High performance reconfigurable computing with cellular automata

Murtaza, S.

Publication date
2010
Document Version
Final published version

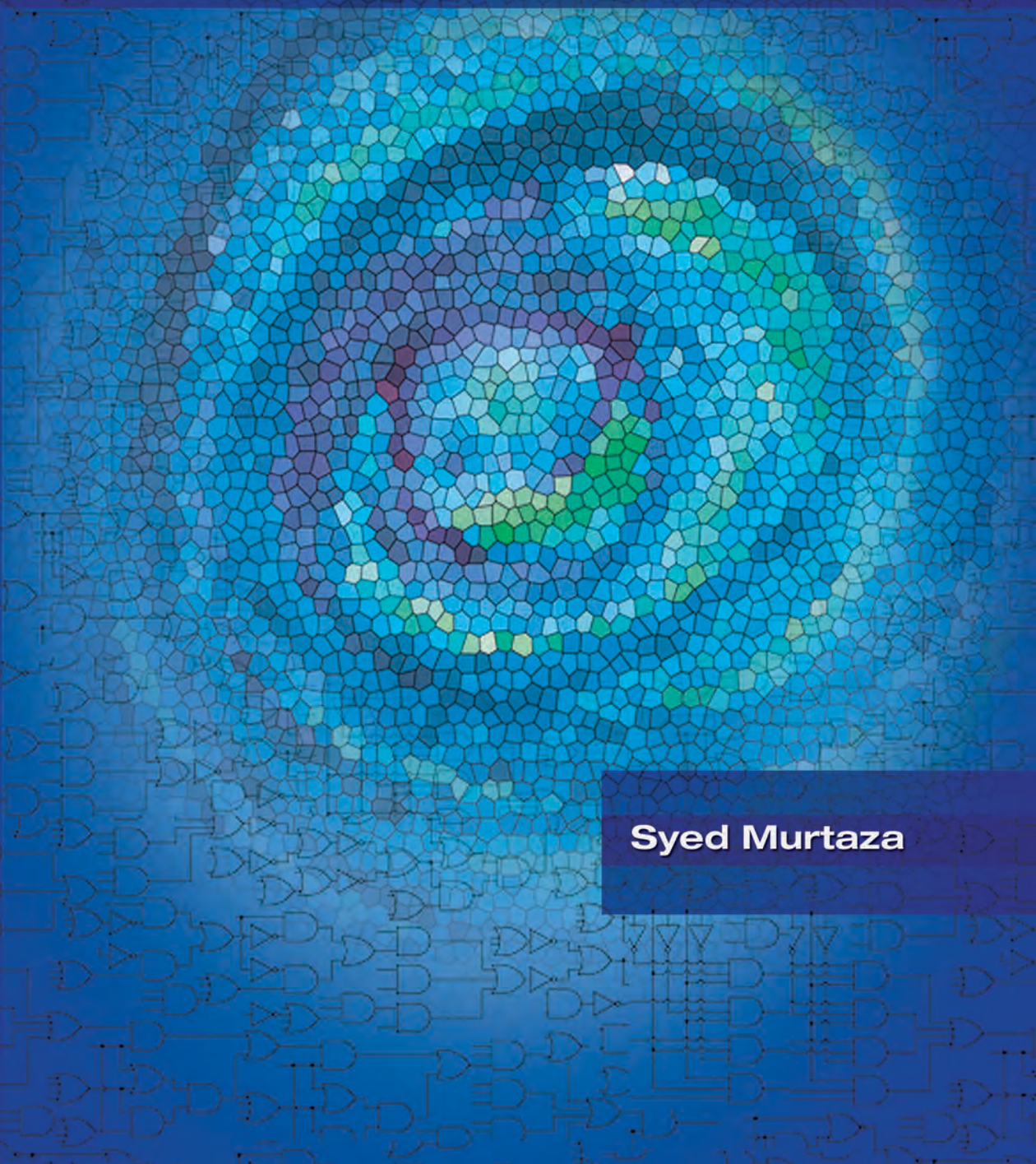
lattice
Boltzmann
Cellular Automata
MPI
Parallel programming
VHDL
C++
Fortran
Java
make
Maxwell- 64 FPGA Supercomputer
von Neumann
multicore
high performance
reconfigurable
computing
latex
compute bound
performance modeling
Moore's law
three walls
Exascale
3D LBM
system design
scalable system
simulations
custom hardware design
hardwired
collision
propagation
latency hiding

cluster
host
mapping
accelerator
cells
complex systems
dynamical systems
space, time and state are discrete
finite state machine
deterministic
parallel processing
boundary conditions
Game of Life
floating point
streaming
HPP model
clock frequency
logic blocks
block RAMs
fluid dynamics
algorithm
efficiency
fractional overheads
PE
pipeline
optimal design
speedup
system parameters
execution time
bulk processing
kernel cells

Syed Murtaza

High Performance Reconfigurable Computing with Cellular Automata

High Performance Reconfigurable Computing with Cellular Automata



Syed Murtaza

High Performance Reconfigurable Computing with Cellular Automata

High Performance Reconfigurable Computing with Cellular Automata

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C van den Boom
ten overstaan van een door het college voor promoties
ingestelde commissie, in het openbaar te verdedigen
in de Agnietenkapel
op dinsdag 25 mei 2010, te 10:00 uur

door

Syed Murtaza
geboren te Kashmir, India

Promotor:	prof. dr. P.M.A. Sloot
Co-promotor:	dr. A.G. Hoekstra
Overige leden:	prof. dr. M.T. Bubak
	prof. dr. B. Chopard
	prof. dr. C.R. Jesshope
	prof. dr. M. Leeser
	prof. dr. R.J. Meijer
Faculteit:	Faculteit der Natuurwetenschappen
	Wiskunde en Informatica

The research reported in this thesis has been carried out at the Section Computational Science, University of Amsterdam, with financial support of EU IST *QosCosGrid* Project (FP6-IST-2005-033833).
Advanced experiments of the work were carried out using Maxwell – a 64-FPGA Supercomputer, at EPCC, University of Edinburgh. The author also worked at EPCC on a one month fully funded *visiting academic programme* through the *FPGA High Performance Computing Alliance*.

Author contact: s.murtaza@uva.nl and syedmurtaza@msn.com

Cover design by Qaiser Azim (darkgreenstudios.com)
Printed by Printforce (printforce.nl)

Contents

Preface

Al-hamdu Lillahi Rabbil 'Alamin, finally the day has arrived when I can talk about my proud PhD journey, a journey that started in the early months of 1992. Then 16, the aim was simply to keep myself safe from the on-going turmoil in Kashmir, India. Leaving my parents, friends, and hometown behind, I spent the following decade in Delhi - the control center for billion plus people. The new city embraced me and was kind enough to equip me with the right mix of skills to face the real world. I thoroughly enjoyed my Delhi days where I completed the last two years of my senior school and four years of engineering from Jamia Milia Islamia, worked for two years, made some good friends, and met my to-be life partner.

During my fresher jobs in Delhi, my interest in computing dawned upon me, and I decided to pursue a masters programme in information technology from the University of Nottingham, UK. Upon graduation, I was eyeing to work for a fortune 500 company. However, policies applicable to non-EU nationals meant– *choose later start now*, get going with the professional world right away. With an opportunity to work as a lecturer at the University of Akureyri in Iceland, I headed north in early 2003. Soon my interest in academia and research became apparent and I was collaborating with Prof. Peter Sloot's research group here at the University of Amsterdam. In August 2006, I moved to Amsterdam and finally after two years of marriage, in December 2006, my wife and I started living together. This was the first time since the start of my journey back in 90's that I finally felt at home. Yes, this has been quite a journey.

Through this journey, many people have played an important role. I would like to thank Mr. A.U. Tak, then in 2001 the District Manager for the State Bank of India, for helping me get a higher education loan for my masters in the UK. Special thanks to Prof. Mark O'Brien, then in 2003 Dean, Faculty of Computer Science at the University of Akureyri, for introducing me to academia, and a mentor during my next four years as a University Lecturer. Thanks to other colleagues at the University of Akureyri for their encouragement and help, especially Dr. Thorsteinn Gunnarsson, then rector, for all the funding and support.

Heartfelt thanks to my promoter, Prof. Peter Sloot and my co-promoter, Dr. Alfons Hoekstra for the opportunity that has changed the course of my life. It has been a privilege working with you. Alfons, you have been a great supervisor and a source of constant support and encouragement.

Special thanks to my thesis committee members, Prof. Marian Bubak, Prof. Bastien Chopard, Prof. Chris Jesshope, Prof. Miriam Leeser and Prof. Robert Meijer for their valuable suggestions and feedback.

Thank you to all my SCS colleagues. It was a pleasure sharing office space with Eric, Gokhan, Qiu, and Hannan. Thank you guys for all our interesting discussions to keep life fun. Special thanks to Eric for sharing his Fortran LBM simulation code. Thanks Bui for the relaxing ping-pong sessions, Breanndan for the stimulating tech talks and Edwin for our interesting FPGA related talks. Special thanks goes to our secretariat office, especially Erik Hitipeuw for all the vital guidance and help. It was such a pleasure working with the wonderfully international mix of people at SCS, awesome, thank you SCS!

Heartiest thanks to Dr. Andy Pimentel for connecting me to the world of FPGAs and the researchers at the Leiden Embedded Research Center. During my PhD I also had the privilege of visiting EPCC at the University of Edinburgh, for one-month fully funded *visiting academic programme* through the *FPGA High Performance Computing Alliance (FHPCA)*. Thanks to Dr. Andrew McCormick and Graham Smart from Alpha-data for introducing me to Dr. Rob Baxter at EPCC. Rob's invitation to EPCC's FHPCA in February 2008 was a brilliant experience. Throughout my FPGA related work, Andrew was a huge support. Cannot imagine working with FPGAs without his expertise and kind help.

I would like to thank all my friends for being there for me. Last but not the least, I would like to thank my family for all their support, love and encouragement. Sincere thanks to my parents for their life long struggle to see my siblings and me do well in life. Thank you to my Didi and kid brother, Mujtaba for their constant guidance and love, and to my little niece, Mariam for her antics and amusing talks that have very often brightened my days. Thanks to my in-laws and extended family members for all the encouragement and duas. Finally a big thanks to my wife, Asra, for standing by me and her constant love and support through this journey, and of course, for proof reading this thesis.

Syed Murtaza

Chapter 1

Introduction

The quest to know everything about everything embarks us upon an infinite journey, one, very often dictated by the technological developments of time. With human evolution, our abilities to push the limits of the unknown have also developed, and more recently been fueled by our capability to compute increasingly faster. It may not be incorrect to say that our ability to know more now heavily relies upon innovation in computing.

1.1 The Road to Parallel Computing

Computers, traditionally based on single thread sequential computation model, also called the von Neumann architecture [114], have evolved from the 50's vacuum tubes based technology [110], to the current state of the art CMOS based nanoscale devices. Since the breakthrough in semiconductor devices, the processor technology has ridden the wave called Moore's law. More recently this wave has hit three walls – memory, power and instruction level parallelism – and has thus defined the limits of the sequential computing paradigm. Simultaneously, the boom in internet and its applications, accompanied by the need to understand and solve problems of global impact, demand unprecedented computing infrastructure and resources. To meet these challenges, the industry has been compelled to improve computing through massively parallel computing paradigms.

1.1.1 Cellular Automata and Parallel Computing

Cellular automata (CA), an inherently parallel computing paradigm, also proposed by von Neumann in 1948 [59], has been successfully applied to a range of computational problems to model the behaviour of complex systems from nature. John Conway's Game of Life [109], Cellular Automata Machines by Toffoli and Margolus [102], and the more recent book by Stephen Wolfram, A New Kind of Science [116], are some of the works synonymous to CA. It is not surprising to find why some of the computing pioneers are

fascinated and intrigued by the capabilities of a set of simple cells, arranged on a regular grid, with local connectivity reproducing complex structures from nature like fluid flow.

Long before Moore's law hit the three walls [5], parallel computing paradigms in general have been employed in high performance computing (HPC) [5, 111]. In HPC, the improvement of computations both in terms of speed and accuracy are of prime importance. In general, one of the ways to improve computations is to understand the process of mapping an application to the available hardware. More recently, reconfigurable devices like Field Programmable Gate Arrays (FPGA) have been integrated into HPC systems for application acceleration. Such systems provide grounds where the application can benefit from exploiting the possibilities of using both fine-grain and coarse-grain levels of computations.

The recent emergence of multicore architectures and the shift towards multicore computing, may have triggered the evolution of von Neumann architecture towards a parallel processing paradigm. Today some even consider the possibilities of migrating to different computing approaches like, CA [120]. This more recent switch from a single threaded sequential paradigm to a massively parallel computing paradigm [5, 53] makes CA and its related work more relevant through the breadth of computing. Based on advances in the CMOS technology, chip companies are foreseeing to manufacture hundreds of cores on a single die [53]. However, simply increasing the number of cores is not of much use if we are unable to scale applications to the available cores [58]. So it is no surprise that computing, in general, is seeking solutions that go beyond conventional paradigms based on von Neumann architecture [1, 5]. Based on this framework we attempt to understand, *what can be achieved by implementing massively parallel computing paradigms like CA, using available reconfigurable devices like FPGAs?*

1.2 Research Motivation

The three building blocks of this work are, HPC in general, stencil based computations like CA in particular, and reconfigurable devices like FPGA. Based on this criteria, the challenges are a) to demonstrate performance gain with running HPC applications like fluid simulations, b) to understand and classify CA based computations to exploit computing resources efficiently, and c) to tailor hardware resources like FPGA as per the application requirements. In other words dissecting the system implementation right through the top-level representation to the low-level implementation details to analyse and understand the overall system performance. Simultaneously, the recent shift towards multicore computing demand such dissections to understand and explore the full potential of multicore

computing. This need to look for new computing approaches to meet the demands of future applications led us to research the following questions:

- What can be achieved by implementing massively parallel computing paradigms like CA, using available reconfigurable devices like FPGAs?
- How to map an application from an abstract level to the underlying bit-level implementations in hardware?
- Is hardware reconfigurability a requirement to harness the benefits of the multi-billion transistor era and the overall system speedup?
- Is CA the potential candidate among the parallel processing alternatives in future?

Addressing our research aims requires an understanding of: how to map CA based high performance applications to reconfigurable device enabled systems. This exercise includes a) formulating a model to predict the behaviour and the performance of the overall system design without going into details of the implementation technology, b) based on the computational structure of the application, exploiting and customising the available hardware both at fine-grained and coarse-grained levels for overall performance gain, c) designing a system that can be scaled from a single to a multiple FPGA based implementations and therefore a capability to simulate large simulation sizes, and d) a custom hardware implementation for a CA algorithm using FPGAs. Based on this scientific endeavour we logically formulate the following objectives:

- Formulating a model to predict the behaviour and the performance of the overall CA system implementation using special purpose hardware.
- Develop a detailed system implementation ranging from a single to a multiple FPGA based system to verify our model.
- Analyse our experimental work within the framework of manycore literature review and draw parallels between the two.

1.3 Thesis Roadmap

Chapter two focuses on the research background, presenting concepts and technologies that form the basis of this work. It starts with an overview of CA with some example algorithms. LBM viewed as a generalised CA is also presented. The chapter moves on to the technological side introducing reconfigurable computing and FPGA in particular. It finally concludes with a section devoted to HPC using FPGAs and related works.

Chapter three introduces performance modeling for FPGA-based CA implementation. It presents a simple and a scalable CA computation method along with the performance model. Based on this model, CA algorithms are categorised as compute or IO-bound

computations. Following this categorisation, various CA algorithms are implemented and presented in the following chapters.

Chapter four presents IO-bound computations, like the Game of life and HPP model, and their implementations. It also discusses in detail, the computation strategy based upon pipelining with internal buffers and the optimisation of model parameters for the efficient mapping to application algorithms on to the hardware logic. The chapter concludes with test case results and conclusions.

In chapter five, the thesis enters its core with focus on high performance floating point based computations like LBM. Categorised as compute bound computations, a single FPGA-based implementation along with details of the architecture is presented. The internals of the processing elements implementing the CA collision function and the propagation mechanism using external memory banks are also presented. This single FPGA-based model sets the stage for the following chapters where it is scaled to multiple FPGA-enabled systems. The chapter concludes with test case results and the speedup achieved in comparison to a pure software implementation.

One of the main focuses of this project has been the scalability, that is, to design a system that can be scaled from a single to a multiple FPGA-enabled PC implementation and therefore, a capability to simulate large simulation sizes. Chapter six moves on from single to a multiple FPGA-enabled PC implementation and explains the required modifications brought about in the software and hardware. Application domain decomposition, boundary exchange across multiple FPGAs for every iteration computation, and latency hiding comes into the scene. How efficiently application implementation is able to perform boundary exchange and exploit latency hiding techniques is discussed along with its limits. In conclusion, limitations of a multiple FPGA-enabled PC and results based on the dual FPGA-enabled PC implementation are presented.

Chapter seven presents an FPGA-based LBM implementation scaled to a highly parallel FPGA setup. This chapter demonstrates scalability as one of the main strengths of the mapping strategy. A single FPGA-based implementation is scaled to a 64-FPGA supercomputer with a capability of computing lattice sizes approaching quarter of a billion cells (214.74 million). The chapter concludes with results.

With all the technical details and the implementation models presented in the previous chapters, chapter eight adopts a pencil and a paper approach to predict the performance of desirable system sizes and configurations. The chapter presents how the three dimensional version of LBM would perform based on the multi FPGA-enabled cluster implementation. Based on the implementation demonstrated in previous chapters and the prediction of three dimensional model, the chapter draws some conclusions on using special purpose devices like FPGA for acceleration in HPC.

Mapping an inherently parallel application like CA to a multiple FPGA-enabled system with each FPGA implementing multiple independent processing elements, demonstrates a massively parallel implementation both in hardware and software. Such example implementations [10, 11] have been considered analogous to the manycore system implementation of the future applications. Chapter nine reviews the current state of manycore computing paradigm and its evolution in the future. Based on the current technological push towards manycore paradigm and our experience of implementing multiple FPGA-enabled parallel system, we draw some conclusions about the expectations and the requirements of mapping a CA application to the manycore chips of the future. The chapter concludes with some of the major challenges facing the computing industry and the future impacts.

Chapter 2

Background

This chapter presents the necessary background for the various topics that form the basis of this work. Since one of the main focus of this research has been the implementation of accelerators for cellular automata, an introduction in general is presented for such algorithms followed by the introduction of the various cellular automata models that were implemented as a part of this work. Another dimension has been the design and implementation of cellular automata accelerators using reconfigurable devices like FPGAs, a section is devoted as an introduction to such devices. A brief discussion is presented on the recent emerging trend using reconfigurable device enabled high performance computers also known as high performance reconfigurable computing (HPRC). The chapter concludes with a discussion on the related work in the field of FPGA based CA computations and HPRC in general.

2.1 Cellular Automata

Cellular Automata are decentralised spatially extended systems consisting of large numbers of simple and identical components with local connectivity [92]. Such systems have the potential to perform complex computations with a high degree of efficiency and robustness, as well as to model the behaviour of complex systems from nature. CAs have been studied extensively in natural sciences, mathematics, and computer science. They have been considered as mathematical objects about which formal properties can be proven and have been used as parallel computing devices, both for high-speed simulation of scientific models and for computational tasks such as image processing. CAs have also been used as abstract models for studying emergent cooperative or collective behaviour in complex systems [96]. In addition CAs have been successfully applied to the simulation of a large variety of dynamical systems such as biological processes including pattern formation, earthquakes, urban growth and most notably in studying fluid dynamics. Their implicit

spatial locality allows for very efficient high performance implementations and incorporation into advanced programming environments. For a selection of the numerous papers in all of these areas, see, for instance, [30, 40, 51, 59, 93, 94, 95].

2.1.1 What are Cellular Automata?

CA are dynamical systems where space, time and state are discrete. And in general are characterised as a regular grid of simple finite state machines (often called cells in CA theory) of the same kind with communication between the machines limited to local interactions. Each individual cell changes its state over time depending on the state of the cells in its local neighbourhood. The next state rules are deterministic and the overall structure can be viewed as a parallel processing device with cells updated concurrently with each time step.

Formal definition

Formally, a CA is specified by a quadruplet $(\mathfrak{L}, S, \mathcal{N}, f)$ where:

- $\mathfrak{L}$ represents the lattice.
- S is the finite set of possible states of each of the cell.
- $\mathcal{N}$ is the finite set of neighbourhood.
- f is the local transition rule.

The lattice is a discrete regular grid of cells within a finite dimensional space ($1, 2, 3$ used in practice) where each cell is defined by its discrete position and its discrete state. Each cell can be in one of the finite number of k possible states. Since the time evolution of the CA is also discrete, S_i the state of the i^{th} cell at a new time $t+1$ is a function of the present state of a finite number of cells in its neighbourhood at time t as:

$$S_i(t+1) = f_i\left(S_j(t)\right) \tag{2.1}$$

where $i \in \mathfrak{L}$ and j is the sub lattice neighbourhood about i^{th} cell.

