HIGH PERFORMANCE MULTISCALE IMAGE PROCESSING FRAMEWORK ON MULTIGPUS WITH APPLICATIONS TO UNBIASED DIFFEOMORPHIC ATLAS CONSTRUCTION

by

Linh Khanh Ha

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computing

School of Computing

The University of Utah

May 2011

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Linh Khanh Ha

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Cláudio T. Silva
	Sarang Joshi
	Jens Krüger
	P. Thomas Fletcher
	Joao Comba

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Linh Khanh Ha in its final form and have found that:
(1) its format, citations, and bibliographic style are consistent and acceptable;
(2) its illustrative materials including figures, tables, and charts are in place;
(3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Cláudio T. Silva
Chair, Supervisory Committee

Approved for the Major Department

Al Davis
Chair/Dean

Approved for the Graduate Council

Charles A. Wight
Dean of The Graduate School

ABSTRACT

Stochastic methods, dense free-form mapping, atlas construction, and total variation are examples of advanced image processing techniques which are robust but computationally demanding. These algorithms often require a large amount of computational power as well as massive memory bandwidth. These requirements used to be fulfilled only by supercomputers. The development of heterogeneous parallel subsystems and computation-specialized devices such as Graphic Processing Units (GPUs) has brought the requisite power to commodity hardware, opening up opportunities for scientists to experiment and evaluate the influence of these techniques on their research and practical applications. However, harnessing the processing power from modern hardware is challenging. The differences between multicore parallel processing systems and conventional models are significant, often requiring algorithms and data structures to be redesigned significantly for efficiency. It also demands in-depth knowledge about modern hardware architectures to optimize these implementations, sometimes on a per-architecture basis.

The goal of this dissertation is to introduce a solution for this problem based on a 3D image processing framework, using high performance APIs at the core level to utilize parallel processing power of the GPUs. The design of the framework facilitates an efficient application development process, which does not require scientists to have extensive knowledge about GPU systems, and encourages them to harness this power to solve their computationally challenging problems. To present the development of this framework, four main problems are described, and the solutions are discussed and evaluated: (1) essential components of a general 3D image processing library: data structures and algorithms, as well as how to implement these building blocks on the GPU architecture for optimal performance; (2) an implementation of unbiased atlas construction algorithms—an illustration of how to solve a highly complex and computationally expensive algorithm using this framework; (3) an extension of the framework to account for geometry descriptors to solve registration challenges with large scale shape changes and high intensity-contrast differences; and (4) an out-of-core streaming model, which enables developers to implement multi-image processing techniques on commodity hardware.

To my father and mother for their dedication and love

CONTENTS

ABSTRACT . . . . . **iii**

LIST OF FIGURES . . . . . **viii**

LIST OF TABLES . . . . . **xiii**

ACKNOWLEDGEMENTS . . . . . **xiv**

CHAPTERS

1. INTRODUCTION **1**

1.1 Parallel computing overview 1
1.1.1 Modern trends in parallel computing 3
1.1.1.1 Multicore systems 4
1.1.1.2 Specialized processors 4
1.1.1.3 Heterogeneous computing 6
1.1.2 Parallel computing challenges 7
1.2 GPU computing 8
1.2.1 GPU computational model 10
1.3 Atlas construction problem 13
1.4 Challenges 16
1.4.1 Baseline research challenges 16
1.4.2 Efficient implementation challenges 17
1.5 Contributions 18

2. GPU IMAGE PROCESSING FRAMEWORK **21**

2.1 Framework overview 21
2.2 Core methods 23
2.2.1 Diffeomorphic image registration algorithms 23
2.2.1.1 Greedy iterative diffeomorphism 23
2.2.1.2 Large Deformation Diffeomorphic Metric Mapping 24
2.3 High performance image processing framework on GPUs 26
2.3.1 Data structures 26
2.3.1.1 Volume image presentation 26
2.3.1.2 Vector field presentation 27
2.3.2 Basic image operators 27
2.3.2.1 Gradient computation 30
2.3.3 ODE integration 31
2.3.4 PDE Solver 32
2.3.5 Successive over relaxation method 33
2.3.6 Conjugate Gradient method 36

2.3.7 Multiscale framework . . . 37
2.3.8 Multi-GPU processing model . . . 39
2.3.8.1 Single node multi-GPU model . . . 39
2.3.8.2 GPU cluster model . . . 41
2.3.8.3 Load balancing . . . 41
2.3.9 Other performance optimization . . . 43
2.3.9.1 Volume clipping optimization . . . 43
2.3.9.2 Scratch memory model . . . 43
2.4 Evaluation and validation of results . . . 44
2.4.1 Quality improvements . . . 45
2.4.2 Performance improvement . . . 46
2.5 Conclusion . . . 47

3. COMBINING PROBABILISTIC AND GEOMETRIC DESCRIPTOR . . . 49

3.1 Introduction . . . 49
3.2 Method overview . . . 51
3.2.1 Anatomical descriptors . . . 51
3.2.2 Registration formulation . . . 53
3.3 Efficient implementation . . . 55
3.3.1 Particle Mesh approximation for currents norm computation . . . 56
3.3.2 Efficient implementation of particle mesh method on GPUs . . . 58
3.3.2.1 Grid building . . . 59
3.3.2.2 Splatting . . . 59
3.3.2.3 Interpolation . . . 63
3.4 Other performance optimizations . . . 65
3.4.1 Parallel surface normal computation on GPUs . . . 65
3.4.2 Multiscale computation on GPUs . . . 66
3.5 Results . . . 67
3.5.1 Registration quality . . . 67
3.5.2 Performance . . . 70
3.5.2.1 Splatting . . . 70
3.5.2.2 Interpolation . . . 71
3.5.2.3 Probabilistic descriptor registration . . . 71
3.6 Conclusions . . . 72

4. AN OUT-OF-CORE FRAMEWORK FOR MULTI-IMAGE PROCESSING . . . 73

4.1 Introduction . . . 73
4.2 Related work . . . 75
4.3 The construction of the multi-image processing framework . . . 77
4.3.1 Multi-image processing operators . . . 77
4.4 MIP out-of-core streaming framework . . . 80
4.4.1 Synchronous out-of-core model . . . 81
4.4.2 Asynchronous optimal performance analyses . . . 82
4.4.3 Implicit streaming model . . . 83
4.4.4 Hardware-aware streaming model . . . 85
4.4.5 Hardware-independent streaming model . . . 86

4.4.6 Discussion on streaming modes 86
4.5 Reordering stages in streaming models 88
4.5.1 Forced synchronizations 88
4.5.2 Reordering pipeline stages 89
4.6 Extension to a full out-of-core framework 91
4.7 Results 92
4.7.1 Full asynchronous processing 93
4.7.2 Synchronous functions 94
4.7.3 Regular out-of-core functions 95
4.8 Conclusions 97

5. CONCLUSIONS AND FUTURE WORK 98

APPENDICES

A. PARALLEL GPU SORTING 100

B. SOFTWARE ARCHITECTURE 114

REFERENCES 125

LIST OF FIGURES

1.1 Parallel computing has been a driving force for the development of many scientific research disciplines and the solution for a number of engineering challenges in domain as diverse as mechanical engineering, nuclear physics, bio-science, applied physics, weather prediction, astronomy, geology, and more (Image courtesy of Blaise Barney, Lawrence Livermore National Laboratory [7]) 1

1.2 The development of GPU processing pipeline from a) a fixed function pipeline to b) a programmable pipeline is a prerequisite for General Purpose Computing on GPUs (GPGPUs). 9

1.3 Hardware architecture and execution model of modern GPUs. Modern GPUs are modeled as stream processors, with a large number of simple, compute centric cores compounded with a high bandwidth parallel memory interface. The multilevel threading hierarchy allows efficient parallel execution model with a fine-grain approach 11

1.4 Brain atlas construction from a population. Here we implement an unbiased atlas construction approach based on averaging in diffeomorphic space. 13

1.5 Automatic segmentation via atlas construction. The process includes two steps: a) Construct the brain atlas from the population - determine the mapping between each image and the atlas. b) Partition the atlas - the segmentation on an individual is done automatically via a reverse mapping from the atlas. 14

1.6 A small part of the letter "C" deforming into a full "C" using 2D Greedy Iterative Diffeomorphism. From left to right: 1. Input and Target Image 2. Deformed template. 3. Grid showing the deformation applied to template. 15

2.1 Forward itergration of the vector field in Greedy Iterative matching 23

2.2 LDDMM estimate the transformation based on both forward and backward integration of vector field in two opposite directions between the source and the target 25

2.3 The computation runtime of basic GPU functions is linearly proportional to the size of the input data. We also observe the similar trends with other GPU functions, it explain why a tight volume presentation is generally preferred in our framework. 27

2.4 Vector field presentation and one-D optimization for vector field computation 28

2.5 n-ary versus classic binary operator with linear interpolation and range normalization function. We use the memory copy from device to device, in other words, a no-op function as reference to show the optimality of our n-ary approach. Runtime is measured in milliseconds on an NVIDIA GTX 260. 29

2.6 n-ary average function versus binary average operator 30

2.7 Reverse mapping based on 3D trilinear interpolation 31

2.8 Parallel block SOR, we assign each CUDA thread warp a block of data to compute the black points inside the blue boundary, and use that result to compute the red point inside the red boundary. Two neighboring compute blocks share a four grid point-wide region. 34

2.9 CG Solver template 36

2.10 Matrix vector multiplication CUDA kernel with implicit Helmholtz Matrix . 37

2.11 Multi-GPUs framework on the GPU cluster. We combine the processing models using a hierarchical strategy, from a single-GPU model to a single node multi-GPUs model using PThreads, and finally to a GPU cluster with MPI communication between processing nodes. The distribution of compute flow and the data updating process happens in the opposite direction of the hierarchy. 40

2.12 MPI-All reduce runtime on an infiniband network with OpenMPI 1.3 shows a linear dependency on the number of nodes 42

2.13 Optimization strategies with the scratch memory model 44

2.14 Atlas results with 3, 5, 7, 9, 11 and 13 inputs constructed by (a) arithmetically averaging rigidly aligned images (top row) and (b) Greedy Iterative Average template construction (bottom row) 45

2.15 Mean entropy and variance of atlases constructed by arithmetically averaging and the Greedy Iterative Average template 46

2.16 Runtime to compute the average atlas of the 20 T1 brain images ($144 \times 192 \times 160$) with multiscale and/or multi-GPUs, cluster implementation in reference to one scale version 47

2.17 Multiscale runtime to compute the average atlas of the 315 T1 brain images ($144 \times 192 \times 160$) with different PDE solver 48

3.1 Registration challenges of human brains at early development stages. The image shows significant shape and size changes of an infant brain of subject 180 from two weeks to two years as well as the changing white matter properties and appearance due to the myelination. 50

3.2 Overview of the proposed registration method that can handle large deformations and different contrast properties, applied to mapping brain MRI of neonates to 2-year-olds. We segment the brain MRIs and then extract equivalent anatomical descriptors by merging the two different white matter types present in neonates. The probabilistic and geometric anatomical descriptors are then used to compute the transformation h that minimizes the distance between the class posterior images, as well as the distance between surfaces represented as currents. 52

3.3 Particle Mesh approximation algorithm to transform the computation from irregular domain to regular domain based on four basic steps: grid construction, splatting, integration and interpolation. 57

3.4 The percent error for different for 5000 randomly generated points with different mesh sizes. 58

3.5 Run time comparisons between direct computation and the particle mesh implementation for various grid sizes. 59

3.6 Geometrical conversion based on a splatting function with zero velocity field v (Eq 3.13). The method served as a bridge to transform the computation from an irregular grid to a regular grid which allows an efficient parallel implementation. 61

3.7 Collision-free splatting implementation using fast parallel sorting. The method is based on ordering the node contribution ID to resolve resource conflicts which allows a parallel efficient integration based on an optimal parallel prefix scan implementation. 62

3.8 Reverse mapping based on 3D trilinear interpolation that eliminates the missing data of a forward mapping. The implementation on GPU exploits the hardware interpolation engine to achieve significant speed up. 64

3.9 Geometries are updated through the interpolation from the velocity field. This step maintains the consistency between probabilistic and geometrical compartments of the mixture model. 65

3.10 Multiscale registration using different sizes of computation kernels help capture large and small scale changes in different levels and also increase the convergence rate of the algorithm. 66

3.11 Registration results of neonates mapped to 2-year-olds. From left to right: (a) neonatal T1 image after affine registration, (b) reference T1 image at 2 years, followed by (c) neonatal T1 after deformable mutual information registration using B-splines, and (d) after combined probabilistic and geometric registration. From top to bottom: subject 0012, 0102, 0106, 0121, 0130, 0146 and 0156. 68

3.12 Registration results of neonates mapped to 2-year-olds. From left to right: (a) neonatal T1 image after affine registration, (b) reference T1 image at 2 years, followed by (c) neonatal T1 after deformable mutual information registration using B-splines, and (d) after combined probabilistic and geometric registration. From top to bottom 0174, 0177 and 0180. 69

4.1 Atlas construction result on the ADNI data set composed of 156 images sized 144×192×160, with different average computations: a) the intensity average and the diffeomorphic atlas constructions with b) random permutation ([56]) with cohort size of 3 images c) random permutation with cohort size of 5 images and d) and all image using our out-of-core streaming framework. It is clear that the ability to compute the atlas using nonlinear diffeomorphic registration with all the image yields a discernible improvement in the quality of the construction. 75

4.2 Basic multi-image operators 78

4.3 General MIMO operators 79

4.4 Sliding window MIMO operators 79

4.5 Overview of data movement in our multi-image processing multilevel out-of-core streaming framework for heterogeneous systems. 80

4.6 Implicit processing model for MIMOs 84

4.7 Pipeline explicit processing model for MIMO operations 85

4.8 Although the hardware-independent model miss-predicts the system configuration, the performance is still optimal 87

4.9 The transformation from a synchronous model to an explicit streaming model preserves semantic correctness. 90

4.10 Streaming optimization using reordering technique. As shown on the figure it is able to eliminate the negative effect of forced-synchronous function . . . 91

4.11 The implementation of hardware-independent model for "full" out-of-core multi-image processing 92

4.12 Runtime comparison of different streaming strategies in ideal conditions. All the permutation of explicit model yield the same performance. The hardware-independent models achieve the optimal performance. 93

4.13 Runtime comparison of different streaming strategies in degenerate conditions 95

4.14 Age regression anlysis on the ADNI dataset by computing the average brain atlases at different ages (65, 70, 75, and 80) corroborates the hypothesis that fluid space is larger because brains atrophy overtime. This analysis, however, could only be performed if the system is capable of processing the whole dataset of 300 healthy brain-images 96

A.1 Global ranking computation for block radix sorting 102

A.2 Illustration of our implicit radix sorting (intermediate steps) a) Inputs b) Implicit-presentation of the input c) The local-prefix sum d) Number of each radix bucket e) Number of previous same bucket elements f) local rank g) presorted result h) Number of radix values in each block i) Start offset j) Sorted output 106

A.3 The flow of our hybrid-data format. The conversion occurred implicitly inside the global shuffling kernel and at the beginning of local counting kernel using texture memory. 108

A.4 Resolve the 4-way memory conflict 109
A.5 Total run-time of presorting step (ms) with Implicit Radix and Satish CUDPP1.1 radix-16 110
A.6 The sorting rate comparison of random 32-bit unsigned inputs 111
A.7 Global shuffling run-time comparison (ns) between our implementation of global shuffling with AoS, SoA structures, and CUDPP1.1 in reference to the device to device memory copy of the same input size: 111
B.1 Atlas construction framework data flow architecture overview. 114
B.2 Software development architecture of the AtlasWerk image registration framework. 116
B.3 A sampler of kernel/interface functions, which adds a constant to an array. The function is stored with .cu file extension and is compiled using CUDA compiler. 116
B.4 C++ template implementation of the multiscale registration 124

LIST OF TABLES

2.1 Runtime comparison in milliseconds of different gradient computations: simple global memory, linear 1D texture, 3D texture and shared memory approaches 31

2.2 Runtime comparison in milliseconds of different 3D interpolation implementations for reverse mapping operator using global memory, 1D linear texture and 3D hardware-accelerated texture 32

2.3 Performance comparison, in GFLOPs, between our implicit method and explicit implementations (larger is faster) 37

2.4 Performance comparison, in milliseconds, between different optimization strategies to implement 3D-Gaussian Filter with different kernel sizes 39

3.1 Overlap measures comparing the registered segmentation maps against the reference segmentation maps for the parenchyma and cerebellum structure, obtained without deformation (None), deformable mutual information registration (MI), and our proposed method (P+G). 69

3.2 Runtime comparison, in milliseconds, of different splatting implementations on volume sized $144 \times 192 \times 160$ and $160 \times 224 \times 160$ using collision-free sorting approach, atomic operation with fixed point presentation, atomic operation on the shared memory and CPU reference. 71

3.3 Time elapsed, in minutes, for registration using deformable mutual information (MI) on the CPU (AMD Phenom II X4 955, 6GB DDR3 1333) and our proposed approach (P+G) on the GPU (NVIDIA GTX 260, 896MB) with 1000 iterations of gradient descent. 71

4.1 Runtime comparison of regular functions with different streaming strategies 95

A.1 Component runtime comparision, in milliseconds, in one iteration of a 16M-pair input between our implicit sorting and the Satish *et al.* implementation. 110

ACKNOWLEDGEMENTS

What does the creator of a dissertation aspire to? Of course, to fulfill academic requirements for a Ph.D., but this is not the only goal. Maybe some want to prepare for an academic career, and some want to impress their peers or their parents. My aspiration is to present a dissertation that does not only make a small but significant contribution to scientific development, a system that not only provides functionality, flexibilities and powers to developers but also has a significant impact on practical applications.

The most important reason for succeeding is the guidance and support which I received from my advisor, Claudio T Silva. I have learned immensely from him. He taught me how to find direction in Ph.D. thesis work, drill down to the essentials, and make a dissertation out of it. I am highly grateful to him for making my Ph.D. thesis work such a smooth and rewarding experience.

The other committee members, Sarang Joshi, Jens Kruger, Joao Comba and Thomas Fletcher, are not only my mentors and co-workers but also my friends. They have given me tremendous help and advice. I find myself a lucky person who had the chance to work directly with all committee members. It makes my dissertation process a pleasant experience. Sarang Joshi, my co-advisor, has brought me a different research viewpoint from the image processing community. Sarang has also taught me how to derive and construct a solid foundation for my research based on mathematical analysis. Jens Kruger worked together with me in almost all submissions. Jens is always available to share experience, to understand the difficulties that I meet, and to give me tireless support from the beginning of my research. I could not complete my thesis work without guidance and encouragement from Jens. Though the amount of time I worked with Joao Comba and Tom Fletcher was less than with the others, it always brought me great experience. I have learned from them not only the knowledge but also methodologies to do research. They gave me beautiful advice and helped me grow in my academic development. I'm very glad that Joao has come to work at our institute in this critical period of my research.

I'm very grateful to have the chance to do summer internships at the Deep Computing group at the IBM Watson research center in 2007 and at the Quantitative Visualization

Group of ExxonMobil Upstream Research company in 2009. My mentors at IBM: James Klosowski and Wagner Correa and my mentors at ExxonMobil: Dominique Gillard and Mark Dobin are the most excellent mentors who brought me working environment, research challenges, and a handful of industry experience that I could not have had in the academic environment. The idea of a dissertation work with significant research implications has originated during my internship in the industry where the importance of research is not only measured by its novel content but how it influences the research of co-workers.

Also, very important to this work is the supportive environment I found at Scientific Computing and Imaging Institute (SCI) where I worked until the end of May 2010. I appreciate my colleagues' interest in my work and their moral support, for which I would like to thank them very much. In particular, the creative atmosphere in the Visualization and Geometric Computing (VGC), originally with Huy Vo, Emanuele Santos, Carlos Scheidegger, John Schreiner, Steve Callahan, Erik Anderson, Louis Bavoil and later with Tiago Etiene supported my ascent to prevail with this dissertation. I wish to thank them all. I have collaborated with many people over the last years. In one way or another, they have influenced my thinking. Particularly important are the discussions I had with Marcel Prastawa, Guido Gerig, Thomas Fogal, and Sam Preston. I would like to thank them very much. In a similar vein, I would like to thank Nikhil Phatak and Le-Thuy Tran. I want to give special thanks to Thomas Fogal for his persistent support on revising the content and writing of not only the thesis but also other research submissions. Tom is always the first, critical reviewer who helps improve the writing as well as the technical content to make our papers strong submissions.

Last but not least, I want to express my appreciation and thankfulness to my parents and my brother in my home country who give me constant spiritual and financial support from the beginning of my PhD. Thanks to my Vietnamese friends at the University of Utah: Anh Vo, Hoa Nguyen, Huong Nguyen, Khiem Nguyen, Trang Pham, Thanh Huynh and many others, who bring me my home culture and beliefs, share happiness and difficulties with me, and provide me balance outside the academic life, which I see as indispensable factors for my academic success. For that I wish to thank them all.

CHAPTER 1

INTRODUCTION

1.1 Parallel computing overview

Parallel computing has transformed a number of science and engineering disciplines, including cosmology and astrophysics, environmental and climate modeling, plasma and condensed matter physics, bioinformatics and computational biology, quantum chromodynamics, device and semiconductor simulation, seismology, turbulence, societal health and safety, earthquakes, geophysical exploration and geoscience, materials science and computational nanotechnology, human/organizational system studies, stockpile stewardship, signals intelligence, defense, etc. [4, 7, 92] (Figure 1.1).

For example, consider cosmology and astrophysics, the study of the structure and evolution of the universe, where one of the most striking paradigm shifts has occurred. A

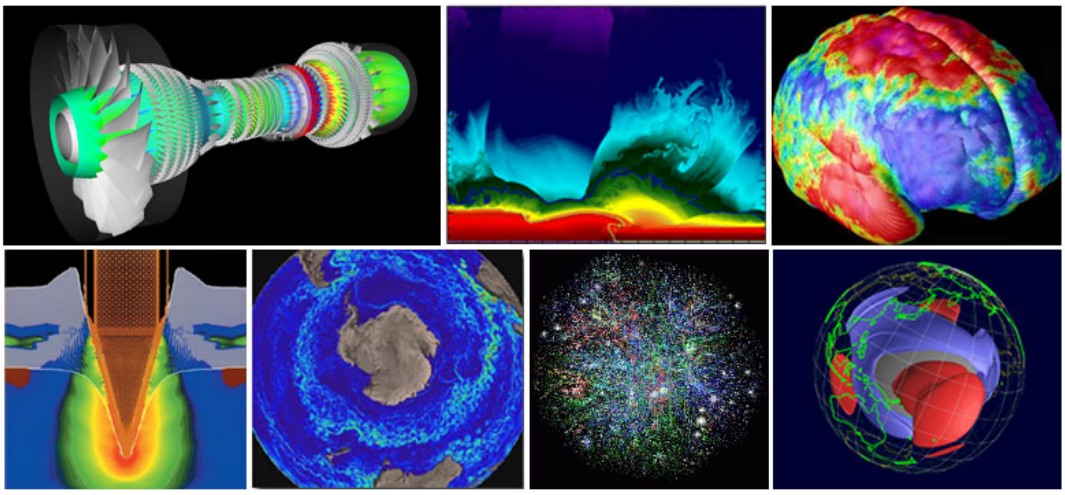


Figure 1.1. Parallel computing has been a driving force for the development of many scientific research disciplines and the solution for a number of engineering challenges in domain as diverse as mechanical engineering, nuclear physics, bio-science, applied physics, weather prediction, astronomy, geology, and more (Image courtesy of Blaise Barney, Lawrence Livermore National Laboratory [7])

number of new, tremendously detailed observations deep into the universe are available from such instruments as the Hubble Space Telescope [1] and the Digital Sky Survey. [2] However, until recently it has been difficult, except in relatively simple circumstances, to tease enough information from mathematical theories of the early universe to allow comparison with observations. Massively parallel computers with large memories, however, have changed all of that. Today, cosmologists can simulate the principal physical processes at work in the early universe over space-time volumes sufficiently large to determine the large scale structures predicted by theoretical models [42, 112]. With such tools, some theories can be discarded as being incompatible with observations [6]. High-performance computing has allowed comparison of theory with observation and thus has transformed the practice of cosmology.

Another example is bioinformatics and computational biology [129], especially in molecular biology, which seeks to understand how cells and systems of cells function, with the goal of improving human health, longevity, and the treatment of diseases. Computer simulations remain the only approach to understand the dynamics of macromolecules and their assemblies. Understanding the characteristics of protein interaction networks and protein-complex networks formed by all the proteins of an organism requires tremendous computational resources. Even when knowledge-based constraints are employed, the protein-folding problem—computational modeling and prediction of protein structures to understand the mechanism that translates gens into proteins—remains computationally intractable.

The complexity of molecular systems, in terms of both the number of molecules and the types of molecules, demands computation to simulate and codify their logical structure [104, 116]. There has been a paradigm shift in the nature of biological computing with the decoding of the human genome and with the technologies this achievement enabled. Equations of physics-based computation are now complemented by massive-data-driven computations and heuristic biological knowledge. In addition to the deployment of statistical methods for large data processing, a countless number of data mining and pattern recognition algorithms are being developed and employed [25, 125]. Finding multiple alignments in the sequences of hundreds of bacterial genomes is a computational

[1] `http://www.nasa.gov/mission_pages/hubble/main/index.html`

[2] `http://archive.stsci.edu/cgi-bin/dss_form`

problem that can be attempted only with novel alignment algorithms on peta-scale supercomputing resources [3, 9]. Large-scale gene identification, annotation, and clustering expressed sequence tags are other large-scale computational problems in genomics [40].

The capability to perform predictive simulations of biochemical processes transform our ability to understand the chemical basis of biological functions. This greatly improves our ability to design new therapeutic drugs, treat diseases, and understand the mechanisms of genetic disorders in addition to its value in basic biological research.

All the experiences from the development of super computing in the late twentieth century as well as hybrid computing in the recent ten years [114] have taught us the importance of building a firm scientific foundation using scalable, parallel computing, which allows us to expand and validate mathematical theories, and to compare simulation experiment and observation. We also have learned that the consistency of the programming model more than the intricacies of the hardware led us to the target. Parallel computing now has a big influence on everyday life and research by providing more accurate, detailed, and trusted predictions. However, moving entire research disciplines to the parallel computing world has imposed significant challenges.

1.1.1 Modern trends in parallel computing

The improvement of processing power has been driven by the Moore's Law [86] which predicts a long term development of fabrication techniques that doubles transistor density every two years. Nowadays, after more than four decades, the principle is still going strong [68]. Though this tendency is likely to be kept for another decade or more the ever-increasing transistor density no longer delivers comparable performance improvements. Adding transistors adds wire delays and speed-to-memory issues. More aggressive single-core designs lead to greater complexity and heat. Furthermore, scalar processors themselves have a fundamental limitation: a design based on serial execution, which makes it extremely difficult to extract instruction-level parallelism from application codes.

These issues are no longer the concern of only high-end users. It is becoming more apparent that major performance improvements could have a profound effect on virtually every scientific field. The demands for trans-petaflop systems require the development of new strategies to augment Moore's Law and to explore innovative High Performance Computing (HPC) architectures that can work around the limitations of conventional systems. These strategies include:

- *Multicore systems* that fabricate two or more cores on a die to continue providing steady performance gains.
- *Specialized processors* that enhance performance in areas where conventional models are inadequate.
- *Heterogeneous computing architectures*, in which conventional and specialized processors work cooperatively.

Each of these strategies can potentially deliver substantial performance improvements.

1.1.1.1 Multicore systems

Placing multiple cores on a die is the fastest way to deliver continuous performance gains in line with Moore's Law. Well-known examples of multiple-core processors are the AMD Opteron [69] and the Intel Xeon [103]. This strategy offers immediate multiple factors of computing density, while reducing per-processor power consumption and heat.

Although multicore processing provides a steady performance gain for many applications, especially those requiring heavy floating-point operations, for other applications which depend on heavy bit manipulation and/or massive data bandwidth such as sorting, signal processing, database searching, data encryption/decryption, improvement of the raw computational power is not enough. These applications often require speeds and memory bandwidth in orders of magnitude beyond what are available today through conventional processors [53]. It is unlikely that the benefit of having multiple fully generic processing cores will grow at the same rate as the transistor integration.

1.1.1.2 Specialized processors

In recent years, architectures based on clusters of commodity processors have overtaken high-end, specialized systems in the HPC community due to their low cost and solid performance for many applications. However, as users begin to experience the inherent limitations in terms of scalability of scalar processing, we are beginning to see a reversal in that trend [64, 114]. Examples of this resurgence include:

- *Vector processors*: vector processors increase computational performance by efficiently pipelining identical calculations on large streams of data, eliminating the rate limitation of instruction of conventional processors [64].
- *Multithreaded processors* [121]: the memory speeds have been increasing at only a fraction of the rate of processor speeds, leading to performance bottlenecks as

serial processors wait for memory. Systems incorporating multithreaded processors such as Intel's Hyper-Threading [79] address this issue by modifying the processor architecture to execute multiple threads simultaneously, while sharing memory and bandwidth resources to increase the memory bandwidth utilization.

- *Digital Signal Processors* (DSPs): DSPs are optimized for processing a continuous signal, making them extremely useful for audio, video and radar applications [110]. Their low power consumption also makes these processors ideal for use in plasma TVs, cell phones and other embedded devices.
- *Specialized coprocessors*: coprocessors such as graphic processing units(GPUs), n-body accelerator such as GRAPE, and FPGAs use multi-simple-core - array processor architectures to provide a large number of arithmetic logical units and floating-point components (multiply/add units) per chip. They can deliver noticeable improvements on mathematically intense functions, such as multiplying, inverting matrices, and visualization.

Processors such as these can deliver substantially better performance than general-purpose processors for some operations. Vector and multithreaded processors are also latency tolerant and can continue executing instructions even when large numbers of memory references are simultaneously underway. These enhancements can allow significant application performance improvement, while reducing intercache communication burdens and real estate on the chip required by conventional caching strategies.

Since specialized processors have traditionally been deployed, they have had serious limitations. Although they can provide excellent acceleration for some operations, they often run scalar code much slower than commodity processors. However, most software used in the real world employs at least some scalar code. Furthermore, these processors traditionally have been incorporated into more conventional systems via the PCI bus-as a peripheral. The inadequate communication bandwidth severely limits the acceleration that can be achieved. Communicating a result back to the conventional system may actually take more time than the calculation itself.

There are also hard economic realities of processor fabrication. Unless the processor has a well-developed market niche that will support commodity production, such as the applicability of DSPs to consumer electronics, few manufacturers are willing to take on the large cost of bringing new designs to market.

These issues lead us to alternative models such as heterogeneous computing models. While it turns out to be very close to the specialized processor model, it attacks the latency and bandwidth issues while allowing mass production support in the guise of graphics processing units.

1.1.1.3 Heterogeneous computing

Heterogeneous computing is the strategy of deploying multiple types of processing units within a single workflow. Each unit performs the tasks to which it is best suited. The model employs specialized processors to accelerate some operations to several magnitudes faster than what scalar processors can achieve, and at the same time it expands the applicability of conventional microprocessor architectures. Different from specialized processor models, heterogeneous models tightly couple processing elements in a single system to exploit the high performance communication bridges to connect between them, significantly reducing the latency between computation units and commodity control hardware.

The main advantage of this model is that HPC applications typically include both code that benefits from acceleration and code that is suited for conventional processing. While there is not a single type of processor that is best for all computations, heterogeneous processing models allow better utilization and performance by using the right processor types for each operation.

Traditionally, there have been two primary barriers to widespread adoption of heterogeneous architectures: the programming complexity required to distribute workloads across multiple processors and the additional effort to communicate between processors of different types. These issues can be substantial, so any potential advantages of a heterogeneous approach must be weighed against the cost and resources required to overcome them.

Nowadays, the rise of multicore systems has already created technology demands that largely change the programming perspective of the HPC software developer, opening the door to new programming strategies and environments. As software designers become more comfortable programming on the multiprocessor platform, they are willing to consider other types of architectures, including heterogeneous systems.

There are several new heterogeneous systems emerging recently. The Cray X1E supercomputer, for example, incorporates both vector processing and scalar processing. The Cell processor architecture (designed by IBM, Sony and Toshiba to accelerate gaming

applications on the new Playstation 3), uses a conventional processor to offload computationally intensive tasks to synergistic processing elements with direct access to memory. Field Programmable Gate Arrays (FPGAs), hardware-reconfigurable devices that can be redesigned to solve specific types of problems efficiently, are attracting strong interests to use as reconfigurable coprocessors [66]. However, the most exciting areas of heterogeneous computing emerging today employ Graphic Processing Units, or GPUs.

1.1.2 Parallel computing challenges

The increase in the accuracy, detail and volume of observation data requires a hand-in-hand development of high performance computing. The moving of the computation from 2D to 3D, even to n-D has demanded not only massive computing power but also novel parallel caching techniques and sophisticated bandwidth strategies. Parallel computing requires the development of advanced data preprocessing, data compression, out-of-core processing, message-passing, and compiling techniques to ease this transition.

Porting code to parallel architecture is much more than simply bringing up an existing code to a new machine [21, 24, 58]; it often presents an opportunity to reformulate the basic code and data structures, more importantly to reevaluate the basic representation of the data or the mapping of the algorithm and its efficiency on the new architecture.

To become a successful high-end technology, a persistent programming model for scalable, parallel computers is essential. This means providing a stable effective programming model over the life time of the application. Application developers need principles and tools that would survive in the long term and isolate them from the changing nature of underlying hardware. On the other hand, they also need the capability to exploit new hardware features and new parallel algorithms. This is even a challenge with the conventional parallel computing model. The software development on supercomputers, for instance, is often highly optimized for specific models, and requires entire code revision to adapt to new hardware. The principal goal of high performance computing has been the development of software and algorithms that address the programmability, portability, and flexibility of parallel applications [43, 34].

However, the expression of an explicit parallel programming models is difficult. The developers often have to specify not only how to partition data and computation among processors but also the data movement and synchronization to achieve high performance and to ensure correctness. Besides, portability is hard to define and difficult to achieve. Different application programming interfaces come from different vendors without a cross-

platform standard, making it tedious to convert the program to run on new platforms. However, portability is not a just a matter of a common interface. Though it is possible to express the program in a reasonably machine independent way, this increased portability often comes at the price of performance. The ability to achieve the highest performance possible on each machine from the same program image, *portable performance*, is a very important topic in the science of parallel computing [90].

In addition, the algorithms themselves are not always portable. To achieve the highest performance, algorithms often need to select a different parameterization specific to the machine it will run on. The changing in the parallel granularity, memory hierarchy and bandwidth, and also caching strategies makes portable programming even more difficult.

Another challenge comes from the complexity of the problems which requires different and extraordinary skills from the application developers. Often it means multiple programming paradigms, and multiple programming languages potentially must co-exist. Interoperability is an indispensable consideration of parallel computing and also a challenge in designing parallel computing languages.

The success of a parallel computing model depends on how sufficiently it addresses these challenges. This explains the convergence in HPC computing to unified architecture and programming models [117] and why hybrid computing, especially GPU computing, is emerging as a major trend in the parallel processing community, gradually replacing conventional computing models.

1.2 GPU computing

A graphics processing unit is a specialized coprocessor that offloads and accelerates 3D or 2D graphics operations from the central processors. There are two primary forms of GPUs: the discrete video cards and those integrated on the main system. In either form, the GPU is an essential, indispensable component of many commodity systems. GPUs have been using in embedded systems, smart phones, personal computers, game-consoles, workstations, etc. [39, 73, 89, 92, 101]. The orthodox appearance is the biggest advantage of GPUs over other specialized coprocessors and secures mass production support, research attention and constant technical improvement.

GPUs started as fixed-function graphic accelerators (Figure 1.2.a), which contain special mathematical operations and a number of graphics primitive operations commonly used in rendering [36]. Over the years, GPUs have become increasingly more powerful

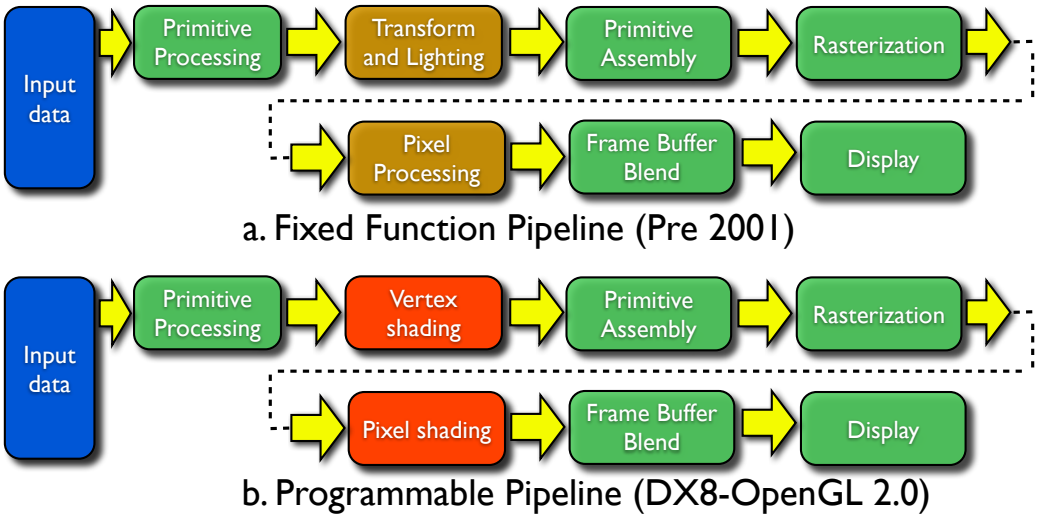


Figure 1.2. The development of GPU processing pipeline from a) a fixed function pipeline to b) a programmable pipeline is a prerequisite for General Purpose Computing on GPUs (GPGPUs).

and programmable with demands to support complex and high-quality scientific visualization [97, 96]. Nowadays, GPUs can deliver up to a teraflop of computing power from the same silicon area as a comparable microprocessor using a small fraction of the power per calculation: higher performance in a smaller footprint, at a lower cost, and using less power. The ability to drive raw computational power and memory bandwidth equivalent to supercomputers in the mid-90s on commodity devices makes GPUs an attractive approach to bring supercomputing power to regular users and to uphold Moore's Law.

In the early 2000s, computer scientists along with researchers in medical imaging and electromagnetics started using GPUs for running general purpose computational applications [85, 119]. They found the excellent floating point performance in GPUs led to a huge performance boost for a range of scientific applications. This was the advent of the movement called GPGPU or General Purpose computing on GPUs [77]. The initial attempts had defined the potential and essential functionality to transform GPUs from specialized coprocessors to more general purpose HPC units.

However, GPUs have had their own historical barriers to widespread adoption. First, they traditionally have been integrated into conventional systems via the PCI bus, which limits their effectiveness compared to other specialized processors mentioned above. More critically, the difficulty in mapping scientific algorithms and data structures to the ren-

dering of graphical primitives is a major obstacle for its use in general HPC problems. Fortunately, for the attractiveness of the raw computational power provided by modern GPUs and the popularity of GPUs in visualization, graphic programming languages such as OpenGL, CG and DirectX have been widely accepted by application programmers, including GPGPU developers.

Graphic vendors have realized the potential to bring this performance to a larger research community and invest in redefining GPU architectures, providing the fully programmable capability and development support for scientific applications [5, 91, 117]. The adoption of high-level languages such as C, C++, and later FORTRAN, the introduction of unified parallel programming models (CUDA, OpenCL, Direct Compute) make it easier for HPC developers to access the GPU computing potential.

The development of the communication channel between GPUs and conventional processors has increased the transfer bandwidth and significantly reduced the data latency. Starting with the introduction of Accelerated Graphic Port (AGP) from 1997 [38], an alternative of PCI bus - a dedicated pathway between a slot and conventional processors, APG 3.5 was capable of delivering transfer rate up to 2.133 Mbps. In 2004, AGP was replaced by PCI express (PCIe) [22]. PCIe 3.0 standard hardware is capable of 16 Gbps transfer rate almost equivalent to the CPU memory bandwidth. In addition to a dedicated communication path between devices, modern GPUs allow an asynchronous execution model that overlaps between computation and data transfer, an effective mechanism to hide the data transfer latency from the computation.

Consequentially, GPUs have been widely adopted in HPC community, increasingly being used to accelerate a wide range of science and engineering applications, in many cases offering dramatically increased performance in comparison to CPUs. In practice, GPUs can compute 100x faster than even the fastest general-purpose processors for some computational problems. Significant biomolecular, computational chemistry, astrophysical, condensed matter physic, weather modeling and seismic stack migration applications have already benefited substantially from or show substantial promise for using GPUs [12, 73, 104].

