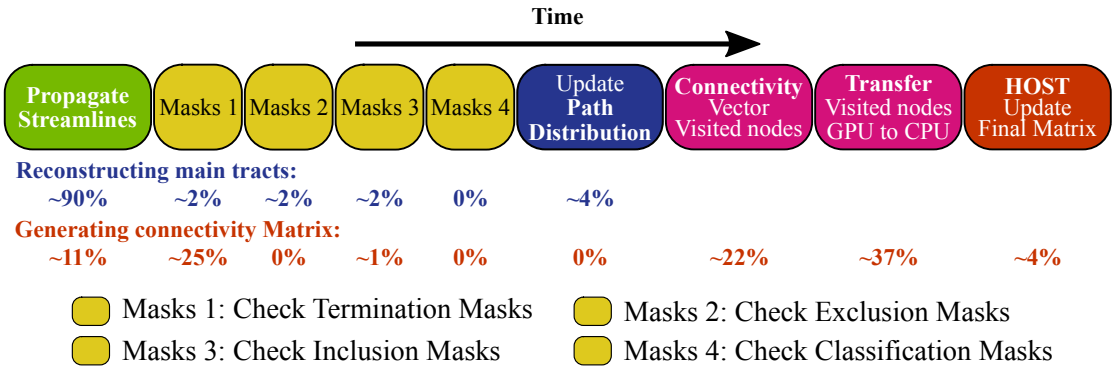

Figure 6.3: Pipeline of the GPU-accelerated probabilistic tractography framework. The routines that implement the different functionality of the application are split into different CUDA kernels. The connectivity matrix is stored in the host memory, and the GPU threads only store the visited nodes of the matrix (in a vector). The host needs to execute a final process for generating and updating the connectivity matrix. The figure shows the percentage of time spent on each kernel when typical tracts are reconstructed, and when a grey matter to grey matter dense connectome is generated. In both cases, not any classification mask was used, and for generating the connectome, not any exclusion mask was employed and the path distribution was not generated, thus, 0% of time was spent on these kernels.

Memory requirements

Another challenge in the GPU design is related to memory requirements. In fact, it may be impossible to use GPUs if the application memory demands exceeds the available memory in the device, and this is true across all levels in the GPU memory hierarchy.

Probabilistic tractography algorithms need in each voxel a distribution of fibre orientations for sampling. The algorithm cannot predict in advance the streamline track locations, and thus the fibre orientation distributions of all the voxels need to be allocated in memory. The amount of required memory depends on the size (spatial dimensions) of the dataset and on the number of samples in the orientation distributions. For instance, the required memory for simply allocating the samples of a Human Connectome Project [2–4] dataset is approximately 1.5 *gigabytes* (after applying a brain mask).

Moreover, our parallel application needs to store the 3D streamline coordinates, but it cannot predict in advance the amount of steps that a streamline will take,

i.e., it cannot predict when a stop criterion will be meet. Thus, memory is allocated for storing the maximum number of coordinates of each streamline. Additionally, volumes and/or surfaces may be used for defining seeds, anatomical constraints and connectome matrix elements, which also need to be allocated in memory.

In our strategy, the framework allocates all the required memory in the GPU without considering the coordinates, and then, it calculates the maximum number of streamlines that can be computed in parallel. If all the requested streamlines cannot be computed in parallel (which is the most common scenario) the application iterates over different streamlines groups.

All this data is allocated in global memory, which leads to high latencies every time the threads access to it. To mitigate these latencies, we use as much as possible efficient memory spaces, in this case Shared and Constant memory. We store in Shared memory the most common accessed data (segments coordinates, segments direction and selected fibre samples). However, the streamlines do not share information between them, and thus, each thread can store in Shared memory only a small portion of data. Some static data, such as space transformation matrices and warps, is stored in Constant memory, which is cached in the Constant cache, leading to lower latencies.

When a dense connectome is generated, the application needs to transfer thousands of vectors with visited elements from the GPU memory to the host, which can significantly degrade the performance of the application. For optimizing the memory transfers between host and device, we allocate the data in the host as *pinned* memory, which force the data to be page-locked. In other words, the data is allocated in consecutive memory locations, and thus, transferred faster to and from the GPU. It is more expensive to allocate this memory, but once is allocated the transfers become faster.

Thread Divergence: A solution using OpenMP and CUDA

One fact that limits the performance of the application is the divergence between threads. The streamlines may be initialised at different seed points and they can choose different samples in each propagation step. Thus, they take different paths causing accesses to different memory locations when sampling, i.e. uncoalesced memory accesses. Furthermore, streamlines may meet stopping/termination criteria at different time points. This causes asynchronous termination and a possible waste of computational resources, as some threads finish their execution before others and stay idle consuming GPU resources. This situation persists until all the threads of the same block finish their execution and the GPU scheduler switches the block with another block. For this reason, we set the block size K to a small size of 64 threads. Although NVIDIA Kepler GPUs only achieve 50% of occupancy with this configuration (because there is a limit in the number of blocks per SM), there are fewer divergences than having larger blocks, and as we explain below (Figure 6.5), the GPU can employ the unused resources (the other 50%) by overlapping other tasks/kernels in parallel.

To understand better the magnitude of the problem, we looked closer at the thread divergences in the streamline propagation kernel. Figure 6.4 shows the distribution of propagation steps taken when reconstructing some example tracts, as in [130], and when generating connectivity matrices using different brain regions as seeds. In many scenarios, many generated streamlines terminate relatively fast (before 100 steps), as they meet a termination criterion. However, there are cases, where a significant percentage of streamlines take twice or even three times as many steps (more than 200 steps in these examples).

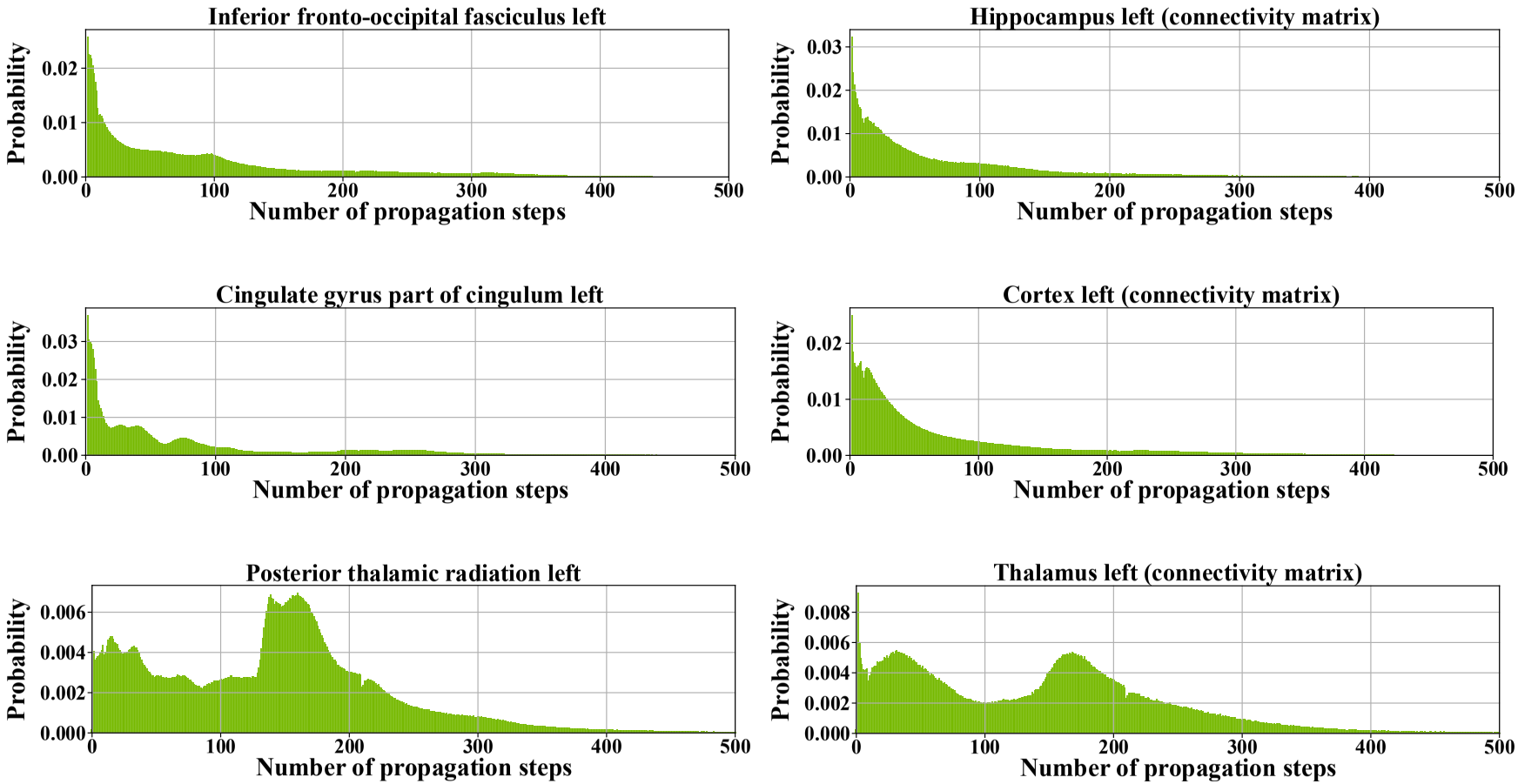

Figure 6.4: Divergence in the number of propagation steps in probabilistic tractography. The figure shows the probability that a streamline takes certain number of steps before meeting a termination criterion. Probability is shown when the application reconstructs three different tracts in the left hemisphere as in [130] (left), and when the application generates a connectivity matrix using three different regions as seeds (right).

A trend that can be extracted from the above examples is that streamlines are more likely to terminate after a small number of propagation steps than using the maximum number of steps allowed. This trend has been also confirmed before [170, 211] and it is also supported by the underlying anatomy: the majority of white matter connections are short in length [207]. We follow and assess two strategies to mitigate the thread divergences caused by this situation.

First, we modify the streamline propagation kernel for being able to stop its execution at a certain number of steps, remove the streamlines that have terminated and re-launch the kernel with less streamlines, as previously suggested by [170, 211]. When the kernel is launched again, only threads with an alive streamline will compete for the GPU resources, and thus, these resources will be used more efficiently.

Second, we take advantage of our pipelined design for overlapping the computation of different groups of streamlines on the same GPU. CUDA offers the possibility of running concurrently several tasks (kernels execution and/or CPU-GPU memory transfers) on the same GPU using different CUDA *streams* [82], which are queue instances for managing the order of execution of the tasks. If a CPU thread uses asynchronous CUDA commands (non-blocking), the thread can send tasks to different CUDA streams (see Figure 6.5), which may be executed concurrently on a GPU if it has enough free resources. In a single CUDA stream, tasks are executed serially, but the order of execution of the tasks from different CUDA streams is undetermined.

The GPU-accelerated framework divides the streamlines into a few groups. A CPU thread can send asynchronously tasks to the GPU, but for each group of streamlines it may need to update the connectivity matrix on the host (if a connectome is required), where the thread will need to perform computations before continuing sending asynchronously the next group of tasks to the GPU. Therefore, in this case, a single CPU thread cannot be used for running concurrently tasks on the GPU. Instead, we use several OpenMP [37] threads, where each one executes a pipeline on different CUDA streams.

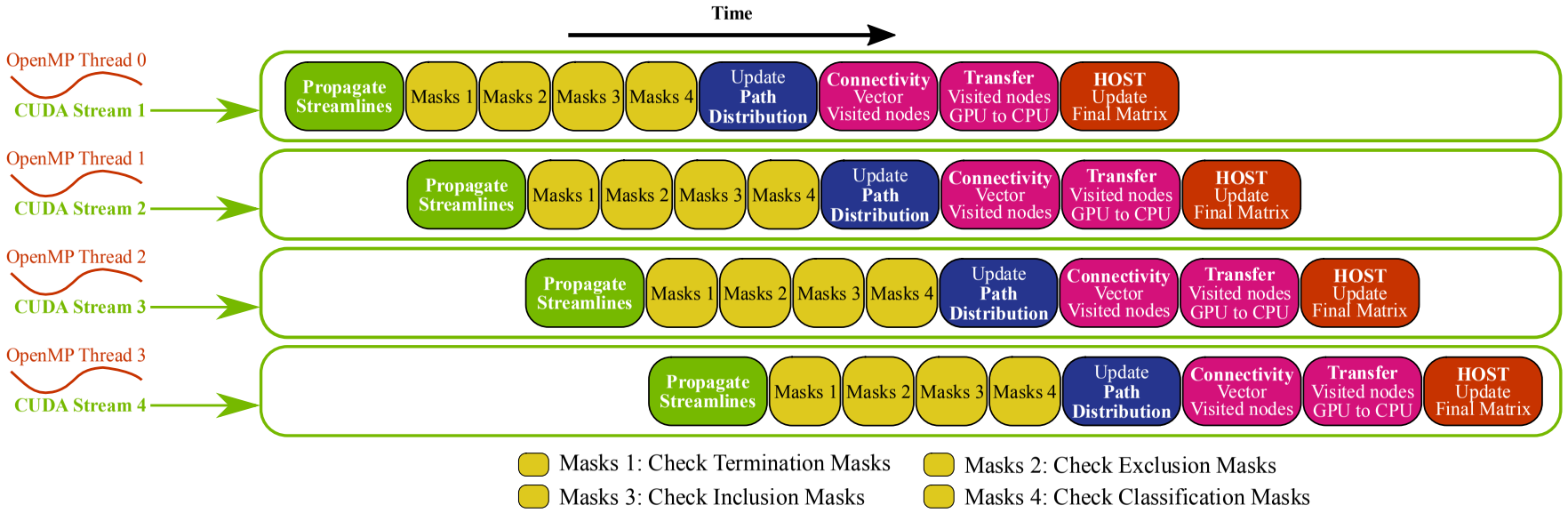

Figure 6.5: Overlap of several pipelines in the GPU-accelerated tractography framework using several CUDA streams and OpenMP threads.

We test our application using these two strategies, with different combinations of number of CUDA streams (and OpenMP threads), and different number of stopping steps for removing terminated streamlines. For the removing strategy we compare the original configuration (without stopping the kernel) to three other configurations, which are based on the uniform and non-uniform space-between-stops strategies proposed in [170], but in our case, we use less kernel stops given the computational cost of reorganising the alive threads at each stop:

- **Strategy A – no stops**: the kernel runs all the propagation steps without stopping until all the streamlines have terminated (original configuration).
- **Strategy B – 9 uniform stops**: the kernel stops 9 times every 45 propagation steps for removing the terminated streamlines.
- **Strategy C – 8 non-uniform stops**: the kernel stops after 5, 20, 45, 85, 145, 225, 325 and 425 propagation steps for removing the terminated streamlines.
- **Strategy D – 5 non-uniform stops**: the kernel stops after 20, 85, 155, 275 and 415 propagation steps for removing the terminated streamlines.

