High Performance Reconfigurable Computing for Science and Engineering Applications

by

Peter Leonard McMahon

Submitted to the Department of Electrical Engineering
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Electrical and Computer Engineering

at the

University of Cape Town

October 2006

Advisor: Professor Michael R. Inggs

Abstract

This thesis investigates the feasibility of using reconfigurable computers for scientific applications. We review recent developments in reconfigurable high performance computing. We then present designs and implementation details of various scientific applications that we developed for the SRC-6 reconfigurable computer. We present performance measurements and analysis of the results obtained.

We chose a selection of applications from bioinformatics, physics and financial mathematics, including automatic docking of molecular models into electron density maps, lattice gas fluid dynamics simulations, edge detection in images and Monte Carlo options pricing simulations.

We conclude that reconfigurable computing is a maturing field that may provide considerable benefit to scientific applications in the future. At present the performance gains offered by reconfigurable computers are not sufficient to justify the expense of the systems, and the software development environment lacks the language features and library support that application developers require so that they can focus on developing correct software rather than on software infrastructure.

Contents

1 Introduction **1**

1.1 Background 1

1.2 Objectives 3

1.2.1 Investigate the State-of-the-art in Reconfigurable Computing 3

1.2.2 Implement Scientific Computing Algorithms on Reconfigurable Computers 4

1.2.3 Analyze the Performance of Scientific Applications on Reconfigurable Computers 4

1.2.4 Provide Guidance on the Methodology for Developing Software for Reconfigurable Computers 4

1.3 Motivation for Problems Studied 5

1.4 Thesis Outline and Summary 6

2 An Introduction to Reconfigurable Computing **10**

2.1 Reconfigurable Computing Hardware 11

2.1.1 Where do reconfigurable computers get their speed from? 12

2.2 Reconfigurable Computing Software 13

2.3 Measuring Performance in Reconfigurable Computing Systems 14

2.4 Conclusion 15

3 Monte Carlo Methods on Reconfigurable Computers **16**

3.1 Monte Carlo Methods 16

3.2 Monte Carlo Estimation of π 17

3.2.1 Implementation of a Parallel Pseudorandom Number Generator . . . 19
3.2.2 Design and Implementation of the Monte Carlo π Estimator . . . 25
3.2.3 Performance Results . . . 29
3.3 Monte Carlo Options Pricing . . . 32
3.3.1 Pricing Asian Options with Monte Carlo . . . 33
3.3.2 Generating Normal Random Variables . . . 36
3.3.3 Design and Implementation . . . 37
3.3.4 Performance Results . . . 43
3.4 Conclusion . . . 45

4 Cellular Automata Simulations on Reconfigurable Computers 48
4.1 An Introduction to Cellular Automata . . . 49
4.1.1 One-dimensional Cellular Automata . . . 49
4.1.2 Two-dimensional Cellular Automata . . . 51
4.2 Conway's Game of Life . . . 52
4.2.1 Design and Implementation . . . 53
4.2.2 Performance Results . . . 71
4.3 Fluid Dynamics Simulations using the Lattice Gas Method . . . 73
4.3.1 Design and Implementation . . . 75
4.3.2 Performance Results . . . 79
4.4 Conclusion . . . 79

5 Image Processing – Edge Detection on Reconfigurable Computers 81
5.1 An Introduction to the Sobel Edge Detection Algorithm . . . 81
5.2 Edge Detection on a Reconfigurable Computer . . . 84
5.2.1 Design and Implementation . . . 84
5.2.2 Results . . . 88
5.3 Conclusion . . . 90

6 Automatic Macromolecular Docking on Reconfigurable Computers 91
6.1 Macromolecular Docking using Global Correlation 93
6.1.1 Design and Implementation 94
6.1.2 Results 96
6.2 Conclusion 98

7 Conclusion 99
7.1 Results 99
7.2 Analysis 99

A Monte Carlo Methods 101
A.1 A Review of Monte Carlo Methods 101
A.1.1 An Early Monte Carlo Algorithm 101
A.1.2 Numerical Integration using Monte Carlo Methods . . 102
A.1.3 Monte Carlo, Beyond Simple Integration 104
A.2 A Review of Parallel Pseudorandom Number Generation . . . 105
A.2.1 Generating Random Numbers on Deterministic Computers 106
A.2.2 Parallel Pseudorandom Number Generation 108
A.2.3 Assessing the Quality of Pseudorandom Number Sequences 110

B The SRC-6 Reconfigurable Computer 114

List of Figures

1.1 A generic reconfigurable computer architecture. 2

2.1 Computational density of FPGAs and Intel microprocessors. Image from ref. [6]. 12
2.2 Measurement of processing time for an application running on a reconfigurable computer. Image from ref. [9]. 14

3.1 Scatter plot of (x, y) pairs randomly sampled from $[0, 1]^2$. . . . 18
3.2 Processing engines in the FPGA, each independently generating pseudorandom numbers. 20
3.3 Simulation of a set of 3 parallel pseudorandom number generators. 21
3.4 The operation of a single processing engine. 22
3.5 Top-level design for random number generators on two FPGAs. 23
3.6 Processing engines each performing N simulations to estimate π. 26
3.7 Percentage of FPGA resources used on a single FPGA, as a function of the number of processing engines included. 28
3.8 Performance of Monte Carlo estimation of π on an SRC-6 MAPe with 15 processing engines, using one FPGA, compared with that of an x86 processor. 30
3.9 Performance of Monte Carlo estimation of π on an SRC-6 MAPe with 30 processing engines, using two FPGAs, compared with that of an x86 processor. 31

3.10 Performance of Monte Carlo estimation of π on an SRC-6 MAPe, using one FPGA, as a function of the number of processing engines used. 32
3.11 Histogram, with 500 bins, of 200000 random numbers generated on the SRC-6 (blue), and 2000000 random numbers generated using MATLAB's `randn` function, scaled (red). . . . 37
3.12 Processing engines each generating N independent paths of stock price movements to simulate possible payoff scenarios. . 39
3.13 Performance of a Monte Carlo options pricing simulation on an SRC-6 MAPe, using one FPGA, showing the speed difference between unflattened loops and flattened loops. 44
3.14 Performance of a Monte Carlo options pricing simulation on an SRC-6 MAPe, compared with that of an x86 processor. . . 46

4.1 Examples of Birth, Survival and Death in the Game of Life. . 52
4.2 A grid representing a configuration in the Game of Life cellular automaton. 54
4.3 A grid decomposed into four subgrids, by dividing the grid vertically and horizontally. 55
4.4 A grid decomposed into five subgrids, by dividing the grid into horizontal strips. 55
4.5 A grid decomposed into five subgrids, showing the 'ghost regions' for each slice. 57
4.6 Sequence of processor communication to synchronize ghost regions, while avoiding deadlock. 59
4.7 Processing engine performing an N timestep simulation, using the design that all communication is done between the processing engines. 61
4.8 Processing engines performing an N timestep simulation. The program forks processing engines, then joins them once every iteration. 64

4.9 Layout and flow of the data in the system during a cellular automata simulation. 66
4.10 Performance of a Game of Life simulation on an SRC-6 MAPe, compared with that of an x86 processor. 72
4.11 Transition of an FHP lattice gas automata. Solid arrows show the configuration at the current timestep, and hollow arrows show the configuration at the next timestep. Image from Luo, ref. [37]. 74
4.12 Example transition rules for FHP lattice gas automata. Image from Luo, ref. [37]. 75
4.13 Layout and flow of the data in the system during a lattice gas automata simulation. 77
4.14 Performance of a Lattice Gas Automata simulation on an SRC-6 MAPe, with 5 processing engines, using one FPGA, compared with that of an x86 processor. 80

5.1 Processing engines each performing Sobel edge detection on a slice of the input image. 85
5.2 Layout and flow of the data in the system during edge detection. 87
5.3 Original image (left). Image with detected edges shown (right). 88
5.4 Caching pixels to reduce the number of reads from onboard memory. 90

6.1 A macromolecular model (yellow) docked in an electron microscope density map (blue mesh). Image from ref. [45]. . . . 92
6.2 Fitting a macromolecular model by rotation and translation. Image from ref. [46]. 93
6.3 An overview of the global correlation docking algorithm. Image from ref. [46]. 95
6.4 Processing engines each search to find a maximum correlation for the Euler angles they are assigned. 97

B.1 The SRC-6 in a rack at the National Center for Supercomputing Applications. 115
B.2 The architecture of the SRC MAP module. 115

List of Tables

3.1 Results of statistical tests for quality of a pseudorandom sequence generated by 15 processing engines. 24

3.2 Place and Route results for a two-FPGA, 30 processing engine Monte Carlo simulation to estimate π. 27

3.3 Place and Route results for a one-FPGA, one processing engine Monte Carlo options pricing simulation. 42

3.4 Place and Route results for a one-FPGA, two processing engine Monte Carlo options pricing simulation. 43

4.1 Place and Route results for a one-FPGA, four-processing engine Game of Life cellular automata simulation. 69

4.2 Place and Route results for a one-FPGA, eight-processing engine Game of Life cellular automata simulation. 70

4.3 Place and Route results for a one-FPGA, five-processing engine Lattice Gas Automata simulation. 78

5.1 Place and Route results for a one-FPGA, three processing engine edge detection program. 87

5.2 Performance of Sobel Edge Detection on an SRC-6 MAPe, with three processing engines, using one FPGA, compared with that of an x86 processor. 89

Acknowledgements

This thesis marks the culmination of my undergraduate career, and so I find this to be the perfect opportunity to acknowledge those individuals who have enriched my undergraduate experience, making it immeasurably more beneficial and exciting than it otherwise might have been. Of course by naming people, I run the risk of making serious omissions. If I do, I apologise and ask that you know that I am not any less grateful for your help over the years.

First and foremost, I am exceedingly grateful to my advisor, Professor Michael Inggs. He granted me an exceptional amount of independence so that I could pursue what I thought to be the most interesting and important problems, and in so doing forced me to see first-hand what the world of research is about. This academic freedom has allowed me to enjoy an excellent six months following my every intellectual whim, and being able to count most of it as work! I am thankful for his unreasonable confidence in a student he had barely met, and for his willingness to put his reputation (and funds!) on the line by recommending me to his collaborators and contacts, and then leaving me to 'go forth and do good things'. I can only hope that the final result, this thesis, is an apt repayment of his faith in me.

This thesis would not exist as it stands were it not for the help I have received from a large number of people. I would like to extend my gratitude to all the folk at the National Center for Supercomputing Applications at the University of Illinois for their hospitality and support. Dr Radha Nandkumar was instrumental in bringing me over to the United States for two months so that I could work with the NCSA's Innovative Systems Laboratory Reconfigurable Computing group, and so I am especially indebted to her. Thank you, Radha! I was apparently the first visitor in an experimental 'International Affiliates' program, and although I certainly gained more from the experience than I was able to give back to the NCSA, I hope they found my visit to be a beneficial exercise.

Dr Craig Steffen was my official host at the NCSA, and I would like to

thank him for the logistical, technical and intellectual support that he provided during my stay. Craig did far more than I ever could have reasonably imagined he would - from driving for 6 hours to Chicago to pick me up 'so that I wouldn't have to worry about catching the train', to making sure I had literally everything I needed to start working straight away. I spent many hours with Craig pestering him about technical matters, discussing approaches to problems and otherwise learning from him. He was an excellent mentor. Craig also had to put up with an enormous number of questions to satisfy my child-like curiosity about why things are the way they are in the U.S. - such as what the holes in electrical plug pins are for, why there is a big, slit rubber membrane in kitchen sinks, and why so many houses are built out of wood. Craig, thanks again for the lunches out and for periodic invites to your home for dinner. Thanks too, to Craig's wife Becky for sharing her home with me and for the cooking. I hope to repay the debt to both of you one day.

The ISL was an exceptionally friendly and productive group to work in. I had a lot of fun on the third floor - thanks, guys. Besides the joking around, I also learned a great deal from discussions I had with you all. Dr Volodymyr 'Vlad' Kindratenko and David Pointer certainly deserve special mention for their technical guidance and general willingness to help me whenever I was stuck or in uncharted waters. Both Vlad and David went out of their way to help with questions I had that they didn't have the answers to. This wasn't very often though - their technical knowledge was an inspiration, and a resource from which I benefited regularly. Thank you.

At the Reconfigurable Systems Summer Institute in Urbana in July, I met Dan Poznanovic and Dr Jeff Hammes, and would like to acknowledge helpful discussions with them. In particular, Jeff, who is a lead developer of the MAPC compiler at SRC, provided numerous useful pieces of advice and workarounds to compiler limitations.

I was able to investigate the diverse mix of academic domains in this thesis that I did thanks to the generosity of several researchers who individually

agreed to meet with me and give me 'crash courses' in their disciplines and work, with virtually no hope of any payoff. Dr Dan Jacobsen from the National Bioinformatics Network drove out to UCT to chat with me about his work in integrating biological databases. The staff at the Square Kilometer Array / Karoo Array Telescope offices in Pinelands kindly showed me round one afternoon, and gave me a useful overview of the projects and their needs. Dr Athol Kemball, an astronomer at the NCSA, offered helpful advice. Artur Szostak, Gareth de Vaux and Bruce Becker from the UCT-ALICE group spent hours explaining the functioning of the dimuon High Level Trigger in the ALICE project to me, and giving me advice on where I would be best advised to focus my efforts.

I am particularly grateful to Dr Zeblon Vilakazi from UCT-ALICE and iThemba Labs. Dr Vilakazi has spent a great deal of time discussing particle physics and the engineering behind detectors with me since I first approached him about his work on the ALICE project in 2004. I have learned a lot from him about what modern experimental physics is all about, even if his confidence in my ability as a dabbling physicist is far greater than my own! I also thoroughly enjoyed a short visit I had to CERN in June 2006 at Dr Vilakazi's invitation - it was certainly a memorable experience, and a much appreciated detour on my way to Illinois.

Dr Michelle Kuttel, besides her direct impact on this thesis, in the form of the work on atomic model docking in electron density maps and on cellular automata, has undoubtedly had a far more general influence as well. Dr Kuttel has been an exceptional mentor since I did a class project under her guidance in 2004. I have spent innumerable hours in her office discussing research in computer science and computational science, and academia and research in general. Besides stimulating intellectual conversations, Dr Kuttel has also given me numerous opportunities to develop - by allowing me to guest lecture in her Distributed Systems course, involving me in the Scientific Clustering Applications Workshops and allowing me to speak at the second SCAW. She has also patiently explained computational chemistry to

a neophyte and walked me through her atomic model docking work. In the CS2 project I took with her, I learned more about cellular automata that I otherwise might have, and had an exceptionally fun time building a CA lattice gas simulator with Shen Tian. This project, although not strictly research, was probably the closest I had come at the time.

Shen, besides being a terrific friend, was the best project partner I could ever ask or hope for. His technical brilliance is nearly unmatched, and he is a hard and dedicated worker (when he isn't distracted by Google or anime!). Our project in second year was undoubtedly the most exciting group project I've been involved in, and I benefited enormously from it, largely due to Shen's involvement. I have continued our work in this thesis, and it goes almost without saying that had I not undertaken the original lattice gas project with Shen, this would not have happened.

I would like to thank Justin Kelleher (now back in Ireland), Professor Andrew Hutchison and Professor Pieter Kritzinger from the Data Network Architectures group, as well as the various group students I've had interactions with, for involving me in the DNA activities, which gave me another perspective on what a research group does and how it goes about its business. Justin during his time at UCT was a great champion for me, and I had an excellent time discussing the state-of-the-art in software engineering with him. I would also like to especially thank Prof. Hutchison for giving me a glimpse into the security research field, and for presenting our paper at ISSA 2006 while I was away.

Professor Vasco Brattka, UCT's lone quantum computing theory expert, has taught me a tremendous amount about what theory research is like. His quantum computing course was truly superb - flawlessly prepared, with excellent notes. Prof. Brattka's deep insights into the mathematical foundations of quantum computing have been enlightening and inspiring. He has the enviable quality of never jumping to conclusions, and always insists on a well-thought-out argument before settling on any statement. I am grateful for his patiently sitting through descriptions of my hair-brained algorithm

ideas. His gentle encouragement to continue working and thinking despite my minimal advances were much appreciated.

Besides the abovementioned faculty, I have encountered several other lecturers who have made my time at UCT more enjoyable. Professor Martin Braae, who headed the Department of Electrical Engineering for my first three years here, and Deputy Dean Professor Barry Downing have both been very supportive of my efforts, never quibbling when I broke rules on curriculum changes or otherwise tried to jump over bureaucratic barriers. Professor Anthony Chan's seemingly unbreakable spirit and work ethic have been an inspiration to watch. In no particular order, Mr Alan Rynhoud, Mr Stephen Schrire, Professor Jonathan Tapson, Dr Fred Nicolls, Dr Andrew Wilkinson, Mr Samuel Ginsberg, Mr Simon Winberg, Professor Cathal Seoighe, Professor Raoul Viollier, Professor Sandro Perez, Dr Roger Fearick, Dr Gary Tupper and Professor Andy Buffler have all taught me at some stage, and were memorable for their depth of knowledge in their respective fields and the quality of their expositions. Mme. Ewa Swida-Reid provided an oft entertaining outlet for '*les ingénieurs*', in our solitary foray into the humanities.

Not entirely outside the academic realm, I would like to thank the editors at Varsity when I worked at the paper, Steven Kenyon, David Wilson and Cathryn Reece, and the rest of the staff, for letting this engineer pretend to be a journalist for a few hours each week.

Last but not least, my friends and family deserve my thanks for their support. In addition, the Smuts community has been excellent, and I'm especially grateful to my peers in the Electrical Engineering department for creating a friendly environment in which to work and learn, and for teaching me all that they have. Thanks finally to the Copeland tribe — Dean, Devin, Jason and Mike — for an excellent year, on top of those that went before.

Chapter 1

Introduction

This thesis investigates the feasibility of using reconfigurable computing technology for performing scientific computations. In this introduction, we provide a brief background and motivation for this investigation, provide details of the objectives of the thesis, and outline the contents of the thesis.

1.1 Background

The past four decades has seen an exponential rise in the speed of processors. Moore's Law, a prediction made by Gordon Moore of Intel in 1965, has proven surprisingly accurate — the number of transistors on processors has doubled nearly every 24 months since Moore's prediction that this would be the case. The ability to fabricate chips with more transistors has resulted in proportionate speed increases.

However, during the past 3 years, microprocessor manufacturers have been experiencing difficulty in dramatically increasing the performance of their processors. Clock speed increases have all but come to a halt due to limitations with present fabrication technology, which has resulted in manufacturers seeking alternative means to providing consumers with greater performance. Unfortunately no revolutionary architectural changes have been forthcoming, so the manufacturers have instead simply opted to build mul-

ticore chips.

Power consumption on present microprocessors has also become a considerable issue, besides the problems that power dissipation causes with attempts to increase performance. Large high performance computing centres often consume megawatts of power, leading to electricity being a considerable operating expense[1]. Thus there is also an economic motivation to investigate lower power technologies.

Field Programmable Gate Arrays (FPGAs) have long been used in digital signal processing (DSP) applications, where relatively simple algorithms, involving primarily integer operations, are performed on large quantities of data. Over the past decade, several projects have been initiated to investigate the use of FPGAs for general-purpose scientific computation [4, 5, 6]. This ideal led to its natural extension — the development of hybrid computational machines that use both traditional microprocessors and FPGAs. Such hybrid architectures, known as *reconfigurable computers* have recently been introduced commercially by vendors such as Cray, SRC and SGI.

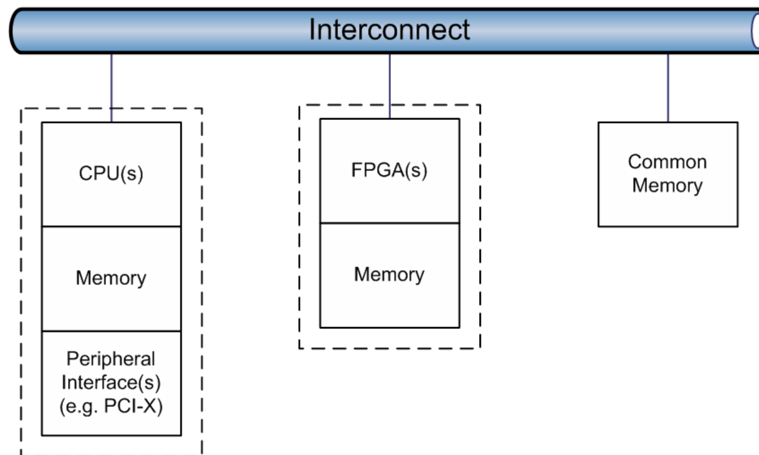

Figure 1.1: A generic reconfigurable computer architecture.

Successes with current reconfigurable computers have largely been lim-

[1]The National Center for Supercomputing Applications at the University of Illinois currently uses more than 3MW of power. The planned National Science Foundation petaflop supercomputer may require upwards of 20MW.

ited to specific signal processing applications or other specialized algorithms. However, it is certainly possible [6] that reconfigurable computers may become important technology for more general high performance computations, and in particular scientific simulation and computation.

1.2 Objectives

In this thesis we aimed to investigate the state-of-the-art in reconfigurable computing, and analyze the ability of current reconfigurable computing technology to perform scientific computations. Methodology for developing software for reconfigurable computers was also to be investigated.

These overall objectives are now described in more detail.

1.2.1 Investigate the State-of-the-art in Reconfigurable Computing

A sizeable number of academic and commercial projects have been attempted over the past decade, with the intention of furthering the use of FPGAs, and more generally reconfigurable computers, in computationally intensive environments outside of signal processing. Academic projects include SPLASH from the early 1990s [4], the MIT FPGA-based Cellular Automata Machine from the mid-1990s, the Berkeley Emulation Engine (BEE) and its successor, BEE2 [6], and Brigham Young University and Boston University's efforts, amongst others. Commercial offerings now include those from Cray[2], SGI, SRC, Nallatech and Linux Networx. Xilinx, the world's leading FPGA manufacturer, has also recently demonstrated its own compiler efforts for a high-level language for programming its FPGAs.

One of the broader aims of this thesis was to survey the current state-of-the-art in reconfigurable computing, to look for key differences between the

[2] The current understanding in the industry is that Cray is discontinuing its XD1 line of reconfigurable computers, but may licence to third parties the technology it acquired from Octigabay that led to the development of the Cray XD1.

available technologies, and provide a review of what a researcher new to the area needs to know.

1.2.2 Implement Scientific Computing Algorithms on Reconfigurable Computers

A core aim of the thesis is the implementation of several scientific computing codes on a reconfigurable computer. We aimed to identify several scientific computing problems that are computationally intensive, and of interest to researchers in South Africa, and rewrite them for execution on a reconfigurable computer. This, of course, implies the need to investigate how to develop software for reconfigurable computers.

The specific problems we chose to implement are: Monte Carlo Simulations; Cellular Automata Simulations; Edge Detection (Image Processing) and Macromolecular Docking.

1.2.3 Analyze the Performance of Scientific Applications on Reconfigurable Computers

For each problem implemented on a reconfigurable computer, we aimed to analyze the performance of the implementation on the reconfigurable versus its performance on a classic x86 microprocessor architecture. We also aimed to investigate how performance on reconfigurable computers scales. Finally, we aimed to analyze the performance and available resources to determine which aspects of current reconfigurable computers are performance bottlenecks.

1.2.4 Provide Guidance on the Methodology for Developing Software for Reconfigurable Computers

During the implementation of algorithms on a reconfigurable computer, we expected to find some effective method for parallelizing and porting code.

We also expected to encounter a series of technical difficulties, given the immature nature of the technology, and aimed to provide a brief reference in this thesis that gives our solutions to common problems one might encounter when porting code.

1.3 Motivation for Problems Studied

As we have mentioned, we implemented four different types of algorithm on reconfigurable computers during our study. These were Monte Carlo Simulations; Cellular Automata Simulations; Edge Detection (Image Processing) and Macromolecular Docking.

In selecting algorithms to implement, one of our broad objectives was to pick those that would be of interest to researchers in South Africa. The first three problems that we selected are very broad, and naturally satisfy this criterion: Monte Carlo simulations are used in a wide variety of scientific computing areas, including computational chemistry and physics, as well as financial mathematics. Cellular automata are less widely used than Monte Carlo simulations, but have equally broad application, as Wolfram [10] has demonstrated. Image processing algorithms, including edge detection, are also widely used in industry in South Africa, and there are several research groups in academia and industry that work extensively with such algorithms.

Macromolecular docking is a far more specific application than the preceding three. It is under active study as part of the National Bioinformatics Network research programme, and is an important research topic in experimental techniques for structural biology.

In selecting the algorithms to implement, we also aimed to pick a set of algorithms that were dissimilar, so that we could observe how different types of algorithms performed on reconfigurable computers. In our study of Monte Carlo simulations, we implemented a simple estimator of π that wasn't very floating-point calculation intensive, and a financial options pricing simulation, which was quite floating-point calculation intensive. Neither

of these applications were data intensive. The cellular automata models we implemented were both discrete, and did not involve any floating-point calculations. Relatively large amounts of data did, however, need to be handled. The edge detection algorithm was also data-intensive, and involved only integer arithmetic. Its advantage was that is performance on FPGAs and on microprocessors is fairly well known, so it provides a good basis for comparison. Finally, the docking problem involves a large amount of floating-point calculation, a significant amount of data processing, and is a fairly complicated algorithm. The complexity of the problem tests the feasibility of using current reconfigurable computing systems to speed up such large problems.

1.4 Thesis Outline and Summary

This thesis is organized in the following manner:

Chapter 2 provides an overview of the use of FPGAs for solving scientific computing problems, from SPLASH through to the current state-of-the-art in projects such as BEE2 and SRC's MAPstation. We briefly explore why FPGA-based computation may be more efficient than that with regular microprocessors, going into the benefits of spatial parallelism in FPGAs, and the higher computing intensity afforded by FPGAs because computing engines are hard-wired for the problem being solved.

Chapter 3 presents our development of Monte Carlo simulation applications on a reconfigurable computer. Specifically, we present an implementation of a parallel pseudorandom number generator, and its use in two Monte Carlo simulations: one that is an estimator of π, and a second that performs an options pricing simulation from financial mathematics.

We found that methods for parallelizing pseudorandom number generation that have been used on other high performance parallel architectures (such as the Percus-Kalos implementation for the NYU Ultracomputer [24],

and the *de facto* standard random number generation library for clusters, SPRNG [23]) can be applied to reconfigurable computers.

Our simulation to estimate π involves only a few floating-point calculations beyond the generation of random numbers, and our most optimized design and implementation on the SRC-6 MapStation was able to deliver a 6x speedup over the same code running for the same number of iterations on just the MAPstation's CPU (i.e. no FPGA acceleration). This 6x gain was achieved by running 30 simultaneous 'processing engines'[3] over two Virtex II Pro FPGAs.

Our options pricing simulation was necessarily far more floating-point calculation intensive, and was a more complicated algorithm, and performance suffered as a result. Only one engine could be fully fit in a single FPGA design, with two engines fitting and functioning correctly only if the clock frequency of the FPGA was reduced. Performance on the reconfigurable computer was 5x worse than that of the MAPstation's CPU for a single processing engine running on a single FPGA. We determined that even if timing requirements could be met without lowering the clock rate, the performance of the reconfigurable computing implementation, using two FPGAs, would only approach parity with the pure microprocessor implementation.

Chapter 4 presents our development of Cellular Automata simulations on a reconfigurable computer. We begin with a discussion of the design to implement Conway's Game of Life, a simple cellular automaton. Cellular automata have been successfully parallelized on cluster computers, and we show in this chapter that the data parallelization strategy can be successfully employed on a reconfigurable computer. We note that when implementing data parallelization, or indeed implement any data-intensive computation on a reconfigurable computer, the program should be carefully designed to allow

[3]A processing engine can be thought of as an individual part of an FPGA design that collectively, with other processing engines, provides the parallelism that is used to obtain speedups.

quick, concurrent access to the data by multiple processing engines.

A Game of Life simulation on a grid that could fit into FPGA Block RAM was speed up by a factor of 4x on the reconfigurable computer, using approximately 70% of the resources available on a single FPGA. Five processing engines were placed in this design. Compiler issues, elaborated on in this chapter, hindered development, but the performance results indicate that performance scales well with the number of processing engines.