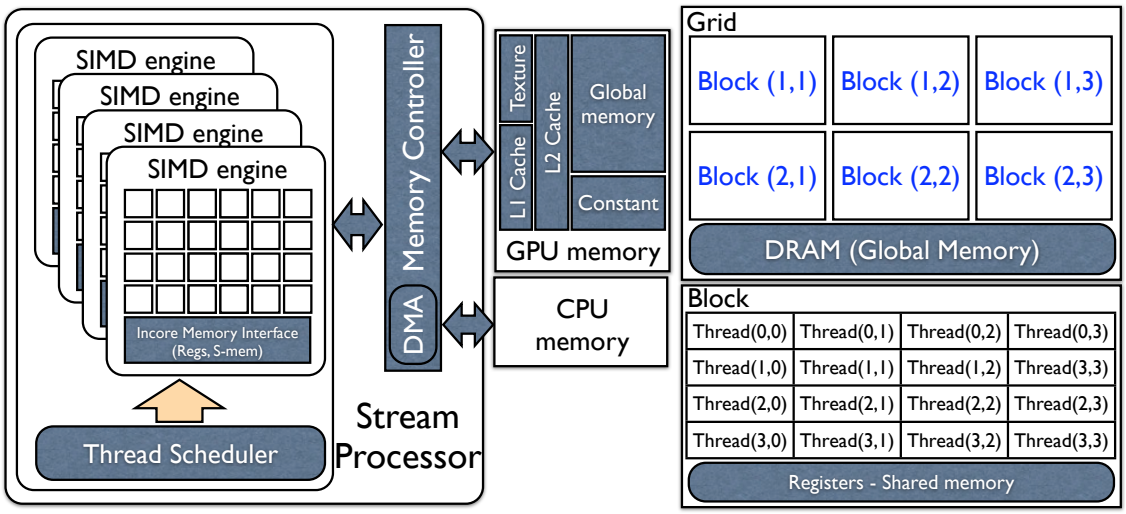
1.2.1 GPU computational model

GPUs are regarded as high-throughput processors that can achieve theoretical peak performance of several tera-flops. GPUs operate on an SIMT (single-instruction multiple thread) basis where thousands of light weight threads execute the same instruction

simultaneously. Much like the SIMD processor, GPUs, however, allow different levels of SIMD execution that only require all cores in the same group (*multiprocessor* or a *wavefront*) execute the same instruction at the same time. Different groups could execute different (or the same) instructions. Furthermore, SIMT handles conditionals somewhat differently than SIMD, where some cores are disabled for conditional operations.

At a broad level, the GPUs consist of several streaming multiprocessors and each of them contains a number of streaming processors and a small shared memory unit (Figure 1.3). For example, an NVIDIA GeForce GTX 580 GPU has 512 processor cores, and a Radeon HD 6870 GPU from AMD has 1120 processors, and each of those processors has five ALUs. There is a global memory that is accessible to all the streaming multiprocessors. The shared memory between streaming processors of the same group has very low latency comparable to processor a register file, is programmable and can be used to coordinate the computation between streaming processors. The GPU memory system provides much higher bandwidth compared to the CPU memory system, but has a higher latency. The caches used in GPUs are relatively small as compared to those used in CPUs. Recent GPUs also support a two-level cache hierarchy.

GPUs can be abstracted as stream processors, which are good at handling data streams



a. Streaming architecture of Modern GPUs

b.GPU hierarchy execution model

Figure 1.3. Hardware architecture and execution model of modern GPUs. Modern GPUs are modeled as stream processors, with a large number of simple, compute centric cores compounded with a high bandwidth parallel memory interface. The multilevel threading hierarchy allows efficient parallel execution model with a fine-grain approach

in parallel with kernels [21, 20]. In this model, the underlying program structure can be described by streams of data passing through computation kernels. A stream is an ordered set of data, and a kernel performs operations on streams in parallel. Given a set of data (an input stream), a series of operations (kernel functions) are applied to each element in the stream and produce another set of output data (an output stream). The program is constructed by chaining these computations together. This formulation has been used to design efficient GPU-based sorting and numerical computations [55, 60, 82].

On modern GPUs, the kernels are executed by multiple threads, which are organized into a two-level hierarchy: blocks and threads. At the top level of the hierarchy, a grid is organized as a two-dimensional array of blocks. At the bottom level, all blocks of a grid are organized into an array of threads. All the threads in the same block can access a small, high-speed shared memory. However, the threads from different blocks can only communicate through relatively slower global memory. GPUs have a texture cache, which exploits both spatial and temporal locality. If a thread accesses a memory location, the next access to a nearby location will be cached. Furthermore, threads in the same group share a common texture cache so that if they request the same data it will be in cache. The texture cache also enables fast and efficient interpolation and filtering. A recent GPU architecture, Fermi, even supports two levels of cache: an L1 cache and an L2 cache. The L1 and L2 caches improve performance for programs with random memory access patterns such as ray tracing and physics. The shared memory could be interpreted as an explicit cache shared between multiple threads of the same block, and so greatly helps improve the performance of GPGPU applications such as video transcoding and image processing.

GPU programming methodologies such as NVIDIA's Compute Unified Device Architecture (CUDA) [30], Khronos Group's Open Computing Language [5] and Microsoft's Direct Compute [15] allow developers access to the virtual instruction set and memory of the parallel computational elements in modern GPUs via "C-extension"-programming languages. The "C"-like working environment enables compilers to optimize the source code to utilized the accelerated hardware. This is also fortified with high-level C++ features such as template and object-oriented programming to facilitate a development process and lower the learning curve. Though we use CUDA for our development, the convergence in the hardware architecture and programming models to a unified model allows us a smooth transition to other GPU methodologies (OpenCL, Direct Compute).

This development is mandatory to maintain cross-platform efficiency and a key solution for portable performance.

1.3 Atlas construction problem

The construction of population atlases (Figure 1.4) plays a central role in medical image analysis, particularly in understanding the variability of brain anatomy. The method projects a large set of images to a common coordinate system, creating a statistical average model of the population, and doing regression analysis of anatomical structures. This average serves as a deformable template which maps detailed atlas data such as structural, developmental, genetic, pathological, and functional information on to the individual or entire population of the brain. This transformation encodes the variability of the population under study. Likewise, the statistical analysis of the transformation between populations reflects the inter-population differences. Apart from providing a common coordinate system, the atlas can be partitioned and labeled, thus providing effective segmentation via registration of anatomical labels (Figure 1.5).

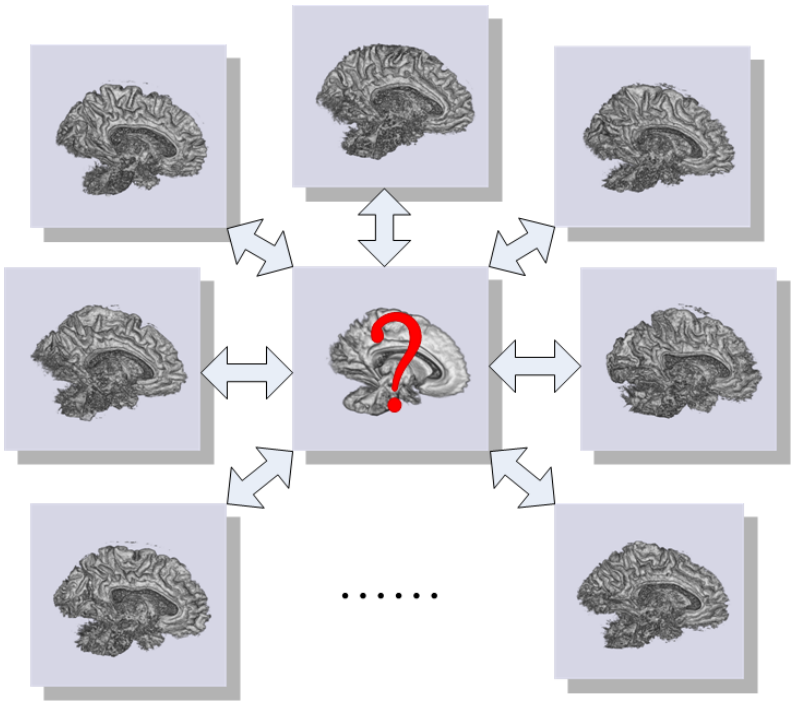


Figure 1.4. Brain atlas construction from a population. Here we implement an unbiased atlas construction approach based on averaging in diffeomorphic space.

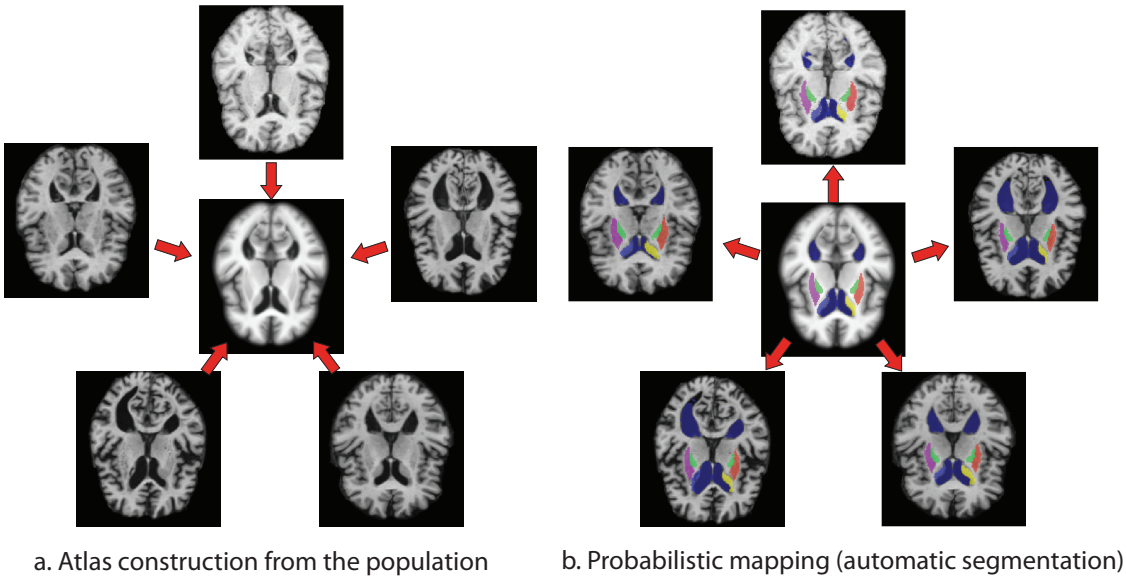


Figure 1.5. Automatic segmentation via atlas construction. The process includes two steps: a) Construct the brain atlas from the population - determine the mapping between each image and the atlas. b) Partition the atlas - the segmentation on an individual is done automatically via a reverse mapping from the atlas.

Brain atlas construction is a powerful technique to study the physiological and evolutionary development of the brain, as well as disease progression. Two desired properties of the atlas construction are that it should be diffeomorphic and nonbiased.

In nonrigid registration problems, the desired transformations are often constrained to be diffeomorphic, i.e., continuous, one to one (invertible) and smooth with a smooth inverse so that the topology is maintained. Connected sets remain connected, disjoint sets remain disjoint, neighbor relationships between structures as well as smoothness of features such as curves are preserved, and coordinates are transformed consistently.

Preserving topology is important for synthesizing the atlas since the knowledge base of the atlas is transferred to the target anatomy through topology preserving transformation which provided automatic labeling and segmentation. Moreover, important statistics such as the total volume of a nucleus, the ventricles, or the cortical subregion can be generated automatically. The diffeomorphic mapping from the atlas to the target can be used to study the physical properties of the target anatomy such as mean shape and variation. Also, the registration of multiple individuals to a standard atlas coordinate space removes the individual anatomical variation and allows information to be combined with a single

conical anatomy. Figure 1.6 shows that the diffeomorphic setting results in a high quality deformation field which is infinitely smooth on a nonself-crossing grid.

The nonbias property guarantees that the atlas construction is consistent. Our atlas construction framework, first proposed by Joshi *et al.* [67], is based on the notion of Frechet mean to define a geometrical average. On a metric space M, with a distance $d : M \times M \rightarrow R$ the intrinsic average μ of a collection of data x_i is defined as a minimizer of the sum-of-square distances to each individual, that is

$$\mu = \underset{p \in M}{\operatorname{argmin}} \sum_{i=1}^{N} w_i d^2(p, x_i)$$

As the computation of the Frechet mean is independent from the order of the inputs, the atlas is inherently nonbiased. The Frechet mean is also rational in terms of minimizing the total energy to deform an average to all images in a population.

The combination of both diffeomorphic and nonbias property results in a minimization energy template problem which is formulated as

$$\{\hat{h}_i, \hat{I}_i\} = \underset{h_i, I}{\operatorname{argmin}} \sum_{i=1}^{N} \int_{\omega} (I_i \circ h_i - I)^2 + \int_0^1 \int_{\omega} ||Lv_i(x,t)||^2 dx dt \tag{1.1}$$

subject to $h_i(x) = \int_0^1 v_i(h_i(x,t),t)dt$ (*)

This is a dual optimization problem on the image matching (the first term) and deformation energy (the second term). The L-operator is a partial differential operator which controls the smoothness of the deformation field. The constraint (*) comes from the theory of large deformation diffeomorphism that the transformations h_i are generated by integrating velocity field v_i forward in time. The method is the extension of elastic registration to handle large deformations.

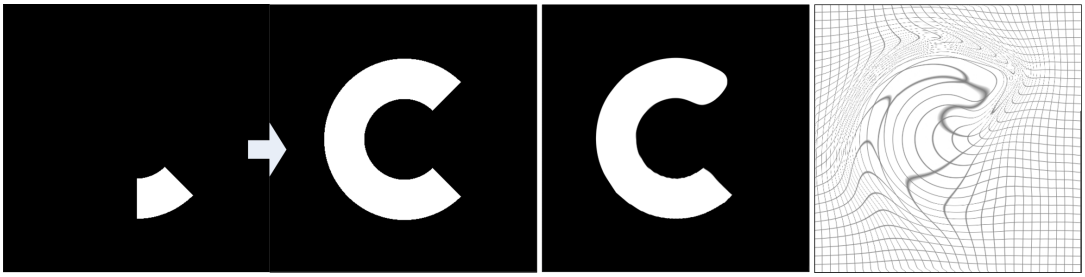


Figure 1.6. A small part of the letter "C" deforming into a full "C" using 2D Greedy Iterative Diffeomorphism. From left to right: 1. Input and Target Image 2. Deformed template. 3. Grid showing the deformation applied to template.

While the optimization seems intractable, by noting that for fixed transformation h_i the best estimation of the average $\hat{I}$ is given by $\hat{I}(x) = \frac{1}{N}\sum_{i=1}^{N} I_i(h_i)$, we come up with a simple solution based on an alternating optimization, as shown on Algorithm 1. In each step, we estimate the atlas by averaging the deformed images, then we compute the optimal velocity fields by solving optimization problems on deformation energy, finally the deformed images are updated. This process is repeated until convergence. Note that with the assumption of a fixed template on the second step, the optimal velocity of an image can be computed independently from the others. This velocity is determined by solving the pairwise matching problem. By tightly coupling the atlas construction problem with basic registration problems—the pair-wise matching algorithms—our framework allows one to implement different techniques and even to combine multiple techniques into a hybrid approach.

1.4 Challenges

1.4.1 Baseline research challenges

One of the interesting features of the atlas construction problem is that it is not a single processing algorithm but rather a class of problems or an abstract algorithm that varies dependently on the image registration technique being used. There are multiple diffeomorphic registration techniques such as the Greedy Iterative, the Large Deformation Diffeomorphic Metric Mapping (LDDMM) and the Metamorphoses. The atlas construction algorithm requires registering hundreds of brain images in a dual-optimization iterative process. Since each technique has a different trade-off between quality of results and the computation involved, there is no ultimate solution. One research challenge is how to quantify the trade-off to suggest a good solution for the problem based on the

Algorithm 1 Atlas construction framework

Input : N volume inputs
Output: Template atlas volume
for $k = 1$ to max_iters **do**
  Fix images I_i^k, compute the template $\hat{I^k} = \frac{1}{N}\frac{\sum_{i=1}^{N} I_i^k w_i}{\sum_{i=1}^{N} w_i}$
  for $i = 1$ to N **do** {loop over the images}
    Fix the template $\hat{I^k}$, solve pairwise-matching problem between I_i^k and $\hat{I^k}$
    Update deformed image I_i^k with current velocity
  end for
end for

inputs and the accessible computational power. This objective demands the approaches to be general and extensible so that we can incorporate and compare different techniques. Our solution for this problem is a registration framework based on generic programming, in particular, C++ template. We discuss this solution in detail in Chapter 2.

Even though large-diffeomorphic registration has long been studied, deformable image registration in the presence of considerable contrast differences and large size and shape changes still represents a significant challenge for image registration. A representative driving application is the study of early brain development in neuroimaging as the growth process can involve very large variation in size and shape, as well as changes in tissue properties and appearance. This requires registration methods to handle large-scale and also nonlinear changes. Furthermore, the process of white matter myelination, which manifests as two distinct white-matter-appearance patterns primarily during the first year of development, imposes another significant challenge as image intensities need to be interpreted differently at different stages. We proposed a new registration method that enhances the registration quality by integrating information of critical landmarks into a conventional large-diffeomorphic registration framework. For more details on the problem and our solution, see Chapter 3.

1.4.2 Efficient implementation challenges

Unbiased diffeomorphic atlas construction, total variation, bounded variation and level-set construction are examples of advanced image-processing functions, powerful algorithms using in computational anatomy. However, the impact of these methods was limited in practice because of two primary challenges: the extensive memory requirements and the intensive computation.

The extensive memory requirement is one of the major obstacles of 3D volume processing in general, as the size of a single volume often exceeds the available memory on a processing node. This becomes more challenging on GPUs as they have even less memory. In addition, the atlas construction requires not just a single volume but a population of hundreds of volumes, which easily exceeds the available memory of practically any computational system. The massive size of the problem is compounded with the complexity of the computation per element. These computations are often not just simple, local kernels but global operations, e.g., an ODE integration using a backward mapping technique.

Generating a brain atlas at an acceptable resolution for a reasonably sized population

takes an impractically long time even with a fully optimized implementation on high-end CPU workstations or small CPU clusters [33]. Acceptable run times could only be obtained by utilizing supercomputer resources. [29, 18]. Consequently, the influence of these techniques in the research community was restricted.

These basic challenges will be addressed in the next chapter. The computational requirement is solved using GPU computing models. Our results show that an implementation using GPUs can handle practical problems on a desktop with a substantial performance gain, on the order of 20 to 60 times faster than a single CPU. This solution is fortified by a multi-image processing framework that allows the solution to run entirely on GPUs to maximize the computational benefit and to resolve potential performance bottlenecks. Furthermore, we introduce a multi-parallel-level implementation that provides solutions from single-GPU desktops to multi-GPU workstations and GPU clusters. The solutions are based on different existing parallel schemes that are suitable for each parallel level. While this approach partially relieves the memory issue, the fourth chapter will wrap up the memory problem with an out-of-core multi-image processing framework. This technique can be generalized at different parallel levels to provide a complete solution for the memory problem.

As the GPU framework is built upon basic algorithmic building blocks, the efficiency of the model largely depends on how well the algorithms map to the GPU architecture. The differences in both the architecture and programming methodologies between GPUs and conventional CPU models make it a challenging but also exciting area of GPGPU research. The reason is that these baseline studies could have profound influences that direct the development of the GPU processing models, which strive to provide the most efficient solution for basic problems to facilitate the implementation of complex algorithms at a larger scale. Appendix A presents our optimized implementation of sorting algorithm on GPU. The results show that it is possible to implement an inherently sequential algorithm efficiently on GPUs.

1.5 Contributions

The contributions and novelty of this dissertation are:

- A general multiscale parallel framework for 3D image processing on GPUs, an optimized implementation of the atlas construction on multi-GPU systems, and

a GPU cluster implementation is used as the illustration for the effectiveness of the framework.

- A novel approach to solve the registration problem with large changes in the size, shape and tissue properties.
- An optimal, asynchronous streaming framework for multi-image processing.
- A high performance basic processing block - sorting.

In the second chapter, the dissertation addresses the problems of porting applications from CPUs to GPUs and the motivation of designing a high performance parallel framework for the 3D image processing. While there exist several general development packages on GPUs such as Thrust, and Nvidia Image Processing (NVP), these packages provide only basic functions for 2D image processing. Our framework targets a complete solution for vector computation and 3D image processing. It presents a hierarchical structure of development APIs from basic functions (low-level APIs) to advanced functions (algorithmic-level APIs) and data structures from regular grids to particle meshes. We also introduce essential optimization techniques to exploit the massive bandwidth and computational power of GPUs. These techniques not only provide a practical solution for the specific problem of image processing problem, but they are also beneficial for other computational tasks.

In the third chapter, the dissertation presents a novel approach that addresses the image registration problem in the presence of considerable contrast differences, large-scale size and shape changes, and also different tissue properties. The method makes use of underlying anatomies, which are represented by both class posteriors and boundary surfaces. This framework can match internal regions and simultaneously preserve a consistent mapping for the boundaries of relevant anatomical objects. We show results of registering neonatal brain MRI to 2-year-old brain MRI of the same subjects obtained in a longitudinal neuroimaging study. Our method consistently provides transformations that better preserve time-varying structures than those obtained by intensity-only registration. In addition, we consider a particle mesh method, used as a solution for the computational problem, as a bridge to connect the computation on regular domains and irregular domains to exploit the advantages from both sides. Furthermore, we yield a unified computation framework that can maximize computational benefits from existing parallel solutions.

In the fourth chapter, the dissertation attacks the memory issue - the primary reason that limits the use of GPU computing methodologies in large scale problems. The dissertation proposes an out-of-core multi-image framework that could handle a large number of images effectively on a single commodity computing system. The method is showed to be a feasible, economical and accessible solution for researchers, providing processing power and large memory space of supercomputing systems to their regular working desktops. The out-of-core framework brings opportunities to scientists to experiment and understand the impact of advanced techniques, which is previously limited due to the memory and computational constraints.

Appendix A is the showcase in which we explain how a sophisticated, inherently sequential algorithm such as sorting could be implemented efficiently on GPUs. The algorithm provides a high performance basic building block that could be exploited in many critical run-time algorithms and applications such as parallel ordering, collision detection, shooting optimization, splatting, etc.

To facilitate the GPU software development using our system, we present the software overview of the system in Appendix B. We discuss how our software architecture adapts to the changing in the programming methodologies and parallel hardware models. We discuss the scalability problem, how to minimize the memory control influence on scalability. Besides, we discuss the programming features and programming rules that we used in code development process to give users initial ideas about coding structure and styles to help them reduce the starting time, to lower the learning curve, and to encourage scientists to use our framework as computational solutions for their research.

Overall, we address the problem of designing a high-performance parallel 3D image-processing framework that is capable of exploiting the processing power at different levels: multicore GPUs, multi-GPU systems and GPU clusters. This framework is essential for the development of GPU computing as it helps developers and scientists reduce the development and maintenance cost of their applications, providing them the massive computing power at an abstract level without having to know the specific low-level detail of the underlying system. We also attack the out-of-core and scalability problem to give a complete solution to the problem. We provide an open-source noncommercial 3D image processing solution that is accessible to scientists. All the problems that we address here are still open problems that assure the research influence and novelty of the dissertation.

CHAPTER 2

GPU IMAGE PROCESSING FRAMEWORK

In this chapter, we present a high performance multiscale 3D image processing framework which can harness the parallel processing power of multiple graphic processing units (Multi-GPUs). We developed GPU algorithms and data structures that can be applied to a wide range of 3D image processing applications and efficiently exploit the computational power and massive bandwidth offered by modern GPUs. Our framework helps scientists solve computationally intensive problems, which previously required supercomputing power. To demonstrate the effectiveness of our framework and to compare to existing techniques, we focus our discussions on atlas construction.

First of all, we start with an overview about the framework and the motivation of why we want to build a framework instead of just a high performance processing library.

2.1 Framework overview

A software framework is a set of code or libraries, which provide common functionality to a class of applications. The basic difference between a framework and a library is the common generic functions or algorithms which target a certain type of application. While a library is considered a collection of discrete functions, a framework often offers a broader range of functions and reusable code abstractions wrapped in a well-defined application programming interface. Rather than rewriting commonly used logic, a programmer can leverage a framework, selectively overriding or specializing to provide specific functionality using their own code. Using a framework will limit the time required to build an application and reduce the possibility of introducing new bugs. Qt [14] is a well-known example of a cross-platform application and UI framework. Using Qt, you can write web-enabled applications once and deploy them across desktops, mobile and embedded operating systems.

The designers of frameworks aim to aid software development via a number of means:

- Using code which has already been built, tested, and used by other programmers increases reliability and reduces programming time and code maintenance. In other words, frameworks promise higher productivity, shorter time-to-market, and thus save money; hence they are critical for developing large-scale software systems.
- Frameworks assist code modularity, allowing programmers to exploit their specialties and to devote their time to meeting software requirements rather than dealing with the more standard, low-level details. For example, using our framework, software users could concentrate on experimental registration methods, while the mundane tasks of data IO input/output, out-of-core processing, and cluster communication are handled separately by the framework.
- Frameworks provide cross-application features that will benefit all the applications using the framework without extra time and cost of integrating them. An example is the uniform interface of Qt which helps to reduce amount of time deverlopers spending on making a user interface. Qt also lowers the learning curve for users, as they become familiar with the visual features of Qt platform.
- Frameworks often help enforce best practices for a platform. At the same time, they could give users the flexibility to select proper algorithmic solutions rather than being strict in a single implementation strategy. Our framework provides the GPU optimized functions for n-ary operators which are significantly faster than even functions using optimized binary operators.
- Upgrading the frameworks automatically enhances the application functionality without extra programming by leaf application developers.

All the benefits of a framework design assist our target of providing a stable development platform for regular scientists to exploit the GPUs processing power. Building a framework that could provide all the aforementioned advantages is a goal of this thesis research. Now that we have discussed the general advantages of a software framework, let us take a closer look at how we analyze and define specific functionality for our 3D image processing framework. First we start with the two basic diffeomorphic registration algorithms which are the core methods of this atlas construction framework.

2.2 Core methods

2.2.1 Diffeomorphic image registration algorithms

As we mentioned in the general atlas construction (Algorithm 1 of Chapter 1), the method is based on the image registration techniques. We discuss here the implementations of the two most common approaches in diffeomorphism space: the Greedy Iterative Algorithm and the Large Deformation Diffeomorphic Metric Mapping Algorithm.

2.2.1.1 Greedy iterative diffeomorphism

The Greedy Iterative Diffeomorphism was proposed by Christensen [28] in 1996. The method separates the time dimension from the space dimension of the problem. At each iteration, a new optimal velocity field is computed given that the current deformation is fixed (i.e., the past velocity fields are fixed). The solution is computed by integrating the optimal solution into a gradient descent approach at each step forward in time (Figure 2.1). The method is locally-in-time optimal which, consequently, reduces the dimension of the optimization problem significantly. At each time step, the algorithm attempts to greedily reach the target within a conservative step. In general, the method will not produce the shortest path connecting images through the space of diffeomorphism. However, the method requires less compute power than other approaches. Furthermore, the result is close to the optimal solution. Hence, this technique is generally preferred in practice. The Greedy Iterative Diffeomorphism is built on the general framework with the Greedy Pair-wise Matching algorithm at its core (Algorithm 2).

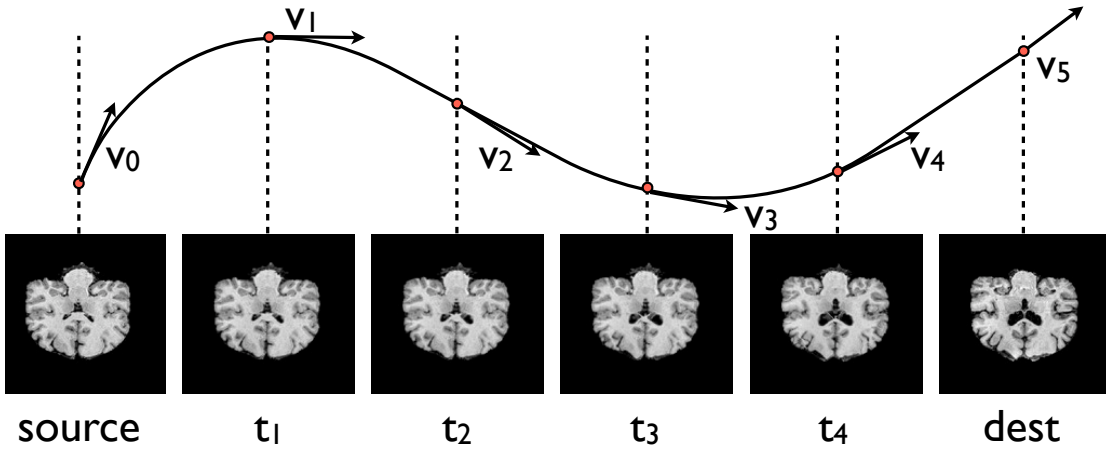


Figure 2.1. Forward itergration of the vector field in Greedy Iterative matching

Algorithm 2 Greedy pairwise matching step

1: **Input** : Original image I_0, target I_1, deformed image $I_0(t)$, deformation field h
2: **Output**: New deformed image $I_0(t)$, deformation field ϕ
3: Compute the force $F = -\left[I_0(t) - I_1\right] \bigtriangledown I_0(t)$
4: Solve the PDE $Lv(x) = F(x)$ where $L = \alpha\nabla^2 + \beta\nabla\nabla + \gamma$ is a smoothing operator. The values $\alpha = 0.01, \beta = 0.01$ and $\gamma = 0.001$ are used for brain images.
5: Update the deformation $\phi_{new} = \phi_{cur}\left(x + \epsilon v(x)\right)$
6: Update the transform image $I_0(t) = I_0\left(\phi_{new}(x)\right)$

2.2.1.2 Large Deformation Diffeomorphic Metric Mapping

While the Greedy Iterative method is less computationally expensive than the full Large Deformation Diffeomorphic Metric Mapping (LDDMM) per iteration, this advantage is impaired by the large number of iterations required by the method. Besides, the results are suffered from the local maximum problem related to gradient decent methods, in any cases it only produces an approximate solution for the problem. As we are capable of performing a large amount of computation in real-time on GPUs, we can apply the full LDDMM framework, which finds the exact solution for the problem, and assures quality results.

The full LDDMM framework is based on deriving the Euler equation of the variational minimization on the vector field. Following Beg *et al.* [10], the optimizing velocity field satisfies the Euler-Lagrange equation:

$$\partial_h E(\hat{v}) = \int_0^1 \langle 2v_t - K\left(\frac{-2}{\sigma^2}|D\phi^v_{t,1}|\left(J^0_t - J^1_t\right)\nabla J^0_t\right), ht\rangle_V dt = 0$$

where $J^0_t = I_0\phi_{t,0}$,$J^1_t = I_1\phi_{t,1}$. This equation leads to a LDDMM registration algorithm that is based on the standard steepest gradient decent scheme. In particular, the matching algorithm initializes iteration $k = 0$ with $v^k_{t_j} = 0, \nabla_{v^k} E_{t_j} = 0, \phi_{t_j,0} = I, \phi_{t_j,T} = Id$. For each iteration, it performs following steps:

1. Compute J^T_t backward in time for each time step

$$\phi_{t_j,T} = \phi_{t_{j+1},T}(x + v_{t_j})$$

2. Compute $D_{\phi_{t,T}}$ backward in time for each time step

$$|D_{\phi_{t_j}}| = |D_{\phi_{t_{j+1}}}| \times |D(x + v_t)|$$

3. Compute J_t^0 forward in time for each time step

$$\phi_{t_{j+1},0} = \phi_{t_j,0}(x - v_{t_j})$$

4. Compute the image gradient ∇J_t^0 for each time step
5. Compute the body force function $F_{t_j} = |D\phi_{t_j,T}^k|\nabla J_{t_j}^0(J_{t_j}^0 - J_{t_j}^T)$
6. Apply the Green kernel, solving the PDE equation $Lu = -F$ such that $L = -\alpha\nabla^2 + \gamma I$
7. Compute the energy gradient function $\nabla E_{t_j} = 2v_{t_j} - \frac{2}{\sigma^2}u_{t_j}$
8. Update the velocity based on the energy gradient $v^{k+1} = v^k - \epsilon\nabla E_{v_{t_j}^k}$

The intuition behind the LDDMM approach is that instead of looking at a local optimal estimation of the deformation field from the source to the destination image as proposed by the Greedy framework, LDDMM estimates the deformation in both backward and forward directions, as shown on Figure 2.2.

The Greedy Iterative and LDDMM algorithms are two examples of methods to solve the atlas construction problem by exploting the robustness of diffeomorphism space. A common bond between these methods is the large computational power and memory requirement. Even the simpler approach, the Greedy Iterative method, requires hours to complete on a high-end 32-core Intel Xeon server, at 2.93 Ghz and 256 Gb of memory. Here, in this chapter we introduce a faster and more economical solution based on GPU processing, which significantly reduces the processing runtime to a few minutes. The key to the performance is a contruction framework that is optimized and runs entirely on GPUs to achieve the maximum performance benefits.

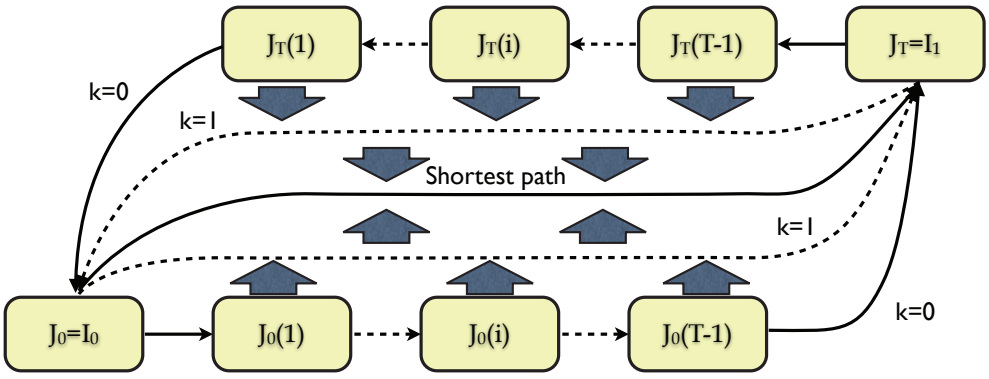


Figure 2.2. LDDMM estimate the transformation based on both forward and backward integration of vector field in two opposite directions between the source and the target

There are several performance keys of a GPU implementation: high throughput data structures and basic functions, high performance advanced functions: optimal ODE integration and PDE solvers, and multiresolution and multi-GPU strategies. We will discuss in detail how to achieve the peak performance in the following section.

2.3 High performance image processing framework on GPUs

2.3.1 Data structures

The data structure was built based on the Resource Acquisition Is Initialization paradigm (RAII), a simple, eloquent and efficient way to manage computational resources. The technique was invented by Bjarne Stroustrup [118] to deal with resource deallocation in C++. It is vital to build a thread-safe working platform. It ensures that the resource is acquired and released appropriately, especially in the case of errors. It helps assure that no resource leaks happen under exceptional control flow. It also makes the code cleaner and safer to use. We exploit the reference counting smart pointers (RCSP), particularly tr1::shared_ptr [113], as an alternative to raw pointers. This also facilitates our design of a programming interface which is easy to use correctly and hard to use incorrectly. Since debugging GPU functions is often a challenge, this design scheme minimizes the possibility of an error happening because of misusing functions.

2.3.1.1 Volume image presentation

We chose a tight 3D volume representation which can represent a 3D volume as a 1D vector. While it is typically recommended to have volume data padded to make volume dimensions be multiples of the GPU warp size ensuring data alignment, our experiments showed that it has negative effect, as the size of the input data increases it also increases the computational runtime as shown on Figure 2.3. Additionally, data padding significantly increases the storage requirement, especially in 3D. Furthermore, it requires extra processing steps and extra running parameters. It has negligible effect on improving performance because GPU data parallel fetching strategies have become more sophisticated and efficient. For example, the "coalesced condition"—the data alignment constraint to achieve maximum memory bandwidth—was eased from a strictly aligned boundary condition for both data and execution threads in CUDA 1.0, to a relaxed data-continuous condition for the data only in CUDA 2.0 hardware, consequently it is easy to achieve with regular array-based processing functions. Obviously, with our presentation,

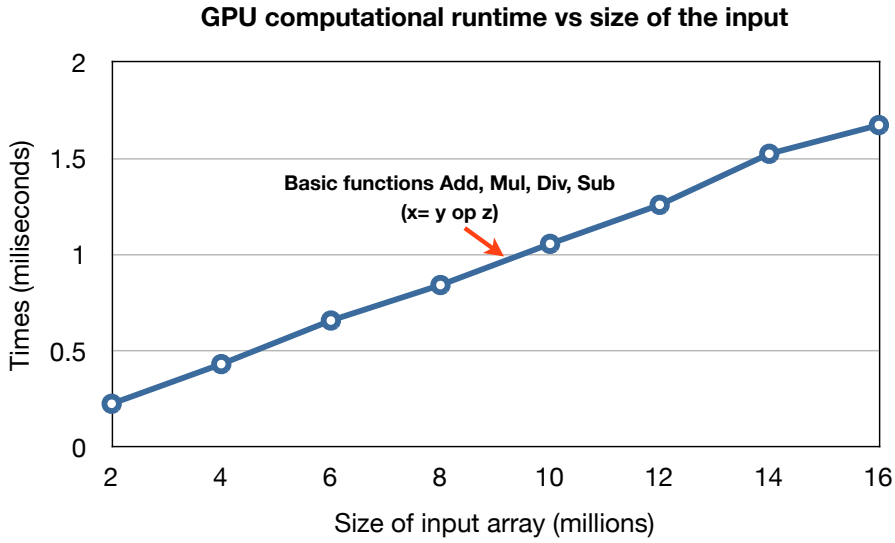


Figure 2.3. The computation runtime of basic GPU functions is linearly proportional to the size of the input data. We also observe the similar trends with other GPU functions, it explain why a tight volume presentation is generally preferred in our framework.

most of the basic operations on 1D can be directly applied to 3D. Additionally, we save two integer shared memory locations and operations per kernel by passing a single volume value instead of three dimensional numbers.

2.3.1.2 Vector field presentation

We also define a special structure for 3D vector fields based on a Structure of Array representation. As shown on Figure 2.4, instead of allocating three separated GPU pointers, we allocate a single memory block and use an offset to address the three components. This presentation allows us to optimize basic operations on the vector field, most of the time as a single image operation. Moreover, it helps us save one shared memory pointer per kernel. Note that the 256-boundary alignment is implicitly produced by CUDA memory allocation to achieve highest bandwidth efficiency. However it does not guarantee the continuity of three allocated arrays as we do here for 1D optimization.

2.3.2 Basic image operators

The goal of our system design is to be able to run the entire processing pipeline on GPUs. This allows one to maximize the computational benefit from GPUs and minimizes idle time. We keep data-flow running on the GPUs, and only use CPUs for cross GPU

```
template<typename T>
struct Vector3DArray{
  T* x, y, z;   // pointer to the component arrays
  std::tr1::shared_ptr<T> data;
  size_t nElems;   // number of elements
  size_t nAligns;  // alignment boundary module 256 or capacity of the array
  bool isContiguous() { return (nAligns == nElems); }
}
void Vector3DArray<T>::allocate(size_t n){
  nAligns = iAlignUp(n, 256);
  nElems  = n;
  data    = std::tr1::shared_ptr<T>(allocate<T>(nAligns * 3), deallocate<T>);
  this->x = data;
  this->y = this->x + nAligns; this->z = this->y + nAligns;
}
bool isOneDEquivalent(Vector3DArray& d_o, const Vector3DArray& d_i, size_t n)
{
  return (d_o.isContiguous() && d_i.isContiguous()
          && (n == d_o.nElems) && (n == d_i.nElems));
}
void Mul(Vector3DArray& d_O, const Vector3DArray& d_i, size_t n){
  if (isOneDEquivalent(d_o, d_i, n))
      Mul(d_o.x, d_i.x, n * 3);
  else {
      Mul(d_o.x, d_i.x, n);
      Mul(d_o.y, d_i.y, n);
      Mul(d_o.z, d_i.z, n);
  }
}
```

Figure 2.4. Vector field presentation and one-D optimization for vector field computation

and cross CPU operations. With the design goal in mind to be optimal, even on a per function level, we provide n-ary basic-3D functions.

The performance of the basic function is constrained by the global memory bandwidth. To improve the performance we need to minimize the bandwidth usage. Most of the functions provided by regular processing libraries such as Thrust [62] or NPP [94] are unary or binary functions which involve one or two arguments as the inputs. Though any n-argument function can always be decomposed into a set of unary and binary functions, this decomposition requires extra memory to store intermediate results, and increases bandwidth utilization by saving and/or reloading the data. Our n-ary operators, on the other hand, load all the components of an n-argument function to the register files at

the same time, and hence no extra saving/loading is required. This allows for optimal memory bandwidth usage. For example, if we consider the image loading operator being one memory bandwidth unit, then the linear interpolation $x = a*y+b*z$ requires 7 units with binary decomposition, while optimal bandwidth is 3 units which is achievable with n-ary operators. The bandwidth ratio is also our expected speed up of our n-ary versus binary functions. In terms of storage requirement, the binary decomposition doubles the memory requirement by introducing an extra template memory per operation, while n-ary functions require no extra memory.

In addition to providing all the basic operations similar to those of the Thrust library [62], we implement n-ary functions combining up to five operations. We also offer n-ary in-place operators which consume fewer registers and less shared memory. The name of these functions reflects their functionality, to preserve the readability and maintainability of the code and to allow further automatic code generation and optimization by the compiler. As shown in Figure 2.5, our normalization function and linear interpolation

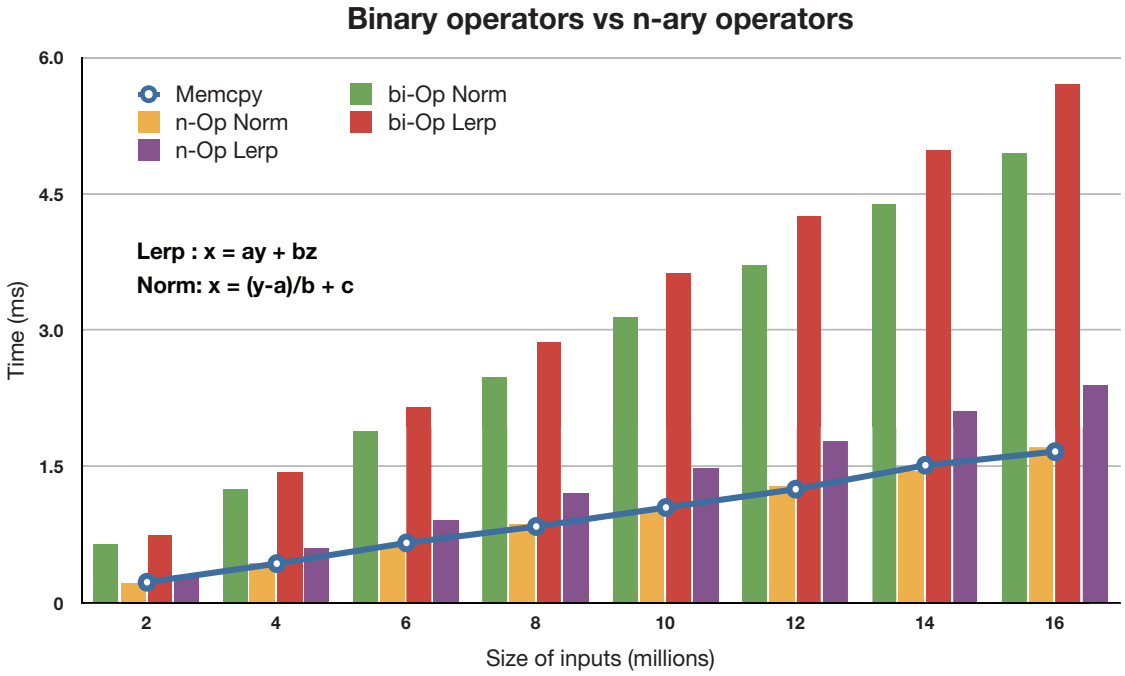


Figure 2.5. n-ary versus classic binary operator with linear interpolation and range normalization function. We use the memory copy from device to device, in other words, a no-op function as reference to show the optimality of our n-ary approach. Runtime is measured in milliseconds on an NVIDIA GTX 260.

achieves speed up factors of up to 2-3 over the implementation using optimized binary operators.