Types of lattice, neighbourhood, and boundary condition

The binary state, nearest neighbour, one dimensional cellular automata is the simplest cellular automata. And a system composed of N uniform cells would look like an array of ones and zeros of width N where the neighbourhood of a cell are the local cells on the either side. The state of all cells form the global configuration of the CA. For two dimensional CAs, various types of lattices are possible, for example, triangular, hexagonal or rectangular. The common types of neighbourhood are Von Neumann (5 cells, consisting

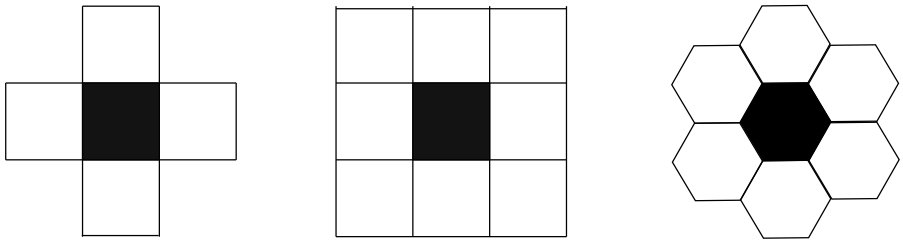


Figure 2.1: *Neighbourhood configurations: (left to right) Von Neumann, Moore and Hexagonal.*

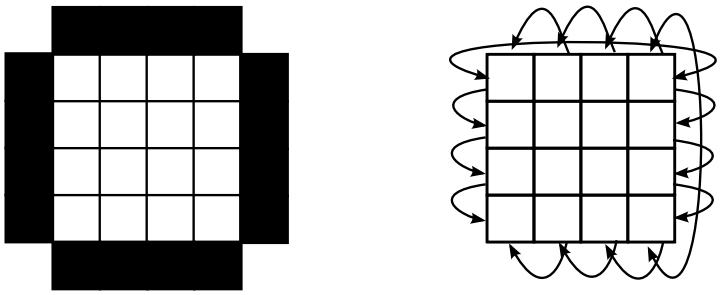


Figure 2.2: *Boundary configurations: (left to right) Fixed and Periodic.*

of the cell along with its 4 neighbours), Moore (9 cells, cell along with its 8 neighbours) and Hexagonal (7 cells, cell along with its 6 neighbours) as shown in Figure 2.1. Various boundary conditions are also possible as shown in Figure 2.2. Fixed boundaries: The boundary cells are assumed to have a *specified* fixed value, for example, in a null boundary configuration the boundary cells are assumed to have a value 0. Periodic boundaries: One in which the lattice is considered to be wrapped as illustrated in Figure 2.2 (right).

According to above given definition for CAs, they are considered to be synchronous and deterministic. However, in some applications it is desired to have some degree of randomness in the rule or to have an asynchronous update scheme.

2.1.2 The Game of Life

The Game of Life, perhaps the most famous CA algorithm, was introduced by the British mathematician John Conway in the beginning of the 70's [115]. The Game of life is a simple model for simulating artificial life. The automata is implemented on two-dimensional square lattice with periodic boundary conditions (Figure 2.2, right one) where each lattice site is either *alive* (represented by value 1) or *dead* (represented by value 0). Therefore, a single bit is used to represent each site and the lattice is a Boolean matrix. The system evolves by updating each of the lattice sites simultaneously based on the update rules that are determined by the state of each cell and their respective neighbourhood (Moore neighbourhood (Figure 2.1, middle one). The exact rules are as shown in Table 2.1. The Game of Life contains many interesting patterns and the most extensively studied pat-

Living neighbours	Next state (if currently alive)	Next state (if currently dead)
0 or 1	dead	dead
2	alive	dead
3	alive	alive
4, 5, 6, 7 or 8	dead	dead

Table 2.1: *Game of Life: The cell update rules.*

tern is known as the *gliders* (see [109] for details), also the first known finite pattern with unbounded growth. And despite its simple architecture, it has been proved that Game of Life with an infinite-sized universum has the same computational power as a *universal Turing machine* [83].

2.1.3 Lattice Gas Cellular Automata

Perhaps the most successful practical application of cellular automata as computing devices has been in the field of fluid dynamics for fluid simulations [92]. Called Lattice Gas Cellular Automata (LGCA), this class of CA mimics a fully discrete fictitious fluid. Both the positions and velocities of the fluid molecules are discrete and tightly coupled to the discrete lattice. All particles perform free streaming from one lattice node to a neighbouring one in a time period. Next, particles arriving at a node collide with each other, thus exchanging momentum in some deterministic or stochastic way. The collisions on the nodes all occur at the same time and the duration of a collision is assumed to take zero time. The enforcement of conservation of mass, momentum, and energy in a collision results in a fully discrete and simplified, yet physically correct microdynamics. The inherent spatial locality of update rules makes it ideal for parallel processing [41, 92].

With this LGCA dynamics we may then investigate macroscopic variables, that is, averaged quantities such as fluid density or momentum, which vary over time and length scales much larger than those of the microdynamics, and for which we can prove that they behave as a real fluid. The most complete account of LGCA is the book by Rivet and Boon [84]. Other influential monographs on LGCA are [30, 85]. [14] provide a through overview of Lattice gases and CA. Finally, [92] explains LGCA in the context of CA modeling, and also discusses issues related to executing LGCA on a computer.

The HPP model

The HPP model was the first LGCA model invented and introduced by Hardy, Pomeau and de Pazzis in 1973 [81]. In this model the particles are restricted to move on the links

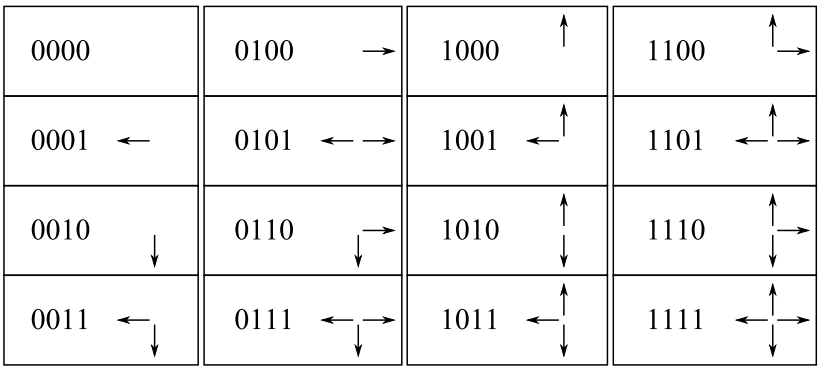


Figure 2.3: *The HPP configurations.*

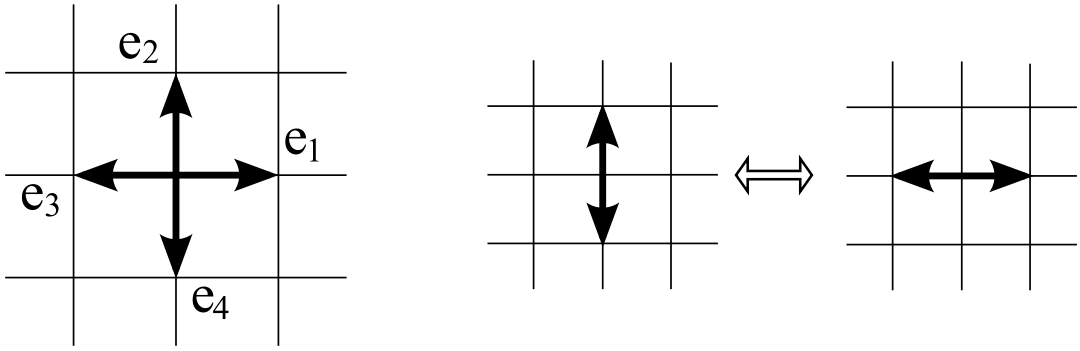


Figure 2.4: *The HPP model: (left to right) Lattice with four velocities and the collision rule.*

of a square lattice (see Figure 2.1 left one). Each particle travels at a unit speed, that is, in each time step it moves from one lattice site to a neighbouring site. A particle can have only one of the four discrete velocities $e_i, i \in 1, 2, 3, 4$ (see also Figure 2.4 left one):

$$e_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}; e_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}; e_3 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}; e_4 = \begin{bmatrix} 0 \\ -1 \end{bmatrix} \tag{2.2}$$

As only one particle is allowed to travel in each direction along a link, a maximum of four particles can arrive at any site at any time step. Therefore, a 4-bit state machine is used to represent each of the HPP lattice site. The possible configurations of particles at each site are shown in Figure 2.3 along with possible coding of the 4-bit state machine. The dynamics of the system from one time-step to another takes place in two successive stages, that is, *collision* and the *streaming* stage. At the start of each time-step, the incoming particles at a node collide following the rules as shown in Figure 2.4 (right). After the collision stage, particles perform a free streaming, that is, each particle travels from its node to the neighbouring node in the direction of their velocity vector. It can be easily seen from the collision rules that both the number of particles and the momentum at each site is conserved, and therefore conserving the mass and momentum for the whole system. Figure 2.5 shows an example system dynamics from time-step $t-1$ to $t+1$, lower

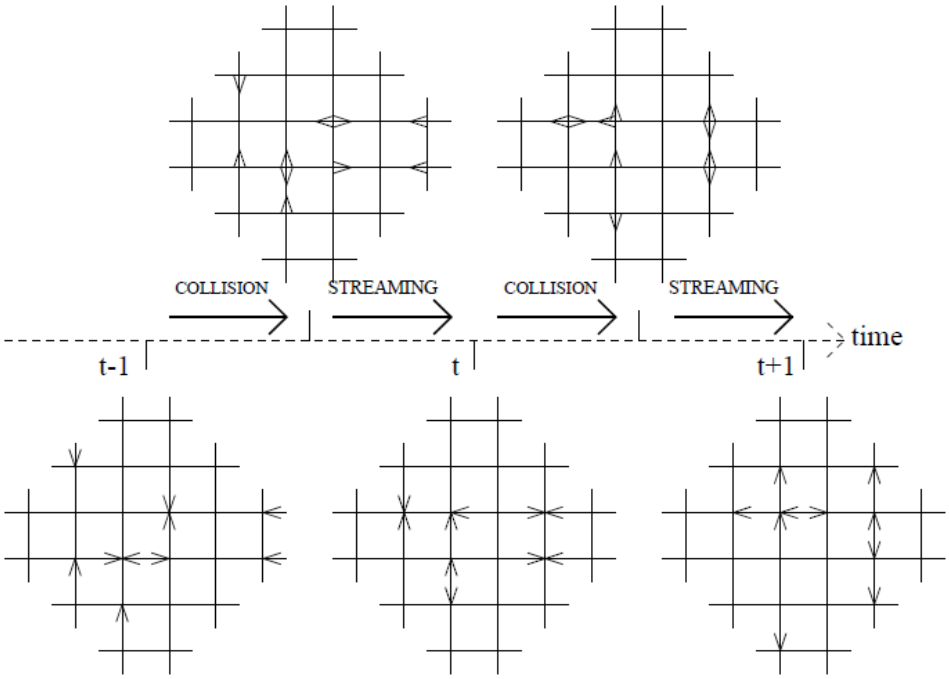


Figure 2.5: *HPP LGA update mechanism. (left to right) Propagation and the collision phases for a portion of a square lattice from time $t-1$ to $t+1$ are shown where arrows represent particles and their directions of motion. Figure taken from [22].*

panel shows the state of the lattice at the three specified time-steps and the top panel shows how system evolves from one time-step to another.

Although the HPP model was the first LGCA, however, it was quickly realised that it was insufficient for correctly simulating flows as governed by the Navier-Stokes equation. Later, it was also realised that this was due to the models underlying square grid and was the main consideration that led to the birth of the FHP model based on a hexagonal lattice where these problems were eliminated (for details see [30, 85]).

2.1.4 Lattice Boltzmann Method

Following the discovery of LGCA as a model for hydrodynamics, in 1988 another model, the lattice Boltzmann method was introduced. This method is reviewed in detail in [100]. The basic idea being that one should not model the individual particles but particle densities, that is, the probability of finding a particle with a certain velocity. This means that particle densities -real numbers- are streamed from cell to cell, and particle densities collide. In a strict sense we no longer have a CA with a Boolean state vector (in fact, the state is now a vector of real numbers, so the state space per node is infinite). However, we can view LBM as a generalized CA, with the same computational structure as explained by [92].

As shown in Figure 2.6, one of the lattice types used for LBM simulations is the D2Q9-lattice where 2 refers to the lattice dimensions and 9 is the number of discrete velocity vectors. For each discrete velocity vector e_0 through e_8 as shown in Figure 2.6 there is a probability density distribution f_0 to f_8 of particles with that velocity. In terms of CA modeling, in every time step, collision and propagation operations are performed at each of the lattice nodes.

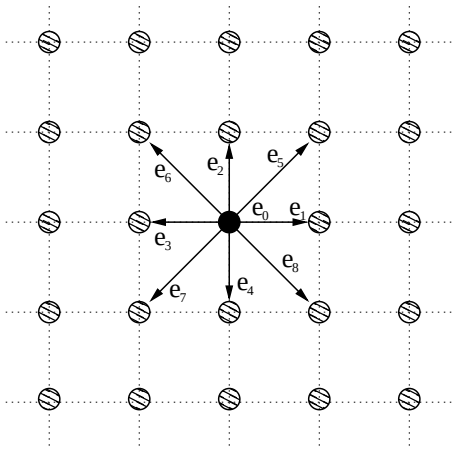


Figure 2.6: *The LBM D2Q9 lattice where $e_i \rightarrow$ discrete velocity.*

The discrete lattice Boltzmann equation with BGK collision approximation (for detailed discussion see [63]) can be described as:

$$f_i(\boldsymbol{x} + \boldsymbol{e}_i\Delta t, t + \Delta t) - f(\boldsymbol{x}, t) = \Omega_i \tag{2.3}$$

where $f_i(\boldsymbol{x}, t)$ is the particle distribution function traveling with discrete velocity $\boldsymbol{e}_i$, with i representing the number of discrete velocities. For a square lattice D2Q9 model as shown in Figure 2.6, particle at rest $\boldsymbol{e}_0$=0, for $i = 1, 2, 3, 4$,

$$\boldsymbol{e}_i = e \begin{pmatrix} cos[(i-1)\pi/2] \\ sin[(i-1)\pi/2] \end{pmatrix} \tag{2.4}$$

and for $i = 5, 6, 7, 8$,

$$\boldsymbol{e}_i = e\sqrt{2} \begin{pmatrix} cos[(i-5)\pi/2 + \pi/4] \\ sin[(i-1)\pi/2 + \pi/4] \end{pmatrix} \tag{2.5}$$

where particle streaming speed $e = \Delta x/\Delta t$. Δx and Δt are the lattice spacing and the step size in time respectively. The right hand side of the Equation (2.3), that is, Ω_i is the so-called BGK collision term that models the collision function of the said model as in Equation (2.6)

$$\Omega_i = -\frac{1}{\tau}[f_i(\boldsymbol{x}, t) - f_i^{eq}(\boldsymbol{x}, t)] \tag{2.6}$$

where τ is the dimensionless relaxation parameter and $f_i^{eq}(\boldsymbol{x}, t)$ the equilibruim distribution function. For D2Q9 lattice, using the macroscopic variables such as density ρ and velocity $\boldsymbol{u}$ one obtains the following form

$$f_i^{eq}(\rho, \boldsymbol{u}) = \rho\omega_i \left[1 + \frac{3(\boldsymbol{e}_i.\boldsymbol{u})}{c^2} + \frac{9(\boldsymbol{e}_i.\boldsymbol{u})^2}{2c^4} + \frac{3(\boldsymbol{u}.\boldsymbol{u})}{2c^2}\right] \tag{2.7}$$

where weighing factor ω_i is $\omega_0 = 4/9$, $\omega_i = 1/9$ for $i = 1, 2, 3, 4$ and $\omega_i = 1/36$ for $i = 5, 6, 7, 8$. And the macroscopic density and velocity are defined by sums over the distribution functions f_i :

$$\rho = \sum_i f_i, \qquad \rho\boldsymbol{u} = \sum_i f_i \boldsymbol{e}_i. \tag{2.8}$$

For FPGA implementations as shown in Chapter 5, parameters $\Delta x = \Delta t = 1$ were in lattice units for the above model.

Ignoring the details of the LBM computation being performed at each node as presented above (or see, for example, [92] and [71] for further details) and to simplify the explanation, we assume each lattice site to be nothing but a black-box. Each black-box is initialised with the lattice node's nine probability density distributions. For the computation, the initialised black box runs a number of mathematical operations over the given nine real numbers and returns with a new set of nine real numbers (collision function: in this case using a BGK collision operator [92]) where f_1-f_8 are exchanged with the node's eight neighbours (propagation function) and f_0 represents the particle at rest internal to the node.

2.2 Reconfigurable Computing

Traditionally to perform a computation, one would write a software program implementing the algorithm that would be executed using a microprocessor- a general purpose computer. Since processors execute a set of software instructions to perform a computation and by changing these software instructions one can modify the functionality of the given system without any changes in the processors' hardware. This general purpose nature of a microprocessor makes it the most flexible method of implementing a computation. However, this flexibility comes at the cost of performance, that is, to execute a program, the processor has to read, decode, and execute each of the instruction thus resulting in a high execution overhead for each instruction. Alternatively, for a faster and efficient computation one would design and build a special purpose computer, customised and hardwired to perform the given computation in the hardware directly. However, this high performance is achieved at the cost of flexibility, that is, the hardwired machine cannot

be altered after fabrication. See [21, 47, 57] for an interesting list of articles on special purpose machines in science in general. At the chip level, Application Specific Integrated Circuits or ASICs [43] are designed specifically to perform specified computations faster and more efficiently, a device such as Grape [72] is an example. Again, the chip cannot be altered after fabrication. Reconfigurable computing with FPGA offer a middle ground that combines the flexibility of the microprocessors and the tailor designed circuitry offered by the ASIC. See [7, 31, 32, 39, 101] for an introduction to reconfigurable computing using FPGAs. The fundamental and powerful concept behind using FPGAs is that, the circuitry within the chip is reconfigurable and tailored as per the application requirements.

2.2.1 FPGA

FPGAs are commodity integrated chips that are completely electrically programmable and can be customised almost instantaneously by the designer to create custom digital logic designs after manufacturing (hence the name field programmable). See [106] for an overview of FPGAs and their programming. [19, 64] are also interesting books concerning digital logic design using FPGAs. Current FPGAs include logic density equivalent to millions of gates per chip [74] and can implement very complex computations. Traditionally, used as glue-logic replacement and rapid-prototyping in networking, telecoms, DSP, etc, FPGAs with improved features like flexibility, capacity and high performance have also opened up new avenues in HPC that form the basis of HPRC.

2.2.2 Structure

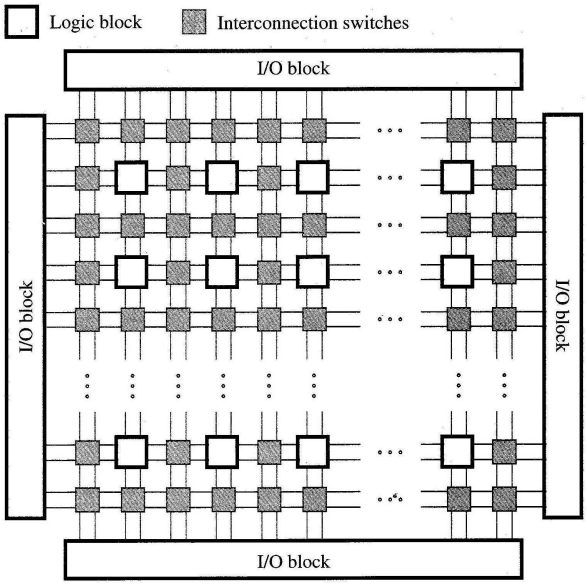


Figure 2.7: *General structure of an FPGA. Image from [19].*

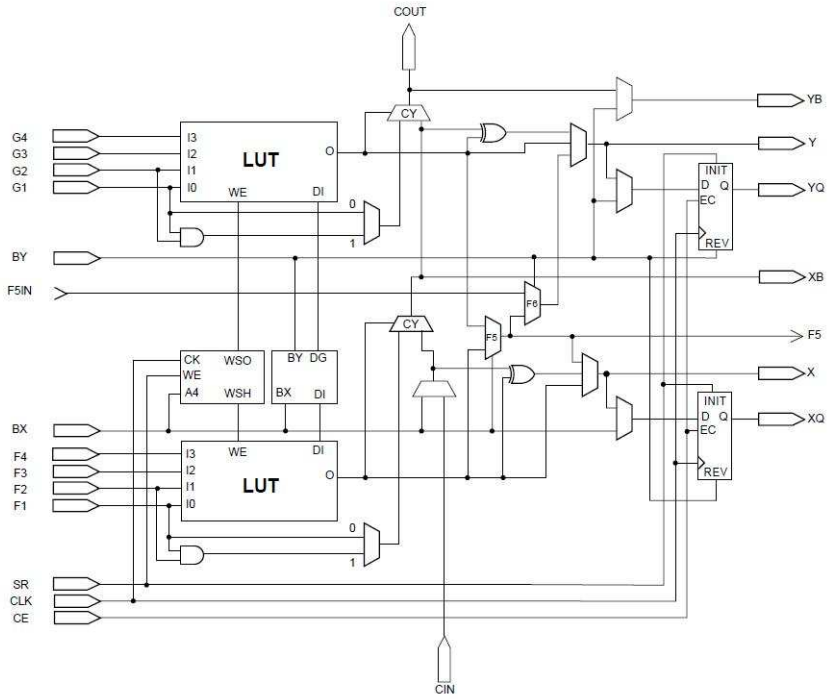


Figure 2.8: *FPGA slice with two 4-input LUT and two Flip-Flops. Image from [43].*

The structure of an FPGA in general as shown in Figure 2.7, is based around a matrix of programmable functional units called configurable logic blocks (CLB) connected via wires that are also programmable. In order to provide connectivity to the outside world, a flexible interface called I/O blocks surrounds this programmable matrix. The majority of the FPGAs are based on SRAM technology [31, 32], that is, the logic blocks and the programmable interconnect use memory cells called static random access memory (SRAM) cells to store programmable information. Hence, the configurable points within the FPGA are SRAM bits and programming these SRAM bits configures the FPGA. Therefore, re-programming these devices is as easy as writing to a standard static RAM in a matter of seconds.