The Game of Life codebase was used as a starting point to implementing the more complicated FHP-III [36] lattice gas automata simulation algorithm. Lattice gas automata models allow us to model fluid dynamics on a lattice using a discrete model. The state transitions are more complicated than for Conway's Game of Life — each lattice point has seven state parameters instead of just one — and hence significantly more logic is required to implement the cellular automata rules for lattice gas. Five processing engines were fitted onto a single FPGA, using up 57% of the FPGA's slices, and this implementation yielded a 1.7x speedup.

Chapter 5 presents our implementation of a simple edge detection algorithm, from the digital image processing domain. We note that the 2D convolution used in Sobel Edge Detection [41] can be implemented in a way that is quite similar in some respects to the implementation of the Game of Life cellular automaton. Our implementation of edge detection on a single image results in a 2.1x slowdown on a reconfigurable computer versus a standard microprocessor. However, there is a large overhead in loading the FPGA design and transferring data between the CPU and FPGA. When comparing just the compute time of a reconfigurable computer versus that of a microprocessor, we saw a 1.67x speedup, using three processing engines.

Chapter 6 presents our implementation of an algorithm for automatically docking macromolecular structure models into electron density maps acquired using cryo-electron microscopy [46]. We analyze the algorithm, and

devise a suitable parallelization scheme and partitioning of the algorithm between the CPU and FPGA. We provide an account of our efforts to port this fairly sizeable application code to the reconfigurable computing platform, and note the major pitfalls encountered.

Chapter 7 is the conclusion and reflects on the current state of reconfigurable computing hardware and software, with analysis done based on the investigations carried out during this thesis. We also comment on the present state of reconfigurable computing software tools, and look at where improvements to the compilers and libraries may result in a software stack that is usable to application scientists without in-depth hardware knowledge.

Chapter 2

An Introduction to Reconfigurable Computing

Reconfigurable computing is a broad area of study that involves the investigation of the use of hybrid FPGA-CPU architectures to speed up computationally intensive algorithms and problems [1]. The use of field programmable gate arrays (FPGAs)[1] in conjunction with microprocessors allows reconfigurable computers to offer much of the flexibility of a general-purpose computing architecture, but at the same time provide many of the performance benefits of having an algorithm implemented in a hard-wired chip.

In this chapter we provide a brief overview of the state-of-the-art in the field, as it presently stands. We can recommend Compton and Hauck's review article [1] to readers interested in finding out about the field in more depth.

Reconfigurable computing has existed as an active field of study since the early 1990's, although the idea of using reconfigurable hardware to build a high performance, general purpose[2] computing device has existed since

[1]FPGAs can be thought of at the most basic level as arrays of logic gates that can be rewired in software. This is a very large simplification though — modern FPGAs are highly complex devices that incorporate not only arrays of lookup tables and logic, but also onboard memory and in the Xilinx Virtex family, even Power PC microprocessor cores!

[2]Here 'general purpose' is used in the sense that the device is not designed for a specific

Estrin et.al. suggested it in 1963 [2].

2.1 Reconfigurable Computing Hardware

Reconfigurable computing hardware relies fundamentally on technology that already exists: FPGAs and microprocessors. The design of the hardware for a reconfigurable computing system is currently a process of selecting which FPGAs and microprocessors to use, and designing an interconnect to join them. The term ‘reconfigurable computer’ is sometimes used for systems that barely even involve microprocessors, and are just collections of FPGAs, such the Berkeley Emulation Engine [3].

The pioneering reconfigurable computing hardware projects were SPLASH [4] and SPLASH 2 [5]. These projects showed the feasibility of building general purpose computing platforms using FPGAs, coupled with microprocessors. One of the major impediments to the adoption of reconfigurable computers is not the hardware, but rather the software environment. Although the present hardware is somewhat lacking in its capability to support scientific computations, as we will see later in this thesis, one of the motivating factors behind reconfigurable computing is that FPGA hardware is predicted to improve far faster than microprocessor technology [6]. Figure 2.1 shows how the computational density of FPGAs has been growing compared with that of Intel microprocessors.

---

application alone, as is the case with an ASIC, but rather that it can adequately run any number of different programs. ‘General purpose’ here does not mean that the device is likely to find use in the average computing scenario.

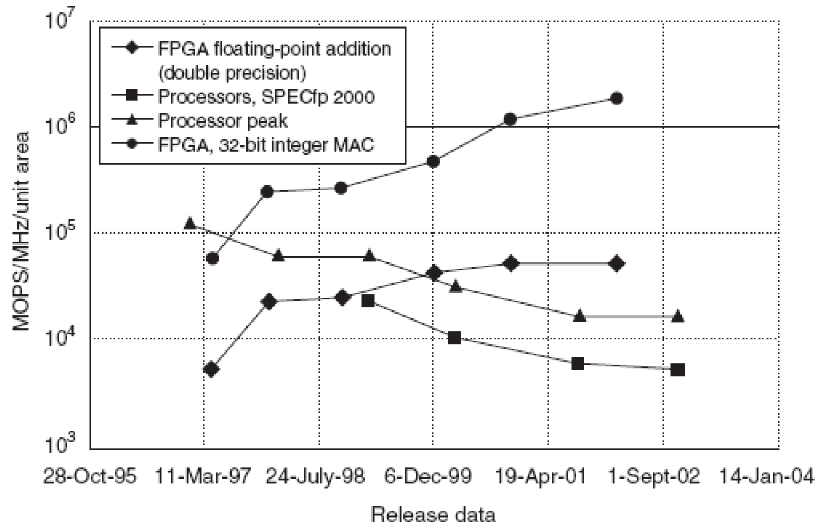

Figure 2.1: Computational density of FPGAs and Intel microprocessors. Image from ref. [6].

2.1.1 Where do reconfigurable computers get their speed from?

FPGAs typically run at clock speeds that are an order of magnitude, or more, slower than high-end microprocessors[3]. At first glance it seems counterintuitive that FPGA-based devices can offer a computational advantage over microprocessors.

A reconfigurable computer can obtain a speed advantage over a microprocessor system primarily due to three factors:

1. **Intensity**. CPUs can, at best, perform an integer operation every two clock cycles [7]. This is in the best case, when there are no cache issues, the pipeline is working as it should, and so on. The worst case is significantly worse. FPGAs only need to implement the functionality that is needed for the particular aplication at hand, which results in

[3]For example, in the SRC-6 reconfigurable computer, the FPGAs are clocked at 100MHz, and the microprocessor in the system is clocked at 2.8GHz.

far less complicated logic (for example, there is no need for a Control Unit, fully-functional ALU, etc.).

2. **Low latency**. Memory on and near the FPGA can be accessed in just a few clock cycles, and with an FPGA you have a far more fine-grained control over where your data in memory is located than you do with a CPU. The low latency of memory access is one of the contributing factors to increased intensity, since the FPGA spends less time waiting for data during computations, on average, than a CPU will.

3. **Spatial parallelism**. We can generate one small, special-purpose pipeline for performing a particular computation, and then replicate it across the FPGA chip.

When designing programs for reconfigurable computers it is important to consider these factors, to make sure that a design exploits the advantages of FPGAs where possible.

2.2 Reconfigurable Computing Software

The principle drawback of using FPGAs for general-purpose computation has traditionally been the lack of software support. Design tools for FPGAs have been created with hardware design engineers in mind, not software application developers. However, in the past decade a large amount of work has been done on developing compiler techniques to convert high level language code (such as C code) into efficient hardware description language code.

In 1995, Gokhale and Schott [8] ported the *data parallel C* language to the SPLASH reconfigurable platform. A variety of research projects have since started that have investigated how to translate high level language code that is typically written with a serial architecture in mind to the intrinsically parallel hardware description languages VHDL and Verilog.

Several high level languages and compilers are now available from industry vendors, many of them as offshoots of the initial academic efforts.

They include SRC's MAPC language and compiler, Handel C, Impulse C and Mitrion C, amongst others.

In this thesis we develop software primarily for the SRC-6 reconfigurable computer using the MAPC language and Carte development environment.

2.3 Measuring Performance in Reconfigurable Computing Systems

The measurement of performance of a reconfigurable computing system running a particular software program does not yet have a standard interpretation. This leads the performance results that various groups and vendors release to be non-comparable.

The problem arises because the total amount of time taken to run an application on a reconfigurable computer is not the same as the amount of time it takes for the computation to complete on the FPGA. Figure 2.2 illustrates this.

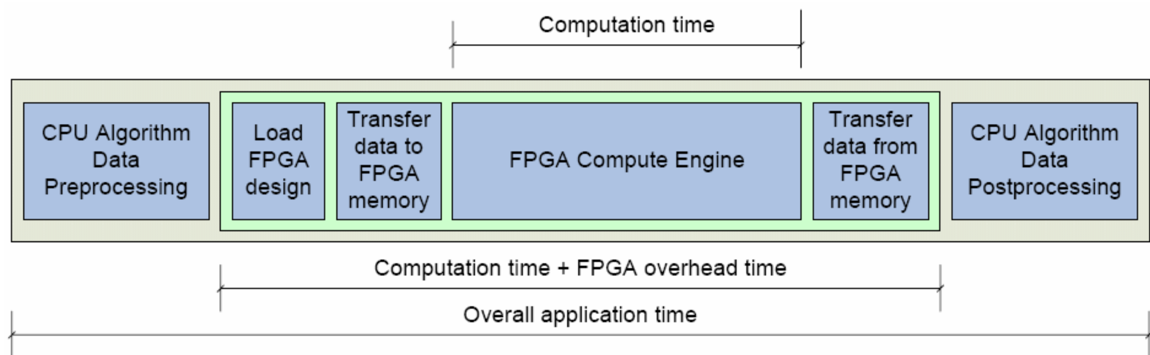

Figure 2.2: Measurement of processing time for an application running on a reconfigurable computer. Image from ref. [9].

For short computations, the amount of time it takes to load the FPGA design and transfer data into the FPGA, and then transfer data back to the CPU, may be non-negligible. In this thesis we always report the overall application time unless otherwise noted.

2.4 Conclusion

In this chapter we have provided an overview of what reconfigurable computing is, where it has come from and why it shows promise for the future. We have also explained why reconfigurable computers can offer performance advantages over traditional microprocessor architectures for some problems. We have mentioned the brief history of software technology for reconfigurable computers, and have explained how we measure the performance of reconfigurable computing applications throughout this thesis.

Chapter 3

Monte Carlo Methods on Reconfigurable Computers

Monte Carlo methods have been an extremely useful and important tool for computational scientists since the birth of computational science in the 1940's and 1950's [11]. Today Monte Carlo methods are used to solve computational problems in physics, biology, chemistry, finance and operations [12], amongst other fields.

In Appendix A we provide a self-contained review of Monte Carlo methods that includes all the material required to follow what we have done in our investigations of Monte Carlo methods on reconfigurable computers.

In this chapter we present our development of two Monte Carlo simulation applications for the SRC-6 MapStation reconfigurable computer. We also report our performance results, with comparisons to performance on a regular microprocessor architecture.

3.1 Monte Carlo Methods

Monte Carlo methods provide a computational complexity advantage over deterministic numerical methods by sampling randomly from a state space R of dimension d, whereas a deterministic method will typically sample uni-

formly from R, and the number of samples it requires will necessarily grow with the dimension d [15]. The simplest expression of this can be seen in the evaluation of the following integral:

$$I = \int_R f(\mathbf{x}) d\mathbf{x},$$

Monte Carlo integration will draw random samples $\mathbf{x}^{(1)}, ..., \mathbf{x}^{(N)}$ from R, and from these an estimate of I, $\widehat{I}$ can be made:

$$\widehat{I} = \frac{1}{N}\{f(\mathbf{x}^{(1)}) + \cdots + f(\mathbf{x}^{(N)})\} = \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}^{(i)})$$

Now in the limit $N \to \infty$, as a result of the Law of Large Numbers [12], $\widehat{I} \to I$:

$$\lim_{N\to\infty} \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}^{(i)}) = I.$$

The error in the estimate $\widehat{I}$ is $\mathcal{O}(\frac{1}{\sqrt{N}})$. i.e. The error depends only on the number of samples drawn, not on the dimension of the region that was being sampled from. The error in estimates using deterministic methods does depend on d, so high-dimensional problems are often best solved using Monte Carlo methods.

The essence of Monte Carlo methods is the sampling of values from a uniform distribution. Implementation of Monte Carlo methods clearly relies on the ability of a program to generate random numbers from a uniform distribution, and then perform simple arithmetic operations on them.

3.2 Monte Carlo Estimation of π

Our first Monte Carlo simulation is merely a proof-of-concept: we aimed to build a simulation that would estimate the value of π. This is a standard example of Monte Carlo that is routinely used to introduce the required concepts, which avoids the need for any specialist domain knowledge. The

Monte Carlo π estimator, however, requires all the infrastructure that is needed to tackle more complex problems.

We note that if we draw (x, y) pairs randomly from $[0, 1]^2$, then there is a relationship between the values drawn and the value of π. Specifically, if we consider each (x, y) pair as a point on a cartesian plane, and we draw a quarter circle on the plane (with the circle's origin at $(0, 0)$, and radius 1), then the ratio of the points that have fallen inside the quarter circle to the total number of points drawn (when this number is large) is related to the value of π as follows:

$$\frac{H}{N} = \frac{\pi}{4},$$

where N is the total number of pairs drawn, and H is the number of pairs (x, y) where $x^2 + y^2 < 1$.

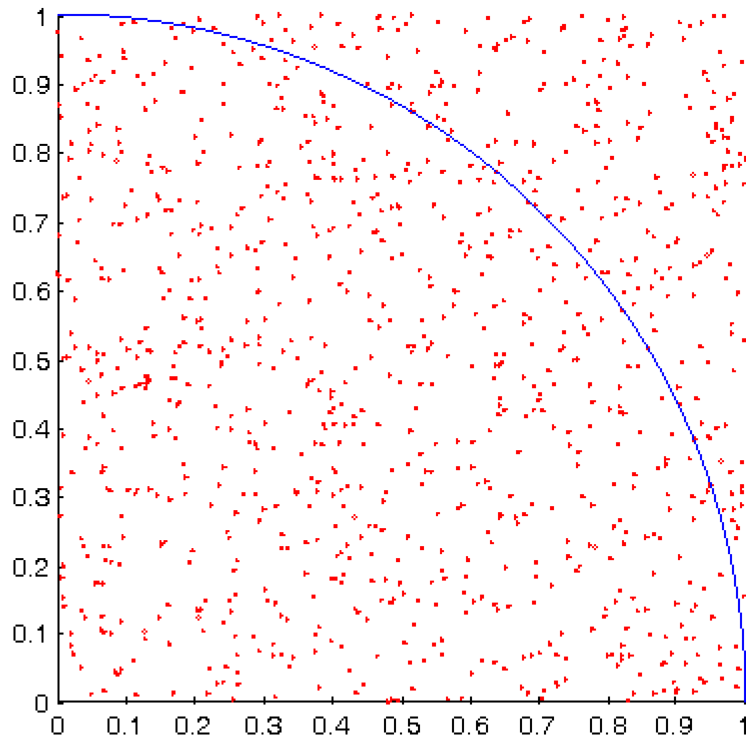

Figure 3.1: Scatter plot of (x, y) pairs randomly sampled from $[0, 1]^2$.

Given this knowledge, we can form an estimator of π as follows:

$$\widehat{\pi} = 4 \cdot \frac{H}{N}.$$

If we let $N \to \infty$, then $\widehat{\pi} \to \pi$. Thus in order to estimate the value of π, we simply need to draw a large number of samples from $[0,1]^2$ and count the number H that fall within the quarter circle with radius 1.

3.2.1 Implementation of a Parallel Pseudorandom Number Generator

Clearly we need to have a way to generate random samples from $[0,1]$. More specifically, we need to be able to generate uncorrelated samples from $[0,1]$ in parallel. That is, we would like to have multiple 'streams' of random numbers being generated in parallel.

Appendix A provides an overview of the theory of parallel pseudorandom number generation. We use the fact that it is possible to generate numbers on a deterministic machine that can pass statistical tests for randomness, even though by definition such numbers cannot be random. In our implementation, we use one of the oldest and most popular pseudorandom number generators, the linear congruential generator (LCG) [21]. Furthermore, we implement a parallelisation of the LCG that is implemented in the cluster computing random number generation library, SPRNG [23].

We wish to generate a sequence of random numbers $(X_n)_{n \in \mathbb{N}}$. A linear congruential generator is defined by the recurrence relation

$$X_n = (aX_{n-1} + b) \mod m$$

and the parameters X_0 (the *seed*), the multiplicative constant a, the additive constant b and the maximum period m. If m is a power-of-two, then we can define a set of additive constants $\left\{b^{(i)}\right\}$, where $b^{(i)}$ are pairwise relatively prime, so that if $i = 1, \ldots, N$, then we can generate N independent sequences indexed by i, $\left(X_n^{(i)}\right)_{n \in \mathbb{N}}$ where

$$X_n^{(i)} = \left(aX_{n-1}^{(i)} + b^{(i)}\right) \mod m.$$

Design and Implementation

We designed a program that implements these recurrence relations in parallel on an FPGA. Each recurrence relation, with its particular parameter $b^{(i)}$, is implemented as a separate piece of logic in the FPGA, and operates independently of the others. This is shown in Figure 3.2.

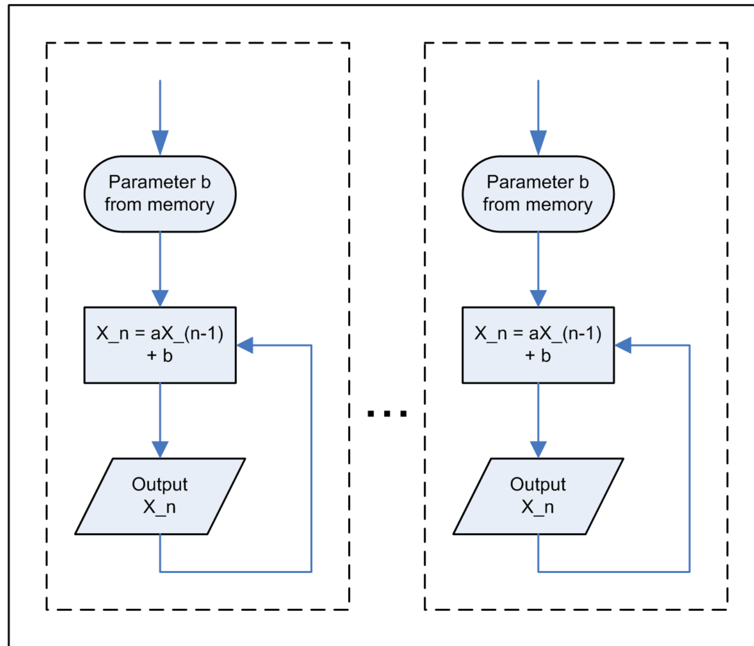

Figure 3.2: Processing engines in the FPGA, each independently generating pseudorandom numbers.

We implemented this setup both on the SRC-6 MapStation using the Carte environment, and with Mitrion-C. The Mitrion IDE provides a full clock-accurate simulator. Figure 3.3 shows the data dependency graph during simulation for three parallel generators, each running 10000 iterations. In this particular instance, the seed X_0 is set to 3157 for all three engines.

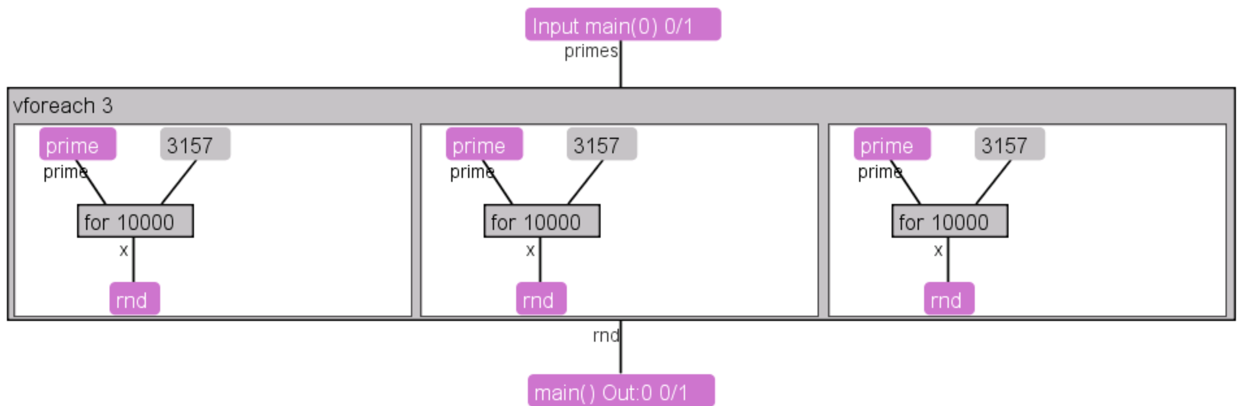

Figure 3.3: Simulation of a set of 3 parallel pseudorandom number generators.

Each processing engine has a *for* loop, which is implementing the LCG recurrence relation, as shown in the detail in figure 3.4.

Here $m = 2^{64} = 18446744073709551616$, and the variable x stores the value corresponding to X_n in the recurrence relation.

We implemented the parallel pseudorandom number generator on the SRC-6 MAPe module, which contains two Xilinx Virtex II Pro FPGAs. Each processing engine is created in MAPC code using the `#pragma src parallel sections` directive. After successfully implementing 15 processing engines on the first FPGA, we modified our design to use both FPGAs. Figure 3.5 shows the top-level view of the design. Since DMA memory transfers can only be made from CPU memory into the primary FPGA, we pass all 30 $b^{(i)}$ parameters to the first FPGA, which then transfers the 15 parameters needed by the 15 processing engines in the secondary FPGA over the bridge between the two FPGAs. Since we wish to reduce the number of clock cycles required per loop iteration as much as possible, the parameters in onboard memory are moved to static variables (which are implemented in the FPGA fabric). The reason for doing this is twofold — firstly, we would like each processing engine to operate independently, and having each engine access the onboard memory every iteration will result in contention for the memory, forcing some engines to wait while one reads its value from memory. Secondly,

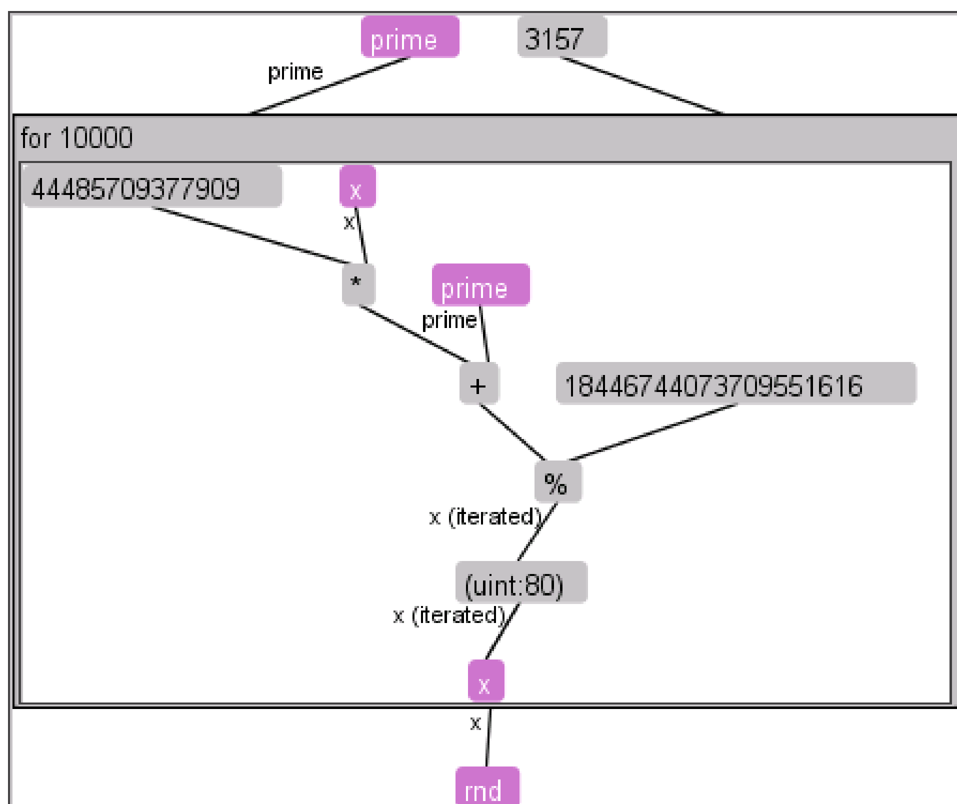

Figure 3.4: The operation of a single processing engine.

reads to onboard memory (and indeed even Block RAM) are relatively costly. However, reads to static variables implemented in the fabric require only 1 clock cycle. Thus by storing the parameters in static variables, we reduce the latency as much as possible.

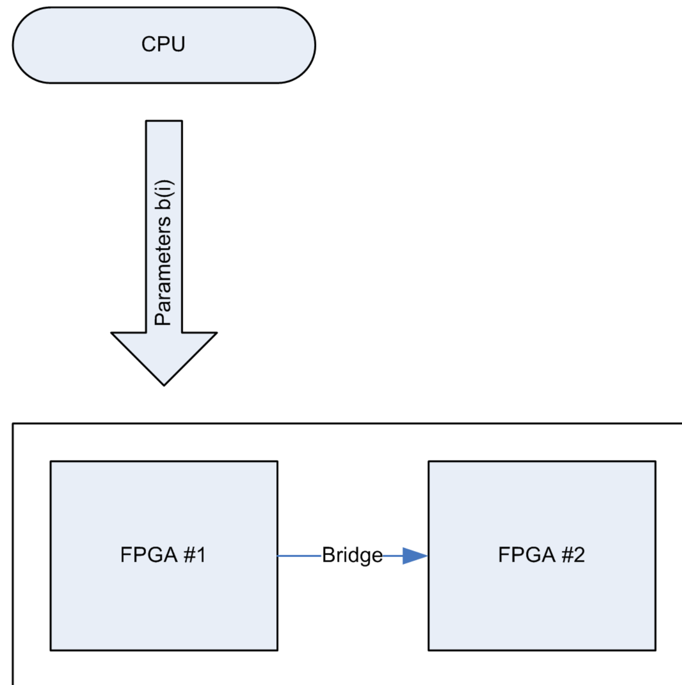

Figure 3.5: Top-level design for random number generators on two FPGAs.

Since we are just generating random numbers and there are no results being computed, we have nothing to send back to the CPU. However, when we implement a Monte Carlo simulation that uses this PRNG design as its basis, we will be able to transfer results from the second FPGA to the first over the bridge, and from the first FPGA to the CPU using a DMA transfer.

Results

In addition to the MAPC implementation, we created a standard C implementation of the random number generator for execution on an x86 CPU. We used this implementation to check the correctness of the MAPC imple-

Table 3.1: Results of statistical tests for quality of a pseudorandom sequence generated by 15 processing engines.

Test	Result
Entropy	7.999704 bits per byte
χ^2 value	1231.02
χ^2 test p-value	0.0001[a]
Arithmetic Mean	127.4234
Error in estimation of π	0.17%
Serial Correlation Coefficient	0.017665

[a] This means that the χ^2 value would be exceeded randomly 0.01% of the time.

mentation, by executing both and verifying that both produced identical results.

A high quality random number sequence is essential for Monte Carlo algorithms, so we tested to make sure that the sequences we generated passed a set of standard tests for randomness. Specifically, we used Walker's *Ent* package [20], which implements Knuth's tests [21] for randomness. An overview of the methods for assessing the quality of pseudorandom number sequences is available in Appendix A, which includes discussion of Knuth's serial correlation, χ^2 and entropy tests.

A sequence of 3000000 bytes, generated using 15 processing engines (i.e. 15 parallel streams), was analyzed with Ent, and the results are shown in Table 3.1. These results indicate a very favourable analysis of the randomness of the data generated. More information on what each test does, and how to interpret the results, is given in Appendix A.

3.2.2 Design and Implementation of the Monte Carlo π Estimator

With a parallel pseudorandom number generator implemented and ready to be used, it is fairly easy to extend the design to implement a Monte Carlo simulation that estimates π.

As we have already discussed, the algorithm idea is to draw random numbers from $[0,1]^2$, and then determine the number of points drawn that fall within the unit quarter circle. We can do this by drawing a value x from $[0,1]$, and y from $[0,1]$.

Our pseudorandom number generator generates integers between 0 and m, where m is the period of the generator. Thus we see that we can generate a floating-point number between 0 and 1 simply by taking a random number $X_n \in \{0,1,\ldots,m\}$ and dividing it by the period. Thus we have $x = X_n/m$. We can then use $y = X_{n+1}/m$.