Based on the same strategy of n-ary operators, we propose a parallel efficient average function with hand-tuned performance for all number of inputs from 1 to 8 as illustrated in Figure 2.6.

2.3.2.1 Gradient computation

Gradient computation is a frequently-used and essential function in image processing. Based on the locality of the computation, several optimization techniques may be applied such as 1D linear texture cache, 3D texture, or implicit cache through shared memory. Among these techniques, we found the 3D stencil method [83] using the shared memory the most effective. Table 2.1 shows the runtime comparison in milliseconds of different gradient computations: simple approach, linear 1D texture, 3D texture and our shared memory implementation.

The result shows that gradient computation on shared memory exploiting 3D stencil technique is twice as fast as using the linear texture cache.

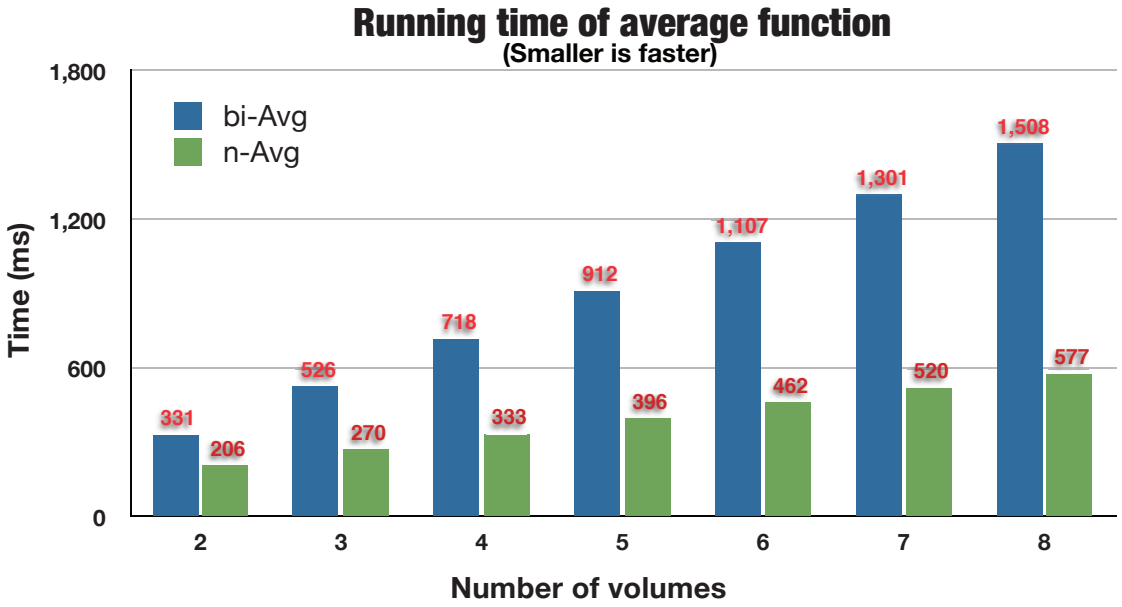


Figure 2.6. n-ary average function versus binary average operator

Table 2.1. Runtime comparison in milliseconds of different gradient computations: simple global memory, linear 1D texture, 3D texture and shared memory approaches

Gradient Method	Simple	1D Linear	3D texture	Shared
$160 \times 224 \times 160$	3.4	3.0	6.8	1.6
$256 \times 256 \times 256$	9.5	8.9	21	5.2

2.3.3 ODE integration

The ODE integration computes the deformation field by integrating velocity along the evolution path. A computationally efficient version of ODE integration is the recursive equation that computes the deformation at time t based on the deformation at time $t-1$, that is $h_t = h_{t-1}(x + v(t-1))$. This computation could be done by the reverse mapping operator (Figure 2.7), which assigns each destination grid point a 3D interpolation value from the grid neighbor points in the source volume. Fortunately, on GPUs, this interpolation process is fully hardware-accelerated with 3D texture volume support from CUDA 2.0 APIs.

Table 2.2 shows the runtime comparison in milliseconds of different 3D interpolation implementations: CPU reference, simple approach (GPU global memory), linear 1D texture, and 3D texture. The result shows that reverse mapping using the accelerated hardware achieves the best performance and is about 38x faster than a CPU-based reference implementation. This implementation, however, has a trade off of decreased floating-point accuracy. When high floating-point precision is needed, a better option is an implementation using 1D-linear texture cache.

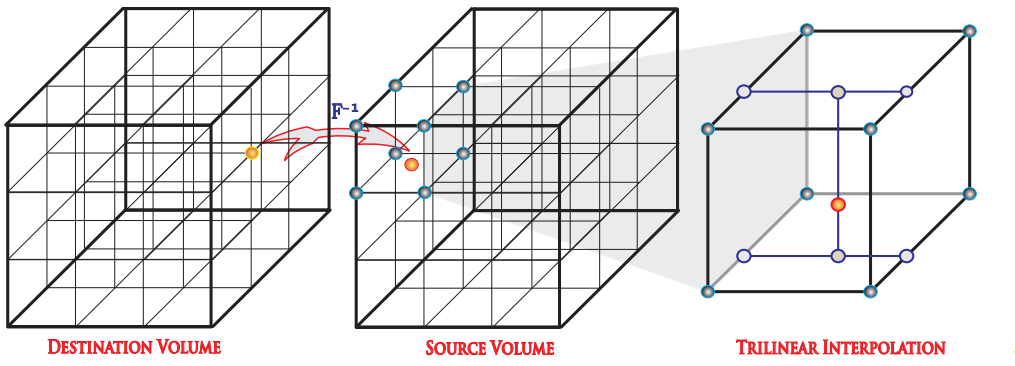


Figure 2.7. Reverse mapping based on 3D trilinear interpolation

Table 2.2. Runtime comparison in milliseconds of different 3D interpolation implementations for reverse mapping operator using global memory, 1D linear texture and 3D hardware-accelerated texture

Method	CPU	GPU global	1D Linear	3D texture
$256 \times 256 \times 256$	777	30	24	19
$160 \times 224 \times 160$	209	10.4	7.3	6.8
$144 \times 192 \times 160$	173	6.8	4.8	5.4

2.3.4 PDE Solver

As shown in Algorithm 2, optimal velocity is computed from the force function by solving the Navier-Stokes equation

$$\alpha\nabla^2 v(x) + \beta\nabla\nabla v(x) + \gamma v(x) = F(x) \tag{2.1}$$

Often β is negligible and Equation (2.1) simplifies to the Helmholtz equation

$$\alpha\nabla^2 v(x) + \gamma v(x) = F(x) \tag{2.2}$$

where $\alpha = 0.01$ and $\gamma = 0.001$ are generally used in practice. Note that there is no crossing term in the Helmholtz equation which means the solver could be run independently on each dimension.

While the ODE computation can be easily optimized simply by utilizing the 3D hardware interpolation, the PDE solver is less amenable to GPU implementation. The PDE is a sparse linear system with size $N^3 \times N^3$, where N^3 is the volume of the input. A direct dense linear package such as CUDA BLAS cannot handle the problem. What we need is a sparse solver. There are many different methods to solve a sparse linear system. The two most common and efficient ways are explicit solvers in the Fourier domain and implicit solvers using iterative refinement methods such as Conjugate Gradient (CG), Successive Over Relaxation (SOR) or multigrid.

In our framework we support different methods such as FFT, SOR, and CG. There are multiple reasons to support multiple techniques rather than a single method. Although the FFT solver is the slowest, it produces an exact PDE solution. While the others are significantly faster, they only produce approximate solutions, which often have local smoothing effects. The inability to account for the influence of spatially distant data points in the initial solution slows-down the convergence rate of these methods in the

long run. Consequently, they require more iterations to achieve the same result as the FFT approach. Due to smoothing properties of the velocity field, the variance in the solution of the PDE solver between two successive steps is often small. This variance can be captured adequately by the iterative solvers in a few iterations. This is made possible by using the previous solution as an initial guess for the iterative solver in the next step. For the first iteration, without a proper guess, iterative solvers are often slow to converge, so they require a large number of iterations and may quickly become slower than the FFT approach. Therefore, we use an FFT solver in the first iteration and then switch to iterative methods. Our experiments show that the hybrid CG solver that starts with an FFT step produces exactly the same results as an FFT method, but is almost three times faster.

For the details on the FFT solvers we refer the reader to [93]. Here we will discuss the implementation of SOR and CG methods.

2.3.5 Successive over relaxation method

Successive over-relaxation (SOR) is an iterative algorithm proposed by Young for solving a linear system [128]. Theoretically, the 3D FFT solver has a complexity of $O(n\log(n))$ versus $0(n^{5/3})$ for SOR. However, SOR is an iterative refinement method whose convergence speed largely depends on the initial guess. With a close approximation of the result as the initial value, it normally requires only a few iterations to converge. The same argument is true for other iterative methods such as CG.

We observe that in the elastic deformable framework with steady fluids, the changes in the velocity field are quite small between greedy steps. The computed velocity field of the previous step is inherently a good approximation for the current one. In practice, we typically need 50 to 100 SOR iterations for the first greedy step, but only 4 to 6 iterations for each following step.

Our framework provides an SOR implementation with Red-Black ordering as shown in Figure 2.8. This strategy allows for efficient parallelism as we only update points of the same color based on their neighbors, which have different color. Also, Red-Black decoupling has proved to have a well-behaved convergence rate with the optimal over-relaxation factor ω defined in 3D as $\omega = \frac{2}{1+\sqrt{1-\frac{1}{3}\left[\cos\frac{\pi}{w}+cos\frac{\pi}{h}+cos\frac{\pi}{l}\right]^2}}$

We incorporate optimization techniques from the 3D stencil buffer problem to exploit the fast shared memory available in CUDA and improve the register utilization of the al-

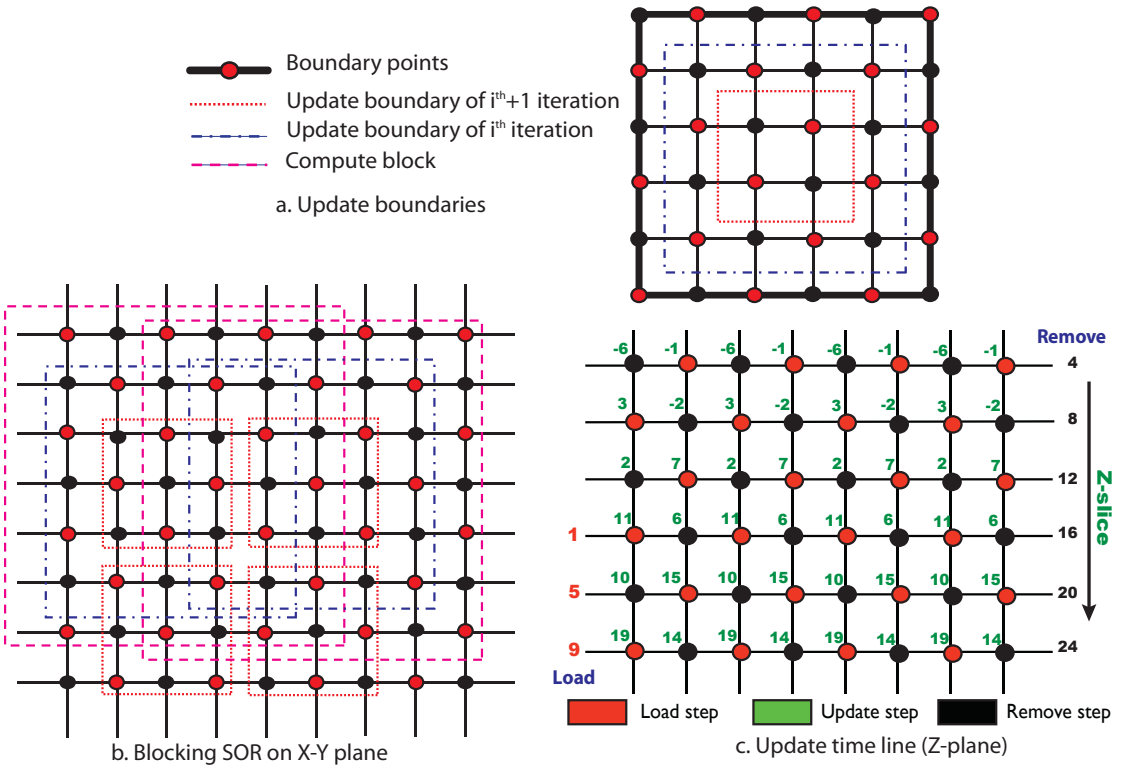


Figure 2.8. Parallel block SOR, we assign each CUDA thread warp a block of data to compute the black points inside the blue boundary, and use that result to compute the red point inside the red boundary. Two neighboring compute blocks share a four grid point-wide region.

gorithm (Algorithm 3). We further improve the performance by increasing the arithmetic intensity of the data. This is done by merging steps of SOR that combine red-updates and black-updates of traditional SOR into one execution kernel. We also proposed a *block-SOR* algorithm in which we divide the input volume into blocks, each fitting onto one CUDA execution block. We then exploit the shared memory to perform the merging step locally on the block. For simplicity, we illustrate the idea in 2D in Figure 2.8, but it is generalized to arbitrary dimensions.

As shown on Figure 2.8 the updated volume is two cells smaller in each dimension than the input. This reduction in size explains why we can not merge an arbitrary number steps in one kernel. To update the whole volume, we allow data overlaps among processing blocks (Figure 2.8(b)). Here, we allow data redundancy to increase memory usage. The configuration, having a 4-point-wide boundary overlap, is able to update one

Algorithm 3 Efficient CUDA PDE block-SOR solver

1: **Input** : Old velocity field v and new force function F
2: **Output**: Compute new velocity field v
3: Allocate 4 shared mem arrays s_{prev},s_{cur}, s_{next}, s_{next2} to store 4 slices of data
4: Load F of 3-first slices to the registers of current thread
5: Load v of 3-first slices to registers and shared memory
6: The boundary thread loads the padding data of v
7: Update the black point of the second slice s_{cur}
8: **for** $k = 1$ to $Z - 2$ **do** {loop over Z direction}
9: Load the F and v of the next slice to the free shared-mem array s_{next2}
10: Update the black points of the s_{next} slice
11: Update the red points of the s_{cur} slice
12: Write s_{prev} to the global output, s_{prev} buffer is free to load the next slice
13: Shift the value of v and F in the registers, $cur \rightarrow prev$, $next \rightarrow cur$, $next2 \rightarrow next$
14: Circular shift the shared memory array pointers, s_{prev} becomes s_{next2}
15: **end for**
16: Update the red points on the last slice close to boundary
17: Write out the last slice

Red-Black merging step over a M^2 block using $(M + 4)^2$ inputs. Likewise, a k-merging step needs a data block of size $(M + 4 * k)^2$. To quantify the benefit of SOR merging steps, we compute a trade-off factor α such that:

$$\alpha = \frac{\text{Minimum needed data size}}{\text{Actual processing data size}} * \text{Speed\_up\_factor} \tag{2.3}$$

In 3D, to update the volume block M^3, we need $(M+4k)^3$ volume inputs, the trade-off factor is $\alpha = (\frac{M+1}{M+4k})^3 * k$. Note that the size of shared memory constrains the block size M and merging level k. In practice, we see benefits only if we merge a single black & red update step per kernel call.

Algorithm 3 shows the pseudocode of our block-SOR implementation on CUDA. We further leverage the trade-off by limiting block-SOR in the 2D plane only, and exploit the coherence between consecutive layers in the third dimension to minimize data redundancy. On the Tesla, our block-SOR implementation using shared-memory is twice as fast as than equivalent version using a 1D texture cache. Figure 2.8(c) shows the updating time line in Z-dimension, in which it is clear that each node is computed by its neighbors which are updated in previous steps.

2.3.6 Conjugate Gradient method

While the SOR method is specialized for solving PDEs on a regular grid, in practice the Conjugate Gradient (CG) approach is often the preferred technique because of several advantages:

- It is capable of solving a PDE on an irregular grid as well.
- It is simple to implement as it built on top of basic linear operations.
- In general, it converges faster than the SOR method.

As shown in Figure 2.9, the CG algorithm is implemented in our framework as a template class with T being the matrix presentation of the system. The only function required from T is a matrix vector multiplication. The template allows for the integration of any sparse matrix vector multiplication package using an explicit presentation such as ELL, ELL/COO [11] and CRS [8], or an implicit representation which encodes the system matrix with constant values in the kernel. Figure 2.10 shows the implementation of the implicit matrix vector multiplication with a Helmholtz matrix and zero-boundary conditions.

The texture cache is used to access neighboring information to achieve maximal memory bandwidth. Our experiments showed that in the case of regular grid, the implicit

```
template<class T>
void CG_impl(float* d_b, T& d_A, float* d_x, int imax,
             float* d_r, float* d_d, float* d_q)
{
  int n = d_A.getNumElements();
  computeResidual(d_r, d_b, d_A, d_x);// r = b - Ax
  copyArrayDeviceToDevice(d_d, d_r, n); // d = r
  float delta_new = cplvSum2(d_r, n);  // delta_new = r^Tr
  float delta0 = delta_new, delta_old, eps=1e-4, alpha, beta;
  for (i=0; (i < imax) && (delta_new > eps * delta0); ++i)
    maxtrixMulVector(d_q, d_A, d_d);     // q = Ad
    alpha = delta_new/cplvDot(d_d, d_q, n);// alpha = delta_new/d^Tq
    cplvAdd_MulC_I(d_x, d_d, alpha, n) ;// x = x + alpha * d
    cplvAdd_MulC_I(d_r, d_q, -alpha, n);// r = r - alpha * q
    delta_old = delta_new;
    delta_new = cplvSum2(d_r, n);        // delta_new = r^Tr
    beta = delta_new / delta_old;        // beta = delta_new / delta_old
    cplvMulCAdd_I(d_d, beta, d_r, n);  // d = beta * d + r
  }
}
```

Figure 2.9. CG Solver template

```
__global__ void helmholtz3D_MV(float* b, float* x,
    float alpha, float gamma, int sizeX, int sizeY, int sizeZ)
{
    uint xid      = threadIdx.x + blockIdx.x * blockDim.x;
    uint yid      = threadIdx.y + blockIdx.y * blockDim.y;
    uint id       = xid + yid * sizeX, planeSize= sizeX * sizeY;
    if (xid < sizeX && yid < sizeY){
        float zo = 0, zc = fetch(id, x), zn;
        for (uint zid=0; zid<sizeZ; ++zid, id += planeSize){
            zn = (zid + 1 < sizeZ) ? fetch(id + planeSize, x) : 0;
            float r  = zo + zn;
            r += (xid   > 0)      ? fetch(id - 1, x)      : 0;
            r += (xid + 1 < sizeX)? fetch(id + 1, x)      : 0;
            r += (yid   > 0)      ? fetch(id - sizeX, x) : 0;
            r += (yid + 1 < sizeY)? fetch(id + sizeX, x) : 0;
            b[id] = zc * (6 * alpha + gamma) - alpha  * r;
            zo = zc; zc  = zn; // shift values on Z-dir
        }
    }
}
```

Figure 2.10. Matrix vector multiplication CUDA kernel with implicit Helmholtz Matrix

approach allows for a more efficient matrix vector multiplication as the matrix does not consume memory bandwidth. As shown on Table 2.3, implicit method is up to 2.5 times faster than explicit implementations [11]. The performance is measured in GFLOPs with Helmholtz Matrix vector multiplication.

2.3.7 Multiscale framework

The concept of our multiscale framework is derived from the multigrid technique, which computes an approximate solution on a coarse grid and then interpolates the result onto the finer grid. As the solution on the coarse grid generates a good initial guess of solution on the finer grid, it speeds up the convergence on the finer level. In

Table 2.3. Performance comparison, in GFLOPs, between our implicit method and explicit implementations (larger is faster)

Matrix size	64^3	96^3	128^3	160^3	192^3	224^3	256^3
Implicit	17	37	53	42	54	51	59
Explicit Dia	25	27	27	25	25	25	27
Explicit Ell	16	17	17	16	16	16	16

addition to reducing the number of iterations, multiresolution increases the robustness with respect to noise in the input data, as it is capable of handling local optimums inherent to gradient-descent optimization. We design a multiscale GPU interface (Algorithms 4) based on two main components: a downscale Gaussian filter and an up-sampling sampler.

The downscaled filter is composed of a low-pass filter followed by a down sampler. The low-pass filter is a 3D-Gaussian filter which is implemented using separable 1D-Gaussian filters along each axis. We discovered that it is more efficient to implement this 3D filter based on a separable, recursive Gaussian filter rather than convolution based or FFT-based approaches. Our recursive version is generalized from the 1D recursive version (see NVIDIA SDK *RecursiveGaussian*) with a circular-dimension shifting 3D transpose. As shown on Table 2.4, the 3D recursive version is the fastest, and its runtime, measured in milliseconds, is independent of the kernel size. The other methods in comparison are: a separable filter, a circular dimension shifting combined with the 1D filter in the fastest dimension, and a FFT-based filter.

While the down sampler simply fetches values from the grid, the up sampler is the

Algorithm 4 multiscale atlas construction

Input : N volume inputs, multiscale information
Output: Template atlas volume
for all $s = 1$ to N_s **do** {loop over the scales}
  Read $factor_s$, $nIters_s$, fluid registration parameters at the scale
  for $i = 1$ to N **do** {loop over the images}
    if $factor_s = 1$ **then** {first level scale - original image}
      $I_{is} \leftarrow I_i$
    else {down sample the image}
      Blur the image $I_i(blur) = GaussFilter(I_i)$
      Down sample $I_{is} = DownSample(I_i(blur))$
    end if
    if $s = 1$ **then** {first iteration}
      $h_{is} \leftarrow Id$, $v_{is} \leftarrow 0$
      Copy the sample image $I^0_{is} = I_{is}$
    else
      Up sample deformation field $h_{is}(x) = UpSample(h_{is}(x))$
      Up sample velocity field $v_{is}(x) = UpSample(v_{is}(x))$ {if needed}
      Update deformed image $I^0_{is} = I_{is}(h_{is}(x)))$
    end if
  end for
  Apply the atlas construction procedure at this scale
end for

Table 2.4. Performance comparison, in milliseconds, between different optimization strategies to implement 3D-Gaussian Filter with different kernel sizes

Half kernel size	Separable	Dim-shift	Recursive	FFT
2	14	17	10	85
4	26	28	10	85
8	49	47	10	80

reverse mapping operation from the grid point of the finer scale to the point value of the coarser grid based on the trilinear interpolation. Here we used the same optimization as for the ODE integration. Our multiresolution framework can be employed in any 3D image processing problem to improve both performance and robustness.

2.3.8 Multi-GPU processing model

Computing systems in practice have to deal with large amounts of data which cannot be processed directly and efficiently by a single processing system. GPUs are no exception to this limitation. Hence, the development of a parallel multi-GPU framework is necessary, especially for exploiting the total power of multi-GPU workstation or GPU clusters.

In the following, we address the two main bottlenecks of multi-GPUs and cluster implementations: the limited CPU-GPU bandwidth, which is about 20 times slower than the local GPU memory bandwidth, and the limited network bandwidth between compute nodes which is an order of magnitude slower than CPU-GPU bandwidth. Our computational model aims at minimizing the amount of data transfer over the slow media, exploiting existing APIs such as MPI, and moving most of the computation from the CPUs to the massively parallel GPUs.

2.3.8.1 Single node multi-GPU model

We first proposed a multiple-input multi-GPU model [56] on a single node. The key idea was to maximize the total volume of inputs that the system can handle. In other words, by maximizing the number of inputs per node we increase the *arithmetic intensity* [21] of each processing node.

We divide the inputs between GPU nodes and assign a GPU memory buffer at each node to serve both as an accumulation buffer and an average input buffer. As an output

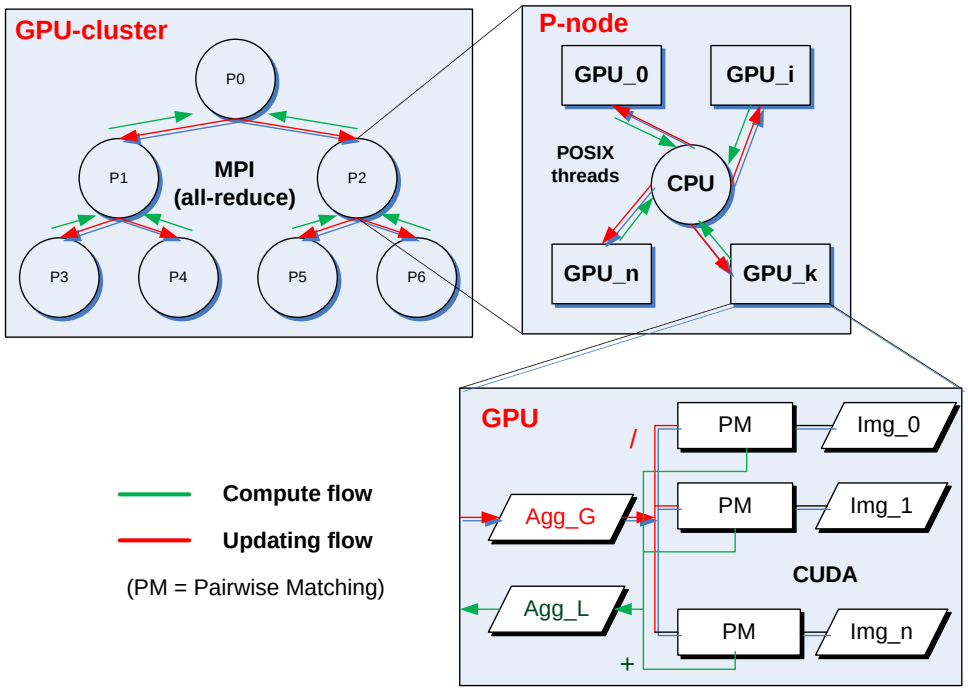


Figure 2.11. Multi-GPUs framework on the GPU cluster. We combine the processing models using a hierarchical strategy, from a single-GPU model to a single node multi--GPUs model using PThreads, and finally to a GPU cluster with MPI communication between processing nodes. The distribution of compute flow and the data updating process happens in the opposite direction of the hierarchy.

buffer, it is used to sum up the local deformed volumes while as an input buffer, it contains the new average and is shared among volumes of the node.

At each iteration, we compute the local accumulation buffers, and send the result to the server to compute the global accumulation buffer. The new aggregate volume is read back to GPU nodes. Next, we perform a volume division on the GPUs to update the average.

Our aggregate model is more efficient than our previous average model [56], since it yields the same memory bandwidth but moves the computation from CPUs to GPUs, hence it is able to exploit the computational power of the GPU. This strategy minimizes both the overall cost per volume element as well as the data transfer over the low bandwidth channel.

2.3.8.2 GPU cluster model

We generalize the multiple input multi-GPU model to a higher level to build a computationally efficient framework on GPU clusters. As displayed in Figure 2.11, we maintain two buffers on a CPU-multi-GPU processing node: an output accumulation buffer and an input aggregate buffer which is shared among its GPUs' members. These two CPU memory buffers are used as the interface memory to other processing nodes communicated via MPIs. As we used the aggregated model instead of the average, we can directly exploit the MPI all-reduce function to efficiently compute and update the accumulated volume to all processing nodes. Next, we address the load distribution problem of our GPU cluster implementation.

2.3.8.3 Load balancing

We consider load balancing on a system with homogeneous GPUs with N_i, N_g, N_p being the number of inputs, GPUs, and CPUs. Our test system is a Tesla S1070 cluster and each node has dual-GPUs, thereby implying that $N_g = 2 * N_p$.

On the cluster, the total run-time per iteration is computed by $T = T_{GPU} + T_{CPU} + T_{Network}$. As the number of GPUs per node is fixed, T_{CPU}- the amount of time to compute the aggregate among GPUs of the same node - is fixed. Consequently, we must reduce T_{GPU} and/or $T_{Network}$ to improve the run-time.

First, we assume that N_p is fixed and then $T_{Network}$-the amount of time to accumulate and distribute result between CPUs-is defined. T_{GPU} depends on the maximum number of inputs per GPU, which is at least $N_{ig} = \lceil \frac{N_i}{N_g} \rceil$. This number is optimal if inputs are distributed evenly between GPUs, not CPUs as CPUs may have a variable number of GPUs attached. So our first strategy is distributing inputs evenly among GPU nodes. With this strategy, there is at most one unbalanced GPU, and the GPU run-time with synchronization is optimal.

Second, it is highly likely that the MPI all-reduce function performs a binary tree down-sweep to accumulate the volume and binary-tree up-sweep to distribute the sum to all nodes, as shown in Figure 2.11. This yields a minimal amount of data transferred over the network, that is $2 * N_p$. It is suggested that the amount of data transfer over the network increases linearly with the number of CPU nodes and therefore fewer CPUs implies smaller delay. This hypothesis is confirmed in our cluster in the experiment (see Figure 2.12)

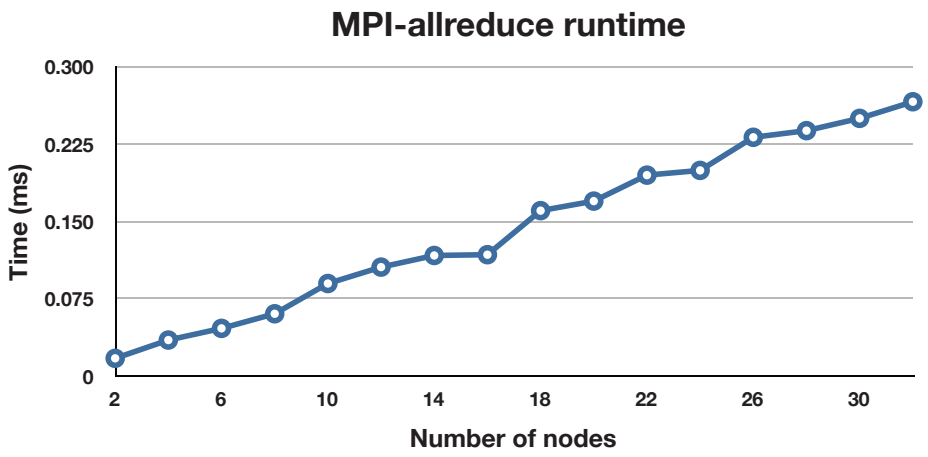


Figure 2.12. MPI-All reduce runtime on an infiniband network with OpenMPI 1.3 shows a linear dependency on the number of nodes

To reduce the number of CPU nodes, we increase the GPU workload. Note that from the first strategy we want to increase all GPUs with the same number of volumes so that the computation is balanced. Let us increase this number by one, the total run time is then

$$T = T_{GPU} * \frac{N_{ig} + 1}{N_{ig}} + T_{CPU} + T_{Network} * \frac{N_p - N_{ps}}{N_p},$$

where N_{ps} is the number of GPUs reduced by increasing the workload. This equation gives us an approximation of running time as the number of volumes per GPU changes. Hence, we can vary the capability on the GPU node to achieve a better configuration. Note that in the dual-GPUs system, if the number of volumes per GPU is less than N_g, when we increase the number of volumes per GPU by one, we can decrease the number of CPUs at least by 2.

Our load balancing strategy is as follows: first, the users choose the number of nodes. Based on this the system computes the number of inputs per GPU. The components' runtime is then determined with one-iteration dry run on the zero-initialized volumes which require no data from the host. Next, the optimizer varies the number of volumes per GPUs, recomputes the number of CPU nodes, and computes the total runtime. This heuristic yields an optimal configuration to handle the problem.

2.3.9 Other performance optimization

To further improve the performance, we now present a volume space clipping optimization and the scratch memory model. These techniques are specially applied for multi-image processing problems.

2.3.9.1 Volume clipping optimization

Volume clipping is the final step of preprocessing, which includes

- Rigid alignment and affine registration
- Intensity calibration and normalization
- Volume clipping

The rigid alignment and affine registration guarantee all inputs to be in the same space while the preprocessing distances between them are minimal. This strategy significantly speeds up the convergence of the image registration process. The intensity calibration ensures that the intensity range of the inputs are matched and are normalized for visualization. While these two preprocessing steps are generally applied in a regular image registration framework, the volume clipping is a special optimization scheme applied for the brain image to reduce processing time.

Point-wise computations on zero-data result in zero; we call these data redundant. This redundancy happens near the boundary of the volume. The volume clipping strategy first computes the nonzero data bounding boxes, and then tightly clips all the volumes to the common bounding box with guarded boundary conditions. In practice, the volume of clipping inputs can be significantly smaller than a typical input volume, for example the 256^3 brain images in our experiment have a common volume of size $160 \times 192 \times 160$, a volume ratio of three. As the runtime of a function is proportional to the volume of the inputs, we experienced three to four times speed up just by applying this volume clipping strategy. Note that this optimization is more effective at PDE SOR solvers than FFT-based solvers as the latter require a power of two volume size to be computationally efficient.

2.3.9.2 Scratch memory model

It is always a challenge to implement 3D processing frameworks on GPUs as the parallel processing scheme often requires more memory than it would on CPUs. To deal with this memory problem, we proposed a scratch memory model, a shared-temporary memory space, coupled with different optimization techniques including:

- Zero-copy operation based on pointer swapping to reduce the redundant memory copy from scratch memory (Figure 2.13a), and
- A circular buffer technique to reduce memory copy redundancy and also memory storage requirements for computation in a loop (Figure 2.13b)

The use of the scratch memory model helps us to significantly reduce the memory requirement. In particular, in the case of greedy iterative atlas construction, we only need a single image buffer and two 3D vector buffers for an arbitrary number of inputs on a single GPU device. Consequently, we are able to process 20 brain volumes with 4GB global memory, or 40 brain volumes on a single dual-GPU node.

2.4 Evaluation and validation of results

The system we used in our experiment is a 64-node Tesla S1070 cluster, each containing two GPUs. Communication from the host to GPU is via the external x16 PCIe bus, and internode communication is implemented through a 20 GBits 4x DDR infiniband interconnect. The program was compiled with CUDA NVCC 2.3. For multiresolution,

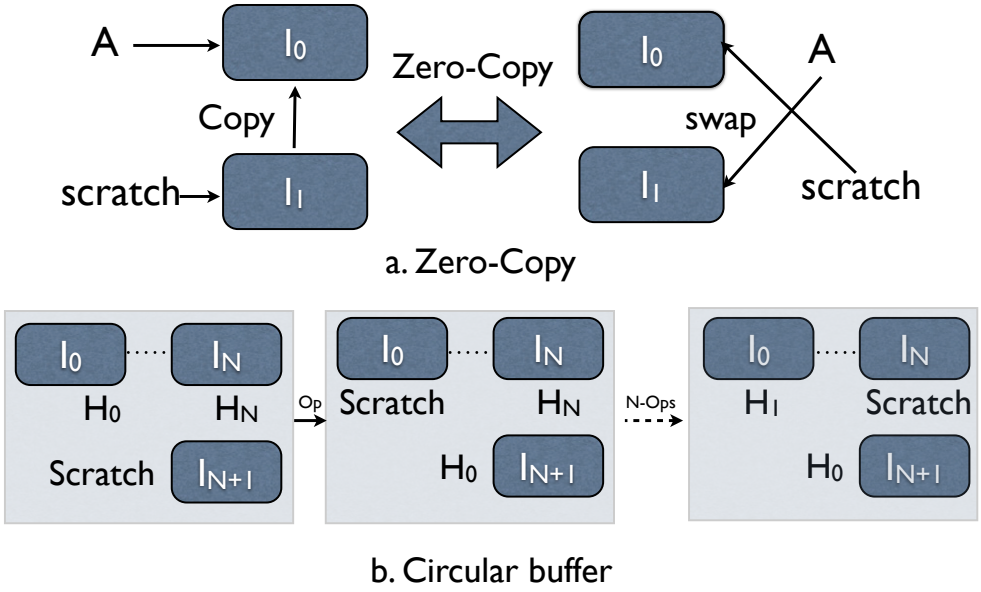


Figure 2.13. Optimization strategies with the scratch memory model

we perform a 2-scale computation with 25 iterations at the coarse level, 50 iterations at the finer level, parameter $\alpha = 0.01$, $\gamma = 0.001$, and *maximum step size = 1.* The three solvers used in the comparison are FFT solver, the block-SOR, and Conjugate Gradient(CG). The runtime does not include the data loading time that depends on the hard disk system.

2.4.1 Quality improvements

To evaluate the robustness and stability of the atlases, we use the random permutation test proposed by Lorenzen *et al.* [76]. The method is capable of estimating the minimum number of inputs required to construct a stable atlas by analyzing mean entropy and the variance of the average template. We generated 13 atlas cohorts, $C_{l,l=2\cdots14}$, each including 100 atlases constructed from l input images chosen randomly from the original data set. The 2D midaxial slices of the atlases are shown in Figure 2.14. The normal average atlases are blurry, and ghosting is evident around the lateral ventricles, and near the boundary of the brain. In this case, the Greedy Iterative Average template appears to be much sharper, preserving anatomical structures.

The quality of the atlas construction is visibly better than the least MSE normal average. The entropy results shown in Figure 2.15 also confirm the stability of our

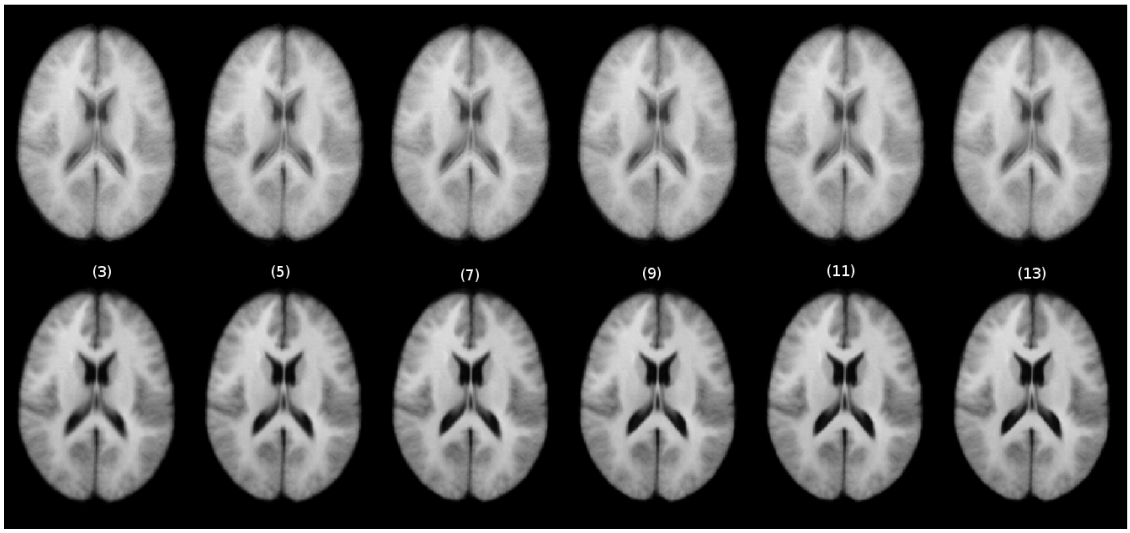


Figure 2.14. Atlas results with 3, 5, 7, 9, 11 and 13 inputs constructed by (a) arithmetically averaging rigidly aligned images (top row) and (b) Greedy Iterative Average template construction (bottom row)

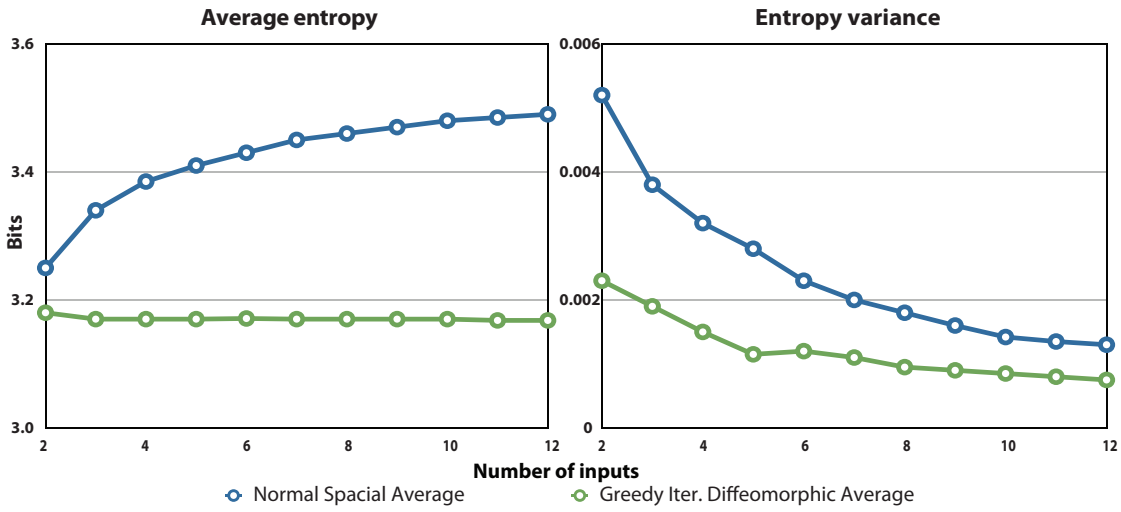


Figure 2.15. Mean entropy and variance of atlases constructed by arithmetically averaging and the Greedy Iterative Average template

implementation. As the number of inputs increases, the average atlas entropy of the simple averaging intensity increases while the Greedy Iterative Average template decreases due to much higher individual sharpness. This quantitatively asserts the visible quality improvement in Figure 2.14. The atlases become more stable with respect to the entropy as the standard deviation decreases with an increasing number of inputs. After cohort C_8, the atlas entropy mean appears to converge. So we need at least 8 images to create a stable atlas representing neuroanatomy.