CLB being the functional units for most of the logic implementation include logic cells as their basic building blocks. As shown in Figure 2.8, each logic cell includes one or more lookup tables (LUT), carry logic, and several storage elements called Flip-Flops. LUTs are essentially memories used as arbitrary logic function generators. The basic idea is to store the output value of an n-input function for each input combination in a table, and using current input value to lookup what the value should be by indexing the table. Typically, LUT have either 4 or 6 inputs. Thus using the hierarchical structure, that is, the combination of small scale functions implemented using LUT are interconnected together to form large scale functions. Therefore, using a FPGA chip, a designer can implement a custom digital logic simply by mapping the design's functional units onto the available

logic blocks. The maximum possible clock frequency that can be used to drive the overall circuit depends on the underlying technology and the depth of the computation between the storage elements and the routing wire delays.

Other than the CLB, IOBs and interconnects as shown in the Figure 2.7, FPGAs also include internal memory blocks called block RAMs (BRAM). High-end FPGAs also include many dedicated functional units and peripherals like DSP units, fast I/Os, multiple clock control units and even embedded processors like RISC cores to further boost the performance of the implemented design.

2.2.3 COTS Boards

Typical FPGA based computing resources used are in the form of COTS (commercial off the shelf) boards, see Appendix 10 for the one used for this research. FPGA based COTS boards are widely available and easy to setup. Apart from including high-density, high-performance FPGA(s), they also include on-board memory chips and various other I/O related components. Usually, FPGA chips are connected to several on-board SRAM or DRAM memory chips that can be used for additional data storage requirements.

2.3 HPC using FPGAs

Applications from scientific computing and their ever-increasing demand for computing resources are the two sides of the same coin, and how to further push the performance of HPC applications using HPC systems is always a challenge. In an attempt to boost the performance of HPC applications, currently HPC systems incorporate hardware accelerators like FPGAs chips. Such systems are an extension to the traditional HPC systems with additional reconfigurable devices like FPGAs and are also known as High Performance Reconfigurable Computing (HPRC) [42, 55, 56] or reconfigurable supercomputers [20].

In addition to exploiting the available coarse-grained parallelism across the microprocessors, inclusion of hardware accelerators like FPGAs provides an opportunity to the HPC programmer to further boost the application performance. This is achieved exploiting the massive fine-grained parallelism and pipelining available through direct hardware execution using FPGAs. FPGAs allow a programmer to design and build a custom circuit to compute his explicit computation and therefore, an opportunity to parallalise and pipeline algorithms on the instruction level to achieve tremendous performance gains. Additionally, HPRC systems have also shown orders of magnitude improvement in performance, power, size and cost over conventional HPC systems [42].

Other than proof-of-concept setups like Maxwell [8], RCC [87] or Janus [10], the vendor provided solutions like Cray's XD1 [60], DRC Computers' RPU [34], SGI's RASC [89] and SRC Computers' MAPstations [33], which couple FPGAs with high-end conventional microprocessors, are of increasing interest to the scientists working in computational science and engineering communities.

2.3.1 Exploiting HPRC Resources

Since HPRC combines high performance microprocessors along with reconfigurable devices like FPGAs, all communication is over a high bandwidth and low latency interconnection network [50], both the processing and connecting capabilities can be tailored to suit the needs of the specified algorithm in order to maximise performance. However, to harness the capabilities of the available resources, it is necessary to a) identify kernels with high computational density, b) map possible kernels to hardware accelerators, c) carefully partition the application into software and hardware to reduce overall communication overheads, and d) optimally utilise the available memory hierarchy.

The resources available to the HPRC programmer include programmable logic, on-chip memory, on-chip decided arithmetic blocks, on-board memory, host memory and the high speed interconnects. Therefore, the overall application performance is defined by how efficiently the available resources are configured to maximise the available bandwidth. [55] lists and explains the possible FPGA computing models that can effectively utilise the available computing resources depending on the application requirements. In general, spatial parallelism available within FPGA can be exploited to implement a) pipelined and parallel algorithms (we have implemented pipelining over time strategy for our I/O bound CA implementations as discussed in Chapter 4, shown in Figure 4.2 and multiple processing elements running in parallel implementation as discussed in Chapter 5, shown in Figure 3.2), b) customised data paths (see lattice Boltzmann processing element implementation as shown in Figure 5.2), and c) using internal buffers like FIFOs for temporary data storage to minimise I/O communications. The overall memory hierarchy (on-chip, on-board, and host memory) is exploited to maximise available bandwidth. For example, in our multi-FPGA based lattice Boltzmann implementation as discussed in Chapter 6 and 7, multiple-independent on-board SDRAM memory banks store the lattice and the boundary data, on-chip memory as FIFOs buffering input and output data streams connected to the processing elements, on-chip memory as register file within each processing element storing the cells state during next state computation, and host memory for boundary update over multiple FPGAs. In Chapter 7, we demonstrate using system profiling, a successful implementation of our latency hiding based boundary update operations by

host machines over multiple FPGAs. Such working latency hiding was achieved due to the successful exploitation of the available memory hierarchy.

2.4 Related Work

As CA provide a mathematical representation of a wide range of complex systems [59], it is not surprising to find numerous attempts by researchers both in the past and present to find methods of how to effectively simulate CA systems in an attempt to understand their emergent behaviour, using the available state-of-art computing resources. As an example, one of the famous cellular automata machines has been CAM by Toffoli and Margolus [102]. Margolus states in [73] "our CAM simulations made Pomeau and others realise that lattice gases were not just conceptual models, but might be turned into powerful computational tools". This is a testimony to the success of such machines. See [48] for a list of CA as parallel computing machines. With the availability of programmable chips, such as FPGAs, one can easily emulate a special purpose chip instead of building a special purpose computer using custom VLSI chips [47] that allows virtually every research an opportunity to design his or her special purpose computer using FPGA chips. Consequently, many FPGA-based CA systems have been proposed and implemented. The following section discusses some of the FPGA-based CA systems reported in the current decade.

Researchers at TU Darmstadt developed a series of FPGA-based CA accelerators called CEPRA-machines to answer "How much speedup could be gained by using FPGA technology compared to optimised software?" [52] speculated a thousand times speedup using the then available latest high end FPGA technology.

In 2001, Cappuccino et al. [25] proposed an FPGA-based CA computational engine called CAREM and demonstrated its implementation for an image thinning algorithm and forest fire simulations with a speedup of 65 and 24 respectively. Though the implementation models were simple with a four or less bits per cell state but engine layout used two independent external memory banks to store the current and the next state of the automata respectively. Storing the lattice in the external memory banks ensures that the available FPGA logic resources do not limit the lattice size for simulations. Our implementation design as discussed in Chapter 3 is formulated around this layout which is important for numeric computations based accelerators.

In [92], Sloot et al. specify fluid simulations using cellular automata as one of the most successful practical application of cellular automata as computing devices, and in [97], Smith et al. report FPGA-based accelerators for computational fluid dynamics promise large improvement in sustained performance at better price-performance rations with

lower overall power consumption than conventional processors. Therefore, it is no surprise that number of FPGA-based accelerators for such simulations have been reported, for example, [66, 78, 86, 90, 91].

Kobori et al. [66], reported in a series of publications, results from their FPGA-based implementations for Lattice Gas Models. For a single FPGA implementation of a 3D FCHC Lattice Gas Model they demonstrate a 200 times speedup compared to a software version on a 1.8GHz Athlon processor. They used a parallel and a pipelined processing arrangement to improve both the computation and the FPGA resource utilisation depending on the available FPGA resources. Again, the implementations were non-numerical computations, that is, bitwise operations that demonstrated a speedup for a 3D lattice and an efficient usage of FPGA resources using pipelining over time. Our I/O bound implementations, as presented in Chapter 4, employ this pipelining over time with additional new features.

Nallatech, a commercial vendor for FPGA-based solutions, also reported a FHP-III lattice gas model implementation [91] using their DIME-C environment in 2005, where they reported a performance of 550 times faster than 1GHz Pentium processor which is equivalent to 100 times faster than 4GHz processor. They quote "LG are CA based and as such they are simple models that need to be repeated many hundreds, thousands or millions of times to get a useful result. This would ideally require a great number of relatively simple computers. The question is where such computers exist?" as their motivation behind their work and further explain their approach as "Conventional computers are ideal for the complex models but as inappropriate for the simple models. With RC, we have reverted to the purer ideas of computing where the computer is designed to perform a single task well and then redesign if it needs to perform another task. Thus we have in RC the ability to create multiple, simple, perfectly implemented cells. Each is capable of implementing the simple rules defined by the CA. Their sum total is capable of producing the complexity of results required."

All of the above mentioned works have focused on CA where the state space per cell is finite and therefore these are non-numeric computations. However, the computational structure of cellular automata, in general, matches the generic stencil based structures as also found in well known numerical techniques such as LBM [100] and in the FDTD algorithm [29] [27] that explicitly solve the time-dependent Maxwell equations. We can view LBM as a generalised CA because of their same computational structure as explained by [92], Nallatech also reported their FPGA-based LBM implementations [90] based on their previous FPGA-based CA work [91] in 2005. This work did not demonstrate any speedup, but was most likely the first work to report floating point based FPGA implementation for LBM and a progress from their FHP-III implementation. In the following year, for

2-dimensional time-dependent fluid dynamics problems, [86] reported their streaming accelerator implemented on a Virtex-4 FPGA with PCI Express x8 interface that achieved a speedup of 2.93 and 2.46 compared to a 3.4 GHz Pentium4 processor and a 2.2 GHz Opteron processor respectively. They reported their implementation methodology based on the optimisation of the equations of LBM and then formulating a streaming computation based on their stream compiler for FPGAs [75].

Finite difference or the FDTD algorithm are also numeric and stencil based computations and some of the their noted FPGA-based implementations are [28], [29], [36] and [37]. Using a fixed-point method implementation, [29] reported a 24 times speed-up of a two-dimensional FDTD and as per [36], though this method was not as accurate as floating point but showed tremendous speed-up over the software calculations. Using fixed-point methods for numeric computations like LBM is an interesting idea that could be implemented using an FPGA and validated against a software implementation for the relative errors.

One of the main aims of this project has been scalability, that is, to design a system that can be scaled from a single- to a multi-FPGA based implementation and therefore a capability to simulate large simulation sizes. In Chapter 7 we present a multi-FPGA based implementation using Maxwell- a 64 FPGA supercomputer [8] from EPCC at the University of Edinburgh.

Gokhale et al. define reconfigurable supercomputing as "combines programmable devices like FPGA with high performance microprocessors, all communicating over a high bandwidth, low latency interconnection network" [50] and presents in details the pros and cons of such systems. For the discussion on the feasibility of FPGA supercomputing see for example [35, 50, 87]. [8, 10, 11, 26, 76, 87] are some of the many research groups that have setup FPGA supercomputers. The main idea behind these machines is to run such applications where the control and I/O portion is executed by the microprocessors and the compute and data-intense portion is accelerated using FPGAs. [50] discusses the various commercially available FPGA supercomputing architectures and the more recent trend where the cluster is augmented with FPGA chips for application acceleration are discussed in [35]. Other than having a microprocessor based communication network, machines like Maxwell [8] also include an independent point-to-point link between the adjacent FPGAs thus enabling an FPGA only setup to perform parallel computations. A wide range of applications from the areas of finance [8, 26], medicine [8], petrochemical, cryptography [76], etc have been ported successfully to FPGA supercomputers, however, majority of such applications are trivially parallel as the Monto Carlo simulations [10] with limited data requirements.

2.5 Summary

This chapter introduces cellular automata in general and the various models that were implemented as a part of this work. It also presents reconfigurable platform like FPGA used to implement cellular automata accelerators and how to exploit the available features for overall performance. One of the key focus of exploiting memory hierarchy available in HPRC systems as a necessary feature in order to maximum overall system performance is also covered. Finally, other interesting works in the field of FPGA-based CA implementations and their influence on this work is also presented.

Chapter 3

Performance Modeling of FPGA based CA Implementation*

The FPGA based computation engines appear to be very attractive for CA algorithms as CAs consist of a uniform structure composed of many finite state machines, thus matching the inherent design layout of FPGA hardware. The computational structure of CAs in general, resembles the generic stencil based structures also found in, for example, finite difference of FDTD algorithm. As per [29] FPGA based FDTD implementations have shown a significant performance gain.

However, there are many trade-offs to consider when designing and implementing the FPGA based CA accelerator. For the best possible mapping of CA algorithm to the FPGA logic space, the most important aspect is to fully understand the behaviour of the specified CA algorithm in terms of its execution times, which is either compute or I/O bound.

Performance modeling is a highly popular and commonly used technique in high performance computing [46]. In contrast, there is hardly published work that demonstrates the application of performance modeling techniques in high performance computing using FPGAs. This chapters introduces the application of performance modeling techniques for FPGA based CA computations. The chapter starts by explaining the basic FPGA enabled PC organisation for CA computations and the related performance model. It discusses a scalable FPGA enabled PC organisation for CA computations with identified parameters that are defined both by CA algorithm and the FPGA hardware. Finally, a

*This chapter is based on the following publications:

- S. Murtaza and A.G. Hoekstra and P.M.A. Sloot, 'Compute Bound and I/O Bound Cellular Automata Simulations on FPGA logic', *ACM Transactions on Reconfigurable Technology and Systems*, **1**, 1-21 (2009)

simple metric that categorises a CA algorithm in terms of its execution times for the said FPGA-PC system organisation is also presented.

3.1 Basic Organisation

Consider a basic setup where the CA lattice is small enough that each CA cell is processed using a dedicated processing element (PE). As the whole CA lattice completely fits into the FPGA space, the resulting computing system is therefore simple. For CA computation, the whole lattice is downloaded to the FPGA from the host machine and is run for the required number of generations. Finally, the resulting generation is uploaded back to the host system where all the required pre- and post-processing is performed. However, this setup is feasible only when the whole CA lattice fits the given FPGA space.

3.1.1 Generic Performance Model

Assuming the host system does all the required pre- and post-processing plus the FPGA for CA computations, the performance model for such a system is defined as follows. We assume that we want to compute a CA for g generations. When executed on a stand alone PC system this would take T_{pc} execution time. Using the FPGA enabled PC system the same computation takes T_{ft} execution time, and we write

$$T_{ft} = T_{fpga} + T_{fo} \tag{3.1}$$

where, T_{fpga} is the pure execution time on the FPGA, and T_{fo} is total overhead time to pre and post process the CA lattice and transfer time to move the data to the FPGA and back to the main memory. Note that in Equation (3.1) we assume a serial implementation, that is, the useful computation (T_{fpga}) does not execute at the same time as the overhead work (T_{fo}). However, in many cases an overlap between computation and the overhead work is possible, and then the model needs to be adapted slightly. We can now define the obtained speedup by running on the FPGA as

$$S = \frac{T_{pc}}{T_{ft}} = \frac{T_{pc}/T_{fpga}}{1 + T_{fo}/T_{fpga}} = \frac{S_{max}}{1 + f_o}. \tag{3.2}$$

The maximum speedup S_{max} that can ever be obtained on the FPGA enabled PC system is T_{pc}/T_{fpga}. Only if the overhead time is zero this speedup will be obtained. For finite overhead times the decrease of maximum speedup is determined by a dimensionless number, the fractional overhead $f_o = T_{fo}/T_{fpga}$. Note that if f_o is small, Equation (3.2) can be written as

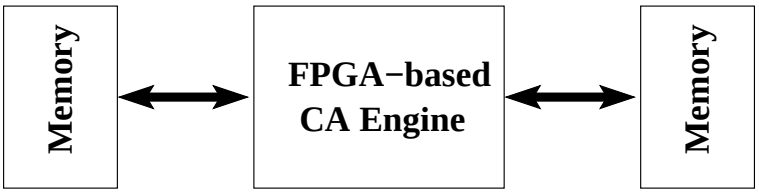


Figure 3.1: *Basic computation model: A scalable FPGA based CA implementation using the two attached on-board memory banks to store the two consecutive CA lattice computations.*

$$S \approx S_{max}(1 - f_o). \tag{3.3}$$

We can also define an efficiency of the FPGA implementation as

$$\varepsilon = \frac{S}{S_{max}}. \tag{3.4}$$

The efficiency is a number between 0 and 1. We assume that the overhead consists of the time to preprocess the CA lattice on the PC (T_{pre}), the time to download the CA lattice from PC to FPGA on-board memory (T_s), the time to upload CA lattice from the FPGA on-board memory to the PC (T_r), and the time to post process the CA lattice on the PC (T_{pos}). With these definitions we can now write

$$T_{fo} = T_{pre} + T_s + T_r + T_{pos} \tag{3.5}$$

and the fractional overhead f_o can now be written as a summation of four different types of fractional overheads, i.e.

$$f_o = f_{pre} + f_s + f_r + f_{pos}. \tag{3.6}$$

with $f_i = T_i/T_{fpga}$ and i $\in \{pre, s, r, pos\}$. The terms above depend on various parameters like clock frequency of the hardware, number of CA cells, number of CA generations computed etc. However, note that such model is only feasible when the whole CA lattice fits the given FPGA space.

3.2 FPGA with Multiple On-board Memory Banks

For real CA applications the lattice is large (say 128^3 cells). Consequently we can assume that the lattice is larger than the FPGA capacity but still fits the memory banks available on the FPGA board. The resulting computing system is composed of a host machine for pre- and post-processing and a FPGA board with on-board memory banks connected as a co-processor for CA computations. With this FPGA based CA accelerator system, the

host machine initially describes the problem to be solved and downloads all the relevant information (CA lattice data) to one of the on-board memory banks of the FPGA board and then starts the FPGA for computations. The FPGA runs the compute engine (CE) implementing the CA transition function and uses the two attached on-board memory banks to store the two consecutive CA lattice computations as shown in Figure 3.1. At each step t, the CE reads the current state S(t) of k cells (where k is the number of CA cells that the FPGA is able to read from the source memory in parallel) from the source memory bank, computes the new state S(t+1) of k cells using p PEs, and writes them out to the destination memory bank. Figure 3.2 shows the system diagram along with the main parameters like k and p. Once the entire CA lattice data is read from the source memory, updated using CE and stored to the destination memory, the roles of the two memory banks are switched and a new iteration similar to the CAREM processor [25] starts. This system organisation enables a totally parallel computation, that is, reading, computing, and writing all are done in parallel and a scalable organisation where the CA lattice size is not confined to the available FPGA capacity but the on-board memory banks which are often quite huge. With the completion of the computations, the host machine uploads the results from the FPGA's on-board destination memory bank for postprocessing.

3.3 Compute and I/O Bound CA Computations

With the general description of the overall system organisation as presented in the previous section, there are many trade-offs to consider when designing and implementing the FPGA based CA accelerator. For the best possible mapping of CA algorithm to the FPGA logic space, the most important aspect is to fully understand the behaviour of the specified CA algorithm in terms of its execution times, which are either compute bound or I/O bound.

We will now restrict ourselves to two-dimensional CA and apply an algorithm based on the alternate use of memory banks as source and destination memory as proposed by [25], and assume that the FPGA is able to read from the source memory, write to the destination memory, and compute cell states all in parallel.