At the same time we tested the application with 5 different number of CUDA streams, 1, 2, 4, 6 and 8. Figure 6.6 shows the execution times reconstructing several tracts, as before, and generating connectivity matrices.

It seems that the stopping strategy barely affects the execution times (or does so in an inconclusive manner across the different examples), whereas the use of more CUDA streams, in general, reduces the execution times. When connectivity matrices are generated, the application is on average 30% faster using 8 CUDA streams than using a single one. This is because the memory transfers from the GPU to the host (transferring the vectors with the visited elements of the matrices) and the update of the connectivity matrix (host process) are overlapped with the execution of CUDA kernels using different CUDA streams, which optimise the performance.

When reconstructing tracts, using 4 CUDA streams made the application 12% faster on average than using a single one. The use of more than 4 CUDA streams

did not reduce further the execution times. In this case, all the tasks involve a kernel execution (there is not any memory transfer or host process), thus, the GPU does not have enough free resources to process more tasks concurrently.

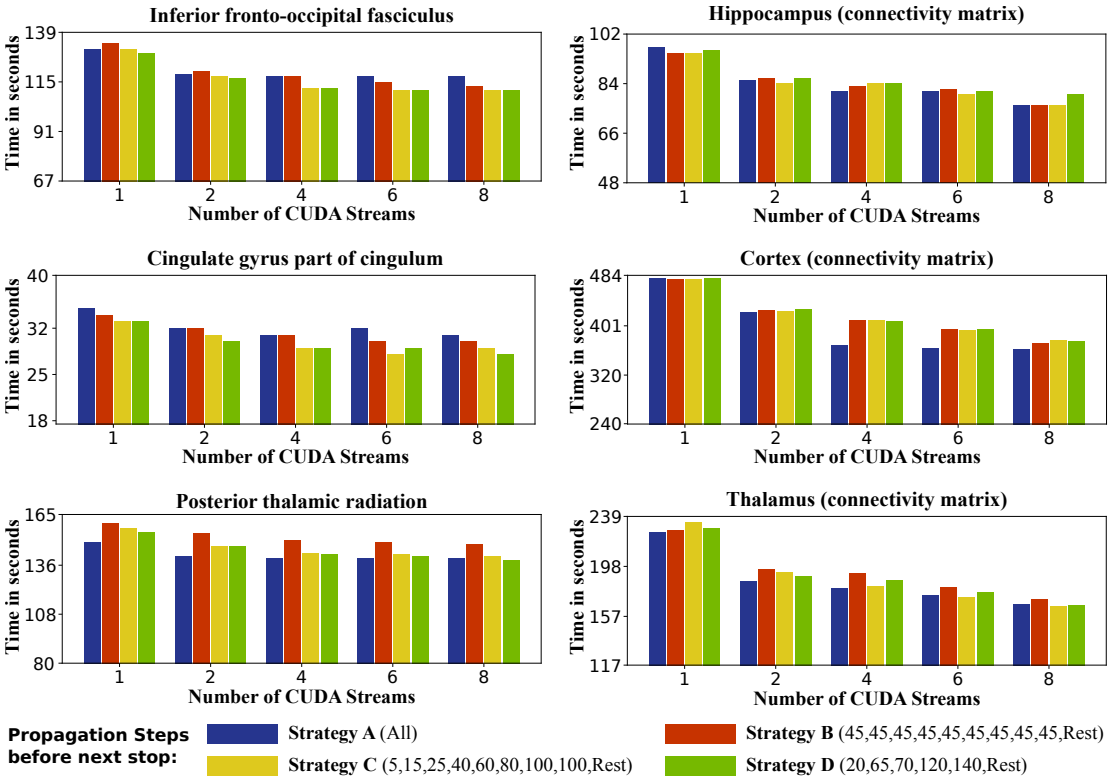

Figure 6.6: Execution times of the GPU tractography framework using different strategies for mitigating the thread divergence issue. The propagation of streamlines kernel is stopped after certain number of steps and the terminated streamlines are removed before continuing the execution. The application is tested with different number of CUDA streams. Note that in all the plots, the y-axis lower limit is $Max_{time}/2$ for better visualisation.

The stopping strategy barely reduces the execution times, and in some cases, it even increases them. The kernel needs to be executed until the last streamline finishes, as in the original version. The difference when using this strategy is that the number of blocks competing for the resources is reduced at every stop, and thus, these resources are available for executing other tasks. However, the cost of stopping the kernel, and removing and reorganising the alive threads seems to cancel out this benefit.

To summarise, the best configurations, and the ones that we use, are strategy A with 8 CUDA streams for generating dense connectomes, and strategy D with 4 CUDA streams for reconstructing tracts.

6.2.4 Implementation considerations

In this section we present some implementation details that we consider relevant for the performance achieved by the GPU framework and the correct execution.

Surfaces

When surfaces are used for defining anatomical constraints, the application needs to check if a streamline crosses any triangle of the surfaces. A line-segment-triangle intersection algorithm is used [210], where the segments are defined by each two spatial coordinates of a streamline, i.e., every time a streamline is propagated a new segment is defined. The application needs to check all the segments of each streamline, which is computationally very expensive, but we implement a localisation-based method for reducing the number of triangles to check. Before starting the tracking, the method identifies the triangles contained, or partially contained, in each voxel. A voxel-wise map with triangle indices is created associating voxels and triangles (see Figure 6.7). During the tracking process, the segment of a streamline is contained inside one or few voxels. Thus, only the triangles that are associated with those voxels need to be checked.

Interpolation

There are two operations in our tractography application where interpolation is performed. First, trilinear interpolation is used for non-linear transformations between diffusion space and input/output space. Fortunately, GPUs are very efficient performing interpolation operations. These devices have been designed for graphics applications where imaging processing operations, including interpolation,

are very common. NVIDIA GPUs has a dedicated memory for storing textures, which resides in global memory and has high latencies. However, accesses to this memory are cached in the on-chip Texture cache (see Figure 2.9) and operated by the texture units (see Figure 2.7). This cache has been optimised for spatial locality memory accesses patterns, and we use it for performing trilinear interpolation.

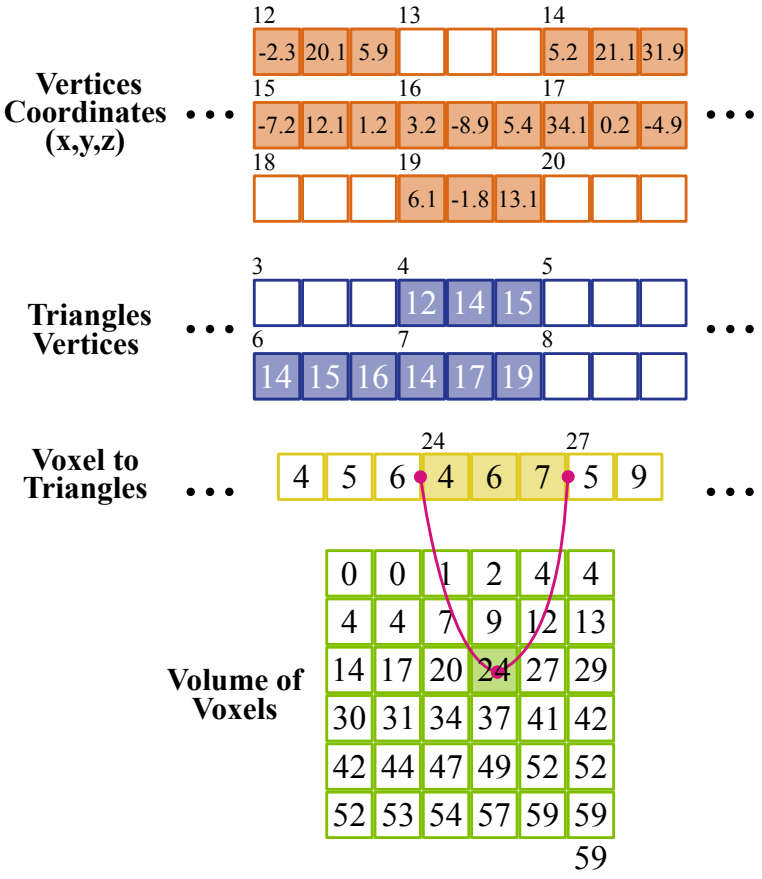

Figure 6.7: Voxel-wise map with the association between voxels and surface triangles. Each voxel has associated a set of triangles IDs. Triangles are defined by an ID, and 3 vertices, which contain 3D spatial coordinates (x, y, z). In this example, the green highlighted voxel contains three triangles, 24^{th} to 27^{th} in the associated list, with IDs 4, 6 and 7, each comprising vertices $(12, 14, 15)$, $(14, 15, 16)$ and $(14, 17, 19)$ respectively. The coordinates of each vertex are shown at the top

Interpolation is also performed during the tracking process for sampling orientations and propagating a streamline. Our framework implements a *probabilistic interpolation* method. Given the current spatial position of a streamline track, and a volume of voxels as reference, the probability of sampling from each voxel of the neighbourhood is inversely proportional to the distance of the spatial position to

the centre of the voxel. In other words, the closer the track position is to a voxel centre, the more chances there are to sample from that voxel.

Loopcheck

We implement an algorithm to avoid the presence of loops in the generated streamlines following the CPU-based implementation [116, 209]. A way for detecting a loop is to check at every new step if the streamline has already visited the voxel at the current position, and in addition, whether the direction of the tract is parallel (or within low angle) to the direction of the tract when previously visiting the same voxel. If these two conditions are met, tracking stops, considering that the streamline is moving into a loop trajectory. A lower resolution (e.g. $\times 5$ lower) voxel map can be used for comparing the different positions visited by the streamline, so that the loop detection sensitivity is higher.

This method is memory demanding, as each streamline needs to have a private history-map. In our GPU framework, instead of storing the whole map, a vector with keys that represent the positions of visited voxels (coordinates-based) is used. This strategy reduces significantly the memory requirements, but in each step, the whole vector needs to be checked.

Atomic Operations for updates

The application generates a path distribution map, and optionally, a connectivity matrix. The path distribution map is updated on the device by all the CUDA threads. The connectivity matrix is updated on the host by different OpenMP threads. In both cases, these updates must be performed sequentially in order to avoid race conditions. We use atomic operations in CUDA and OpenMP for updating values in the path distribution map and the connectivity matrix respectively.

6.3 Results: Anatomical Constraints in Tractography

6.3.1 Using pial surfaces

A useful feature that we have added to our tractography framework is the possibility of using surfaces for imposing more accurate anatomical constraints. For instance, we can use a pial surface as termination mask for avoiding the propagation of streamlines outside this surface, and avoiding non-plausible connections along the CSF in the subarachnoid space. As shown in the Figure 6.8, surfaces allow us to describe accurately structures, even the ones with a complex shape such as the gyrus.

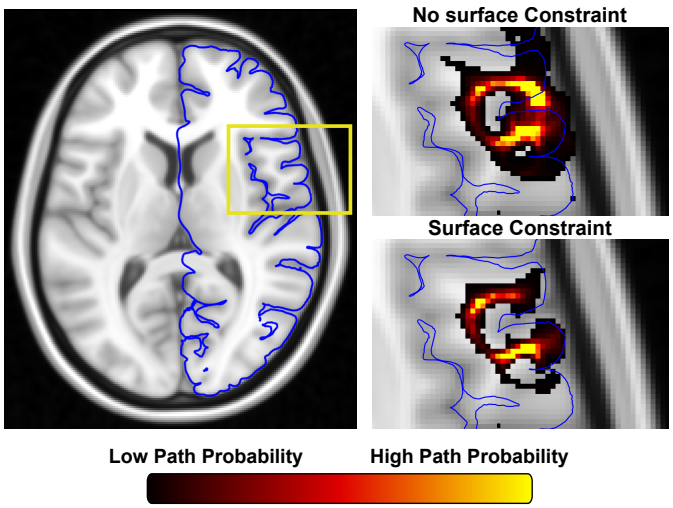

Figure 6.8: Example of the use of surfaces for imposing anatomical constraints. Probabilistic tractography is performed using as seed and target points the right inferior frontal gyrus. Without using a surface constraint, wrong paths that jump between neighbouring gyri can be generated.

6.3.2 Advanced termination anatomical constraints

A more sophisticated termination mask mechanism has been added to the GPU framework. Commonly, termination masks force the algorithm to stop the propagation of the streamlines the first time they hit the mask, but sometimes it is reasonable to allow the propagation until certain conditions are met. For instance, to increase the chances of getting "direct" connections, it is desired that a streamline

crosses the WM/GM boundary no more than twice when reconstructing cortico-cortical networks, once at the starting point and once at the end point. However, it does not seem plausible to have pathways running in and out of the WM/GM boundary or in parallel along the cortex connecting several regions (see Figure 6.9). Thus, in our framework a special kind of termination masks can be used for stopping the streamlines when they cross a surface twice.

Similarly, to encourage direct cortico-subcortical connections, it is undesirable that a streamline visits several subcortical regions, but ideally we would like a streamline to be able to propagate within a subcortical region. As in [212], our framework can use the special termination masks for stopping the streamlines upon exiting these regions, while allowing propagation within them.

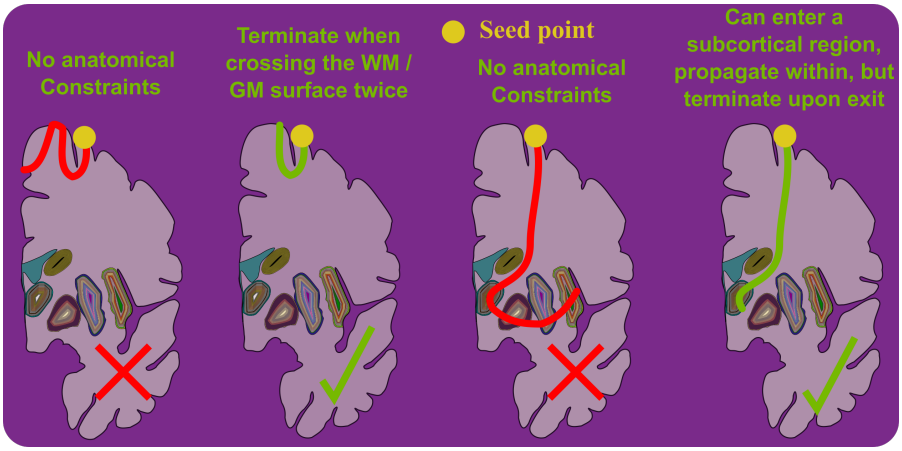

Figure 6.9: Advanced termination masks. The tractography framework adds the possibility of stopping the streamlines when they cross a surface twice and/or streamlines can be propagated inside a subcortical region but the framework stops them upon exit.