2.4.2 Performance improvement

We compare the speedup of the multiscale framework to the single scale version with a pairwise matching problem to produce comparable results. Experiments show that we generally need 25 iterations in the second level and 50 iterations in the first level to produce similar results whereas we need 200 iterations with a single scale implementation. The speed up factor is about 3.5 and comes primarily from lowering the number of iterations in the finest level.

We quantify the compute capability and scalability of our system in two cases. First by applying the scratch memory technique, we are able to handle 20 T1 image of size $160 \times 224 \times 160$ on a single GPU device. We measure the performance with one GPU device (multiscale), one single node with dual-GPUs (multiscale multi-GPUs), two, four and five

GPU nodes (multiscale cluster) in reference to a single scale version on the 20-brain input set. As shown in Figure 2.16 the multiscale version is about 3.5 times faster than the single scale version, while our multi-GPU version is twice as fast as a single device. The cluster version shows a linear performance improvement to the number of nodes.

Second, we experiment with the full data set of 315 volumes of T1 input size $144 \times 192 \times 160$. For the first time we handle the whole data set on 8 nodes of the GPU cluster. We measure the performance with 8, 12, 16, 20, 24, 28 and 32 nodes. On the 32-core Intel Xeon server X7350, 2.93 Ghz with 196 GB shared memory, which is able to load the whole data set in-core, the CPU-optimized greedy implementation took 2 minutes for a single iteration. As shown on Figure 2.17, it only takes the SOR solver about 70 seconds to compute the average on 8 nodes of the GPU cluster and only 20 seconds on 32 nodes which is two orders of magnitude faster than the 32-core CPU server.

2.5 Conclusion

In this chapter we have presented our implementation of the Unbiased Greedy Iterative Atlas construction on multi-GPUs; however, this is only a showcase to illustrate the computing power and efficiency of our processing framework. As we mention in the

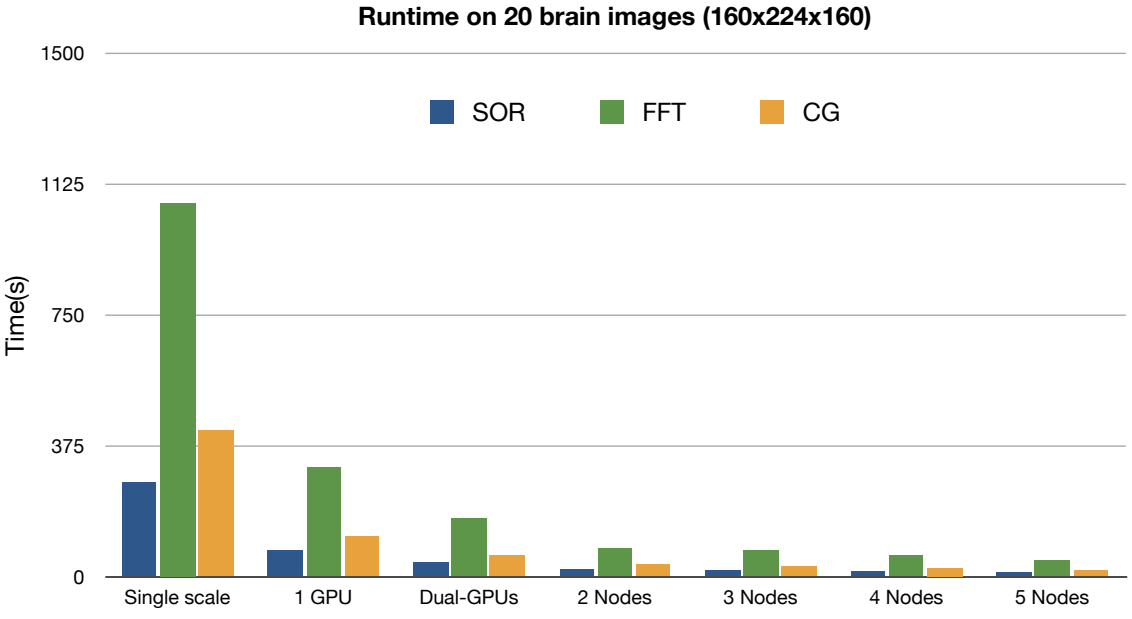


Figure 2.16. Runtime to compute the average atlas of the 20 T1 brain images ($144 \times 192 \times 160$) with multiscale and/or multi-GPUs, cluster implementation in reference to one scale version

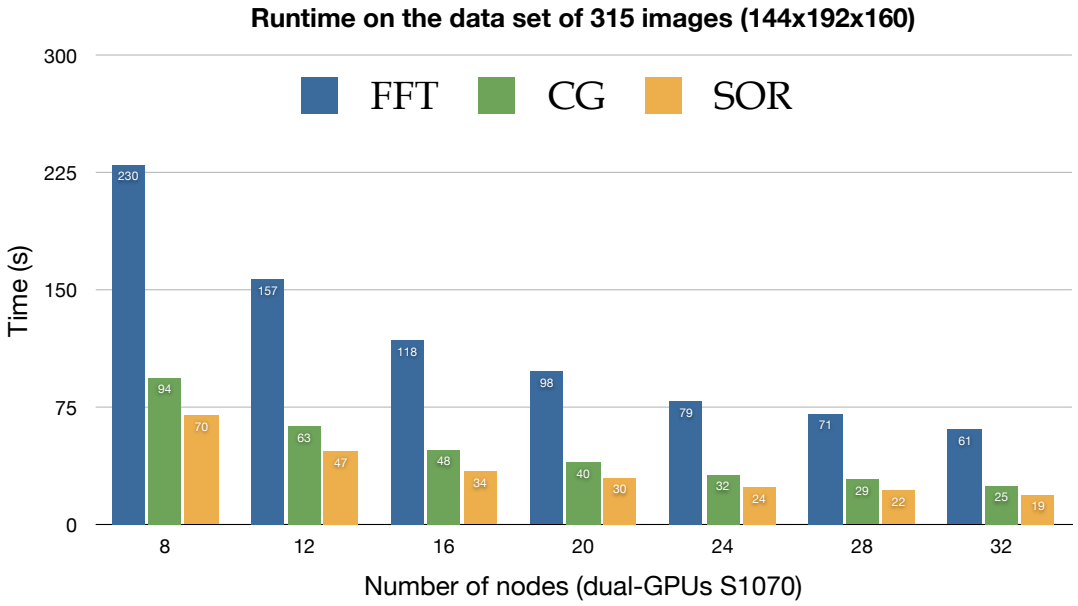


Figure 2.17. Multiscale runtime to compute the average atlas of the 315 T1 brain images ($144 \times 192 \times 160$) with different PDE solver

introduction, the atlas construction problem is a basic foundation for a class of diffeomorphic registration problems to study the intrapopulation variability and interpopulation differences. The ability to produce the result in real-time give us the ability to understand the research influence of this powerful technique. Also the framework allows us to implement more sophisticated registration problem such as LDDMM, Metamorphosis, or Image Current to name just a few. While each technique has a different trade-offs between quality of results and the computation involved, our framework is capable of quantifying those trade-offs to suggest a good solution for the practical problem suitable with inputs and the accessible computational power.

Though the system has the capability to handle large amounts of data, it requires a single matching pair to be completely solvable on single GPU node. However, such compute power is not always available. So we consider extending the processing power of single GPU system using an out-of-core technique in Chapter 4. This requires a major redesign of our system; however, it is a required feature of our processing system to handle the ever growing size of data.

CHAPTER 3

COMBINING PROBABILISTIC AND GEOMETRIC DESCRIPTOR

Deformable image registration in the presence of considerable contrast differences and large size and shape changes present significant research challenges. First, it requires a robust registration framework that does not depend on intensity measurements and can handle large nonlinear shape variations. Second, it involves the expensive computation of nonlinear deformations with high degrees of freedom. Often it takes a significant amount of computation time and thus become infeasible for practical purposes. In this chapter, we present a solution based on two key ideas: a new registration method that generates a mapping between anatomies represented as a multicompartment model of class posterior images and geometries, and an implementation of the algorithm using Particle Mesh approximation on Graphical Processing Units (GPUs) to fulfill the computational requirements. We show results on the registrations of neonatal brain MRIs to 2-year-old infant MRIs. Quantitative validation demonstrates that our proposed method generates registrations that better maintain the consistency of anatomical structures over time and provides transformations that better preserve structures undergoing large deformations than transformations obtained by standard intensity-only registration. We also achieve the speed up of three orders of magnitude compared to a CPU reference implementation, making it possible to use the technique in time-critical applications.

3.1 Introduction

Our work is motivated by the longitudinal study of early brain development in neuroimaging, which is essential to predict the neurological disorders in early stages. The study, however, is challenging for two primary reasons: the large scale - nonlinear shape changes (the image processing challenge) and the huge amount of computational power the problem requires (the computational challenge). The image processing challenge involves robust image registration to define anatomical mappings. While robust image

registrations have been studied extensively in the literature [44, 84, 98], registration of the brain at early development stage is still challenging as the growth process can involve very large-scale size and shape changes, as well as changes in tissue properties and appearance. Knickmeyer *et al.* [70] showed that the brain volume grows by 100% the first year and 15% the second year, whereas the cerebellum shows 220% volume growth for the first and another 15% for the second year (Figure 3.1). These numbers indicate very different growth rates of different anatomical structures. Through regression on shape representations, Datar *et al.* [32] illustrated that the rapid volume changes are also paralleled by significant shape changes, which describe the dynamic pattern of localized, nonlinear growth. A major clinical research question is to find a link between cognitive development and the rapid, locally-varying growth of specific anatomical structures. This requires registration methods to handle large-scale and also nonlinear changes. Also, the process of white matter myelination, which manifests as two distinct white matter appearance patterns primarily during the first year of development, imposes another significant challenge as image intensities need to be interpreted differently at different stages.

To approach these problems, a robust registration method is necessary for mapping longitudinal brain MRI to a reference space so that we can perform reliable analysis of the tissue property changes reflected in MR measurements. This method should not rely

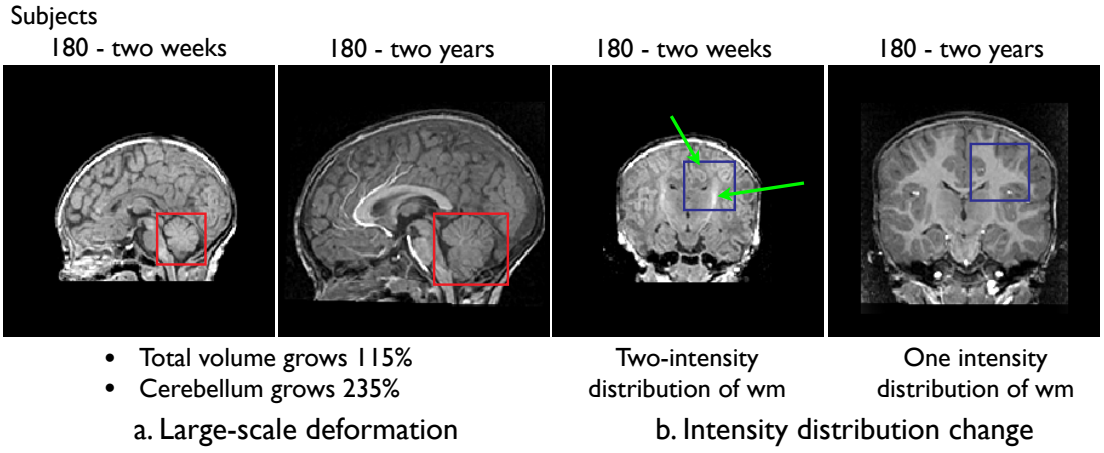


Figure 3.1. Registration challenges of human brains at early development stages. The image shows significant shape and size changes of an infant brain of subject 180 from two weeks to two years as well as the changing white matter properties and appearance due to the myelination.

on raw intensity measurements, while it should be capable of estimating large structural deformations. Xue *et al.* [127] addressed these issues by proposing a registration scheme for neonatal brains by registering inflated cortical surfaces extracted from the MRI.

In this chapter, we propose a new registration framework for longitudinal brain MRI that makes use of underlying anatomies, which are represented by class posteriors and geometries. This framework can match internal regions and simultaneously preserve a consistent mapping for the boundaries of relevant anatomical objects. We show results of registering neonatal brain MRI to 2-year-old brain MRI of the same subjects obtained in a longitudinal neuroimaging study. Our method consistently provides transformations that better preserve time-varying structures than those obtained by intensity-only registration [105].

3.2 Method overview

We propose a new registration method that makes use of the underlying anatomy in the MR images. Figure 3.2 shows an overview of the registration process. We begin by extracting probabilistic and geometric anatomical descriptors from the images, followed by computing a transformation that minimizes the distance between the anatomical descriptors.

3.2.1 Anatomical descriptors

We represent brain anatomy as a multicompartment model of tissue class posteriors and manifolds. We associate each position x with a vector of tissue probability densities. In a given anatomy, we capture the underlying structures by estimating, for each image, the class posterior mass functions associated with each of the classes. Given Ω as the underlying coordinate system of the brain anatomies, each anatomy $\mathcal{A}_{i=1,\cdots,N}$ is represented as

$$\mathcal{A}_i = \{p_{i,c=1}(x), \cdots, p_{i,c=N_c}(x), \mathcal{M}_{i,j=1}(2), \cdots, \mathcal{M}_{i,j=N_s}(2) \subset \Omega\}, \tag{3.1}$$

where N_c is the number of probability images, N_s is the number of surfaces, $p_c(x)$ is the class posterior for tissue c at location x, and $\mathcal{M}_j(2)$ are 2-dimensional submanifolds of Ω (surfaces).

The classification of brain MR images with mature white matter structures into class posteriors are well studied. We extract the posteriors from 2-year-old brain MR images using the segmentation method proposed by van Leemput *et al.* [122]. The

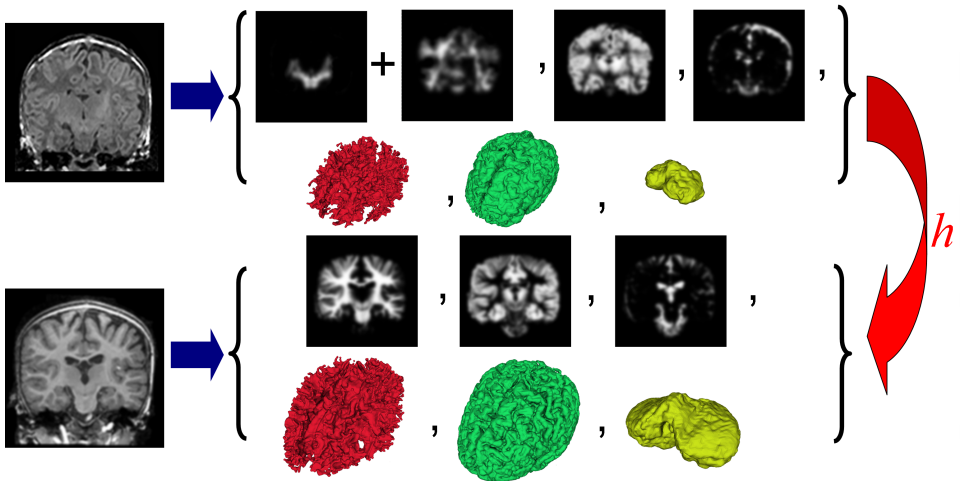


Figure 3.2. Overview of the proposed registration method that can handle large deformations and different contrast properties, applied to mapping brain MRI of neonates to 2-year-olds. We segment the brain MRIs and then extract equivalent anatomical descriptors by merging the two different white matter types present in neonates. The probabilistic and geometric anatomical descriptors are then used to compute the transformation h that minimizes the distance between the class posterior images, as well as the distance between surfaces represented as currents.

method generates posterior probabilities for white matter (wm), gray matter (gm), and cerebrospinal fluid (csf). These probabilities can then be used to generate surfaces from the maximum a posteriori tissue label maps.

The classification of neonatal brain MR images is challenging as the white matter structure undergoes myelination, where the fibers are being covered in myelin sheaths. Several researchers have proposed methods that make use of prior information from an atlas or template that takes into account the special white matter appearance due to myelination [124]. We use the method described by Prastawa *et al.* [102] for extracting the tissue class posteriors of neonatal brain MRI, which includes for myelinated wm, nonmyelinated wm, gm, and csf. These can then be used to create an equivalent anatomy to the 2-year-old brain by combining the two white matter class probabilities which then leads to a single white matter surface.

For the results in this chapter, we compute the probabilities $\{p_{wm}(x), p_{gm}(x), p_{csf}(x)\}$ and we use the surfaces of white matter, gray matter, and cerebellum. The cerebellum surfaces are generated from semiautomated segmentations that are obtained by affinely registering a template image followed by a supervised level set segmentation. The cerebellum has a significant role in motor function and it is explicitly modeled as it undergoes

the most rapid volume change during the first year of development and thus presents a localized large-scale deformation.

3.2.2 Registration formulation

Given two anatomies $\mathcal{A}_1$ and $\mathcal{A}_2$, the registration problem can be formulated as an estimation problem for the transformation h that minimizes

$$\hat{h} = \arg\min_{h} E(h \cdot \mathcal{A}_1, \mathcal{A}_2)^2 + D(h, e)^2, \tag{3.2}$$

where $h \cdot \mathcal{A}_1$ is the transformed anatomy, $E(\cdot, \cdot)$ is a metric between anatomies and $D(\cdot, e)$ is a metric on a group of transformations that penalizes deviations from the identity transformation e. The anatomy is transformed using backward mapping for probability image and forward mapping for geometries:

$$\begin{aligned} h \cdot \mathcal{A}_1 &= h \cdot \{p_{i,c=1}(x), \cdots, p_{i,c=N_c}(x), \mathcal{M}_{i,j=1}(2), \cdots, \mathcal{M}_{i,j=N_s}(2)\} \\ &= \{p_{i,c=1}(x) \circ h^{-1}, \cdots, p_{i,c=N_c}(x) \circ h^{-1}, h(\mathcal{M}_{i,j=1}(2)), \cdots, h(\mathcal{M}_{i,j=N_s}(2))\} \end{aligned} \tag{3.3}$$

We define distance between anatomies E by defining a norm on an anatomy as a combination of the L^2 norm on the class posteriors and a Reproducing Kernel Hilbert space norm on the manifolds defined as "currents" through Glaunes [45]. The currents norm does not require geometric correspondence and thus can be used to register manifolds with different resolutions. For an oriented surface $\mathcal{M}(2)$ in R^3 the norm $[\mathcal{M}(2)]$ is the vector valued Borel measure corresponding to the collection of unit-normal vectors to $\mathcal{M}(2)$, distributed with density equal to the element of surface area ds and can be written as $\eta(x)ds(x)$, where $\eta(x)$ is the unit normal and $ds(x)$ is the surface measure at point x.

Given an anatomy $\mathcal{A}$ the k-norm of $[\mathcal{A}]$ is composed as

$$\left\| [\mathcal{A}] \right\|_k^2 = \|P(x)\|_{L^2} + \|[\mathcal{M}(2)]\|_k \,, \tag{3.4}$$

where the probabilistic norm is defined as

$$\begin{aligned} \|P(x)\|_{L^2} &= \sum_{c=1}^{N_c} \|p_{1,c}(x) - p_{2,c}(x)\|_k^{L^2} \\ &= \int_\Omega |p_{1,c}(x) - p_{2,c}(x)|^2 dx, \end{aligned} \tag{3.5}$$

and the currents norm is given by

$$\|[\mathcal{M}(2)]\|_k = \int_{\mathcal{M}(2)} \int_{\mathcal{M}(2)} k(x, y) \, \langle \eta(x), \eta(y) \rangle \, d\mu(x) d\mu(y), \tag{3.6}$$

where $k(\cdot,\cdot)$ is a shift-invariant kernel (e.g., Gaussian or Cauchy).

When $\mathcal{M}(2)$ is a discrete triangular mesh with N_f faces, a good approximation of the norm can be computed by replacing $[\mathcal{M}(2)]$ by a sum of vector-valued Dirac masses

$$\|[\mathcal{M}(2)]\|_k^2 = \sum_{f=1}^{N_f}\sum_{f'=1}^{N_f} \left\langle \eta(f), \eta(f') \right\rangle k(c(f), c(f')), \tag{3.7}$$

where N_f is the number of faces of the triangulation, and for any face f, $c(f)$ is its center and $\eta(f)$ its normal vector with the length capturing the area of each triangle.

Having defined the norm on probability images and surfaces, the dissimilarity metric between anatomies $\left\|[\mathcal{A}_1] - [\mathcal{A}_2]\right\|_k^2$ is given by:

$$\begin{aligned} & w_p \sum_{c=1}^{N_c} \|p_{1,c}(x) - p_{2,c}(x)\|_k^{L^2} + w_g \sum_{j=1}^{N_s} \|[\mathcal{M}_{1,j}(2) - \mathcal{M}_{2,j}(2)]\|_k^2 \\ = w_p \sum_{c=1}^{N_c} \int_\Omega |p_{1,c}(x) - p_{2,c}(x)|^2 dx & + w_g \sum_{j=1}^{N_s} \|[\mathcal{M}_{1,j}(2) \cup (-\mathcal{M}_{2,j}(2))]\|_k^2 , \end{aligned} \tag{3.8}$$

where $\|[\mathcal{M}_{1,j}(2) - \mathcal{M}_{2,j}(2)]\|_k = \|[\mathcal{M}_1(2) \cup (-\mathcal{M}_2(2))]\|_k$ is the distance between two surface currents, computed as the norm of the union between surface $\mathcal{M}_1(2)$ and surface $\mathcal{M}_2(2)$ with negative measures, w_p and w_g are scalar weights that ballance the influence of probabilistic and geometric presentations.

We use the large deformation framework [84] that generates dense deformation maps in R^d by integrating time-dependent velocity fields. The flow equation is given by $\frac{\partial h^v(t,x)}{\partial t} = v(t, h^v(t,x))$, with $h(0,x) = x$, and we define $h(x) := h^v(1,x)$, which is a one-to-one map in $\mathbb{R}^d$ (diffeomorphism). We define an energy functional that ensures the regularity of the transformations on the velocity fields: $\|v(t,\cdot)\|_V^2 = \int_{\mathbb{R}^d} \langle Lv(t,x), Lv(t,x)\rangle \, dx$, where L is a differential operator acting on vector fields. This energy also defines a distance in the group of diffeomorphisms:

$$D^2(h,e) = \inf_{v, p^v(1,\cdot)=h} \int_0^1 \|Lv(t)\|_V^2 \, dt. \tag{3.9}$$

The registration optimizations in this chapter are performed using a greedy approach by iteratively performing gradient descent on velocity fields and updating the transformations via an Euler integration of the O.D.E. At each iteration of the algorithm the velocity field is calculated by solving the p.d.e:

$$Lv = F(h), \tag{3.10}$$

where v is the transformation velocity field, $L = \alpha\nabla^2 + \beta\nabla\cdot\nabla + \gamma$, and $F(h)$ is the variation of $\left\|[h\cdot\mathcal{A}_1] - [\mathcal{A}_2]\right\|_k^2$ with respect to h. This variation is a combination of the variation of the L^2 norm on the class posteriors and of the currents norm; computed using the gradient:

$$\frac{\partial \left\|[\mathcal{M}(2)]\right\|_k^2}{\partial x_r} = \sum_{f|x_r\in f}\left[\frac{\partial\eta(f)}{\partial x_r}\right]\sum_{f'=1}^{N_f} k(c(f'),c(f))\eta(f') + \frac{2}{3}\sum_{f'=1}^{N_f}\frac{\partial k(c(f),c(f'))}{\partial c(f)}\eta(f')^t\eta(f), \tag{3.11}$$

given that points $\{x_r, x_s, x_t\}$ form the triangular face f and its center $c(f) = \frac{x_r+x_s+x_t}{3}$ and its area-weighted normal $\eta(f) = \frac{1}{2}(x_s - x_r)\otimes(x_t - x_r)$.

The currents representation is generalized to account for not only surface meshes but also other m-submanifolds such as point sets or curves. The currents associated to an oriented m-submanifold $\mathcal{M}$ is the linear functional $[\mathcal{M}]$ defined by $[\mathcal{M}](\omega) = \int_{\mathcal{M}}\omega$. When $\mathcal{M}(0) = \bigcup x_i$ is a collection of points $[\mathcal{M}(0)]$ is a set of Dirac delta measures centered at the points i.e. $[\mathcal{M}(0)] = \sum_i \alpha_i\delta(x - x_i)$. When $\mathcal{M}(1)$ is a curve in $\mathbb{R}^3$, $[\mathcal{M}(1)]$ is the vector valued Borel measure corresponding to the collection of unit-tangent vectors to the curve, distributed with density equal to the element of length dl:

$$\left\|[\mathcal{M}(1)]\right\|_k^2 = \sum_{l=1}^{N_l}\sum_{l'=1}^{N_l}\left\langle\tau(l),\tau(l')\right\rangle k(c(l),c(l')), \tag{3.12}$$

where N_l is the number of line segments, and for any segment l with vertices v_0 and v_1, $c(l) = \frac{v_o+v_1}{2}$ is its center and $\tau(l) = v_1 - v_0$ is its tangent vertor with its length capturing the length of the line segment.

Using extra submanifold presentation helps capture important properties of the target anatomy, and hence could potentially direct the registration and improve the result, see Glaunes [45] for more details.

3.3 Efficient implementation

The implementation of our registration framework is based on two critical sections: large deformation diffeomorphic image registration and currents norm computation. The former requires a linear solver (Eq. 3.10) on a $M\times M$ matrix where M is the number of input volume elements (≈ 10 millions on typical brain image). The linear system is sparse and there exists efficient solver with complexity of $O(M\log(M))$. The performance is even further amortized using a multiscale iterative method resembling a multigrid solver. The method maps well to the GPU architecture and significantly reduces the running time

from several hours on eight-cores sever to a few minutes on commodity hardware. We refer to the work by Ha *et al.* [54] for details of the method and implementation of large deformation diffeomorphic registration on GPUs. Here, we concentrate on the problem of how to implement norm computation efficiently based on GPU methodologies.

3.3.1 Particle mesh approximation for currents norm computation

The major challenge of computing the currents norm (Eq. 3.7) for real brain surfaces is the high computational cost to compute the dissimilarity metric of all pairs of surface elements, which is $O(N_f^2)$ where N_f is the number of faces. A surface extracted from a N^3 volume has the average complexity of $N^{2.46}$ faces [107], that produces millions surfaces for a typical 256^3 input.

For computational tractability, Durrleman *et al.* [37] used a sparse representation of the surface based on matching pursuit algorithm. On the other hand, an efficient framework based on the standard fast Gauss transform [51] requires the construction and maintenance of the kd-tree structures on the fly. The primary problem of these approaches is that while the performance is insufficient for realtime applications on conventional systems, they are too sophisticated to make use of processing power of modern parallel computing models on GPUs. Also in practice, we use large kernel width for the currents norm to match major structures. This is not ideal for kd-tree based implementations that are designed for querying small set of neareast neighbor. Implementing these ideas on GPUs imposes other challenges, and they are unlikely to be efficient.

Here, we employ a more parallelizable approach based on the particle mesh approximation (PM). This approximation has been extensively studied in a closely related problem - the cosmological N-body simulation, which requires the computation of the interaction between every single pair of objects (see Hockney and Eastwood [63] for details). The particle mesh approximation, as shown on Fig. 3.3, includes four main steps :

- **Grid building** determines the discretization error or the accuracy of the approximation. It also specifies the computational grid, the spacial constraints of the computation. The quantization step in each spacial direction determines the grid size, hence, the complexity of the grid computation. The finer the grid, the higher the quality of the approximation but the more computation involved.

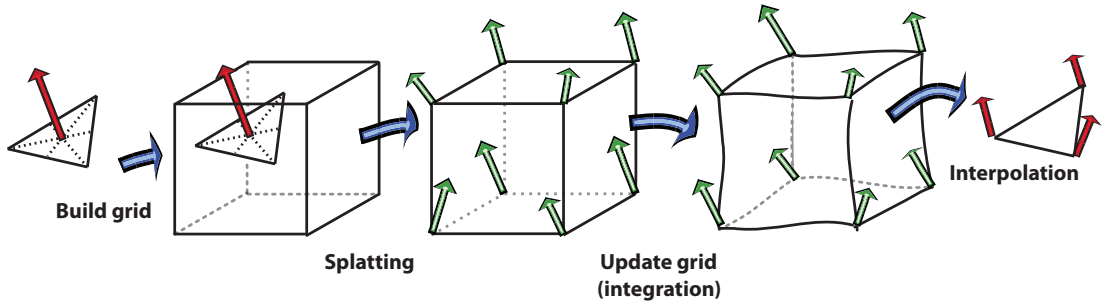


Figure 3.3. Particle Mesh approximation algorithm to transform the computation from irregular domain to regular domain based on four basic steps: grid construction, splatting, integration and interpolation.

- **Splatting** maps computation from an unstructured grid to a structured grid. It is the inverse operation of the interpolation.
- **Integration** performs the grid computation and updating step. As the computation, which involves kernel convolution and gradient computation, is taken place in a regular domain, the integration can exploit the parallel processing power of special computing units such as GPUs.
- **Interpolation** interprets computational results from the image space back to the geometrical space, in other words, to reconstruct the unstructured grid out of the structured domain. Marching Cube [75] is an example of techniques using interpolation to extract iso-surfaces from MR images.

The splatting/interpolation operation pair works as a connection between the computation on regular domain and irregular domain. We will go into details of how to implement this interface on the parallel architecture as the method can be widely used not only for the norm computation but any mixed—geometric and probabilistic—computation in general. We consider this strategy a crucial method for efficient parallel computation on an irregular domain.

The error in particle mesh approximation is influenced by two factors: the grid spacing and the width of the convolution kernel, as shown on Fig. 3.4. We chose the image grid spacing; thus the error is bounded by the image resolution. As aforementioned, we use large kernel widths in practice which is ideal for PM. Note that PM approximation breaks down when kernel width is less than grid spacing.

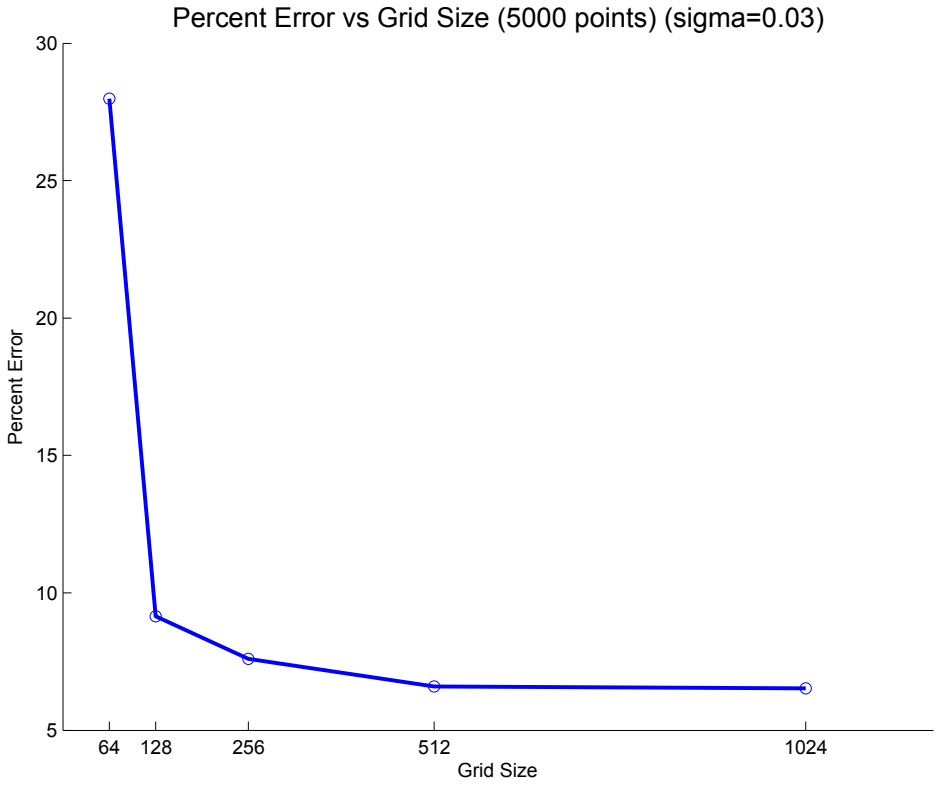


Figure 3.4. The percent error for different for 5000 randomly generated points with different mesh sizes.

While the approximation helps reduce the complexity from N_f^2 to $M \log M$ where M is the volume size of the embedded grid (Figure 3.5), the total complexity of the method is still very high. On a high-end workstation with 8-CPU cores, a highly optimized multithreaded implementation in C++ takes several hours for one matching pair, and hence cannot be used for parameter exploration and real-time analysis. Based on the GPU framework by Ha *et al.* [54], we developed an implementation that runs entirely on the GPU to exploit parallel efficiency of regular grid presentation.

3.3.2 Efficient implementation of particle mesh method on GPUs

To achieve the maximum performance efficiency, we optimized the four-steps of particle mesh method on GPUs. Here, we describe the performance keys and important details to implement these steps.

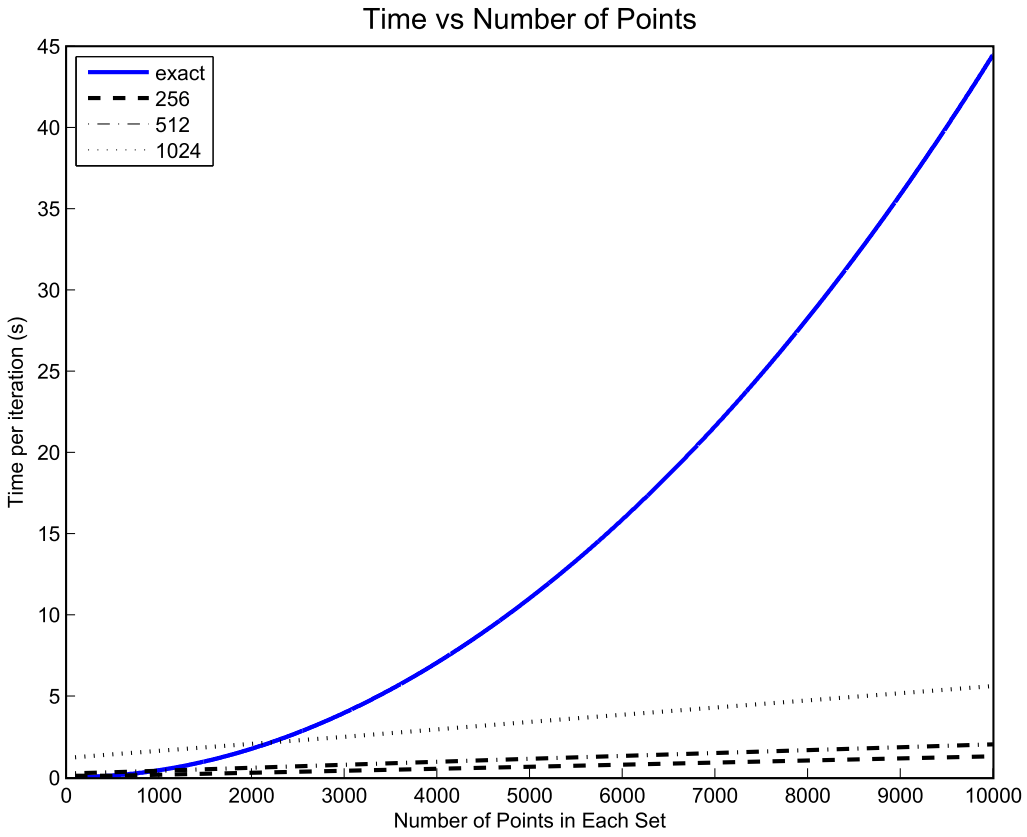


Figure 3.5. Run time comparisons between direct computation and the particle mesh implementation for various grid sizes.

3.3.2.1 Grid building

Without prior information, computational grid is typically chosen as a discretization of the bounding box with extra border regions to prevent out-of-bound quantization error. Since probabilistic and geometric descriptors co-exist in our representation, the computational grid is effectively chosen as the original grid. This selection guarantees that it will not introduce further quantization errors than the original discretized errors inherent to the construction of geometric descriptors. This strategy also limits the complexity of the combining technique to the original order of computation if we use only probabilistic terms.

3.3.2.2 Splatting

The main purpose of the splatting function is to construct a regular n-dimensional scalar or vector field from its discrete sample points. The constructed grid should

satisfy an inverse operation, the interpolation, so that when this operation is applied to the reconstructed grid it will reproduce the sample points. In other words, with E is an arbitrary input $Interpolation(Splatting(E)) = E$. This duality of splatting and interpolation reflects the fact that probabilistic and geometry descriptors are just the domain representations of the same subject. Hence, we could unify their computation without losing accuracy. We also exploit the duality to validate the correctness of our implementation of the splatting function through its dual-counterpart.

The splatting function is defined by Trouvé and Younes [120] through a linear operator $\aleph$ that applies a mapping vector field $v : \mathbb{Z}^d \to \mathbb{R}$ to a discrete image $I : \mathbb{Z}^d \to R$ to perform an interpolation on the grid $G_v = \{x + v(x) | x \in \mathbb{Z}^d\}$, mathematically saying

$$(\aleph I)(x) = (\mathfrak{I})(x + v(x)), \tag{3.13}$$

with $\mathfrak{I}$ being linear interpolation, defined by

$$(\mathfrak{I})(I)(x) = \sum_{\epsilon \in \{0,1\}^d} c_\epsilon(x) I(\lfloor x_1 \rfloor + \epsilon_1, \lfloor x_2 \rfloor + \epsilon_2, \ldots, \lfloor x_d \rfloor + \epsilon_d),$$

with $\lfloor z \rfloor$ being the integer part of real number z and $\{z\} = z - \lfloor z \rfloor$ is the fractional part. The coefficient $c_\epsilon(x)$ is defined as

$$c_\epsilon(x) = \prod_{i=1}^{d} (\epsilon_i + (1 - 2\epsilon_i) x_i).$$

While the splatting operator was defined through a vector field, the splatting conversion from the irregular grid to the regular domain for an arbitrary input is defined with being a zero vector field. Figure 3.6 displays the construction of a regular grid presentation of geometrical descriptors in 2D through splatting operator. The value at a grid point is computed by accumulating values interpolated at that point from its geometrical neighbors. Thus, closer neighbors will have more influence on the value of the point than farther points. In fact, we only need to consider the one-ring neighbors as farther points have a negligible contribution to its final value. We also assume that the field is continuous and smooth.

Though the splatting operator has a linear complexity in terms of the size of geometry descriptors, it is the performance bottleneck in practice. The single CPU thread-based splatting function is too slow for interactive applications. Even close discrete points do not share the same cache as the definition of a neighbor in 3D does not map to a neighbor in

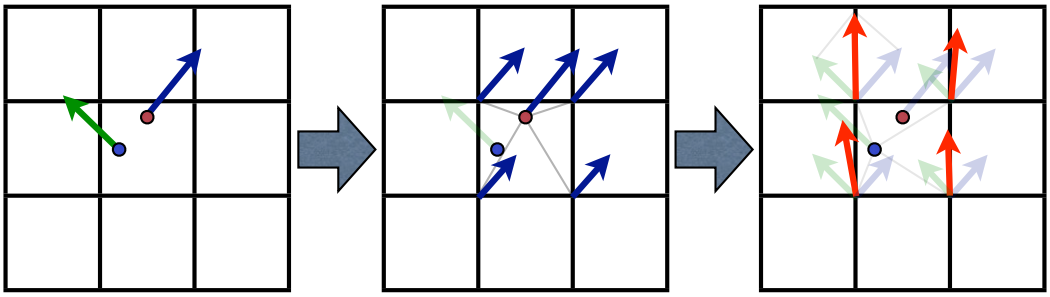


Figure 3.6. Geometrical conversion based on a splatting function with zero velocity field v (Eq 3.13). The method served as a bridge to transform the computation from an irregular grid to a regular grid which allows an efficient parallel implementation.

the linear CPU cache. The multithread-based CPU splatting, which assigns each thread a single geometrical element, however, has a resource fighting problem. That is, when we integrate grid value from its neighbor submanifold elements, it is likely that there are several elements in the neighbor, and these elements, which are assigned different threads, may try to accumulate the grid value at the same time. GPU implementation also has to face the resource-fighting problem.

We can apply mutex locking to resolve the conflict. However, it is inefficient with thousands of threads on GPUs. A better solution is based on atomic operations, which are guaranteed to complete without being interrupted by the actions of other threads. Currently, CUDA does not support atomic operations for floating point numbers but integer numbers. Here we propose two different approaches for splatting computation: the collision-free splatting scheme via a fast parallel sorting and the atomic splatting scheme using a fixed-point representation.