To categorise a specified CA algorithm as a compute bound or an I/O bound, consider a CA lattice with N cells. Assume, the FPGA running p PEs (each PE updates a single cell at a time) in parallel and reads k cells (where k is the number of CA cells that the FPGA's CE is able to read from the source memory in parallel) in time τ_{r}, and the time

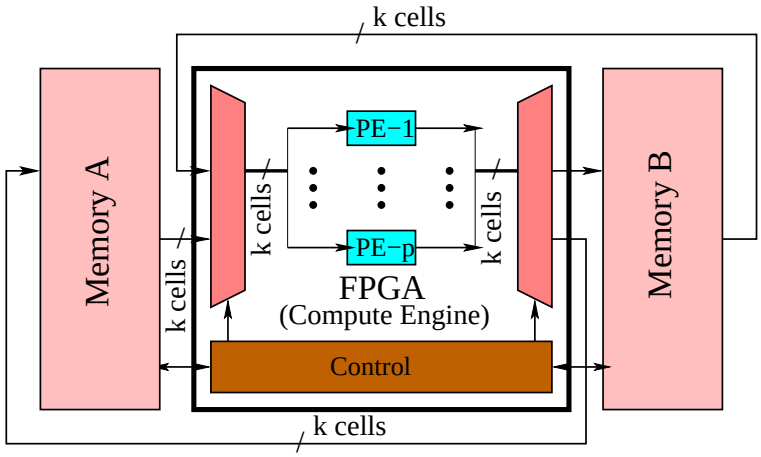


Figure 3.2: *Compute engine implementing the CA transition function with two attached on-board memory banks. Compute engine has p processing elements (PE), where each PE updates a single CA cell at a time. Compute engine reads k CA cells from the source bank and also writes out same to the destination bank.*

to write out k cells from the FPGA into the destination memory is τ_{w} and the time to update a cell is τ_{c}. The total time to compute N cells is $\tau_{\mathrm{c}} \left\lceil \frac{N}{p} \right\rceil$ ($\lceil x \rceil$ is the ceiling of x, i.e., rounding upwards), time to read N cells is $\tau_{\mathrm{r}} \left\lceil \frac{N}{k} \right\rceil$ and time to write N cells is $\tau_{\mathrm{w}} \left\lceil \frac{N}{k} \right\rceil$. Now, we can classify a CA algorithm as *compute bound* computations as long as the following is true:

$$\tau_{\mathrm{c}} \left\lceil \frac{N}{p} \right\rceil \quad > \quad \max(\tau_{\mathrm{r}}, \tau_{\mathrm{w}}) \left\lceil \frac{N}{k} \right\rceil . \tag{3.7}$$

Assuming that $\tau_{\mathrm{r}} \geq \tau_{\mathrm{w}}$ (which for our CA computations is usually true, as the amount of input data per cell is larger then the amount of output data per cell), Equation (3.7) becomes

$$\tau_{\mathrm{r}} \quad \leq \quad \tau_{\mathrm{c}} \left\lceil \frac{N}{p} \right\rceil \left\lceil \frac{N}{k} \right\rceil^{-1} . \tag{3.8}$$

Usually $N \gg p$ and $N \gg k$, and therefore,

$$\tau_{\mathrm{r}} \quad \leq \quad \tau_{\mathrm{c}} \frac{k}{p}. \tag{3.9}$$

Similarly, for I/O bound computations the reverse holds.

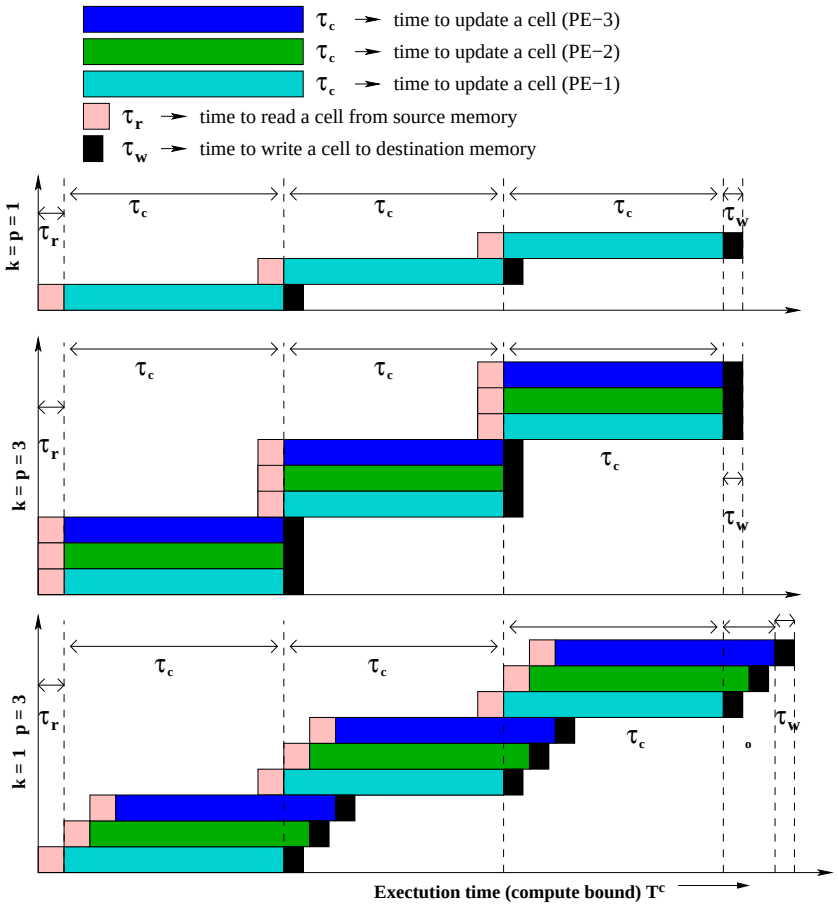


Figure 3.3: *Compute bound scenarios: in upper and the middle panel k (number of CA cells that the FPGA's CE is able to read from the source memory in parallel) is equal to p (number of PEs, each PE updates a single CA cell at a time) and in lower panel $k > p$.*

3.3.1 Execution Time for Compute Bound Computation

For compute bound operations we assume that on start-up the FPGA first reads k cells, after which PEs start to compute. Several scenarios are possible as shown in Figure 3.3. If $p \leq k$ (upper and the middle panel of Figure 3.3) all PEs start to compute, whereas if $p > k$ (lower panel of Figure 3.3) only k PEs will start computing. While computing, a next batch of k cells are read. If PEs still sit idle, they can now start, if not, PEs will immediately start computing on a next cell once they have completed computing on a previous cell. In parallel, the result is written to a destination memory location. Finally, after completion of all computations, remaining data is written to memory. The detailed execution profile depends on k and p, and in Figure 3.3 a number of examples are shown. So, except for start up and final writing of data to memory, all execution time is determined by the computation time.

Define T_{c} as the execution time for compute bound computations. An initial start-up time τ_{r} is required to start the PEs to update cells. The PEs are now busy updating lattice cells for a duration $\tau_{\mathrm{c}} \left\lceil \frac{N}{p} \right\rceil$. At the end of the computations, CE waits for a time $\tau_{\mathrm{r}} \left\{ \left\lceil \frac{p}{k} \right\rceil - 1 \right\}$ for the PEs to complete their remaining computations (this happens only when $p > k$, otherwise this term is 0) followed by a time τ_{w} to write out remaining data to the destination memory bank. The sum of the above mentioned time durations results in the overall execution time for the compute bound scenario as specified below.

$$T_{\mathrm{c}} \;=\; \tau_{\mathrm{r}} + \tau_{\mathrm{c}} \left\lceil \frac{N}{p} \right\rceil + \tau_{\mathrm{r}} \left\{ \left\lceil \frac{p}{k} \right\rceil - 1 \right\} + \tau_{\mathrm{w}}. \tag{3.10}$$

Validating Equation (3.10) for $(k = p = 1)$, we get

$$T_{\mathrm{c}} \;=\; \tau_{\mathrm{r}} + N\tau_{\mathrm{c}} + \tau_{\mathrm{w}}.$$

This is exactly as shown in Figure 3.3 case(a). Similarly, when $(k = p)$ in Equation (3.10) (middle panel in Figure 3.3), we find

$$T_{\mathrm{c}} \;=\; \tau_{\mathrm{r}} + \tau_{\mathrm{c}} \left\lceil \frac{N}{p} \right\rceil + \tau_{\mathrm{w}}.$$

3.3.2 Execution Time for I/O Bound Computation

For I/O bound computations, the CE just starts reading k cells, after which computations start. The reading of cells is repeated $\left\lceil \frac{N}{k} \right\rceil$ times. All computations are done in parallel to this reading. Having read all cells, a final compute and write cycle is required to update the last batch of cells that were read into the FPGA. Figure 3.4 shows a number of possible

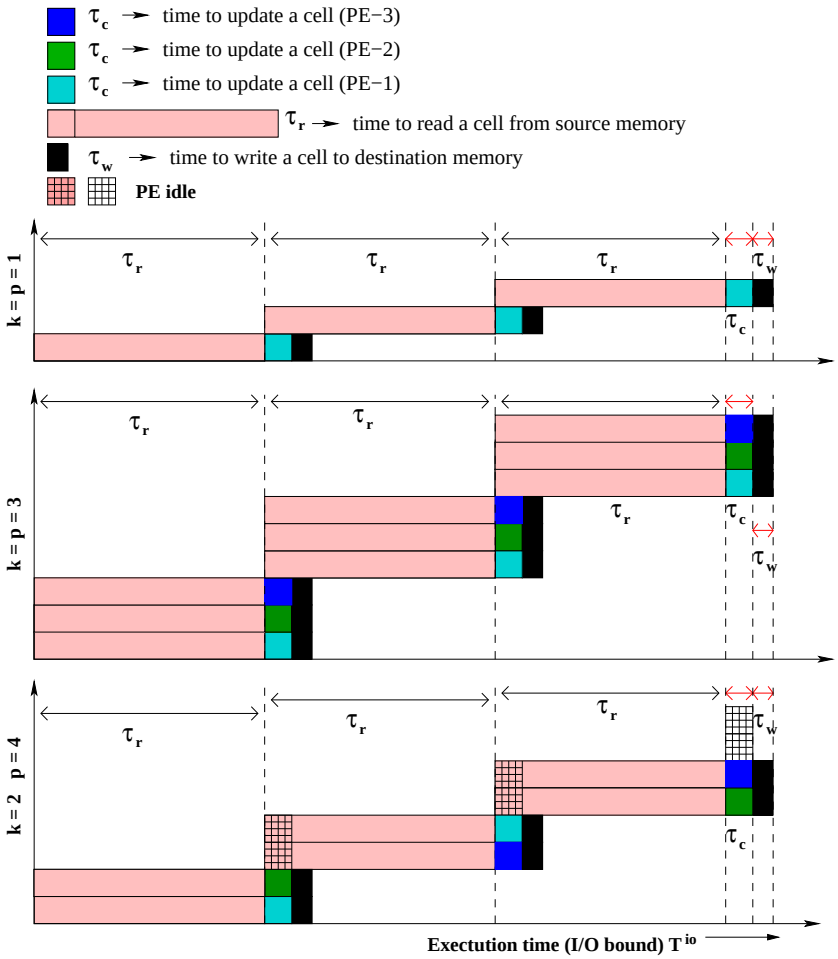


Figure 3.4: *I/O bound scenarios: in upper and the middle panel k (number of CA cells that the FPGA's CE is able to read from the source memory in parallel) is equal to p (number of PEs, each PE updates a single CA cell at a time) and in lower panel, $k > p$ where the mesh of small squares represent idle PEs.*

scenarios, for different values of k and p. Note that when $p > k$ we have the situation that PEs remain idle for the full computation as shown in Figure 3.4 lower panel. In that case one can implement for instance a more advanced CA algorithm, realising a pipeline over CA generations. This is further explained in Section 4.1.2.

The execution time (T_{i}) for I/O bound computations is

$$T_{\mathrm{i}} \;=\; \tau_{\mathrm{r}} \left\lceil \frac{N}{k} \right\rceil + \tau_{\mathrm{c}} + \tau_{\mathrm{w}}. \tag{3.11}$$

Equation (3.10) and Equation (3.11) define T_{c} and T_{i} respectively. They should result in the same execution time when we cross from the Compute bound to the I/O bound case, that is, when (from Equation (3.8))

$$\tau_{\mathrm{r}} \;=\; \tau_{\mathrm{c}} \left\lceil \frac{N}{p} \right\rceil \left\lceil \frac{N}{k} \right\rceil^{-1} \tag{3.12}$$

In substituting Equation (3.12) back in Equation (3.10) and Equation (3.11) respectively, we find a small difference. This has to do with the fact that the expressions for the execution time, take correctly, the start up and end effects into account, whereas these where discarded when deriving Equation (3.8). However, in the limit $N \to \infty$, the small differences disappear, because then the start up and end effects can be neglected.

3.4 Summary

This chapter presents a scalable FPGA based PC organisation where FPGA uses stream processing model for CA computations. Based on FPGA stream processing model, a simple methodology is proposed to categorise a CA algorithm either as I/O bound or as a compute bound algorithm. As presented in the following chapters, this methodology not only hides the details concerning the bit manipulations performed within the FPGA fabric, but also allows accurate prediction of the entire computation performance. Further, I/O and the compute bound categorisation is also used as the foundation for the formulation of the hardware algorithms and their performance model for each of the CA implementations presented in the following chapters.

Chapter 4

I/O Bound CA on FPGA*

In this chapter CA algorithms like the Game of Life and the Lattice Gas HPP model, as introduced in Chapter 2, are of our interest. Next state computation for these types of CAs are a few bitwise operations, implying each cell update lasts a few hardware clock ticks and the PE implementation requires minimal hardware resources.

Considering these two main factors, their hardware implementation has to ensure that input and the output data streams into the CA engine match up with the engine's processing speed and the optimal utilisation of the FPGA resources. Therefore, once a CA algorithm is categorised as I/O bound using a formulation as presented in Section 3.3, the next step is to formulate a hardware design that ensures the maximum utilisation of FPGA resources, and maximises the number of CA iteration computations per a complete data stream flow from source to destination memory bank via FPGA. Standard techniques like pipelining and memory hierarchy can be employed to improve the overall system organisation.

This chapter presents the FPGA-based implementations for the I/O bound CA test benches, that is, the Game of Life and the HPP lattice gas. Various hardware algorithms employed to improve the overall system performance are discussed in detail. System parameters like depth of pipelining, memory storage, bandwidth etc and how they define an optimal design are also discussed in detail. The chapter concludes with some results and possible future extensions.

*This chapter is based on the following publications:

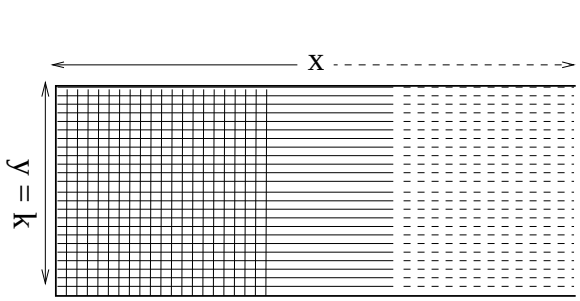
- S. Murtaza, A.G. Hoekstra, and P.M.A. Sloot, "Performance modeling of 2D Cellular Automata on FPGA," in *17th International Conference on Field Programmable Logic and Applications (FPL'07)*, pp. 74–78. IEEE, Amsterdam, August 27–29, 2007.
- S. Murtaza, A.G. Hoekstra, and P.M.A. Sloot, "Performance evaluation of FPGA-based Cellular Automata accelerators," in *Proceedings of the Third Annual Reconfigurable Systems Summer Institute (RSSI'07)*, (on line proceedings) +7. Reconfigurable Systems Summer Institute, National Center for Supercomputing Applications, Urbana, Illinois, July 17–20, 2007.

4.1 I/O Bound 2D CA on FPGA

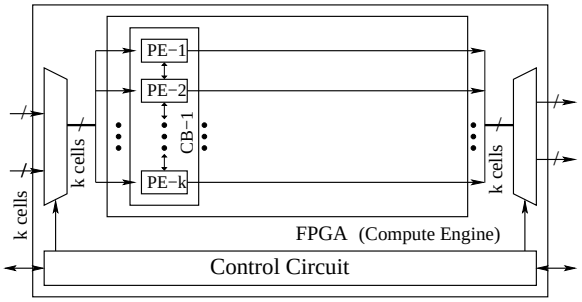
The Game of Life and the Lattice Gas HPP model as introduced in Chapter 2, are both very simple CAs with one-bit and a four-bits state per cell respectively. Since their computational structure is simple, the FPGA resource utilisation per PE is quite low (that is, few logic gates to compute the next state and few memory elements for state storage). Given the fixed bandwidth between the FPGA and the memory banks, and the low FPGA resource utilisation per PE, a simple implementation would read k cells from the source memory into the FPGA, and have equal number of PEs implemented in the FPGA for parallel computations. As shown in Figure 4.1, a k PE implementation to compute k cells in parallel make a compute block (CB). Though this design layout reads, computes and writes k cells in parallel respectively, the FPGA resources are massively under utilised. In order to improve both the computation and the FPGA resource utilisation, a number of CBs are arranged in a pipeline as proposed by [65] and as shown in Figure 4.2. The number of CB implementations also depends on the available FPGA resources as explained in Section 4.1.2.

4.1.1 Basic Computation Model

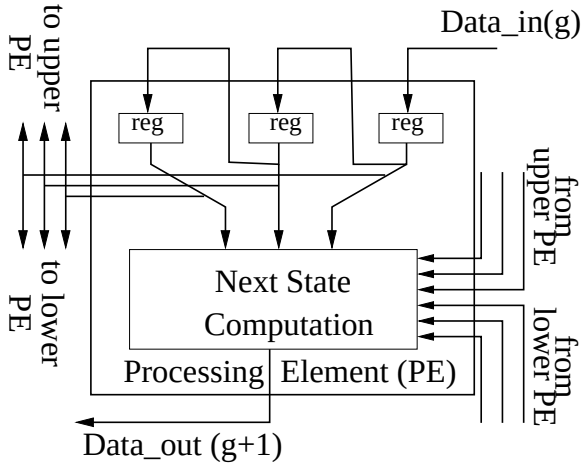
In order to explain the detailed strategy of the computation method for our I/O bound CA accelerator implementation as shown in Figure 4.2, consider a simple scenario with a CA grid consisting of x columns and y ($= k$, FPGA-memory bandwidth) rows as shown in Figure 4.1(a) with periodic boundary conditions (boundary conditions are discussed in section 2.1.1, Figure 2.2). A simple implementation with a single CB as shown in Figure 4.1, reads and computes a column (which is equal to k cells) in parallel. For a CB to compute a column implies, performing a collision and propagation within the FPGA, and to do this each CB has to store three consecutive columns, that is, left, middle, and the right column to compute the next state for the middle column. Therefore, each PE implementing the CA collision and propagation function are as shown in Figure 4.1(c). With each clock tick, a PE updates a single cell. In order to update a cell, a PE: a) stores three consecutive cells of a row, where the one in the middle is to be updated, b) receives the required upper three neighbouring cells from the neighbouring top PE, and c) receives the required lower three neighbouring cells from the neighbouring lower PE. And with all of the information available, the PE updates a cell and flushes out it's updated state. For this setup, lets assume, the FPGA reads the whole column in parallel in time τ_{r}. Since the boundary conditions along the top and bottom edge of the lattice are available within the y cells, the single CB outputs exactly y cells with correct computed next state. With the FPGA engine able to read (and write in parallel as well) k cells per column in time τ_{r}, the



(a) CA lattice.



(b) I/O bound Compute Engine having one Compute block, that is, k-PEs



(c) PE without any buffering

Figure 4.1: *I/O bound FPGA-CA mapping (basic scenario).*

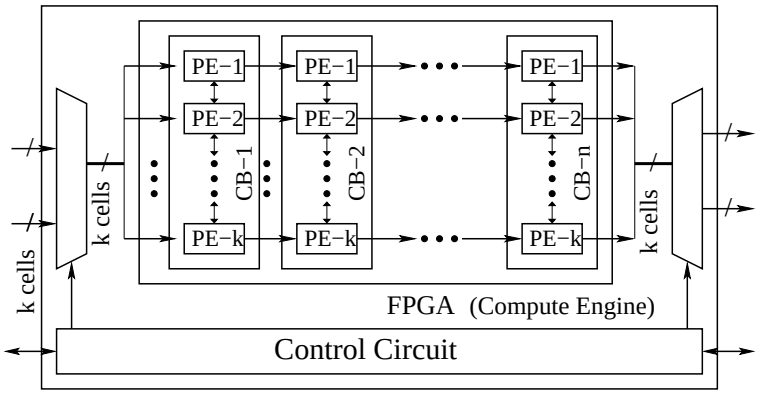


Figure 4.2: *I/O bound system: I/O bound implementation extends the basic computation model with single CB as shown in Figure 3.2 with n CB pipeline.*

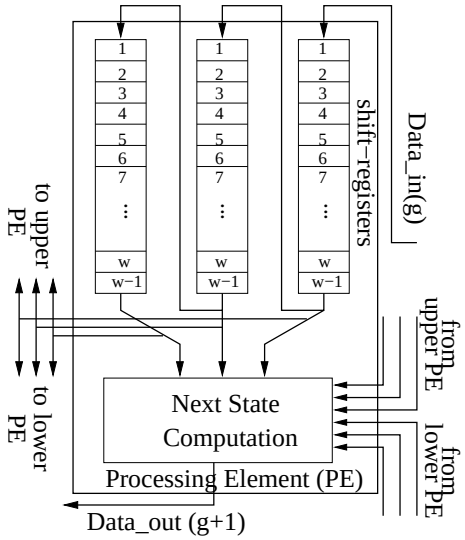


Figure 4.3: *I/O bound influenced PE implementation: PE stores a portion (depending on the available FPGA resources) of the CA lattice's three consecutive columns i.e. left, middle and the right column in order to compute (collision function) the next state for the middle column. With the completion of collision function, PE propagates the update state to its two neighbouring PEs within the same compute block as itself.*

execution times are completely I/O bound. Therefore, the time to compute x columns is just the time needed to read them into the FPGA: $x \times \tau_{\mathrm{r}}$.