To illustrate an example, we have used advanced termination masks for generating a dense connectome. The masks are defined with a WM/GM surface and volumes that include several subcortical structures (accumbens, amygdala, caudate, cerebellum, hippocampus, pallidus, putamen and thalamus). Figure 6.10 shows a comparison of the results in the connectivity from the sensori-motor part of the thalamus with and without advanced termination masks. Without using this mode, a streamline from a ROI (Thalamus in this case) can visit several subcortical structures. Also, the streamlines can cross several times the cortex and continue

propagating, generating a number of false positives (for instance see hotspots along the medial surface). With the use of the advanced termination mask this situation is avoided, and a more realistic connectivity map is obtained, connecting the sensorimotor part of the thalamus to the sensorimotor cortical regions.

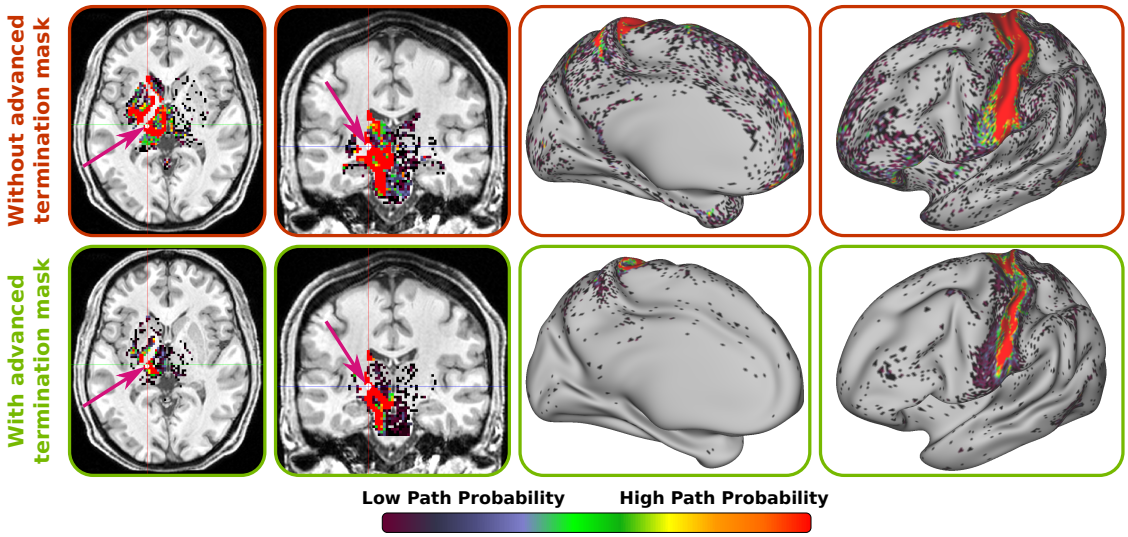

Figure 6.10: Connectivity from a voxel inside the left Thalamus (fuchsia arrow) using and not using advanced terminations masks. The first two columns show the connectivity with other subcortical structures. The last two columns show the connectivity with all the vertices on the left hemisphere cortex.

6.4 Results: Performance gains using the GPU implementation

We perform two different tests for assessing the performance gains of the GPU-accelerated probabilistic tractography framework. First we reconstruct 15 major white matter tracts as in [130], while in a different test, we generate dense connectomes using the HCP greyordinates (91K seed points [143]). We report the execution times and speedups achieved comparing the GPU-accelerated framework with a CPU-based tractography tool included in FSL [116, 209]. All the tests were executed 10 times and we present the execution times and speedups.

6.4.1 Reconstructing white matter tracts

We reconstruct 15 major tracts, with the GPU-accelerated framework and with the CPU-based tool, using the "autoptx" plugin protocols [130, 213]. Autoptx defines masks in standard space (MNI) as seeds, inclusion, and exclusion regions of interest for reconstructing a number of white matter tracts by performing probabilistic tractography. Table 6.1 summarises the number of seeds points and number of samples used for reconstructing each tract. Different numbers of streamlines are used for reconstructing the different tracts, which depends on their geometry complexity [130].

Most of these tracts have a bilateral homologue, a tract in the left hemisphere and a tract in the right hemisphere.

Figure 6.11 reports the processing times reconstructing 15 tracts (27 counting homologues) individually. A single CPU core was used for running the CPU-based framework, and a single GPU and four CUDA streams for processing the tracts with the GPU-accelerated framework. On average, a speedup of 238× was achieved, in the range of 80× and 357×. In all cases, except for the reconstruction of the Acoustic Radiation, the GPU-based application achieves accelerations of more than two orders of magnitude. In general, if the reconstruction of a tract involves several constraints that makes the algorithm to stop or discard streamlines at early steps, including tight termination and exclusion masks, the GPU-based framework performs worse, as these masks are not checked until the propagation of the streamlines has completely finished (see Figure 6.3). The reconstruction of the Acoustic Radiation uses a very tight exclusion mask and thus the achieved performance is lower compared with the reconstruction of other tracts.

	Number of seeds	Samples per seed	Number of streamlines	Left/Right
Uncinate fasciculus (**UNC**)	1,692	1,200	2,030,400	+
Medial lemniscus (**ML**)	1,926	1,200	2,311,200	+
Corticospinal tract (**CST**)	723	4,000	2,892,000	+
Anterior thalamic radiation (**ATR**)	3,181	1,000	3,181,000	+
Parahippocampal part of cingulum (**PHC CGH**)	1,887	3,000	5,661,000	+
Middle cerebellar peduncle (**MCP**)	2,075	4,400	9,130,000	−
Forceps major (**FMA**)	18,159	600	10,895,400	−
Inferior longitudinal fasciculus (**ILF**)	9,207	1,200	11,048,400	+
Forceps minor (**FMI**)	19,195	600	11,517,000	−
Superior longitudinal fasciculus (**SLF**)	32,831	400	13,132,400	+
Superior thalamic radiation (**STR**)	21,019	800	16,815,200	+
Cingulate gyrus part of cingulum (**CGC**)	1,137	20,000	22,740,000	+
Inferior fronto-occipital fasciculus (**IFO**)	15,412	4,400	67,812,800	+
Posterior thalamic radiation (**PTR**)	3,669	20,000	73,380,000	+
Acoustic radiation (**AR**)	23,105	10,000	231,050,000	+

Table 6.1: List of reconstructed tracts sorted by number of propagated streamlines. Different number of seed points and samples are used for reconstructing the tracts. Some tracts have a bilateral homologue (+) and some others no (−).

Figure 6.12 reports the total execution time reconstructing all the 27 tracts. When the CPU-based tool is used, the reconstruction of several tracts can be parallelised. Tracts are completely independent and thus their reconstruction can be processed by different threads. A total of 27 CPU cores were used in this case, using different CPU threads for reconstructing different tracts. A single GPU and four CUDA streams were used again for processing the tracts with the GPU-accelerated framework, processing sequentially the different tracts. A speedup of 26.5× was achieved using the GPU-accelerated solution.

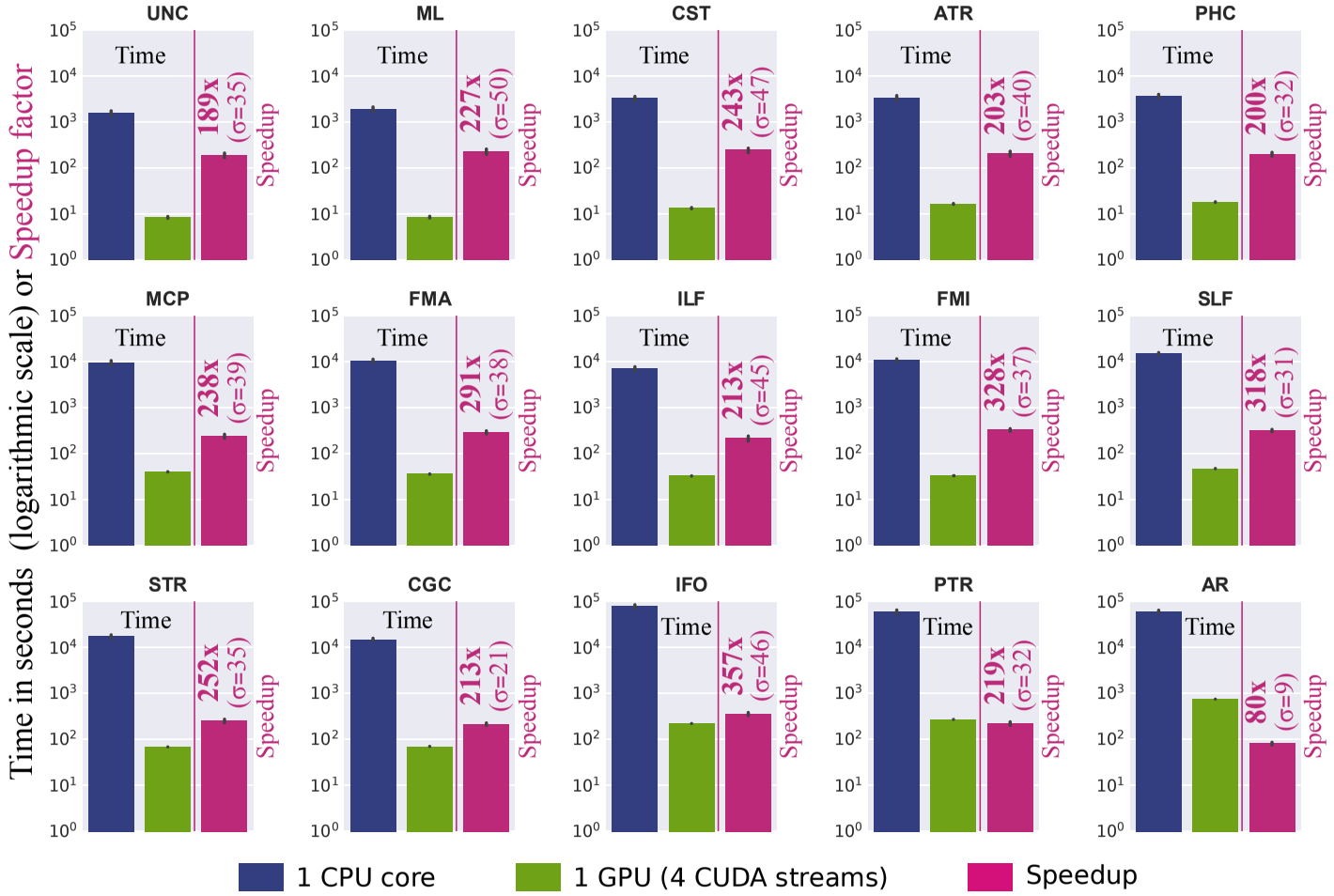

Figure 6.11: Execution times (in logarithmic scale) and speedup (standard deviation σ is also shown) in the reconstruction of 15 tracts comparing a GPU-based with a CPU-based probabilistic tractography framework. See Table 6.1 for tracts acronyms.

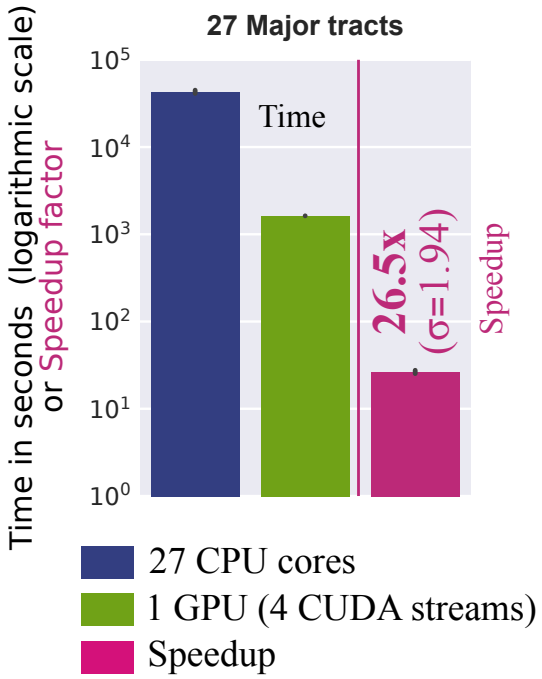

Figure 6.12: Execution times (in logarithmic scale) and speedup (and its standard deviation σ) reconstructing 27 tracts.

6.4.2 Generating dense connectomes

We use the CPU-based and the GPU-based frameworks for generating a dense connectome. 91,282 seed points and 10,000 samples per seed point were employed, having a total of 912.82 millions of streamlines.

For generating the connectome with the CPU-based application we used 100 CPU cores, each one propagating 100 streamlines from each seed point. This process took on average 3.38 hours. At the end of the process, the different generated connectomes (on different CPU cores) need to be added. This merging process took on average 6.5 hours (due to the size of the final connectivity matrices).

We used 1 single GPU and 8 CUDA streams for generating the connectome with the GPU-based application. The process took on average 2.95 hours.

Figure 6.13 reports these execution times and the speedup achieved by the GPU-based framework, with and without considering the merging process required

by the CPU multi-core application. Without considering the merging process both applications reported similar execution times. Considering the merging process, the GPU application was more than three times faster than the CPU multi-core application. In both cases, 1 GPU offers at least two orders of magnitude speed-up compared with a single CPU core.

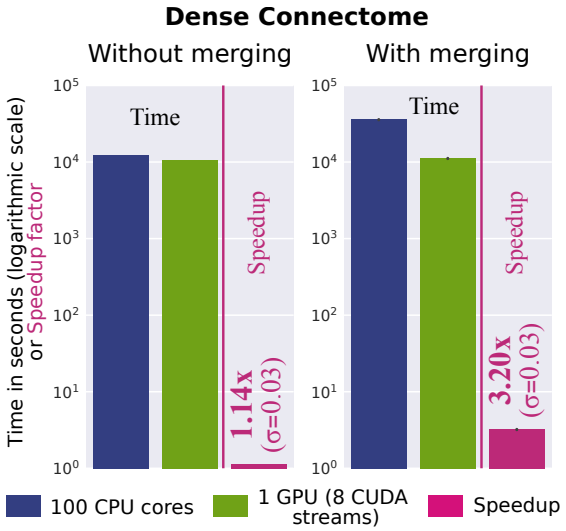

Figure 6.13: Execution times (in logarithmic scale) and speedup (and its standard deviation σ) generating a dense connectome.

6.5 Validation: Comparison of GPU to CPU designs

For validating the GPU-accelerated probabilistic tractography framework we performed various tests and we compared the results with the results obtained using a CPU-based tractography application [116, 209], for both white matter tract reconstruction and connectome generation. Given the stochastic nature of probabilistic tractography methods, we expect some variability in the results of both frameworks, but given the high number of samples that we have used in the tests, we expect this variability to be small. Nevertheless, we run every experiment 10 times and we compare the differences between CPU and GPU results with the run-rerun variability.