The **collision-free splatting** scheme is applied for systems without any atomic operation support. As shown on Fig. 3.7, we employ a fast parallel sorting to resolve the shared-resource fighting problem. The algorithm involves three steps:

- Compute the contribution of each geometrical descriptor to grid nodes.
- Sort the contribution based on node indexes. The contribution array is segmented based on node indexes.
- Apply a parallel segmented prefix-sum scan [60] to integrate all node values.

All of these steps are implemented efficiently in parallel on the GPU. The first step is simply a point-wise computation. For the second step, we apply the fast parallel

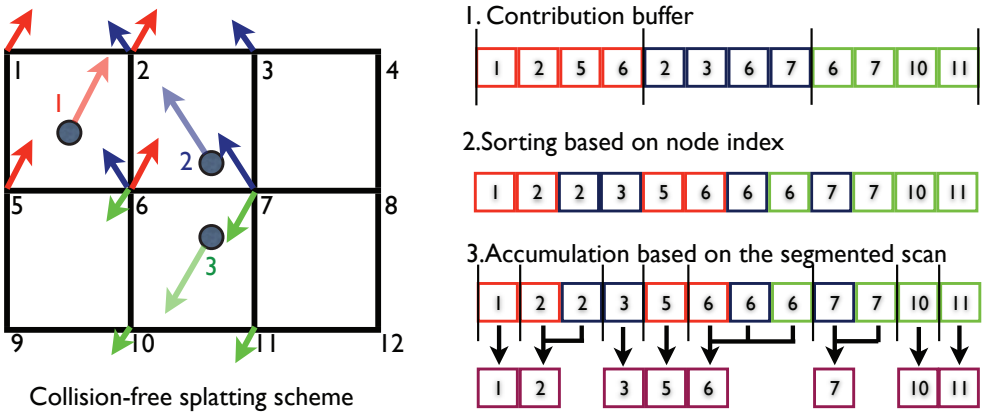


Figure 3.7. Collision-free splatting implementation using fast parallel sorting. The method is based on ordering the node contribution ID to resolve resource conflicts which allows a parallel efficient integration based on an optimal parallel prefix scan implementation.

sorting [82]. The third step is performed using the optimal segmented scan function in the CUDA Performance Processing library (CUDPP) [60]. The sorting scheme on CUDA is a magnitude faster than an optimal multithreaded, multicore implementation on CPUs [33]. While this scheme is quite efficient and is the only solution on CUDA 1.0 devices, its performance largely depends on implmentations of two essential functions: the parallel sorting and the segmented scan. Also the memory requirement of the method is proportional to the number of shooting points (which can be as large as the grid size) and the size of the neighbor (which is eight for 3D implementation). The memory usage becomes even worse as fast parallel sorting based on radix sorting that could not perform in-place but out-of-place sorting so the method requires another copy of the contribution array. In many circumstances, we found a better solution both in terms of performance and memory usage based on atomic operations supported on the CUDA 1.1 and later devices.

The **atomic splatting** scheme resolves the shared-resource fighting problem using atomic operations. While atomic floating point operations are currently not supported, it is possible to simulate this operation based on a fixed-point presentation. In particular, instead of accumulating the floating point buffer, we explicitly convert floating point values to integer representations through a scale. This allows the accumulation to be

performed on integer buffers.

The parallel splatting accumulation is implemented by assigning each geometrical descriptor a GPU thread, which computes the contribution to the neighbor grid points based on its current value and distances to the neighbor grids. These floating point contribution values are then converted to integer presentation through a scale number, which normally chosen as a power of two (we use 2^{20}, in practice) so that a fast shifting function is sufficient to perform the scale. The atomic integer adding operator allows values to be accumulated atomically at each grid point concurently from thousand of threads. In our implementation, the contribution computations—upscale and the integer accumulation steps—are merged to one processing kernel to eliminate (1) an extra contribution buffer, (2) extra memory bandwidth usage to store, reload, and rescale the contribution buffer from the global memory, and (3) the call overheads of the three different GPU processing kernel. The accumulation result is then converted back to floating value by the division to the same scale value.

We further amortize the performance on later generations of GPU devices using the atomic shared-memory operations, which are a magnitude faster than operations on GPU global memory. We exploit the fact that in fluid registration the velocity field is often smooth and shows large coherence between neighbors, so it is likely that two close points will share the same neighbors. Thus, it would be better to accumulate the values of the shared neighbors in the shared-memory instead of the global memory. We assign each block of threads a close set of splatting point and maintain a shared memory accumulation buffer between threads of the same block. The accumulation results on the shared memory are then atomically added to the accumulation buffer on the global memory. This approach exploits the fast atomic functions on the shared memory and at the same time reduces the number of global atomic operations. This optimization is especially effective on a dense velocity field, which shows significant coherency between neighbor points.

3.3.2.3 Interpolation

Even though the probabilistic and geometric descriptors are represented by independent data structures on separate domains, they are, in fact, different representatives of the same anatomical subject that is updated during ODE integration under the influence of the time-dependent velocity field along a registration evolution path. While the com-

putation occurs on the regular grid, interpolation is necessary to maintain the consistency of multicompartment anatomies as they undergo deformation. Given a deformation h, we update probabilistic images using backward mapping and geometries using forward mapping (Eq. 3.3).

A computationally efficient version of ODE integration is the recursive equation that computes the deformation at time t based on the deformation at the time $t-1$. That is, $h_t = h_{t-1}(x + v(t-1))$. This computation is done by a reverse mapping operator (Fig. 3.8), which assigns each destination grid point a value interpolated from the source volume grid's neighbor points. The reason for using a reverse mapping operator instead of a forward mapping one is to avoid missing data values at the grid points that make computation of forward mappings intractable. A reverse mapping requires the maintenance of reverse velocity fields. The update of geometric descriptors is based on a forward vector field derived by inverting direction of the reverse velocity field. Algorithmically, the probabilistic and geometric descriptors are updated in opposite directions. The updating process of geometric descriptors is illustrated on Fig. 3.9.

While the selection of interpolation strategies such as 3D linear interpolation, cubic interpolation, high order interpolation depends on the quality requirement of the registration, the updating process of both probabilistic and geometric descriptor need to share the same interpolation strategy so that they are consistent with one another. In practice, 3D linear interpolation is the most popular technique because it is computationally simple and efficient, and it can produce satisfactory results especially with large kernel width for

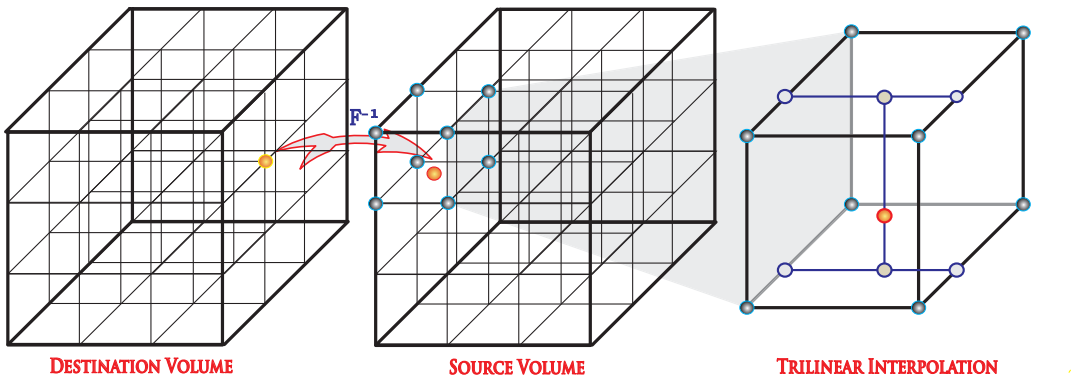


Figure 3.8. Reverse mapping based on 3D trilinear interpolation that eliminates the missing data of a forward mapping. The implementation on GPU exploits the hardware interpolation engine to achieve significant speed up.

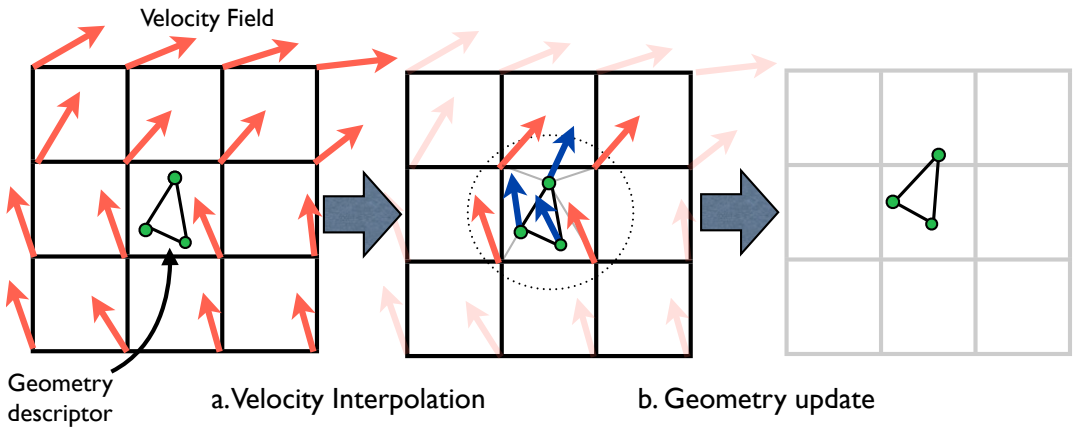


Figure 3.9. Geometries are updated through the interpolation from the velocity field. This step maintains the consistency between probabilistic and geometrical compartments of the mixture model.

currents norm. On GPUs, this interpolation process is fully hardware accelerated with 3D texture volume support from CUDA 2.0 APIs. Other optimization is based on the texture cache that helps improve the look up time from the source volume due to large coherency in the diffeomorphic deformation fields.

3.4 Other performance optimizations

Besides an optimized, parallel implementation for particle mesh computation, we further improve the performance with parallel surface normal and multiscale computation on GPUs. These optimizations keep the entire processing flow on GPUs, eliminating the need to transfer the data back and forth between CPU memory and GPU memory which is the main bottleneck for many GPU applications.

3.4.1 Parallel surface normal computation on GPUs

While the geometrical descriptor involved in our registration framework was defined as a surface element (a triangle) with all property values on its vertices, the computation was defined at the centroid following its normal direction and weighted by the size of the surface element (Eq. 3.11). This computation requires the computation of a weighted normal at the centroid of each surface element from the geometric descriptors. We perform this operation in parallel on the GPU by assigning each surface element a thread. We then employ the texture cache to load the geometrical data from global memory. While

the neighbor triangle shared the same vertices, the loading values are highly likely in the cache and cost almost the same amount of time to access from the shared memory. We also store the three components of the normal in three separated arrays to allow coalesced access that gives better memory bandwidth efficiency.

3.4.2 Multiscale computation on GPUs

Multiscale registration is an advanced registration technique to improve quality of the results by registering anatomies at different scale levels. The method also handles the local optimal matching of gradient-descent optimization. In our registration framework, the primary purpose of doing multiscale computation is to capture both the large changes in the shape and also the small changes as the registration anatomy converged to the target. The method effectively handles the nonlinear, localized shape changes, as shown on Fig. 3.10. It also serves as an effective method to increase the convergence rate and reduces the running time significantly. The challenge of applying multiscale computation is that there is no mathematical foundation for exact multiscale computation on a regular grid. The Level-Of-Detail techniques (LOD) are the only approximations that gives no guarantee on the quality. Here, we achieve the multiresolution effect through changing the size of a registration kernel, such that we use a larger kernel width and step size to mimic the effect of large scale and smaller kernel width and step size to capture the details. Our method did not require re-sampling of the grids, so there are no additional quantization errors.

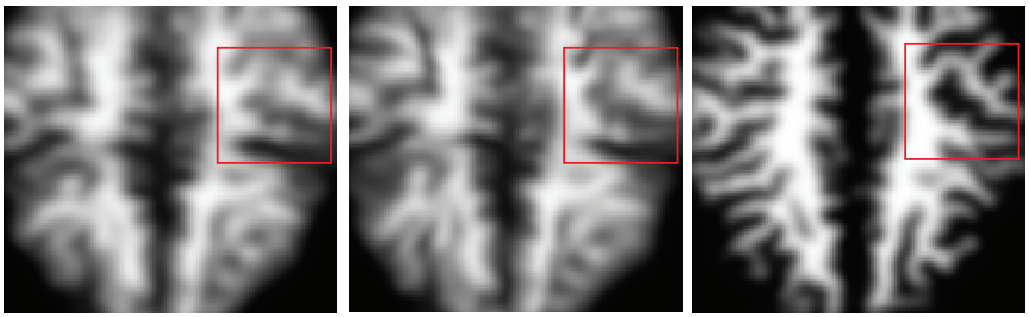


Figure 3.10. Multiscale registration using different sizes of computation kernels help capture large and small scale changes in different levels and also increase the convergence rate of the algorithm.

3.5 Results

For evaluation, we used an AMD Phenom II X4 955 CPU commodity system, 6GB DDR3 1333, with NVIDIA GTX 260 GPU 896MB. We quantify both aspects of the method: registration quality and performance. Runtime is measured in milliseconds.

3.5.1 Registration quality

We have applied the registration method for mapping neonatal MRI scans to 2-year MRI scans of the same subjects in 10 datasets. The datasets are taken from an ongoing longitudinal neuroimaging study with scans acquired at approximately 2 weeks, 1 year, and 2 years of age. Due to rapid early brain development, each longitudinal MR scan shows significant changes in brain size and in tissue properties. For comparison, we also applied the standard intensity based deformable registration using mutual information (MI) metric and B-spline transformation proposed by Rueckert *et al.* [105], which has been applied for registering 1-year-old and 2-year-old infants [2]. The T1 weighted images before and after registration using the different approaches for the first three subjects are shown in Fig. 3.11,3.12.

A quantitative study of the performance of the registration method is performed by measuring the overlap between the transformed segmentation maps of neonates to the segmentation maps of 2-year-olds. Since we consider the segmentation maps at 2 years of age to be the standard, we use the following overlap metric:

$$Overlap(h \cdot S0, S2) = \frac{|h \cdot S0 \cap S2|}{|S2|}, \tag{3.14}$$

where $h \cdot S0$ is the transformed neonate segmentation map, $S2$ is the reference 2-year segmentation map, and $|\cdot|$ indicates the volume of a binary map. We note that this metric gives considerably lower values for deviation from $S2$ than the standard Dice coefficient.

Table 3.1 shows the quantitative analysis for the brain parenchyma (a combination of white matter and grey matter) and cerebellum segmentation maps without registration, using standard MI registration , and our method. We use brain parenchyma since white matter and grey matter on their own are hard to distinguish in early developing brains. Registration using MI fails for parenchyma because it does not account for the two white matter distribution in neonates.

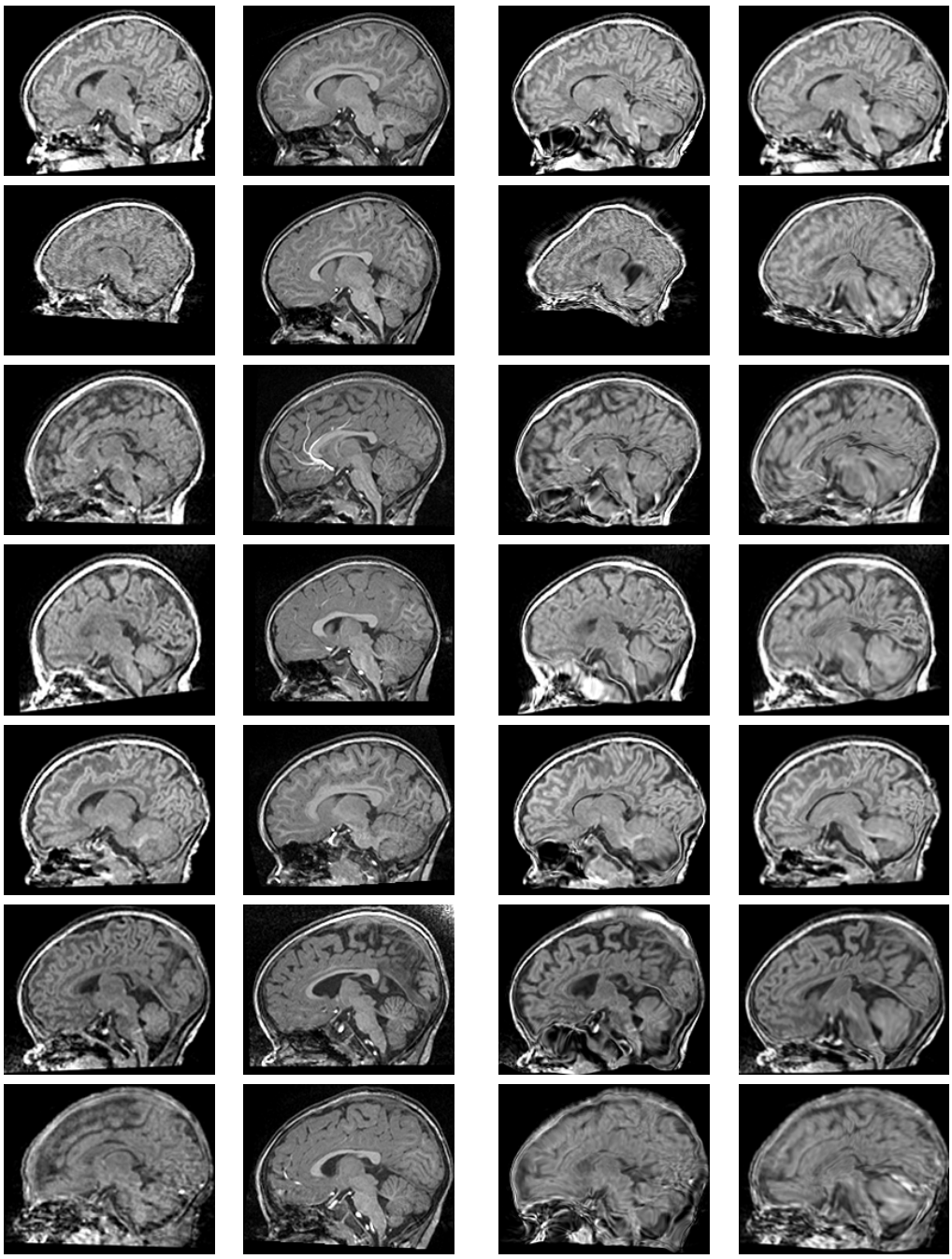
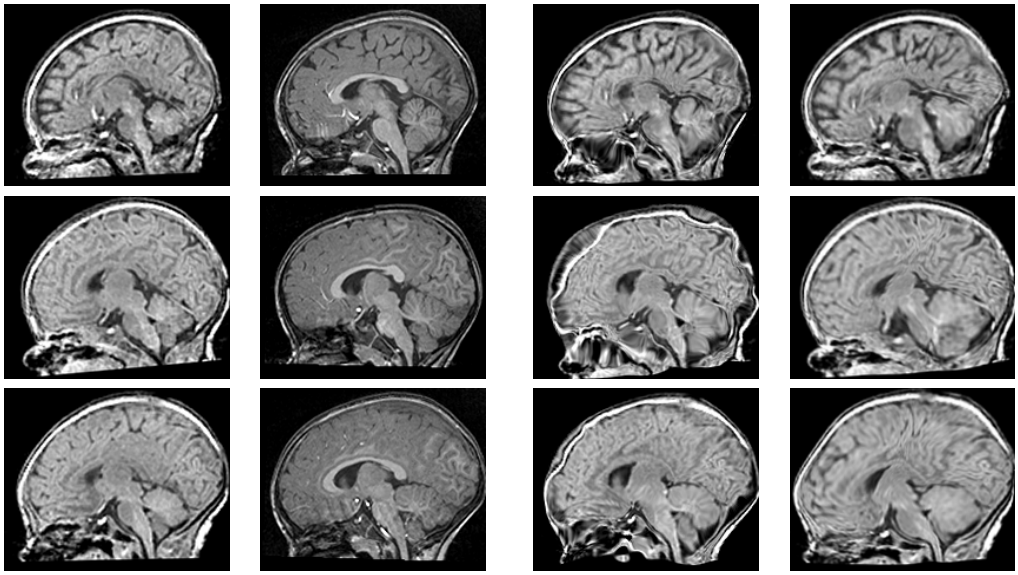


Figure 3.11. Registration results of neonates mapped to 2-year-olds. From left to right: (a) neonatal T1 image after affine registration, (b) reference T1 image at 2 years, followed by (c) neonatal T1 after deformable mutual information registration using B-splines, and (d) after combined probabilistic and geometric registration. From top to bottom: subject 0012, 0102, 0106, 0121, 0130, 0146 and 0156.



(a) (b) (c) (d)

Figure 3.12. Registration results of neonates mapped to 2-year-olds. From left to right: (a) neonatal T1 image after affine registration, (b) reference T1 image at 2 years, followed by (c) neonatal T1 after deformable mutual information registration using B-splines, and (d) after combined probabilistic and geometric registration. From top to bottom 0174, 0177 and 0180.

Table 3.1. Overlap measures comparing the registered segmentation maps against the reference segmentation maps for the parenchyma and cerebellum structure, obtained without deformation (None), deformable mutual information registration (MI), and our proposed method (P+G).

Subject		0012	0102	0106	0121	0130	0146	0156	0174	0177	0180
Parenchyma	None	0.83	0.55	0.81	0.83	0.92	0.75	0.82	0.84	0.78	0.71
	MI	0.80	0.45	0.75	0.78	0.90	0.71	0.78	0.83	0.77	0.69
	P+G	0.90	0.88	0.88	0.87	0.88	0.86	0.88	0.88	0.91	0.87
Cerebellum	None	0.57	0.26	0.51	0.51	0.64	0.56	0.54	0.50	0.53	0.59
	MI	0.76	0.21	0.59	0.52	0.73	0.82	0.71	0.57	0.63	0.78
	P+G	0.88	0.82	0.88	0.88	0.86	0.90	0.91	0.89	0.90	0.89

Registration using both probabilistic and geometric descriptors provides better results which are generally more stable for the structures of interest. In particular, our method better preserves the shape of the cerebellum, which has weak intensity boundaries in regions where it touches the cerebrum and thus cannot be registered properly using only image based information. Another significant challenge is that the cerebellum growth is distinctly different from the growth of neighboring structures. Using cerebellum boundary represented by currents, our method capture the growth better than MI registration.

3.5.2 Performance

We quantify the performance with two critical steps in Particle Mesh approach: the splatting and the interpolation. We measured the performance with typical volume sizes.

3.5.2.1 Splatting

The splatting performance varies largely depending on the regularity of the deformation fields due to the memory collision problem. Here we measured with three types of deformation fields: a random deformation—which maps points randomly over the whole volume, a diffeomorphic deformation—the typical type of deformation from the registration of brain images that we use in our framework, and a singular deformation—which collapses to a point in the volume. Table 3.2 shows the runtime comparison in milliseconds of different splatting implementation mentioned in Section 3.3.2.2: CPU reference, collision-free sorting approach, atomic fixed-point operation, and atomic operation with shared memory. The result shows that the performance gain of GPU approaches varies depends on the regularity of the deformation field inputs. The singular deformation has the lowest performance gain because most of the value accumulated to a small point neighbor and hence parallel accumulation is greatly limited. Though having better performance gain, the random deformation spreads out the whole volume that leads to ineffective caching (both in GPUs and CPUs). Fortunately, our atomic optimization with shared memory achieved the best performance gain with diffeomorphic deformation which we used in practice. The main reason is that the diffeomorphic deformation shows large coherence between neighbor points that allows more effective caching through GPU shared memory. The collision-free approach based on sorting shows stable performance since it is independent from the memory collision of other approaches.

Table 3.2. Runtime comparison, in milliseconds, of different splatting implementations on volume sized $144 \times 192 \times 160$ and $160 \times 224 \times 160$ using collision-free sorting approach, atomic operation with fixed point presentation, atomic operation on the shared memory and CPU reference.

Size	Method	CPU	Sorting	Atomic	Atomic-shared
$144 \times 192 \times 160$	Random	826	105	29	30
	Diffeomorphic	331	110	105	14
	Singular	224	105	40	41
$160 \times 224 \times 160$	Random	1435	215	75	76
	Diffeomorphic	775	224	152	21
	Singular	347	215	144	144

3.5.2.2 Interpolation

The interpolation implementation result has been dicussed in Chapter 2. The runtime shows that reverse mapping using the accelerated hardware achieves the best performance and is about 38x faster than CPU reference implementation. However, this method suffers from lower floating-point accuracy. To not futher introduce more errors to the approximation, we apply the 1D-linear texture-cache implementation instead which is as fast as the accelerated hardware but retains the floating point precision. The method produces results equivalent to the CPU reference.

3.5.2.3 Probabilistic descriptor registration

We have also compared the performance between our method and the standard MI registration. Registrations using our approach on the GPU takes 8 minutes on average, while registration on the CPU using mutual information metric and B-spline transformation takes 100 minutes on average. Detailed time measures are listed in Table 3.3.

Table 3.3. Time elapsed, in minutes, for registration using deformable mutual information (MI) on the CPU (AMD Phenom II X4 955, 6GB DDR3 1333) and our proposed approach (P+G) on the GPU (NVIDIA GTX 260, 896MB) with 1000 iterations of gradient descent.

Subject	0012	0102	0106	0121	0130	0146	0156	0174	0177	0180
MI on CPU	92	63	103	92	101	112	106	99	91	96
P+G on GPU	9	8	8	8	8	7	9	8	7	7

Overall, computing the currents norm and its gradient between a surface with 160535 triangular faces and another with 127043 faces takes approximately 504 seconds on CPU, while it takes 0.33 seconds with our GPU implementation. The speed gain is in order of three magnitudes over the equivalent CPU implementation using particle mesh, while the computing time for the exact norm on CPU is difficult to measure since it takes significantly longer. The proposed algorithm typically converges in 1000 iterations, so on average it takes less than 8 minutes to register two anatomies. This allows us to perform parameter exploration and real-time analysis on a single desktop with commodity GPUs.

3.6 Conclusions

We have proposed a registration framework that makes use of the probabilistic and geometric structures of anatomies embedded in the images. This allows us to enforce matching of important anatomical features represented as regional class posteriors and tissue boundaries. Our framework allows us to register images with different contrast properties by using equivalent anatomical representations, and we have demonstrated results for registering brain MRIs with different white matter appearances at early stages of growth. The overlap validation measures in Table 3.1 show that geometric constraints, particularly for the cerebellum, are crucial for registering structures undergoing significant growth changes.

In the future, we plan to apply this framework in early neuro-developmental studies for analyzing the effects of neurological disorders such as autism and Fragile X syndrome. The proposed registration framework is generic and independent of the application domain, it can thus be applied to any registration where one encounters large-scale deformation and different appearance patterns. We also want to incorporate other submanifold representations and their computation such as point sets ($\mathcal{M}(0)$) and curves ($\mathcal{M}(1)$). Such additional representations are potentially critical in clinical applications involving anatomical landmark points (e.g., Anterior Commissure and Posterior Commissure) as well as curve structures (e.g., blood vessels, sulcal lines, white matter fiber tracts). All these computations can be done efficiently on GPUs, and potentially will improve the results by guiding the registration process to preserve critical geometries. The efficiency of the GPU method also provides an opportunity to apply the algorithm for high quality atlas formation using our framework on a GPU cluster, which gives us the ability to perform statistical tests that are previously impossible due to excessive time requirements.

CHAPTER 4

AN OUT-OF-CORE FRAMEWORK FOR MULTI-IMAGE PROCESSING

The construction of a brain atlas often requires applying image processing operations to multiple images (often hundreds of volumetric datasets), which is challenging due to the large amount of computational and memory the construction requires. In this chapter, we will introduce MIP, a Multi-Image Processing streaming framework to harness the processing power of heterogeneous CPU/GPU systems. With MIP we show specially designed streaming algorithms and data structures that provides an optimal solution for out-of-core multi-image processing problems both in terms of memory usage and computational efficiency. MIP makes use of the asynchronous execution mechanism supported by parallel heterogeneous systems to efficiently hide the inherent latency of the processing pipeline of out-of-core approaches. Consequently, with computationally intensive problems, the MIP out-of-core solution could achieve the same performance as the in-core solution. We demonstrate the efficiency of the MIP framework on synthetic and real datasets.

4.1 Introduction

Multi-image processing is an advanced image-processing technique that relies on processing a multitude of images (often hundreds) which describe a certain aspect of a population to harness the abundance of the input data to significantly improve the quality and robustness over single-image comparision approaches. In noise reduction, for example, while most of the single image processing methods reduce the noise but end up softening the image as well, the multi-image averaging has the power to reduce the noise without compromising fine details [47]. Furthermore, it improves the bit depth of the combined image and can achieve high-end photography effects from low-end devices. The technique is common in low-light photographs such as night photography or astro-photography [17, 1]. The motion estimation techniques, for example the optical

flow, extract the velocity from a sequence of adjacent frames captured by cameras. These methods play key roles in visual robot control, surveillance, virtual analytic, virtual training, virtual simulation as well as video compression [57, 35]. Multi-image processing techniques are also applied in 3D reconstruction based on analyzing different images from different view points of an object under various lighting conditions and structured light patterns [57]. The method is common in nondestructive and large object 3D scan. It provides feasible solutions for 3D reconstruction of extremely large and immovable objects such as monuments, large statues or buildings [46].

Multi-image processing does not necessarily require a preprocessed, or normalized input dataset, but typically performs the analysis from hundreds to millions of images. The method has received growing research interest as the development of sensor networks produces more data, and multi-image datasets become more accessible through public-shared photograph databases such as Google and Flicker [46, 111, 61]. By analyzing thousands of images from large collections of unorganized photographs taken by different cameras in various conditions of the same scene, Snavely [111] proposed a through-view synthesis method that allows virtual tourism of the world's interesting and important sites. James Hays and Alexei Efros [61] presented a new image completion algorithm that creates pleasant, human-indistinguishable synthetic images of nature from millions of images collected from the Web.

Here, we consider an atlas construction problem, as shown on Figure 4.1, from the viewpoint of a multi-image processing technique. This leads us to our introduction of the multi-image processing framework, a generalization of our GPU image processing framework. There are two primary challenges in the implementation of a multi-image processing framework: First, the techniques involve huge amounts of data that easily exceed the direct processing capability of the system. Second, they require massive amounts of computation, which results in the computations requiring days or even months to complete. As a result, using multi-image techniques often involve supercomputing systems [29] or large-scale clusters to run [111, 56], which limits the use of multi-image processing techniques to large laboratories. A solution based on commodity hardware will make this technique available to smaller labs, increase the influence of these techniques in research, and present robust solutions for many existing problems.

In this chapter, we discuss a solution for the multi-image processing problems on commodity hardware using graphic processing units (GPUs) combined with an out-of-core

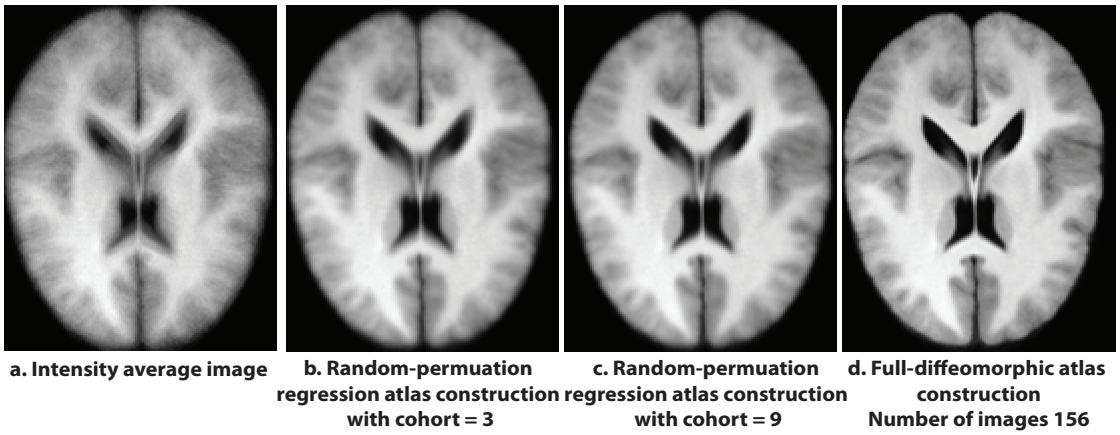


Figure 4.1. Atlas construction result on the ADNI data set composed of 156 images sized $144 \times 192 \times 160$, with different average computations: a) the intensity average and the diffeomorphic atlas constructions with b) random permutation ([56]) with cohort size of 3 images c) random permutation with cohort size of 5 images and d) and all image using our out-of-core streaming framework. It is clear that the ability to compute the atlas using nonlinear diffeomorphic registration with all the image yields a discernible improvement in the quality of the construction.

streaming model. The main contributions of this chapter are:

- We introduce a high-performance, multi-image processing framework with a proof-of-concept optimal streaming model.
- We define basic building blocks of a general framework which allows efficient implementation of multi-image algorithms.
- We introduce concepts for implicit and explicit pipelining and prove that these are optimal solutions.
- We analyze reasons for streaming degeneracy and provide a solution based on an order-independent model.
- Our performance analyses serve as the guidance to help developers to profile performance and to make quantitative decisions.

4.2 Related work

While the use of GPUs appears to be a good solution to the computing requirements of multi-image processing techniques, the large memory footprint remains an open problem. Though providing ample memory bandwidth, the size of the on board GPU memory is very limited. But as GPU programs can only access on-board memory, all required data need to be present on the card, so out-of-core methods must be employed.

Out-of-core processing is a class of cache-friendly techniques of *external memory* algorithms [123, 95] generally applied to handle extremely large data which are unable to be addressed by traditional, in-core processing methods. Out-of-core techniques are specially designed to reduce the I/O bottleneck inherent to external memory algorithms. The techniques have received special research interest as the amount of data is growing rapidly. Goodrich *et al.* [48] developed I/O efficient algorithms for a collection of problems in computational geometry. Chiang *et al.* [27] gave I/O efficient techniques for a wide range of computational graph problems. Independent from the amount of system memory, the out-of-core approaches are the more scalable and affordable solutions for commodity computing systems than shared-memory systems.

There are three primary approaches to out-of-core programming. The first is to use virtual memory based on operating system support. It is simple and unified for both in-core and out-of-core processing. However, due to a lack of application-specific knowledge about the data dependence and parallelism, this method often leads to a poor performance [126]. The second approach is to use compiler directed I/O to convert a program from in-core to out-of-core [16, 87, 19]. For programs with complicated data dependencies this approach is not as effective as the third approach that we use here: the explicit I/O controls by developers. These methods concentrate on techniques to improve the cache coherency such as *caching* and *prefetching* [78, 23, 26, 65, 13] to reduce the I/O necessary for blocks already in main memory and/or by overlapping I/O operations with main-memory computations. This method exploits particular computational properties of each individual problem as part of the algorithm design. While the explicit I/O controls are mostly application-specific, our method is able to be applied to a wide class of applications such as out-of-core multi-image processing.

Our out-of-core strategy exploits two key performance concepts: prefetching and data-transfer-hiding based on an asynchronous streaming execution model. Asynchronous processing is a pipeline-concurrent execution model that exploits the availability of multiple execution units in the system to run independent tasks concurrently [71]. This strategy reduces idle stages and increases the resource usage. It can also hide data transfer by prefetching data. When processing units finish current tasks, they can start the next tasks without delay. In many circumstances, using this model significantly increases the overall system throughput.

The asynchronous processing is realized with streaming models for both tasks and

data. Streaming is an efficient model for parallel processing in that a task is divided into smaller entities to allow their parallel executions. A stream is an abstraction of an execution unit; in particular, it represents a sequence of commands that are executed or accessed in a particular order. Pure data streams encourage a data parallelism processing model, while pure task streams are more amendable to the task parallelism model. In practice, a stream may be data-based, tasked-based, or even a mixture of the two. The only restriction in a stream is the execution order that is satisfied by a sequential consistency model [72], which makes a stream equivalent to a synchronous process. Different streams, on the other hand, may execute their commands out-of-order with respect to each other.

4.3 The construction of the multi-image processing framework

As we can see from the atlas construction Algorithm 5 [54], a multi-image algorithm involves several multi-image operations, most of which are direct extensions of single-image processing operations through a loop over all the input. We build our multi-image processing framework upon the single-image high-performance multiscale processing framework proposed by Ha *et al.* [56] so that we are able to exploit the optimized performance of the existing framework.

4.3.1 Multi-image processing operators

We define the multi-image processing framework using a construction method that builds regular multi-image operators from basic building blocks. This strategy allows fine-grained and multilevel parallelism in that we could exploit different execution strategies on each implementation level to make use of available resources. Here, we classify

Algorithm 5 Atlas construction framework

1: **Input** : N volume inputs
2: **Output**: Template atlas volume
3: **for** $k = 1$ to max_iters **do**
4: Fix images I_i^k, compute the template $\hat{I^k} = \frac{1}{N}\frac{\sum_{i=1}^{N} I_i^k w_i}{\sum_{i=1}^{N} w_i}$
5: **for** $i = 1$ to N **do** {loop over the images}
6: Fix the template $\hat{I^k}$, solve pairwise-matching problem between I_i^k and $\hat{I^k}$
7: Update deformed image I_i^k with current velocity
8: **end for**
9: **end for**

basic multi-image operators into two main groups based on Flynn's taxonomy [41]: the Multiple-Input-Multiple-Output operators (MIMO) and the Multiple-Input-Single-Output operators (MISO).

The basic MIMO operators are defined as functions with equal numbers of inputs and outputs, whereas the n-th output image depends solely on the n-th input images (Figure 4.2a). These functions are the most frequently used in multi-image processing as they are direct extensions of single-image operations. Examples for such operations include adding, shifting, scaling, smoothing, filtering, denoising images, and normalizing the intensity range.

The MISO operators, as illustrated in Figure 4.2b, produce a single or few outputs. Examples for such operations include the computation of an average image, the image energy, cross-correlation, cross-product of images, and finding the maximal and minimal values.

The implementation of general multi-image operators is based on a decomposition strategy that breaks a complex function into multiple basic operations. For example, a general MIMO function that has a number of outputs M which is different from the number of inputs N, and the k-th output depends on multiple inputs, could be implemented as M instances of a MISO operator as shown on Figure 4.3.

Another group of frequently used multi-image operators is the sliding-window operator (Figure 4.4a). This operator computes an output image based on all values in a fixed-size

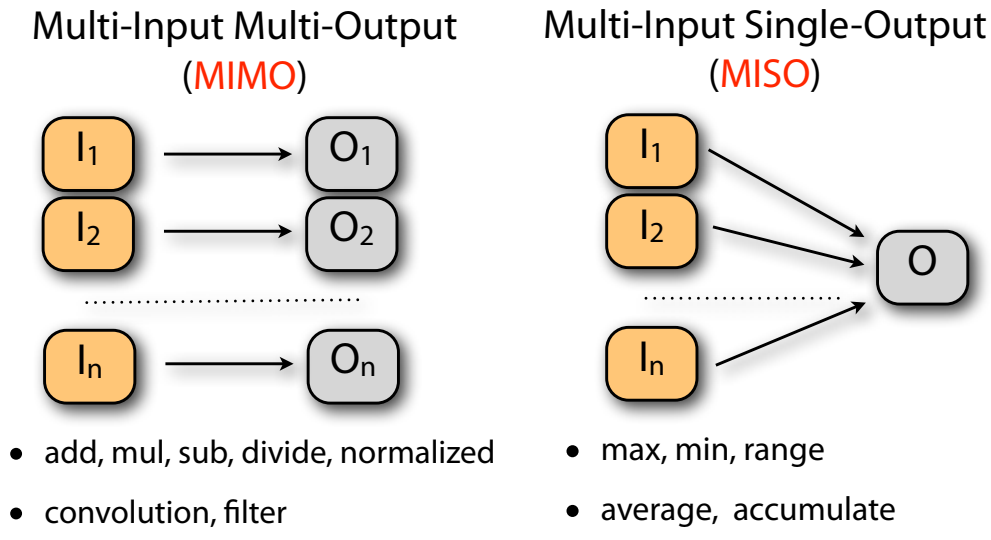


Figure 4.2. Basic multi-image operators

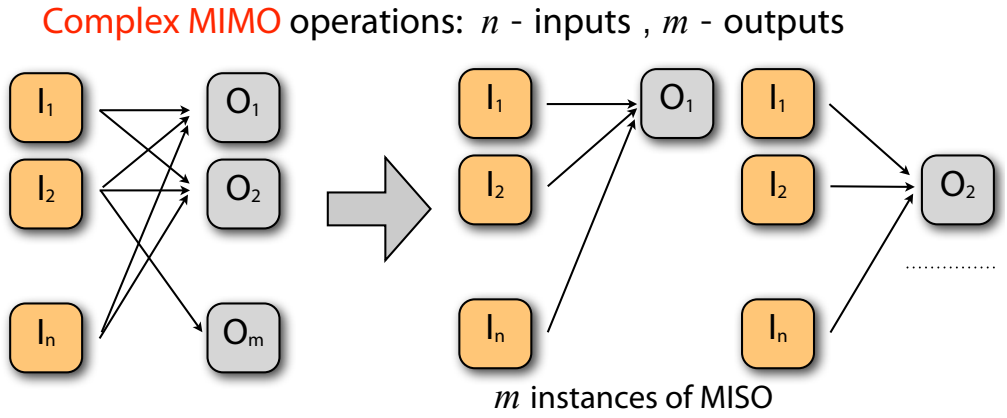


Figure 4.3. General MIMO operators

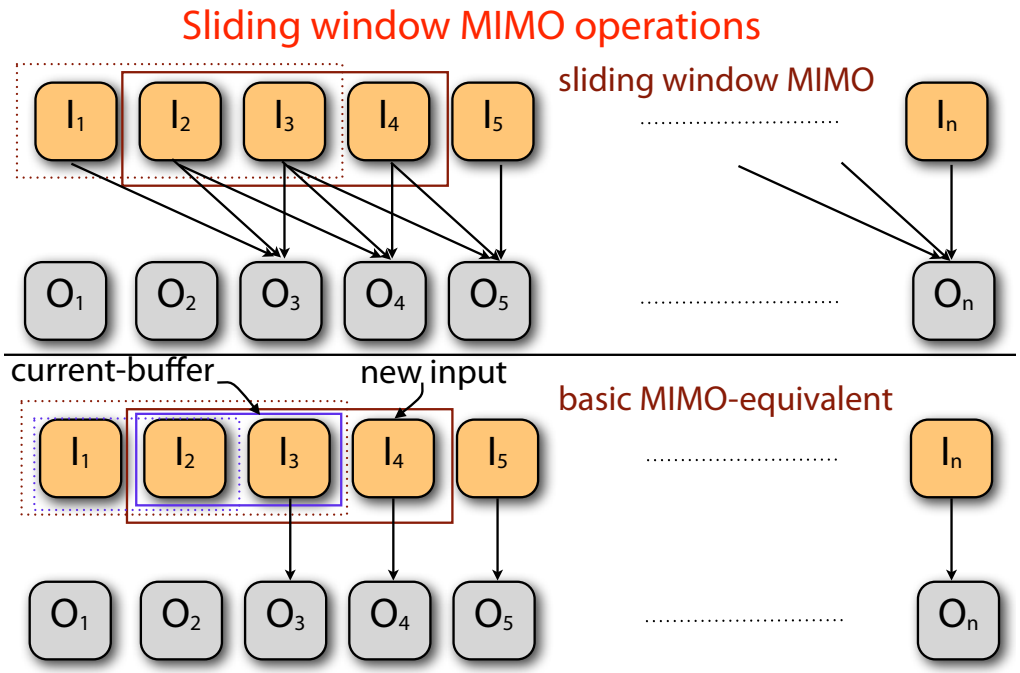


Figure 4.4. Sliding window MIMO operators

sliding window of the input. This window moves as we compute the next output image. As shown on Figure 4.4b, if we keep an input buffer with the size of the sliding window, as the window moves, we need to replace an entry of the window with the new input data. In other words, the computation of a current output requires only a single input. Algorithmically, it is equivalent to the basic MIMO model. Overall, we can implement arbitrarily complex multi-image functions based on the basic MIMO and MISO functions.