However, if the boundary conditions along the sides of the lattice are not available at the beginning of the computation (which is true for periodic boundary conditions), the CE has to re-read the first two columns of the lattice once it is done reading the x columns, in order to correctly compute the next state of the y cells of the last and the first column of the lattice. This results in an overhead of re-reading two columns when the number of CB=1 in the CE. If hard boundary conditions are considered then this overhead is zero.

4.1.2 Pipelined Computation Model

Based on the computation model presented in the previous section, that is, the CE with a single CB (similar to Figure 3.4, upper panel), the CE first reads the state of k cells from the source memory bank. After the CB computes the next state of k cells, the CE finally writes out this new state into the destination memory bank. Unlike Figure 3.4, lower panel, where the FPGA space is under utilised with some PEs sitting idle, we can further improve the computations and FPGA hardware resources by implementing multiple CBs in the CE. Instead of using only a single stage CE engine where only a single generation is computed, we can use multiple CBs connected in a pipeline as shown in Figure 4.2. This pipelined CE results in the computation of n generations in a single sweep (that is,

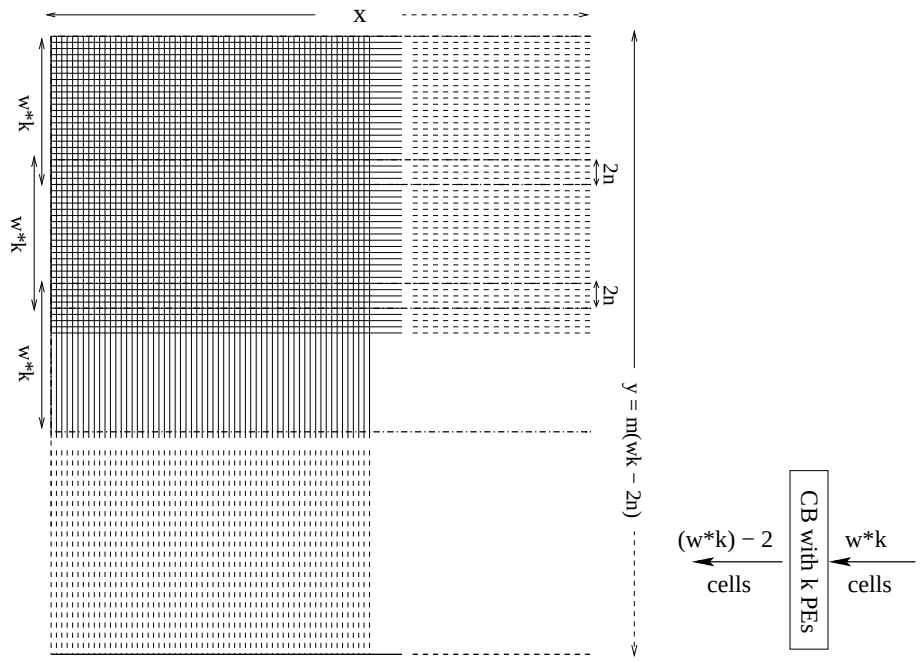


Figure 4.4: *(left) CA lattice with $y \gg k$, (right) Loss due to boundary conditions.*

the whole CA lattice from source memory is passed through the CE for computations, and the computation results are stored in destination memory) where n is the number of CBs connected together. This pipelined model has been successfully implemented by [65] but without alternating the roles of the two memory banks as source, and the destination memory after every single sweep.

With a pipelined CE employing n CBs, we still get y cells with correct output but, the overhead of re-reading columns due to boundary conditions along the sides of the lattice goes up by $2n$. And the overhead time is $2n\tau_{\mathrm{r}}$.

Moreover, with n CBs, it requires some time to fill the pipeline before the CE starts writing out computed data. This startup time is $(n+1)\tau_{\mathrm{r}}$.

Therefore, for I/O bound CA simulations with $y = k$, the total execution time to compute n generations is

$$T_{\mathrm{i}} = (x + 3n + 1)\tau_{\mathrm{r}} \tag{4.1}$$

With alternate use of memory banks as source and destination memory, we can compute g generations simply by reusing the CE that has n CBs only by swapping the role of memory banks as required. So to compute generation g, it requires sweeping the whole CA lattice through the CE b times where $b = \dfrac{g}{n}$, then the total execution time is simply $b \times T_{\mathrm{i}}$.

4.1.3 Pipelined Computation Model with Internal Buffers

Next we consider a more realistic scenario with $k <$ FPGA internal memory $\leq y$ as shown in Figure 4.4 (left). Now, we can use the available internal memory within the FPGA to buffer $w \times k$ cells within each CB (each CB has k PEs as shown in Figure 4.2 and the three internal buffers of every PE now have a capacity of w each as shown in Figure 4.3). This enables us to store the data for w computational planes as shown in Figure 4.4 during CA computation.

As $y \gg w$, the CE no longer has the boundary conditions available along the two edges of the $w \times k$ cells wide computational plane. Therefore, each CB outputs the two cells along the two edges with wrong states as shown in Figure 4.4 (right). With the n CB pipelined engine, the CE outputs $2n$ cells with wrong states and this results in the overlap of computation planes along x-axis that need to be recomputed in order to get correct results. As shown in Figure 4.4, to compute n generations, that is, sweeping the CA lattice once through the CE, the CE has to sweep the whole CA lattice from the source to destination memory in m computational planes where m is

$$m = \frac{y}{wk - 2n}. \tag{4.2}$$

With this setup, to compute g generations, the total execution time is now

$$T_{\mathrm{i}} = bmw(x + 3n + 1)\tau_{\mathrm{r}}. \tag{4.3}$$

Substituting for b and m, we get

$$T_{\mathrm{i}} = \frac{gyw(x + 3n + 1)\tau_{\mathrm{r}}}{n(wk - 2n)}. \tag{4.4}$$

4.1.4 Optimal Design

The number of system parameters like k, n, w, etc are now identified. How these parameters are related, helps to describe the optimal system implementation. Next we try to find an expression for the optimal value of n, that is, the depth of the pipeline. Since we can safely assume that $x >> 3n + 1$, we can reduce Equation (4.4) to

$$T_{\mathrm{i}} = \frac{gwxy\tau_{\mathrm{r}}}{n(wk - 2n)}. \tag{4.5}$$

Equation (4.5) is minimised by taking n equal to

$$n_{\mathrm{optimal}} = \frac{wk}{4}. \tag{4.6}$$

Substituting n_{optimal} for n in Equation (4.5), we get

$$T_{\mathrm{i}} = \frac{8gxy\tau_{\mathrm{r}}}{wk^2}. \tag{4.7}$$

thus confirming the intuitive expectation that a FPGA with larger internal memory or capacity to hold more logic and improved I/O interface between the FPGA and on-board memory banks improves the execution time accordingly. Equation (4.7) also states explicitly how the parameters defined by the implementing technology can impact the performance of the CA accelerator.

The CE implementation based on the pipelined model having n CBs, and alternating the roles of the two memory banks as source and destination memory after every single sweep (provided the two on-board memory banks are completely independent), reads k cells from source memory, computes $n \times k$ cells and writes k cells to the destination memory in parallel. With the FPGA configured, the user initiates the process of initialising the source memory, next the CE is run for a specified number of generations. The CE alternatively reads out the data from one of the on-board memory banks and writes out the computed results to the other memory bank. When the CE is done with the computations, it signals the host accordingly.

4.2 Test Cases

To demonstrate the proposed performance model for I/O bound two dimensional CAs, we implemented and validated our model for two different CA algorithms: The Game of Life and the HPP lattice gas automata.

Consider a 1024x1024 Game of Life lattice that requires to be computed for 512 generations. The performance model for such an implementation as per Equation (4.5) has parameters $x = y = 1024$, $g=512$ and these values are defined by the application. The rest of the parameters for Equation (4.5) are specified by the FPGA implementation technology, that is, the number of bit-wide Game of Life cells the CE can read from source SRAM specifies k= 16 (see Appendix 10 for Spartan-3 FPGA-board details), w and n depend on the available logic space within the FPGA chip and τ_{r} is determined by the hardware clock frequency. With varying values for n and w in Equation (4.5), the respective execution times for the Game of Life are as shown in Figure 4.5.

Similar to the Game of Life computation, if we again consider a 1024x1024 square lattice for the HPP model that requires to be computed for 512 generations, all the variables defined by underlying FPGA implementation technology remain the same except k=4, which is the number of four-bit wide HPP cells CE can read from source SRAM (see Appendix 10 for Spartan-3 FPGA-board details). Again, with varying values for n and w Equation (4.5) for the HPP model, the respective execution times for the model behave somewhat similar to the Game of Life model as shown in Figure 4.5.

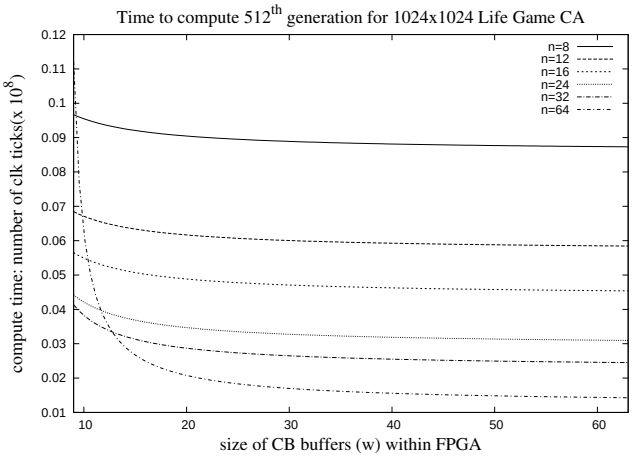


Figure 4.5: *Performance model as specified by Equation (4.5) to compute* 512^{th} *generation of The Game of Life. Each curve represents the execution time for a given number of pipeline stages (that is, number of CBs* $n = 8, 12, 16, 24, 32, 64$ *respectively) as a function of internal buffer size of a pipeline stage or CB.*

4.3 Results

The Spartan-3 board (for details see Appendix 10) has two SRAM chips which are however, not completely independent. Therefore, we had to read and write to the source and destination memory alternatively from the CE running on the FPGA. Since the Spartan-3 board runs with a maximum clock frequency of 50MHz and the available asynchronous SRAM chips on board have access time $\leq$ 10ns, this permitted us to read, compute and write in two clock cycles, implying that τ_r is 2×20ns for our system implementation.

Specific to our Spartan-3 board, for the Game of Life implementation, *k=16* and the maximum logic that was possible to fit the underlying FPGA chip was with *n=16* and *w=9*. These values also resulted in the best possible computation time for the available FPGA hardware resources for the given setup as shown in Figure 4.6.

For the HPP model implementation, Spartan-3 board defines *k=4* and the maximum logic that was possible to fit the underlying FPGA chip was again with *n=16* and *w=9*. But, for this case these values for n and w respectively result in non-optimal computation time for the given setup. The best computation time was achieved with *n=8* and *w=16* as shown in Figure 4.7.

4.4 Conclusion and Future Work

The two dimensional I/O bound CA implementations presented in this chapter were based on the combination of hardware algorithms as proposed by [25, 65] respectively. Other than combining the best design features from [25, 65] implementations, our work presents a

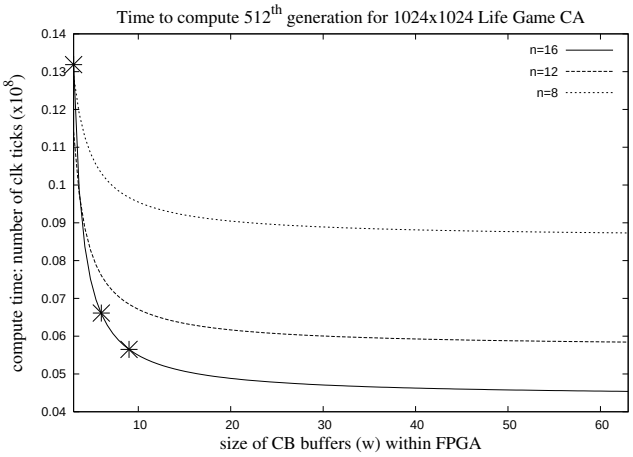


Figure 4.6: *Execution time to compute* 512^{th} *generation for The Game of Life. Curves are the performance model as specified by Equation (4.5)and the points are the measured execution times and specify the best possible values for w and n for implementation using Spartan-3 board (see Appendix 10 for Spartan-3 board details).*

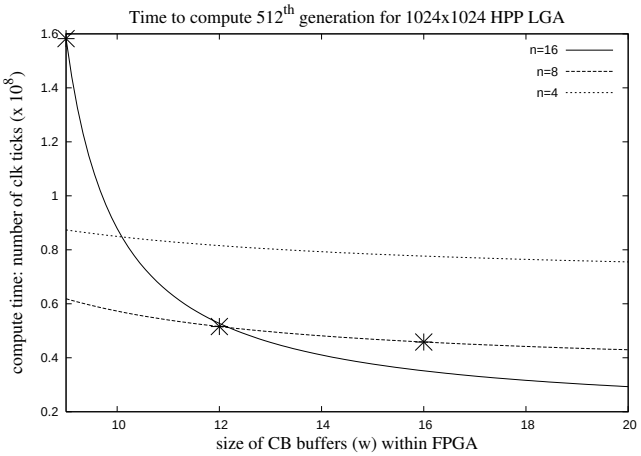


Figure 4.7: *Execution time to compute* 512^{th} *generation for The HPP model. Lines are the performance model as specified by Equation (4.5) and the points are the measured execution times and specify the best possible values for w and n for implementation using Spartan-3 board.*

model to evaluate the performance of a two dimensional CA executed on a typical Spartan-3 FPGA board. The performance model explicitly showed how the various parameters defined by the FPGA technology control the CA accelerator engine implemented in the Game of Life and the HPP model. Using this approach we were able to predict the performance of the algorithm and exploit the parameters optimally for a specific FPGA technology, long before the algorithm was implemented and run in reality. As a result of this, the FPGA hardware resources are used in a way that enable the implementation of the fastest possible accelerator, for the application, using specific technology. The pipelined CA computation engine as proposed by [65] was further improved by a) alternate uses of memory banks as source and destination memory, b) the parameters that define the number of pipeline stages as specified in Equation (4.6) and categorise the overall hardware design as a scalable algorithm for I/O bound CA implementations.

Possible future extensions are: to look into the data transfer between the host machine and the FPGA, and include the measurements in the given speedup equations; to use advanced hardware platform with V5 Xilinx FPGAs with multiple independent memory banks for implementations; and to further investigate three dimensional I/O bound CAs.

Chapter 5

Compute Bound CA on FPGA*

After discussing bitwise operation based CA algorithms in the previous chapter, we next move on to the numeric computation based CA algorithms like the lattice Boltzmann method ([92] explains how we can view LBM as a generalised CA). The lattice Boltzmann method is a well known stencil based numerical technique for physical simulations. The floating-point numbers constitute the state of a lattice Boltzmann cell, therefore, its next state computation is based on floating-point operations. Floating-point based CA computations result in a longer cell update and more FPGA resource utilisation per PE implementation. The compute time and the FPGA resource utilisation, per lattice Boltzmann method PE implementation, suits our goal of validating of our performance model for compute bound computations.

Once a CA algorithm is categorised as compute bound using the formulation as presented in Section 3.3 and considering the floating-point computation requirements, formulating a hardware design to ensure the optimal utilisation of FPGA resources and the overall performance gain is a challenge. Hardware design has to ensure that the source and destination memory are not overwhelmed by the computation engine's data streams. Additionally, each PE implementation demands huge FPGA resources, hence pushing more PEs in a design becomes challenging.

In this chapter, using the D2Q9 Lattice Boltzmann Method, we implement and validate our proposed performance model for the compute bound two-dimensional CA algorithm. Hardware algorithms, employed to improve the overall system performance, are also discussed in detail. The chapter concludes with performance results and the possible future extensions.

*This chapter is based on the following publications:

- S. Murtaza and A.G. Hoekstra and P.M.A. Sloot, 'Compute Bound and I/O Bound Cellular Automata Simulations on FPGA logic', *ACM Transactions on Reconfigurable Technology and Systems*, **1**, 1-21 (2009)

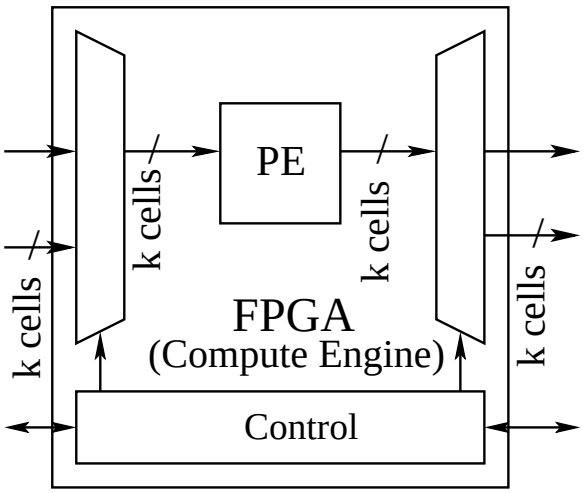


Figure 5.1: *A single PE LBM implementation.*

5.1 Compute Bound 2D CA on FPGA

Influenced by the computational structure and quite high FPGA resource utilisation per PE, we chose LBM as a test case for our compute bound CA accelerator implementation. And to maximise the overall FPGA resource utilisation, we begin by extending a single PE to a p PE implementation, as shown in Figure 5.1 and 3.2 respectively.

5.1.1 Computation Model

The computation method for our compute bound CA accelerator implementation as shown in Figure 3.2 is same as specified in Equation (3.10). We start the computations with an assumption that each of the CA cell needs to compute its collision function followed by the propagation function. So the input data is the state of a cell (nine numbers representing the particle densities in the D2Q9 model) plus memory locations to which the post collision particle densities must be propagated. For the computation of the collision function, each of the cells is self contained, that is, has all the data it requires to compute its collision function. This simplifies the accelerator implementation, with the simplest scenario being a single PE implementation as shown in Figure 3.3 top panel. A single PE implementation layout is as shown in Figure 5.1. PE as explained in next section, is the hardware core that implements the collision function of a compute bound CA algorithm. In order to start the computations for our chosen test bench and the PE implementation, a CE at least needs a single cell's data to set a PE running. To initialise the CE, the specified CA cell's data (that is, nine 32-bit floating-point number, where eight are the particle densities from the cell's eight respective neighbours and the last one representing the cell's own particle at rest) is read from the source memory bank and stored to a PE's register file.

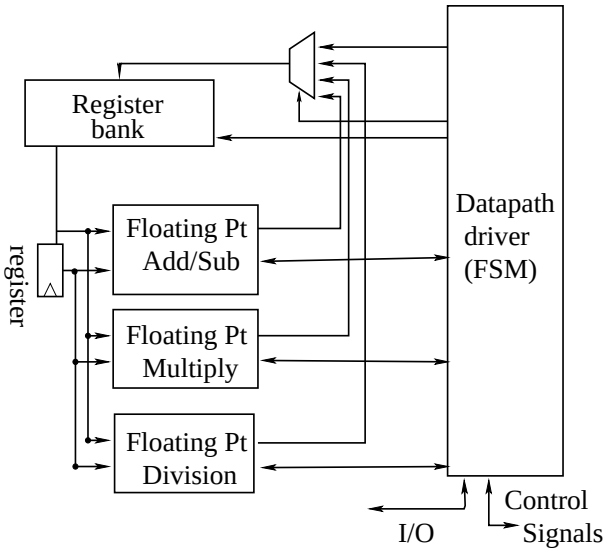


Figure 5.2: *PE implementing a LBM transition: Main components of a PE includes a) three floating point cores (each for sums, multiplications and divisions operations respectively) b) 64 register (each 32-bit wide) and control block to drive the engine.*

Once initialised, the PE is busy updating cells (collision computation phase), while in the meantime the CE buffers input data cells from the source memory bank in order to have it processed in the following cycles by the PE. As the PE computes each cell's new state, that is, nine new floating-point numbers (completion of collision phase), the CE writes them out to the specified location (propagation phase) in the destination memory bank.

For us to further improve and maximise the execution time and the FPGA hardware resource utilisation respectively, we implement p PEs in the CE as shown in Figure 3.2 and have them update p cells in parallel. The CE uses memory banks A and B alternatively as source and destination memory to hold the CA lattice. The maximum possible value for p is defined by the FPGA resource utilisation per PE and the chosen D2Q9 LBM test bench's time to compute a cell. In order to appreciate how the chosen test bench's execution time limits p, assuming unlimited availability of FPGA resources, p can be increased as long as $p \times \tau_{\mathrm{r}} < \tau_{\mathrm{c}}$ as shown in Figure 3.3 lower panel.