We need to maintain a count of how many (x, y) pairs satisfy the condition $x^2 + y^2 < 1$. We call a pair that satisfies this condition a 'hit'. To parallelize the problem we can make each processing engine draw pairs in parallel. Each processing engine must maintain a count of how many of the pairs it drew were hits. Figure 3.6 shows the flow diagrams of the algorithms implemented in the processing engines. Once all p processing engines have completed a run of a set number of simulations, say N, then the count of hits from each engines can be summed. If we call this sum H, then $\widehat{\pi} = 4 \cdot \frac{H}{pN}$.

We implemented both a single-FPGA 15 processing engine version, and a 30 processing engine version of the simulation, which spanned two FPGAs. The two-FPGA version used the same architecture as the two-FPGA random number generator (see Figure 3.5), and the bridge between the two FPGAs was used to transfer the hits counts from the processing engines on the secondary FPGA to the primary FPGA[1]. Table 3.2 shows the FPGA resources used on the primary and secondary FPGAs in the 30 processing

[1]More specifically, the hits counts on the secondary FPGA were summed, and this sum was transferred to the primary FPGA, which then added this subtotal to its own.

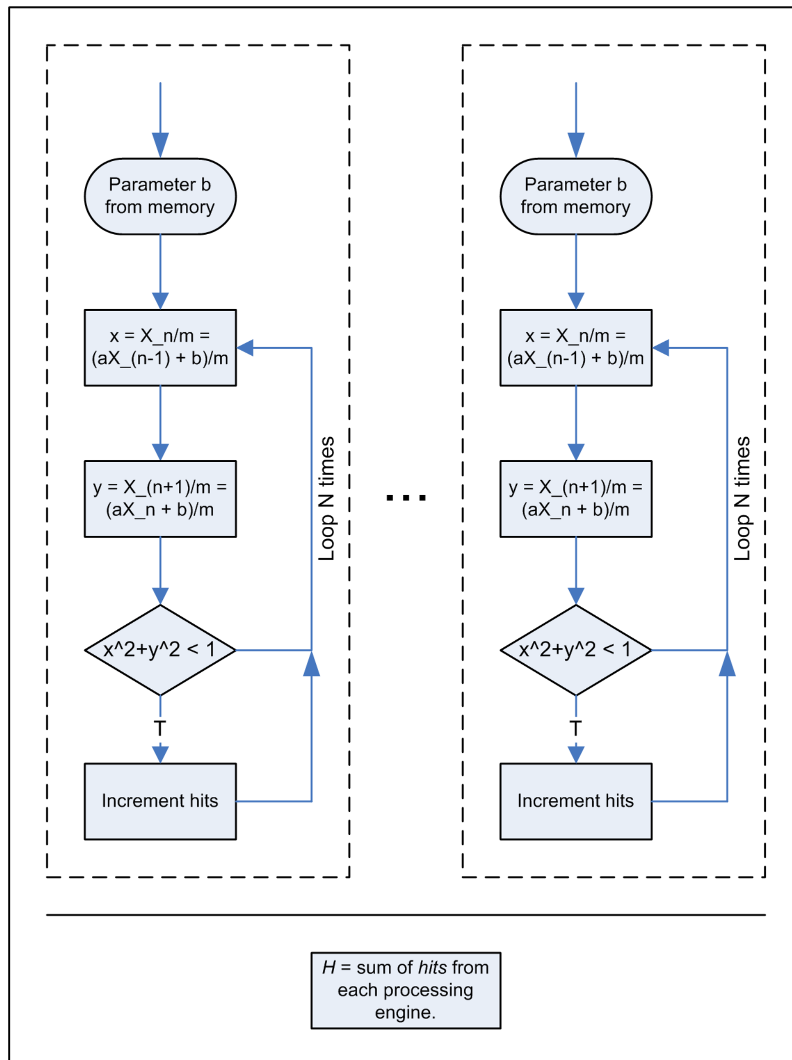

Figure 3.6: Processing engines each performing N simulations to estimate π.

Table 3.2: Place and Route results for a two-FPGA, 30 processing engine Monte Carlo simulation to estimate π.

Primary FPGA			
Resource	Used	Available	% Used
Slice Flip Flops	50,152	88,192	56%
4 input Lookup Tables	35,354	88,192	40%
Occupied Slices	35,794	44,096	81%
Hardware Multipliers	216	444	48%
Secondary FPGA			
Resource	Used	Available	% Used
Slice Flip Flops	44,180	88,192	50%
4 input Lookup Tables	34,659	88,192	39%
Occupied Slices	32,899	44,096	74%
Hardware Multipliers	144	444	32%

engine version.

Figure 3.7 shows the FPGA resources used as we scale up the number of processing engines. We see that the required number of hardware multipliers scales exactly linearly with the number of processing engines used. We expect this, since multipliers shouldn't ordinarily be need for 'infrastructure', and should be purely dependent on the number of multiplications done in the code. The other resources scale almost linearly with the number of processing engines used. The relationship is not perfectly linear since some of the resources are used in infrastructure for getting data into and out of the FPGA, and so on. Thus the number of slices used when two processing engines are placed is less than double the number used when just one processing engine is placed. Another factor to consider is that the Xilinx Place and Route tool doesn't necessarily generate an optimal solution, and if there is suboptimal use of resources, then we can't expect exact linear scaling.

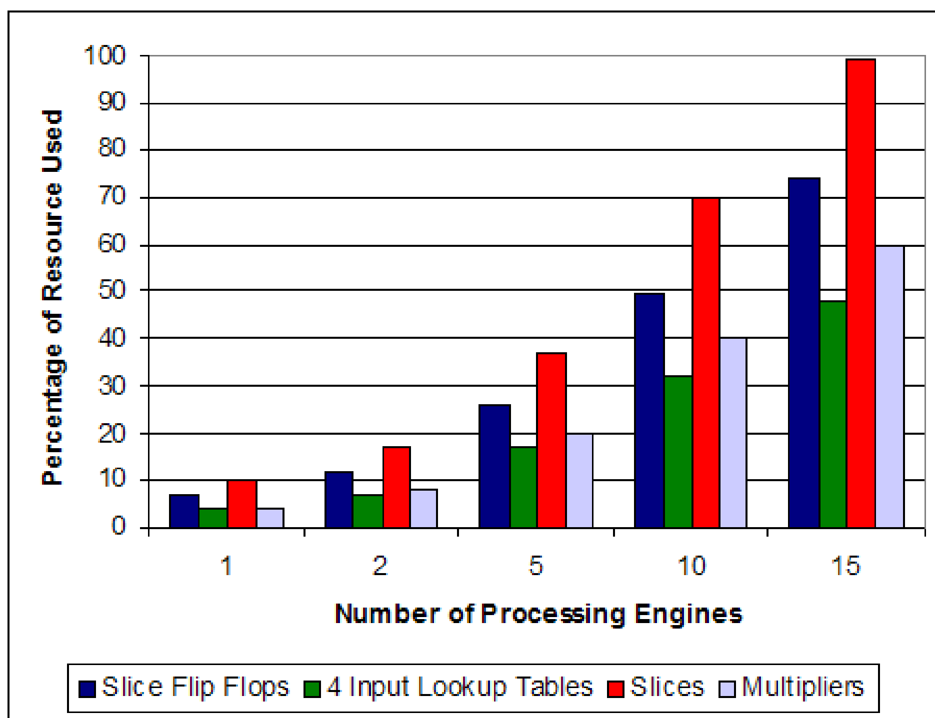

Figure 3.7: Percentage of FPGA resources used on a single FPGA, as a function of the number of processing engines included.

3.2.3 Performance Results

We implemented a functionally identical copy of the π estimator program for the SRC-6 in standard C, for execution on an x86 processor, so that performance could be compared. In this section, we present the performance results from both the x86 implementation, and the SRC-6 implementation. We also show how the performance of the SRC-6 implementation scales with the number of processing engines used.

Figure 3.8 shows the speedup obtained using the SRC-6 over a conventional x86 microprocessor architecture for the Monte Carlo simulation, using 15 processing engines. Here we define the speedup s as: $s = \frac{t_{CPU}}{t_{RC}}$, where t_{CPU} is the time taken for a task to complete on our x86 platform, and t_{RC} is the time taken for the functionally equivalent task to complete on our reconfigurable computing (SRC-6) platform. Figure 3.9 shows the speedup obtained using the SRC-6, using both FPGAs in the MAPe module, and a total of 30 processing engines — 15 engines on each FPGA. With just one FPGA, we obtain a speedup approach 3x, and with two FPGAs, we obtain a speedup approaching 6x. Given that one FPGA produces a speedup s, we expect n FPGAs to produce a speedup ns, since there is no communication between processing engines, and no communication between the FPGAs until each engine has completed its simulations.

In both Figures 3.8 and 3.9 we see that the speedup for a smaller number of iterations is lower than for larger numbers of iterations. This is the case because the total time for the simulation to run to completion is less than 5 seconds, and the amount of time it takes to load the FPGA design becomes significant (Figure 2.2 shows the constituents of the completion time of an application).

Figure 3.10 shows how the performance of the simulation scales with the number of processing engines used. The number of iterations was kept constant at $1.5 \times 10^9 = 1500000000$. In this figure the speedup is defined as the completion time of a design with i processing engines, divided by the completion time of the design with one processing engine, running the same

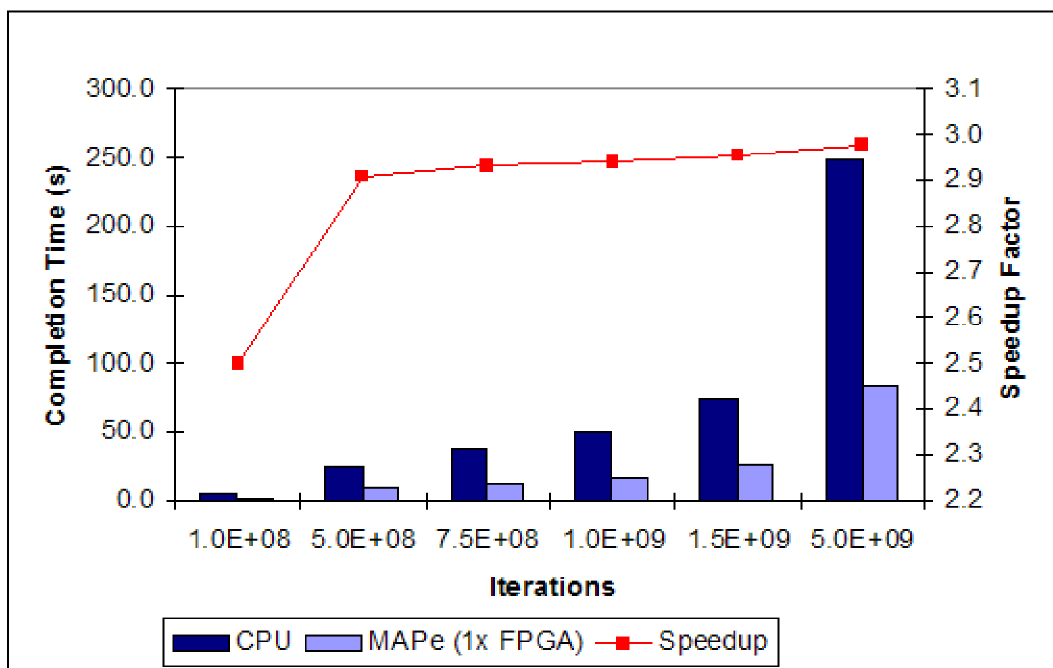

Figure 3.8: Performance of Monte Carlo estimation of π on an SRC-6 MAPe with 15 processing engines, using one FPGA, compared with that of an x86 processor.

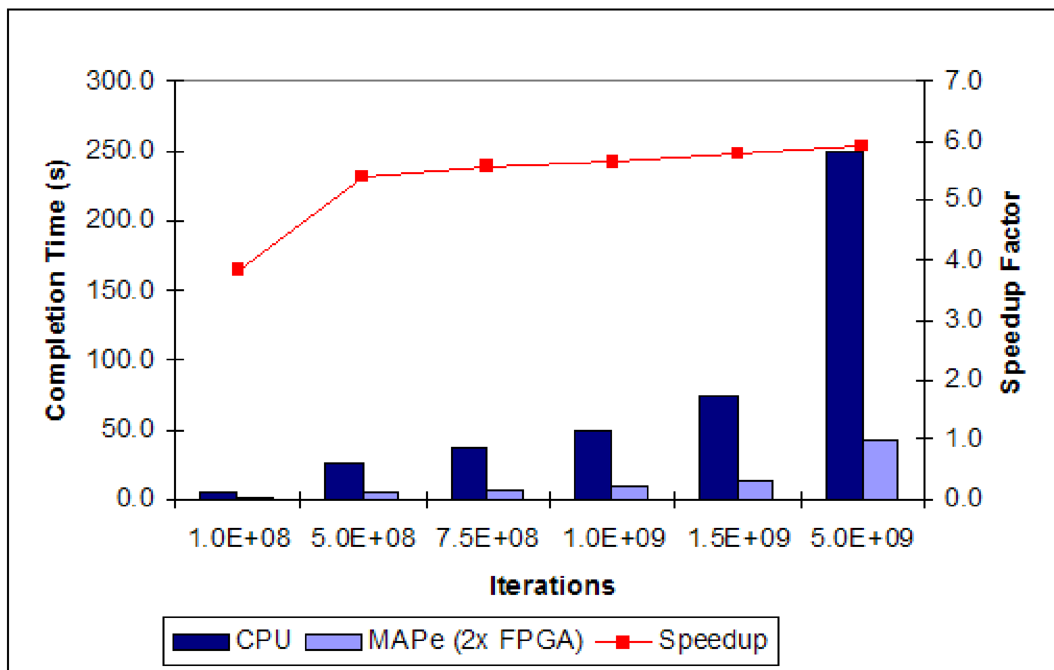

Figure 3.9: Performance of Monte Carlo estimation of π on an SRC-6 MAPe with 30 processing engines, using two FPGAs, compared with that of an x86 processor.

simulation. We see that there is a very nearly linear relationship between the speedup and the number of processing engines used. This figure also very clearly illustrates that without the parallelism enabled by the FPGA, and this favorable performance scaling relationship, we would not see the performance gains shown in Figure 3.8.

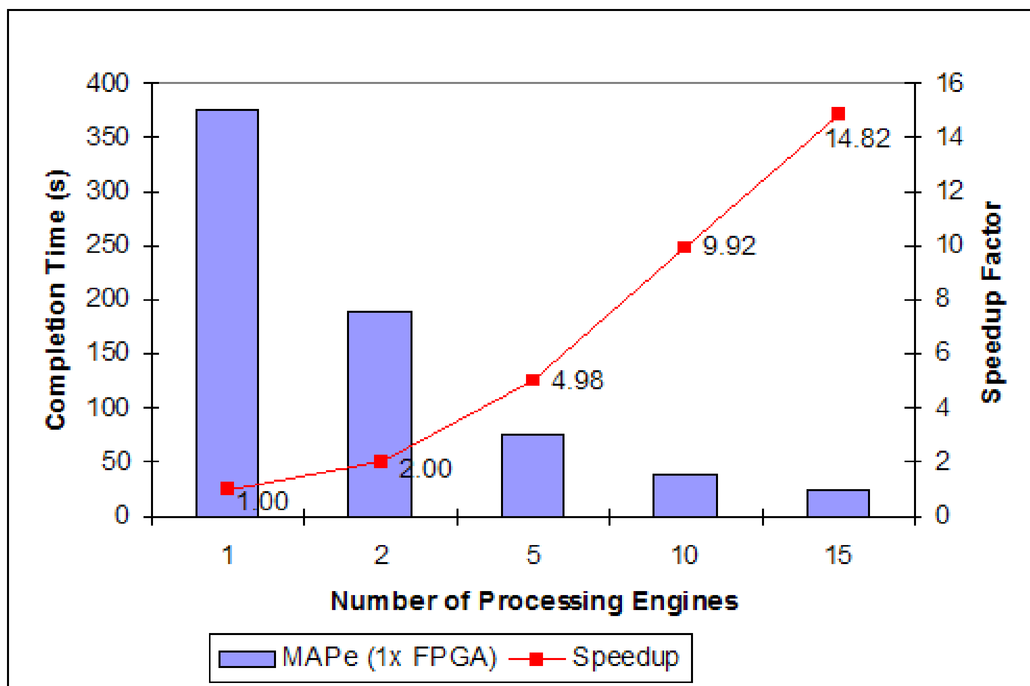

Figure 3.10: Performance of Monte Carlo estimation of π on an SRC-6 MAPe, using one FPGA, as a function of the number of processing engines used.

3.3 Monte Carlo Options Pricing

An important area in financial mathematics and financial engineering is that of *options pricing* [26]. An *option* is a financial device that gives a party (the *holder*) the opportunity to either sell or purchase an asset at a particular price on any date in a set of dates in the future. This party will pay for the right to have this opportunity — he will purchase the option from a *writer*. A

call option gives the holder the opportunity to *buy* an asset, and a *put option* gives the holder the opportunity to *sell* an asset. The writer is obligated to honor the decision of the holder of the option — either to buy or sell an asset at the price specified in the option contract.

Clearly an institution that sells options to investors would like to calculate the value of an option at its purchase date as accurately as possible. Unfortunately the formulae that exist for pricing options often result in analytically intractable expressions. Financial engineers resort to numerical methods to solve them, and Monte Carlo methods are often used.

We have implemented a Monte Carlo algorithm to price a specific type of option, an *Asian option*, which has no simple closed form expression [27].

3.3.1 Pricing Asian Options with Monte Carlo

An Asian option is an option whose payoff is determined by the average value of the asset to which it pertains over the duration of the contract. This is distinct from, for example, a European option, whose payoff depends on the value of the asset on the expiry date alone [26].

In general, the payoff h of a call option can be described as follows:

$$h(K, S) = \max(S - K, 0)$$

Here S is called the *spot price* and K is called the *strike price*. For a simple option, such as a European option, the spot price is simply the value (market price) of the underlying asset on the expiry date. In a European call option, the strike price is the price that the holder can purchase the asset for. Clearly if $K < S$, then it would be to the holder's advantage to exercise the option; the payoff would be $S - K$. However, if $S < K$, then if he exercised the option, he would be purchasing an asset for more than it is worth. In this case he would choose not to exercise the option, so his payoff would be 0.

For a put option, this is simply reversed, so the payoff h of a put option can be described as

$$h(K, S) = \max(K - S, 0)$$

From now on, we will only work with call options, but the methods we describe can easily be modified to apply to put options. We now present the background from Haugh [28] that is necessary to understand the problem we investigated. Haugh [28], Jäckel [29] and Glasserman [30] provide suitable introductions to Monte Carlo methods in finance for those interested in learning more about this domain.

Assume we have an asset A with a value at time t, which we denote as A_t. Let the expiry date of an Asian option be T days from the date the option is written. The value of the asset will be recorded at a specific time on m days, with regular intervals between recordings. The payoff of the option will then be

$$h(\mathbf{X}, K) = \max\left(\frac{1}{m}\sum_{i=1}^{m} A_{\frac{iT}{m}}, 0\right)$$

where $\mathbf{X} = \left(A_{\frac{T}{m}}, A_{\frac{2T}{m}}, \ldots, A_T\right)$. $\mathbf{X}$ is a vector of recordings of the price of the asset at regular intervals. The payoff is dependent on an average of the prices of the asset, rather than just the price of the asset when the option expires.

The value of the option when it is written (i.e. at time 0) is

$$C_0 = \mathbf{E}^Q\left[e^{-rT} h(\mathbf{X})\right]$$

where $\mathbf{E}^Q[.]$ is the expectation value under the risk-neutral probability measure. We now assume that the asset A is a stock. We can find an estimate of the price C_0, $\widehat{C_0}$ for the Asian option by simulating many different paths of the stock price over the relevant time period, and averaging the payoffs that we can calculate based on each path.

What we need to do is generate a set of different possible paths that the asset price A_t could take. A single vector $\mathbf{X}$ denotes a single path. Let us

generate a set $\left\{\mathbf{X}^{(1)},\ldots,\mathbf{X}^{(n)}\right\}$ of vectors $\mathbf{X}^{(i)}$, such that each vector represents a path that the asset price could take, independent of the other paths. Clearly there are infinitely many paths that the stock price can take, so it is not possible for us to generate all possible paths. Instead, we choose to generate random paths, and if the number of paths n is large enough, this suffices. This situation is analogous to the classic Monte Carlo integration scenario where we wish to evaluate a multidimensional integral, but sampling uniformly from the region of integration and using regular numerical integration techniques is too computationally expensive, so we instead draw random samples from the region of integration and proceed from there.

Clearly each value in a vector $\mathbf{X}^{(i)}$ is not independent of the others, so we can't simply draw them from some uniform distribution, as we might be able to in other, simpler Monte Carlo simulations. Each path is modelled as stochastic processes, specifically a *Geometric Brownian Motion*. We look briefly now at how to simulate such a process.

A stochastic process $\{X_t : t \geq 0\}$ is a Brownian motion[2] $B(\mu,\sigma)$ if

1. For $0 < t_1 < \ldots < t_n$, $(X_{t_2} - X_{t_1}), (X_{t_3} - X_{t_2}), \ldots, \left(X_{t_n} - X_{t_{n-1}}\right)$ are mutually independent.

2. For $s > 0$, $X_{t+s} - X_t \sim N(\mu s, \sigma^2 s)$, where $N(a,b)$ is the normal distribution with mean a and variance b.

A standard Brownian motion (SBM) is denoted as B_t, and is a Brownian motion with $\mu = 0$ and $\sigma = 1$. We assume that $B_0 = 0$.

We note that if $X \sim B(\mu,\sigma)$, and $X_0 = x$, then

$$X_t = x + \mu t + \sigma B_t.$$

Thus we can generate a Brownian motion using a standard Brownian motion.

[2]This is also known as a Wiener process.

A stochastic process $\{X_t : t \geq 0\}$ is a geometric Brownian motion $GBM(\mu, \sigma)$ if $\log(X) \sim B(\mu - \sigma^2/2, \sigma)$. We call μ the drift and σ the volatility. If $S \sim GBM(\mu, \sigma)$ then

$$S_t = S_0 e^{\left(\mu - \sigma^2/2\right)t + \sigma B_t}.$$

We can see that to simulate S, it will suffice to simulate the standard Brownian motion B. i.e. By generating the B_t for the required number of steps, we can simply insert these values into the expression to find the corresponding S_t, since all the other values are known.

To simulate B_{t_i} for $t_1 < t_2 < \ldots < t_n$, we simply do the following: Set $t_0 = 0$ and $B_{t_0} = 0$. For each i, generate $X \sim N(0, t_i - t_{i-1})$ and set $B_{t_i} = B_{t_{i-1}} + X$.

We have now described how we can price an Asian option, and have provided almost all the detail of which quantities a Monte Carlo simulation will need to calculate. We have seen above that to generate a geometric Brownian motion, we need to be able to draw samples from a normal distribution. However, we have thus far only described our development of a pseudorandom number generator that draws values from the uniform distribution. We describe now how we can use these values to generate normally distributed values.

3.3.2 Generating Normal Random Variables

We have an implementation of a pseudorandom number generator that draws samples from $U(0, 1)$. Several methods exist for generating samples drawn from $N(\mu, \sigma)$ given samples drawn from $U(0, 1)$. We use the Box-Muller method, as shown in [28].

If we generate $U_1 \sim U(0, 1)$ and $U_2 \sim U(0, 1)$, then if we define $Z = \sqrt{-2\log(U_1)}\cos(2\pi U_2)$, it can be shown that $Z \sim N(0, 1)$.

We note that if $Z \sim N(0, 1)$, then we can generate an X such that $X \sim N(\mu, \sigma)$ by setting $X = \mu + \sigma Z$.

3.3.3 Design and Implementation

As we have noted, the options pricing algorithm we implemented required the use of random variables drawn from a normal distribution. Thus we needed to modify our pseudorandom number generator, which draws samples from $U(0,1)$, to generate samples drawn from $N(0,1)$. We implemented the Box-Muller algorithm, and Figure 3.11 shows a set of data generated by our implementation on the SRC-6, as compared to normal random variables generated using MATLAB's `randn` function. We see that our data set corresponds well to the histogram generated by a trusted normal distribution generator. We ran the Lilliefors test for normality on our data, which confirmed that we should accept the null hypothesis that the data was taken from a normally distributed population.

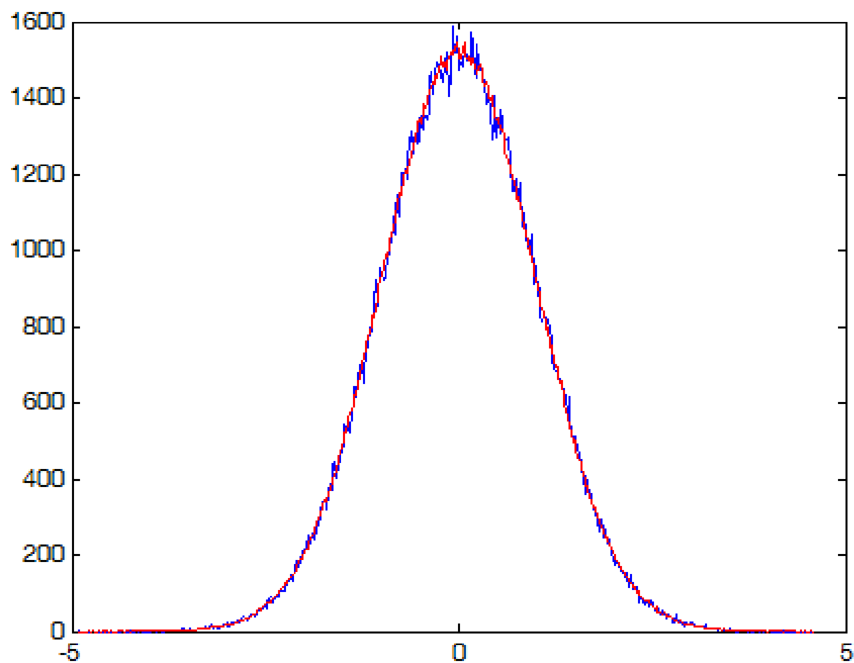

Figure 3.11: Histogram, with 500 bins, of 200000 random numbers generated on the SRC-6 (blue), and 2000000 random numbers generated using MATLAB's `randn` function, scaled (red).

With the capability to generate normally distributed random variables

working, we were able to extend our program to generate a Brownian motion, and following that, a geometric Brownian motion. The remainder of the options pricing algorithm is then relatively simple — we generate N geometric Brownian motions in a loop, and for each, calculate if the generated stock price path would have resulted in a positive payoff. The payoffs can be averaged, and the option price can be calculated from that average.

Because every stock price path is independent of all the others, it is easy to parallelize this algorithm — we simply have p processing engines each run N path simulations, and we can average the payoffs once all the processing engines have finished. Figure 3.12 shows the flow diagrams of the algorithms implemented in the processing engines. The output is a sum of the payoffs from each processing engine. Thus $\mathbf{E}^Q\left[h\left(\mathbf{X}\right)\right] = sum \cdot \frac{p}{N}$. Therefore our estimate of the option price C_0 is $\widehat{C_0} = \mathbf{E}^Q\left[e^{-rT}h\left(\mathbf{X}\right)\right] = e^{-rT} \cdot sum \cdot \frac{p}{N}$.

Flattening Loops for Loop Optimization

The MAPC compiler in the SRC Carte environment automatically performs some optimizations on the innermost loop in MAPC code. Specifically, it creates a data dependency graph and creates a pipeline that is optimized to the extent possible based on the constraints implicit in the data dependency graph.

The MAPC compiler performed this optimization for our Monte Carlo π estimator without any special effort on our part, but for our options pricing simulation we needed to 'flatten' our loops to get maximum benefit from this compiler capability.