We focus our discussion on how to efficiently implement these out-of-core operators.

Note that the framework of Ha *et al.* [54] already has support for multi-image and large data processing through the GPU-cluster implementation using MPI. It also offers a multi-GPU implementation to exploit available computing resources and to increase the amount of in-core GPU memory on a single processing node. Both approaches, however, have the limitation that they depend on the total amount of system memory. The out-of-core approach we introduce here, however, has no restrictions on data input and can process the entire 3D-image brain dataset in a PC desktop equipped with commodity GPUs. Hence, our solution is more complete and accessible to researchers and scientists.

4.4 MIP out-of-core streaming framework

We introduce a flexible out-of-core solution with two levels of streaming operations: out-of-core GPU in-core-CPU, and fully out-of-core. The former utilizes the availability of the larger CPU memory system; in some cases the CPU (but not the GPU) memory may be sufficient for the entire computation. In the latter case, the dataset does not even fit into CPU memory and the data must be transferred through two memory levels: between disks and CPU main memory, and between CPU main memory and GPUs. Figure 4.5 shows the data flow in these two streaming levels. We show that our streaming strategies could be generalized through multiple memory hierarchy levels. In the following discussion, GPUs are processing devices in the first out-of-core level; consequently, in-core memory refers to the GPU global memory while the CPU system memory plays the role

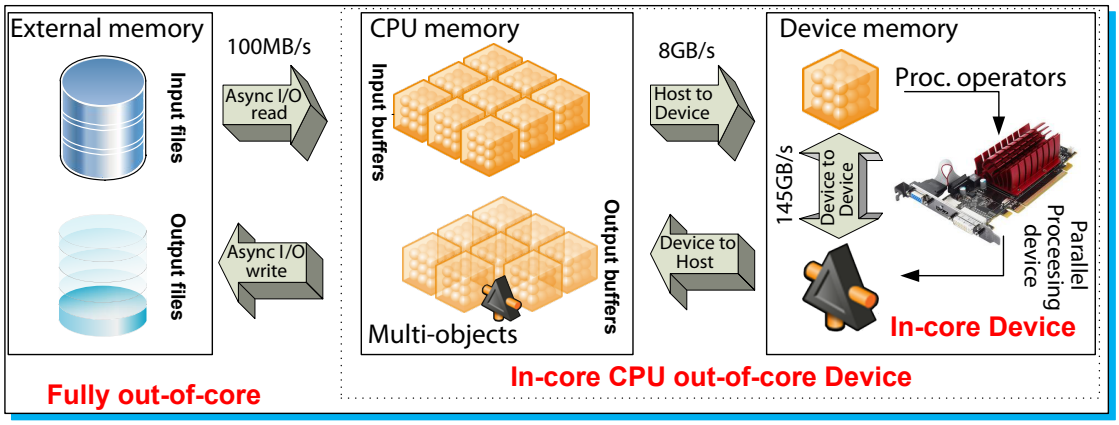


Figure 4.5. Overview of data movement in our multi-image processing multilevel out-of-core streaming framework for heterogeneous systems.

of storage devices.

4.4.1 Synchronous out-of-core model

A simple solution for out-of-core processing problems is a *synchronous model* in which the order of executions and outputs is the same as the order of functions in the source code. This requires a function to start only when all preceding functions have been completed. The advantage of the synchronous processing model is determinism: given the same sequence of inputs the same sequence of outputs is produced. In other words, the model preserves the semantic order from the code. Consequently, the system is easier to understand and debug. It is also easier to verify as there is a limited number of stages. Futhermore, this mechanism avoids any potential shared resource conflicts such as read-after-write, write-after-read, or write-after-write hazards [99]. The synchronous model deals with these resource conflicts by serializing the access to the shared resources. So at any moment, there is only one device working on the shared resources. The under-utilization of the resources is a primary shortcoming of synchronous model.

The *asynchronous model* exploits multiple execution units existing in the system; these units can run in parallel for improving the performance, in some cases significantly. However, the implementation of asynchronous models requires applications to synchronize the access to the shared resources to prevent potential hazards. Asynchronous models also increase the complexity of the application, making it harder to verify and debug. The potential performance gain is the main motivation for us to apply asynchronous processing models to build our high performance out-of-core streaming framework.

Considering the execution model at the API level, we can divide any out-of-core applications into three dominant processes: data uploading, data processing, and data downloading. In a synchronous execution model, these three steps are executed in three lock-steps: data are uploaded from the storage device to processing device, the program then runs in-core to process the data, and the results are then written back to storage media (Algorithms 6, 7). Multi-image processing allows better resource utilization using an asynchronous pipelining strategy that overlaps between the computation of one data chunk at iteration k with the data transfer of the other data chunk at iteration $k + 1$.

The transfer from a regular in-core function to a synchronous out-of-core implementation is straightforward, as we show on Algorithm 6 for MIMO operators and Algorithm 7 for MISO operators. We use these implementations as references for the correctness and performance improvement of our asynchronous implementations. We

Algorithm 6 Synchronous out-of-core MIMO operators

1: **Input** : N input images
2: **Output**: N processed output images
3: **for** $k = 1$ to N **do**
4: Upload the k-th image from the storage device to the processing device
5: Process the input in-core on the processing device
6: Download the output image back to the storage device
7: **end for**

Algorithm 7 Synchronous out-of-core MISO operators

1: **Input** : N input volumes
2: **Output**: few numbers(sum, max/min, etc) or single output image
3: **for** $k = 1$ to N **do**
4: Upload the k-th image from the storage device to the processing device
5: Process the input in-core on the precessing device
6: Update the accumulated output buffer on the processing device
7: **end for**
8: Write the final output to the storage device

compare different methods to implement out-of-core multi-image operations: an implicit model, a hardware-aware model, and a hardware-independent model. We will prove that the proposed strategies are optimal. But first, let's do some analyses on the best achievable performance of an asynchronous algorithm.

4.4.2 Asynchronous optimal performance analyses

To evaluate the performance, we use a typical hardware configuration with three components: one computational unit (GPU) and two data transfer units(one for uploading, the other for downloading data). For performance analysis, we use following notation:

- n : the number of input images
- n_s : the number of execution units
- $\tau_{i,j}$: the runtime of the i-th execution unit on the j-th input image.
- T_s, T_a : the total synchronous/asynchronous processing time
- T_u, T_e, T_d: the uploading, executing, and downloading runtime per image.
- $\mathcal{T}_i$ the total amounts of time spent by the execution unit i
- $\mathcal{T}_u = n \times T_u$,$\mathcal{T}_e = n \times T_e$,$\mathcal{T}_d = n \times T_d$: the total amounts of time spent on upload, execution and download process.
- $\mathcal{T}_{max} = max(\mathcal{T}_1, \mathcal{T}_2, \cdots \mathcal{T}_{n_s})$ the maximum amounts of time spent by a single execution unit.

Our analysis is based on the assumption that all images have similar sizes, and therefore require almost the same amount of running time. This assumption is normally satisfied with preprocessing multi-image data.

First, we determine the optimal asynchronous runtime, which we use as a reference to evaluate the efficiency of proposed implementation method. In the ideal case, all execution units run independently parallel. However, as a single execution entity, they perform tasks in sequential order. The total amounts of time that an execution unit spends is $\mathcal{T}_i = \sum_{j=1}^{n} \tau_{i,j}$ that equals $n \times \tau_i$ where τ_i runtime of i-th stream on a single-image. Since the multi-image operation is only completed when all the execution units have completed their tasks, the runtime the entire operation will be at least $\mathcal{T}_{max} = max(\mathcal{T}_1, \mathcal{T}_2, \cdots \mathcal{T}_{n_s})$ or $T_a \geq \mathcal{T}_{max} = n \times \tau_{max}$. This is the optimum runtime that the system can accomplish. Note that with the hardware configuration of upload, execution, and download units $\tau_{max} = T_{max} = max(T_u, T_e, T_d)$.

4.4.3 Implicit streaming model

The *implicit streaming model* (Algorithms 8) is solely based on data parallelism which assigns each image to a stream which works as a logical execution unit and process the entire pipeline (Figure 4.6). As streams operate on different memory spaces, the data transfer on one stream can be overlapped with processing tasks for other streams. This is in contrast to *explicit streams* (Algorithms 9, 10): *hardware-aware* and *hardware-independent* models, which depend on task parallelism. The former maps each hardware execution unit to a single stream while the latter delineates a stream to a fixed function.

Algorithm 8 Implicit pipelining MIMO operator

1: **Input** : N input volumes
2: **Output**: N processed output volumes
3: **for** $k = 1$ to N **do**
4: Load the data $iImg[k]$ from storage device to processing device, d_k on the k-th stream
5: **end for**
6: **for** $k = 1$ to N **do**
7: Apply the operator on data $d_o = oper(d_k)$ on the k-th stream
8: **end for**
9: **for** $k = 1$ to N **do**
10: Write output d_o to the storage device $oImg[k]$ on the k-th stream
11: **end for**

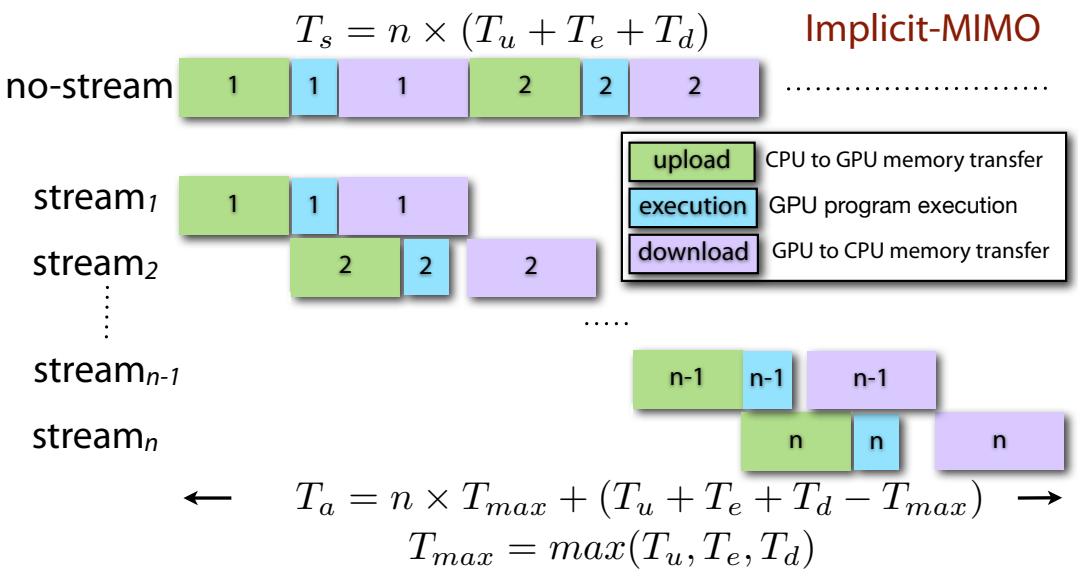


Figure 4.6. Implicit processing model for MIMOs

Algorithm 9 Explicit pipelining MIMO operator

1: **Input** : N input volumes, device input buffers $d_i[3]$ and device input buffers $d_o[3]$
2: **Output**: N processed output volumes
3: **for** $k = 1$ to $N + 2$ **do**
4: **if** $k <= N$ **then**
5: Load the data $iImg[k]$ from storage device to device buffer $d_i[k\%3]$ on the*H2D* stream
6: **end if**
7: **if** $k > 1$ and $k - 1 <= N$ **then**
8: Apply the operator on device buffer $d_o[(k-1)\%3] = oper(d_i[(k-1)\%3])$ on *D2D* stream
9: **end if**
10: **if** $k > 2$ and $k - 2 <= N$ **then**
11: Write output $d_o[(k-2)\%3]$ to the storage device $oImg[(k-2)]$ on the *D2H* stream
12: **end if**
13: **Synchronize streams**
14: **end for**

Algorithm 10 Explicit pipelining MISO operator

1: **Input** : N input volumes, device input buffers $d_i[2]$ and device input buffers $d_o[2]$
2: **Output**: single volume output or few values (max, min, sum ..)
3: **for** $k = 1$ to $N + 1$ **do**
4: **if** $k <= N$ **then**
5: Load the data $iImg[k]$ from storage device to device buffer $d_i[k\%2]$ on the *H2D* stream
6: **end if**
7: **if** $k > 1$ and $k - 1 <= N$ **then**
8: Apply the operator on device buffer $d_o[(k-1)\%2] = oper(d_i[(k-1)\%2])$ on *D2D* stream
9: **end if**
10: Store/Accumulate result on processing device
11: **Synchronize streams**
12: **end for**

Figure 4.6 illustrates the execution of an implicit streaming model for a MIMO problem (Algorithm 8). It can be seen that with the number of images being significantly larger than the number of streams, the overall processing time is approximately $n \times t_{max}$ which is the optimal runtime of asynchronous processing.

4.4.4 Hardware-aware streaming model

The execution of the hardware-aware processing model for MIMO problems is illustrated in Figure 4.7. In this model, there are three streams mapping to three execution devices. Timing analysis of the method shows that the processing time in this case is also optimal. However, it requires developers to have prior information about the architecture

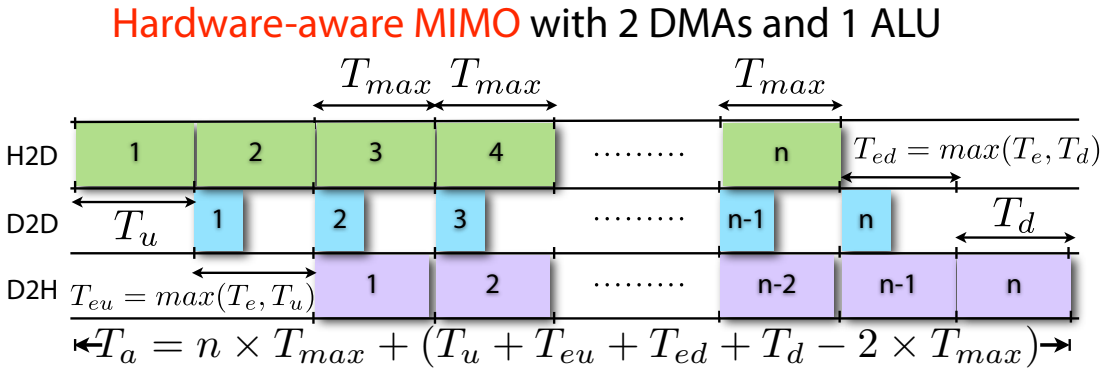


Figure 4.7. Pipeline explicit processing model for MIMO operations

of the underlying system, because the hardware-aware model reflects the actual execution of the asynchronous processes in the system. That is, it requires different implementations on different hardware.

4.4.5 Hardware-independent streaming model

The last processing strategy, the hardware-independent model, is a generalization of the hardware-aware model. Instead of decomposing tasks based on actual hardware configuration, we assume that there exists one special execution unit for every task, and we can assign each task a single stream. In the case of MIMO operations, there are three primary tasks to apply to each image: data upload, processing, and data download. On a system with two data transfer units and one processing unit, it results in a streaming scheme similar to hardware-aware models; consequently, this model also achieves the optimal runtime.

Normally, however, there are more tasks than the actual number of execution units. In this case it is possible that several tasks are mapped to the same execution unit, for example, data uploading and downloading will map to the same unit in a single-data-unit system. The question is how efficient it is when it incorrectly predicts the underlying systems, in particular, when there are multiple streams sharing the same execution unit.

Data independence results in no performance loss, as the system can very quickly switch between one task and the other. This function is done automatically as sharing info is available only at the system level. Figure 4.8 shows the runtime analysis of an optimal solution for MIMO operation on a system with one DMA and one ALU using the hardware-aware and hardware-independent implementation. The result shows that although the hardware-independent model incorrectly predicts the underlying execution system, it still performs optimally.

4.4.6 Discussion on streaming modes

The primary advantage of the implicit approach is that developers are relieved from the burden of asynchronous scheduling. Furthermore, the stream has the same execution flow as processing a single-image, no further change is required, and no synchronization is needed since each stream works on different data. However, it has several disadvantages:

- The method does not reduce the memory usage and all the data must be loaded in-core. Hence, this method cannot be used for out-of-core processing.

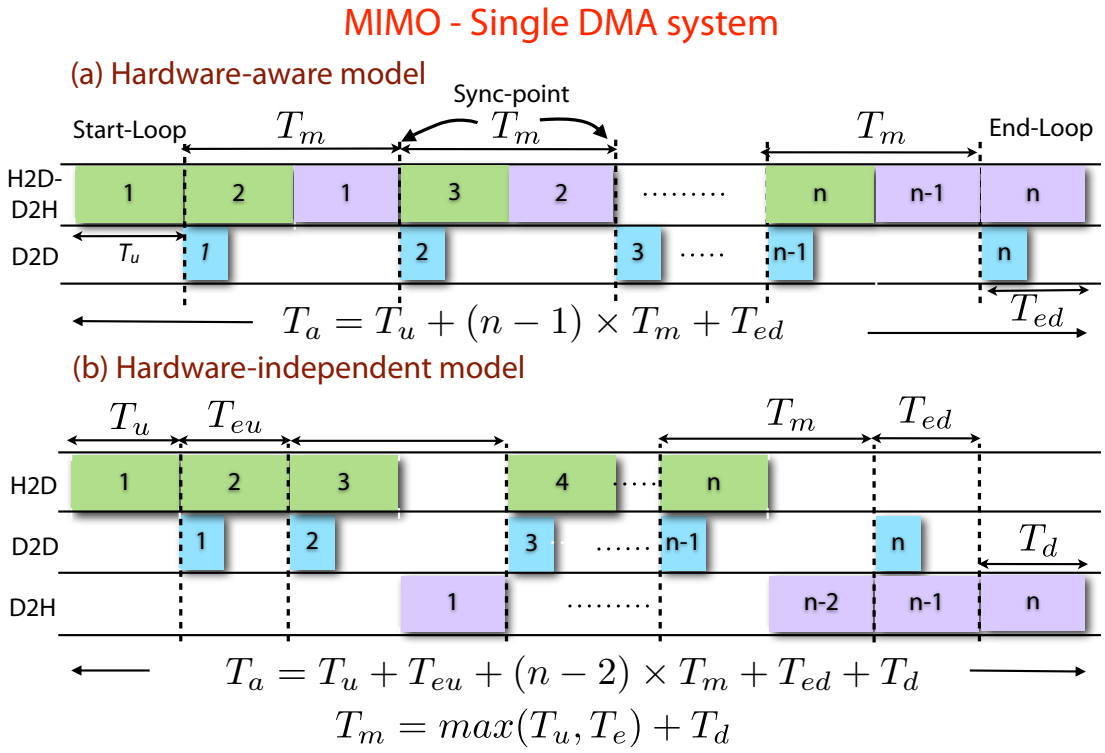


Figure 4.8. Although the hardware-independent model miss-predicts the system configuration, the performance is still optimal

- It requires the capability of decomposing input data and combining output results, which is not always satisfied.
- Although automatic scheduling hides executions from developers, understanding the physical execution is essential to profile the performance and to estimate the benefit of the method. This estimation is an important factor for making optimization decisions.
- The performance efficiency of the implicit streaming model is largely dependent on the scheduling algorithm used by the operating system or the concurrent controller. In fact, the optimal scheduling problem is NP-hard. This explains why, in practice, this approach does not always provide the predicted optimal performance.
- The implicit model has an order-dependency that limits the execution of the streams. Particularly, all streams execute in the same order of the logical flow: uploading-processing-downloading. However, flexible reordering is an effective strategy to handle degenerate cases, including synchronous functions calls.

Most of the weaknesses of the implicit model can be handled by explicit approaches.

- Explicit methods require a much lower memory footprint, which is equal to the number of hardware devices with the hardware-aware model or number of decomposed tasks with the hardware-independent model. That means they are suitable for out-of-core processing.
- As it is always possible to divide an out-of-core algorithm into three primary tasks, it is easier to decompose tasks than partition data.
- The explicit method uses an explicit scheduler. That means the execution is controlled, providing several benefits. First, developers can estimate the performance before they actually run it. Second, it reduces the complexity of the scheduling problem to a trivial mapping, so it is even optimal without any automatic scheduler supports. Finally, it helps to understand why degeneracy happens, how it affects the performance, and how to deal with it.

4.5 Reordering stages in streaming models

The aforementioned approaches are simple and theoretically optimal. They are straightforward to transfer from single-image processing to multi-image processing through the generalization of basic multi-image operators. However, the optimal performance is hardly achieved in practice, the primary reason for this being the streaming degeneracy.

To maximize the benefit of the asynchronous processing model, it is necessary that all functions run in an asynchronous mode. Though synchronization is necessary to coordinate between concurrent tasks and to resolve resource conflicts, the use of synchronous functions should be avoided, if possible. To maintain the semantic order of the source code, a synchronous call will block until all the preceding functions, even asynchronous ones, have been completed and it causes the subsequent functions to wait until its completion. This breaks the flow of asynchronous pipelines. It reduces the effectiveness of pipelining models, causing degeneracy in streaming code.

4.5.1 Forced synchronizations

There are three primary variations of degeneracies that may appear in streaming models

- Synchronous function calls
- Asynchronous stream mismatches

- Cross-stream function calls

The most common reason for an unintended synchronous function call is that the application requires an external call to a library function that was designed for synchronization execution. Another reason is the mixed use of synchronous and asynchronous functions.

Even when all functions support asynchronous execution, they might be designed using different schemes. The strategies are often incompatible and cannot work together efficiently. For example, a kernel function defined to run on a logical stream is incapable of running in parallel with a data-transfer function on the physical stream with the same identity. These functions frequently require explicit synchronization to switch between the different asynchronous modes.

Cross-stream calls occur when the implementation requires data access and computation to or from different streams. As a result, the compiler forces these streams to synchronize at cross-reference points to preserve the semantic order of the original program. One example is the traditional implementation of the class of reduction functions in CUDA. Though the computations run in-core on GPU-devices, the output of these functions, which are typically used for branching on a CPU host, require the result to be copied from device memory to host memory. This operation is a cross-stream function between the computational stream on the devices and the data transfer stream between a device and its host. The popularity of the reduction functions is the main obstacle for applying asynchronous models on existing GPU architectures. Our solution for the reduction-like function is an on-device model that outputs the result only to device memory. It requires subsequent functions to use on-device parameters, and to delay or remove the branching in the codes.

4.5.2 Reordering pipeline stages

In many cases, when a forced synchronization is unavoidable, though negative effects can be minimized using a reordering technique. This out-of-order execution is applied in modern compilers to reduce the number of mis-predicted branches, to avoid data spilling, to keep the instruction pipelines filled, and especially to allow parallel execution on a system of multiprocessors.

In this case of streaming with degeneracy, the reordering optimization cannot be done automatically using the compiler. The reason is that the uploading and downloading

are IO processes which have side effects. This constrains the order of function execution and requires the compiler-generated code to execute in the same order as it appears in the API levels. Even worse, the forced synchronous functions impose a restriction in the order of the outputs. So reordering without compiler support needs to be done explicitly.

Allowing different streams working on independent images allows our explicit models to break the order-execution dependency inside the loop, replacing it with an equivalent order-independent streaming model. As shown in Figure 4.9, the order dependency of the original loop is still preserved in the order of loop execution. In other words, the logical correctness of the processing model is guaranteed by construction.

As the order of streams inside a loop becomes unimportant, we can change the order of streams at the API level from the regular order of upload-process-download to upload-download-process, or process-upload-download. The ability to change the processing order allows streaming optimization. This optimization is particularly effective when asynchronous stream degeneracy is unavoidable.

In the implicit model, when the synchronizations exists in the execution process, it is unable to overlap the uploading and downloading stream as the uploading process has to finish before the synchronization points, while the downloading only happens after the synchronization points. As shown on Figure 4.10, changing the order of streams in code

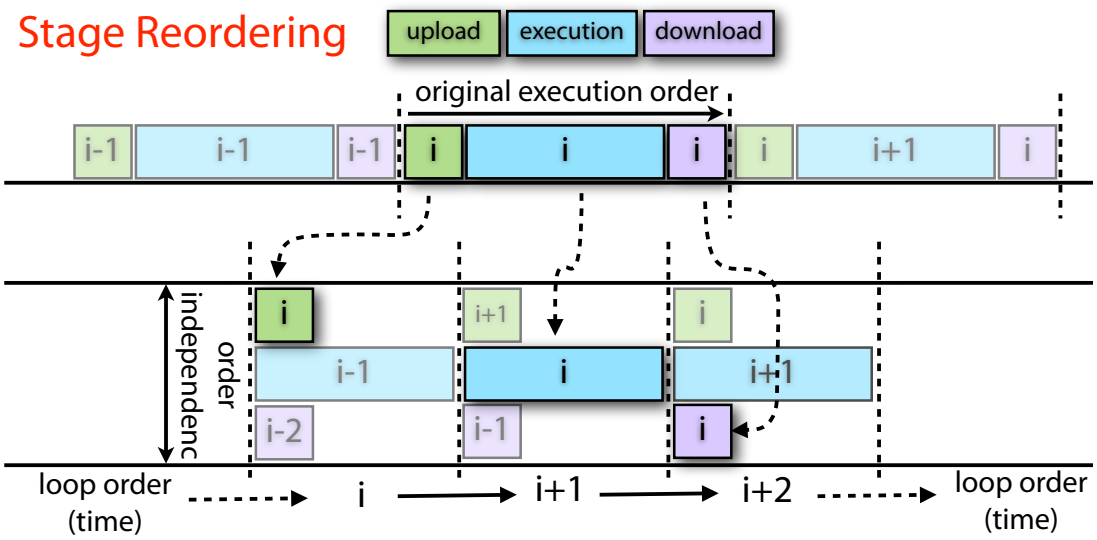


Figure 4.9. The transformation from a synchronous model to an explicit streaming model preserves semantic correctness.

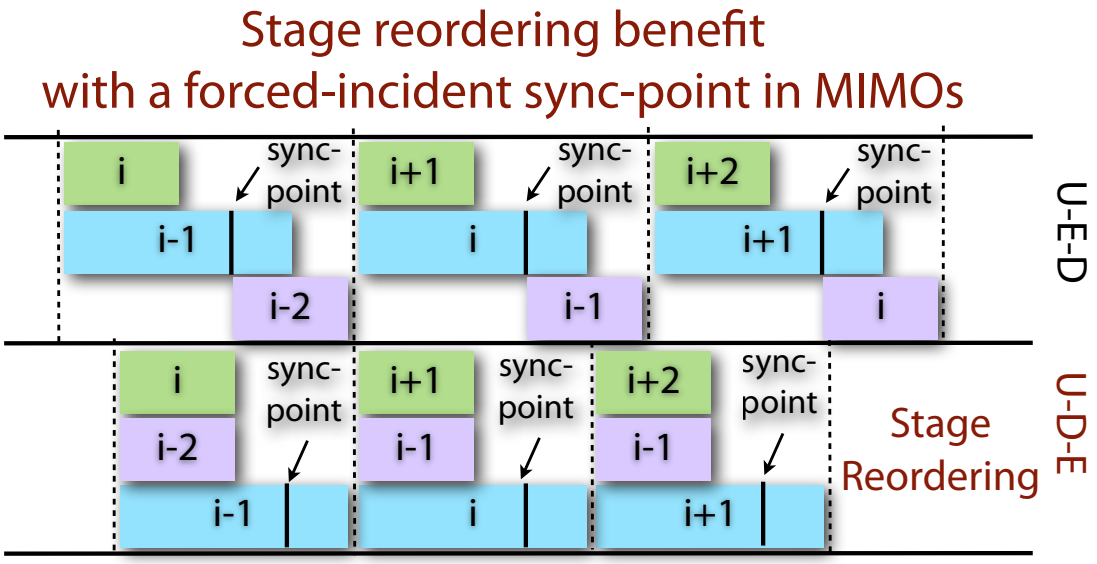


Figure 4.10. Streaming optimization using reordering technique. As shown on the figure it is able to eliminate the negative effect of forced-synchronous function

using the explicit model allows the upload and download stream to be fully overlapped even when a synchronization point is present. Thus, reordering helps reduce the run-time per iteration as well as the overall run-time. The ability to semantically reorder the stream execution in the code allows us to adapt a performance heuristic that profiles the performance and selects the optimal order.

4.6 Extension to a full out-of-core framework

The extension from the partial out-of-core model with one level of memory hierarchy to a full out-of-core model with two memory levels comes naturally with the hardware-independent model. By adding two more stages to the algorithm decomposition—the upload from disk to main memory and download from main memory to disk—we realize the transition to a fully out-of-core model. The execution of this model for MIMO operation is displayed on Figure 4.11.

Using the same logic as the partial out-of-core model, we can prove that the hardware-independent model for out-of-core processing is optimal. Note that we use the term "full" to mean that the data could be stored on the storage media of a single machine. However, our hardware-independent model could be further extended to other out-of-core models, such as data streaming on a network and a system with more memory hierarchy levels,

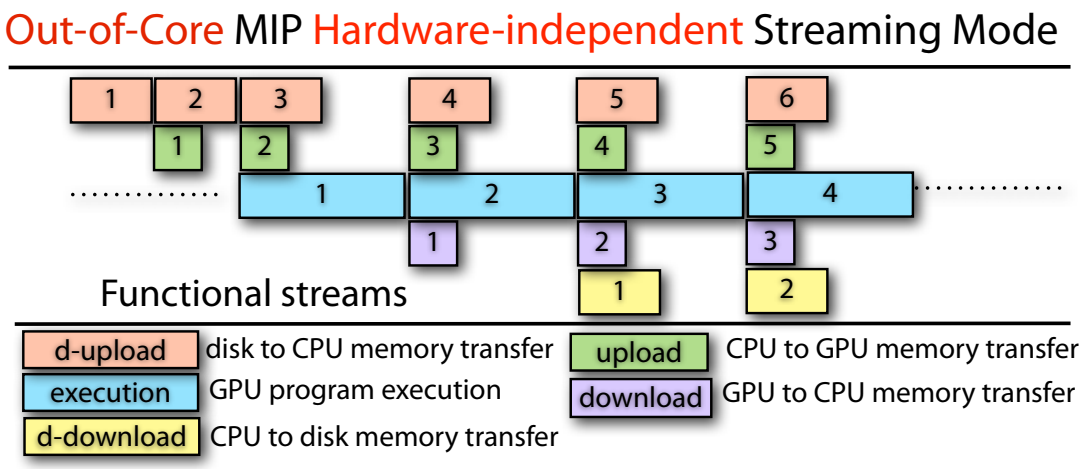


Figure 4.11. The implementation of hardware-independent model for "full" out-of-core multi-image processing

and one could still prove that the proposed models are optimal.

4.7 Results

The system we used in our experiment is a PC desktop, Intel Core i7-980X, 12-GB DDR3 1600, with a single NVIDIA GTX 480. Communication from the host to GPU is via the external x16 PCIe bus and is controlled by a single DMA. The program is compiled with CUDA NVCC 3.1. Run-time of each function is measured in milliseconds.

We made a synthetic test on a data set of 32 volumes, sized $256 \times 256 \times 256$. The test mimics a typical out-of-core multi-image processing program using three processes: upload, execution, and download. Note that the execution time and data transfer times scale proportionally to the number of images and the sizes of the image. We also achieve similar performance curves with different number of images ranging from 10 to 180 (the maximum number of volumes we can fit onto the 12GB of available memory).

The existing architecture on commodity hardware has a single DMA unit, so the upload and download process has to be performed sequentially. This architecture allows a two-device, hardware-aware model with only two memory buffers. There are two options for its implementation: (1) the upload of the k-th volume in parallel with the execution and the download of $k-1$-th volume (U_ED); (2) the upload and execution of the k-th volume in parallel with the download of $k-1$-th volume (UE_D). Our hardware-independent model still decomposes the algorithm into three processes regardless of the

system configuration. There are six permutations for the implementation of the hardware independent model. However, here we report the performance for three permutations: (1) regular upload-execution-download (UED) (2) execution-download-upload (EDU) (3) download-upload-execution (DUE).

4.7.1 Full asynchronous processing

First, we perform our test using the ideal cases, fully asynchronous processing function, without a single synchronous call in the execution. Here we measure the influence of the ratio betwen computation and data transfer (processing ratio) on the performance of different asynchronous processing models, denoted $r_e = E/(U + D)$. This ratio indicates different types of out-of-core functions: data-transfer dominance ($r << 1$), processing dominance ($r >> 1$), and balanced functions ($r \approx 1$). In the ideal case, the results on Figure 4.12 show:

- In all the tests, the three hardware independent implementations give us the same performance. The hardware-aware and implicit models give similar runtimes. The

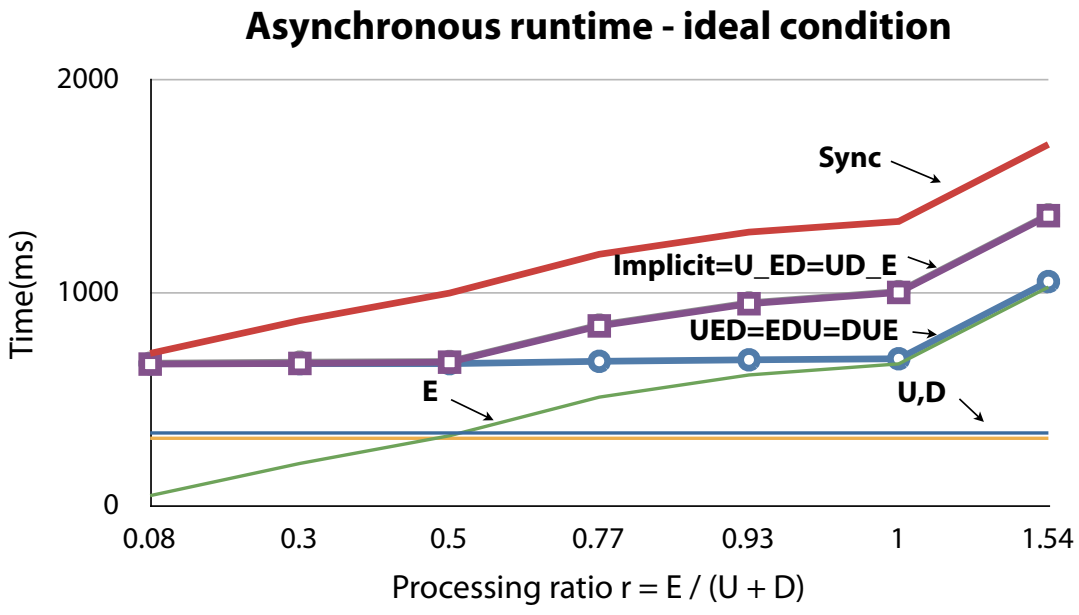


Figure 4.12. Runtime comparison of different streaming strategies in ideal conditions. All the permutation of explicit model yield the same performance. The hardware-independent models achieve the optimal performance.

U_ED is slightly faster than UE_D since the upload takes a bit longer than the download.

- If the function is transfer-dominant ($r_e < 0.5$), all the models give optimal solutions.
- When the execution time is larger than the upload or the downloading time, the first two models still give strong performance, approximately $T_u + T_e$. However, it is not the optimal of $max(T_u + T_d, T_e)$ achieved with the hardware-independent model.
- When the function is balanced or processing-dominant ($r_e \geq 1$), the hardware-independent model gives the optimal runtime T_e and the data transfer is completely hidden.
- The asynchronous function gives the best speedup in comparison to the synchronous models when the loads between two execution units are balanced ($r_e = 1$).

4.7.2 Synchronous functions

Second, we test the result with the use of a synchronous function. Here we fix the run-time of the three basic processes but change the position of the synchronous function inside the execution process to measure the influence of sync points inside the functions. We vary the synchonous ratio $r_s = E1/(E1+E2)$. With the existence of the synchronous function, the results in Figure 4.13 show:

- The position of the sync point within the asynchronous code directly affects the performance of the given implementations.
- The three hardware-independent implementations give us different performance characteristics. No single hardware-inpendent implementation gives us the best running time overall. However, the best result always is achieved with one of the hardware-independent implementations.
- The implicit model no longer gives us the optimal result, and is as slow as the synchronous implementation. It simply cannot find a schedule for asynchronous execution.
- The hardware-aware model could not give us optimal results in all the tests. However, it is still far better than the implicit model. Note that their two implementations also give different runtimes.

Though we show the results with execution-dominant function here, we also draw the same conclusions from transfer-dominant and balanced functions.

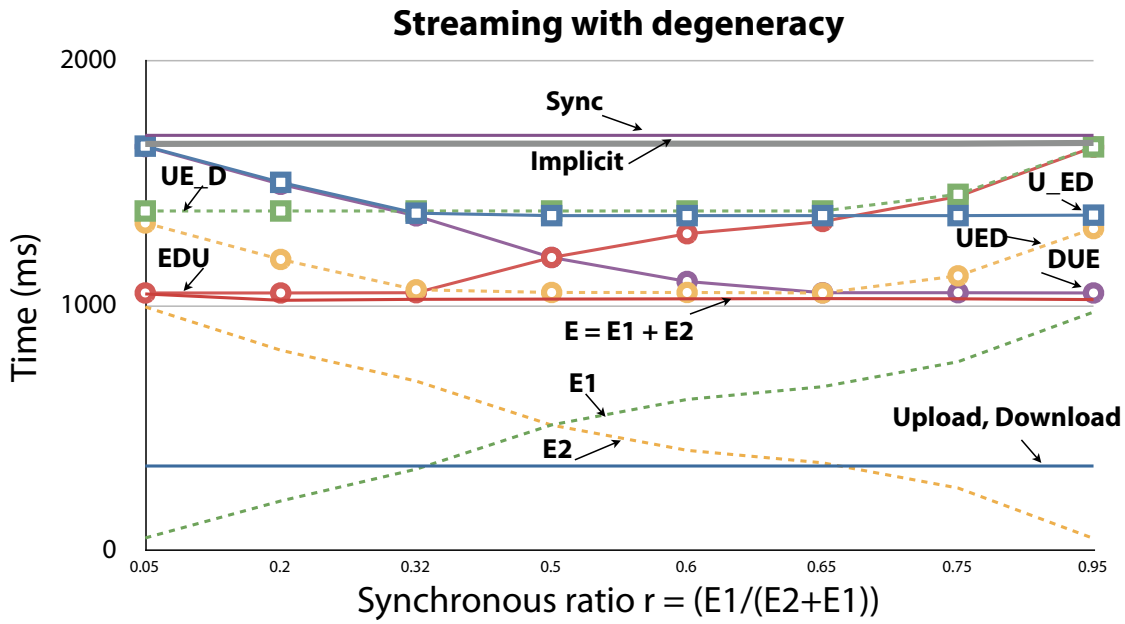


Figure 4.13. Runtime comparison of different streaming strategies in degenerate conditions

4.7.3 Regular out-of-core functions

On the third experiment, we focus on the regular out-of-core function sets such as a maximum value of all images, normalization, averaging, Gaussian filtering, product (energy computation), and atlas building. The results from Table 4.1 confirm that when the computation only requires simple functions (max, product, normalization, averaging, etc.), the asynchronous streaming does give you the benefit of hiding the computational cost. However, it is negligible in comparison to the transfer cost. As the complexity of the functions increases (for example, Gaussian filtering function), we start seeing significant

Table 4.1. Runtime comparison of regular functions with different streaming strategies

Function	U	E	D	Sync	Impl	Hrd-aware	Hrd-indp
Max	347	13	0	360	349	349	349
Energy	692	20	0	710	698	700	698
Averaging	347	20	11	378	360	363	361
Normalization	347	28	322	694	696	687	677
Gaussian	347	431	322	1099	735	770	678
Atlas	201446	213423	1359583	555204	NA	372567	340356

benefits of asynchronous streaming strategies, especially with the hardware independent model.