Figure 6.14 shows the run-rerun variability of each framework independently and the distribution of correlation coefficients between the CPU-based and the GPU-based frameworks. In the reconstruction of the 15 tracts the correlation was voxel-wise calculated. In the generation of the dense connectome, the correlation was calculated from all the elements of the connectivity matrix.

The individual run-rerun correlation coefficients are higher than 0.999 in all the cases, for both the CPU and the GPU frameworks. This indicates that the methods are very robust, and although there is some variability (there is not a perfect correlation), it is very low in both frameworks.

In all the cases, the correlation coefficients between CPU and GPU are higher than 0.998, although they are slightly lower than between the individual run-rerun results (CPU vs. CPU and GPU vs. GPU). This is expected, as some mathematical operations have different implementations (e.g. rounding modes) and different precision in a GPU compared with a CPU [169].

Figure 6.15 shows a qualitative comparison of the reconstruction of six exemplar tracts using both frameworks, and Figure 6.16 shows an example comparing the connectivity map of a vertex in the motor cortex extracted from dense connectomes generated with these frameworks.

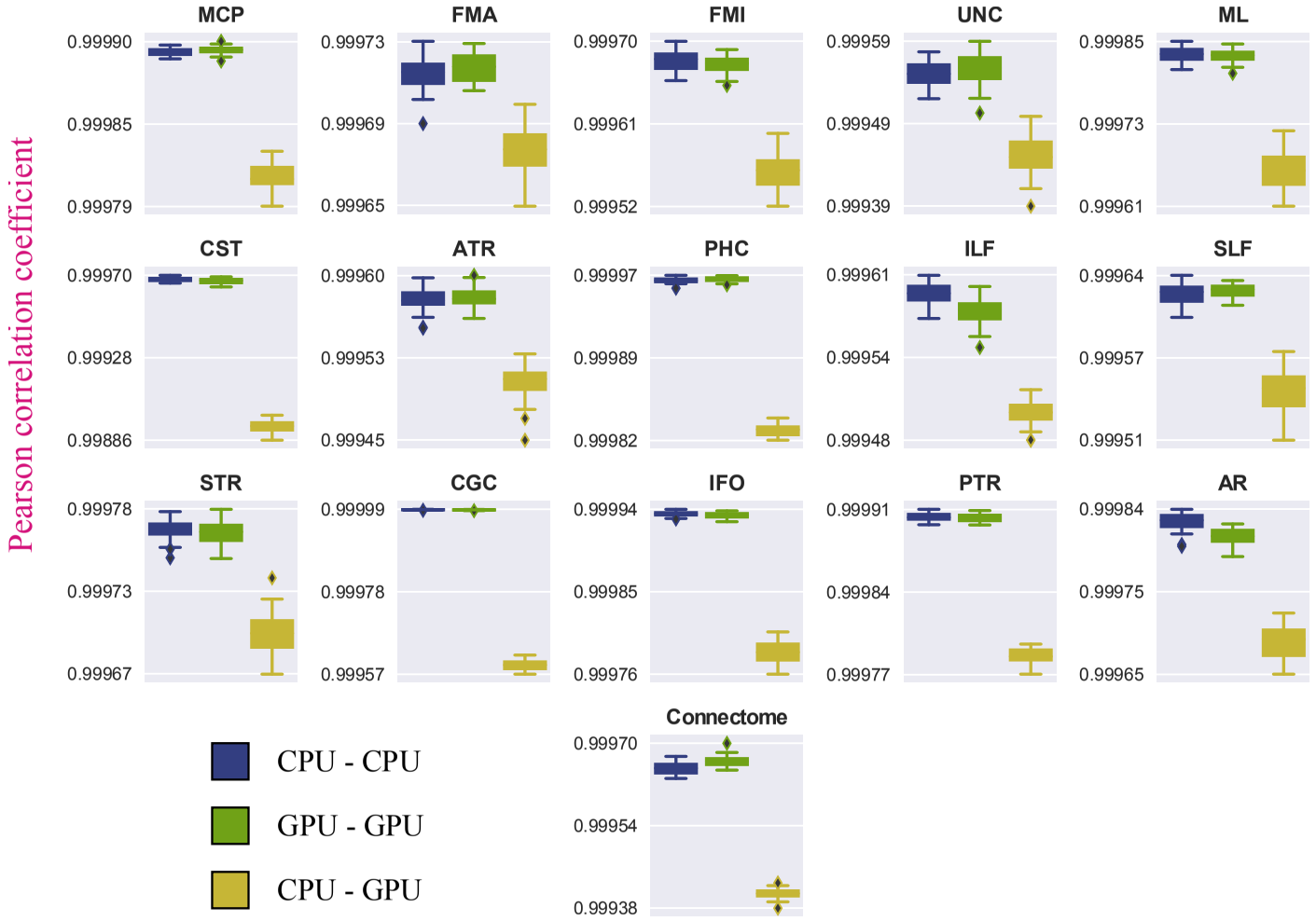

Figure 6.14: Run-rerun variability of CPU-based and GPU-based probabilistic tractography frameworks and distribution of the correlation coefficients between both frameworks. Results are showed in the reconstruction of 15 major tracts and in the generation of a dense connectome. Each experiment was run 10 times. The 45 combinations of correlations between re-runs were considered and 45 out of the 100 combinations of CPU vs. GPU correlation coefficients were chosen randomly.

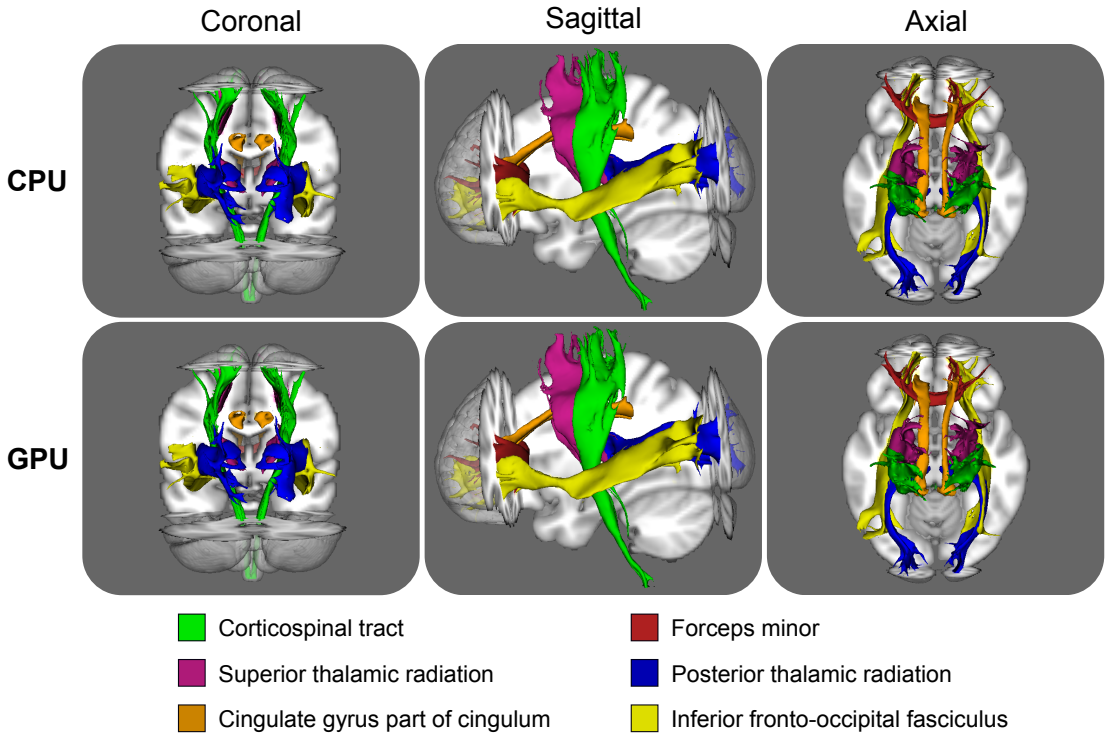

Figure 6.15: Coronal, sagittal and axial views comparing CPU-based and GPU-based frameworks performing probabilistic tractography and reconstructing some major white matter tracts. Each colour represents a different brain white matter tract. These paths are binarised versions of the path distributions after being thresholded at 0.5%.

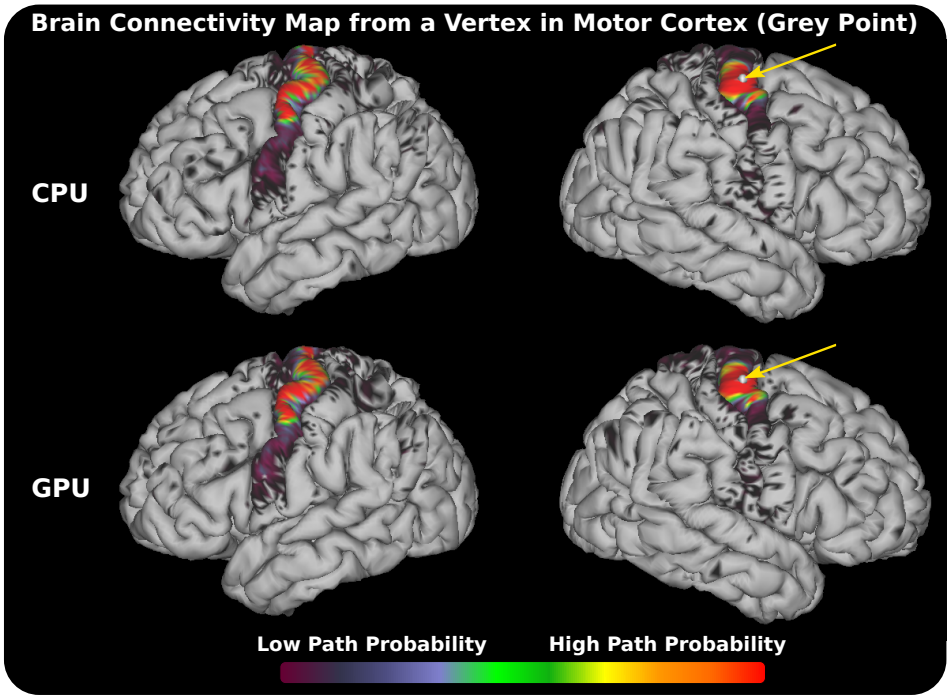

Figure 6.16: Path probability map from a vertex in the Motor Cortex. The map was extracted from dense connectome matrices reconstructed with CPU-based and GPU-based probabilistic tractography applications.

6.6 Discussion

We have presented a GPU-accelerated framework for performing probabilistic tractography that allows accelerations of two orders of magnitude, either in the context of generating white matter tracts or estimating whole-brain connectomes. Additionally, the implementation offers the possibility of defining tractography protocols with either volumes or surfaces. We have shown the benefits of using surfaces for defining structures with a complex shape, such as a pial surface as stopping criterion. We have also included the possibility of using advanced termination masks that allow more accurate anatomical constraints.

The parallelisation of these tractography algorithms on GPUs poses different challenges compared to the local voxel-wise modelling methods presented in the previous chapters. Each GPU thread propagates a streamline in one direction, integrating the local information of different voxels. A thread accesses to different memory locations during its execution, and these accesses cannot be anticipated, as the propagation movements are decided on the fly. Moreover, given the stochastic way for choosing the orientation samples, the threads may diverge even if their streamlines are initialised from the same seed point. This behaviour leads to uncoalesced memory accesses and unbalanced life duration of the threads, and consequently, to a waste of GPU resources.

We have explored two approaches for mitigating this situation. First, we used the approach presented in [170, 211], where we stop the propagation of streamlines after a certain number of steps and we remove the threads that are idle, i.e., the threads with terminated streamlines. We tried several strategies for deciding the number of steps to use before stopping the algorithm. The process for removing the idle threads is executed on the host and it has some extra cost that may cancel out the gains of this approach.

Contrary to suggestions in [170, 211], we could not find enough supporting evidence that this approach results in significant performance gains. We believe that

the extended and more complex functionality offered by our tractography framework, compared with the other tools, is the main cause of these differences. Given the complex functionality, more information needs to be stored and reloaded by the GPU threads every time that the CUDA kernel is stopped, causing an overload.

In a second approach, we divided the streamlines into groups, and overlapped several CUDA kernels (which compute these streamlines) on the GPU using CUDA streams. This approach is more successful, especially when the application generates dense connectomes. The processing time is reduced by 30% running 8 CUDA streams in parallel, which have to be managed by 8 OpenMP CPU threads.

Another challenge for having an efficient GPU-accelerated framework comes from the heavy tasks that are performed, especially when complex functionality is added to the application. Having a single CUDA kernel for performing all these tasks is very inefficient. Heavy threads consume a lot of GPU resources, and thus a low occupancy is achieved. Instead, we divide the tasks into subtasks, each one implemented in a different kernel. A main kernel propagates the streamlines, checking only the basic stopping criteria (curvature, out of mask, loopcheck and anisotropy), and subsequent kernels apply the anatomical constraints masks, generate a distribution map and a dense connectome (if required). This strategy leads to lighter kernels that increase the occupancy. Additionally, the pipelined execution makes this design more suitable and flexible for overlapping several kernels (and/or kernels with memory transfers) on the GPU with CUDA streams.

One of the most challenging requirements of the application is the large amount of required memory. Given the non-predictable path and length of the streamlines, all the orientation samples and memory for storing the maximum possible number of visited coordinates need to be allocated. The GPU global memory is used for storing this data. This fact restricts the number of streamlines that can be propagated in parallel. However, most modern GPUs have at least 3 GB of global memory, and they still can run a considerable number of streamlines ($\sim$ 40,000) in parallel. When the complex functionality of the application is used, such as

generating a dense connectome, this number can be reduced by 60% (as more data per streamline is needed) and GPUs with at least 5 GB should be used for achieving a good performance.

Another requirement is the need of a large amount of memory on the host for storing the whole connectivity matrix when a dense connectome is generated. On the GPU each thread stores only the visited elements of the connectivity matrix in a vector. These vectors are transferred to the host where the final matrix is generated. For improving the performance, we use Pinned memory on the host and we overlap the transfers with the execution of kernels. We use dense matrices on the host side for storing the connectivity matrix. This may require a lot of memory. In a greyordinates to greyordinates connectivity study, as in [143], a 91K $\times$ 91K matrix is required, i.e., more than 30 GB. Normally, this connectivity matrix is highly sparse. In the greyordinates connectivity example only 7% of the matrix are non zero elements. An implementation using a sparse matrix would relax this requirement, however, the insertion and update of its elements would be slower, and the framework may suffer from performance degradation.