If we have an inner and an outer loop, and would like the compiler to optimize both, we need to somehow combine the loops, since the compiler can only optimize the innermost loop. In our case we have a loop structure as follows:

```
// Generate N paths to average over
for (int i = 0; i < N; i++)
{
```

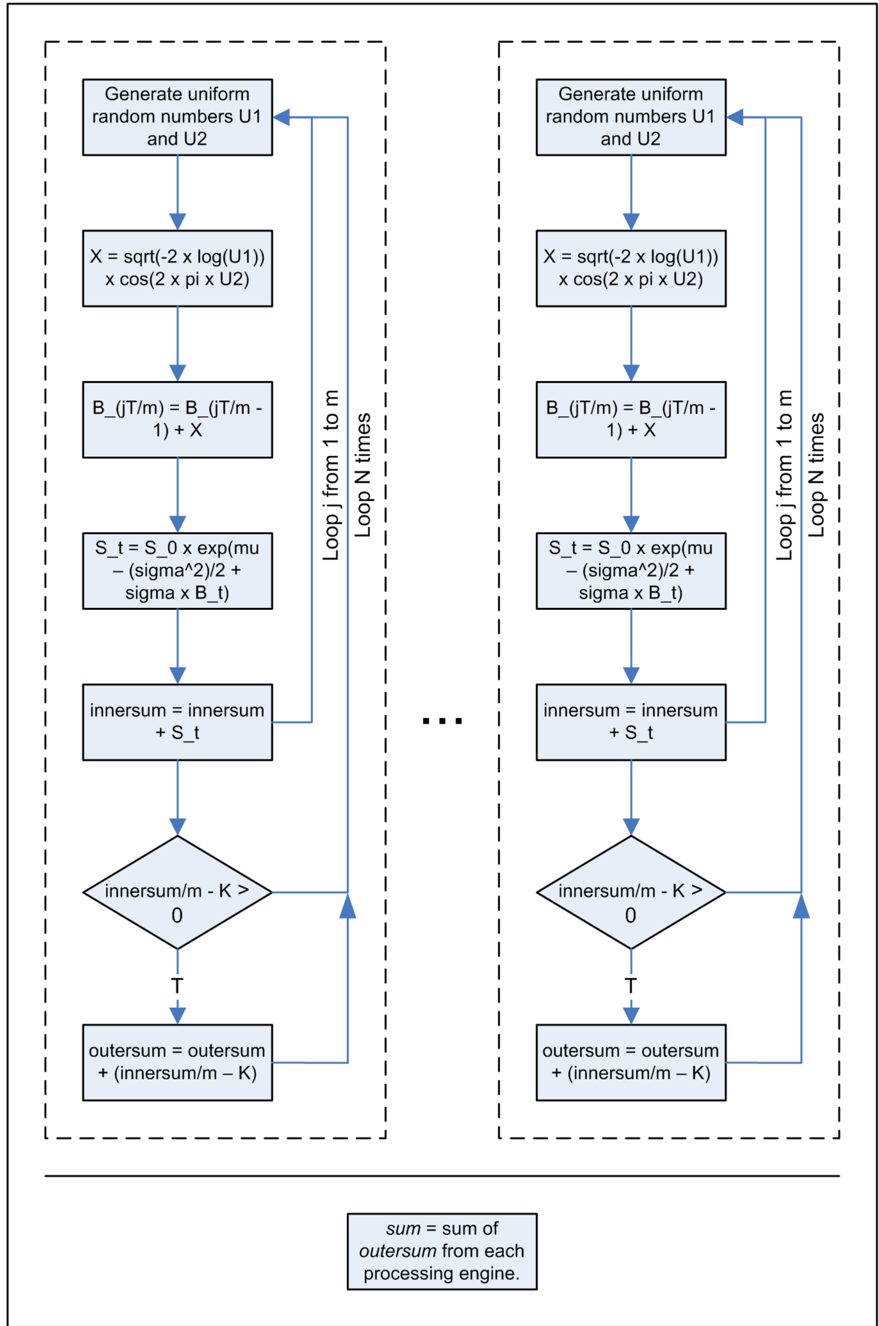

Figure 3.12: Processing engines each generating N independent paths of stock price movements to simulate possible payoff scenarios.

```
  // Generate m stock prices
  for (int j = 1; j <= m; j++)
  {
    // Generate stock price at next timestep
  }
}
```

We need to flatten the loops so that the compiler can optimize both loops. This is especially true in this case since m is likely to be relatively small, and N large, so having the optimizations performed only on the inner loop will not greatly improve performance. We flattened the loops as follows:

```
int i, j;
for (int count = 0; count < N * m; count++)
{
  i = count / m;
  j = (count % m) + 1;
  // i and j now cycle through the same values as they
  // did before flattening
}
```

The division and modulo operations are relatively expensive to implement, and it is preferable to use a counter with a ceiling value to obtain the same result. MAPC contains a function, `cg_count_ceil_32`, with signature `void cg_count_ceil_32 (int en, int reset_val, int reset_en, int ceiling, int *res);` that implements such a counter. We could replace the division and modulo operations in the flattened loop with the appropriate counters and get the same result:

```
int i, j;
for (int count = 0; count < N * m; count++)
{
  cg_count_ceil_32(1, 1, count == 0, m, &j);
  cg_count_ceil_32(j == 0, 0, count == 0, VAL_LONG_MAX, &i);
```

```
    // i and j now cycle through the same values as they
    // did before flattening
}
```

These counters increment in one clock cycle. Unfortunately they only function correctly if there are no branches within the loop. We needed to use several `if` statements within the loop, so we were forced to continue with the division/modulo flattening method. In the following section, which shows our performance results, Figure 3.13 shows the performance advantage that was obtained by flattening the loop in a simulation with one single processing engine, running on a single FPGA.

Build Results

Just a single processing engine consumes a considerable proportion of the resources available on the MAPe FPGA, as shown by the Place and Route report in Table 3.3. We see that nearly half the flip flop and lookup tables are used, and more the half the available slices are used. We also note that even though our algorithm doesn't require a very large number of floating point multiplications, with just one processing engine 13% of the integer hardware multipliers are used.

We assume that the high percentage of flip flop, LUT and slice usage is due primarily to the control flow that takes place in the algorithm. There are necessarily numerous conditional statements, and we expect that everywhere code branches, the MAPC compiler will have to create spatially distinct logic in hardware that needs to be connected for signaling and data transfer. Thus we conjecture that a considerable amount of the resources in the FPGA is 'wasted' helping route signals and data, rather than performing useful computations.

We attempted to compile a two-processing engine version of the simulation, which was successful with the caveat that the clock speed of the FPGA needed to be lowered. Specifically, table 3.4 shows the report from the Place and Route. We see that nearly all the slices were needed. It may be surpris-

Table 3.3: Place and Route results for a one-FPGA, one processing engine Monte Carlo options pricing simulation.

Resource	Used	Available	% Used
Slice Flip Flops	37,958	88,192	43%
4 input Lookup Tables	38,187	88,192	43%
Occupied Slices	28,048	44,096	63%
Block RAMs	3	444	1%
Hardware Multipliers	60	444	13%

ing at first glance that the Xilinx tools were able to generate a design that fitted at all, given that the one-processing engine version used 63% of the slices. However, the Place and Route algorithms used are genetic algorithms for optimization, and in the case of the first build, the algorithms likely did not find the smallest design possible, but rather just a design that satisfied the constraints and that works. Secondly, even though the Xilinx tools were able to find a design that fits for the two-processing engine version, this came with the caveat that the clock speed had to be dramatically dropped — from the regular frequency of 100MHz to a frequency of 56.1MHz. This will have been necessary since routing from one physical end of the FPGA to another end may have required a complicated path that could not operate at 100MHz.

Clearly this is a major problem, since we can at most hope for a 2x speedup by having two processing engines, but now the clock speed is nearly halved, so any speedup will be cancelled. We do, however, know that if the number of slices were increased by a relatively small amount (at most, a 25% increase would be necessary), the generated design would be suitable to run at 100MHz, so for a 25% increase in the number of slices, we will likely get nearly double the performance that we have with the present design on the SRC MAPe FPGA[3].

[3]The MAPe features two Xilinx Virtex II Pro FPGAs, and Xilinx has now already released members of the Virtex-5 family, which feature sufficiently many slices.

Table 3.4: Place and Route results for a one-FPGA, two processing engine Monte Carlo options pricing simulation.

Resource	Used	Available	% Used
Slice Flip Flops	68,581	88,192	77%
4 input Lookup Tables	70,564	88,192	80%
Occupied Slices	44,094	44,096	99%
Block RAMs	11	444	2%
Hardware Multipliers	113	444	25%

We also developed and built a version of the simulation that runs on both FPGAs in an SRC MAPe module, in the same way this was achieved for the Monte Carlo π estimator and illustrated in Figure 3.5. Since each FPGA runs independently of the other throughout the majority of the simulation, the resources required on each are very similar to those for the single FPGA case. We built versions of the software that provided one processing engine per FPGA, and two processing engines per FPGA, for a two-FPGA setup, and the Place and Route reports showed very similar usage percentages, as we expected. In both cases the Xilinx tools had to reduce the clock frequency in the design to meet the timing requirements.

3.3.4 Performance Results

First we look at the difference in performance between a version of the algorithm coded using unflattened, nested loops, and a version using flattened loops. As we have explained, the MAPC compiler only optimizes the innermost loop. In Figure 3.13 we see that the compiler is able to perform optimizations that improve the performance of the code by approximately 22% if we flatten the loops. This test was done using only one processing engine, on only one FPGA.

The performance of the flattened version is limited largely by the data dependencies in the code. The algorithm is iterative, since the next value

in a standard Brownian motion is dependent on the present value, and thus the optimizations that the compiler can perform when building a pipeline are restricted. This is reflected in the fact that each loop iteration requires 79 clock cycles.

As we might expect, the performance appears to scale linearly with the number of iterations we perform. Here iterations refers to the number N of paths we simulate. We ran the simulations with parameters $T = 1$ (year) and $m = 11$ (months).

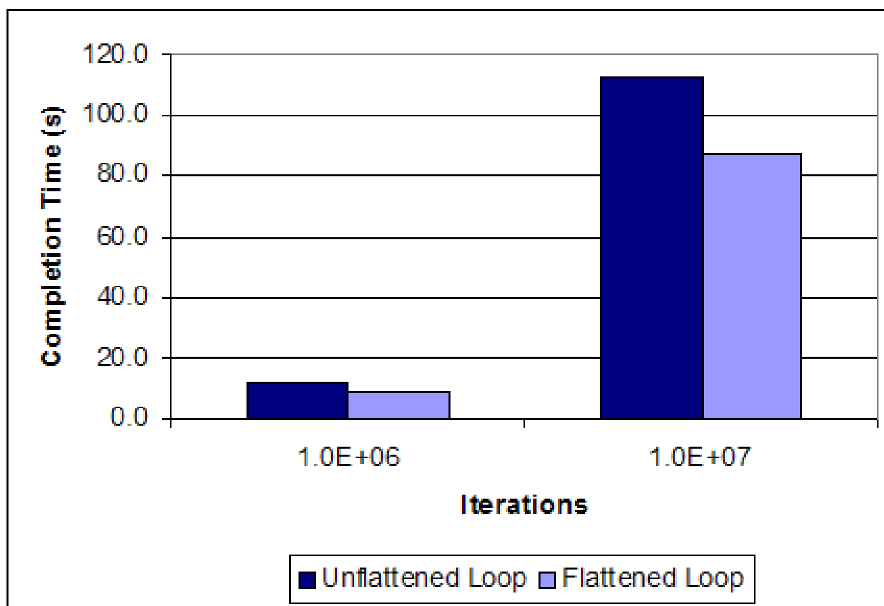

Figure 3.13: Performance of a Monte Carlo options pricing simulation on an SRC-6 MAPe, using one FPGA, showing the speed difference between unflattened loops and flattened loops.

Figure 3.14 shows that the SRC-6 implementation of the options pricing simulation performs considerably worse than a standard C implementation running on the MAPstation's Intel Xeon 2.8GHz processor. If we run a simulation on a single FPGA on the MAPe module, with just one processing engine, the time to complete the simulation takes approximately 4.8x longer than on the CPU. If we use two processing engines on a single FPGA, we

obtain only a slight improvement, since, as we explained in the previous section, the design can only be run at 56.1MHz, which is nearly half the normal clock speed. Thus we lose nearly all the advantage of having two processing engines tackle the problem in parallel because each engine is running at just over half the normal speed.

We have plotted estimates for completion time for two further scenarios: firstly, two processing engines running on a single FPGA, and secondly, two processing engines each running on two FPGAs. These estimates make the assumption that the FPGA is sufficiently large that the designs can be accommodated without the need to lower the clock speed[4].

We see that even if we used a larger FPGA, which allows the design to be placed such that they can run at 100MHz, the performance of the two FPGA implementation (with two processing engines in each FPGA) would not beat the performance of the CPU, although the speedup/slowdown would approach unity.

3.4 Conclusion

We were able to successfully implement a parallel pseudorandom number generator for the SRC-6 MAPstation reconfigurable computer, and use it as a basis to develop two Monte Carlo simulations: an estimator of π, and an options pricing calculation.

For the simpler simulation, the π estimator, we achieved a speedup of 6x using two FPGAs and 15 processing engines in each.

For the options pricing simulation, we found that even with a larger FPGA, the SRC-6 reconfigurable computing implementation would not be able to rival the performance of the Xeon 2.8GHz CPU.

In both cases, the number of slices on the FPGA was the limiting factor

[4] We estimated in the previous section that at most 25% more slices per FPGA would be required, based on our Place and Route results for the build of the one-processing-engine, one-FPGA implementation.

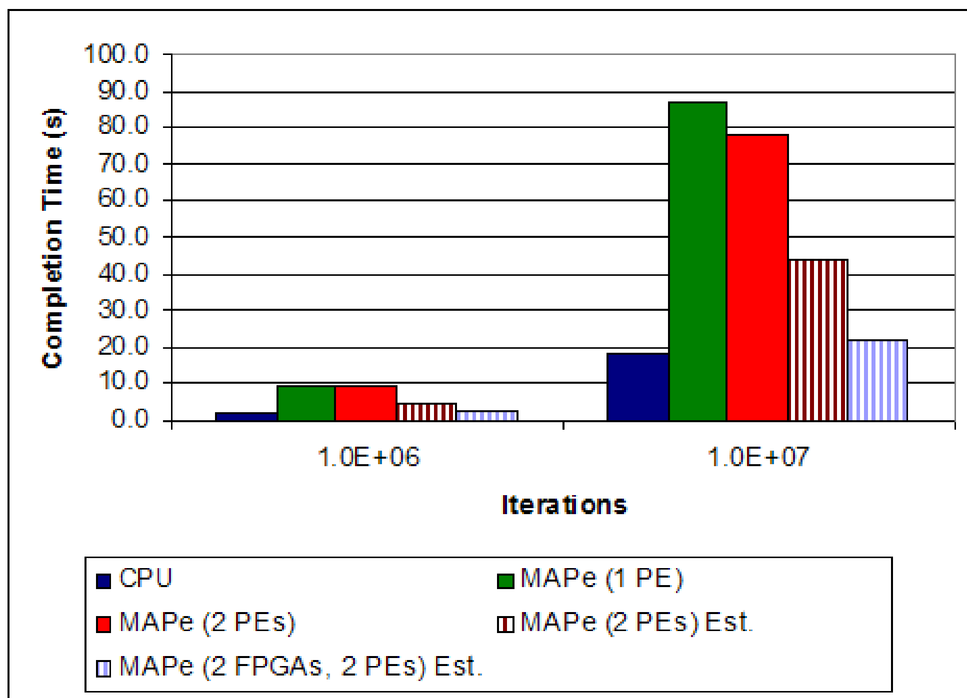

Figure 3.14: Performance of a Monte Carlo options pricing simulation on an SRC-6 MAPe, compared with that of an x86 processor.

in our ability to scale up performance. However, from the π estimator we saw encouraging results that indicate that performance scales linearly with the number of processing engines we can fit onto an FPGA. We also verified that the resources required by a program scale nearly linearly with the number of processing engines. Thus as FPGAs become larger, we will be able to fit proportionately more processing engines onto an FPGA, and the performance of the programs will grow proportionately.

Chapter 4

Cellular Automata Simulations on Reconfigurable Computers

Cellular automata are discrete models that operate according to a fixed set of rules, uniform across space. Cellular automata were first extensively studied by von Neumann [31], who founded the area of research while attempting to find a simple system that could exhibit self-replication. Self-replication is a common feature in biology, and von Neumann wanted to show that self-replication needn't result from an extremely complicated system and set of rules.

Von Neumann was successful in finding a self-replicating system, and since then cellular automata have been studied to model other complex behavior. Wolfram [10] has published a comprehensive volume that includes a wide variety of examples of physical phenomena that can be modeled, with a certain degree of success, using cellular automata. A particularly interesting aspect of cellular automata is how a system with very simple rules can exhibit extraordinarily complex behavior that is seemingly impossible to predict (without actually simulating the system).

We now introduce cellular automata formally, following the presentation in [32] for the one-dimensional case, and extending the definitions to the two-dimensional case.

4.1 An Introduction to Cellular Automata

A cellular automaton is a discrete model described by a space S, a set of states Q, a function $F_S : Q^S \to Q^S$ that describes how the system evolves in time, and an initial configuration $X \in Q^S$. At time $t \in \mathbb{N}$, the configuration will be described by $F_S^t(X)$.

The function F_S takes as input the present configuration of the system, and outputs the configuration after one 'timestep'. If we need to define F_S in a piecewise fashion for all possible configurations in Q^S we would not be able to work with sizeable systems. The size of the space S, $|S|$, will typically be far greater than 100, and the number of possible states, $|Q|$ will be at least 2. This would mean having to define at least 2^{100} state transitions.

Instead of defining F_S directly, we can define a local function $f : Q^n \to Q$ that acts locally on individual 'cells' in the space. This function can be used across all cells in the space to obtain the same effect as F_S.

4.1.1 One-dimensional Cellular Automata

For example, if we have a finite one-dimensional space $S = \{0, 1, \ldots, N-1\}$, with two possible states for each cell, i.e. $Q = \{0, 1\}$, the configuration at timestep $t+1$, X', may be calculated using the configuration at timestep t, X:

$$
\begin{aligned}
X' &= F_S(X) \\
&= F_S\left(\bigotimes_{i \in S} X_i\right) \\
&= \bigotimes_{i \in S} f(X_{i-a}, \ldots, X_i, \ldots, X_{i+b}).
\end{aligned}
$$

Here X_i represents the state of the ith cell in configuration X. We have thus been able to redefine F_S in terms of the local function $f : Q^{a+b+1} \to Q$. f is the transition of the state of the ith cell, X_i, to the state of the cell at the next timestep, X_i'. The state X_i' depends only on the state of the

cell in the previous iteration, X_i, and the states of its $a + b$ neighbours, $X_{i-a}, \ldots, X_{i-1}, X_{i+1}, \ldots, X_{i+b}$.

The function f has only $|Q|^{a+b+1} = 2^n$ elements in its domain, where $n = a+b+1$. The 'size' of the local function thus depends only on the number of possible states that each cell can be in, and the number of neighbour states that need to be considered when calculating the transition of a single cell.

We can now define the function f using a finite table. If $a = b = 1$, then $n = 3$, the function can be defined using $2^3 = 8$ mappings. In this example, the state X'_i depends on the previous state X_i and the states of the two immediate neighbours of the ith cell. We can implement the rule that the state of a cell should be flipped if its neighbors are in the same state[1] by defining the function as follows:

$$
\begin{aligned}
f(0,0,0) &= 1 & \qquad f(1,0,0) &= 0 \\
f(0,0,1) &= 0 & \qquad f(1,0,1) &= 1 \\
f(0,1,0) &= 0 & \qquad f(1,1,0) &= 1 \\
f(0,1,1) &= 1 & \qquad f(1,1,1) &= 0
\end{aligned}
$$

Thus if we have a configuration $X = (0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ \ldots\ 0\ 1)$, we can calculate the configuration at the next timestep using N evaluations of f — one evaluation for each cell. For example, assuming zero-based indexing[2] the state X'_1 can be calculated by evaluating $f(X_0, X_1, X_2) = f(0, 1, 0)$, which we can see from the table is 0. Thus in the next timestep, the state of the second cell[3] of the automaton will switch from 1 to 0.

[1] More precisely, we wish to set $X'_i = 1 - X_i$ if $X_{i-1} = X_{i+1}$.

[2] i.e. The first cell has index 0, so if the cellular automaton is in configuration X, the state of the first cell is X_0.

[3] An issue surrounds how we should handle boundary conditions — the first and last cells only have one neighbour each, and the local transition function requires input from two neighbours. We typically just pretend that each of these cells has a second neighbour, and that the state of that neighbour is some fixed, arbitrary value. For example, we may

4.1.2 Two-dimensional Cellular Automata

Most cellular automata models that attempt to model some physical process use two or more dimensions, since we more often than not would like to investigate two- or three-dimensional physical systems. We can extend our one-dimensional cellular automata definition to two dimensions fairly easily.

Again we have a set of states Q, and a space S, where S is now two-dimensional. A global transition function $F_S : Q^S \to Q^S$ can again be redefined in terms of a local transition function $f : Q^n \to Q$. A configuration $X \in Q^S$ specifies the state of every cell in the two-dimensional space S, which can now be thought of as a matrix, or grid of cells. We denote as $X_{(i,j)}$ the state of the cell in the ith row and the jth column. Now we can find the next configuration X' from the current configuration X using just local functions by redefining F_S as follows:

$$\begin{aligned} X' &= F_S(X) \\ &= F_S\left(\bigotimes_{(i,j)\in S} X_{(i,j)}\right) \\ &= \bigotimes_{(i,j)\in S} f\left(X_{(i-a,j-b)}, \ldots, X_{(i,j)}, \ldots, X_{(i+c,j+d)}\right). \end{aligned}$$

The essential difference is that the local function now considers neighbours in two dimensions rather than one. The transition from state $X_{(i,j)}$ in the cell at position (i, j) to $X'_{(i,j)}$ now relies on the state $X_{(i,j)}$ as well as the state of the cell 'near' it. Specifically, we define a rectangular grid around the cell at (i, j). The topmost, leftmost cell in this grid is at position $(i - a, j - b)$, and the cell at the bottommost, rightmost cell in this grid is at position $(i + c, j + d)$.

choose the state 0, and can then calculate the time evolution of the first cell by evaluating $f(X_{-1}, X_0, X_1) = f(0, X_0, X_1)$.

4.2 Conway's Game of Life

The canonical example of a cellular automaton is Conway's Game of Life (GoL) [33], which is a simple binary (two-state), 2D cellular automaton. In GoL, $a = b = c = d = 1$; that is, the state $X'_{(i,j)}$ depends purely on the state of the cell at the current time at position (i, j), and the states of the cell's immediate neighbours.

In GoL, the two states are labelled 'dead' and 'alive'. We will call the state 0, 'dead', and state 1, 'alive'. The rules can be summarised as follows: if a cell is alive, it will 'survive' (that is, remain alive) if the total number of its immediate neighbours that are also alive is either two or three. If a cell is dead, there will be a 'birth' (the cell will transition from dead to alive) if the number of its neighbours that are alive is exactly three. If the number of 'alive' neighbours a cell has is less than two, or greater than three, then the cell 'dies' and will be dead in the next timestep. Figure 4.1 shows four situations that demonstrate all three scenarios. In each case the centre cell is the one whose transition we are considering.

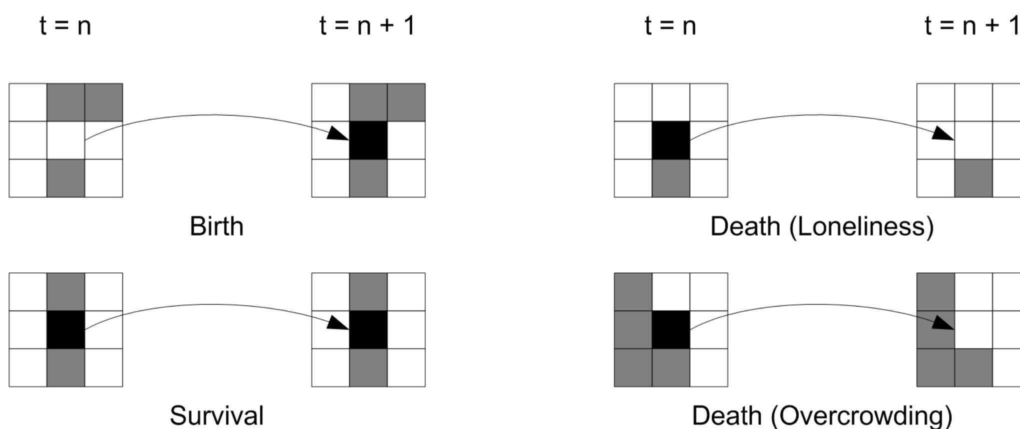

Figure 4.1: Examples of Birth, Survival and Death in the Game of Life.

Clearly the transition function f can be specified by a piecewise definition with $|Q|^{(a+c+1)\cdot(b+d+1)} = 2^9 = 512$ entries. For example, the following mapping (which represents a birth) may be defined:

$$(0,1,1,0,0,0,0,1,0) \equiv \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \mapsto 1$$

However, in practice when simulating the Game of Life, it is easier to just apply the rules as we stated them above, and implement them by performing a count of the number of neighbours that are alive, and then use `if` statements to determine the transitioned state based on these counts.

This is not the case in general, and we will see later that another two-dimensional CA, a lattice gas, is most easily implemented by just using a lookup table that acts very similarly to f.

4.2.1 Design and Implementation

We designed and implemented a program to perform a Game of Life simulation on the SRC-6 MAPstation. Our two primary considerations were how we should go about parallelising the problem, and where we should place the data.

Parallelizing Two-Dimensional Cellular Automata

Since the transition of a cellular automaton configuration from one timestep to the next can be calculated using just applications of a local function, cellular automata are excellent candidates for *data parallelization*. Data parallelization [34] is a classic parallelization scheme whereby a problem is parallelized by splitting up the input data, distributing it to different processors, and having each processor perform the same function on the data it has been sent. The results can then be merged once each processor is done.

Figure 4.2 shows a small Game of Life grid in some configuration. If we wish to compute the next configuration, we need to loop through each cell in the grid, and apply the Game of Life rules to it, to find the state of the corresponding cell in the next configuration. Let's say we have a set

of processors[4]. We would like to divide the work so that each processor in the set does an equal amount of work, and the next configuration can be computed together faster than just one processor would be able to do it.

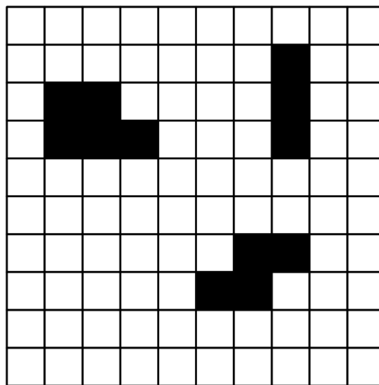

Figure 4.2: A grid representing a configuration in the Game of Life cellular automaton.

One option is to divide the grid horizontally and vertically, as shown in Figure 4.3. This, however, creates unnecessary communication overhead — each processor will now have to communicate with two others, since each subgrid borders two others. We obviously would like to reduce communication as much as possible, since communication results in wasted computation time.

Another possibility is to divide the grid into horizontal 'slices', as shown in Figure 4.4. This scheme is both simpler to implement, and had fewer communications requirements. It also doesn't place any restrictions on the number of processors that can be used — the previous scheme doesn't allow for odd numbers of processors. We chose to use this data parallelization scheme in our design.

If we consider each slice in the decomposition in isolation, we are unable to apply the local transition function f to every cell. Specifically, every cell

[4]We use the word 'processors' here in the general sense — that is, it is an entity that performs information processing, and may be a 'processing engine' in an FPGA; a physical microprocessor, or indeed even a human following instructions.

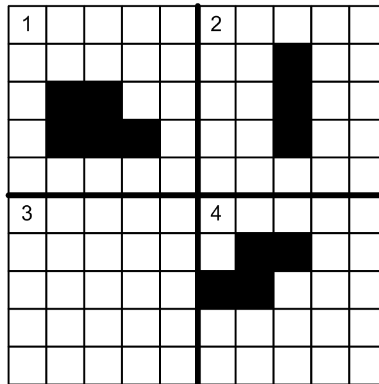

Figure 4.3: A grid decomposed into four subgrids, by dividing the grid vertically and horizontally.