In atlas construction, which is performed on the ADNI dataset (Figure 4.14), as we increase the complexity of computational functions and reduce the cost of data transfer by merging all the functions together on a single loop, we yield significant performance improve over the synchronous out-of-core version. The performance is compared to the in-core performance (execution time only) even though we could process a significant larger amount of data than that of an in-core version.

Overall, our results confirm our theoretical analysis. All the strategies are able to achieve optimal performance. However, only the hardware-independent model gives the best performance in all the tests. In the degenerate cases, the implicit model completely fails. The presence of synchronization points makes it impossible to find an efficient schedule automatically. Note that in this case–a greedy approach—which immediately executes whenever the resource is available—also fails. The hardware-aware model gives better performance even with the degenerate cases, although it is optimal. It is always possible to find the best runtime between hardware-independent implementations. In other words, the optimal performance is always achievable with the hardware-independent model.

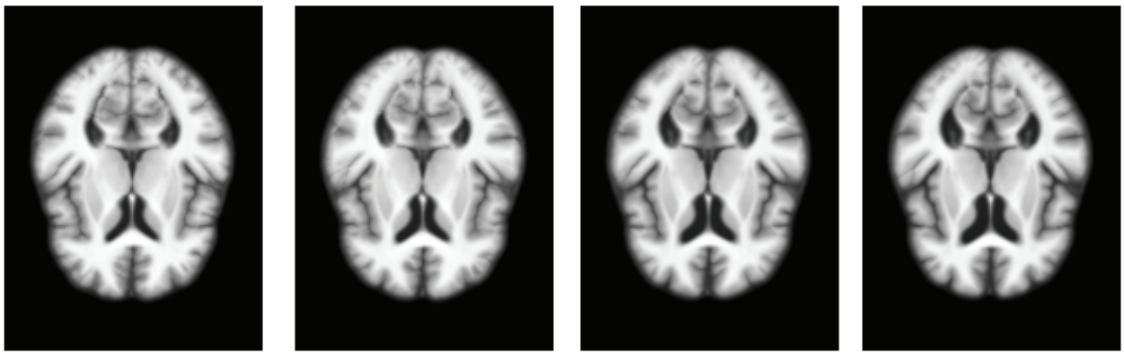


Figure 4.14. Age regression anlysis on the ADNI dataset by computing the average brain atlases at different ages (65, 70, 75, and 80) corroborates the hypothesis that fluid space is larger because brains atrophy overtime. This analysis, however, could only be performed if the system is capable of processing the whole dataset of 300 healthy brain-images

4.8 Conclusions

In this chapter, we have presented an optimized, parallel, multi-image processing framework for heterogeneous commodity systems extending from an existing single-image, parallel processing framework. We have introduced multi-image operators, serving as the connection between the single-image processing model and the multi-image processing variant. We proposed two basic multi-image operators: the MIMO and the MISO, which are utilized to construct other multi-image operators, allowing us to build a complete multi-image processing framework. We have also presented optimal streaming models for the multi-image processing framework. We have analyzed the advantages and disadvantages of various streaming strategies, and proposed a generalized streaming model based on functional decomposition that is optimal, hardware-independent, and highly scalable on future hardware. Our experimental results show that our hardware-independent model adapts to underlying hardware configurations, out-performs other streaming strategies, and gives optimal performance in all tests.

We also evaluated the efficiency of streaming models, and presented a quantitative evaluation that serves as a model for developers. We have investigated an optimal streaming strategy in unfavorable conditions based on reordering from order-independent properties of the explicit-streaming models. We also give insight to the causes of unfavorable streaming conditions that help developers locate the performance degradation points in their implementations. Though we use a GPU computational model to illustrate the efficiency, our framework makes no specific assumptions about the underlying architecture and hence can be generalized to any heterogeneous parallel processing system.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

The dissertation has introduced a high-performance image processing framework to harness parallel processing power of modern GPUs for computational challenging tasks. In particular, this framework provides atlas construction algorithms on different GPU hardware configurations: single-GPU desktops, multi-GPU workstations, and GPU clusters. The approach allows significant performance benefits in multiorder of magnitudes to be achieved. The implementation is described and evaluated to demonstrate practical applications of the tools developed.

In the second chapter, we introduced essential elements of an image processing framework on GPUs. We also proposed a multiscale approach that improves the quality of processing methods as well as reduces the processing time of advance image processing techniques. We developed our framework using generic programing feature of C++ to provide a great flexibility for the system's users to customize with their own implementations and to apply the framework to solve computational problems.

In the third chapter, we proposed the idea of using a multicompartment presentation of an anatomy to perform the computation on different domains so that they complement the others to provide a solution for registration challenging tasks. The computational strategies that we used in this chapter are not only the key performance of the method. More important, the method serves as a bridge for the high performance computation on irregular domains, which can be employed for many other mesh-based computations as well.

The fourth chapter shows another computational perspective of the atlas construction. As we approach it from a different view point, it requires a different methodology to solve the problem. By abstracting it as a multi-image processing problem we come up with an optimal out-of-core streaming framework for multi-image processing. In addition, the chapter presents a simple transforming strategy that we prove to give optimal results for out-of-core processing problems. We also analyze the reasons for the degraded per-

formance which regularly happens in practice. We believe these reasons are complete. That is, they could be used as performance checking constraints in an asynchronous implementation to assure that performance goals are reachable. Our reordering strategy is an effective method to deal with unavoidable synchronisation at APIs levels. It can improve the performance significantly while requiring insignificant code modification. Further more, this optimization can be done automatically at APIs level.

Appendix A presents a sorting implementation which is fast and efficient on current hardware. In this Appendix, we analyze the performance bottleneck of existing approaches. Our optimization is based on our revisited concepts of arithmetic intensity, which better reflect the target of an optimization process. This concept is not only useful for specific sorting problems but can be applied for the other optimization challenges such as a prefix scan, segmented sorting, sparse matrix vector multiplication. We also proposed a hybrid data structure that can achieve significantly higher bandwidth efficiency when coalesced condition is not satisfied. The key idea is to use proper data structure for each algorithmic stage. We can improve the performance further than what we can achieve with reducing the computational complexity solely.

Overall, the dissertation is a complete work on building a high performance framework on GPUs. We have addressed different aspects of the problems from different perspectives, from baseline research to implementation challenges. For future work, I would like to extend the current framework to a more general solution for not only image processing tasks but also visualization, particle simulation and mesh processing (in which we have achieved some encouraging results). With this extension, we want to attack scalability problems and out-of-core processing problems in general, for which we believe there exist optimal solutions in many different cases.

Though diffeomorphic registration framework is robust and mathemetically well behaved, it requires all the registration objects to be diffeormorphic to each other. However, this constraint is not always satisfiable in practice. While we could generally assume that healthy brains are diffeomorphic, a damaged brain is certainly not. We see multicompartment models as one step to a more general model that only requires a compartment to be diffeomorphic to its correspondent partner. We will further investigate in this direction to broaden the application field of our framework. We can clearly see the potential of the framework in 3D automatic volume warping, 3D animation, damaged brain registration and many other time-critical applications.

APPENDIX A

PARALLEL GPU SORTING

In this appendix, we present a high performance sorting function on GPUs that is able to exploit the parallel processing power and memory bandwidth of modern GPUs to sort large quantities of data at a very high speed. We revisit the traditional radix sorting framework, analyze the weaknesses, and then propose a solution based on the implicit counting data presentation and its associated operations. We also improve the bandwidth utilization with our hybrid data structure and redefine the concept of arithmetic intensity as a guidance for GPU optimization process.

A.1 Introduction, problem statement and context

A.1.1 Motivation

Sorting is undeniably one of the most fundamental algorithmic building blocks and one of the most widely-studied problem in computer science literature. There are numerous algorithms in which sorting is an essential component. Thus our algorithm can significantly improve the performance of many applications, such as data querying, exploration, classification, visualization, physical-based simulation, computer games. Hence, the results of this work are of interest for general research and development in HPC and GPGPU communities.

Modern GPUs offer massive parallel computational power and extreme memory bandwidth, the foundations for fast sorting algorithms. Previous GPU sorting approaches, however, were not able to exploit these advantages. In particular, scattered write operations prevent coalesced data movement, a key component for efficient GPU programming. Consequently, GPU sorters were memory bound with low compute-memory efficiency. In this chapter, we analyze these issues and propose two major improvements: First, an implicit counting structure with associated operations, and second a hybrid Structure of Arrays (SoA) and Array of Structures (AoS) data presentation.

A.1.2 GPU sorting overview

The dramatic changes of GPU architectures over the past decade have a big influence on both comparison-based and counting-based GPU sorting algorithms.

A.1.2.1 Comparison-based sorters

Most traditional GPU sorting implementations have been based on sorting networks, in particular the bitonic sorting network. The main idea is that a given network configuration will sort the data in a fixed number of steps using static communication paths. This property suits the traditional GPU architectures well, because sorting algorithms can be expressed in terms of shader functions, which have very limited branching and no scattering support. The complexity of such sorting networks, however, is $0(n \log_n^2)$, which is higher than that of the optimal comparison-based sorting, $0(n \log_n)$.

The complexity drawback was tackled by Gres *et al.* [52], who employed an adaptive bitonic sorting strategy to lower the complexity to the optimal bound of $O(n \log n)$. Cache strategies were also considered to improve the performance; Govindaraju *et al.* [49] presented an improved bitonic sorting network with more cache-efficient data access and data layout to speed up GPU based sorting by about a factor of 1.5.

The introduction of general parallel processing architectures and high level GPU programming languages such as CUDA, OpenCL gave developers full access to the computational power and memory bandwidth of modern GPUs. These programming features offered developers more control of the memory cache, parallel thread execution, and efficient branching with fine-grain hierarchical memory-execution structure.

Peters *et al.* [100] implemented a fast bitonic sorting algorithm in CUDA which reached 60M pairs per second on the GTX 280. A competitive performance is achieved by the parallel merge sort of Satish *et al.* [106], which became part of the Thrust library[62]. So far the fastest comparison-based sorter, however, is the GPU Sample Sort by Leischner *et al.* [74], which is about 30 percent faster than the parallel merge sort.

Despite achieving considerable improvement over CPU-based sorters, the log-factor of comparison-based approaches is costly, especially when dealing with a large number of inputs. Comparison-based sorters are only considered when inputs are noninteger or have variable length, and when in-placed sorting is the main concern. Otherwise, a more efficient approach is the counting-based sorting scheme with a linear bound complexity.

A.1.2.2 Counting sorters

Though counting-based sorters were introduced later to the GPU, they have achieved remarkable performance improvement and have proved to be the more GPU friendly and scalable approaches. In 2007, the hybrid sorting algorithm by Sintorn and Assarsson [109] based on a vectorized mergesort in combination with a bucket-sort using atomic GPU operations, was twice as fast as the previous fastest GPU-based bitonic sorting algorithm [49]. The most efficient GPU counting-based scheme, however, is the radix sorting. The GPU radix-16 by Satish *et al.* [106] is the first single-device sorter that is capable of sorting more than a hundred million key-index pairs in a second.

Radix-sorting algorithm is often referred as *radix-r*, where r is the number of radix buckets. In practice, the key is 32-bit length, hence it requires $[32/log_2(r)]$ passes, each pass performs a radix step on $\log_2(r)$-bit of the key from the right most bit to the left most bit (Least Significant Bits strategy - LSB). The radix sorting can be used for arbitrary number-typed inputs: float and integer, and with arbitrary key-length [55].

In a single pass, each key is placed into one of r buckets. The position of the r-sorted output element, called *global rank*, is equal to the total number of elements in lower buckets and those preceding in the same bucket. For parallel efficiency, the global rank is computed using a fine-grain approach by adding a *local count* (the order of the number on the radix inside its block) with the number of the same radix value on previous blocks, then with the total number of elements in lower radix buckets, as illustrated on Figure A.1. When the global ranks are computed for all input elements, the final step shuffles inputs onto locations determined by their ranks. Then the attention is moved to the next higher

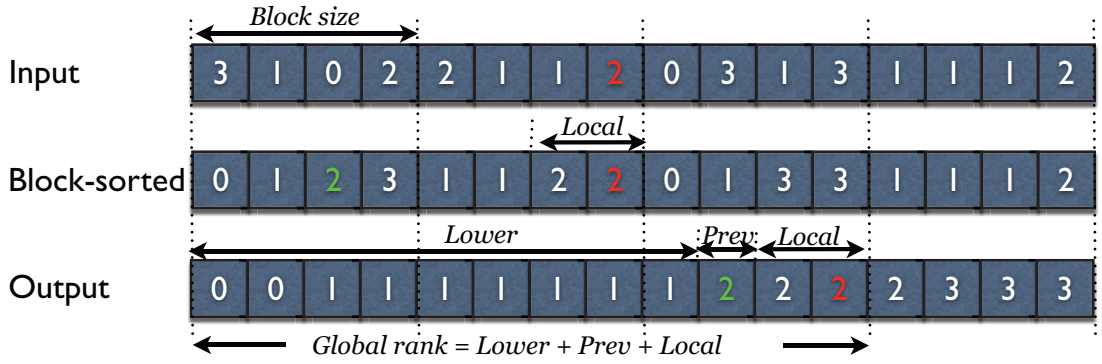


Figure A.1. Global ranking computation for block radix sorting

bit group and the process continues until all the input bits are sorted.

The performance of GPU radix-sorting depends on how fast the global ranking computation is and how cache-friendly the shuffle step can be implemented. There are two main schemes to compute global rank: histogram-based methods [109, 50] and scan-based methods [55, 106].

Histogram-based methods explicitly compute a histogram for all radix buckets. Sintorn and Assarsson [109] exploited CUDA atomic functions on CUDA 1.1 hardware to count the number of elements in each bucket. Therefore, their performance depends heavily on the input distribution, and suffers from parallel resource fighting. To tackle this drawback, Le Grand [50] exploited the on-chip fast-access explicit cache(*shared memory*) for radix counters, and divided parallel threads onto thread groups. Each thread group has different radix counters; hence, resource fighting between groups was eliminated. However, the method serializes the increment of radix counters sharing between threads of the same group.

Scan-based methods depend on prefix sum operation to implicitly compute the histogram. First presented by Harris *et al.* [60], the GPU scan operator can achieve optimal bandwidth of streaming operations on the GPUs. As a direct result, Sengupta *et al.* [108] implemented a binary-radix sorting which requires n radix passes with n being the key-length in bits. The method is bandwidth-bound and under-utilized GPU power, resulting in similar performance as the hybrid sort but slower than Le Grand's radix-16.

To exploit the parallel processing power of the GPUs and to reduce the number of radix passes, Ha *et al.* [55] proposed a fast 4-way radix sorting that took advantages of the instructional parallelism to perform four scan counting paths at the same time. Satish *et al.* [106] exploits the simplicity and efficiency of the implicit binary-radix sorting to perform multiple radix passes on the GPU's shared memory. Both methods were based on a modified radix sorting with a local presorting step to handle the noncoalesced pattern of the final mapping step. As a result, Satish's radix sorting is almost six-times faster than Le Grand's radix-16 with the capability to sort 140 millions input pairs per second on the NVIDIA GTX 280.

A.1.2.3 A recent breakthrough in radix sorting

Satish's radix sorting is well-known as the fastest published results for both GPU and CPU sorting on a single desktop preceding to our work. However, a recent work by Duane Merrill and Andrew Grimshaw [81], which was presented at IPDPS 2010 right after our

submission was accepted to GPU Computing Gems Volume II, proposed a new radix sorting approach that achieved 482 key-value pairs per second on GT200-based model, that is 3.7 times faster than Satish radix sorting. The method is based on the their new multiscan technique [80] which is twice faster than CUDPP scan implementation from which our sorting framework and Satish's were based on. The authors also presented the *visiting logic*—a new optimization technique–to improve the system utilization. These techniques are orthogonal to the techniques that we presented in this chapter, and hence it is likely that a combination between our method and Duane Merrill and Andrew Grimshaw's work could yield a faster sorting result. Since the main contribution of our work is to improve directly over the Satish's work, throughout this chapter, we will only discuss and compare our result to Satish's work to highlight the key optimizations. For more discussion on a potential combination solution, see Section A.5.

A.2 Core sorting frameworks

To further improve the efficiency of ranking computation and the cache coherency of the mapping step, Satish *et al.* [106], and Ha *et al.* [55] proposed an improved framework that performs sorting in 3 main steps:

- Parallel local radix counting and presorting
- Global radix ranking
- Coalesced global shuffling

The basic difference of the improved framework from the traditional one is the local presorting step, which happens inside the shared memory and is incorporated into the regular local counting step. The presorter divides data into radix blocks, which then move together in the final mapping. This strategy greatly increases the cache coherency of the data. To further improve the performance, a coalesced mapping step was proposed [106, 55] that assigns each thread to the data based on its output location to satisfy the coalesced mapping condition.

A.2.1 Revision of the arithmetic intensity concept

An analysis of the computational characteristics of existing sorting algorithms shows that few arithmetic operations are involved, i.e., the counting with radix-based solutions and simple comparisons with other sorting solutions. Data movement is the most common

operation. Consequently, sorting algorithms are memory-bounded, low-compute efficient, and rarely able to benefit from the huge computational power of GPUs.

Though the improved framework tried to tackle the noncoalesced effect, the memory bandwidth efficiency of the global mapping step is still a fraction of the full memory bandwidth. Together with low-compute efficiency of presorting step, they are the two major performance bottlenecks. We see these issues as the problem of low arithmetic intensity of existing approaches.

There are two typical views about arithmetic intensity: Dally *et al.* [31] defined "arithmetic intensity" as "math operation per memory op", Buck *et al.* [21] defined "computational intensity" as "time spent on computation over data transfer". Though these definitions are helpful, they do not reflect the actual efficiency of a kernel and insufficiently capture the goal of optimization. We rather consider efficiency as "the overall amount of work done over the data", i.e., work per time so that a more efficient kernel will do more effective work per data unit (i.e., implicit binary vs binary sorting) and spend less time to complete the same amount of work, i.e., sorting task. Our definition considers both computational and memory usage efficiency in the optimization process.

A.2.2 Algorithmic improvements

Using this definition as guidance, we propose two major algorithmic improvements: an implicit parallel counting and a mixed-data structure. The implicit counting exploits GPU instructional parallelism to reduce the number of passes inside the shared memory by a factor of two in comparison with Satish's method. The mixed-data structure allows a more efficient mapping step which is immune to the nonideal coalesced effect. Both strategies successfully address the efficiency issues, leading to a significant improvement over the highly optimized solution of Satish. In the next section, we will discuss in detail our sorting method.

A.3 Algorithms and implementations

A.3.1 Implicit counting - Improving compute efficiency

Two major components of this arithmetic improvement are the implicit counting number and its associated operations. An implicit counting number encodes three counters in one 32-bit register, each counter is represented with 10 bits, in particular

$$impl_{cnt} = cnt_0 + (cnt_1 \ll 10) + (cnt_2 \ll 20)$$

where cnt_0, cnt_1, cnt_2 are the counting values of radix value 0,1, and 2.

For a single radix value, the corresponding implicit counting value (Figure A.2 b) is computed

$$impl_{val} = (val < 3) \ll (10 * val)$$

Note that the implicit counting value of the radix mask 3 is 0 in the example given in Figure A.2b.

The *radix counting operation* for a radix value is computed implicitly by adding $impl_{val}$ to the common counter $impl_{cnt}$ (as shown on Figure A.2b,c)

$$impl_{cnt} = impl_{cnt} + impl_{val}$$

The counting values of the three first radix buckets are easily restored from the common counter using shift operations (Figure A.2d)

$$cnt[val] = impl_{cnt} \gg (10 * val)$$

The *fourth counting value* - radix bucket 3, can be computed based on the three others using

$$cnt[3] = id - cnt[0] - cnt[1] - cnt[2]$$

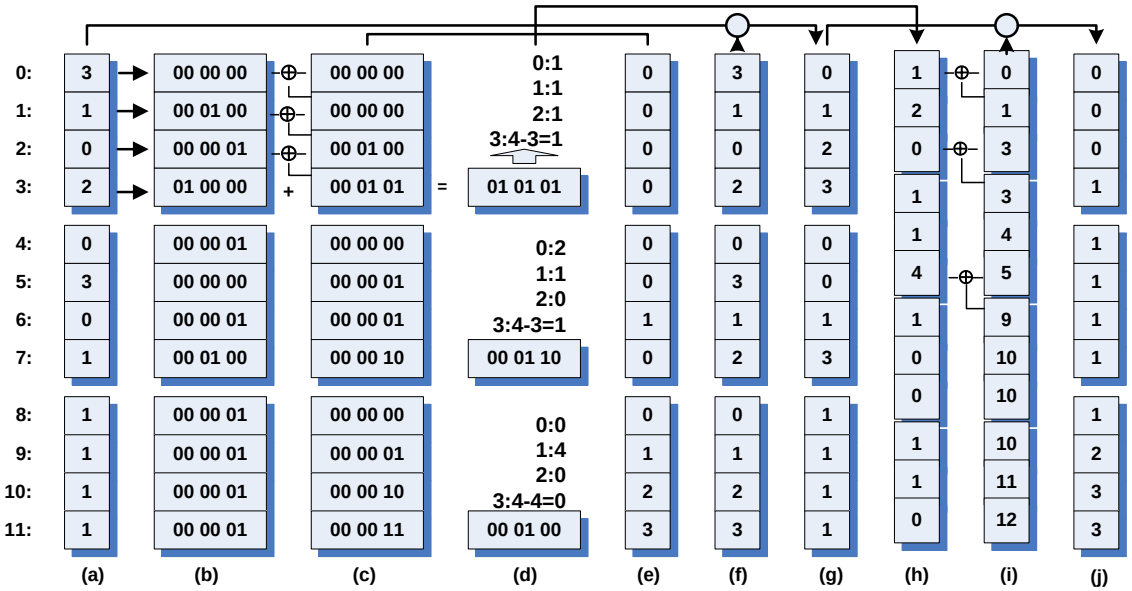


Figure A.2. Illustration of our implicit radix sorting (intermediate steps) a) Inputs b) Implicit-presentation of the input c) The local-prefix sum d) Number of each radix bucket e) Number of previous same bucket elements f) local rank g) presorted result h) Number of radix values in each block i) Start offset j) Sorted output

because the total number of preceding elements in the four radix buckets to an element index *id* is exactly *id*.

We apply the idea of implicit counting twice: First to compute the fourth counting value from the common counting values of the three other buckets, and second to reduce number of scan paths from four to one. The implicit counting function allows us to compute the four radix buckets with only a single sweep. This is twice as efficient as the implicit binary approach of Satish *et al.*.

A.3.2 Improving memory bandwidth

A.3.2.1 Hybrid data representation

To increase the memory bandwidth efficiency in the global shuffling step we propose a hybrid data representation that uses SoA for the input and AoS for the output. The conversion is illustrated in Figure A.3. The key observation is that although the proposed mapping methods [106, 55] are coalesced, the input of the mapping step still comes in fragments, we call this a nonideal effect. When it happens, the longer data format (i.e., int2, int4) suffers less performance degradation than the sorter one (int). Therefore, our AoS output data structure significantly reduces the suboptimal coalesced scattering effect in comparison to SoA output. Moreover, the multifragments require multiple coalesced shuffle passes which turns out to be costly. We saw the improvement by applying only one pass on the presorting data. We also achieved the full memory bandwidth for input which is 4× int2 length, using the texture cache.

A.3.2.2 Shared memory bank conflict-free access

We applied a bank conflict-free access pattern that stores a long format data structure, such as *float4*, into separate arrays. This handles the bank conflict inherently to the access of long format data on GPU shared memory. We then perform the operation on each component and write results back to the register. The bank conflict free mechanism is illustrated in Figure A.4. A similar concept has been applied by Satish *et al.*, but without a deeper analysis. In contrast we propose the bank-free conflict mechanism as a general optimization technique when working with long format data.

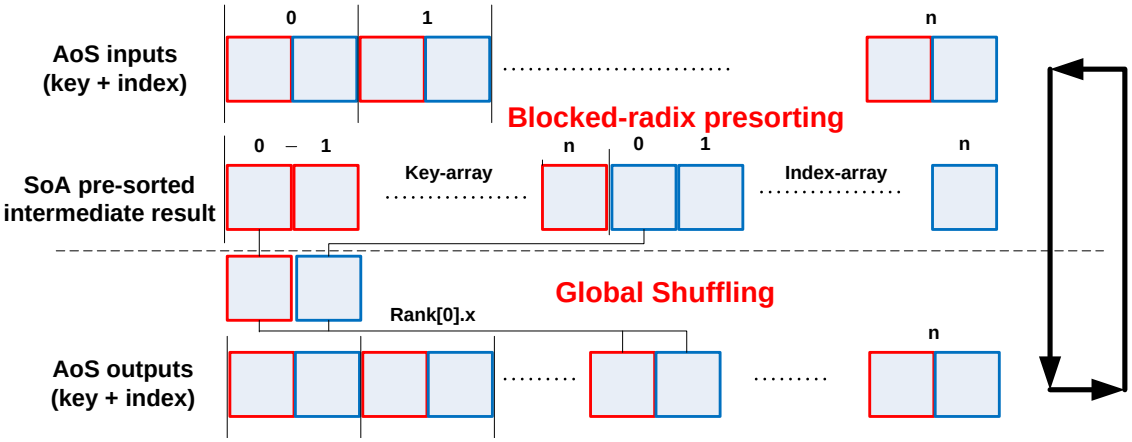


Figure A.3. The flow of our hybrid-data format. The conversion occurred implicitly inside the global shuffling kernel and at the beginning of local counting kernel using texture memory.

A.3.3 Performance tuning

A.3.3.1 Range limiter

While radix sorting time scales with the number of bits used to represent the data, the actual number of sorting bits may be substantially lower than the full length of the sorting key. For example sorting of the point-based simulation on the 256^3 grid only require 24 lower bits.

Our method exploits this prior knowledge about input ranges to reduce the number of radix passes. We use a simple scale and bias to map arbitrary numbers from the range $[a, b]$ to $[0, b - a]$. On the GPU we can quickly determine the range of the inputs by applying a reduce operation, which is as fast as a memory copy device operation [59].

While this works well with integers, such a simple mapping technique is not very efficient with floating point numbers as the range in its integer-converted format is likely to require as many as 32 bits, even for a small data range. However, as floating point numbers in the range of $[2^n, 2^{n+1})$ share the same leading exponential bits, we can reduce the range from full 32-bits to 24-bits of fractional data using the normalized linear mapping from $[a, b]$ to $[0.5, 1)$ range. This mapping yields a 30% performance improvement.

While the mapping is linear, it certainly is not one-to-one due to the adaptive range of floating point number representation, hence it is possible that two numbers may be mapped to the same number in the normalized range. This sorting result is an approximate sorting of the input. For many real time applications—especially in computer

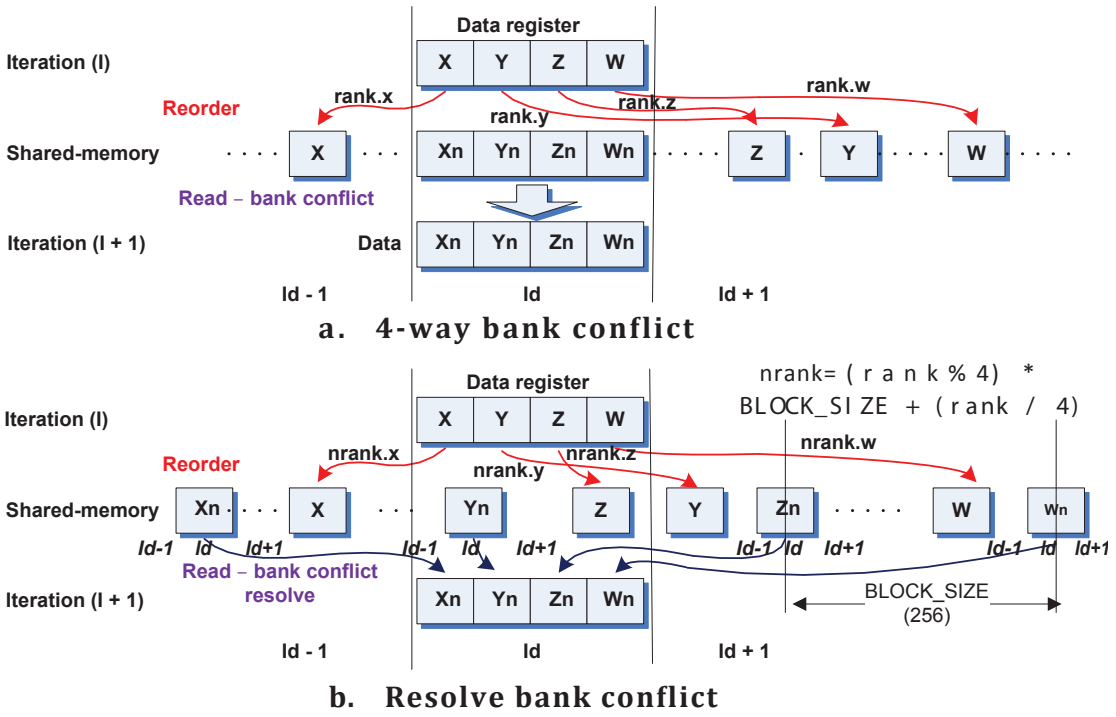


Figure A.4. Resolve the 4-way memory conflict

graphics and visualization—this approximation is acceptable.

A.4 Final evaluation

Our method extends and improves the fastest previously published implementation of Satish *et al.* [106] (CUDPP1.1) in both the presorting and global shuffling steps. Next, we will take a closer look at those two improvements.

We first focus on the presorting step. Please note that all timings are given in microseconds on an NVIDIA GTX 260 with 192 CUDA cores and 896MB memory. The size of the input $N(M)$ is the number of key-index input pairs in millions. To demonstrate the consistently improved behavior of our method we perform the presorting step with different input sizes N ranging from 2M to 16M. As shown in Figure A.5, our presorting step is about 1.5 to 1.8 times faster than the CUDPP 1.1 implementation.

Next, we take a look at the global shuffling improvements. We demonstrate our global shuffling step on 100 random radix-16 presorted arrays, which are partially sorted with a 16-bin radix in groups of 1024 elements, with sizes ranging from 1 to 16M key-value input pairs. The results show that by using an AoS structure instead of SoA as the

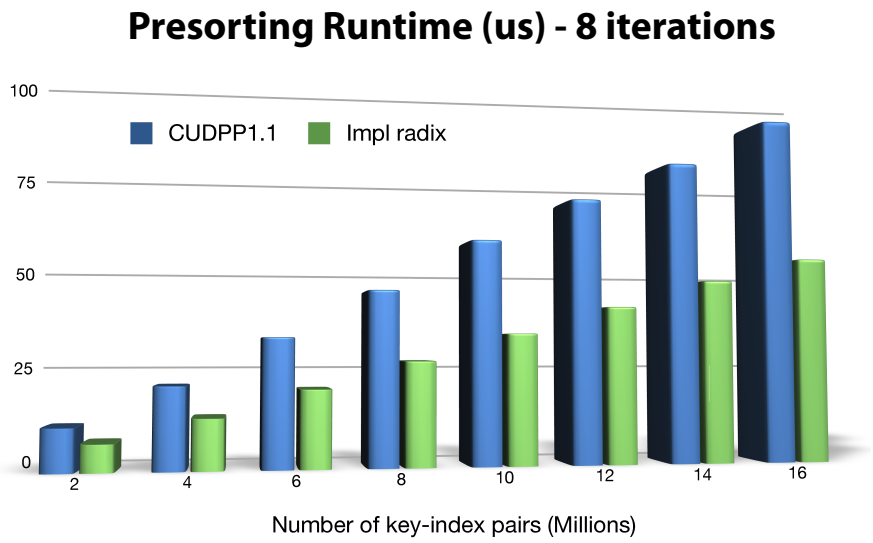


Figure A.5. Total run-time of presorting step (ms) with Implicit Radix and Satish CUDPP1.1 radix-16

output format, we improve the performance by 25%. At the same time, our one-pass implementation of SoA shuffling is more efficient than CUDPP1.1 by an additional 15%. Overall, our global shuffling is 1.4 times faster than that of CUDPP as illustrated in Figure A.7. It is approximately 1.4 times more expensive than a fully-coalesced memory copy operation, the upper bound.

Finally, we compare the component runtime in one iteration of a 16M-pair input between our implicit sorting and the Satish *et al.* (CUDPP1.1) implementation (Table A.1)

In Figure A.6, we measure the sorting rate (million-pairs per second) for random unsigned integer input arrays with size ranging from 1M to 16M. Both our method and the Satish *et al.* implementation require eight iterations for the 32-bit key. As can be seen, our method is able to sort about 180M key/value pairs per second on the target hardware, making it a factor 1.5 times faster than the the previous radix-16 implementation on

Table A.1. Component runtime comparision, in milliseconds, in one iteration of a 16M-pair input between our implicit sorting and the Satish *et al.* implementation.

16M pairs	Presort	Glb rank	Glb Shuff	Total	Memcpy*DtoD*
Satish et al	12.25	0.15	5.15	17.55	
Impl radix 16	8.15	0.15	3.75	12.05	2.78

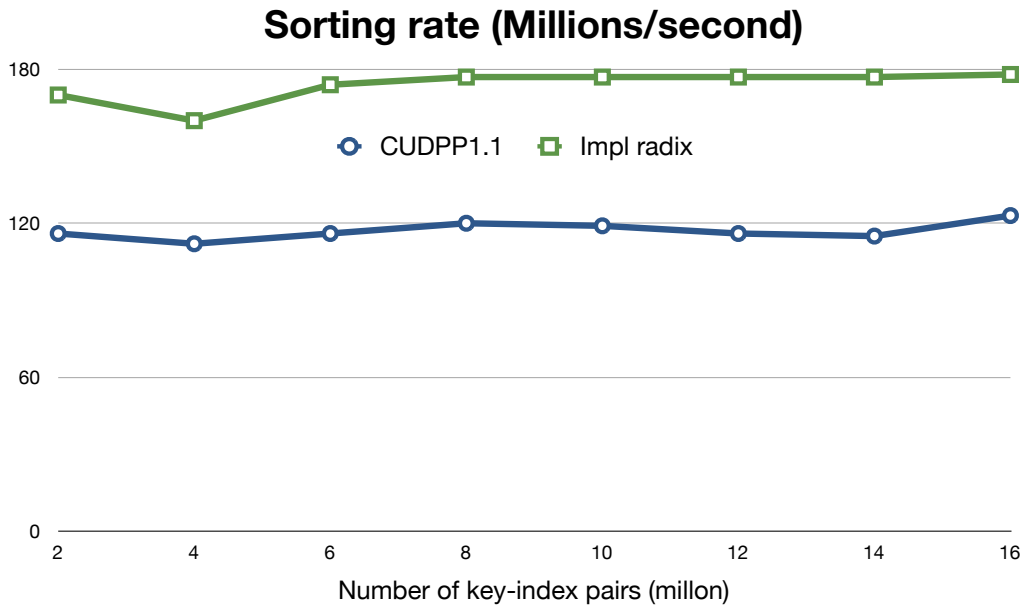


Figure A.6. The sorting rate comparison of random 32-bit unsigned inputs

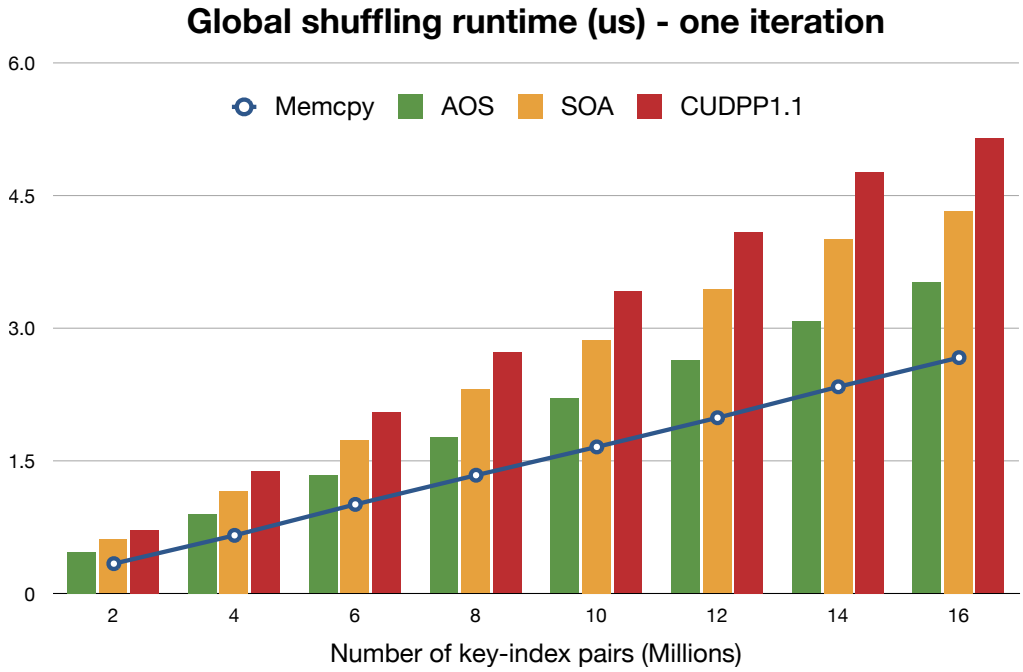


Figure A.7. Global shuffling run-time comparison (ns) between our implementation of global shuffling with AoS, SoA structures, and CUDPP1.1 in reference to the device to device memory copy of the same input size:

the same hardware. When using our approximate single precision floating point sorting scheme we achieve another 30% speedup as we need only sort 24 bits of the 32 bits key. We also observe significant performance improvements with integers when the dynamic range does not cover the full 32 bit range.

A.5 Discussion and future directions

In this chapter, we propose a new sorting algorithm to improve the performance of GPGPU implementations on modern GPU architectures including:

- A revision of the arithmetic intensity concept to evaluate the efficiency of GPU algorithms, which can be used as a guideline for optimization
- A new data structure and operations to exploit instructional parallelism, reducing significantly the amount of computation
- An adaptive data structure concept to tune performance at each algorithm stage

Our sorting framework efficiently addresses performance issues of existing approaches and, to some extent, successfully exploits both the compute power and memory bandwidth of modern GPUs.

While the constraint of 1024-element block sizes seem to affect the scalability of the method for future devices, we believe this is not the case since the number of threads in one block (256) sufficiently hide the memory latency. Moreover, with a minor change in the algorithm, we could increase the block size to 2048 elements with one implicit counting bit to achieve a 33-bit implicit counter. However, on the current architecture 1024 elements is the optimal size. Although our approach increases the arithmetic intensity of sorting solution, the full power of the GPU has not yet been exploited. One possible solution is to combine our implicit counting and multiple parallel scan path of Ha *et al.* [55], which also overcomes the 1024 block size limitation.

As we mentioned in related work section A.1.2, a combination of our technique and Duane Merrill and Andrew Grimshaw's work [81] would potentially result in a faster sorting implementation. In particular, our presorter and counting step could benefit from their fast multiscan implementation, which is twice faster than CUDPP scan framework we currently based on. And also our framework could exploit their visiting logic technique to reduce the number of operation and also increase the memory bandwidth utilization. On the other hand, their framework might employ our two-level implicit counting to reduce the complexity of counting step, and they also can apply our hybrid data to

increase memory bandwidth utilization further. Their sorter or any radix-based sorting framework will be enhanced with our range limiter and approximate floating point sorting strategies.

Our future work will concentrate on analyzing the benefits and orthogonality of different sorting frameworks, then combine these techniques to find a solution that fully exploits the potential processing power and bandwidth of modern GPUs. Beside, we want to exploit radix algorithm further in building other high performance algorithmic building blocks. For example, parallel segmented sorting, an algorithm to sort multiple segments of the input at the same time, can be easily extended from the radix sorting framework. Segmented sorting has applications in visual sorting when fragments are sorted per rays.

Last but not least, we want to combine GPU and CPU sorting to exploit both memory bandwidth and processing power of GPUs and CPUs to achieve the highest performance and to handle extremely large data sets on GPU clusters.

APPENDIX B

SOFTWARE ARCHITECTURE

In this appendix, we present a high level description of the software package of our GPU framework. We provide an overview of the architecture, essential functions and modules, coding styles, and the development features that allow developers to adapt to the future changing of the system hardware.