We have assessed the performance gains of the GPU-accelerated probabilistic tractography framework. We compared the execution times of our framework to the execution times of a CPU-based tractography application [116, 209] reconstructing major white matter tracts, as in [130], and generating dense connectomes. On average, reconstructing the tracts, the GPU-accelerated solution achieves speedups of 285$\times$ compared with a single CPU core. Generating a dense connectome, on average, the GPU-accelerated solution achieves speedups of 3.2$\times$ compared with 100 CPU cores running the CPU-based application.

In the reconstruction of a specific tract, the acoustic radiation, the GPU framework performs relatively worse compared with the other tracts. The speedup achieved is 80$\times$ comparing a single GPU with a single CPU core. The anatomical constraints used in the reconstruction of this tract, specifically the tight exclusion and termination masks, make the streamlines to stop relatively quickly or to be discarded.

These are cases, where the GPU design will not perform at maximum efficiency. Figure 6.17 shows the path distribution obtained after running tractography and the masks used in the reconstruction.

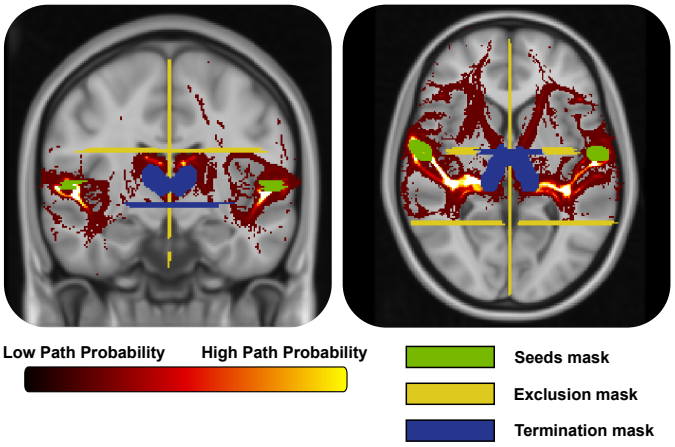

Figure 6.17: Reconstruction of the Acoustic Radiation. The figure shows the path distribution without using any threshold, the mask used for defining the seed points of the tracts and the exclusion and termination masks used for imposing anatomical constraints.

In the case of the CPU-based application the streamlines are discarded at early steps of the method, as the masks are checked after every single step. However, the GPU-based framework propagates the streamlines without checking the anatomical constraint masks. In subsequent and different kernels, these masks are checked, and a lot of streamlines are discarded. Thus, a large amount of the consumed resources are wasted. In the case of this tract, only 0.13% of the streamlines survive after applying all the exclusion criteria. The integration of the streamline propagation kernel with the exclusion mask kernel may improve this situation, but this would lead to heavy threads. Nevertheless, this is the only tract where this happens out of the examples we tested.

We validated the framework comparing the results of the reconstruction of 15 major tracts and a dense connectome with the results obtained using a CPU-based application. Almost identical results were obtained from both frameworks, and moreover, both showed a very low variability, which indicate a high robustness of the methods. We performed these tests running millions of streamlines, and in the case of the dense connectome, running almost a billion of streamlines.

As a final note, we evaluated the number of samples that are needed per seed point when generating a dense connectome in order to achieve convergence. To do that, we generated a dense connectome multiple times using a different number of samples per seed point. Figure 6.18 shows the correlation coefficients with respect to a "converged" dense connectome generated with 100,000 samples per seed. The figure also shows the correlation coefficients between consecutive runs in terms of number of samples per seed. It seems that even with 1,000 samples and 18 minutes run, the results are almost converged. Using 10,000 samples per seed achieves convergence, while the time for generating the connectome is still reasonable, less than 3 hours using a single GPU and 8 CUDA streams.

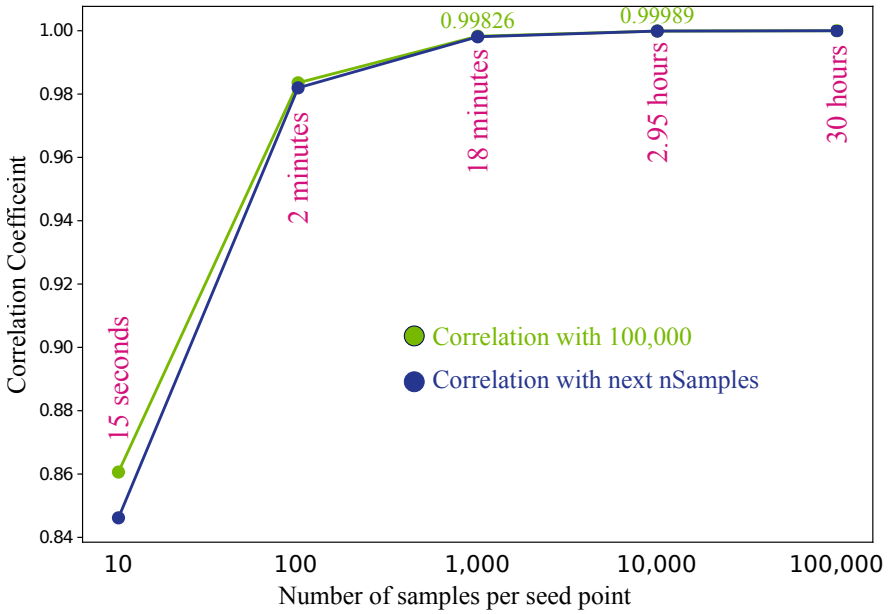

Figure 6.18: Correlation coefficient, generating a dense connectome (all greyordinates to all greyordinates) on the GPU-based framework, between re-runs, modifying the number of samples per seed point. The figure reports the correlation coefficients with respect to a dense connectome generated with 100,000 samples (green) and with respect to the connectome generated with the next number of samples in the plot (blue). The figure also shows the execution times generating these dense connectome on a single NVIDIA K80 GPU.

Appendix 6A

Streamline-surface mesh intersection

We have used the method in [210] to test when a streamline intersects a surface mesh. In particular, we consider the segment of a streamline and a close-by surface mesh triangle. The method uses a number of geometric elements (Figure 6A.1a), specifically:

- A streamline segment, defined by two 3D sets of coordinates at each propagation step, which are the spatial positions before and after propagating the streamline, and a vector indicating its direction, **s**.
- A surface mesh triangle, the plane that contains it and the normal vector to the plane, **n**.
- A distance vector **w** between the triangle and the segment. It is a vector from any of the vertices of the triangle to the origin point of the segment.

First, the method determines if the segment intersects the plane that contains the triangle. If **n** and **s are orthogonal** (dot product is 0), the segment is parallel to the plane. In this case there are two possible scenarios:

- If **n** and **w** are not orthogonal (dot product is not 0) the segment is not on the plane and there is no intersection (Figure 6A.1a).
- If **n** and **w** are orthogonal (dot product is 0) the segment is on the plane. Then, an intersection is assumed (Figure 6A.1b).

If the segment is not parallel to the plane, there are more possible scenarios:

- **n** and **s have the same sign** (with respect to the plane). In this case the segment is going away from the plane, so it will never intersect the plane (Figure 6A.1c).
- **n** and **s** have opposite signs. Then the segment may intersect the plane and:

– If the projection of **w** onto **n** is larger than the projection of **s** onto **n** ($\mathbf{w} \cdot \mathbf{n} > \mathbf{s} \cdot \mathbf{n}$) the segment does not reach the plane and there is no intersection (Figure 6A.1d).

– If the segment reaches the plane, the method calculates the intersection point, which is given by:

$$Intersection_{Point} = Segment_{InitialPoint} + \frac{\mathbf{w} \cdot \mathbf{n}}{\mathbf{s} \cdot \mathbf{n}} \mathbf{s} \tag{6A.1}$$

If the intersection point is outside the triangle, there is no intersection (Figure 6A.1e). Otherwise, the segment intersects the triangle (Figure 6A.1f).

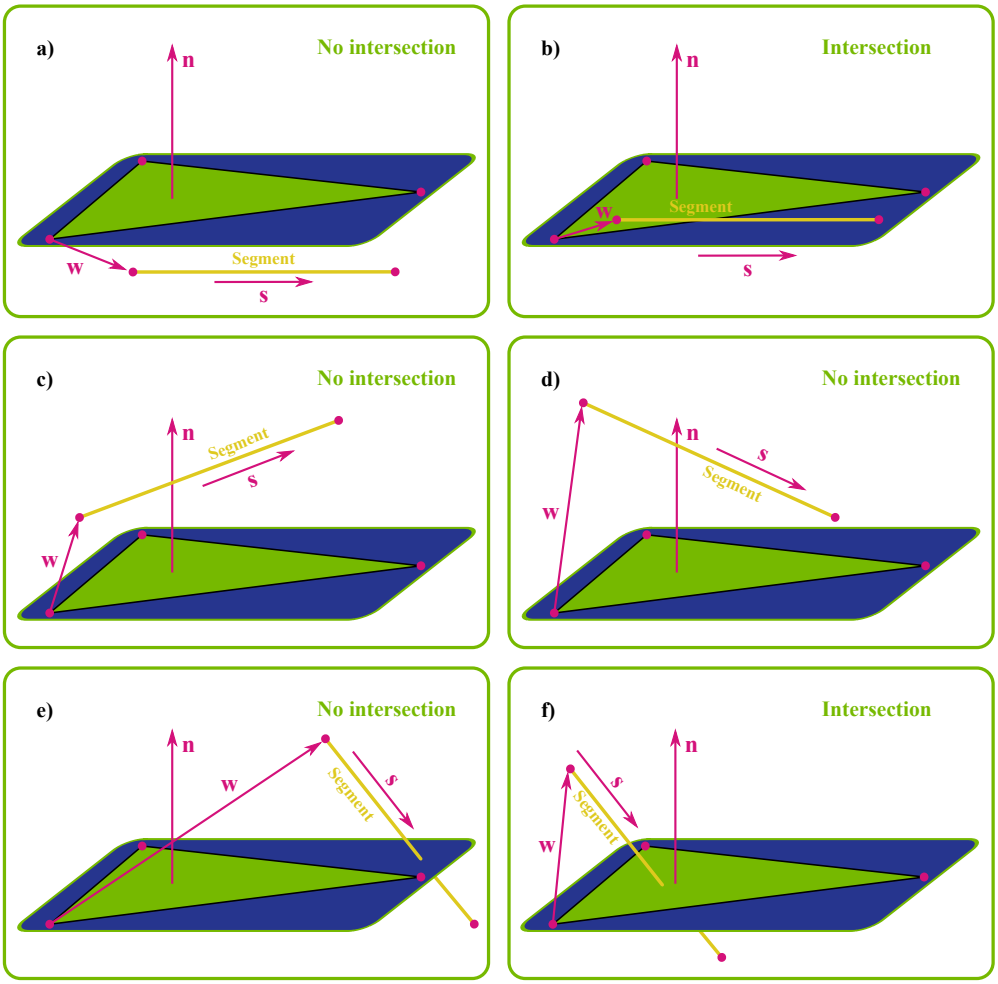

Figure 6A.1: Segment-triangle intersection scenarios. The tractography algorithm checks each scenario to evaluate if a streamline with direction **s** crosses the triangles of a surface. **n** is a vector normal to the plane of the triangle, and **w** is a distance vector from any vertex of the triangle to the origin of the segment.

7

Conclusion

Contents

7.1 Summary and conclusions 213
7.2 Future perspectives 217

7.1 Summary and conclusions

In this thesis, we have designed and implemented parallel computational frameworks for the analysis of diffusion MRI (dMRI) using graphics processing units (GPUs). We have illustrated the huge potential of using GPUs for scientific computing in a number of problems with different requirements and parallelisability challenges. By achieving accelerations of more than two orders of magnitude, GPU-based computations can change the perspective of what is computationally feasible in neuroscience research. This can have a direct effect on both model design (more demanding models can be built and fitted in realistic time frames) and model application (inherently slow applications can now run on significantly more datasets). Even if we focused specifically on the analysis of dMRI data, the generic principles shown in this thesis for parallel design and for overcoming computational challenges can be a guide for other applications. We have introduced in Chapter 2 the main aspects and strategies for designing and implementing efficient GPU-accelerated

solutions, and throughout the research chapters we illustrated how these can be considered and implemented in practical problems.

The applications that we considered focused on computational analysis of dMRI, which uniquely allows studies of brain microstructure (using biophysical models) and brain connectivity (using tractography methods). These approaches can be very computationally demanding (e.g. non-linear optimisation over many datasets and/or Bayesian inference), due to the inherently multi-dimensional nature of the data. This is particularly true when high-resolution (spatial and angular) data is acquired, or many datasets are analysed, such as in the Human Connectome Project (HCP) or the UK Biobank project. In Chapter 3 we gave an overview of the diffusion MRI theory, including an overview of voxel-wise dMRI models for resolving complex fibre configurations and an overview of tractography algorithms.

GPU-accelerated solutions for fitting non-linear models were presented in the first two research chapters. In Chapter 4 we presented several parallel designs for estimating the fibre orientations using a specific voxel-wise model, the ball & sticks model. Two fitting routines were parallelised using GPUs, Levenberg-Marquardt and Markov Chain Monte Carlo (MCMC). We presented a number of design choices aiming to exploit as much as possible the enormous performance potential offered by modern GPUs. Our preferred design, with multi-level parallelisation (between voxels and between data-points within a voxel), achieved accelerations of more than two orders of magnitude compared with sequential designs. We validated the GPU framework comparing it with a popular CPU-based application included in the FSL library. Our GPU tools have been widely used so far. They are freely available, distributed worldwide through FSL, and are being used in cornerstone projects, such as the HCP (data from 1,200 adults), the Developing Human Connectome Project (dHCP) (data from 1,000 babies) and the UK funded "Biobank Imaging" (data from 100,000 adults). Clusters of GPUs have been created in different institutions in order to take advantage of our tools and analyse all these datasets. Without

the GPU designs, instead of days, it would take months or even years to process all this data using typical CPU clusters (with hundreds of cores).

The solutions presented in Chapter 4 were specific to a few variants of the ball & sticks model. In Chapter 5 we presented a generic framework, the CUDA diffusion modelling toolbox (cuDIMOT), that provides a model-independent front-end, and automatically performs the CUDA implementation of fitting routines for user-defined voxel-wise MRI models. The framework offers the user an interface for specifying a new model, including its predicted signal, partial derivatives and parameter constraints. The specification is then used by the framework to automatically generate a GPU-accelerated binary for fitting the model, including various optimisation options (greedy optimisation, non-linear optimisation using Levenberg-Marquardt and Bayesian inference using MCMC). The framework also includes many features that make it generic and flexible for fitting any non-linear model. We reported accelerations of more than two orders of magnitude using cuDIMOT compared with the sequential implementations of some dMRI models.