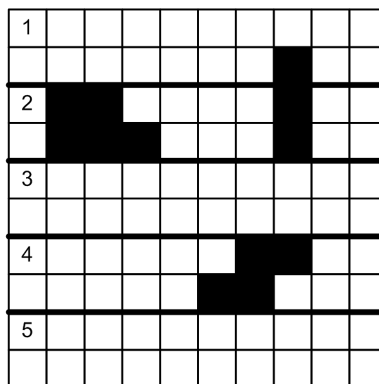

Figure 4.4: A grid decomposed into five subgrids, by dividing the grid into horizontal strips.

that is on a border with another slice will be troublesome. For the small grid in Figure 4.4, this is the case for every cell. Why are we unable to compute f on cells that lie on the borders? For the Game of Life, f requires as input the states of all the cell's neighbours (as well as the state of the cell itself). In every case, a cell's neighbours on the same row will be 'visible'. However, if we are for example in the second slice, and wish to calculate the transition of the third cell in the first row of the slice, we can see the states of the cell's neighbours on the same row, and on the row beneath it (which is also part of the second slice), but the cells in the row above it belong to the first slice, and are not visible.

This problem can be fixed by the introduction of 'ghost regions' — at every border between slices, we insert two rows of cells, one 'above' the border, and one 'below' it. In other words, each slice has a ghost, or buffer region of one row of cells, placed above the topmost row of cells, and another placed below the bottommost row of cells. We needn't have buffer regions at the top of the topmost slice, nor at the bottom of the bottommost slice, although it does not harm to include them.

In these ghost regions, we place copies of the cell states from the corresponding states on the other side of the border. Thus in the second slice, the top ghost region will be a replica of the configuration of the bottommost row of cells in the first slice. The bottom ghost region in the second slice will be a replica of the configuration of the topmost row of cells in the third slice.

Figure 4.5 shows our Game of Life grid with ghost regions inserted. The ghost regions are coloured light grey for cells that are dead, and a darker grey for cells that are alive. With the ghost regions in place, every slice can be processed independently of the others.

After every timestep, the ghost regions need to be updated. This is because the cell states in the non-ghost regions reflect the next configuration, but the ghost region cell states are based on the configuration before the transition. To update, we simply take the top row of the second slice and place it in the bottom ghost region of the first slice; take the bottom row

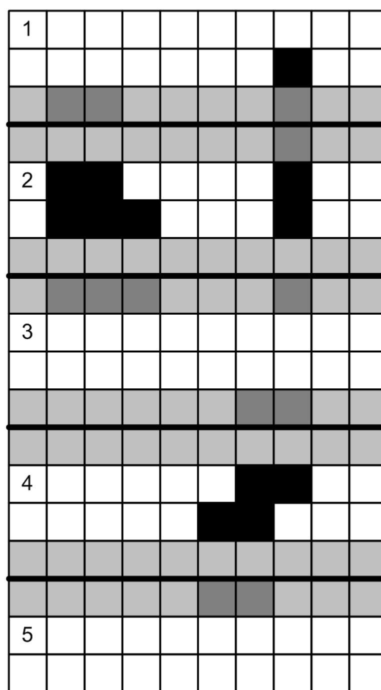

Figure 4.5: A grid decomposed into five subgrids, showing the 'ghost regions' for each slice.

of the first slice and place it in the top ghost region of the second slice, and so on. In principle in shouldn't matter which order we copy the new configurations to the ghost regions in, but in practice we may need to be more careful.

If each slice is processed by a separate processor, so slice 1 is on processor 1, slice 2 is on processor 2, and so on, then we can use the following updating scheme: processor 1 sends the bottom row of slice 1 to processor 2. Processor 2 receives the row from processor 1, and then sends the bottom row of slice 2 to processor 3. This continues until we reach the last processor, p. Once processor p has received the bottom row of slice $p-1$, from processor $p-1$, it sends the top row from slice p to processor $p-1$. Processor $p-1$ then sends the top row of slice $p-1$ to processor $p-2$, and so on. This continues until processor 1 has received the top row of slice 2, from processor 2.

This scheme will avoid deadlock even if receiving calls are blocking. Figure 4.6 illustrates the synchronization scheme, with the order of operations entered clockwise. Each circle node in the graph represents a processor, which is responsible for a single slice of the grid.

Implementing Ghost Region Synchronization in MAPC

SRC MAPC provides a variety of language constructs to enable concurrent programming, and communication between parallel sections. The `#pragma src parallel sections` directive allows us to create parallel 'processing engines'. We can think of this directive as forcing an operation similar to the Linux process programming model `fork()` and `join()` operations. Thus when we start a region with parallel section we are effectively forking, and then when end the region with parallel sections, we are joining the processes back again[5].

The first design for the program that we implemented is shown in Figure

[5] In practice, the SRC Carte environment simulator actually uses OpenMP to implement parallel sections, and the programming model of OpenMP is nearly identical to that of parallel sections in MAPC.

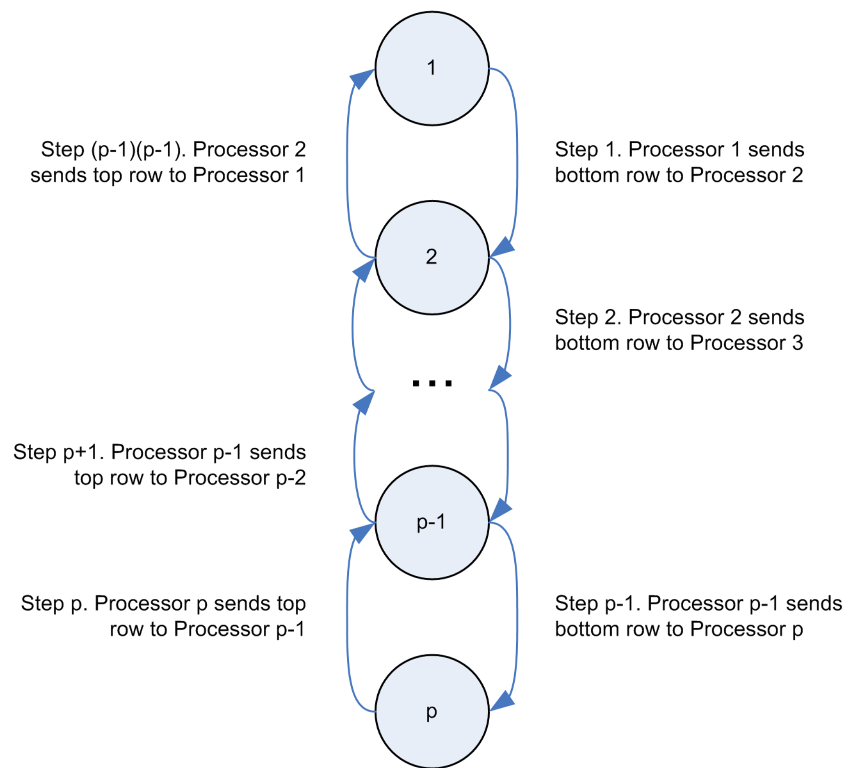

Figure 4.6: Sequence of processor communication to synchronize ghost regions, while avoiding deadlock.

4.7. The overall idea behind the design is that we create multiple processing engines, and each engine exists for the lifetime of the whole simulation. Each processing engine is assigned a slice of the CA grid (one slice per processing engine). The processing engines operate by performing a transition on their slice from one configuration to the next, and then updating their ghost regions by communicating with their neighbouring processing engines. The synchronization scheme used is based on that shown in Figure 4.6.

There are various possibilities for implementing this design in MAPC. The preferred technique is the use of *streams*. Streams in MAPC are unidirectional, and are used to implement the producer-consumer pattern [38]. We first attempted to implement the design using $(p-1)(p-1)$ streams, one stream for each communication shown in Figure 4.6. For example, we had a stream `s_1_to_2` that was used to send the bottom row of the slice in processor 1 to the ghost region in processor 2. Another stream, `s_2_to_3`, was created to transfer states from processor 2 to processor 3. Another set of streams was used to copy states in the other direction, from the top row of processor i to the ghost region in processor $i-1$.

The major issue with this setup is that we have no explicit mechanism for ensuring that every processing engine is on the same iteration. We would expect, in an idealized situation, that there would be no need for an explicit barrier[6], since each processing engine is doing identical work, implemented using nearly identical logic. However, to begin with, the first and last processing engines perform slightly less work than the others, since they only have to communicate with one neighbouring processing engine each. In addition, the compiler doesn't guarantee that two functionally identical parallel sections will be implemented in an indentical fashion in hardware, and will be started on exactly the same clock tick.

To solve this, we problem we attempted to create our own barrier. We

[6]By 'barrier', we mean a point at which all processing engines should stop until all the others have reached that point. We use the term in the same sense as the `MPI_Barrier()` function in the Message Passing Interface parallel programming API.

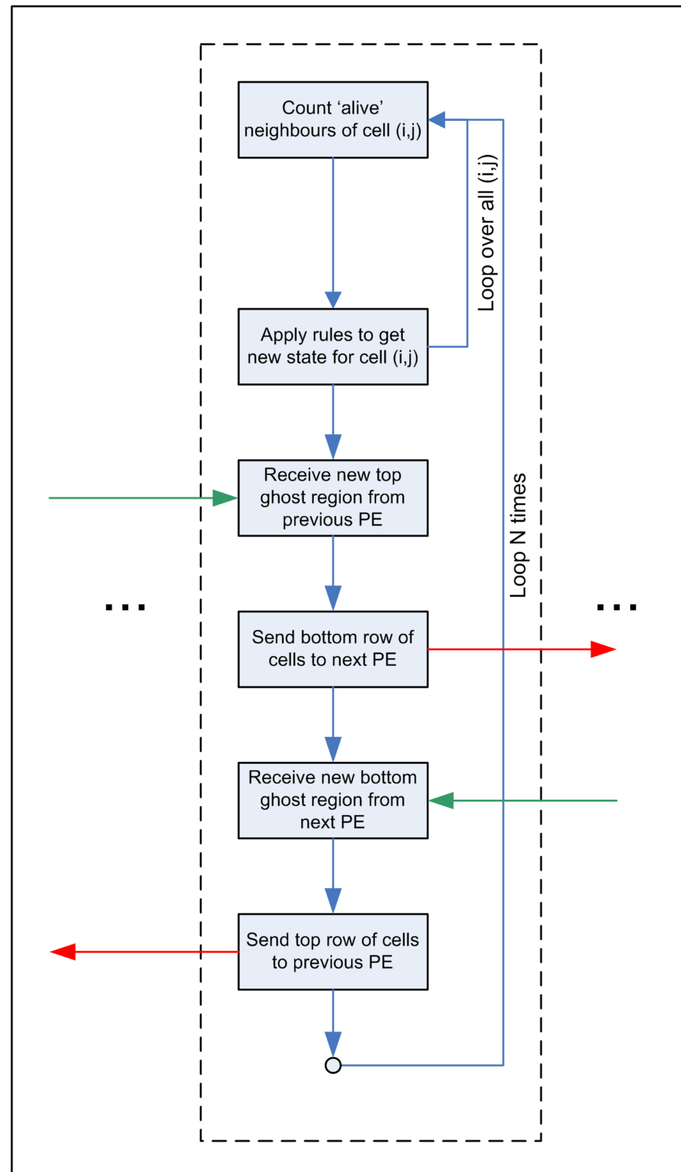

Figure 4.7: Processing engine performing an N timestep simulation, using the design that all communication is done between the processing engines.

devised and implemented several schemes to this end. The first involved designating the first processing engine as a 'master process', and making it a 'checkpoint' for all the other processing engines. In this scheme each processing engine must send a signal to the first processing engine once it has finished updating its ghost regions. Each processing will then wait until it receives a signal from the first processing engine. The first processing engine waits until it has received 'ready' signals from all the processing engines before sending out a signal to every processing engine to allow them to continue.

An alternative scheme that we tried used shared variables, locked using critical sections, to create a barrier. Each processing engine, once it has updated its ghost regions, sets a flag on a shared variable marking it as 'ready'. It then waits until all the flags from the other processing engines are set. In this way all the processing engines reach a barrier until every processing engine has set its flag. Each processing engine then resets its own flag and continues.

In one other variation of this scheme that we tried we didn't use any streams, and relied solely on shared memory and variables. We will discuss the layout of the data in more depth later, but a key feature is that we can arrange the data so that any processing engine can access any other processing engine's data. The scheme then works like this: the first processing engine accesses the cell data from the second processing engine and copies the first row to update its own ghost region. It then copies its own bottom row of cell data to the top ghost region in the second processing engine. Once the first processing engine is done, it send a signal to the second processing engine. The second processing engine then does the same for itself and the third processing engine, and so on.

This, however, is no different than if we just had one processing engine handle all the ghost region synchronization. We will see that this is ultimately what we resorted to.

In all the above schemes we tried there were quirks that resulted in dead-

lock, or just occasional anomolous behaviour. Our attempts to use streams, critical sections and shared flag variables to create a barrier clearly go beyond what these langugage features were intended for. A different approach to synchronization was required.

The design we switched to is shown in Figure 4.8. Whereas in the design shown in Figure 4.7, we forked multiple processing engines once, in this design we fork and join the processing engines once every iteration.

We can think of the design as having p processing engines and a separate 'master process' that orchestrates the computation. When the parallel sections region ends (i.e. when we effectively perform a join operation), we have essentially created a barrier. This is intrinstic to the SRC parallel sections implementation.

In this scheme we have the 'master process' update the data in the ghost regions directly — it copies data directly from various arrays storing slice data.

Data Layout

Data layout is a particularly important aspect of the program design, since the time it takes a computation to complete can easily double, or worse, if we make poor choices.

The onboard memory (OBM) banks take 5 clock cycles to read from. If access onboard memory from within a loop, every loop iteration will be slowed down by these accesses. Thus if we have 9 memory accesses in a loop (which, in the simplest case, we will have since there are 9 cells whose values need to be read per iteration — the cell we're updating, and its 8 neighbours), the loop will require at least $9 \times 5 = 45$ clock cycles per iteration.

Loops should ideally be designed to use as few clock cycles per iteration as possible. One of the simplest ways of improving matters is by moving the data to memory that is quicker to access. We decided to store the grid data in Block RAM (BRAM), which is on the FPGA. BRAM requires only one clock cycle to access, so by making this design choice, we improve loop

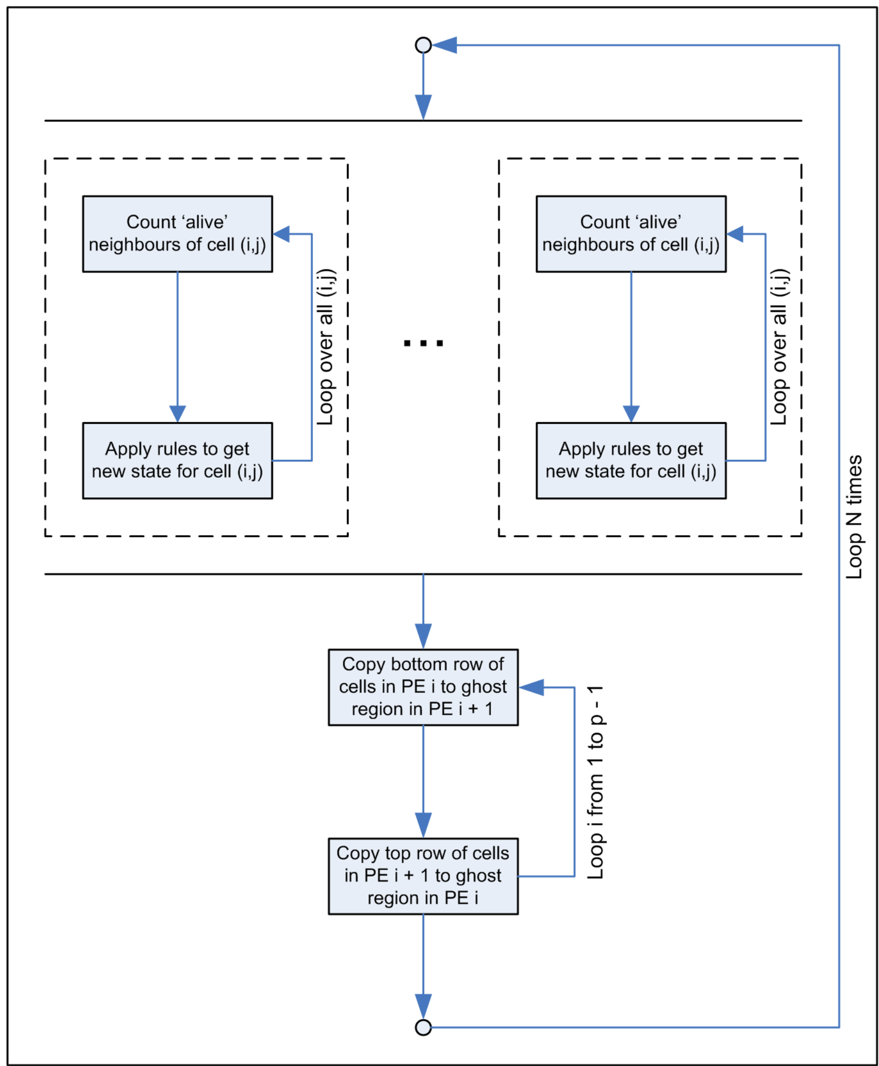

Figure 4.8: Processing engines performing an N timestep simulation. The program forks processing engines, then joins them once every iteration.

performance dramatically — from at least 45 clock cycles per iteration, to at least 9 clock cycles per iteration.

Another issue we had to consider was that related to the parallelism in the system. We would like the loops in each processing engine to run as fast as possible. However, a single array defined in MAPC, which is realized in BRAM, allows only one read to the entire array per clock cycle. Thus if we used a single array to store the grid data, there would be contention for access to the memory amongst the processing engines. Our solution was to create a separate array for each processing engine. With this design, each processing engine can access its own BRAM array once per clock cycle, and it is not necessary for the programmer or the compiler to have to orchestrate reads amongst the processing engines[7].

Figure 4.9 shows the design of the system and the data flow within in. The initial cellular automaton configuration is transferred from the CPU to the onboard memory in the MAP module using a direct memory access transfer. The data is then copied and distributed from the onboard memory to the appropriate BRAM arrays[8]. The processing engines can then perform the cell updates by reading from and writing to their BRAM arrays.

Once the simulation is finished, the data flows in the opposite direction: the BRAM data is packed into an OBM bank, and from there a DMA transfer is used to send it to the CPU.

A final issue to consider for cellular automata simulations in general, is where one should store the lookup tables that define the transition function f. In the case of Game of Life, however, we implemented the update rules in logic, rather than using a lookup table.

---

[7]The onboard memory banks have the same characteristic as BRAM: it is only possible to make one read from a bank at a time. Since there are only 6 banks, at best it would be possible to have 6 processing engines. Since we rely on parallelism in the FPGA to obtain a speed advantage over a microprocessor, this is a fairly severe restriction.

[8]Depending on the size of an array, it may span more than one BRAM block.

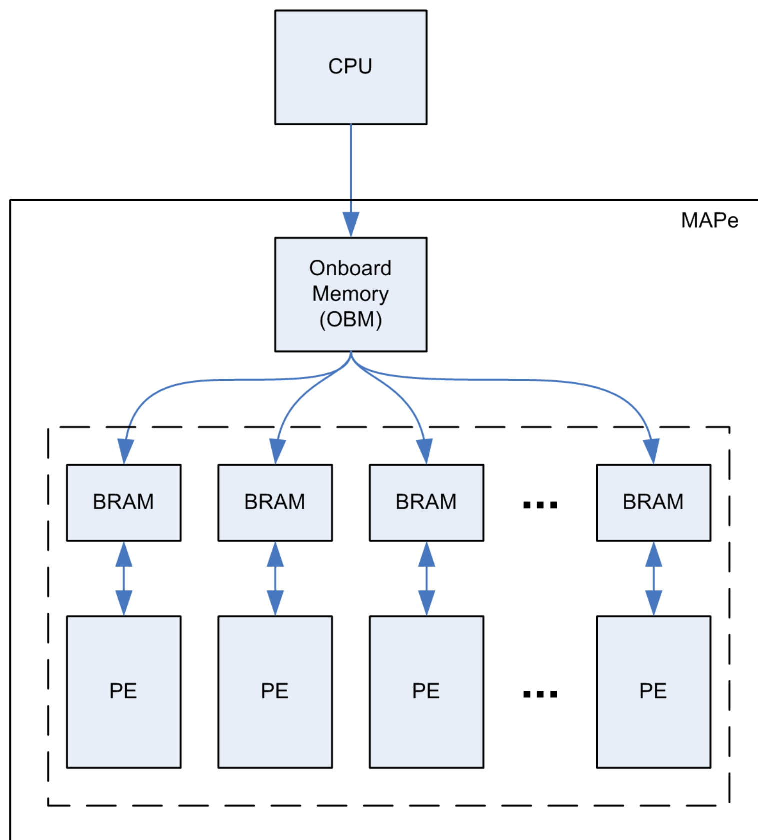

Figure 4.9: Layout and flow of the data in the system during a cellular automata simulation.

BRAM Lexical Write Limits

Figure 4.9 shows that after data is transferred to the MAPe module from the CPU, it needs to be moved from the onboard memory into BRAM. The OBM banks are 64-bits wide, but the cell data is 8-bits per state[9]. Thus when we read a value from OBM, we need to unpack it into its 8 constituent values (each value obtained from OBM is 64 bits wide, and each cell state contains 8 bits, so each OBM value contains 8 cell states). Once it has been unpacked, we need to write these values to BRAM.

MAPC provides the `split_64_8()` function to split the value in the first parameter into 8 separate variables, whose addresses are provided in the next 8 parameters. Assume that the array referring to OBM is called `OBMARR` and that the BRAM array we want to copy data from OBM into is called `BRAMARR`. Then we might expect to use the following code to copy N values from OBM into BRAM:

```
int i;
unsigned char b0, b1, b2, b3, b4, b5, b6, b7;
for (i = 0; i < N; i++)
{
    split_64to8(OBMARR[i], &b7, &b6, &b5, &b4, &b3,
      &b2, &b1, &b0);
   BRAMARR[(i * 8)] = b0;
   BRAMARR[(i * 8) + 1] = b1;
   BRAMARR[(i * 8) + 2] = b2;
   BRAMARR[(i * 8) + 3] = b3;
   BRAMARR[(i * 8) + 4] = b4;
   BRAMARR[(i * 8) + 5] = b5;
   BRAMARR[(i * 8) + 6] = b6;
   BRAMARR[(i * 8) + 7] = b7;
```

[9] We only need 1 bit per cell for the Game of Life, but the lattice gas requires more data per cell to store the cell's state. We thus chose to use 8-bit variables in preparation for the development of the lattice gas code.

```
}
```

Unfortunately using this method causes a significant problem later on in the program. The MAPC compiler has a limitation on the number of *lexical* writes that can be made to a BRAM array[10]. In the above code, we have made eight lexically distinct writes to `BRAMARR`. In the version of the Carte environment that we used, the limit of lexical writes was 8. Thus after loading the OBM into BRAM, it was not possible to perform any more writes to BRAM.

The solution to the problem, recommended by the compiler team at SRC, is to find an alternative way to load the data from OBM to BRAM using as few lexical BRAM writes as possible. Clearly it is still necessary to perform 8 logical writes. The following code uses a single lexical write by shifting the 8 values needing to be written through a single variable, which is repeatedly written to BRAM:

```
int i, j;
unsigned char b0, b1, b2, b3, b4, b5, b6, b7;
for (i = 0; i < N; i++)
{
  split_64to8(OBMARR[i], &b7, &b6, &b5, &b4, &b3,
    &b2, &b1, &b0);
  for (j = 0; j < 8; j++)
  {
    BRAMARR[(i * 8) + j] = b0;
    b0 = b1;
    b1 = b2;
    b2 = b3;
    b3 = b4;
```

[10]To understand why such a limit exists, consider that to implement a lexical write to Block RAM, the compiler has to generate a logic design that can take results from multiple sources and send it to one BRAM bank. This necessitates the use of a multiplexer, and clearly this can't be arbitrarily large.

```
        b4 = b5;
        b5 = b6;
        b6 = b7;
    }
}
```

We have thus reduced the number of writes used from eight to one. Unfortunately only having 7 remaining writes for each BRAM array is rather limiting, and it is not possible, or at least is impractical, to combine different types of writes in the way described above. This technique works well when there are a number of writes together, and they can easily be indexed by a loop. The cellular automata simulation algorithm fortunately does not involve many different stages where writing to BRAM is necessary, so we were able to successfully reduce our number of lexical writes below the compiler limit.

Build Results

Table 4.1 shows the results of the Place and Route stage after building a version of the program that implemented four processing engines on a single FPGA.

Table 4.1: Place and Route results for a one-FPGA, four-processing engine Game of Life cellular automata simulation.

Resource	Used	Available	% Used
Slice Flip Flops	23,620	88,192	26%
4 input Lookup Tables	13,454	88,192	15%
Occupied Slices	16,084	44,096	35%
Block RAMs	24	444	5%

The grid size was 128x128. Since each cell uses an 8-bit value to store its state, the total space required to store a configuration of the grid is $128 \times 128 = 16384$ bytes, which is 16 kbytes. However, we have broken the grid into

slices 32 cells high and 128 cells wide. In addition, each slice has an extra two rows, to store the ghost region, and two extra columns that act as a buffer on each side. Thus each slice actually consists of $(128+2) \times (32+2) = 4420$ cells. Each cell is a represented by a single byte. The Block RAMs store a maximum of 2048 bytes each. Thus to store a single slice, the FPGA needs to use $\lceil 4420/2048 \rceil = 3$ BRAMs. Therefore to store 4 slices, the FPGA needs to use 12 BRAMs.

When the cell transitions are being calculated we use two copies of the cell data — one to store the data from the current configuration, and another copy where we place the new configuration. Since we have two copies of the slice data, the FPGA uses 24 BRAMs in total.

Table 4.2 shows the Place and Route results after building a version of the program with eight processing engines.

Table 4.2: Place and Route results for a one-FPGA, eight-processing engine Game of Life cellular automata simulation.

Resource	Used	Available	% Used
Slice Flip Flops	43,413	88,192	49%
4 input Lookup Tables	25,504	88,192	28%
Occupied Slices	30,144	44,096	68%
Block RAMs	32	444	7%

As we would expect, the number of slice flip flops, lookup tables and slices approximately double between the four-engine and eight-engine versions. Our discussion above of why 24 BRAMs are used in the four-engine version is instructive, and a similar line of reasoning for the eight-engine version shows that each slice will contain $(128 + 2) \times (16 + 2) = 2340$ cells. Therefore each slice will be allocated $\lceil 2340/2048 \rceil = 2$ BRAMs. We have 8 slices, each stored twice, so we have $8 \times 2 \times 2 = 32$ BRAMs.

The overhead of the ghost regions for each slice is significant — 12.5% of each slice stored in BRAM contains ghost region (as opposed to 7.25%

for the four-processing engine version). In addition, a 128×128 grid is sufficiently small that rounding up to the nearest BRAM when allocating BRAMs to store slices results in significant waste. This is the primary factor that results in the eight-engine version using 33% more BRAMs than the four-engine version.

If we were to build a simulation with a far larger grid, the overhead would be far less significant, as would the waste of BRAMs resulting from rounding up during allocation. For example, if we have a 1024×1024 grid, the four-engine version would use 130 BRAMs per slice, and therefore 520 BRAMs in total. The eight-engine version would use 66 BRAMs per slice, and 528 BRAMs in total. In an ideal case, if there were no ghost regions, and the compiler allocated data across BRAMs rather than rounding up, we would need 512 BRAMs. Thus the eight-engine version uses approximately 3% more BRAMs than the BRAMs required to store just the cell state data, where BRAM allocation has been done without rounding up.

4.2.2 Performance Results

Figure 4.10 shows the performance results of our Game of Life simulation program on a 128×128 grid. We obtained a speedup of approximately 4x using the eight-processing engine simulator, and approximately 2.7x using the four-processing engine simulator.

We see that the speedup does not change significantly with the number of iterations, and that the amount of time taken to complete a given simulation depends linearly on the number of iterations, as we would expect.