5.1.2 PE Design

Processing Engine (PE) implementation for our chosen test bench Lattice Boltzmann method as shown in Figure 5.2, includes a 64 x 32-bit register file, three floating point cores (one for addition, multiplication, and division respectively), and a control block to drive the data path. To compute a specified CA cell's next state, first the PE's register file is initialised with the nine 32-bit floating-point number that represent the cell's current state. With the PE initialised, the control block (a collection of finite state machines)

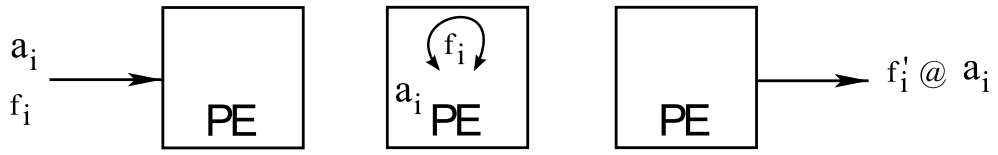


Figure 5.3: *Collision and Propagation: (Left) To compute the next state of a LBM cell, each cell's data, that is, particle distribution function represented as f_i (see Chapter 2 for LBM introduction) and the propagation information as a_i is fed to the PE. (Middle) Once loaded, PE stores propagation addresses within the internal buffers and computes the next state using f_i. (Right) Finally, the computed next state of the cell, f_i' is written out (which is propagation part) to the specified memory locations a_i in the destination memory bank.*

drives the data path to compute the collision function. The collision function computation includes around 75 floating-point operations that are performed sequentially over the given nine numbers using the required embedded floating-point cores within the PE. During the collision function computations, the register file is also used to store the intermediate results. Finally, the nine new floating-point numbers (completion of collision phase) are written out to specified memory locations in the destination memory bank, that is, the propagation phase. The three phases of a PE (initialisation, collision and propagation) to compute a specified CA cell's next state are as shown in the top, middle and the lower panel of Figure 5.3.

5.2 Test Cases

For demonstrating the proposed performance model for the compute bound two-dimensional CA algorithm, we implemented and validated our model for the D2Q9 Lattice Boltzmann Method. Since LBM calculations require the use of 32-bit floating-point numbers throughout the computations, this results in longer cell update and more FPGA resources per PE implementation. For floating-point computations multiple floating-point IP-cores are embedded and employed within each PE implementation. And for each LBM cell update, the PE performs 75 floating-point operations (38 additions/subtractions, 27 multiplications and 10 divisions). The compute time and FPGA resource utilisation per LBM PE implementation suits our goal of validation of our performance model for compute bound computations.

We implemented our LBM system with periodic boundary conditions as shown in Figure 3.2 with number of PEs p equal to two, four, eight, and sixteen respectively. Each of the LBM implementations was tested for four different lattice sizes (N equal to 8×8, 16×16, 32×32 and 64×64) that were computed for $g = 512$ generations.

The state of each LBM cell is defined by nine floating-point numbers and as a result the whole LBM lattice grid is composed of $9 \times N$ floating-point numbers. Since we start the computations with an assumption that each of the CA cells needs to compute its collision function followed by the propagation function, we also load the source memory bank with the data required for the propagation function. For the propagation function, each cell requires nine address locations, and for the whole LBM lattice this adds up to another $9 \times N$ numbers. Therefore, the source memory is loaded with $2 \times 9 \times N$ (32-bit) words. With source memory loaded, LBM CB starts reading cells from source memory, updates cells (collision function) in PEs, and writes out (propagation function) updated cells to destination memory all in parallel. Once the whole LBM lattice is updated for a single generation and loaded into destination memory accordingly, the computation comes to an end. Other parameters like τ_{r}, τ_{c}, and τ_{w} depend on the FPGA chip and are discussed further in the results section.

5.3 Results

For the LBM implementation we used an ADM-XRC-4FX PCI Mezzanine board from Alpha-data. It is a Xilinx Virtex-4 FX140 based PMC with four independent 256MB DDR2-SDRAM banks. Each DDR2 bank can be accessed independently from the FPGA (user logic) and via PCI interface from the host machine. Detailed FPGA board specifications are presented in Appendix 10. The board comes with a software development kit, including drivers, header and library files, that support a C or C++ program running in the host machine to communicate directly with the board. Additional code is provided for board initialisation and selection, control of the programmable clocks and handling of FPGA configuration files. VHDL was used to describe the behaviour and structure of the algorithms. The VHDL code was compiled and synthesised using Xilinx ISE 9.1 design tools.

Specific to our ADM-XRC board, for the LBM implementation $k = 1/10$ and a maximum of 16 PEs p were implemented. Our LBM PE implementation consumes 674 FPGA core clock-cycles to update one LBM cell. This number was obtained from the VHDL simulations of our LBM PE. Transferring a single word (64-bits) from a source memory bank into the FPGA takes on an average 1.1 FPGA core clock-cycles (1.1 specifies that our system implementation manages 90% bandwidth utilisation since we include the DDR2 memory overheads, that is, latency due to the first read from core, effects like memory page changes and refreshes etc).

First, to demonstrate our proposed performance model for the compute bound two-dimensional CA algorithm, we implemented and validated our model for the D2Q9 Lattice

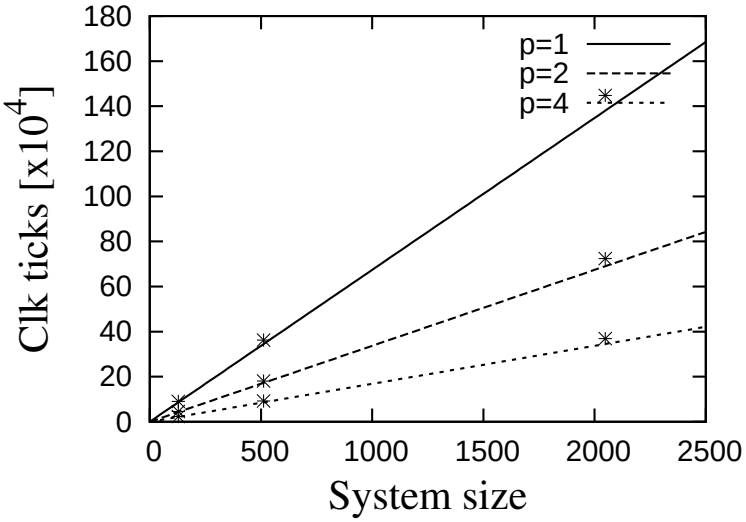


Figure 5.4: *Execution time to compute a single generation for the LBM model. Lines represent the performance model as specified by Equation (3.10) and the points represent the measured execution times using ADMXRC Virtex4-FX140 board.*

Boltzmann Method with number of PEs p equal to two, four and eight respectively. With this setup as shown in Figure 5.4, varying simulation sizes defined N equal to 8×16, 16×32 and 32×64 respectively. Plugging these numbers in Equation (3.10) we get the theoretical number for our LBM implementation. For the hardware side measurements we used a counter implemented in the FPGA core. The counter measures the number of FPGA core clock-cycles for the entire duration of a single generation of the LBM lattice, starting from the initialisation of the CB in FPGA to the flushing out of the data at the end of the computation. Figure 5.4 shows measurements together with theoretical results where the lines represent our model and points are the measured execution times. The LBM implementation takes longer than the theoretical results due to usage of DDR2-SDRAM banks, and some redundant finite state machine cycles within our LBM CB implementations. However, the model predictions are accurate within 7%.

Further, we used a Dell PC with an Intel P4 2.99-GHz and 3.25-GB RAM to measure the microprocessor-based performance for our D2Q9 Lattice Boltzmann implementation in Fortran, and compared this with our FPGA-based implementations. For this setup we extended our FPGA-based LBM implementation to a maximum of $p = 16$ PEs implementation. Each of the LBM implementations with number of PEs p equal to two, four, eight, and sixteen respectively were tested for four different lattice sizes (N equal to 8×8, 16×16, 32×32 and 64×64) that were computed for $g = 512$ generations. The resulting execution times are shown in Figure 5.5. For the overall execution times, we measured the wall-clock time using software running on the host machine. Execution times included the host machine's pre and post CA processing and the time taken by FPGA engine to

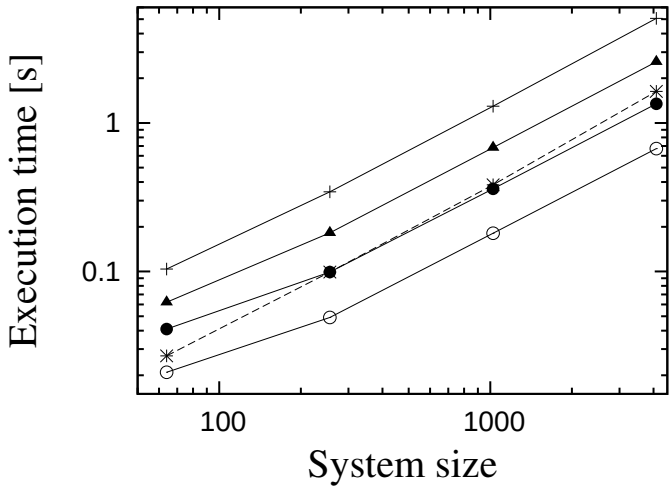


Figure 5.5: *Execution times for computing* 512 *generations of the LBM D2Q9 grid of size N using Fortran and a single FPGA-enabled PC implementations. Solid lines represent FPGA-based results and the dotted line represents Fortran code results. A plus represents a FPGA with 2PEs, a triangle for a FPGA with 4PEs, a filled circle for a FPGA with 8PEs, and an open circle for an FPGA with 16PEs execution times respectively.*

compute the required number of generations. As shown in Figure 5.6, with the increase in the number of PEs the overall execution time decreases proportionally as expected from Equation (3.10). When using all the logic available on the FPGA, that is, for 16 PE's, we achieved a speed-up of 2.3 as compared to the Fortran implementation.

5.4 Conclusion and Future Work

This chapter presented a detailed discussion on a LBM D2Q9 FPGA-based implementation. Based on the computational structure of the chosen test bench its FPGA-based implementation was classified as a compute bound computation algorithm. Several test runs were performed using FPGA-based implementations, followed by comparing its wall-clock measurements to our Fortran implementations. For a single FPGA-based implementation, an overall speed-up of 2.3 as compared to a Fortran implementation was achieved.

The performance model for the single FPGA-enabled D2Q9 LBM implementation implies it to be compute bound as long as the implementation has $p \leq 61$. Therefore, with the current-generation FPGA devices, for example, Virtex-5 FPGA chip would improve the speed-up simply by including more PEs per chip and higher FPGA clock frequency.

Possible future extensions are, to have a design with programmable number of PEs embedded within the implementation. Having programmable number of PE implementations would enable smooth experiments and measurements. A new addition could be

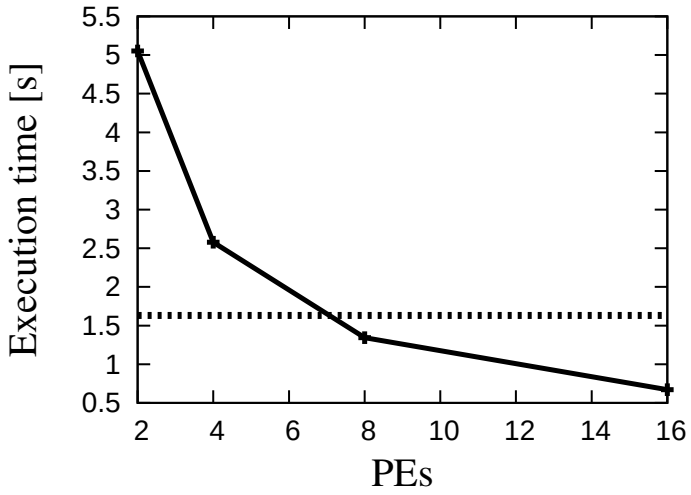


Figure 5.6: *Execution times for computing* 512 *generations of the* 64×64 *LBM D2Q9 grid on a single FPGA-enabled PC implementations. FPGA-based execution times are represented by stars and the dotted line represents the Fortran implementation.*

to share the resources among the PEs. Since each PE internally is a sequential circuit with its own set of floating-point cores for addition, multiplication and division respectively, the sharing of floating-point cores among the PEs is a possibility though not trivial. Compared to 2D LBM model, a 3D LBM implementation would greatly benefit with the given hardware design. Therefore, extending the 2D to 3D LBM implementation is highly recommended.

Chapter 6

CA on Multiple FPGA Enabled PC*

If the logic resources available within the chip limit the number of possible PEs within our implementation, then could we attempt to push this limit outside the chip, that is, have another chip in parallel? FPGAs operate at a clock speed of more than one order of magnitude slower than microprocessors [50]. Will this provide enough room for the exchange of data between the multiple FPGAs running parallel over the host interface? These questions motivated us to port our implementation to a multiple FPGA setups for further performance investigations. In this chapter we look into multiple FPGA enabled PC based CA implementations where the concept of latency hiding [56] is exploited. It also focuses on a dual FPGA enabled PC setup for performance measurements. A dual FPGA enabled PC should provide a realistic view on the feasibility of implementing multiple FPGA based CA accelerators.

*This chapter is based on: S. Murtaza and A.G. Hoekstra and P.M.A. Sloot, 'Performance of floating-point based Cellular Automata Simulations using a dual FPGA system', *submitted*. Initial results based on this chapter's dual FPGA-based implementation were presented at the Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications (Austin, Texas, 2008) and published in [77].

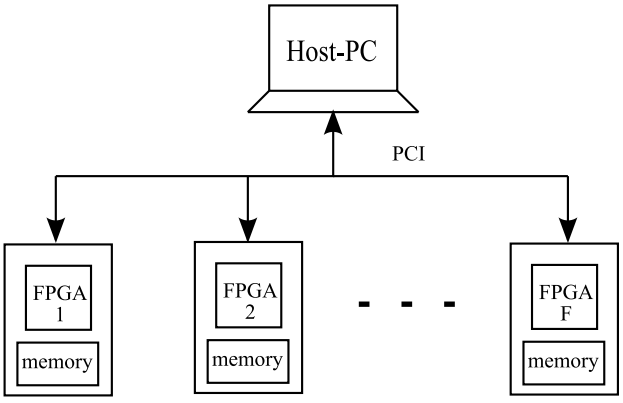


Figure 6.1: *Multiple FPGA enabled PC setup.*

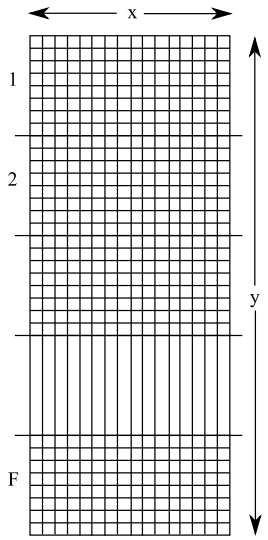


Figure 6.2: *Rectangular lattice (with x columns and y rows) divided along y-axis into F-chunks. Each chunk is processed by a dedicated FPGA based LBM engine.*

6.1 Multiple FPGA Enabled PC

Applying Equation (3.9) to our single FPGA enabled D2Q9 LBM implementation (with system parameters $k = 1/10$, $\tau_c = 675$, and $\tau_r \approx 1.1$, see Section 5.3 for details) guarantees our system to be compute bound as long as the system implementation has $p \leq 61$. FPGA logic resources available on our single FPGA enabled system (see Appendix 10 for hardware details) limited our implementation to a maximum of sixteen PEs (i.e. $p = 16$ LBM cores). With these limitations imposed by the available logic resources, we were able to achieve barely 27% of the theoretical possible p PEs for our LBM hardware accelerator implementation. To port a single FPGA based LBM implementation to a multiple FPGA enabled PC system, required modifications both within hardware and the software cores of our existing implementation. The multiple FPGA enabled PC system is composed of a host PC with multiple FPGA boards connected via PCI interface as shown in Figure 6.1. Each FPGA board includes multiple on-board memory banks as shown in Figure 3.1. For CA implementation, the resulting computing system's PC does all the pre- and post-processing and FPGA boards are connected as coprocessors for CA computations. With this multiple FPGA enabled PC based CA accelerator system, the host machine initially describes the problem to be solved, and downloads all the relevant information (chunk of CA lattice data) to each of the FPGA's on-board source memory bank. Once initialised, the FPGAs start computing, and the host machine keeps track of the overall lattice computations (that is, completion of each iteration and boundary exchange across the FPGA boards) via interrupts from the host accessible control registers available on each of the FPGA.

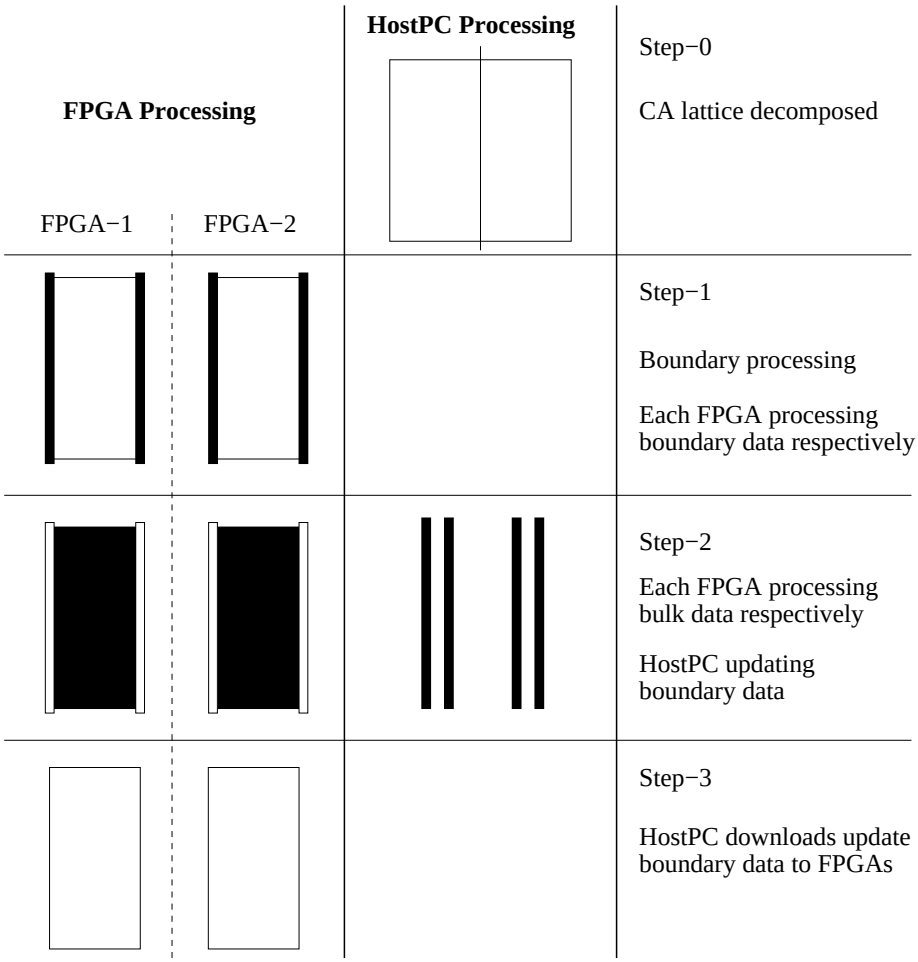


Figure 6.3: *Dual FPGA enabled PC computation model: System initialisation (step-0): host machine decomposes the CA lattice into two halves and downloads each half to the respective FPGA's source memory bank. Boundary processing (step-1): Each FPGA starts with processing boundary data and signals the host via an interrupt. Bulk processing (step-2) FPGA's continue processing bulk data cells and in parallel the host machine updates boundary data across the two FPGA boards. Boundary download (step-3): With the completion of the whole lattice iteration computation, host machine downloads the updated boundary data to each of the FPGA's source memory bank.*

6.1.1 Execution Time

In order to determine the execution time for the compute bound CA computation using multiple FPGA enabled PC, consider the F FPGA enabled PC implementation as shown in the Figure 6.1 and assume the problem domain is a rectangular lattice of size N (with x columns and y rows). The host-PC initially divides the LBM lattice into F-chunks (as shown in Figure 6.2) and downloads all the relevant information ($\frac{N}{F}$ cells) to each of the FPGA module's source memory bank. Once the source memory banks of all of the FPGA modules are loaded, the FPGAs start computing. Computation within the hardware part, that is, processing of $\frac{N}{F}$ cells by each of the FPGA can further be categorised into the following three distinct phases (see Figure 6.3 for dual FPGA enabled PC system implementation).