In Chapter 5, we further used cuDIMOT to explore diffusion models that characterise fibre orientation dispersion. We implemented recently developed fibre dispersion models, such as NODDI and ball & rackets, and we compared and extended some of these models. For instance, we fitted crossing fibre models with and without dispersion using MCMC, and we identified regions in the brain where data supports more strongly one model over the others using model selection approaches. We proved that the framework is very useful for exploring, designing and implementing new dMRI protocols and models. cuDIMOT can be used to improve the model characterisation of the diffusion process, extract useful biophysical parameters and contribute to the development of new biomarkers. It is easy to use and generates very efficient GPU-accelerated solutions.

Finally, in Chapter 6, we presented a GPU-accelerated framework for a problem that poses considerably different challenges than the voxel-wise models. We developed a probabilistic tractography toolbox that achieves accelerations of more

than two orders of magnitude compared with a sequential solution, and can handle situations ranging from simple white matter tracking tasks to dense connectome generation. Thread divergence, heavy tasks performed by the threads and very high memory requirements are some of the challenges that make the application less than ideally suited for GPUs. Nevertheless, we showed how to overcome these challenges by using a pipelined design that includes simple tasks, and by overlapping tasks on the GPU. The GPU-accelerated tractography solution achieves good performance while maintains high flexibility, including the ability to handle both volumes and surfaces, options that allow a range of anatomical constraints to be imposed and the possibility of generating dense connectomes. We have shown the benefits of using this extended functionality, and we have validated the framework comparing it with a popular CPU-based application included in the FSL library.

All the parallel solutions presented in this thesis have been optimised for GPU architectures, designing solutions with several levels of parallelism and specific communications schemes that assume a large number (hundreds or thousands) of cores arranged in groups (SMs), and where threads execute the same set of instructions (SIMT). They would not be very efficient in multi-core processors, as only a few threads (tens of threads) can be executed in parallel, the multi-level parallelism will not be exploited, and the designs complexity will not be beneficial.

The implementations of the GPU-parallel designs have been optimised for the NVIDIA Kepler architecture, specifically for NVIDIA Tesla K80 GPUs. Ideally, when these implementations are used in different NVIDIA architectures, the arrangement of threads, i.e. number of threads per block and voxels distribution, should be optimised for the specific architecture (recompiling the source code). However, the benefits obtained by re-optimising the design for a different architecture are not expected to be major, and we prioritise here the simplicity and usability of the toolboxes.

In summary, in the three research chapters we have studied the computations performed in the analysis of dMRI, and we have proposed three novel

GPU-accelerated frameworks for allowing efficient and fast estimation of brain microstructure and connectivity. The results of this work are now widely used by the neuroimaging community.

7.2 Future perspectives

Large neuroscience projects involving thousands of subjects are becoming more common (see NIH's Connectome Coordination Facility: https://www.humanconnectome.org). Understanding the human brain is one of the key scientific challenges of the 21^{st} century and Big Data analysis is going to be a key for understanding the underlying anatomical and functional organisation of the brain [2–4]. Big Data have also the potential of revolutionising biomarker development and disease risk factor discovery [32]. Moreover, advances in MRI protocol acceleration, such as multiband acquisition [167, 168], will allow the collection of much more data in a given amount of time, leading to an increase in the resolution of the acquired data [174, 175, 214]. Thus, the computational demands will continue growing in the coming years, and parallel computing solutions for processing this large amount of data, as the ones proposed in this thesis, will be indispensable.

Fortunately, the performance offered by high performance computing systems is being increased every few months. Advances in memory and storage technology, and a wider adoption of parallel processors, such as GPUs and the new Knights Landing Intel Xeon Phi coprocessor [60], are leading the way to the new era of supercomputers. NVIDIA, one of the main GPU manufacturers, has announced the new Volta architecture [215], which provides more than 15 $TeraFLOPS$ of performance in single precision (vs. 10.6 $TeraFLOPS$ in Pascal architecture), and incorporates a second generation of the memory interconnection bus NVLINK, providing 150 GB/s in each direction (vs. 80 GB/s in Pascal). In part, this is the response to the growing demands of deep learning applications, from speech or visual recognition to object detection [216]. It seems that the field of deep learning is going to keep

increasing these computational demands, and undoubtedly, scientific applications are going to benefit from this improvement in computational performance.

The exploration of tissue microstructure is probably going to be fundamental for developing new biomarkers and gain new insight into the brain's structural organisation. Information about features of the tissue microstructure can be gained by modelling the diffusion process using biophysical parameters. Over the recent years, many modelling approaches have been proposed for resolving the complex configurations of fibres in some areas of the white matter, and for improving the specificity of the model parameters in explaining the diffusion process. Efforts still need to be done in this direction, as fibre configurations such as fanning, bending or kissing remain to be inaccurately resolved [217]. For instance, we have proposed in Chapter 5 the extension of a model than consider two fibres for characterising fanning patterns, and preliminary results show a considerable improvement in some brain areas such as the corona radiata. The toolbox that we have developed in Chapter 5, cuDIMOT, can be very helpful for developing and easily implementing new diffusion models. Given its high performance, complex models can be tested on the whole brain in few minutes.

Despite the large amount of dMRI modelling approaches, it seems that no single approach can resolve all the complex fibre configurations [189, 203]. Thus, applications that consider several models for selecting the best one in each voxel seems to be a potential solution. Given the computational cost of fitting these models, parallel solutions like cuDIMOT will be essential for performing this analysis.

Another approach for improving the complex fibre configuration issue, is the integration of neighbourhood information, instead of considering only the individual voxel-wise information [190]. Moreover, a new method where datasets are acquired with multiple spatial resolutions, low and high resolution, has been recently proposed [4, 172] to obtain the benefits from both strategies. This approach increases the complexity of parallel designs and the amount of data to be processed. In Chapter 4 we presented a GPU-accelerated solution for these kinds of approaches.

Many probabilistic tractography algorithms have also been proposed in the last years. It has been shown that these algorithms have limitations [133]. For instance, typically the voxel-wise fibre orientation is represented with symmetric patterns, however, taking into account the voxel neighbourhood information an asymmetric patterns can be distinguished [218]. In Chapter 6 we have proposed a GPU parallel framework for performing probabilistic tractography. The functionality of these new tractography approaches can be easily implemented in our application.

Tractography applications also have a great potential in clinical applications. For instance, tractography is used more and more for neurosurgical planning [219]. We believe that the GPU-accelerated solutions proposed in this thesis, which allow performing the analysis of dMRI in close to real-time, are going to make these methods more appealing to clinical practice.

Finally, a fact that must been taken into consideration is the possibility of improving the numerical methods used in this thesis given the accelerations obtained using GPUs. For instance, the optimisation routines used for performing tissue microstructure modelling could be replaced by more accurate methods that previously were prohibitive given their high computational demands, such as evolutionary algorithms [220].

Bibliography

[1] F. S. Collins and V. A. McKusick. “Implications of the human genome project for medical science”. *Jama*, 285(5): pp. 540–544. (2001).

[2] D. C. Van Essen and K. Ugurbil. “The future of the human connectome”. *NeuroImage*, 62(2): pp. 1299–310. (2012).

[3] D. C. Van Essen et al. “The Human Connectome Project: a data acquisition perspective”. *NeuroImage*, 62(4): pp. 2222–31. (2012).

[4] S. N. Sotiropoulos et al. “Advances in diffusion MRI acquisition and processing in the Human Connectome Project”. *NeuroImage*, 80: pp. 125–43. (2013).

[5] T. B. Murdoch and A. S. Detsky. “The Inevitable Application of Big Data to Health Care”. *Jama*, 309(13): pp. 1351–1352. (2013).

[6] F. F. Costa. “Big data in biomedicine”. *Drug Discovery Today*, 19(4): pp. 433–440. (2014).

[7] K. R. Foster, R. Koprowski, and J. D. Skufca. “Machine learning, medical diagnosis, and biomedical engineering research - commentary”. *BioMedical Engineering OnLine*, 13(1): p. 94. (2014).

[8] J. Schmidhuber. “Deep Learning in neural networks: An overview”. *Neural Networks*, 61: pp. 85–117. (2015).

[9] NVIDIA. “NVIDIA Tesla P100 Whitepaper. The Most Advanced Datacenter Accelerator Ever Built”. (2016).

[10] C. Zimmer. “100 trillion connections”. *Scientific American*, 304(1): pp. 58–61. (2011).

[11] M. Catani et al. “Connectomic approaches before the connectome”. *NeuroImage*, 80: pp. 2–13. (2013).

[12] D. E. Haines. *Neuroanatomy: An atlas of structures, sections, and systems*. Sixth Edit. (2004).

[13] A. E. Kelley, V. B. Domesick, and W. J. H. Nauta. “The amygdalostriatal projection in the rat - An anatomical study by anterograde and retrograde tracing methods”. *Neuroscience*, 7(3). (1982).

[14] P. C. Lauterbur. “Image formation by induced local interactions. Examples employing nuclear magnetic resonance”. *Nature*, 242: pp. 190–191. (1973).

[15] P. Mansfield and A. A. Maudsley. “Medical imaging by NMR”. *British Journal of Radiology*, 50(591): pp. 188–194. (1977).

[16] D. Le Bihan and B. E. “Imagerie de diffusion in-vivo par résonance magnétique nucléaire”. *Comptes-Rendus de l’Académie des Sciences*, 93(5): pp. 27–34. (1985).

[17] H. Johansen-Berg and T. E. J. Behrens. *Diffusion MRI*. Second Edit. (2014).

[18] Y. Assaf, H. Johansen-Berg, and M. Thiebaut de Schotten. “The role of diffusion MRI in neuroscience”. *NMR in Biomedicine*: e3762. (2017).

[19] M. R. Arbabshirani et al. “Single subject prediction of brain disorders in neuroimaging: Promises and pitfalls”. *NeuroImage*, 145: pp. 137–165. (2017).

[20] B. A. Ardekani et al. “Diffusion tensor imaging reliably differentiates patients with schizophrenia from healthy volunteers”. *Human Brain Mapping*, 32(1): pp. 1–9. (2011).

[21] M. Ingalhalikar et al. “Diffusion based abnormality markers of pathology: Toward learned diagnostic prediction of ASD”. *NeuroImage*, 57(3): pp. 918–927. (2011).

[22] N. Ohno et al. “Myelination and Axonal Electrical Activity Modulate the Distribution and Motility of Mitochondria at CNS Nodes of Ranvier”. *Journal of Neuroscience*, 31(20): pp. 7249–7258. (2011).

[23] P. J. Basser et al. “In vivo fiber tractography using DT-MRI data.” *Magnetic resonance in medicine*, 44(4): pp. 625–32. (2000).

[24] M. Catani et al. “Virtual in Vivo Interactive Dissection of White Matter Fasciculi in the Human Brain”. *NeuroImage*, 17(1): pp. 77–94. (2002).

[25] M. Bozzali et al. “Anatomical connectivity mapping: a new tool to assess brain disconnection in Alzheimer’s disease.” *NeuroImage*, 54(3): pp. 2045–51. (2011).

[26] M. Bozzali et al. “Anatomical brain connectivity can assess cognitive dysfunction in multiple sclerosis.” *Multiple Sclerosis Journal*, 19(9): pp. 1161–8. (2013).

[27] P. Fang et al. “Increased Cortical-Limbic Anatomical Network Connectivity in Major Depression Revealed by Diffusion Tensor Imaging”. *PLoS ONE*, 7(9). (2012).

[28] C. Nimsky, O. Ganslandt, and R. Fahlbusch. “Implementation of fiber tract navigation.” *Neurosurgery*, 58(ONS Suppl 2): ONS–292–304. (2006).

[29] C. Nimsky et al. “Intraoperative visualization of the pyramidal tract by diffusion-tensor-imaging-based fiber tracking.” *NeuroImage*, 30(4): pp. 1219–29. (2006).

[30] N. Mikuni et al. “Clinical impact of integrated functional neuronavigation and subcortical electrical stimulation to preserve motor function during resection of brain tumors.” *Journal of neurosurgery*, 106(4): pp. 593–8. (2007).

[31] M. Bastiani et al. “The developing Human Connectome : automated processing pipeline and quality control for neonatal dMRI”. In: *The Organization for Human Brain Mapping (OHBM)*. Vancouver (Canada), (2017).

[32] K. L. Miller et al. “Multimodal population brain imaging in the UK Biobank prospective epidemiological study”. *Nature Neuroscience*, 19(11): pp. 1523–1536. (2016).

[33] F. Alfaro-Almagro et al. “Image Processing and Quality Control for the first 10,000 Brain Imaging Datasets from UK Biobank”. *NeuroImage*, 166: pp. 400–424. (2018).

[34] B. W. Kernighan and D. M. Ritchie. *The C programming language*. (2006).

[35] B. Stroustrup. *The C++ programming language*. Pearson Education India, (1995).

[36] J. Nickolls et al. “Scalable parallel programming with CUDA”. *AMC Queue*, 6(April): pp. 40–53. (2008).

[37] B. Chapman, G. Jost, and R. V. D. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, (2008).

[38] M. Jenkinson et al. "Fsl". *NeuroImage*, 62(2): pp. 782–790. (2012).

[39] S. M. Smith. "Fast Robust Automated Brain Extraction". *Human brain mapping*, 17(3): pp. 143–155. (2002).

[40] Y. Zhang, M. Brady, and S. Smith. "Segmentation of Brain MR Images Through a Hidden Markov Random Field Model and the Expectation-Maximization Algorithm". *IEEE transactions on medical imaging*, 20(1): pp. 45–57. (2001).

[41] D. S. Marcus et al. "Human Connectome Project informatics: Quality control, database services, and data visualization". *NeuroImage*, 80: pp. 202–219. (2013).

[42] The MathWorks. *MATLAB*. (2016).

[43] The Spyder Project Contributors. *Spyder version 3.1.2*. URL: `https://pypi.python.org/pypi/spyder/3.1.2`.

[44] Inkscape's Contributors. *Inskcape version 0.91*. URL: `http://www.inkscape.org`.

[45] P. Brachet. *Texmaker version 5.0*. URL: `http://www.xm1math.net/texmaker/`.

[46] P. E. Ceruzzi. *Computing: A Concise History*. (2012).

[47] N. Wirth. "A plea for lean software". *Computer*, 28(2): pp. 64–68. (1995).

[48] B. G. E. Moore. "Cramming more components onto integrated circuits". *Electronics*, 38(8): pp. 114–117. (1965).

[49] M. B. Giles and I. Reguly. "Trends in high-performance computing for engineering calculations". *Philosophical Transactions of the Royal Society of London A*, 372(2022): p. 20130319. (2014).

[50] M. Horowitz et al. "Scaling, power, and the future of CMOS". In: *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International*. (2005), pp. 9–15.

[51] J. L. Henning. "SPEC CPU2006 Benchmark Descriptions". *ACM SIGARCH Computer Architecture News*, 34(4): pp. 1–17. (2006).

[52] J. Von Neumann and M. D. Godfrey. "First Draft of a Report on the EDVAC". *IEEE Annals of the History of Computing*, 15(4): pp. 27–75. (1993).