Since the speedup of the four-processing engine version is 2.7x, we might reasonably expect the eight-processing engine version to produce a speedup of 5.4x. However, it is 4x. This discrepency likely arises due to *communication costs*. Specifically, we noted in the previous section that 12.5% of the data stored in the eight-processing engine version is that from ghost regions. At the end of every iteration, we have a serial process that updates each ghost region. Thus the amount of time taken to perform the communica-

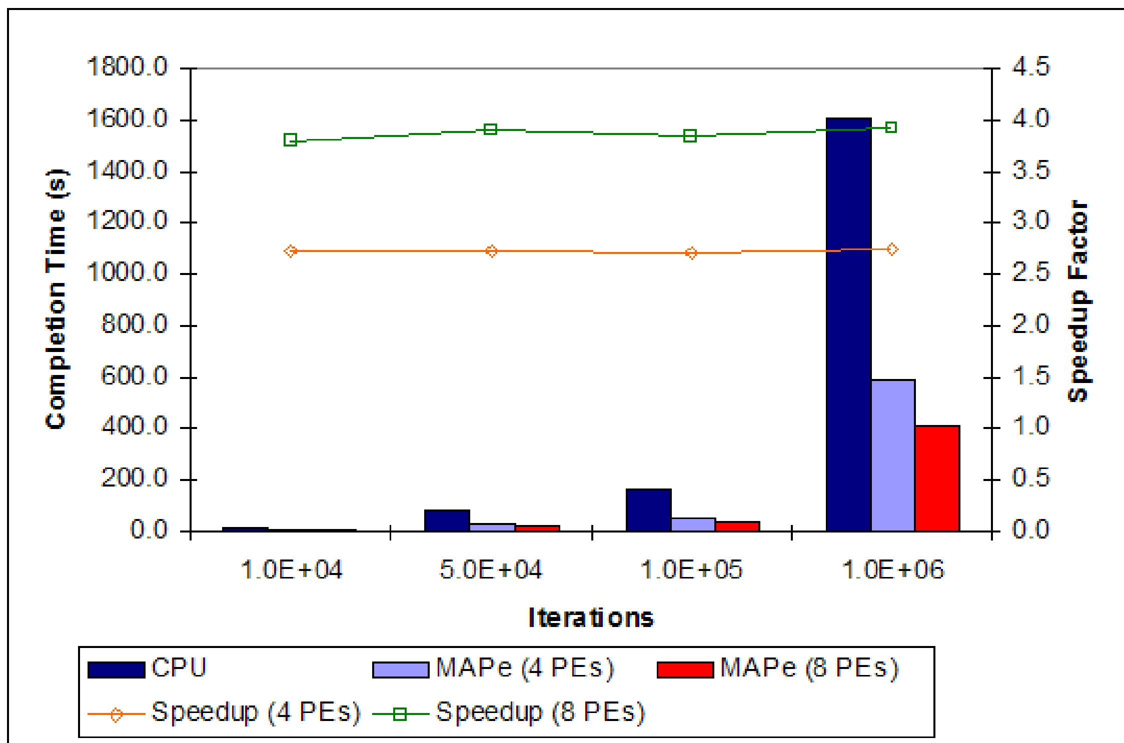

Figure 4.10: Performance of a Game of Life simulation on an SRC-6 MAPe, compared with that of an x86 processor.

tion between the processing engines at the end of every iteration depends on the size of the ghost regions, and the proportion of this time to the time it takes to compute a full transition is dependent on the proportion of data that stores ghost regions.

In the previous section we also noted that 7.25% of the data stored in the four-processing engine version is that from ghost regions. Thus there is a proportionately smaller communication cost for the four-processing engine version, and we can therefore expect it to run more efficiently. If we consider that the communication cost in the eight-processing engine version is likely about $12.5/7.25 \approx 1.7$ times greater than that in the four-processing engine version, and that the communication cost is likely a significant contributer to the running time, then it seems reasonable that we obtain a 4x speedup for our eight-processing engine simulation, rather than 5.4x.

We expect that if the size of the grid is increased to the point where the ghost regions consist of $< 1\%$ of the data, then doubling the number of processing engines will double the performance.

4.3 Fluid Dynamics Simulations using the Lattice Gas Method

Wolfram [35], and Frisch, Hasslacher and Pomeau (FHP) [36] both published seminal theoretical results in 1986 showing that it is possible to create a cellular automaton whose dynamics, in the continuum limit, simulate the Navier-Stokes equation for fluid flow. In general cellular automata methods for modelling fluid flow are called *lattice gas automata* (LGA). Frisch et. al. presented a model, commonly referred to as the "FHP" model, where a fluid is represented as a discrete lattice, where each vertex on the lattice is connected to six others. Fluid particles may reside at a lattice point[11], or be transiently passing into or out of a vertex.

[11] This was not part of the original FHP model, but is an extension that is now often used.

Figure 4.11 shows a sample FHP LGA lattice, which illustrates how transitions are made from one timestep to the next.

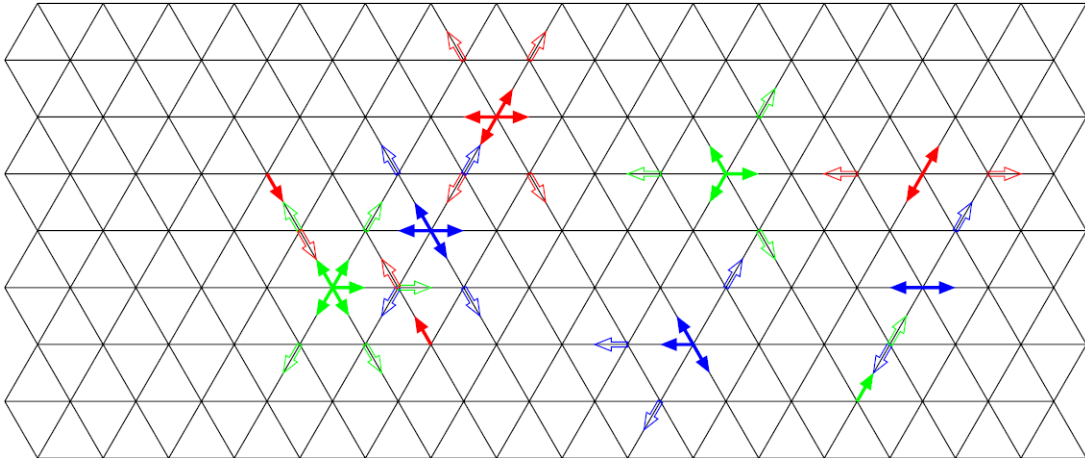

Figure 4.11: Transition of an FHP lattice gas automata. Solid arrows show the configuration at the current timestep, and hollow arrows show the configuration at the next timestep. Image from Luo, ref. [37].

Each vertex on the lattice is a point in space. It is assumed that fluid particles can only travel along the edges of the lattice. For any particular lattice point (vertex, or in the language of cellular automata, 'cell'), at any particular timestep, we can describe the state of the cell by observing the six edges connected to it. Each edge will either have a particle moving away from the vertex on it, or will be empty. The vertex can also have a transiently stationary particle located there.

Because the fluid particles can only travel along the lattice edges, we can define finitely many possibilities for what happens when particles collide with each other at lattice points. Figure 4.12 shows a few examples of rules that can be applied to transition an input state (the state at the present timestep) to an output state at the next timestep. In some cases there are multiple possibilities for the results of a collision; this figure shows both in the two of examples of this.

We assume the model whether there can be particles entering the vertex

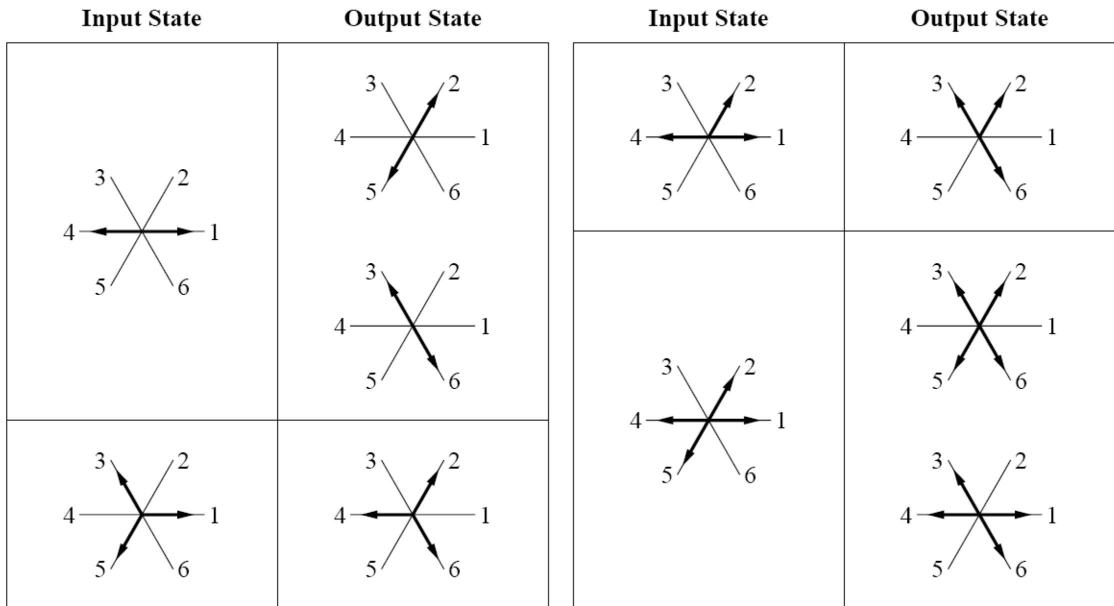

Figure 4.12: Example transition rules for FHP lattice gas automata. Image from Luo, ref. [37].

on any of the six edges, and there can be a particle that is stationery at a vertex for one or more timesteps. Therefore in this model there are a total of $2^7 = 128$ states. The rules simply map each state to one or more other states. These rules have been carefully designed so that they are physically realistic[12].

4.3.1 Design and Implementation

Since the lattice gas automata model is just an extended cellular automata model, we were able to reuse a large portion of our design for the Game of Life simulation in designing the lattice gas simulation program.

The principle differences between the Game of Life and the lattice gas automata are:

[12]By this we mean that there is conservation of energy and moment. For example, it would not be physically realistic to have an input state that shows three particles entering a vertex, and have an output state that shows just one particle leaving.

1. The space in which the simulation takes place is not basic grid. Rather it is a lattice where each vertex is connected to six neighbours, as shown in Figure 4.11.

2. Each cell can be in one of $2^7 = 128$ states in the lattice gas, whereas in the Game of Life each cell could only be in one of two states.

3. In the Game of Life, we didn't need to explicitly define the transition function f, since the transition rules could be encapsulated in a pithy rule. With the lattice gas, this is not possible, and it is necessary to explicitly define each of the 128 transitions.

Our changes to the Game of Life design take into account these differences between the two otherwise similar models.

As with the Game of Life, we parallelized the problem using a data parallelization strategy. Again we opted to simply divide the lattice into slices, and assign each slice to a processing engine. The communication scheme used to update the ghost regions is identical.

Figure 4.13 shows the design of the data flow in the system.

Each cell can be in one of $2^7 = 128$ states. Therefore 7 bits can be used to represent the state of a single cell. MAPC does not allow for arbitrary size types, so we store cell states in the `unsigned char` type, which is 8 bits long. We also allow for a cell to be marked as a 'boundary' cell, which is a cell that is a solid object and which fluid particles will always bounce off. By defining a group of boundary cells together in the initial lattice configuration, we can effectively define an obstacle around which the fluid must flow.

Each cell state is therefore recorded by $7 + 1$ bits, stored in a byte — the highest order bit records whether or not the cell is a boundary. The remaining 7 lower order bits each record whether or not there is a particle entering the cell at that particular timestep in one of the six lattice edge directions, or if there is a particle transiently stationary at the cell.

Because in the lattice gas we need to use lookup tables to implement the transition function, the system needs to store more data than the Game of

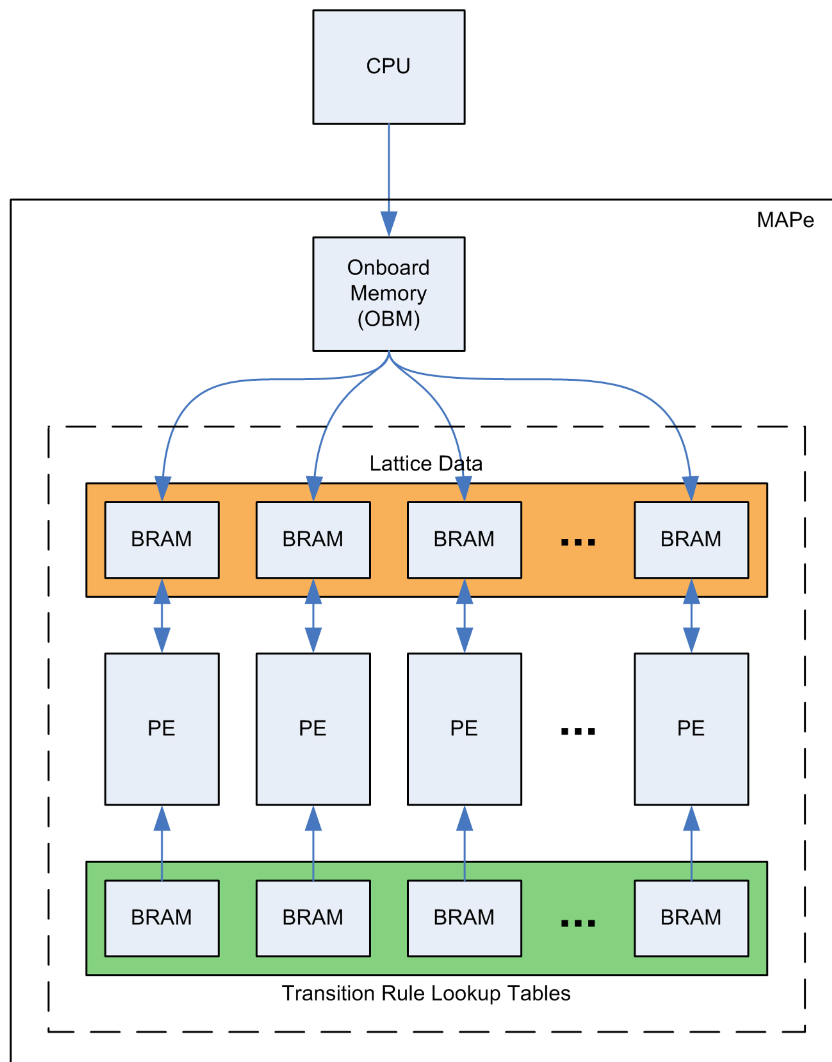

Figure 4.13: Layout and flow of the data in the system during a lattice gas automata simulation.

Life system. Each processing engine needs to remain independent during the transition of one configuration to the next, so we have designed the system so that each processing engine has a dedicated lookup table in BRAM. Since the lookup table is used often, it needs to be placed in memory that can be accessed quickly. BRAM is thus preferred. In addition, we want to avoid the possibility of contention amongst processing engines for access to the lookup tables, and providing each processing engine with its own lookup table resolves this.

Build Results

Table 4.3 shows the Place and Route results after building a version of the program with five processing engines, for a lattice size of 480×480.

Table 4.3: Place and Route results for a one-FPGA, five-processing engine Lattice Gas Automata simulation.

Resource	Used	Available	% Used
Slice Flip Flops	37,629	88,192	42%
4 input Lookup Tables	21,039	88,192	23%
Occupied Slices	25,246	44,096	57%
Block RAMs	325	444	73%

The limiting resource in this instance happens to be the number of Block RAMs available, since every processing engine needs to be assigned its own lookup table in BRAM. A four-processing engine design was built, and it required 260 BRAMs, or 58%. Thus it may be possible to build six-processing engine version, but to add more processing engines we would need to modify the design to decouple the lookup tables from the processing engines. Of course this would have performance implications, and we would need to find a way to compensate for the fact that two or more processing engines would have to share each lookup table[13].

[13] One possible design idea, which we did not attempt to implement, would be to stagger

4.3.2 Performance Results

Figure 4.14 shows the performance results for the five-processing engine version of the program, running a simulation on a 480×480 lattice. The reconfigurable computer delivers a 1.7x speedup over the x86 processor.

The speedup gained is less than the speedup that the four-processing engine Game of Life program obtained over its x86 counterpart. This may be a result of the fact that the transition algorithm for the Game of Life is far simpler than that for the lattice gas, and that the reconfigurable computer may not be able to speed up the serial portion of the cell transitions in the lattice gas as much as it can the transitions in the Game of Life program.

4.4 Conclusion

We successfully implemented two cellular automata simulation programs on the SRC-6 MAPstation reconfigurable computer: Conway's Game of Life, and an extended FHP Lattice Gas. We also implemented these programs in standard C code for the x86 platform to make performance comparisons.

We obtained a speedup of approximately 4x in the Game of Life simulation, using one FPGA and eight processing engines, versus a Xeon 2.8GHz CPU. We obtained a speedup of approximately 1.7x in the Lattice Gas Automata simulation, using one FPGA and five processing engines.

In the case of the Game of Life, the number of slices on the FPGA were the limiting resource to scaling to greater numbers of processing engines. However, in the Lattice Gas Automata program, the number of Block RAMs was the limiting resource.

---

the processing engines in such a way that no processing engine ever attempts to read from a lookup table while another is reading. Since the loops take 8 clocks per iteration, and BRAM reads take only one clock cycle, it may be possible to share one BRAM lookup table amongst 8 processing engines. If you could share one lookup table amongst 8 processing engines, the available BRAM would then likely no longer be the limiting factor in building the design, but rather the available slices would.

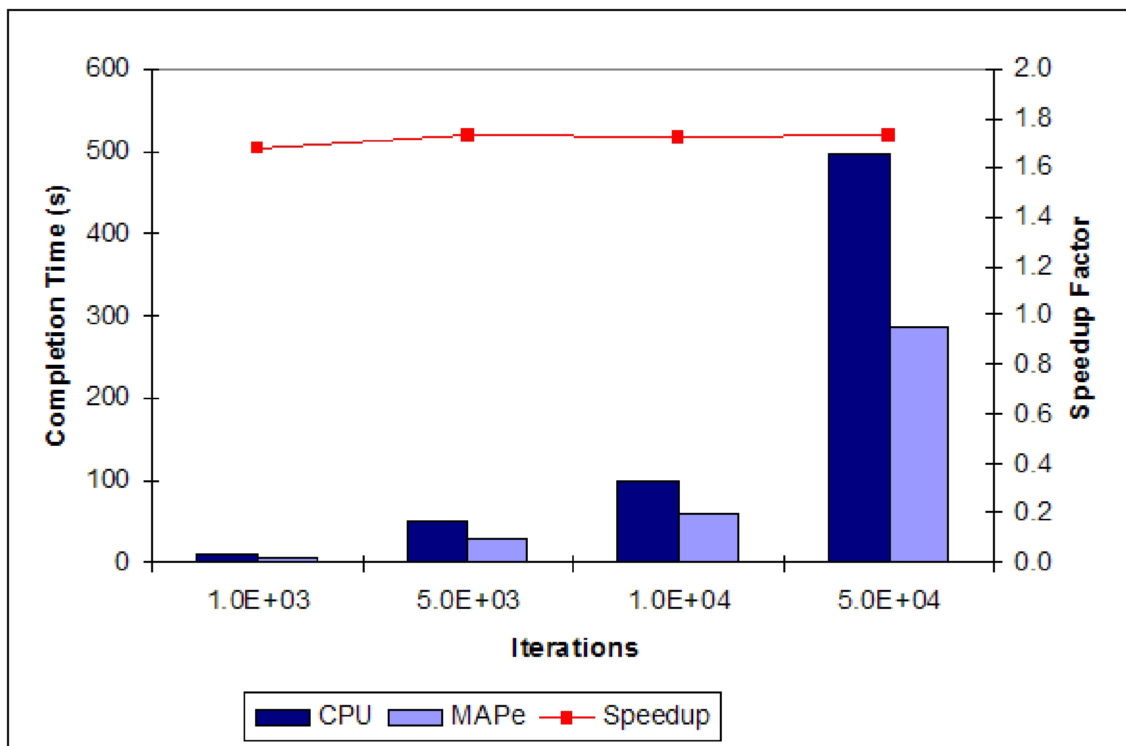

Figure 4.14: Performance of a Lattice Gas Automata simulation on an SRC-6 MAPe, with 5 processing engines, using one FPGA, compared with that of an x86 processor.

Chapter 5

Image Processing – Edge Detection on Reconfigurable Computers

FPGAs have been extensively used in digital signal processing applications, including image processing [39, 40]. Image processing applications tend to use fairly simple algorithms that are computationally intensive due primarily to large data sizes and rates, since as real-time video processing.

In this chapter we present our implementation of Sobel edge detection on the SRC-6, and find that for this application the reconfigurable computing platform delivers a modest speedup over a pure microprocessor architecture.

5.1 An Introduction to the Sobel Edge Detection Algorithm

Sobel Edge Detection [41] is one of the early, simple-yet-effective edge detection algorithms, and forms the basis of Canny Edge Detection [42], which is considered the optimal edge detector. We now present a brief introduction to Sobel's method for edge detection.

We can consider an image as a discrete function with two non-negative

integer parameters, $f : \mathbb{N}^2 \rightarrow \mathbb{N}$, which maps a pair of spatial coordinates to a non-negative integer value representing colour intensity[1]: $(x, y) \mapsto z$, where $x, y, z \in \mathbb{N}$.

One approach to finding edges in an image is based on the idea that an edge is an area in an image whether there is a fast change in the values of the image function, and that this can be captured by finding the gradient of the image function, ∇f.

In the case of the image function, the gradient can be defined as a combination of the partial derivative of the function in the horizontal (x) direction, and the partial derivative of the function in the vertical (y) direction:

$$\nabla f = \frac{\partial f}{\partial x}\vec{i_x} + \frac{\partial f}{\partial y}\vec{i_y}.$$

Here $\vec{i_x}$ and $\vec{i_y}$ are the unit vectors in the x and y directions respectively.

Clearly it is not possible to find the partial derivatives analytically. Indeed, since f is not continuous, we can at best find approximations to the partial derivatives of the underlying continuous image function. Once we have determined approximations to the partial derivatives, we obtain an approximation of the magnitude of the gradient, $|\nabla f| = \sqrt{(\frac{\partial f}{\partial x})^2 + (\frac{\partial f}{\partial y})^2}$. If the gradient magnitude is above a certain threshold at a point (x, y), it is likely that the pixel at (x, y) forms part of an edge, and we mark it as such.

One method for approximating the partial derivatives $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ is to convolve the image function with specifically constructed *convolution masks*[2].

[1]This assumes that we are working with a greyscale image, where the intensities may range from, for example, 0 as black, to 255 as white, and values inbetween represent varying shades of grey. We can extend our model to full colour images, which may represent the colour at a particular coordinate pair as a set of integers. For example, one common model is to use a triplet (r, g, b), where the values represent the intensities of red, green and blue components in the pixel colour. One simple method of extending the greyscale model of edge detection to full colour is by averaging the colour intensities, or using some other method to convert the image to greyscale, before continuing with the standard greyscale edge detection algorithm. However, in this chapter we work only with greyscale images.

[2]In the literature these may also be referred to as *convolution kernels*, *gradient filters*

A 3×3 approximation to the partial derivatives is given by convolution with the *Prewitt operators* P_x and P_y [43]:

$$P_x = \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}$$

$$P_y = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}$$

We can find an approximation to $\frac{\partial f}{\partial x}$ at a particular point (x, y) by convolution:

$$\frac{\partial f}{\partial x} = P_x \otimes f(x, y).$$

The convolution used is the standard 2D convolution:

$$P_x \otimes f(x, y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} P_x(i+1, j+1) f(x-i, y-j)$$

The approximation to $\frac{\partial f}{\partial y}$ at a particular point (x, y) is similarly given by:

$$\frac{\partial f}{\partial y} = P_y \otimes f(x, y).$$

The *Sobel operators* S_x and S_y [41] are similar to the Prewitt operators, but give a greater weighting to the centre pixel:

$$S_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

$$S_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

---

or *operators*.

In Sobel edge detection, the partial derivatives are approximated using convolution with the Sobel operators.

We can find the magnitude of the gradient using the square root of the squares of the partial derivatives, but both multiplication and finding square roots are expensive operations. Thus in practice the further approximation $|\nabla f| = \left|\frac{\partial f}{\partial x}\right| + \left|\frac{\partial f}{\partial y}\right|$ is used. If, for a certain threshold T at a point (x, y), $|\nabla f| > T$, then at that point we draw a dark pixel to represent that that point is part of an edge. Otherwise we draw a light pixel. We repreat this for all points in the image function f.

5.2 Edge Detection on a Reconfigurable Computer

5.2.1 Design and Implementation

We designed and implemented a program to perform edge detection on an input bitmap image on the SRC-6 MAPstation.

In some respects, the edge detection algorithm and computing problem is similar to that of cellular automata algorithms. In both problems we must iterate through a large volume of 2D data, and on each element perform some computation involving it and its neighbours. Given this similarity, there are similarities in our design and implementation approaches to both problems.

We parallelized the edge detection algorithm by using a data parallelization scheme, as we did with the cellular automata algorithms. In particular, we divided an input image into three slices, which can be processed independently of one another by three processing engines. Figure 5.1 shows the algorithm design for our edge detection application. The loop in each processing engine was flattened to allow the compiler to optimize and pipeline it. Ultimately each loop required 8 clock cycles per iteration.

One of the major differences between our design for the cellular automata programs and this design for edge detection is that we have accounted for the

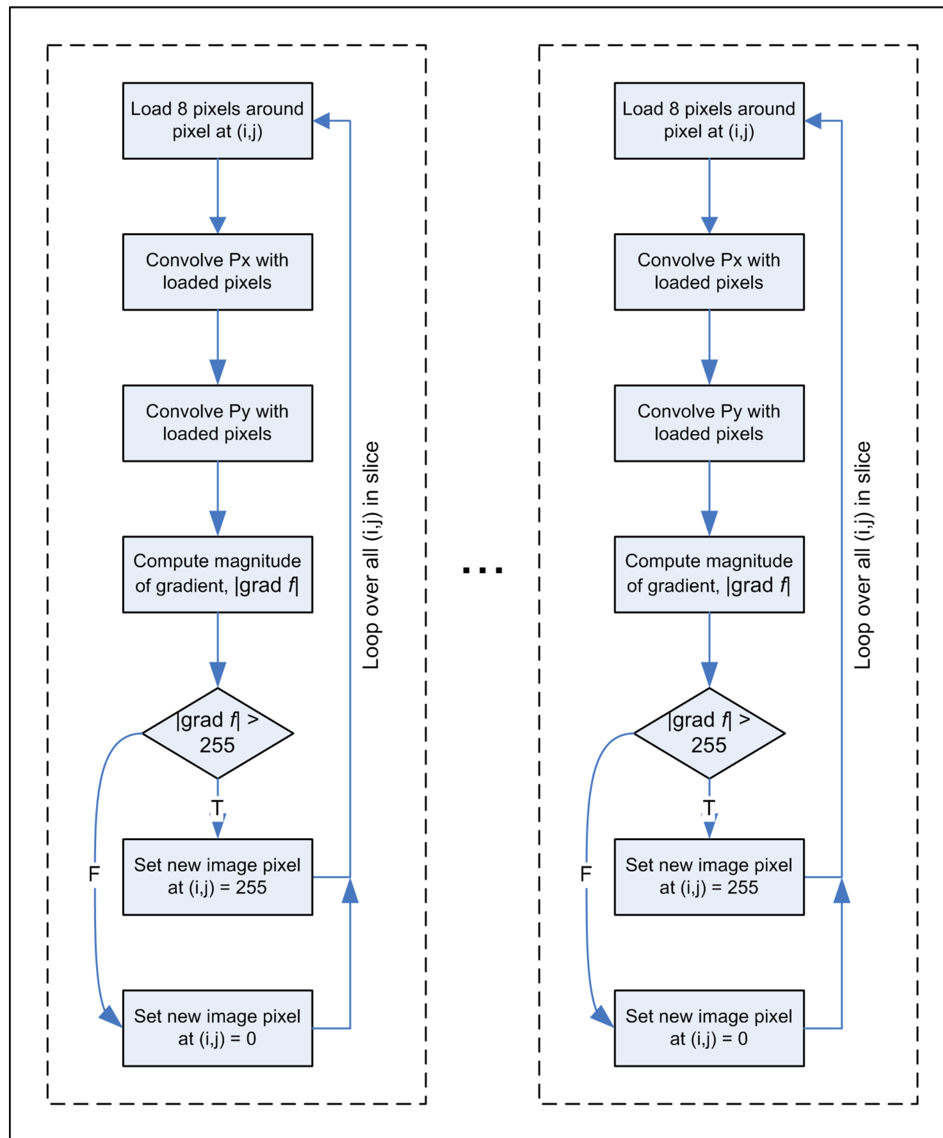

Figure 5.1: Processing engines each performing Sobel edge detection on a slice of the input image.

fact that in edge detection there is only one iteration, whereas in a cellular automata simulation there are many.