6.1.1.1 Boundary Processing

Each FPGA's compute engine starts by processing boundary data. With boundary data being only a small part of the overall lattice (when $N \gg 2xF$), the compute engines write out boundary related results to an additional third memory bank (thus mirroring the boundary data results independently) available on their respective boards. Immediately after the completion of processing boundary cells, the compute engine releases the third memory bank for host DMA. For large system sizes especially when $N \gg p$, we can approximate the ceiling functions behaviour and rewrite Equation (3.10) as:

$$T_1 = \tau_r + \left\{\frac{N}{p}\right\}\tau_c + \left\{\frac{p}{k} - 1\right\}\tau_r + \tau_w. \tag{6.1}$$

Assume that all FPGAs start computing simultaneously, with the initial startup time (τ_r), each FPGA starts with processing boundary cells. This combined initial startup time and processing boundary cells denoted by T_b, can be expressed as

$$T_b = \tau_r + \left\{\frac{2x}{p}\right\}\tau_c \tag{6.2}$$

6.1.1.2 Bulk Processing

During this phase of computation the system relies on the effective use of latency hiding. Herbordt et al. [56] highlights latency hiding as a basic technique for achieving high performance in parallel applications, and further recommends this as one of the main design techniques to be exploited in HPC/FPGA applications. Upon completion of boundary data processing, the compute engines continue processing the bulk lattice cells. Inclusion of the third memory bank for mirroring boundary data on each of the FPGA module is assumed to allow latency hiding, that is, the host machine updates the boundary data

across all of the FPGA modules and the bulk data processing by the compute engines are done in parallel. As long as the time to process the bulk data (T_{k}) is larger than the host machine's time to update the boundary data (T_{u}), latency hiding is effective and the overall execution time remains compute bound. Otherwise the host machine processing the boundary data across the FPGA boards over the PCI bus dominates the overall execution time. If τ_{b} is the time to update a boundary cell by the host machine and τ_{d} is the time to download a cell from the host machine to a FPGA or vice versa, execution time for bulk cells computation can be expressed as $\max\{T_{\mathrm{u}}, T_{\mathrm{k}}\}$, where ($T_{\mathrm{k}}$) and ($T_{\mathrm{u}}$) are expressed as:

$$T_{\mathrm{k}} = \frac{1}{p}\left\{\frac{N}{F} - 2x\right\}\tau_{\mathrm{c}} + \left\{\frac{p}{k} - 1\right\}\tau_{\mathrm{r}} + \tau_{\mathrm{w}} \tag{6.3}$$

$$T_{\mathrm{u}} = F\{2x\tau_{\mathrm{d}} + 2x\tau_{\mathrm{b}}\} \tag{6.4}$$

6.1.1.3 Boundary Download

With the completion of a next generation computation, each FPGA interrupts the host machine. Following this the host machine downloads the updated boundary data to each of the FPGA modules source bank. The time to download boundary cells is expressed as:

$$T_{\mathrm{d}} = F\{2x\tau_{\mathrm{d}}\} \tag{6.5}$$

The sum of the above mentioned time durations result in an overall execution time for the multiple FPGA enabled PC implementation

$$T_{\mathrm{f}} = T_{\mathrm{b}} + \max\{T_{\mathrm{u}}, T_{\mathrm{k}}\} + T_{\mathrm{d}}. \tag{6.6}$$

The above three steps are repeated in the computation of every single CA iteration and repeated until the required number of iterations are computed. Once done, the host machine uploads the whole data from each of the FPGA module's destination memory bank for further postprocessing.

6.1.2 Speedup

For multiple FPGA enabled PC implementation, when $T_{\mathrm{k}} \gg T_{\mathrm{u}}$, T_{f} is expressed as

$$T_{\mathrm{f}} = \frac{T_1}{F} + \frac{p}{k}\left\{\frac{F-1}{F}\right\}\tau_{\mathrm{r}} + \left\{\frac{F-1}{F}\right\}\tau_{\mathrm{w}} + 2x\{F\}\tau_{\mathrm{d}} \tag{6.7}$$

and the obtained speedup is

$$S = \frac{T_1}{T_{\mathrm{f}}} = \frac{F}{1 + f_{comm}}, \tag{6.8}$$

where the total fractional communication overhead f_{comm} is the sum of: fractional boundary data downloading overhead (f_b), fractional PE completion overhead (f_{pe}) and fractional writing memory overhead (f_w). Each of the fractional overhead is defined as:

$$f_b = 2x \left\{ \frac{F^2}{T_1} \right\} \tau_{\mathrm{d}} \tag{6.9}$$

$$f_{pe} = \frac{p}{k} \left\{ \frac{F-1}{T_1} \right\} \tau_{\mathrm{r}} \tag{6.10}$$

$$f_w = \left\{ \frac{F-1}{T_1} \right\} \tau_{\mathrm{w}} \tag{6.11}$$

As long as T_1 is big enough, the fractional overheads will be very small and a speedup very close to F may be expected.

When $T_{\mathrm{u}} \gg T_{\mathrm{k}}$, T_{f} is expressed as

$$T_{\mathrm{f}} = \tau_{\mathrm{r}} + \frac{2x}{p}\tau_{\mathrm{c}} + F\left\{2x\tau_{\mathrm{b}} + 4x\tau_{\mathrm{d}}\right\} \tag{6.12}$$

and the obtained speedup is

$$S = \frac{1}{\frac{1}{T_1}\left\{\tau_{\mathrm{r}} + \frac{2x\tau_{\mathrm{c}}}{p} + 2xF\tau_{\mathrm{b}} + 4xF\tau_{\mathrm{d}}\right\}}. \tag{6.13}$$

6.2 Test Cases and Results

In order to validate multiple FPGA enabled PC based two-dimensional CA accelerator implementation, we implemented and validated our model for the dual FPGA enabled PC setup. For porting our single FPGA to the dual FPGA enabled PC implementation, two of the FPGAs available on Maxwell- a 64 FPGA-based supercomputer (see Appendix 10 for setup details) were used and benchmarked for varying number of square lattice sizes. For square lattice (that is, $x = \sqrt{N}$) CA simulations and dual FPGA enabled PC setup (that is, $F = 2$), the multiple FPGA enabled PC implementation performance model, as specified in the previous section, simplifies as follows

$$T_{\mathrm{b}} = \tau_{\mathrm{r}} + \left\{ \frac{2\sqrt{N}}{p} \right\} \tau_{\mathrm{c}} \tag{6.14}$$

$$T_{\mathrm{k}} = \frac{1}{p}\left\{\frac{N}{2} - 2\sqrt{N}\right\}\tau_{\mathrm{c}} + \left\{\frac{p}{k} - 1\right\}\tau_{\mathrm{r}} + \tau_{\mathrm{w}} \tag{6.15}$$

$$T_{\mathrm{u}} = 2\left\{2\sqrt{N}\tau_{\mathrm{d}} + 2\sqrt{N}\tau_{\mathrm{b}}\right\} \tag{6.16}$$

$$T_{\mathrm{d}} \;=\; 2\left\{2\sqrt{N}\tau_{\mathrm{d}}\right\} \tag{6.17}$$

The overall execution time for the dual FPGA enabled implementation is the sum of the above mentioned time durations and is expressed as:

$$T_2 \;=\; T_{\mathrm{b}} + \max(T_{\mathrm{u}}, T_{\mathrm{k}}) + T_{\mathrm{d}}. \tag{6.18}$$

6.2.1 Minimum Required System Size

For square lattice dual FPGA based implementation, the minimum system size (N^*) required for latency hiding to work successfully is determined when T_{u} and T_{k} are equal.

$$4\sqrt{N}\left\{\tau_{\mathrm{d}} + \tau_{\mathrm{b}}\right\} \;=\; \frac{1}{p}\left\{\frac{N}{2} - 4\sqrt{N}\right\}\tau_{\mathrm{c}} + \left\{\frac{p}{k} - 1\right\}\tau_{\mathrm{r}} + \tau_{\mathrm{w}}$$

$$N^* \;=\; \left\{4 + \frac{4p\tau_{\mathrm{d}}}{\tau_{\mathrm{c}}} + \frac{4p\tau_{\mathrm{b}}}{\tau_{\mathrm{c}}} + \frac{p}{\tau_{\mathrm{c}}}\sqrt{16\left\{\tau_{\mathrm{d}} + \tau_{\mathrm{b}} + \frac{\tau_{\mathrm{c}}}{p}\right\}^2 - \frac{2\tau_{\mathrm{c}}\tau_{\mathrm{r}}}{k} + \frac{2\tau_{\mathrm{c}}\tau_{\mathrm{r}}}{p} - \frac{2\tau_{\mathrm{c}}\tau_{\mathrm{w}}}{p}}\right\}^2$$

6.2.2 Speedup

For dual FPGA enabled PC implementation, when $T_{\mathrm{k}} \gg T_{\mathrm{u}}$, T_2 is expressed as

$$T_2 \;=\; \frac{T_1}{2} + \left\{\frac{p}{2k}\right\}\tau_{\mathrm{r}} + \left\{4\sqrt{N}\right\}\tau_{\mathrm{d}} + \left\{\frac{1}{2}\right\}\tau_{\mathrm{w}} \tag{6.19}$$

and the obtained speedup is

$$S = \frac{2}{1 + f_{comm}}, \tag{6.20}$$

where fractional overheads f_b, f_{pe} and f_w are defined as:

$$f_b = \left\{\frac{8\sqrt{N}}{T_1}\right\}\tau_{\mathrm{d}} \tag{6.21}$$

$$f_{pe} = \left\{\frac{p}{kT_1}\right\}\tau_{\mathrm{r}} \tag{6.22}$$

$$f_w = \left\{\frac{1}{T_1}\right\}\tau_{\mathrm{w}} \tag{6.23}$$

And as long as T_1 is big enough, the fractional overheads will be very small and a speedup very close to *two* may be expected.

When $T_{\mathrm{u}} \gg T_{\mathrm{k}}$, T_2 is expressed as

$$T_2 = \tau_{\mathrm{r}} + \left\{ \frac{2\sqrt{N}}{p} \right\} \tau_{\mathrm{c}} + F \left\{ 2\sqrt{N} \right\} \tau_{\mathrm{b}} + F \left\{ 4\sqrt{N} \right\} \tau_{\mathrm{d}} \tag{6.24}$$

and the obtained speedup is

$$S = \frac{1}{\frac{1}{T_1} \left\{ \tau_{\mathrm{r}} + \frac{2\sqrt{N}\tau_{\mathrm{c}}}{p} + 2\sqrt{N}F\tau_{\mathrm{b}} + 4\sqrt{N}F\tau_{\mathrm{d}} \right\}}. \tag{6.25}$$

From Equation (6.25), in combination with the fact that in this case $N < N^*$, we can conclude that the performance will increasingly deteriorate by the fact that the execution is communication bound, and the speedup will be much smaller than *two*. One can even expect a speed down, that is, and execution time being larger as compared to the single FPGA enabled execution.

6.2.3 System Parameters

Several test case runs were performed to determine system parameter τ_{d}, that is, the time to upload an LBM cell (each cell includes ten 64-bit words where nine represent the nine velocities of a cell and an additional one for future system implementations to include complex boundary computations) from host-PC to the DDR2-SDRAM, available on the attached FPGA board or vice versa. Using the software process running on the host-PC, varying number of data packets (ranging from 16KB to 2MB) were first downloaded to the on-board DDR2-SRAM bank followed by uploading them back to the host memory. The round trip timing were measured as shown in Figure 6.6. The average bandwidth calculated from the experiment was 33MB/sec and accordingly used to define $\tau_{\mathrm{d}} = 2.3\mu s$. A number of test case runs for updating 2048 LBM cells were performed to determine τ_{b} time to update an LBM boundary cell, and the average time recorded was $13ns$.

6.2.4 Results

For the dual FPGA enabled PC system, we ported our single FPGA-based implementations with one, two, four, and eight PEs respectively to one of the Maxwell's available nodes (each node includes two FPGAs attached via PCI interface). These implementations were tested for five different square lattice sizes (N equal to 32^2, 64^2, 128^2, 256^2 and 512^2) that were computed for ($g = 256$) number of iterations. The resulting execution times for the corresponding single and the dual FPGA based implementations cases are shown in Figure 6.4. The corresponding speedup is shown in Figure 6.5.

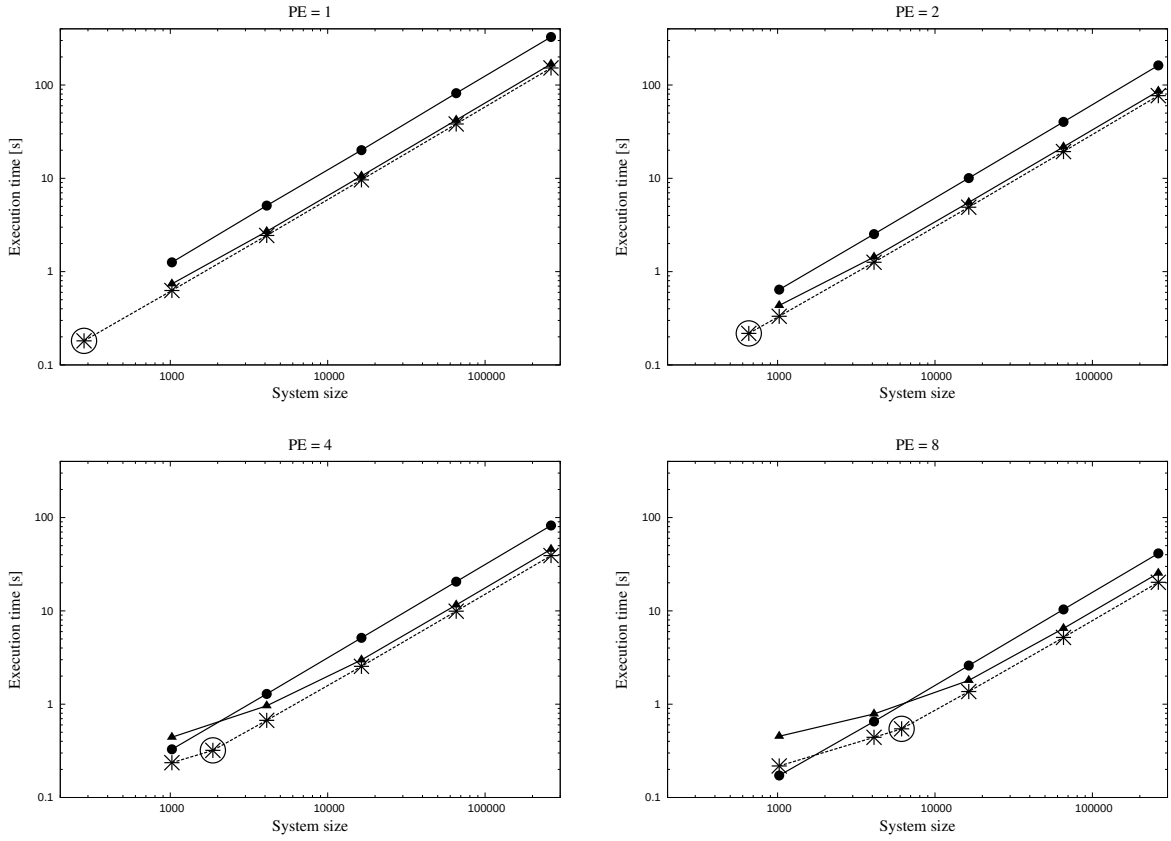


Figure 6.4: *Execution times for computing* 256 *iterations of the square domain LBM D2Q9 grid of size N using a single and dual FPGA enabled system implementation. Broken line represents performance model Equation (6.19) for the dual FPGA enabled execution. Filled circles represent the measured execution times for a single FPGA enabled system implementation, and filled triangles represent a dual FPGA enabled system implementation with (a) 1PE, (b) 2PE, (c) 4PE, and (d) 8PE implementation respectively. Big circles specified on the broken line highlight the minimum system size required for latency hiding to work and for the said implementation to be compute bound.*

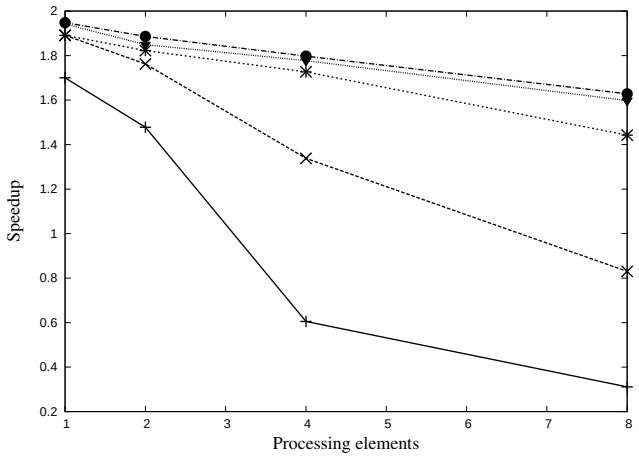


Figure 6.5: *Speedup achieved using dual- over single-FPGA based implementations. Speedup measurements are based on execution times for computing* 256 *iterations of varying square domain LBM D2Q9 system sizes for varying number of PE implementations, using a single and dual FPGA enabled PC implementations respectively. Pluses represent* 32^2*, crosses* 64^2*, stars* 128^2*, triangles* 256^2*, and circles* 512^2 *system sizes respectively.*

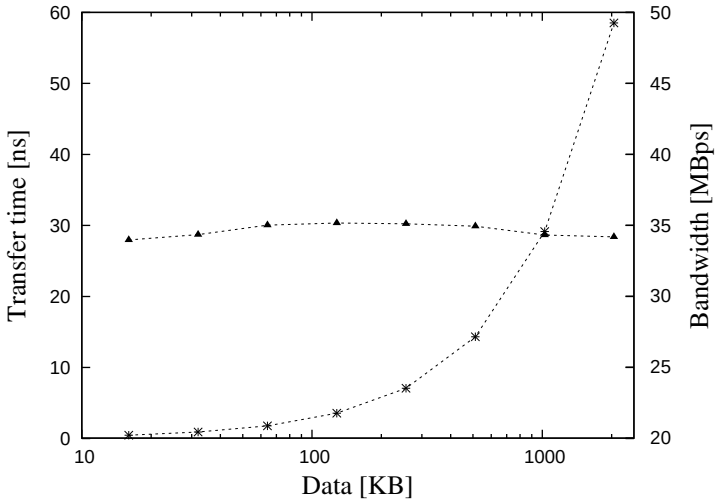


Figure 6.6: *System parameter(τ_{d}): Stars represent the time to upload the data packets (ranging from 16KB to 2MB) from the DDR2 SDRAM available on the FPGA board to the host-PCs memory over the PCI interface or vice versa. Average of the calculated bandwidth from the test runs, as shown by triangles, was used to define τ_{d}, that is, the time to upload (or download) an LBM cell from host-PC to the memory banks available on the attached FPGA board.*

With an increasing number of PEs the fractional PE completion overhead (f_{pe}) as shown in Equation (6.10) increases and the overall speedup goes down. And with increase in system size N, the fractional boundary data downloading overhead (f_b) as shown in Equation (6.9) goes down and the overall speedup goes up. All of the three fractional overheads f_b, f_{pe}, and f_w are always larger than zero and overall result in the loss of speedup. However, f_b can be minimised with the inclusion of an extra memory bank for downloading of updated boundary data by the host machine to each of the FPGA board. Moreover, the difference between compute bound speedup (that is, for larger system sizes) and communication bound speedup (or even speed down) is clearly visible due to the fact that latency hiding no longer is effective, for the smaller system sizes.

6.3 Conclusion and Future Work

This chapter presented a detailed discussion on porting a single- to a dual-FPGA based LBM D2Q9 implementation. Based on the single FPGA enabled model as specified in the previous chapters, a model to evaluate the performance of a two dimensional CA executed on multiple FPGA enabled PC system was presented. Further the model was validated for a dual FPGA based setup for the square domain LBM D2Q9 implementations. Results from the dual FPGA based implementations demonstrate how the included latency hiding techniques were a success and the overall execution time was decreased by a factor close to *two*. A working latency hiding also demonstrates what [56] has identified as one of the important HPC/FPGA application design techniques.

Chapter 7

CA on FPGA Enabled PC Cluster

A multiple FPGA enabled PC setup, as introduced in the previous chapter, demonstrates a master-slave system configuration where the host PC controls the allocation of job to each of the attached FPGA via a PCI interface. The proposed performance model was implemented and validated for a dual FPGA enabled PC setup. Other than performing the pre- and postprocessing of the CA lattice (which happens to be the only operation performed by the host-PC for a single FPGA based implementation as presented in the Chapter 3), the host-PC also updates the boundary data across the multiple FPGAs over the PCI bus for every single CA iteration. With this master-slave setup, boundary updates across multiple FPGAs by the host-PC is purely a sequential process and with the increase in the number of attached FPGAs, the inter FPGA communication via host-PC ends up as the bottleneck for the overall performance of the CA implementation. As a result, the host-PC time to update the boundary data increases with the increase in the number of FPGAs connected. The host-PC not only takes longer for boundary updates but the FPGAs might also end up waiting for the boundary update completion.

To get around this master-slave setup situation, a fully parallel FPGA based platform is desirable. In this chapter we look into using Maxwell - a 64 FPGA supercomputer [8] for our CA implementations. In addition we discuss in detail a performance model and the results from the test case runs.

7.1 FPGA Enabled PC Cluster

Maxwell - a 64 FPGA supercomputer (see Appendix 10 for details) is a PC cluster where each PC node is a dual-core machine and includes two separate FPGA cards connected via a PCI interface. For the fully parallelised multiple FPGA based CA implementation, the rectangular problem domain is decomposed along the longest dimension into F equal sized chunks (1-D domain decomposition) along the y-axis and distributed to F FPGA LBM engines for CA computations. PTK [9]- a thin software layer along with the standard

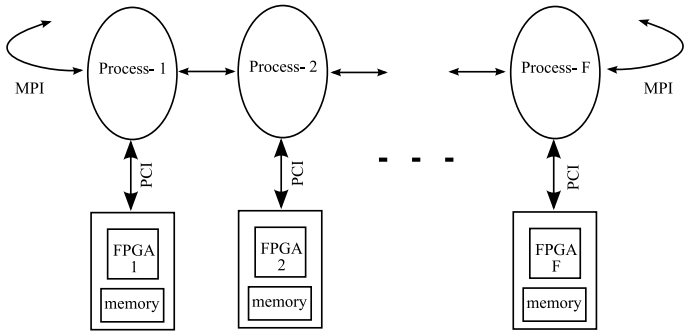


Figure 7.1: *Logical representation of the FPGA enabled PC cluster setup using FHPCA Parallel toolkit [9].*

MPI library allows Maxwell's FPGAs setup in a ring topology as shown in Figure 7.1 and therefore a one-to-one mapping of F-chunk CA lattice onto a F FPGA ring. Using PTK, one can configure the required number of Maxwell's nodes, where each node is defined as a software process running on the host PC together with a FPGA hardware. And the inter-process (that abstracts inter-FPGA communication) communication is performed over the cluster's Ethernet network using the MPI library.