[53] H. Sutter and J. Larus. "Software and the concurrency revolution". *Queue*, 3(7): pp. 54–62. (2005).

[54] G. E. Karniadakis and R. M. Kirby II. *Parallel Scientific Computing in C++ and MPI*. Cambridge University Press, (2003).

[55] E. E. Schadt et al. "Computational solutions to large-scale data management and analysis". *Nature Reviews Genetics*, 11(9): pp. 647–657. (2010).

[56] K. Asanovic et al. "The Landscape of Parallel Computing Research : A View from Berkeley". *University of California, Berkeley.Technical Report*, UCB/EECS-2006-183. (2006).

[57] W. Hwu, K. Keutzer, and T. G. Mattson. "The Concurrency Challenge". *IEEE Design and Test of Computers*, 25(4): pp. 312–320. (2008).

[58] M. J. Flynn. "Some Computer Organizations and Their Effectiveness". *IEEE Transaction on Computers*, C-21(9): pp. 948–960. (1972).

[59] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Second Edit. The MIT press, (1999).

[60] A. Sodani et al. "Knights Landing: Second-Generation Intel Xeon Phi Product". *IEEE Micro*, 36(2): pp. 34–46. (2016).

[61] R. M. Russell. "The CRAY-1 computer system". *Communications of the ACM*, 21(1): pp. 63–72. (1978).

[62] J. E. Stone, D. Gohara, and G. Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". *Computing in Science & Engineering*, 12(3): pp. 66–73. (2010).

[63] S. Wienke et al. "OpenACC - First experiences with real-world applications". *Euro-Par 2012 Parallel Processing*: pp. 859–870. (2012).

[64] T. Hoshino et al. "CUDA vs OpenACC: Performance case studies with Kernel benchmarks and a memory-bound CFD application". In: *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. (2013), pp. 136–143.

[65] J. Effers and J. Reinders. *Intel Xeon Phi Coprocessor High-Performance Programming*. Morgan Kaufmann, (2013).

[66] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, (1999).

[67] J. Carabano et al. "An exploration of heterogeneous systems". In: *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, (2013), pp. 1–7.

[68] C.-T. Yang, C.-L. Huang, and C.-F. Lin. "Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters". *Computer Physics Communications*, 182(1): pp. 266–269. (2011).

[69] R. M. Tomasulo. "An efficient algorithm for exploiting multiple arithmetic units". *IBM Journal of research and Development*, 11(1): pp. 25–33. (1967).

[70] G. M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*. New York, New York, USA, (1967), p. 483.

[71] J. L. Gustafson. "Reevaluating Amdahl's law". *Communications of the ACM*, 31(5): pp. 532–533. (1988).

[72] D. B. Kirk and W. W. Hwu. "Chapter 2 - History of GPU Computing". In: *Programming Massively Parallel Processors. A Hands-on Approach*. Second Edit. Morgan Kaufmann, (2013), pp. 23–29.

[73] M. J. Harris et al. "Physically-based visual simulation on graphics hardware". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. (2002), pp. 109–118.

[74] W. W. Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, (2011).

[75] W. W. Hwu. *GPU Computing Gems Jade Edition*. Morgan Kaufmann, (2011).

[76] E. Lindholm et al. "NVIDIA Tesla: A unified graphics and computing architecture". *Ieee Micro*, 28(2): pp. 39–55. (2008).

[77] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. Tech. rep. NVIDIA Corporation, (2009).

[78] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210*. Tech. rep. NVIDIA Corporation, (2014).

[79] NVIDIA. "NVIDIA GeForce GTX 980. The Most Advanced GPU Ever Made". (2014).

[80] R. Budruk, D. Anderson, and T. Shanley. *PCI express system architecture*. Addison-Wesley Professional, (2004).

[81] NVIDIA. *NVIDIA NVLink High-Speed Interconnect: Application Performance*. Tech. rep. (2014).

[82] NVIDIA. *Cuda C Programming Guide v7.5*. (2015).

[83] J. Cheng, M. Grossman, and T. McKercher. *Professional Cuda C Programming*. John Wiley & Sons, (2014).

[84] M. Andersch et al. "Analyzing GPGPU Pipeline Latency". In: *Tenth International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*. Fiuggi, Italy, (2014).

[85] V. Volkov. "Better performance at lower occupancy". In: *Proceedings of the GPU Technology Conference*. (2010).

[86] J. D. Little. "A proof for the queuing formula: L= λW". *Operations research*, 9(3): pp. 383–387. (1961).

[87] M. B. Giles et al. "Designing OP2 for GPU architectures". *Journal of Parallel and Distributed Computing*, 73(11): pp. 1451–1460. (2013).

[88] R. Brown. "A brief account of microscopical observations made in the months of June, July and August 1827, on the particles contained in the pollen of plants; and on the general existence of active molecules in organic and inorganic bodies". *Philosophical Magazine Series 2*, 4(21): pp. 161–173. (1828).

[89] A. Einstein. "Über die von der molekularkinetischen Theorie der Wärme geforderte Bewegung von in ruhenden Flüssigkeiten suspendierten Teilchen". *Annalen der Physik*, 322(8): pp. 549–560. (1905).

[90] C. Beaulieu. "The basis of anisotropic water diffusion in the nervous system - a technical review." *NMR in biomedicine*, 15(7-8): pp. 435–55. (2002).

[91] E. L. Hahn. "Spin echoes". *Physical Review*, 80(4): pp. 580–594. (1950).

[92] E. O. Stejskal and J. E. Tanner. "Spin Diffusion Measurements: Spin Echoes in the Presence of a Time-Dependent Field Gradient". *The Journal of Chemical Physics*, 42(1): pp. 288–292. (1965).

[93] D. Le Bihan et al. "MR imaging of intravoxel incoherent motions: application to diffusion and perfusion in neurologic disorders." *Radiology*, 161: pp. 401–407. (1986).

[94] P. Basser, J. Mattiello, and D. LeBihan. "Estimation of the Effective Self-Diffusion Tensor from the NMR Spin Echo". *Journal of Magnetic Resonance, Series B*, 103(3): pp. 247–254. (1994).

[95] P. J. Basser, J. Mattiello, and D. LeBihan. “MR diffusion tensor spectroscopy and imaging”. *Biophysical journal*, 66(1): pp. 259–267. (1994).

[96] K. Levenberg. “A method for the solution of certain problems in least squares”. *Quarterly of applied mathematics*, 2: pp. 164–168. (1944).

[97] D. W. Marquardt. “An Algorithm for Least-Squares Estimation of Nonlinear Parameters”. *Journal of the Society for Industrial and Applied Mathematics*, 11(2): pp. 431–441. (1963).

[98] J. J. More. “The Levenberg-Marquardt Algorithm: Implementation and Theory”. In: *Numerical Analysis. Lecture Notes in Mathematics.* Vol. 630. Springer-Verlag, (1977).

[99] S. Pajevic and C. Pierpaoli. “Color schemes to represent the orientation of anisotropic tissues from diffusion tensor data: Application to white matter fiber tract mapping in the human brain”. *Magnetic Resonance in Medicine*, 42(3): pp. 526–540. (1999).

[100] P. Basser. “Inferring Microstructural Features and the Physiological State of Tissues from Diffusion-Weighted Images”. *NMR in Biomedicine*, 8: pp. 333–344. (1995).

[101] P. J. Basser and C. Pierpaoli. “Microstructural and Physiological Features of Tissues Elucidated by Quantitative-Diffusion-Tensor MRI”. *Journal of magnetic resonance B*, 111(2): pp. 209–219. (1996).

[102] C. Pierpaoli and P. J. Basser. “Toward a quantitative assessment of diffusion anisotropy.” *Magnetic Resonance in Medicine*, 36(6): pp. 893–906. (1996).

[103] M. E. Moseley et al. “Clinical Aspects of DWI”. *NMR in Biomedicine*, 8(7): pp. 387–396. (1995).

[104] M. Bozzali et al. “White matter damage in Alzheimer’s disease assessed in vivo using diffusion tensor magnetic resonance imaging”. *J Neurol Neurosurg Psychiatry*, 72: pp. 742–746. (2002).

[105] M. Filippi et al. “Diffusion tensor magnetic resonance imaging in multiple sclerosis”. *Neurology*, 56(3): pp. 304–311. (2001).

[106] J.-D. Tournier, S. Mori, and A. Leemans. “Diffusion tensor imaging and Beyond”. *Magn Reson Med*, 65: pp. 1532–1556. (2011).

[107] B. Jeurissen et al. “Investigating the prevalence of complex fiber configurations in white matter tissue with diffusion magnetic resonance imaging”. *Human Brain Mapping*, 34(11): pp. 2747–2766. (2013).

[108] V. J. Wedeen et al. “Mapping complex tissue architecture with diffusion spectrum magnetic resonance imaging”. *Magnetic Resonance in Medicine*, 54(6): pp. 1377–1386. (2005).

[109] D. S. Tuch et al. “Diffusion MRI of Complex Neural Architecture”. *Neuron*, 40(5): pp. 885–895. (2003).

[110] D. S. Tuch. “Q-ball imaging”. *Magnetic Resonance in Medicine*, 52(6): pp. 1358–1372. (2004).

[111] J. D. Tournier et al. "Direct estimation of the fiber orientation density function from diffusion-weighted MRI data using spherical deconvolution". *NeuroImage*, 23(3): pp. 1176–1185. (2004).

[112] A. W. Anderson. "Measurement of fiber orientation distributions using high angular resolution diffusion imaging". *Magnetic Resonance in Medicine*, 54(5): pp. 1194–1206. (2005).

[113] B. Scherrer and S. K. Warfield. "Why multiple b-values are required for multi-tensor models. Evaluation with a constrained log-euclidean model". In: *IEEE International Symposium on Biomedical Imaging: From Nano to Macro*. (2010), pp. 1389–1392.

[114] D. C. Alexander and G. J. Barker. "Optimal imaging parameters for fiber-orientation estimation in diffusion MRI". *NeuroImage*, 27(2): pp. 357–367. (2005).

[115] T. E. J. Behrens et al. "Characterization and propagation of uncertainty in diffusion-weighted MR imaging." *Magnetic resonance in medicine : official journal of the Society of Magnetic Resonance in Medicine / Society of Magnetic Resonance in Medicine*, 50(5): pp. 1077–88. (2003).

[116] T. E. J. Behrens et al. "Probabilistic diffusion tractography with multiple fibre orientations: What can we gain?" *NeuroImage*, 34(1): pp. 144–55. (2007).

[117] E. Panagiotaki et al. "Compartment models of the diffusion MR signal in brain white matter: A taxonomy and comparison". *NeuroImage*, 59(3): pp. 2241–2254. (2012).

[118] U. Ferizi et al. "A ranking of diffusion MRI compartment models with in vivo human brain data". *Magnetic Resonance in Medicine*, 72(6): pp. 1785–1792. (2014).

[119] Y. Assaf et al. "New modeling and experimental framework to characterize hindered and restricted water diffusion in brain white matter". *Magnetic Resonance in Medicine*, 52(5): pp. 965–978. (2004).

[120] Y. Assaf and P. J. Basser. "Composite hindered and restricted model of diffusion (CHARMED) MR imaging of the human brain". *NeuroImage*, 27(1): pp. 48–58. (2005).

[121] Y. Assaf et al. "AxCaliber: a method for measuring axon diameter distribution from diffusion MRI." *Magnetic resonance in medicine*, 59(6): pp. 1347–54. (2008).

[122] D. C. Alexander. "A general framework for experiment design in diffusion MRI and its application in measuring direct tissue-microstructure features". *Magnetic Resonance in Medicine*, 60(2): pp. 439–448. (2008).

[123] D. C. Alexander et al. "Orientationally invariant indices of axon diameter and density from diffusion MRI." *NeuroImage*, 52(4): pp. 1374–89. (2010).

[124] H. Zhang et al. "NODDI: practical in vivo neurite orientation dispersion and density imaging of the human brain." *NeuroImage*, 61(4): pp. 1000–16. (2012).

[125] S. N. Sotiropoulos, T. E. J. Behrens, and S. Jbabdi. "Ball and rackets: Inferring fiber fanning from diffusion-weighted MRI." *NeuroImage*, 60(2): pp. 1412–25. (2012).

[126] D. S. Tuch et al. “High angular resolution diffusion imaging reveals intravoxel white matter fiber heterogeneity”. *Magnetic Resonance in Medicine*, 48(4): pp. 577–582. (2002).

[127] B. Jeurissen et al. “Diffusion MRI Fiber tractography of the brain”. *NMR Biomed*, (February): pp. 1–22. (2017).

[128] S. Mori et al. “Three-dimensional tracking of axonal projections in the brain by magnetic resonance imaging.” *Annals of neurology*, 45(2): pp. 265–9. (1999).

[129] T. E. Conturo et al. “Tracking neuronal fiber pathways in the living human brain”. *Proceedings of the National Academy of Sciences*, 96(18): pp. 10422–10427. (1999).

[130] M. de Groot et al. “Improving alignment in Tract-based spatial statistics: evaluation and optimization of image registration.” *NeuroImage*, 76: pp. 400–11. (2013).

[131] D. K. Jones. “Determining and visualizing uncertainty in estimates of fiber orientation from diffusion tensor MRI”. *Magnetic Resonance in Medicine*, 49(1): pp. 7–12. (2003).

[132] B. Whitcher et al. “Using the wild bootstrap to quantify uncertainty in diffusion tensor imaging”. *Human Brain Mapping*, 29(3): pp. 346–362. (2008).

[133] S. Jbabdi and H. Johansen-Berg. “Tractography: where do we go from here?” *Brain connectivity*, 1(3): pp. 169–83. (2011).

[134] S. Jbabdi et al. “Measuring macroscopic brain connections in vivo”. *Nature Neuroscience*, 18(11): pp. 1546–1555. (2015).

[135] H. Johansen-Berg et al. “Changes in connectivity profiles define functionally distinct regions in human medial frontal cortex.” *Proceedings of the National Academy of Sciences of the United States of America*, 101(36): pp. 13335–40. (2004).

[136] T. E. J. Behrens et al. “Non-invasive mapping of connections between human thalamus and cortex using diffusion imaging.” *Nature neuroscience*, 6(7): pp. 750–7. (2003).

[137] M. Catani and M. Thiebaut de Schotten. “A diffusion tensor imaging tractography atlas for virtual in vivo dissections”. *Cortex*, 44(8): pp. 1105–1132. (2008).

[138] S. Wakana et al. “Fiber Tract–based Atlas of Human White Matter Anatomy”. *Radiology*, 230(1): pp. 77–87. (2004).

[139] O. Sporns, G. Tononi, and R. Kötter. “The human connectome: A structural description of the human brain”. *PLoS Computational Biology*, 1(4): pp. 0245–0251. (2005).