In our cellular automata program designs we made the assumption that loading data from onboard memory into block RAM at the start of the program, and writing it back from BRAM into onboard memory at the end, would take a negligible amount of time compared to the total execution time, and would thus definitely be worth the difference it makes.

However, with edge detection this assumption is no longer valid. Loading data into BRAMs, which essentially act as a cache, when each element will be read only 8 times is no longer a strategy that will reduce the clock cycles per loop iteration by an order of magnitude.

Instead we created a design for the flow of data that is significantly different to that of the cellular automata programs. Figure 5.2 shows the layout and flow of data in the program. The image data is transferred to onboard memory using direct memory access. The data is split into three slices on the CPU, so the first three onboard memory banks receive the data for the three slices. The SRC-6 has six onboard memory banks, and the remaining three are used by the three processing engines to write out the resulting image.

It is necessary for each processing engine to have its own dedicated onboard memory bank, since each bank can only serve one request at any one time, so having multiple processing engines per bank would require logic to prevent contention.

Build Results

The build results for the program are shown in Table 5.1. Since only three processing engines are used, the resource utilization on the Virtex II FPGA is not particularly high. With our design the limiting resource is not on the FPGAs, but rather the number of memory banks in the MAPe module that can be concurrently accessed.

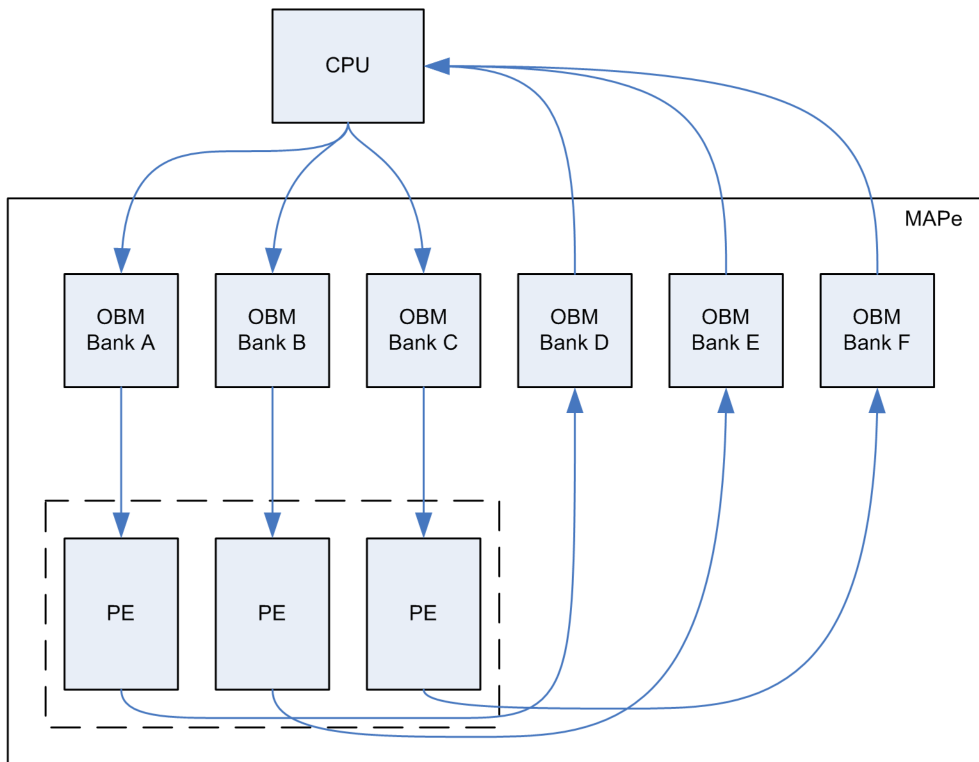

Figure 5.2: Layout and flow of the data in the system during edge detection.

Table 5.1: Place and Route results for a one-FPGA, three processing engine edge detection program.

Resource	Used	Available	% Used
Slice Flip Flops	16,089	88,192	18%
4 input Lookup Tables	8,452	88,192	9%
Occupied Slices	10,290	44,096	23%

5.2.2 Results

Figure 5.3 shows an input image and the output produced by the edge detection algorithm.

Figure 5.3: Original image (left). Image with detected edges shown (right).

Performance Results

The total execution time for edge detection on a 'reasonable' size image is very small. The largest image we tested the application on was 700 by 2100 pixels, and all the variants of both the classical and RC programs we tested the image on took a second or less to complete.

For such short execution times we can't draw any reliable conclusions, since the time is dominated by tasks other than the edge detection calculation — in the classical implementation, the time to load the input image and write the output image to disc is dominant, and in the RC implementation the IO time, as well as the time taken to load the FPGA design is dominant.

Therefore to record meaniful performance data, we inserted timing code into the classical and RC implementations to determine exactly how long the edge detection computation was taking in each. In the classical program the times reported are the durations between when the image data has been loaded into an array in memory and when the edge-detected image has been

Table 5.2: Performance of Sobel Edge Detection on an SRC-6 MAPe, with three processing engines, using one FPGA, compared with that of an x86 processor.

Input Image Size	CPU Computation Time (s)	MAPe Computation Time (s)	Speedup Factor
700×2100	0.0646	0.0388	1.67

produced in memory. The times reported for the RC program are the durations between when the DMA transfer of data into the FPGA has completed, and just before the DMA transfer of the results out of the FPGA is about to begin.

To confirm our timing results, we also modified the edge detection programs (both the x86 and RC versions) and made each perform the edge detection calculations several thousand times, and recorded how long the entire execution took. In this scenario, the overhead times no longer dominate, and we were able to verify that our timing results were accurate.

Table 5.2 shows the performance results. A modest speedup factor of 1.67x is achieved over the x86 processor.

One aspect of the implementation that could possibly be improved upon is the reuse of data. Reads from onboard memory are relatively expensive, and our implementation makes 8 reads to an OBM bank per pixel that is checked. We could plausibly reduce the average number of clocks per iteration by caching pixel data that we have already read from the onboard memory.

Figure 5.4 illustrates the concept. The black square represents the pixel that is currently under inspection. The blue squares represent pixels that have not yet been read from memory, and orange squares represent pixels that have been read from memory before, and thus may be cached. Clearly the very first calculation will need to load all required pixels from onboard memory (stage 1). We can see that by the time we reach the second pixel (stage 2) we can save two reads from onboard memory. By the time we reach

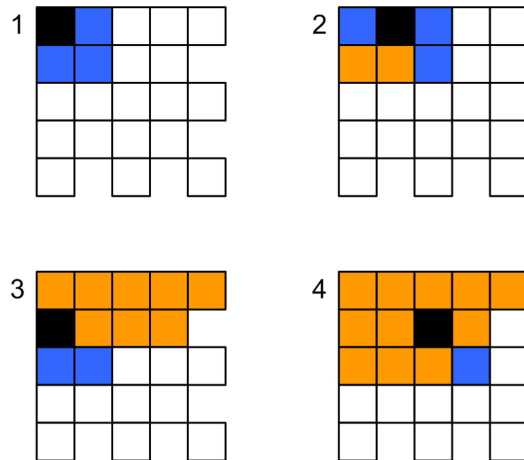

Figure 5.4: Caching pixels to reduce the number of reads from onboard memory.

the start of the second row (stage 3), we can save three reads, and when we're beyond the first pixel in the second row onwards (stage 4), we only need to make one read from onboard memory per pixel gradient calculation.

With this improvement our design is likely to yield a speedup near 10x.

If more concurrently-accessible memory banks were added to the MAP module, it would be possible to add more processing engines to our design, which would result in a linear increase in speed, since the algorithm is embarrasingly parallel and involves no communication.

5.3 Conclusion

We successfully implemented Sobel edge detection on the SRC-6 MAPstation reconfigurable computer. Our design used all the available memory banks on the SRC-6 MAPe module in an attempt to maximize the number of areas of an input image that can be concurrently accessed.

We obtained a speedup of 1.67x over a Xeon 2.8GHz CPU, and have provided a design idea involving the caching of pixels that are read which may result in speedups of approximately 10x.

Chapter 6

Automatic Macromolecular Docking on Reconfigurable Computers

Cryo-electron microscopy (cryo-EM) is a commonly-used and important technique in macromolecular structure determination [44]. Recently there has been interest in finding methods that can dock high-resolution structures, determined using X-ray crystallography or nuclear magnetic resonance techniques, into the low-resolution density maps produced by cryo-EM [44, 45, 46].

Figure 6.1 shows an example of a macromolecular model docked in a density map. Given the density map and the model, the problem is to find the best docking of the model into the map.

The methods that have been developed for finding dockings are in principle fairly simple. The key idea used by Roseman [45], Wriggers et. al. [44, 46] and others[1] is that we can measure how well a particular placement of a model within a density map 'fits' by computing the correlation between the model and the map. The objective of a broad class of docking algorithms is to find the placement of a given macromolecular model within a given

[1] The list of references in [46] provides comprehensive coverage of the field.

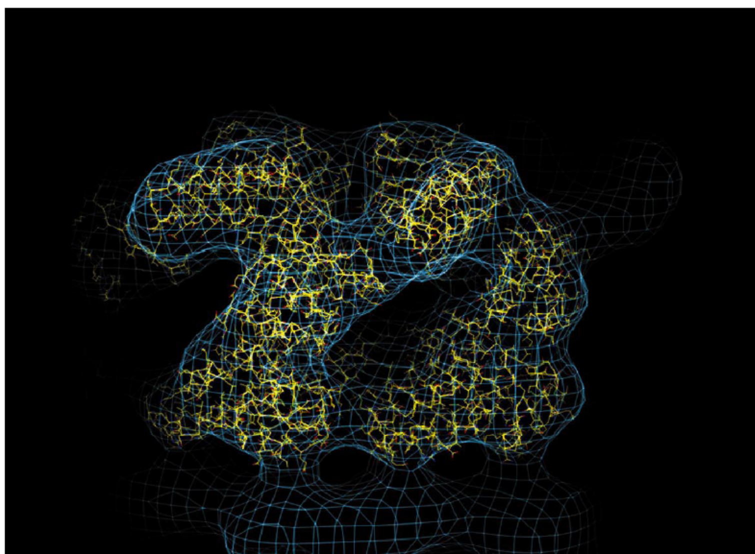

Figure 6.1: A macromolecular model (yellow) docked in an electron microscope density map (blue mesh). Image from ref. [45].

density map that yields the highest correlation. This is done using a 6D search: the position and orientation of the 3D density map is kept constant, and the macromolecular model is systematically translated and rotated. At each new configuration of the model, the correlation between the model in its new configuration, and the density map, is computed. The rotational and translation parameters that result in the highest[2] correlation are stored. Figure 6.2 illustrates how a model may be translated and rotated to find a best fit.

[2]The available docking packages typically store a list of the n highest correlations and the rotation and translation parameters that resulted in them, rather than just the single highest correlation.

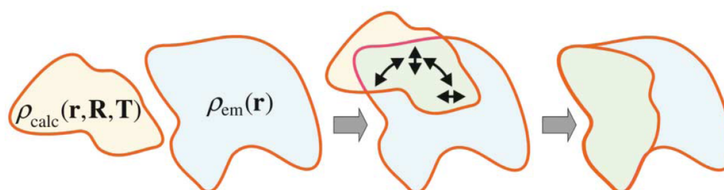

Figure 6.2: Fitting a macromolecular model by rotation and translation. Image from ref. [46].

6.1 Macromolecular Docking using Global Correlation

Wriggers et. al. note in [46] that for typical data sizes, correlation-based docking algorithms take one or more hours of CPU time on current high performance workstations. Clearly any reduction in the compute time of docking programs would be welcome.

Using the DOCKside docking package, developed by Snowden et. al [47] as our base, we have designed and implemented the global correlation docking algorithm for the SRC-6 MAPstation.

The computational expense of docking is a result not of there being an inordinate amount of data to process, but rather that the requisite 6D search quickly results in large compute times as the size of the data increases beyond trivially small. This is due to the fact that 6D search has a computational complexity of $\mathcal{O}(n^6)$, or $\mathcal{O}(n^3 \log n^3)$ using *Fourier-accelerated* docking [46]. The computation time rises quickly with the size of n, even in the accelerated case.

The fact that variants of the docking algorithm exist that use the Fourier transform to accelerate the algorithm is a boon, since FPGAs have long been used to accelerate fast Fourier transforms in DSP applications.

6.1.1 Design and Implementation

The algorithm for global correlation docking is summarized in Figure 6.3. The EM map is loaded, and may be filtered. If the Fourier-accelerated docking method is being used, then the fast Fourier transform is applied.

Next, the atomic structure is loaded. Sets of *Euler angles* Θ, Φ, Ψ are generated. These angles are going to be used to rotate the molecular model, and are generated in such a way that guarantees that there is a regular spacing of rotations.

With the structure loaded and the angles generated, the iterative part of the algorithm can begin. We start by generating a rotation matrix $\mathbf{R}$ from the first set of Euler angles, and rotating the atomic model using it. Once the model has been rotated, it is filtered so that it is at a similar resolution to the density map. Next the correlation between the rotated, filtered model and the density map is computed. Since this is the first iteration, we will save the computed correlation and the Euler angles used in this iteration. In general the computed correlation will be compared to the previous highest correlation, and if it is larger, it will become the highest correlation, and the Euler angles used will be stored.

We note that each iteration of the algorithm happens in isolation, and that the algorithm can therefore be parallelized in the following way: we can separate the generated rotations into p sets, and assign each of p processors a set of rotations. Each processor can use the iterative docking algorithm to find a *locally maximum* correlation, and the corresponding Euler angles leading to a rotation of the model that resulted in that correlation. Once all the processors have computed the locally maximum correlation for their sets of Euler angles, a master processor can find the maximum of the local correlation maxima, and hence obtain the globally maximum correlation, and its corresponding Euler angles.

Since resources on FPGAs are quite severely limited, we wish to create a design that uses the FPGA only for the most computationally intensive parts of the algorithm. Figure 6.4 shows how the parallelized version of the

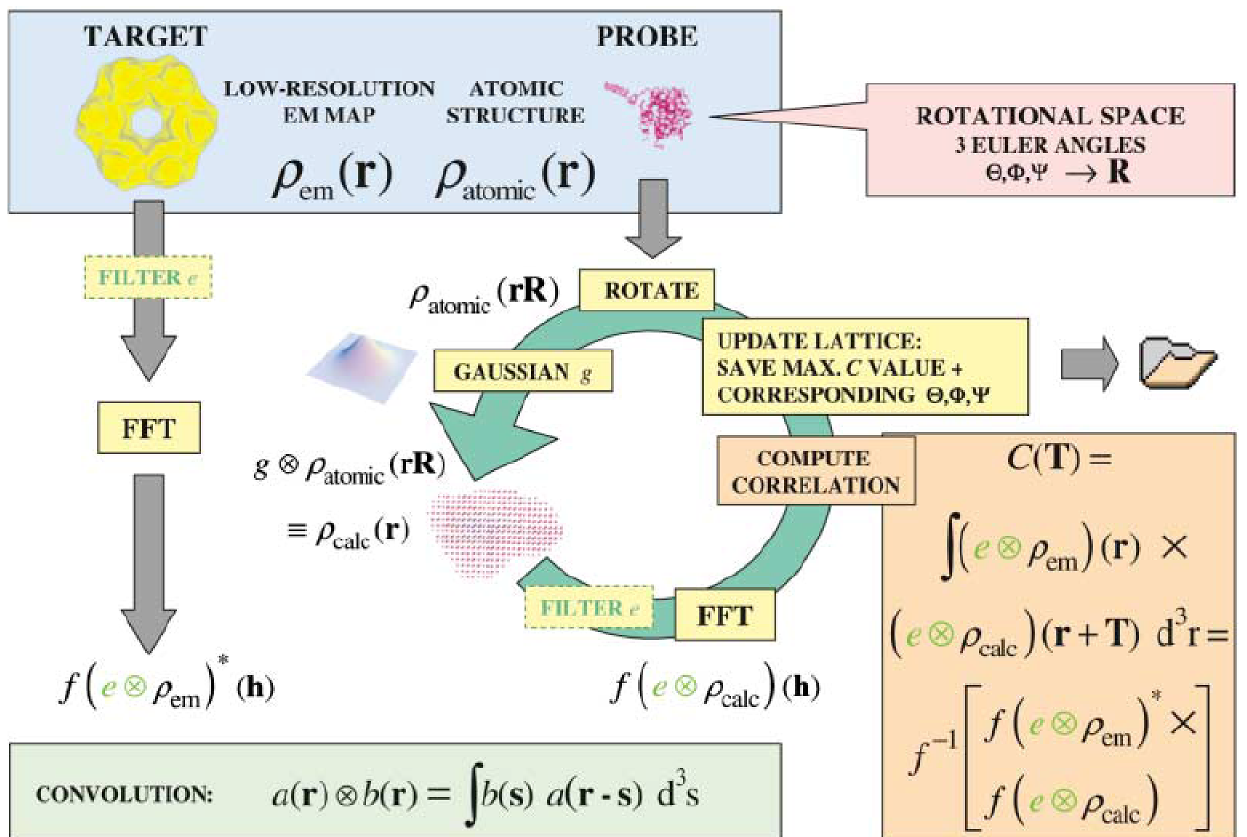

Figure 6.3: An overview of the global correlation docking algorithm. Image from ref. [46].

global correlation docking algorithm may be implemented on the SRC-6.

6.1.2 Results

We were successful in developing an implementation of global correlation docking in MAPC. However, during development we were repeatedly set back by deficiencies or limitations in the language. We used Snowden's DOCKside package as our reference implementation. DOCKside was written in C++, so the first step towards obtaining a functioning MAPC implementation was to port the DOCKside docking engine implementation from C++ to C. This is in itself an unpleasant exercise, as all the abstractions that C++ allows need to be unpacked.

The process of porting object-oriented C++ code to C is made more difficult by the fact that MAPC does not allow for dynamically sized arrays. All data to be stored in Block RAM needs to have a known size (or at least a known upper bound), so it is necessary to calculate such upper bounds based on the parameter values that can be expected. This requirement is, of course, motivated by the fact that there is only a set amount of Block RAM on an FPGA, and unlike with a microprocessor, it is not possible to rely on virtual memory if your program happens to request more memory than is physically available.

Nevertheless, regardless of the motivation of the requirement, it is an impediment to productivity.

We were able to create an implementation of the docking algorithm that builds correctly under the SRC Carte debug (simulator) environment. However, when we attempted to build the program for use on the SRC-6 hardware, the Xilinx build tools crashed after the machine performing the build ran out of RAM (of which it had 3GB).

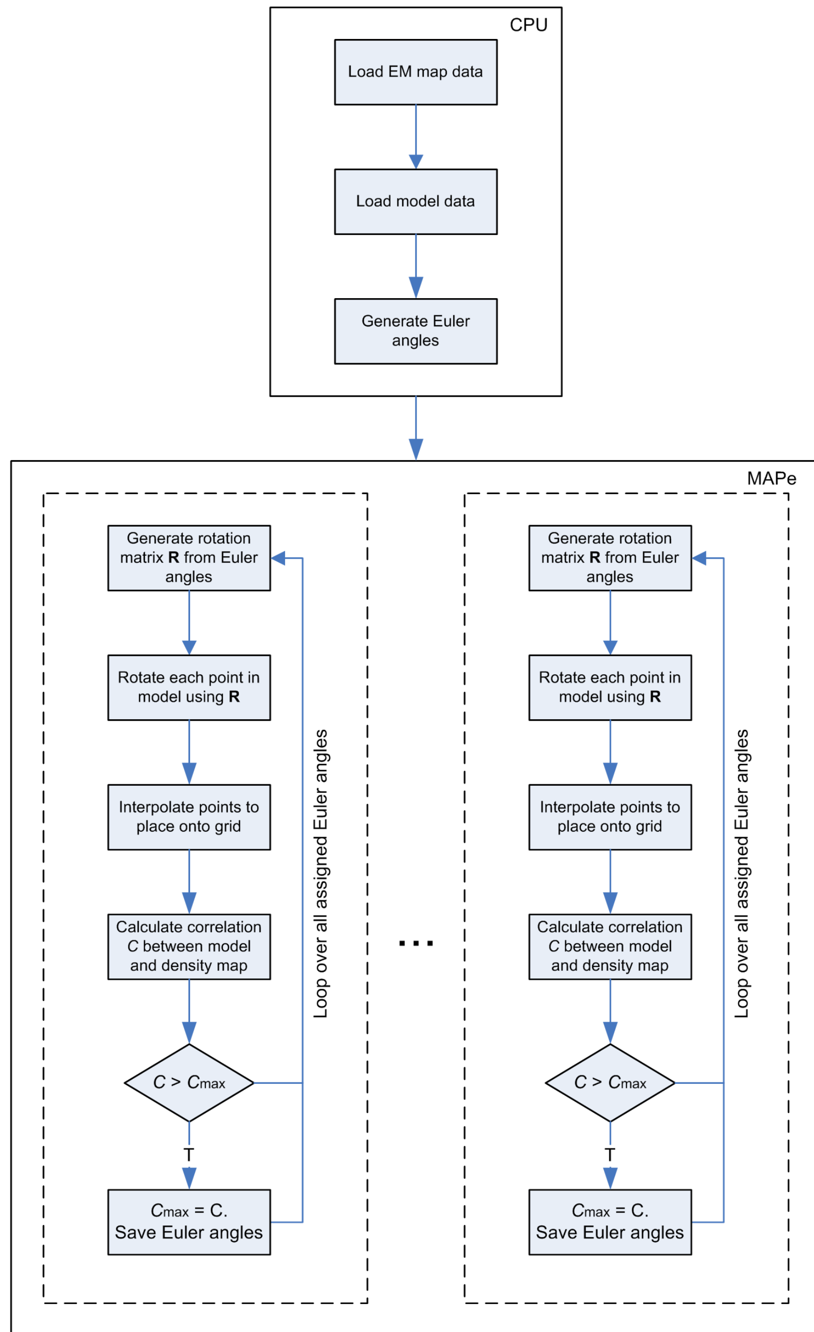

Figure 6.4: Processing engines each search to find a maximum correlation for the Euler angles they are assigned.

6.2 Conclusion

We identified the macromolecular docking problem as one that currently requires lengthy computation times, and that may be amenable to speedup on a reconfigurable computer. We designed a parallelized algorithm based on the global correlation docking algorithm, and implemented a version of the algorithm in MAPC for the SRC-6. The program built successfully in the SRC simulator environment, but unfortunately the build process failed while creating a hardware design for the program, as the build server ran out of RAM.

Although the SRC toolset for programming reconfigurable computers is one of the leaders in the field, the environment is still rather immature, and even porting code from C++ to MAPC is an exercise that is perhaps unnecessarily difficult.

Chapter 7

Conclusion

7.1 Results

In this thesis we have described our successful design and implementation of several scientific computing and engineering applications on an SRC-6 MAPstation reconfigurable computer:

- Monte Carlo estimation of π
- Monte Carlo financial options pricing
- Cellular Automata Game of Life
- Cellular Automata Lattice Gas simulation
- Sobel Edge Detection

We managed to obtain speedups for all these applications. In addition, we implemented a macromolecular docking algorithm, which we were unable to build for the SRC-6 due to the build cluster not having enough RAM.

7.2 Analysis

Although the speedups we obtained were all less than 10x, we found in several of the algorithms encouraging scaling results that indicate that as the density

of FPGAs increases, and as more dedicated hardware multipliers are added by the vendors, the performance of the applications we implemented will increase proportionately.

The development of software for the SRC-6 using the MAPC language was a considerable improvement on using a hardware description language such as Verilog or VHDL. However, the software development environment is still very immature and is lacking in both language features (such as pointers, dynamic arrays, object-orientation) and library support (little more than the most basic mathematical functions are currently provided). In addition, programming the SRC-6 in MAPC requires a fairly thorough knowledge of the quirks of the language and the compiler — for example, it is necessary to know where the compiler will store data based on what type of variable you used[1]. There are also numerous compiler limitations that hinder development. For example, there is a fairly low limit on the number of lexical writes that can be made to a static array or a variable[2].

In summary, we found that the Carte development environment and MAPC were far easier to build scientific applications with than Verilog or VHDL, but that considerable work is still required before using such tools will be as easy as developing software for traditional computing platforms.

[1] Statically defined arrays are implemented in BRAM, but static variables are implemented in the FPGA fabric.

[2] As at time of writing, the limit was 16.

Appendix A

Monte Carlo Methods

A.1 A Review of Monte Carlo Methods

Monte Carlo methods originated [11] in the first half of the 20th century as a useful technique when performing scientific simulations. Many scientific problems involve the evaluation of integrals [12], and in scientific computation, these integrals often have no analytical solutions and furthermore resist traditional numerical solution due to their involving a high number of dimensions.

A.1.1 An Early Monte Carlo Algorithm

Any algorithm that involves at its core the use of random numbers is generally referred to as one that uses a *Monte Carlo* method. One well-known algorithm for estimating π was conceived by Georges Louis Leclerc Comte de Buffon in 1777 [13]. de Buffon stated that if one drops a needle of length l onto a flat surface, which has parallel lines each a distance D apart, with $D > l$, drawn on it, then the probability that the needle will intersect one of the lines is $2l/\pi D$. From this observation, de Buffon reasoned that it would be possible to estimate the value of π by repeatedly dropping a needle onto such as surface, and recording how many times the needle intersected a line. If the experimenter performs N drops, and p is the number of times the

needle intersected a line divided by N, then an estimate of π is obtained[1]:

$$\widehat{\pi} = \frac{2l}{pD}$$

If $N \to \infty$, then $\widehat{\pi} \to \pi$. In other words, if a large number of drops is performed, the estimate of π, $\widehat{\pi}$ will approach the actual value of π. This happens to be a highly inefficient technique for calculating π (even crude Taylor Series Expansion-based methods are considerably better), but de Buffon's method illustrates the important concept of how an experiment involving an element of randomness can be used to compute a deterministic quantity.

A.1.2 Numerical Integration using Monte Carlo Methods

Estimating π using de Buffon's technique may be a neat trick, but as we mentioned, it's much slower than other known methods of calculating π. Are there any problems that can be solved using Monte Carlo techniques more efficiently than other methods? If so, what are they?

Of course, Monte Carlo methods wouldn't have become a useful technique in scientific computing if they weren't faster than other methods for at least some important problems. It so happens that numerical integration is one such problem, and that in fact numerical integration underlies a wide variety of scientific computing problems.

Given the task of computing the integral I, where

$$I = \int_R f(\mathbf{x}) d\mathbf{x},$$

regular numerical integration techniques become ineffectual [15] if the region R is in a high-dimensional space. Specifically, methods such as Simpson's and Gauss Quadrature rely on computing $f(\mathbf{x})$ for values of $\mathbf{x}$ sampled regularly on the region R. The number of values that need to be sampled grows exponentially with the dimension of the integral.

[1] A complete derivation is available in [14], amongst other sources.

Let's define N as the number of samples of x we have to draw from R in order to get an acceptable estimate[2] for the integral I. Call the dimension of the space we are integrating in D. The number of values that have to be generated grows exponentially with the dimension of the integral, and thus the time taken[3] to find the solution will be of order $\mathcal{O}(N^D)$. This exponential dependence on the dimension of the problem is often referred to as the *curse of dimensionality*, since solving integrals numerically in high dimensions using such techniques becomes impractical because of this exponential dependence.

Fortunately *Monte Carlo Integration* provides a means to sidestep this problem. The core idea of Monte Carlo integration is that to get a good estimate for I, it is not necessary to sample regular values from the D-dimensional space in which R exists. Instead, if we pick independent and identically distributed random samples, $\mathbf{x}^{(1)}, ..., \mathbf{x}^{(N)}$ from R, then we can obtain the following estimate for I:

$$\widehat{I} = \frac{1}{N}\{f(\mathbf{x}^{(1)}) + \cdots + f(\mathbf{x}^{(N)})\} = \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}^{(i)})$$

The Law of Large Numbers can be used to show [12] that if a large number of samples N are drawn, then $\widehat{I} \to I$. That is,

$$\lim_{N\to\infty} \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}^{(i)}) = I.$$

More surprising though, is the fact that the error in the estimate $\widehat{I}$ is $\mathcal{O}(\frac{1}{\sqrt{N}})$, which is a result of the Central Limit Theorem. This result holds regardless of the dimension of the integral. As we have already mentioned, for other known methods it is necessary to evaluate exponentially more points in R as the dimension grows in order to get an accurate estimate. However, this is not the case for Monte Carlo integration - since the error in the estimate is only reliant on the number N of sample vectors $\mathbf{x}^{(i)}$ drawn from R, the

[2] The specific error threshold that is chosen isn't important, just so long as it is kept constant.