B.1 Overview architecture

B.1.1 Atlas construction data flow

The overview of data flow architecture of our GPU atlas construction framework is shown on Figure B.1. The inputs are separated into data files and parameters files. We allow different formats of input data using ITK and VTK IO functions. However, in the

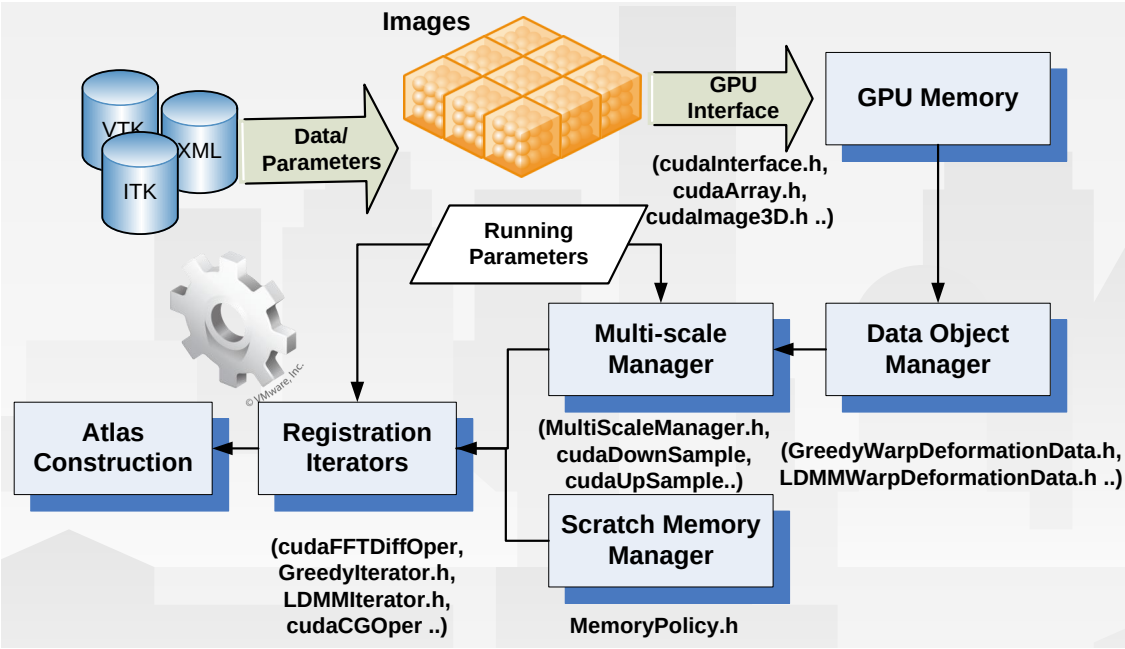


Figure B.1. Atlas construction framework data flow architecture overview.

streaming mode, each image is saved in a binary format which maps directly to the CPU memory. The parameters are stored in XML format to provide developers the flexibility to change these parameters and to integrate our system with existing user interfaces easily. The GPU interface provides functions to exchange data between CPU and GPU memory. Data are managed using data manager objects that pre-allocate essential memory buffers for computation. Data is resampled using multiscale managers for multiscale processing. The scratch memory manager provides temporary memory buffers required by GPU algorithms. It provides memory for computation, reuses available allocated memory, and minimizes the amount of memory control. The registration iterators perform registration strategies—in particular, the greedy iterative and the LDDMM algorithms—to compute the optimal deformation field to register two objects. The outputs are then used to compute the atlas. Figure B.1 also displays essential modules, which offer processing functions for each stage. Developers are allowed to customize these modules with their own implementations or to extend the framework with new functionality.

B.1.2 Software development overview

The overview software architecture of the system is shown on Figure B.2. Functions are implemented using C++-based languages: CUDA and C++. There are four development levels divided into two main stages: device kernels and algorithm modules.

B.1.2.1 Device kernel and interface functions

Figure B.3 illustrates a typical kernel/interface pair that performs the adding with a constant function. The device kernels are developed using the CUDA programming model and complied using NVIDIA CUDA compiler. Device kernels are stored with ".cu" extension to differentiate from algorithm modules implemented using general C++ programming and stored with ".cpp" extension. The device kernels are named with a *_kernel* suffix. Each kernel is attached with an interface function so that it is called from users' code as a regular C++ function. For consistency, an interface function shares the same name with its kernel without the *_kernel* suffix.

Execution of a kernel is configured through parallel configuration parameters—*threads* and *grids*—which define how tasks and data are divided among multiprocessors of the GPUs and among multiple threads of each multi-processor. The interface functions compute these parameters based on sizes of the input and hardware configuration of the systems. The unified model of GPU programming allows a stable multilevel, hierarchical

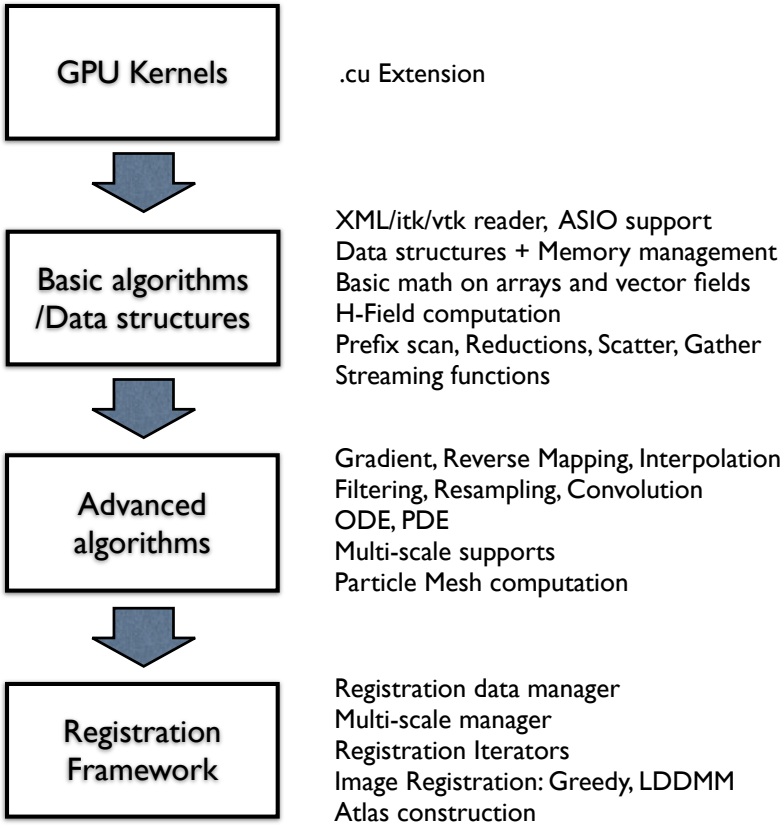


Figure B.2. Software development architecture of the AtlasWerk image registration framework.

```
namespace cplVectorOpers {
  template<class T> __global__ void AddC_kernel(T* do, T* di, T c, int n){
    uint blockId = get_blockID();
    uint      id = get_threadID(blockId);
    if (id < n)
        do[id] = di[id] + c;
  }
  template<class T> void AddC(T* do, T* di, T c, int n, cudaStream_t st){
    dim3 threads(BLOCK_ALIGN);
    dim3 grids=makeGrid(iDivUp(n,threads.x));
    AddC_kernel<<<grids, threads, 0, st>>>(do, di, c, n);
  }
}
```

Figure B.3. A sampler of kernel/interface functions, which adds a constant to an array. The function is stored with .cu file extension and is compiled using CUDA compiler.

execution structure which maps to coarse-grain and fine-grain parallelism levels. The mapping between kernel configurations and execution grids is defined using the inline functions such as *get_threadID* and *get_blockID*. These functions can be customized to adapt users' mapping strategies.

While the resource allocating strategy remains, the granularity, *BLOCK_ALIGN*, might change from one hardware generation to the other. We encode these constants in a header file and allow developers to choose optimal granularity parameters depending on hardware configuration of the running system. Further hardware-specific optimization is performed inside kernel functions to harness processing power from particular hardware.

No resource allocation is allowed at the device kernel level. The purpose is to ensure that there is no hidden cost because of excessive resource-allocating. This strategy enforces memory reuse based on a temporal memory model—a *scratch pad*. This optimization is effective, especially with multi-image processing operations as these computations potentially share the same scratch memory buffer.

B.1.2.2 Algorithms

The algorithm development of our framework is divided into three levels: general data structures and functions, advanced image processing functions, and registration algorithms.

The *general data structure and function* modules provide the implementation of basic functions on basic data structures: 1D array, 3D image, and 3D vector field. These functions are built on the top of the device kernel layer. It provides one level of code protection with parameter checking to eliminate potential bugs due to users' misuses of the functions. This programming feature helps developers to isolate bugs quickly and to reduce debugging time. We classify functions into several namespaces and groups based on the similarity in algorithm structure and functionality. The most important basic algorithm function sets are basic array and vector field operations, and reduction functions.

The basic array and vector field computations (*cplVectorOpers* and *cplVector3DOpers*) are implemented using the n-ary optimization strategy, see *VectorMath.h*. Typically, the CUDA implements an execution model that uploads kernel parameters from CPU memory to shared-memory. As shared-memory registers have very low latency and are shared between execution threads, the computation is fast and efficient. However, this execution model requires input parameters located on host memory. That is, if an output

of a function is used as an input parameter of following calls, it needs to be copied over the host. This requires a synchronization between the device computation flow and the data transfer process from the device to host, an unintended cross-stream synchronization that potentially reduces the effectiveness of the asynchronous processing model applied in out-of-core multi-image processing. To deal with this problem, we implement a *device-parameter* computational mode with extended functions which load parameters directly from device memory. Using texture cache to upload parameters from the device, we are able to achieve optimal performance equivalent to the regular execution model while we eliminate the need to copy back data to the host. See *VectorMathExt.h* for examples of how to implement these functions.

The reduction classes, i.e., *cudaReduce.h,cudaReduceStream.h*, contain implementations of most reduction functions from single-input single-operation functions such as max, min, and sum value of an array to multiple-input multiple-operation functions such as the vector product, the vector range, and max of absolute value, sum of absolute value and sum of square value of an array. The reduction functions are implemented using the template programming model. Based on the similarity of reduction optimization, we use the template model for both data types and operations. Thus, we can easily extend these functions to cover different types and operations. This strategy helps us save the coding and debugging time and maintain the implementation efficiency. We implement two versions of the reduction class: a regular, cross-stream version which returns output values to the CPU memory and an in-stream version which returns the values to GPU device memory. The in-stream version accompanied by the aforementioned extended functions is used for asynchronous processing.

We build *advanced image processing* functions—such as gradient, interpolation, filtering, reverse mapping, ODE and PDE computation—on the top of the basic functions. On the highest level, the *registration framework* combines basic and advanced image processing functions to implement registration algorithms. The framework defines the data and control flow between modules in the lower levels. Beside a registration framework, we support several advanced programming features, such as memory management and unit testing, to ease the code development process, to increase the scalability, to provide optimal performance and to adapt to the changing hardware.

B.2 Memory management

The scalability of a framework depends not only on the scalability of the computational algorithms but also on how the memory management is applied. As memory bandwidth of a system is limited and memory control operations (allocation, deallocation) are inherently sequential, maximizing memory bandwidth efficiency and minimizing the number of memory control operations are essential to optimize the performance.

B.2.1 Customize memory allocation functions

One of the key ideas for maximizing bandwidth efficiency in parallel processing models (CUDA, Open CL, and SIMD programming) is to access data on aligned buffers. While memory buffers allocated by the CUDA memory allocator are aligned on a 256-byte boundary, this is provided without any guarantees. Furthermore, there is no restriction that memory assigned with subsequent calls will be mapped continuously. To allow the 1D optimization, we build n-D image structures hierarchically on the top of 1D array representation. Our array allocator provides boundary alignment and automatic memory cleanup to prevent memory leaks. We apply Resource Acquisition Is Initialization (RAII) paradigm to make functions thread-safe.

The hierarchical structure of memory objects allows us to perform data constraints checking at each level to guarantee that the use of a function on particular data is safe. Furthermore, it provides the ability to make use of optimized functions when the inputs are satisfied certain conditions. This structure also helps developers to detect memory problems quickly during debugging process.

B.2.2 Preallocate memory buffer

To minimize performance influence of memory control, we apply a preallocating memory strategy that computes the amount of required memory and then allocates the memory in advance. This strategy prevents memory fragment and potential memory leaks. Furthermore, a preallocated temporal buffer is employed. As input images typically have similar sizes, this memory could be efficiently shared among inputs as well as functions.

B.2.3 Eliminate data copy redundancy

Though the data copy is normally considered an inexpensive operation, we found that in some applications this operation might have a significant influence to the total runtime.

While GPUs provide both massive amounts of computational power and large memory bandwidth, data copy functions do not make use this computational power while they still consume the memory bandwidth. As we have shown on the performance graph with n-ary functions (Figure 2.5), the memory copy reference spends almost the same amount of time as an n-ary function. To amortize the performance, we need to minimize the amount of extraneous data memory copies. Instead of doing a memory copy, it would be better to combine the copy operation with another arithmetic operation that operates on the same memory data. For example, a function pair $b = a$ and $b+ = c$ is equivalent to the $b = a + c$. The latter is twice as fast with c is a constant as it consumes half of the memory bandwidth in comparison to the former.

To provide the capability to optimize the memory bandwidth usage and eliminate redundant memory copies, we support both in-place functions—which have outputs among the inputs, and out-of-place functions which have outputs separated from the inputs. The flexibility to choose different implementations allows developers to amortize memory bandwidth consumption. In addition, by introducing in-place and out-of-place functions, we allow further computation and memory optimization from GPU compilers. Without extra hints from the developers, this optimization could not be done. The extra memory copy can also be eliminated using our memory scratch buffer, which is organized as a circular buffer so that instead of copying data from the scratch buffer, a swap memory pointer is sufficient.

B.3 Programming features

B.3.1 Scalability and portability with macro and inline functions

It is often required that developers to specialize in their implementations to optimize on the particular hardware. Fortunately, the convergence of parallel hardware architecture, especially in GPU computing allows more stable and scalable programming methodologies to develop, such as CUDA and OpenCL. GPU computing models are mainly based on data parallelism, and hence it is scalable to growth of the data. The multi-GPU and multi-CPU architectures provide a higher level of parallelism supporting both data parallelism and task parallelism. In our software development architecture, the kernels are optimized to exploit fine-grain parallelism, while high level functions make use of coarse-grain parallelism. The hierarchical development of the software make higher level functions more stable while the convergence of the architecture allows expressing

low level algorithms, the kernels, independently from the hardware. Consequently, it only requires to change the granularity and/or mapping strategies in the kernel implementation when the hardware configurations change. We encode these constants and mapping strategies using macro and inline functions. This allows the compilers to optimize the binary execution based on specific system hardware. Similar approaches have been deployed in Intel Integrated Performance Primitives library (IPP) [115].

B.3.2 Naming and scope

As mentioned earlier, we classify the algorithms based on the functionality. We group functions using high level programming features such as *namespace* and *class*. For example, the *cplVectorOpers* namespace combines basic functions on arrays, while the *cplReduce* class contains the implementation of reduction functions. The decision for choosing a class over a namespace depends on whether a data management for the implementation of functions is required. As an illustration, the reduction functions require a fixed-size memory buffer both on host and device to implement it on CUDA. This buffer is preallocated so that we eliminate the overhead of creating this buffer every time a reduction function is called. It also makes the implementation of reduction functions more transparent as the supporting memory buffer is hidden inside the class.

A consistent naming strategy is applied in the framework to facilitate the coding process and to lower the learning curve for the system development. Besides naming a kernel with a "_kernel" suffix, a GPU function is prefixed with "cpl" to indicate a *CUDA processing library* module. The naming for multioperation basic functions based on how the function is spelled out; an underscore "_" is used to separate groups of operations, and the suffix "_I" implies an in-place processing function, which has the first parameter served as both the output and the first input. For example, a "cplAdd_Mul_I" function, where input parameters are three arrays a, b, c, performs the function $a+ = b * c$ on GPUs. We also follow the *output-first* rule to imply that the first parameter is the output of the functions. This complies with the regular expression of an assignment function.

B.3.3 Unit testing

We apply unit testing strategy to aids the code development and maintenance. This is to ensure an implementation of a function meets design requirements. For each of the bugs spotted, we provide a testing function to ensure the bug will not reoccur when the source code is refactorized for efficiency. We produce the ground truth with both synthetic data

whose outputs can be computed implicitly and real data. For numerical functions, it is important that the output of a function meets its ground truth with desired accuracy. As the debugging process on GPU is difficult, we also provide a CPU reference code to allow step-by-step comparison as it is easier to develop, to debug and to ensure the correctness of CPU code. The CPU reference code can be used for performance comparison. The testing functions are indicated with the "test" prefix.

B.3.4 Extensible with generic programming

As briefly mentioned in Chapter 2, we apply a generic programming paradigm (GP) with CG template solver to provide the flexibility and extensibility to our framework. Pioneered by Alexander Stepanov and David Musser [88], GP most prominent success is the Standard Template Library, which became part of the ANSI/ISO C++ standard. The approach is an effective mechanism to build a generalizable framework without sacrificing efficiency. There are several advantages to generic programming.

- Generic programming is about generalizing software components so that they can be easily reused in a wide variety of situations. For example, by making algorithms as templates, it could save developers significant time to make a comparison test for different strategies to solve a problem or even provide an optimal solution which can adapt to different parameter sets and ranges.
- If the code variations with different data structures and algorithms are the major concerns, generic code is easier to write and to get correct. You create only one generic version of your class or function instead of manually creating specializations. By reducing the duplication, generic programing reduces manual code bloating, which is error-prone, and increase the maintainability of the codes.
- Besides providing a benefit similar to macro programing, which allows automatic code generation and further compiler optimization with no-runtime overhead, generic programing offers type safety and function encapsulation through specialization that makes the code base more robust.
- While the generalization abstracts generic code from algorithms and encourages the code modularization that increases the credibility and correctness, the specialization allows functions to be overridden to adapt to the variation of data types and algorithms to achieve the highest efficiency. This partial specialization even offers further flexibility and advantages to developers such as default implementations,

which shelter developers from parameter explosion and helps them to choose the best available implementations.

- Generic programing concentrates on the logic of the solution rather than the fill-in details. It sets the constraints for the implementation of template data structures and functions. As long as these constraints are satisfied, the correctness and the complexity are profilable. For example, sorting algorithms with template types guarantee the same algorithmic complexity with integer or floating point keywords.
- Generic programing separates the potential bugs of generic codes to logical error and the specialization codes to the constraint miss-matched. Furthermore, fixing bugs from generic codes will guarantee all the related codes are free from the fixed bugs.

All the benefits of generic programming fit with the desired properties of a stable, error-free, and high performance processing framework, so we employ this programing paradigm throughout the code base, especially in designing primary algorithms. Figure B.4 shows the multiscale image registration implementation using C++ template programming, which has the ability to incorporate and compare different registration algorithms with different data structures and parameter sets such as Greedy or LDDMM algorithms into the same multiscale framework.

Using the generic programing concept, we separate the design of a multiscale registration framework into three parts: the deformation data structure and functions, the scaling functions through a scaling manager, and the registration algorithms through the registration iterators. We define the functionality, the constraints and the interfaces between them. We achieve an encapsulation level equal to or even higher than the object polymorphism of object oriented programming since we did not require the same function calls but only a similar interface. In addition, we provide developers a more flexible, and higher performance design without the polymorphism overhead.

B.4 Conclusions and future work

In this appendix, we present the overview architecture of our software system. While the target of this appendixis to give developers initial ideas about the framework to facilitate their software development, we also discuss our perspectives on how to deal with the scalability and portability problem, how to adapt to change in future hardware as well as how to achieve the optimal performance. We believe the development of the

```
template<typename DeformationStructure,
         typename MultiScaleManagerPolicy,
         typename IteratorPolicy>
void MultiscaleInterface<DeformationStructure,
                         MultiScaleManagerPolicy,
                         IteratorPolicy>::Run()
{
    // Preprocessing : Initialize the memory, normalize the data
    mDeformation->Initialize();
    mScaleManager->Start();
    int nScaleLevel = mScaleManager->getNumLevels();
    for (size_t is = 0; is < nScaleLevel; ++is) {
        // Update the data
        mDeformation->UpdateScale(mScaleManager);
        // Update the iterator
        mIterator->UpdateScale(mScaleManager);
        // Run the iterator
        int nIter = mScaleManager->getNumIters();
        for (size_t iter=0; iter < niters; ++iter) {
            mIterator->Iterate(mDeformation);
        }
        // Goto the next scale
        mScaleManager->Next();
    }
    mDeformation->Finalize();
}
```

Figure B.4. C++ template implementation of the multiscale registration

framework should be able to address these problems. For future work, we would like to apply compile optimization techniques, such are register-count optimization and register allocation, to analyze the execution of the algorithm. Our target is to provide a more flexible memory management that adapts to different system configuration, from systems with limited resources to the systems with a large amount of memory and to ensure the optimal performance is achieved.

Note that our code is a part of the AtlasWerks project which is free for research and is available to download at `http://www.sci.utah.edu/software/13/370-atlaswerks.html`

REFERENCES

[1] Alattar, A. A probabilistic filter for eliminating temporal noise in time-varying image sequences. In *Circuits and Systems, 1992. ISCAS '92. Proceedings., 1992 IEEE International Symposium on* (May 1992), vol. 3, pp. 1491 –1494 vol.3.

[2] Aljabar, P., Bhatia, K., Murgasova, M., Hajnal, J., Boardman, J., Srinivasan, L., Rutherford, M., Dyet, L., Edwards, A., and Rueckert, D. Assessment of brain growth in early childhood using deformation-based morphometry. *NeuroImage 39*, 1 (2008), 348 – 358.

[3] Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. SETI@home: An experiment in public-resource computing. *Commun. ACM 45* (November 2002), 56–61.

[4] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. The landscape of parallel computing research: A view from Berkeley. Tech. Rep. UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.

[5] ATI. *AMD Accelerated Parallel Processing OpenGL Programming Guide*, January 2011.

[6] Barnes, D. G., and Fluke, C. J. Incorporating interactive three-dimensional graphics in astronomy research papers. *New Astronomy 13*, 8 (2008), 599 – 605.

[7] Barney, B. Introduction to parallel computing, Nov 2007.

[8] Baskaran, M. M., and Bordawekar, R. Optimizing sparse matrix-vector multiplication on gpus. Tech. rep., IBM Technical Report, 2008.

[9] Beberg, A. L., Ensign, D. L., Jayachandran, G., Khaliq, S., and Pande, V. S. Folding@home: Lessons From Eight Years of Volunteer Distributed Computing. In *8th IEEE International Workshop on High Performance Computational Biology (HiCOMB 2009) in Conjunction with the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009)* (2009).

[10] Beg, M. F., Miller, M. I., Trouvé, A., and Younes, L. Computing large deformation metric mappings via geodesic flows of diffeomorphisms. *Int. J. Comput. Vision 61* (February 2005), 139–157.

[11] Bell, N., and Garland, M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), ACM, pp. 1–11.

[12] Belleman, R. G., Bedorf, J., and Zwart, S. F. P. High performance direct gravitational n-body simulations on graphics processing units ii: An implementation in cuda. *New Astronomy 13*, 2 (2008), 103 – 112.

[13] Bittner, J., Wimmer, M., and Piringer, H. Coherent Hierarchical Culling: Hardware occlusion queries made useful.

[14] Blanchette, J., and Summerfield, M. *C++ GUI Programming with Qt 4*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[15] Blythe, D. The Direct3D 10 system. *ACM Trans. Graph. 25*, 3 (2006), 724–734.

[16] Bordawekar, R., Choudhary, A., Kennedy, K., Koelbel, C., and Paleczny, M. A model and compilation strategy for out-of-core data parallel programs. *ACM SIGPLAN Notices 30*, 8 (1995), 1–10.

[17] Boyce, J. Noise reduction of image sequences using adaptive motion compensated frame averaging. In *Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on* (Mar. 1992), vol. 3, pp. 461 –464 vol.3.

[18] Bro-Nielsen, M., and Gramkow, C. Fast fluid registration of medical images. In *VBC '96: Proceedings of the 4th International Conference on Visualization in Biomedical Computing* (London, UK, 1996), Springer-Verlag, pp. 267–276.

[19] Brown, A., Mowry, T., and Krieger, O. Compiler-based i/o prefetching for out-of-core applications. *ACM Transactions on Computer Systems (TOCS) 19*, 2 (2001), 170.

[20] Buck, I. Gpu computing: Programming a massively parallel processor. In *Proceedings of the International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2007), CGO '07, IEEE Computer Society, pp. 17–.

[21] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. Brook for GPUs: Stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers* (New York, NY, USA, 2004), ACM, pp. 777–786.

[22] Budruk, R., Anderson, D., and Solari, E. *PCI Express System Architecture*. Pearson Education, 2003.

[23] Caron, E., Desprez, F., and Suter, F. Out-of-core and pipeline techniques for wavefront algorithms. *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers 01* (Apr 2005).

[24] Chatterjee, S., Blelloch, G. E., and Zagha, M. Scan primitives for vector computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing* (Los Alamitos, CA, USA, 1990), Supercomputing '90, IEEE Computer Society Press, pp. 666–675.

[25] Cherfils, J., and Janin, J. Protein docking algorithms: Simulating molecular recognition. *Current Opinion in Structural Biology 3*, 2 (1993), 265 – 269.

[26] Chiang, Y., El-Sana, J., Lindstrom, P., Pajarola, R., and Silva, C. Out-of-core algorithms for scientific visualization and computer graphics. *IEEE Visualization* (2003).

[27] Chiang, Y.-J., Goodrich, M. T., Grove, E. F., Tamassia, R., Vengroff, D. E., and Vitter, J. S. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 1995), SODA '95, Society for Industrial and Applied Mathematics, pp. 139–149.

[28] Christensen, G., Rabbitt, R., and Miller, M. Deformable templates using large deformation kinematics. *Image Processing, IEEE Transactions on 5*, 10 (oct 1996), 1435 –1447.

[29] Christensen, G. E., Miller, M. I., Vannier, M. W., and Grenander, U. Individualizing neuroanatomical atlases using a massively parallel computer. In *Computer* (1996), vol. 29, IEEE Computer Society, pp. 32–38.

[30] Corp, N. *NVIDA CUDA Computer Unified Device Architecture Programming Guide 2.0*, Jul 2008.

[31] Dally, W. J., Labonte, F., Das, A., Hanrahan, P., Ahn, J.-H., Gummaraju, J., Erez, M., Jayasena, N., Buck, I., Knight, T. J., and Kapasi, U. J. Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing* (2003), IEEE Computer Society, p. 35.

[32] Datar, M., Cates, J., Fletcher, P., Gouttard, S., Gerig, G., and Whitaker, R. Particle based shape regression of open surfaces with applications to developmental neuroimaging. In *MICCAI* (2009), no. 5762 in LNCS, Springer Verlag, pp. 167–174.

[33] Davis, B., Fletcher, P., Bullitt, E., and Joshi, S. Population shape regression from random design data. *ICCV 2007* (Oct. 2007), 1–7.

[34] Dongarra, J., Beckman, P., Aerts, P., Cappello, F., Lippert, T., Matsuoka, S., Messina, P., Moore, T., Stevens, R., Trefethen, A., and Valero, M. The international exascale software project: A call to cooperative action by the global high-performance community. *Int. J. High Perform. Comput. Appl. 23* (November 2009), 309–322.

[35] Dufaux, F., and Moscheni, F. Motion estimation techniques for digital tv: A review and a new contribution. *Proceedings of the IEEE 83*, 6 (June 1995), 858 –876.

[36] Duke, K. A., and Wall, W. A. A professional graphics controller. *IBM Systems Journal 24*, 1 (1985), 14 –25.

[37] Durrleman, S., Pennec, X., Trouvé, A., and Ayache, N. Sparse approximations of currents for statistics on curves and surfaces. In *MICCAI 2008* (2008).

[38] Dzatko, D., and Shanley, T. *AGP System Architecture*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[39] Fernando, R. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, Old Tappan, USA, 2004.

[40] Florea, L., Hartzell, G., Zhang, Z., Rubin, G. M., and Miller, W. A computer program for aligning a cDNA sequence with a genomic DNA sequence. *Genome Research 8*, 9 (September 1998), 967–974.

[41] Flynn, M. J. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on C-21*, 9 (Sep 1972), 948 –960.

[42] Fraedrich, R., Schneider, J., and Westermann, R. Exploring the millennium run - scalable rendering of large-scale cosmological datasets. *IEEE Transactions on Visualization and Computer Graphics 15* (2009), 1251–1258.

[43] Geist, A., and Lucas, R. Major computer science challenges at exascale. *Int. J. High Perform. Comput. Appl. 23* (November 2009), 427–436.

[44] Glaunès, J., Trouvé, A., and Younes, L. Diffeomorphic matching of distributions: A new approach for unlabelled point-sets and sub-manifolds matching. In *CVPR* (2004), IEEE Computer Society, pp. 712–718.

[45] Glaunes, J., Trouvé, A., and Younes, L. Diffeomorphic matching of distributions: A new approach for unlabelled point-sets and sub-manifolds matching. In *CVPR* (2004).

[46] Goesele, M., Snavely, N., Curless, B., Hoppe, H., and Seitz, S. Multi-view stereo for community photo collections. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on* (Oct 2007), pp. 1 –8.

[47] Gonzalez, R. C., and Woods, R. E. *Digital Image Processing*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1992.

[48] Goodrich, M. T., Tsay, J.-J., Vengroff, D. E., and Vitter, J. S. External-memory computational geometry. In *Proceedings of the 1993 IEEE 34th Annual Foundations of Computer Science* (Washington, DC, USA, 1993), IEEE Computer Society, pp. 714–723.

[49] Govindaraju, N., Gray, J., Kumar, R., and Manocha, D. GPUTeraSort: High performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2006), ACM, pp. 325–336.

[50] Grand, S. L. Broad-phase collision detection with cuda. In *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley, Reading, Massachusetts, USA, Aug. 2007.

[51] Greengard, L., and Strain, J. The fast gauss transform. *SIAM J. Sci. Stat. Comput. 12*, 1 (1991), 79–94.

[52] Greß, A., and Zachmann, G. GPU-ABiSort: Optimal parallel sorting on stream architectures. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (Rhodes Island, Greece, 25–29 April 2006).

[53] Guiang, C. S., Milfeld, K. F., Purkayastha, A., and Boisseau, J. R. Memory performance of dual-processor nodes: Comparison of intel xeon and amd opteron memory subsystem architectures. In *Proceedings for ClusterWorld Conference and Expo 2003* (2003).

[54] Ha, L., Krüger, J., Joshi, S., and Silva, C. T. *Multiscale Unbiased Diffeomorphic Atlas Construction on Multi-GPUs*, vol. I. Elsevier, Maryland Heights, USA, Jan 2011.

[55] Ha, L., Kruger, J., and Silva, C. T. Fast 4-way parallel radix sorting on GPUs. *CGF, Computter Graphic Forum 8* (2009), 2368–2378.

[56] Ha, L. K., Krüger, J., Fletcher, P. T., Joshi, S., and Silva, C. T. Fast parallel unbiased diffeomorphic atlas construction on multi-graphics processing units. In *EUROGRAPHICS Symposium on Parallel Graphics and Visualization 2009* (2009).

[57] Haralick, R. M., and Shapiro, L. G. *Computer and Robot Vision*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1992.

[58] Harris, M. Mapping computational concepts to gpus. In *ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), SIGGRAPH '05, ACM.

[59] Harris, M. Optimizing parallel reduction in cuda. `http://tinyurl.com/6dazkd/reduction.pdf`, 2007.

[60] Harris, M., Owens, J., Sengupta, S., Zhang, Y., and Davidson, A. CUDPP: CUDA data parallel primitives library. `http://www.gpgpu.org/developer/cudpp/`, 2007.

[61] Hays, J., and Efros, A. A. Scene completion using millions of photographs. *ACM Trans. Graph. 26* (July 2007).

[62] Hoberock, J., and Bell, N. Thrust: A parallel template library, 2009. Version 1.3.

[63] Hockney, R. W., and Eastwood, J. W. *Computer Simulation Using Particles*. Taylor and Francis, Bristol, PA, USA, 1989.

[64] Hockney, R. W., and Jesshope, C. R. *Parallel Computers Two: Architecture, Programming and Algorithms*, 2nd ed. IOP Publishing Ltd., Bristol, UK, UK, 1988.

[65] Hu, C., Yao, G., Wang, J., and Li, J. Transforming the adaptive irregular out-of-core applications for hiding communication and disk i/o. *Proceedings of the 2007 OTM Confederated International Conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, ODBASE, GADA, and IS Part II* (Nov 2007).

[66] Intel. Rethink flexibility with a configurable Intel Atom TM processor, Nov 2010.

[67] Joshi, S., Davis, B., Jomier, M., and Gerig, G. Unbiased diffeomorphic atlas construction for computational anatomy. *Neuroimage 23 Suppl. 1* (2004), S151–S160.

[68] Kanellos, M. New life for Moore's Law. *CNET News.com* (April 2005).

[69] Keltcher, C. N., McGrath, K. J., Ahmed, A., and Conway, P. The amd opteron processor for multiprocessor servers. *IEEE Micro 23* (March 2003), 66–76.

[70] Knickmeyer, R. C., Gouttard, S., Kang, C., Evans, D., Wilber, K., Smith, J. K., Hamer, R. M., Lin, W., Gerig, G., and Gilmore, J. H. A structural MRI study of human brain development from birth to 2 years. *J. Neurosci. 28* (Nov 2008), 12176–12182.

[71] Kogge, P. M. *The architecture of pipelined computers.* McGraw-Hill, New York, USA, 1981.

[72] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. 28* (September 1979), 690–691.

[73] Lee, M., Jeon, J.-h., Bae, J., and Jang, H.-S. Parallel implementation of a financial application on a GPU. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human* (New York, NY, USA, 2009), ICIS '09, ACM, pp. 1136–1141.

[74] Leischner, N., Osipov, V., and Sanders, P. GPU sample sort, 2009.

[75] Lorensen, W. E., and Cline, H. E. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph. 21* (August 1987), 163–169.

[76] Lorenzen, P., Davis, B., and Joshi, S. Unbiased atlas formation via large deformations metric mapping. In *Med Image Comput Comput Assist Interv Int Conf Med Image Comput Comput Assist Interv (MICCAI)* (2005), J. Duncan and G. Gerig, Eds., vol. 8 (Pt. 2), pp. 411–418.

[77] Luebke, D., Harris, M., Govindaraju, N., Lefohn, A., Houston, M., Owens, J., Segal, M., Papakipos, M., and Buck, I. Gpgpu: general-purpose computation on graphics hardware. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2006), SC '06, ACM.

[78] Macovksi, A. Tolerating latency through software-controlled data prefetching. *en.scientificcommons.org* (Jan 1994).

[79] Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A., and Upton, M. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal 6*, 1 (2002), 4–15.

[80] Merrill, D., and Grimshaw, A. A. parallel scan for stream architectures. Tech. Rep. CS2009-14, U of Virginia, Dept of Computer Science, Charlottesville, VA, USA, 2010.

[81] Merrill, D., and Grimshaw, A. Revisiting sorting for GPGPU stream architectures. Tech. Rep. CS2010-03, U of Virginia, Dept of Computer Science, Charlottesville, VA, USA, 2010.

[82] Merrill, D. G., and Grimshaw, A. S. Revisiting sorting for GPGPU stream architectures. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2010), PACT '10, ACM, pp. 545–546.

[83] Micikevicius, P. 3D finite difference computation on GPUs using CUDA. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units* (New York, NY, USA, 2009), ACM, pp. 79–84.

[84] Miller, M, I., and Younes, L. Group action, diffeomorphism and matching: A general framework. *Int. J. Comp. Vis 41* (2001), 61–84.

[85] Miyazaki, R., Yoshida, S., Nishita, T., and Dobashi, Y. A method for modeling clouds based on atmospheric fluid dynamics. In *Proceedings of the 9th Pacific Conference on Computer Graphics and Applications* (Washington, DC, USA, 2001), PG '01, IEEE Computer Society, pp. 363–.

[86] Moore, G. E. Cramming more components onto integrated circuits. *Electronics 38*, 8 (April 1965), 114–117.

[87] Mowry, T., Demke, A., and Krieger, O. Automatic compiler-inserted i/o prefetching for out-of-core applications. *ACM SIGOPS Operating Systems Review 30*, si (1996), 3–17.

[88] Musser, D. R., and Stepanov, A. A. Generic programming. In *Proceedings of the International Symposium ISSAC'88 on Symbolic and Algebraic Computation* (London, UK, 1989), ISAAC '88, Springer-Verlag, pp. 13–25.

[89] Nguyen, H. *GPU Gems 3*, first ed. Addison-Wesley Professional, NY, USA, 2007.

[90] Nickolls, J., Buck, I., Garland, M., and Skadron, K. Scalable parallel programming with cuda. *Queue 6* (March 2008), 40–53.

[91] NVIDA . *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, October 2010.

[92] NVIDIA. CUDA community show case, May 2008.

[93] NVIDIA. CUDA technical training. Tech. rep., NVIDIA Corporation, 2009.

[94] NVIDIA. NVIDIA Performance Primitives, 2009. Version 1.0.

[95] Oldfield, R., and Kotz, D. Applications of parallel i/o. Tech. rep., Dartmouth College, Hanover, NH, USA, 1998.

[96] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. GPU computing. *Proceedings of the IEEE 96*, 5 (May 2008), 879–899.

[97] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26*, 1 (Mar. 2007), 80–113.

[98] P. Lorenzen, B. Davis, S. J. Unbiased atlas formation via large deformations metric mapping. In *MICCAI* (2005).

[99] Patterson, D. A., and Hennessy, J. L. *Computer Organization and Design, The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*, 4th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[100] Peters, H., Schulz-Hildebrandt, O., and Luttenberger, N. Fast comparison-based in-place sorting with CUDA. Tech. rep., Christian-Albrechts-University Kiel, German, 2009.

[101] Pharr, M., and Fernando, R. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, Reading, MA, USA, 2005.

[102] Prastawa, M., Gilmore, J. H., Lin, W., and Gerig, G. Automatic segmentation of mr images of the developing newborn brain. *MedIA 9*, 5 (2005), 457–466.

[103] Regnier, G., Minturn, D., McAlpine, G., Saletore, V. A., and Foong, A. Eta: Experience with an intel xeon processor as a packet processing engine. *IEEE Micro 24* (January 2004), 24–31.

[104] Rodrigues, C. I., Hardy, D. J., Stone, J. E., Schulten, K., and Hwu, W.-M. W. Gpu acceleration of cutoff pair potentials for molecular modeling applications. In *Proceedings of the 5th Conference on Computing Frontiers* (New York, NY, USA, 2008), CF '08, ACM, pp. 273–282.

[105] Rueckert, D., Sonoda, L., Hayes, C., Hill, D., Leach, M., and Hawkes, D. Nonrigid registration using free-form deformations. *IEEE Trans Med Imag 18*, 8 (1999), 712–721.

[106] Satish, N., Harris, M., and Garland, M. Designing efficient sorting algorithms for manycore GPUs. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel Distributed Processing* (2009), IEEE Computer Society, pp. 1–10.

[107] Scheidegger, C. E., Schreiner, J. M., Duffy, B., Carr, H., and Silva, C. T. Revisiting histograms and isosurface statistics. *IEEE TVCG 14*, 6 (2008), 1659–1666.

[108] Sengupta, S., Harris, M., Zhang, Y., and Owens, J. D. Scan primitives for GPU computing. In *Graphics Hardware 2007* (Aug. 2007), ACM, pp. 97–106.

[109] Sintorn, E., and Assarsson, U. Fast parallel GPU-sorting using a hybrid algorithm. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)* (2007).

[110] Smith, S. W. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, San Diego, CA, USA, 1997.

[111] Snavely, N., Garg, R., Seitz, S. M., and Szeliski, R. Finding paths through the world's photos. *ACM Trans. Graph. 27* (August 2008), 15:1–15:11.

[112] Springel, V., White, S., Jenkins, A., Frenk, C., Yoshida, N., Gao, L., Navarro, J., Thacker, R., Croton, D., Helly, J., Peacock, J., Cole, S., Thomas, P., Couchman, H., Evrard, A., Colberg, J., and Pearce, F. Simulations of the formation, evolution and clustering of galaxies and quasars. *Nature 435*, 7042 (Jun 2005), 629–636. 10.1038/nature03597.

[113] Standards", O. Draft technical report on c++ library extensions. Tech. rep., Open Standard Technical Report, June 2006.

[114] Steen, A. J. V. D. Overview of recent supercomputers, 2010.

[115] Stewart, E. *Intel Integrated Performance Primitives: How to Optimize Software Applications Using Intel IPP.* Intel Press, Santa Clara, CA, USA, 2004.

[116] Stone, J., Phillips, J., Freddolino, P., Hardy, D., Trabuco, L., and Schulten, K. Accelerating molecular modeling applications with graphics processors. *J. Comput. Chem. 28*, 16 (2007), 2618–2640.

[117] Stone, J. E., Gohara, D., and Shi, G. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering 12* (2010), 66–73.

[118] Stroustrup, B. *The C++ Programming Language*, 3rd ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[119] Trendall, C., and Stewart, A. J. General calculations using graphics hardware with applications to interactive caustics. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000* (London, UK, 2000), Springer-Verlag, pp. 287–298.

[120] Trouvé, A., and Younes, L. Metamorphoses through lie group action. *Foundations of Computational Mathematics 5*, 2 (2005), 173–198.

[121] Ungerer, T., Robič, B., and Šilc, J. A survey of processors with explicit multithreading. *ACM Comput. Surv. 35* (March 2003), 29–63.

[122] Van Leemput, K., Maes, F., Vandermeulen, D., and Suetens, P. Automated model-based tissue classification of MR images of the brain. *IEEE Trans. Medical Imaging 18* (October 1999), 897–908.

[123] Vitter, J. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys (CSUR) 33*, 2 (2001), 209–271.

[124] Warfield, S. K., Kaus, M., Jolesz, F. A., and Kikinis, R. Adaptive, template moderated, spatially varying statistical classification. *MedIA 4*, 1 (Mar 2000), 43–55.

[125] Waterman, M., Arratia, R., and Galas, D. Pattern recognition in several sequences: Consensus and alignment. *Bulletin of Mathematical Biology 46*, 4 (1984), 515 – 527.

[126] Womble, D., Greenberg, D., Riesen, R., and Wheat, S. Out of core, out of mind: Practical parallel i/o. *Scalable Parallel Libraries Conference, 1993., Proceedings of the* (2002), 10–16.

[127] Xue, H., Srinivasan, L., Jiang, S., Rutherford, M. A., Edwards, A. D., Rueckert, D., and Hajnal, J. V. Longitudinal cortical registration for developing neonates. In *MICCAI* (2007), pp. 127–135.

[128] Young, D. *Iterative Solution of Large Linear Systems.* Academic Press, New York, NY, USA, 1997.

[129] Zomaya, A. Y. *Parallel Computing for Bioinformatics and Computational Biology (Wiley Series on Parallel and Distributed Computing).* Wiley-Interscience, NY, USA, 2005.