[140] O. Sporns. “The human connectome: A complex network”. *Annals of the New York Academy of Sciences*, 1224(1): pp. 109–125. (2011).

[141] S. N. Sotiropoulos and A. Zalesky. “Building connectomes using diffusion MRI: Why, how and but”. *NMR in Biomedicine*, (May): pp. 1–23. (2017).

[142] R. S. Desikan et al. “An automated labeling system for subdividing the human cerebral cortex on MRI scans into gyral based regions of interest”. *NeuroImage*, 31(3): pp. 968–980. (2006).

[143] M. F. Glasser et al. "The minimal preprocessing pipelines for the Human Connectome Project". *NeuroImage*, 80: pp. 105–124. (2013).

[144] M. Hernández et al. "Accelerating Fibre Orientation Estimation from Diffusion Weighted Magnetic Resonance Imaging Using GPUs". *PLoS ONE*, 8(4). (2013).

[145] C. Pierpaoli et al. "Diffusion tensor MR imaging of the human brain." *Radiology*, 201(3): pp. 637–648. (1996).

[146] C. G. Koay et al. "A unifying theoretical and algorithmic framework for least squares methods of estimation in diffusion tensor imaging". *Journal of Magnetic Resonance*, 182(1): pp. 115–125. (2006).

[147] A. L. Alexander et al. "Analysis of partial volume effects in diffusion-tensor MRI." *Magnetic Resonance in Medicine*, 45(5): pp. 770–780. (2001).

[148] C. Pierpaoli et al. "Water diffusion changes in Wallerian degeneration and their dependence on white matter architecture." *NeuroImage*, 13(6): pp. 1174–1185. (2001).

[149] C. Poupon et al. "Regularization of Diffusion-Based Direction Maps for the Tracking of Brain White Matter Fascicles". *NeuroImage*, 12(2): pp. 184–195. (2000).

[150] M. R. Wiegell, H. B. Larsson, and V. J. Wedeen. "Fiber Crossing in Human Brain Depicted with Diffusion Tensor MR Imaging." *Radiology*, 217(3): pp. 897–903. (2000).

[151] K. K. Seunarine and D. C. Alexander. "Chapter 6 - Multiple Fibers: Beyond the Diffusion Tensor". In: *DIFFUSION MRI. From Quantitative Measurement to In-vivo Neuroanatomy*. Second Edit. Academic Press, (2014), pp. 105–123.

[152] A. Szafer et al. "Diffusion-weighted imaging in tissues: theoretical models." *NMR in biomedicine*, 8(7-8): pp. 289–296. (1995).

[153] T. Niendorf et al. "Biexponential diffusion attenuation in various states of brain tissue: implications for diffusion-weighted imaging". *Magnetic Resonance in Medicine*, 36(6): pp. 847–857. (1996).

[154] G. J. Stanisz et al. "An analytical model of restricted diffusion in bovine optic nerve". *Magnetic Resonance in Medicine*, 37(1): pp. 103–111. (1997).

[155] Y. Assaf and Y. Cohen. "Non-mono-exponential attenuation of water and N-acetyl aspartate signals due to diffusion in brain tissue". *Journal of Magnetic Resonance*, 131(1): pp. 69–85. (1998).

[156] R. V. Mulkern et al. "Multi-component apparent diffusion coefficients in human brain." *NMR in biomedicine*, 12(1): pp. 51–62. (1999).

[157] S. Jbabdi et al. "Model-based analysis of multishell diffusion MR data for tractography: How to get over fitting problems". *Magnetic Resonance in Medicine*, 68(6): pp. 1846–1855. (2012).

[158] H. J. Motulsky and L. a. Ransnas. "Fitting curves to data using nonlinear regression: a practical and nonmathematical review". *FASEB Journal*, 1(5): pp. 365–374. (1987).

[159] C. T. Kelley. *Iterative Methods for Optimization*. Society for Industrial and Applied Mathematics, (1999), p. 188.

[160] A. Tarantola. *Inverse Problem Theory and Methods for Model Parameter Estimation*. Society for Industrial and Applied Mathematics, (2005).

[161] N. Metropolis and S. Ulam. "The Monte Carlo Method". *Journal of the American Statistical Association*, 44(247): pp. 335–341. (1949).

[162] W. Hastings. "Monte Carlo sampling methods using Markov chains and their applications". *Biometrika*, 57(1): pp. 97–109. (1970).

[163] J. L. R. Andersson, M. Jenkinson, and S. M. Smith. "Non-linear optimisation. FMRIB technical report TR07JA1". *University of Oxford FMRIB Centre: Oxford, UK*. (2007).

[164] N. Metropolis et al. "Equation of State Calculations by Fast Computing Machines". *The Journal of Chemical Physics*, 21(6): pp. 1087–1092. (1953).

[165] NVIDIA. "Profiler User's Guide". (2017).

[166] NVIDIA. "TESLA K80 GPU ACCELERATOR. Board Specification". (2015).

[167] S. Moeller et al. "Multiband multislice GE-EPI at 7 tesla, with 16-fold acceleration using partial parallel imaging with application to high spatial and temporal whole-brain fMRI." *Magnetic resonance in medicine*, 63(5): pp. 1144–53. (2010).

[168] K. Setsompop et al. "Blipped-controlled aliasing in parallel imaging for simultaneous multislice echo planar imaging with reduced g-factor penalty". *Magnetic Resonance in Medicine*, 67(5): pp. 1210–1224. (2012).

[169] N. Whitehead and A. Fit-florea. *Precision & Performance : Floating Point and IEEE 754 Compliance for NVIDIA GPUs*. (2011).

[170] M. Xu et al. "Probabilistic Brain Fiber Tractography on GPUs". *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*: pp. 742–751. (2012).

[171] L.-C. Chang et al. "GPU acceleration of nonlinear diffusion tensor estimation using CUDA and MPI". *Neurocomputing*, 135: pp. 328–338. (2014).

[172] S. N. Sotiropoulos et al. "Fusion in diffusion MRI for improved fibre orientation estimation: An application to the 3T and 7T data of the Human Connectome Project". *NeuroImage*, 134: pp. 396–409. (2016).

[173] S. N. Sotiropoulos et al. "RubiX: Combining Spatial Resolutions for Bayesian Inference of Crossing Fibers in Diffusion MRI". *IEEE transactions on medical imaging*, 32(6): pp. 969–82. (2013).

[174] R. M. Heidemann et al. "k-space and q-space: Combining ultra-high spatial and angular resolution in diffusion imaging using ZOOPPA at 7T". *NeuroImage*, 60(2): pp. 967–978. (2012).

[175] A. T. Vu et al. "High resolution whole brain diffusion imaging at 7T for the Human Connectome Project". *NeuroImage*, 122: pp. 318–331. (2015).

[176] D. Purves et al. "Studying the Nervous Systems of Humans and Other Animals". In: *Neuroscience*. Third Edit. Sinauer Associates, Inc., (2004). Chap. 1.

[177] J. M. Edgar and I. R. Griffiths. "White Matter Structure: A Microscopist's View". In: *DIFFUSION MRI. From Quantitative Measurement to In-vivo Neuroanatomy*. Second Edit. Elsevier, (2014). Chap. 7.

[178] M. Tariq et al. "Bingham-NODDI: Mapping anisotropic orientation dispersion of neurites using diffusion MRI". *NeuroImage*, 133: pp. 207–223. (2016).

[179] D. J. MacKay. "Developments in Probabilistic Modelling with Neural Networks - Ensemble Learning". In: *Neural Networks: Artificial Intelligence and Industrial Applications. Proc. of the 3rd Annual Symposium on Neural Networks*. (1995), pp. 191–198.

[180] J. A. McNab and K. L. Miller. "Sensitivity of diffusion weighted steady state free precession to anisotropic diffusion". *Magnetic Resonance in Medicine*, 60(2): pp. 405–413. (2008).

[181] S. C. Deoni. "Quantitative Relaxometry of the Brain". *Topics in magnetic resonance imaging: TMRI*, 21(2): pp. 101–113. (2010).

[182] S. Foxley et al. "A comparison of multiple acquisition strategies to overcome B1 inhomogeneities in diffusion imaging of post-mortem human brain at 7T". In: *24th International Society for Magnetic Resonance in Medicine*. Singapore, (2016).

[183] S. Foxley et al. "Improved tract identification of post-mortem human brain with high-resolution DTI at 7T". In: *21st Annual Meeting of the Organization for Human Brain Mapping*. Honolulu (Hawaii, US), (2015).

[184] G. Schwarz. "Estimating the dimension of a model". *The Annals of Statistics*, 6(2): pp. 461–464. (1978).

[185] H. Akaike. "A New Look at the Statistical Model Identification". *IEEE Transactions on Automatic Control*, 19(6): pp. 716–723. (1974).

[186] H. Gudbjartsson and S. Patz. "The Rician Distribution of Noisy MRI Data Hákon". *Magn Reson Med.* 34(6): pp. 910–914. (1995).

[187] J. Mollink et al. "Evaluating fibre orientation dispersion in white matter: Comparison of diffusion MRI, histology and polarized light imaging". *NeuroImage*, 157: pp. 561–574. (2017).

[188] M. D. Budde and J. Annese. "Quantification of anisotropy and fiber orientation in human brain histological sections". *Frontiers in integrative neuroscience*, 7(February): p. 3. (2013).

[189] A. Ghosh, D. Alexander, and H. Zhang. "Crossing Versus Fanning: Model Comparison Using HCP Data". In: *Computational Diffusion MRI*. (2016), pp. 159–169.

[190] M. Rowe et al. "Beyond crossing fibers : Tractography exploiting sub-voxel fibre dispersion and neighbourhood structure". In: *International Conference on Information Processing in Medical Imaging*. (2013), pp. 402–413.

[191] K. V. Mardia and P. E. Jupp. "Distributions on spheres". In: *Directional Statistics*. (2000), pp. 159–192.

[192] C. Bingham. "An antipodally symmetric distribution on the sphere". *The Annals of Statistics*: pp. 1201–1225. (1974).

[193] H. Zhang et al. "Axon diameter mapping in the presence of orientation dispersion with diffusion MRI". *NeuroImage*, 56(3): pp. 1301–1315. (2011).

[194] J. L. R. Andersson, M. Jenkinson, and S. Smith. *Non-linear registration, aka Spatial normalisation. FMRIB Technial Report TR07JA2*. Tech. rep. (2007).

[195] Microstructure Imaging Group - University College London. *NODDI Matlab Toolbox*. URL: http://mig.cs.ucl.ac.uk/index.php?n=Tutorial.NODDImatlab.

[196] P. E. Gill et al. "Procedures for optimization problems with a mixture of bounds and general linear constraints". *ACM Transactions on Mathematical Software (TOMS)*, 10(3): pp. 282–298. (1984).

[197] A. Daducci et al. "Accelerated Microstructure Imaging via Convex Optimization (AMICO) from diffusion MRI data". *NeuroImage*, 105: pp. 32–44. (2015).

[198] A. Kume and A. T. A. Wood. "Saddlepoint approximations for the Bingham and Fisher-Bingham normalising constants". *Biometrika*, 92(2): pp. 465–476. (2005).

[199] P. Koev and A. Edelman. "The efficient evaluation of the hypergeometric function of a matrix argument". *Mathematics of Computation*, 75(254): pp. 833–846. (2006).

[200] C. P. Reddy and Y. Rathi. "Joint multi-fiber NODDI parameter estimation and tractography using the unscented information filter". *Frontiers in Neuroscience*, 10(APR): pp. 1–10. (2016).

[201] R. L. Harms et al. "Robust and fast nonlinear optimization of diffusion MRI microstructure models". *NeuroImage*, 155(April): pp. 82–96. (2017).

[202] H. Farooq et al. "Microstructure Imaging of Crossing (MIX) White Matter Fibers from diffusion MRI". *Scientific Reports*, 6(September): p. 38927. (2016).

[203] U. Ferizi et al. "White matter compartment models for in vivo diffusion MRI at 300mT/m". *NeuroImage*, 118: pp. 468–483. (2015).

[204] S. N. Jespersen et al. "Modeling dendrite density from magnetic resonance diffusion measurements". *NeuroImage*, 34(4): pp. 1473–1486. (2007).

[205] W. Press et al. *Numerical Recipes in C: The Art of Scientific Computing*. (1987).

[206] L. Li et al. "The effects of connection reconstruction method on the interregional connectivity of brain networks via diffusion tractography". *Human Brain Mapping*, 33(8): pp. 1894–1913. (2012).

[207] C. J. Donahue et al. "Using Diffusion Tractography to Predict Cortical Connection Strength and Distance: A Quantitative Comparison with Tracers in the Monkey." *The Journal of neuroscience: the official journal of the Society for Neuroscience*, 36(25): pp. 6758–70. (2016).

[208] J. Harwell et al. "GIfTI : Geometry Data Format for Exchange of Surface-Based Brain Mapping Data". In: *OHBM*. Melbourne, Australia, (2008).

[209] S. M. Smith et al. "Advances in functional and structural MR image analysis and implementation as FSL". *NeuroImage*, 23(SUPPL. 1): pp. 208–219. (2004).

[210] J. O'Rouke. "Search and Intersection". In: *Computational Geometry in C*. (1998).

[211] A. Mittmann, E. Comunello, and A. von Wangenheim. "Diffusion tensor fiber tracking on graphics processing units." *Computerized medical imaging and graphics : the official journal of the Computerized Medical Imaging Society*, 32(7): pp. 521–30. (2008).

[212] R. E. Smith et al. "Anatomically-constrained tractography: Improved diffusion MRI streamlines tractography through effective use of anatomical information". *NeuroImage*, 62(3): pp. 1924–1938. (2012).

[213] M. De Groot. *AutoPtx*. URL: `https://fsl.fmrib.ox.ac.uk/fsl/fslwiki/AutoPtx`.

[214] W. Wu et al. "High-resolution diffusion MRI at 7T using a three-dimensional multi-slab acquisition". *NeuroImage*, 143: pp. 1–14. (2016).

[215] NVIDIA. *NVIDIA TESLA V100 GPU ARCHITECTURE: The World's Most Advanced Data Center GPU*. Tech. rep. (2017).

[216] Y. LeCun, Y. Bengio, and G. Hinton4. "Deep learning". *Nature*, 521(7553): pp. 436–444. (2015).

[217] K. K. Seunarine and D. C. Alexander. "Multiple Fibers: Beyond the Diffusion Tensor". In: *Diffusion MRI*. (2014), pp. 105–123.

[218] M. Bastiani et al. "Improved tractography using asymmetric fibre orientation distributions". *NeuroImage*, 158(May): pp. 205–218. (2017).

[219] W. I. Essayed et al. "White matter tractography for neurosurgical planning: A topography-based review of the current state of the art". *NeuroImage: Clinical*, 15: pp. 659–672. (2017).

[220] T. Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, (1996).