[3] Here $\mathcal{O}$ represents the 'big-O' notation for order analysis.

dimension can be increased arbitrarily without affecting the accuracy of the result or the time complexity of the algorithm.

To give a concrete example of Monte Carlo integration, assume that we wish to calculate the following integral:

$$I = \int_0^1 x^3 dx.$$

This integral has an analytical solution, and is also best solved using non-Monte Carlo numerical integration techniques if we insist on solving it numerically. Nevertheless, it will suffice as an illustration (and for this purpose, it having a well-known solution makes the numerical estimate easier to check).

First we need to draw a number of random values, which we shall label $\mathbf{x}^{(i)}$. Each $\mathbf{x}^{(i)}$ will be drawn from $[0, 1]$. Let's draw 1000 values. We now have $N = 1000$ values in the range of the integration, for example: $\mathbf{x}^{(1)} = 0.4351, \mathbf{x}^{(2)} = 0.1791, ..., \mathbf{x}^{(1000)} = 0.6301$. Now we can compute an estimate of I:

$$\widehat{I} = \frac{1}{1000}\{0.4351^3 + 0.1791^3 + \cdots + 0.6301^3\}$$

Obviously the result you get will depend on the actual random values in $[0, 1]$ that were drawn. We obtained the result $\widehat{I} = 0.2506$ for one run of the algorithm, which is reasonably close to the actual answer $I = 0.25$.

A.1.3 Monte Carlo, Beyond Simple Integration

Solving high dimensional integrals is an important and useful task, and it so happens that a fairly wide variety of interesting scientific problems can be reduced to the computation of an integral. One particularly important method is *Markov chain Monte Carlo* (MCMC) [12]. MCMC may be used when we would like to sample from a probability density function $p(\mathbf{x})$, but

doing this directly is not feasible[4]. The technique pioneered by Metropolis et. al. [16] is rather crafty. Instead of tackling the problem of drawing from $p(\mathbf{x})$ directly, an MCMC algorithm sets up a Markov chain[5] with $p(\mathbf{x})$ as its stationary distribution, and draws samples from this Markov chain.

A host of methods based on MCMC now exist, and have been applied in a wide variety of fields, ranging from computational chemistry to Bayesian inference.

A.2 A Review of Parallel Pseudorandom Number Generation

In the review of Monte Carlo methods, we noted that multidimensional integrals could be more efficiently computed using random sampling than standard numerical techniques. We saw that the defining element of Monte Carlo algorithms is randomness — all Monte Carlo algorithms require random samples to be drawn from a distribution. This leads us to our first problem: computers are deterministic devices, so how can we generate random numbers on them? Next we need to deal with the issue parallel pseudorandom number generation. Finally, we'll take a brief look at assessing the quality of pseudorandom number sequences.

[4] Why would it not be possible to sample from $p(\mathbf{x})$ directly? Clearly if the p.d.f. is not known *a priori* then this is one case where direct sampling is not possible. This situation arrises in molecular modelling, for example, where we may wish to compute the potential energy surface of a molecule, which we know exists but for which we don't yet have a description.

[5] A Markov chain is a discrete-time stochastic process with the condition that future states are independent of past states. The probability of transitioning from one state to another is dependent only one the present state, and not on the previous states in which the process has been.

A.2.1 Generating Random Numbers on Deterministic Computers

> "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin." – John von Neumann (1951).

The problem of generating sequences of random numbers sampled from some distribution wasn't widely investigated until computers became powerful enough to perform useful calculations using Monte Carlo methods. However, as soon as they did, the difficulty of generating suitable sequences was soon realized to be a considerable obstacle. Even today techniques for random number generation on computers is an important research area[6].

Since computers are deterministic devices, it's not possible to generate truly random numbers using a procedure on a computer. Truly random numbers can be obtained from physical processes that are truly random, such as the time between beta particle emissions in radioactive decay[7].

Given that physical processes are too slow or expensive to extract large numbers of random bits from, it has been necessary to devise methods of generating random numbers on computers. However, as we've discussed, we can't generate truly random numbers — instead we have methods to generate numbers *pseudorandom* numbers, which can pass a certain standard for randomness and are thus suitable for use in Monte Carlo simulations.

One of the simplest and most well-known pseudorandom number generators is the *linear congruential generator* (LCG). LCGs, described in detail in [21], are described by the recurrence relation

[6] In 1997, Matsumoto and Nishimura developed the Mersenne twister algorithm [17], which is both faster than other commonly-used methods, and has a large period. It is now widely used, but is still far from perfect.

[7] Radioactive decay is well-covered in the literature, with many textbooks giving it a thorough treatment. See [18], for example. HotBits [19] is an online service that provides a stream of random numbers based on the timing between β decay in $^{85}\mathbf{Kr} \rightarrow {}^{85}\mathbf{Rb} + \beta^- + \gamma$.

$$X_n = (aX_{n-1} + b) \mod m.$$

Here X_n denotes the nth number in the sequence $(X_n)_{n \in \mathbb{N}}$. The nth number is computed using the $(n-1)$th number in the sequence X_{n-1}, and the parameters a, b and m.

LCGs are highly sensitive to the parameters, as well as the initial value X_0, also known as the *seed*. Numerous studies have been done to determine what parameter values yield LCGs that produce usable pseudorandom number sequences. See, for example, [15] and [21]. If the parameters are poorly chosen, then the resulting LCG may produce a sequence that is highly correlated and fails to pass any reasonable test for randomness.

Loosely, the *period* of an LCG is the number of values it will produce before it begins repeating its starting sequence. The period will be at most m, but it may be less, depending on the values of a, b and X_0. To obtain a full period, it is necessary that:

1. b and m be relative prime.
2. $a - 1$ be divisible by all prime factors of m.
3. $a - 1$ be a multiple of 4 if m is a multiple of 4.
4. $m > \max(a, b, X_0)$
5. $a > 0$ and $b > 0$

If we let $Y_n = X_n/m$ then $Y_n \in [0, 1)$. If the LCG parameters are chosen carefully, then we can consider each Y_n as having been randomly sampled from the uniform distribution between 0 and 1, i.e. $Y_n \sim \mathcal{U}(0, 1)$.

Many Monte Carlo algorithms require that we sample not from $\mathcal{U}(0, 1)$, but from some other probability density function $p(\mathbf{x})$. We will review later in the chapter how values drawn from $\mathcal{U}(0, 1)$ can be transformed such that samples are effectively drawn from other distributions.

A.2.2 Parallel Pseudorandom Number Generation

We have seen one method of generating a single sequence of pseudorandom numbers. However, in order to speed up Monte Carlo simulations on parallel computers, it is necessary to provide each logical processor with its own stream of pseudorandom numbers. We will soon see that this is also true for Monte Carlo algorithms on reconfigurable computers.

With the rise of parallel computers, came the need for methods of generating uncorrelated streams of random numbers. Assume we have a parallel computer with p logical processors labelled $P_1, ..., P_i, ..., P_p$. Clearly it is unacceptable to have each processor use the same LCG, with the same parameters. In that case, each processor will use an identical sequence of pseudorandom numbers.

What we need is for each processor P_i to have a sequence of pseudorandom numbers at its disposal that is uncorrelated with the sequences used by any other processor. Several schemes have been devised to accomplish this. We now review some, as they are presented in [22].

Perhaps the most obvious is the *central server* scheme. If we dedicate a single processor to generating a sequence of random numbers, this processor can send the next number in the sequence to the next processor that requests one. The biggest drawback of this scheme is that the central server will likely become a bottleneck.

Cycle division eliminates the requirement of having a central server. Instead, each processor P_i generates a subsequence of the overall sequence $(X_n)_{n \in \mathbb{N}}$. One popular realization of this scheme is simply to have each processor use the same LCG but use different seed values on each processor. Another version, know as the *leap frog* method, has the first processor, P_1, generate the subsequence $(X_1, X_{p+1}, X_{2p+1}, ...)$, the second processor, P_2, generate the subsequence $(X_2, X_{p+2}, X_{2p+2}, ...)$, and so on. Yet another method, *cycle splitting*, allocates the first N numbers in the sequence $(X_n)_{n \in \mathbb{N}}$ to the processor P_1, the second N numbers in the sequence to processor P_2, and so on.

Unfortunately the subsequences generated by cycle division on an overall sequence do not necessarily have the same statistical properties as the overall sequence. Another disadvantage is that the period of each subsequence is necessarily a factor of p smaller than the period of the overall sequence.

The *cycle parameterization* scheme suggests modifying one or more of the parameters of the generator on each processor, so that each processor will generate a different sequence of numbers.

In SPRNG[8], the *Scalable Library for Pseudorandom Number Generator*, Mascagni et. al. [23] provide versions of the linear congruential generator $X_n = (aX_{n-1} + b) \mod m$ that are parameterized by the multiplicative constant a and the additive constant b.

If m is prime, then a different value for a can be used by each processor, so that each LCG generates a different sequence. The permissible values of a are found by applying the rules for an LCG given in the previous section.

If m is a power-of-two, the additive constant b can be parameterized. This method was pioneered by Percus et. al. [24] for generating multiple sequences of random numbers in parallel for the NYU Ultracomputer. In this method, a set of additive constants $\left\{b^{(i)}\right\}$ is chosen, where the elements $b^{(i)}$ are pairwise relatively prime. In SPRNG, Percus et. al.'s suggestion to let $b^{(i)}$ be the ith prime is used.

Thus the additive constant parameterization of the linear congruential generator results in the ith processor, P_i, using the following generator:

$$X_n^{(i)} = \left(aX_{n-1}^{(i)} + b^{(i)}\right) \mod m.$$

SPRNG provides parallelizations of various other pseudorandom number generators, but it does this precisely because there is no consensus on which generator, and which parallelization, if any, is universally the best. The additive constant parameterized version of the linear congruential generator is representative of the other parallel generators available in SPRNG, and

[8]SPRNG is the *de facto* standard library for random number generation on cluster computers.

suffices as an example for this chapter.

A.2.3 Assessing the Quality of Pseudorandom Number Sequences

> "Random number generators should not be chosen at random."
> – Donald Knuth (1986).

Because of the importance of ensuring that a pseudorandom number generator is producing 'good' results, a host of tests have been devised to test generators. Using an unsuitable generator in Monte Carlo simulations can render its results meaningless, and moreover, without verifying the 'correctness' of the generator, it may be very difficult to detect errors.

Ideally the numbers a generator produces should be independent and identically distributed in the range of the generator. Clearly a sequence such as $(1, 1, 1, 1, 1, 1, ...)$ isn't 'sufficiently random' to satisfy the needs of a Monte Carlo algorithm. Neither is $(1, 0, 1, 0, 1, 0, ...)$. However, in general it is necessary to use a computer test to determine if a sequence is somehow biased; it is impossible to do so merely by inspection.

A variety of tests have been developed to test sequences for randomness. Knuth's standard tests [21] are simple, but still remain popular. The Diehard tests by Marsaglia [25] are considered to be one of the more stringent sets of tests available. We now briefly review several popular methods for assessing the randomness of sequences.

Entropy measures

Entropy is the measure of the information contained in a signal. In a truly random sequence, of sufficient length, an entropy calculation should reveal that the sequence has an entropy of 8 bits per byte. That is, before we read a byte, we have absolutely no information about it. Entropy is defined as:

$$H(X) = \sum_{i=1}^{n} p(x_i) \log_2 \left(\frac{1}{p(x_i)} \right) = - \sum_{i=1}^{n} p(x_i) \log_2 p(x_i).$$

Here X is a discrete random variable with states $x_1, x_2, ..., x_n$. $p(x_i) = \Pr(X = x_i)$ is the probability of being in the ith state, x_i.

Thus if we have a sequence $Y_1, Y_2, ..., Y_n$ of bytes (i.e. $Y_i \in [0, 255], Y_i \in \mathbb{N}$), then we can compute its entropy as follows. Therefore, in this example, the states x_i are the integers between 0 and 255, so $x_i = i - 1$. First, we need to find the probabilities of Y_i being in the states x_i. This can be done by binning the data. For example, $p(x_1) = p(0)$ can be calculated by counting the number of Y_i equal to $x_1 = 0$, and dividing this count by n.

With the probabilities $p(x_1), p(x_2), ..., p(x_{256})$ computed, the entropy can be computed using the definition:

$$H(X) = - \sum_{i=1}^{256} p(x_i) \log_2 p(x_i).$$

The expected entropy of a random sequence is 8 bits per byte, so this crude test can be used to invalidate sequences with sufficiently low entropies.

The χ^2 test for goodness-of-fit

The χ^2 goodness-of-fit test is widely used in statistics[9] to determine if observed data fits an expected distribution. Knuth [21] suggests that the χ^2 test is also a suitable test for measuring the randomness of a sequence. In a truly random sequence, the probability of any given value appearing at some position in the sequence should be equal to the probability of any other value appearing at that same position. We can use the χ^2 goodness-of-fit test to determine whether the values in a given sequence can reasonably be considered to be following a uniform distribution.

[9] For example, in survey statistics, we may wish to know if one property we are gathering data on is affected by another, and then, if so, how the one property influences the other. The χ^2 test is used to determine whether or not the two properties are independent before investigating further.

In order to perform a χ^2 test on a sequence $Y_1, Y_2, ..., Y_n$ of bytes, we must compute the χ^2 test statistic:

$$\chi_s^2 = \sum_{i=1}^{k} \frac{(m_i - Np_i)^2}{Np_i}$$

The test statistic is a measure of the deviation of a sample from an expected distribution. p_i are the expected probabilities of state x_i appearing, and m_i are the observed frequences. N is the sample size.

For the case of the sequence $Y_1, Y_2, ..., Y_n$ of bytes, as with the entropy test, we first need to bin the data and determine the frequency with which each state $x_i = i - 1$ appeared, for all $k = 256$ possible states. Thus to compute the frequency m_1, simply count the number of Y_i equal to $x_1 = 0$, and so on for $m_2, ..., m_k$. We note that $p_i = 1/256$ for all $i \in \{1, 2, ..., 256\}$. The test statistic can then be computed as:

$$\chi_s^2 = \sum_{i=1}^{256} \frac{(m_i - n/256)^2}{n/256}$$

If the null hypothesis (that each state x_i is equally likely to appear) is true, then χ_s^2 will be drawn from a χ^2 distribution with degrees of freedom $d = k - 1 = 255$.

To test if the null hypothesis is true, we need to determine how likely it is for the χ_s^2 value to have been drawn from the χ^2 distribution. If this is unlikely, then it is unlikely that the null hypothesis is true, and thus it is unlikely that the tested sequence is random.

The probability that a calculated χ_s^2 value, with d degrees of freedom, is due to chance is:

$$Q_{\chi_s^2,d} = \left[2^{d/2}\Gamma\left(\frac{d}{2}\right)\right]^{-1} \int_{\chi_s^2}^{\infty} \left(\chi^2\right)^{\frac{d}{2}-1} e^{-\chi^2/2} \, d\left(\chi^2\right)$$

Here $\Gamma(x) = \int_0^\infty t^{x-1}e^{-1}dt$. We can then use this probability to decide whether we should reject the null hypothesis or not.

Serial Correlation

In a random sequence, the number at a certain position should be uncorrelated with the number at the previous position. Statistical methods have been developed to measure correlation between successive values in time series, and these techniques are well-suited to determining if values in a random sequence are indeed uncorrelated.

For a sequence $Y_1, Y_2, ..., Y_n$ of bytes, a serial correlation coefficient r may be computed as follows:

$$r = \frac{n \sum_{i=2}^{n} Y_{i-1} Y_i - \left(\sum_{i=2}^{n} Y_i\right)^2}{n \sum_{i=2}^{n} \left(Y_i\right)^2 - \left(\sum_{i=2}^{n} Y_i\right)^2}$$

A value of $r = 1$ indicates that the tested sequence is completely predictable. You can easily see that the sequence $(1, 1, 1, 1, 1, 1, ...)$ will result in a coefficient of 1. A value of $r = 0$ indicates that the sequence exhibits no serial correlation, and a random sequence can be expected to yield a r value near 0.[10] A 'typical' non-random sequence, such as encoded text from English writing, may yield a coefficient near 0.5.

[10] Note that r can be positive or negative.

Appendix B

The SRC-6 Reconfigurable Computer

Figure B.1 shows the SRC-6 MAPstation used during the duration of this thesis project. Figure B.2 shows the architecture of the MAP module, which contains onboard memory and two FPGAs. The MAP module is connected to a switch, which also has a traditional microprocessor-based computer (with a Xeon 2.8GHz CPU and 1GB RAM) attached. This computer initiates computations on the MAP.

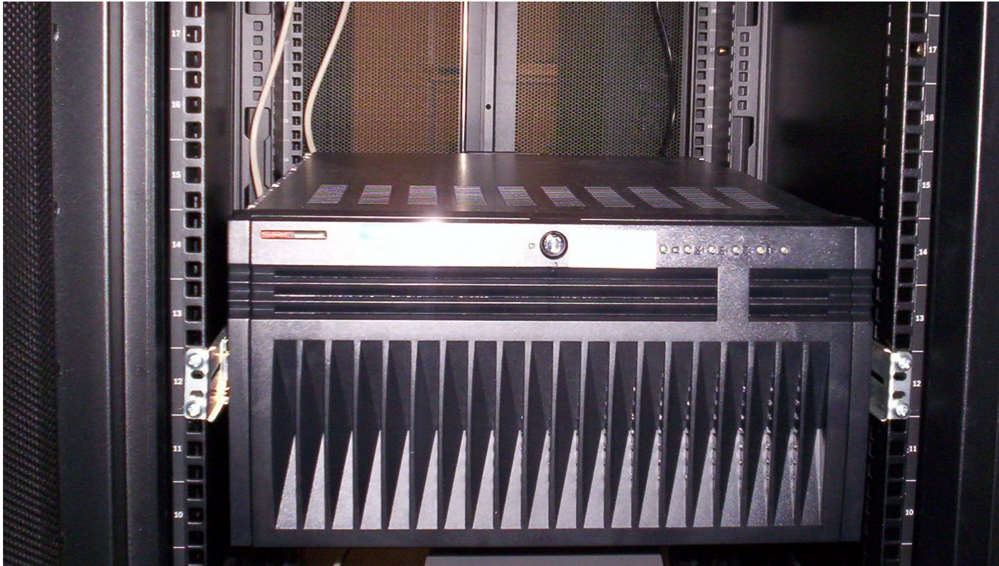

Figure B.1: The SRC-6 in a rack at the National Center for Supercomputing Applications.

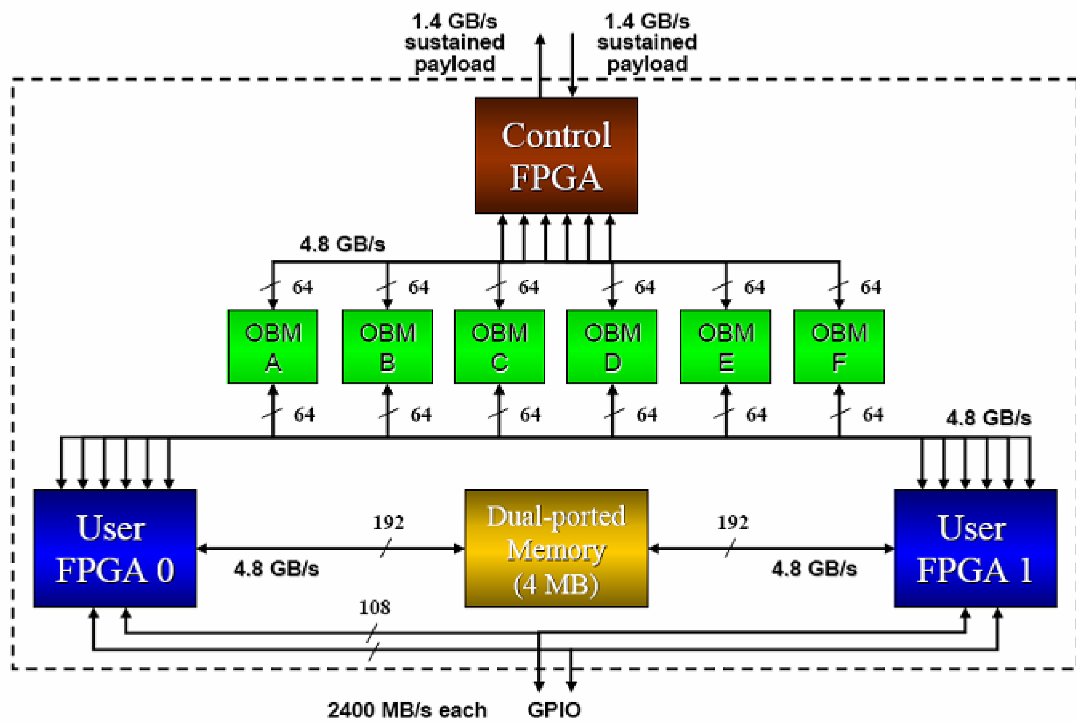

Figure B.2: The architecture of the SRC MAP module.

Bibliography

[1] K. Compton and S. Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Comput. Surv.*, **34**, 2, 171–210 (2002).

[2] G. Estrin, B. Bussel, T. Turn and J. Bibb. Parallel processing in a reconstructable computer system. *IEEE Trans. Elect. Comput.*, 747–755 (1963).

[3] C. Chang, K. Kuusilinna, B. Richards and R. Brodersen. Implementation of BEE: a real-time large-scale hardware emulation engine. In *Proc. Eleventh ACM Intl. Symp. on FPGAs*, 91–99 (2003).

[4] M. Gokhale, et. al. SPLASH: A Reconfigurable Linear Logic Array. In *Proc. Intl. Conf. Parall. Proces.*, 526–532 (1990).

[5] J. Arnold, D. Buell and E. Davis. Splash 2. In *Proc. Fourth Annual ACM Symp. on Parall. Algor. Arch.*, 316–322 (1992).

[6] C. Chang, J. Wawrzynek and R. Brodersen. BEE2: A high-end reconfigurable computing system. *IEEE Design and Test of Comput.*, **22**, 2, 114–125 (2005).

[7] R. Mykland. *Ascenium: A Continuously Configurable Architecture.* Ascenium White Paper. April 2006.

[8] M. Gokhale and B. Schott. Data-parallel C on a reconfigurable logic array. *J. Supercomput.*, **9**, 3, 291–313 (1995).

[9] V. Kindratenko, D. Pointer, D. Raila and C. Steffen. Comparing CPU and FPGA Application Performance. *ISL White Paper, National Center for Supercomputing Applications* (2006).

[10] S. Wolfram. *A New Kind of Science.* Champaign, IL: Wolfram Media (2002).

[11] N. Metropolis and S. Ulam. The Monte-Carlo method. *J. Amer. Stat. Assoc.*, **44**, 335–341 (1949).

[12] J. Liu. *Monte Carlo Strategies in Scientific Computing.* New York, NY: Springer-Verlag (2002).

[13] H. Dorrie. *100 Great Problems of Elementary Mathematics.* Mineola, NY: Dover Publications (1965).

[14] D. Bertsekas and J. Tsitsiklis. *Introduction to Probability.* Nashua, NH: Athena Scientific (2002).

[15] W. Press, S. Teukolsky, W. Vetterling and B. Flannery. *Numerical Recipes in C.* Cambridge, England: Cambridge University Press (1992).

[16] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of State Calculations by Very Fast Computing Machines. *J. Chem. Phys*, **21**, 1087 (1953).

[17] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. Model. Comput. Simul.* **8**, 3 (1998).

[18] B. Martin. *Nuclear and Particle Physics.* West Sussex, England: John Wiley & Sons (2006).

[19] http://www.fourmilab.ch/hotbits/. J. Walker.

[20] http://www.fourmilab.ch/random/. J. Walker.

[21] D. Knuth. *The Art of Computer Programming*, Volume 2: *Seminumerical Algorithms.* Boston, MA: Addison-Wesley (1997).

[22] A. Srinivasan, D. Ceperley and M. Mascagni. Random Number Generators for Parallel Applications. In *Monte Carlo Methods in Chemical Physics*, *Advances in Chemical Physics* **105**. New York, NY: John Wiley & Sons (1998).

[23] M. Mascagni, D. Ceperley and A. Srinivasan. SPRNG: A Scalable Library for Pseudorandom Number Generation. *ACM Trans. Math. Soft.*, **26** (2000).

[24] O. Percus and M. Kalos. Random Number Generators for MIMD Parallel Processors. *J. Par. Distr. Comput.*, **6**, 477–497 (1989).

[25] http://www.stat.fsu.edu/pub/diehard/. G. Marsaglia.

[26] J. Hull. *Options, Futures and Other Derivatives.* London, England: Prentice Hall (1997).

[27] M. Fu, D. Madan and T. Wang. Pricing Asian Options: A Comparison of Analytical and Monte Carlo Methods. *J. Comput. Fin.*, **2**, 49–74 (1999).

[28] M. Haugh. *Monte Carlo Simulation: IEOR E4703* course notes. Columbia University (2004).

[29] P. Jäckel. *Monte Carlo Methods in Finance.* West Sussex, England: John Wiley & Sons (2002).

[30] P. Glasserman. *Monte Carlo Methods in Financial Engineering.* New York, NY: Springer-Verlag (2003).

[31] J. von Neumann. *Theory of Self-Reproducing Automata.* Urbana, IL: University of Illinois Press (1966). (Edited and completed by A. Burks).

[32] W. van Dam. *Quantum Cellular Automata.* Master's Thesis, Computing Science Institute, University of Nijmegen (1996).

[33] M. Gardner. *The Game of Life* in *Wheels, Life and other Mathematical Amusements.* New York, NY: W.H. Freeman (1983).

[34] G. Perrin and A. Darte (Editors). *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications.* Lecture Notes in Computer Science **1132**. Berlin, Germany: Springer-Verlag (1996).

[35] S. Wolfram. Cellular Automaton Fluids I: Basic Theory. *J. Stat. Phys.*, **45**, 3/4, 471–526 (1986).

[36] U. Frisch, B. Hasslacher and Y. Pomeau. Lattice Gas Automata for the Navier-Stokes Equation. *Phys. Rev. Lett.*, **56**, 14, 1505–1508 (1986).

[37] L-S. Luo. The Lattice-Gas and Lattice Boltzmann Methods: Past, Present and Future. In *Proc. Intl. Conf. on Applied Comput. Fluid Dyn. (ACFD '00)*, Beijing, China (2000).

[38] M. Grand. *Patterns in Java, Volume 1.* Indianapolis, IN: Wiley (2002).

[39] N. Ratha, A. Jain and D. Rover Convolution on Splash 2. In *Proc. of IEEE Symposium on FPGA's for Custom Computing Machines (FCCM '95).* p. 204 (1995).

[40] C. Torres-Huitzil and M. Arias-Estrada. An FPGA Architecture for High Speed Edge and Corner Detection. In *Proc. of Fifth IEEE International Workshop on Computer Architectures for Machine Perception (CAMP'00).* p. 112 (2000).

[41] I. Sobel. *Camera models and machine perception.* Ph.D. Thesis, Stanford University (1970).

[42] J. Canny. A Computation Approach to Edge Detection. *IEEE Trans. Pattern Anal. and Mach. Intel.*, **8**, 6, 679–698 (1986).

[43] J. Prewitt. Object enhancement and extraction. In *Picture Processing and Psychopictories*. New York, NY: Academic Press (1970).

[44] W. Wriggers, R. Milligan and J. McCammon. Situs: A package for docking crystal structures into low-resolution maps from electron microscopy. *J. Struct. Bio.*, **125**, 185–195 (1999).

[45] A. Roseman. Docking structures of domains into maps from cryo-electron microscopy using local correlation. *Acta Cryst.*, D**56**, 1332–1340 (2000).

[46] W. Wriggers and P. Chacón. Modeling tricks and fitting techniques for multiresolution structures. *Structure*, **9**, 779–788 (2001).

[47] A. Snowden. *Automatic Docking of Molecular Structures into Low-Resolution Electron Microscopy Graphs.* B.Sc. (Hons.) Thesis, Department of Computer Science, University of Cape Town (2005).