7.1.1 Execution Time

For the FPGA enabled PC cluster based implementation, consider a rectangular LBM lattice of size N cells (x-columns and y-rows) as shown in Figure 6.2. The master-node (one of the nodes in a cluster) initially performs the 1-D decomposition of the the LBM lattice along the y-axis into F chunks (where F is the number of available FPGA nodes) and downloads all the relevant information ($\frac{N}{F}$ cells) to each of the FPGA module's source memory bank. Once the source memory banks of the FPGA modules are loaded, the FPGAs start computing.

Computation within the hardware part, that is, processing of $\frac{N}{F}$ cells by each of the FPGA can be further categorised into the following three distinct phases which are largely similar to those described for multiple FPGA enabled PC setup in the previous chapter.

7.1.1.1 Boundary Processing

The boundary processing phase is the same as with the multiple FPGA enabled PC setup. Each FPGA starts with processing boundary data and writes out the results to an additional third memory bank available on their respective boards. Once done with the boundary data processing, the third memory bank is released immediately by the FPGA for host DMA. Assuming all the FPGA boards start computing simultaneously, with the

initial startup time (τ_r) each board starts with processing boundary cells. This combined initial startup time and processing boundary cells denoted by T_b, can be expressed as

$$T_b = \tau_r + \left\{ \frac{2x}{p} \right\} \tau_c \tag{7.1}$$

7.1.1.2 Bulk Processing

Upon completion of the boundary data processing, the compute engines continue processing the bulk lattice cells. Inclusion of the third memory bank for mirroring boundary data on each of the FPGA module is assumed to allow latency hiding, that is, the host machine updates the boundary data across the three FPGA modules (one attached to itself, to the right and to the left neighbour) and the bulk data processing by the compute engines is done in parallel. To update the boundary data across the multiple FPGA boards (processes own FPGA, left and the right neighbour processes FPGA respectively) each host process performs three steps; a) sends out the right boundary column to its right neighbouring process (and as a result it also receives a boundary column from its left neighbouring process), b) updates the two boundary columns and c) sends out the left column (now updated) to its left neighbouring process (again as a result receives one from its right neighbour). As long as the time to process the bulk data (T_k) is larger than the host processes time to update the boundary data (T_u), latency hiding is effective and the overall execution time remains compute bound. Otherwise the host node processing the boundary data across its neighbouring nodes over the PCI bus and the Ethernet dominates the overall execution time. If τ_m is the time to send (or receive) a cell from one host process to the other over Ethernet network using MPI library, execution time for bulk cells computation can be expressed as $\max\{T_u, T_k\}$, where (T_k) and (T_u) are expressed as:

$$T_k = \frac{1}{p} \left\{ \frac{N}{F} - 2x \right\} \tau_c + \left\{ \frac{p}{k} - 1 \right\} \tau_r + \tau_w \tag{7.2}$$

$$T_u = 2x \{ \tau_d + \tau_b + \tau_m \} \tag{7.3}$$

7.1.1.3 Boundary Download

With the completion of a next generation computation, each of the FPGA interrupts the host process. Next the host process downloads the updated boundary data to its connected FPGA modules source bank. The time to download boundary cells (T_d) is $2x\tau_d$.

The sum of the above mentioned time durations results in an overall execution time for the FPGA enabled PC cluster implementation.

$$T_{\mathrm{F}} \;=\; T_{\mathrm{b}} + \max\left\{T_{\mathrm{u}}, T_{\mathrm{k}}\right\} + T_{\mathrm{d}}. \tag{7.4}$$

The above three steps are repeated in computation of every single generation and repeated until the required number of generations are computed. Once done, each of the host process uploads the data from its attached FPGA modules destination memory banks and finally, the master-node collects and combines all the data chunks together for further postprocessing.

7.1.2 Speedup

For $T_{\mathrm{k}} \gg T_{\mathrm{u}}$, T_{F} is expressed as

$$T_{\mathrm{F}} \;=\; \frac{T_1}{F} + \frac{p}{k}\left\{\frac{F-1}{F}\right\}\tau_{\mathrm{r}} + \left\{\frac{F-1}{F}\right\}\tau_{\mathrm{w}} + 2x\tau_{\mathrm{d}} \tag{7.5}$$

and the obtained speedup using FPGA enabled PC cluster implementation is

$$S = \frac{F}{1 + f_{comm}}, \tag{7.6}$$

where total fractional communication overhead f_{comm} is the sum of: fractional boundary data downloading overhead (f_b), fractional PE completion overhead (f_{pe}), and fractional writing memory overhead (f_w). Each of the fractional overhead is defined as:

$$f_b = 2x\left\{\frac{F}{T_1}\right\}\tau_{\mathrm{d}} \tag{7.7}$$

$$f_{pe} = \frac{p}{k}\left\{\frac{F-1}{T_1}\right\}\tau_{\mathrm{r}} \tag{7.8}$$

$$f_w = \left\{\frac{F-1}{T_1}\right\}\tau_{\mathrm{w}} \tag{7.9}$$

As long as T_1 is big enough, the fractional overheads will be very small and a speedup very close to F may be expected.
For $T_{\mathrm{u}} \gg T_{\mathrm{k}}$, T_{F} is expressed as

$$T_{\mathrm{F}} \;=\; \tau_{\mathrm{r}} + \frac{2x}{p}\tau_{\mathrm{c}} + 2x\left\{\tau_{\mathrm{b}} + \tau_{\mathrm{d}} + \tau_{\mathrm{m}}\right\} \tag{7.10}$$

and the obtained speedup is

$$S = \frac{1}{\frac{1}{T_1}\left\{\tau_{\mathrm{r}} + \frac{2x}{p}\tau_{\mathrm{c}} + 2x\tau_{\mathrm{b}} + 2x\tau_{\mathrm{d}} + 2x\tau_{\mathrm{m}}\right\}}. \tag{7.11}$$

7.2 Details of the Test Case

In order to validate our performance model for FPGA enabled PC cluster based CA implementations, rectangular domains with periodic boundary conditions of varying sizes were used as test cases. For each of the test case, the number of PEs per FPGA was fixed to $p = 8$ as these were the maximum number of PEs we could fit into an FPGA chip available on Maxwell (see Appendix 10 for details). The master node determines the problem domain and performs the 1-D domain decomposition of the CA lattice along the y-axis into F equal sized chunks, where F is the number of host processes (or FPGAs, as each host process includes one FPGA for acceleration) used for the test runs. The CA lattice once prepared, the master node setups up the required number of processes (that is FPGA nodes) in a ring topology and uses $MPI\_Scatter$ to distribute the lattice chunks to each of the FPGA nodes. Using the ring topology and the 1-D domain decomposition along the x-axis ensures, that the each computing node has the required boundary data along the y-axis. However, boundary data along the x-axis needs to be exchanged among the neighbouring computing nodes. Once the lattice is distributed each of the host process loads the available source memory bank on its attached FPGA board with the given chunk of lattice. Table 7.1 lists all the system sizes used as the test cases. Each system size was computed for 10-iterations using a varying number of FPGA nodes. During each single iteration, host processes use $MPI\_SendRecv$ to exchange and update boundary across their respective left and the right neighbouring nodes. With the number of specified iterations computed, master node uses MPI_Gather to collect the updated lattice chunks from each of the host process for further postprocessing. Note, the execution times presented in the following sections relate to the part between the first $MPI\_Scatter$ and last MPI_Gather and therefore, do not include the system setup time, that is, loading FPGA boards and lattice scatter/gather related operations.

$N \backslash F$	2 FPGAs	4 FPGAs	8 FPGAs	16 FPGAs	20 FPGAs	25 FPGAs
1000*1000	500*1000	250*1000	125*1000	-	-	-
1200*1000	600*1000	300*1000	150*1000	-	-	-
1600*1000	800*1000	400*1000	200*1000	-	-	-
1800*1000	900*1000	450*1000	225*1000	-	-	-
2000*1000	1000*1000	500*1000	250*1000	125*1000	100*1000	80*1000

Table 7.1: *LBM system sizes N used as test cases for the F FPGA enabled PC cluster implementation. Right column represents the total system sizes used as a test case and rest of the columns show the lattice chunk size for the specified number of FPGA enabled setup.*

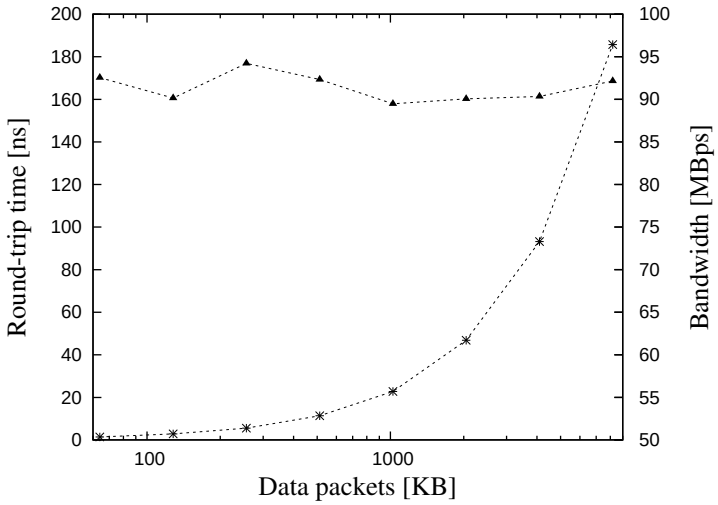


Figure 7.2: *System parameter(τ_{m}): Stars represent the round trip times for the data packets (ranging from 64KB to 8MB) over the gigabit Ethernet from one PC to another using MPI-based ping-pong test runs. The average of the calculated bandwidth from the test runs, as shown by triangles, was used to define τ_{m}, that is, the time to send an LBM cell from one PC to another over Ethernet using MPI routines.*

7.2.1 System Parameters

Several test case runs were performed using MPI Send-Receive routines to determine system parameter τ_{m}, that is, the time to send (or receive) a LBM cell (each cell includes ten 64-bit words, where nine represent the nine velocities of a cell and an additional one for future system implementations to include complex boundary computations) from host process to the another host process over gigabit Ethernet. Data packets of varying sizes, ranging from 64KB to 8MB, were used. The round trip times were measured as shown in Figure 7.2. The average bandwidth calculated from the experiment was 91MB/sec and accordingly used to define $\tau_{\mathrm{m}} = 835ns$.

7.3 Performance Results

The subsequent sections discuss the test case results performed for the varying number of system sizes (as specified in the Table 7.1) and number of FPGA nodes.

7.3.1 Two Million Cells Lattice

Figure 7.4 shows the execution times for a two million cells 2D LBM lattice with periodic boundary conditions computed for 10 iterations. This system size was also the largest lattice used as a test case (cluster configuration on the Maxwell machine imposed this

system size limitations). The test case was run for $F = \{1, 2, 4, 8, 16, 20, 25\}$ number of FPGAs. Dividing the given system size into equal sized chunks limited the maximum number of FPGAs to 25 though the total number of available FPGAs on Maxwell is 32 (see Appendex 10). With the specified set of parameters, latency hiding work as per the model, that is, Equation (7.5) and as represented by a broken line in Figure 7.4. The results from our implementation, as represented by stars, closely follow the given performance model as shown in the Figure 7.4. And as expected with the increase in the number of FPGAs the execution time also goes down. Figure 7.5 shows the speedup for the same system configuration. However, with the increase in FPGAs especially after $F > 8$, the measured speedups deviate away from the model predictions. We performed system profiling to further investigate our model and measure the relative errors. During each system iteration computation, each host process along with its attached FPGA accelerator computes a lattice chunk in three distinct phases, as explained in previous sections, where each stage progress is updated across process and its accelerator via interrupts. Using software timers we explicitly monitored these three phases of the computation as shown in Figure 7.3. Time to compute and download the boundary data is around $10ms$ which is negligible compared to the bulk cells computation time, that is, in the order of hundreds of milliseconds as shown in Figure 7.3. Therefore, relative error(s) with the boundary cells compute plus download times are hardly noticeable in the overall execution time. Please note, our performance model is based on the FPGA execution times and the profiling measurements use the software timers within the host processes. It is not surprising to see the errors with boundary compute time which are in the order of a millisecond from FPGA perspective but, registered by host processes operating system within few milliseconds. However, the main contribution to the overall execution time is the bulk computation. As shown in Figure 7.3 (top left), implementation measurements (shown as stars) closely follows the model represented by the broken line. The relative error with the bulk compute time is around seven percent which is consistent as reported in our single FPGA based implementation in Chapter 5. We are aware of this error, and it is work-in-progress. In future we plan to accommodate these relative errors in our improved performance model where system profiling would include: time to switch the roles of the attached source and destination memory banks performed after every iteration, and the time to write out the next state results to the destination memory banks, that is, τ_{w} (the current model assumes it to be equal to τ_{r}).

7.3.2 Other System Size

The other system sizes as specified in the Table 7.1 were also used as test cases and all were computed for 10 iterations using $F = \{1, 2, 4, 8\}$ number of FPGAs. Figure 7.6

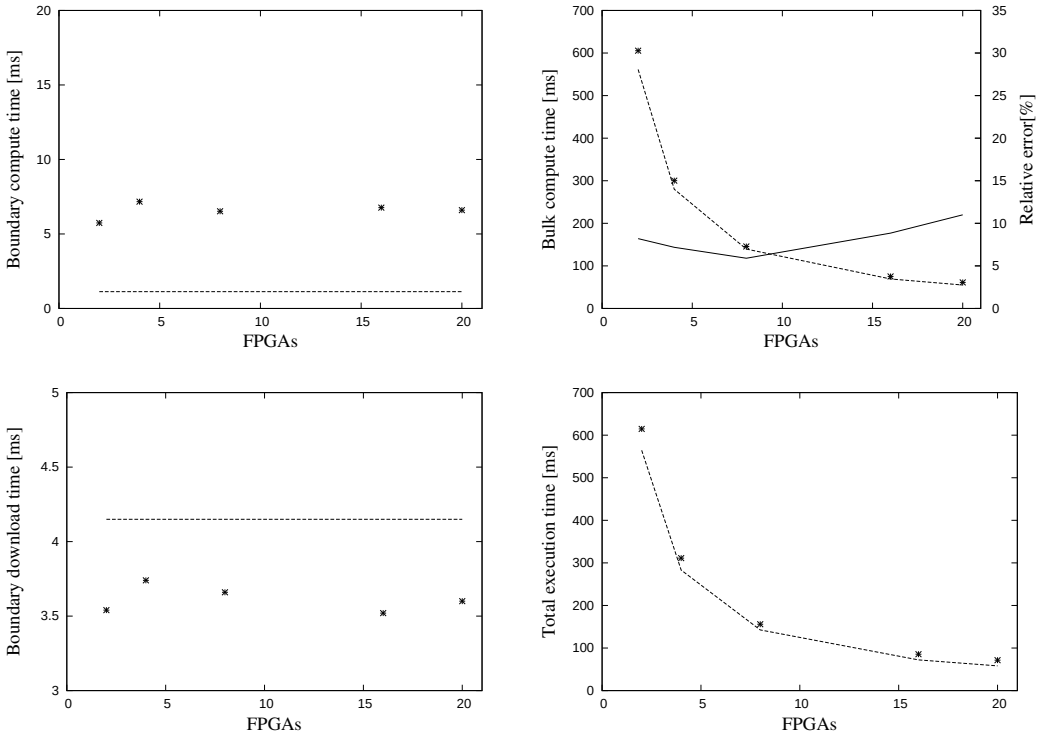


Figure 7.3: *Profiling three stages of a single iteration computation: 2 million cells system size computed for a single iteration using specified number of FPGAs where broken line represents the model predictions and stars stand for the measurements. Each iteration starts with FPGA computing boundary cells (shown in top left figure), followed by computing bulk cells (shown in top right figure) and in parallel host process update boundary cells (successful latency hiding assumes time to compute bulk cells by a FPGA to be greater than the boundary update time by the host process). And once the FPGA is done with computing the lattice chunk, the host process downloads the update boundary cells to the respective FPGA's on-board memory bank (shown in lower left figure). The over all execution time is shown in lower right figure. Bulk cells computation as shown in top right figure also shows the relative error represented by a solid line.*

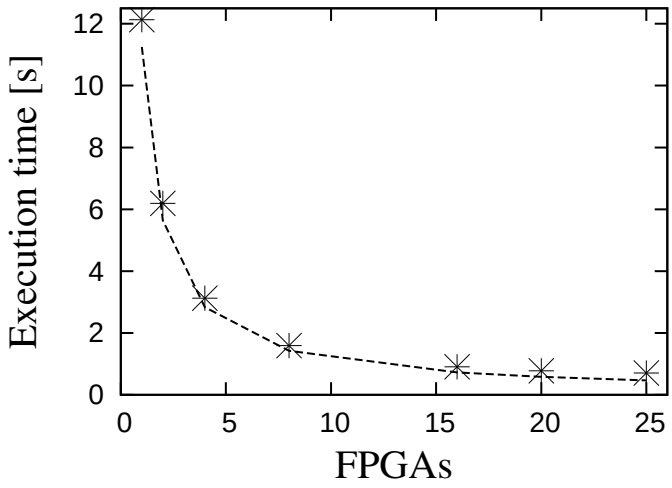


Figure 7.4: *Execution time as a function of FPGAs. Measurements are from the LBM lattice of size 1000x2000 cells computed for 10 iterations where the broken line represents the model and stars represent the measurements.*

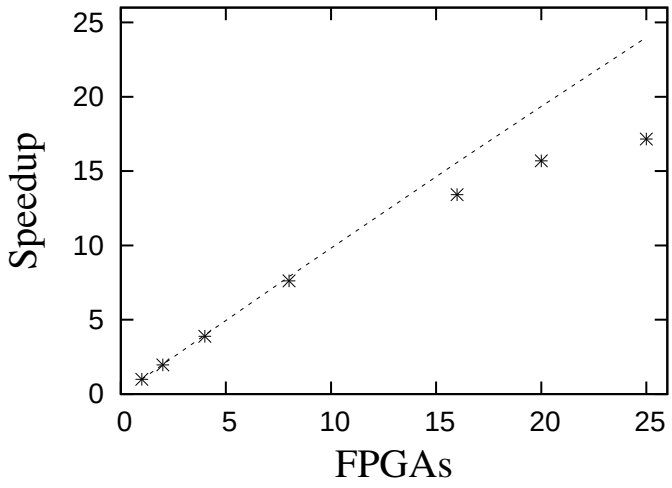


Figure 7.5: *Speedups as a function of FPGAs for a 1000x2000 cells system size. Broken line represents the performance model as specified in Equation (7.6) and stars show the implementation measurements.*

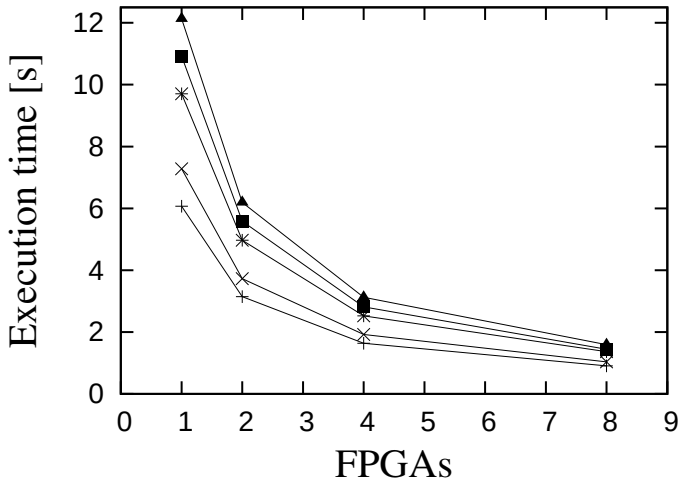


Figure 7.6: *Execution time as a function of FPGA. Each curve represents the execution times for computing a specified system size (triangle for 2-million, square for 1.8-million, star for 1.6-million, cross for 1.2-million and plus for 1-million LBM cells respectively) for 10 iterations for varying number of FPGAs.*

and Figure 7.7 show the execution times as a function of FPGAs and the system sizes respectively. As expected with the increase in the FPGAs, the execution time goes down and with increase in the system size the execution time goes up. Speedup and the efficiency graphs as shown in Figure 7.8 clearly show how the overall system performance improves and converges to the model with an increase in the lattice system size.

7.4 Conclusion and Future Work

As presented in Chapter 5, we started with a single FPGA based implementation, where the design and implementation is more focused towards manipulating the FPGA fabric to perform bit-level operations. On the other side, this chapter demonstrates, how a single FPGA based LBM compute core with additional features and modification, is used in the parallel multiple FPGA system, where the focus of the overall system design is on the transfer and manipulation of chunks of data in the order of mega bytes. The chapter also presented a performance model for a FPGA enabled PC cluster based 2D CA accelerator, with periodic boundary conditions. The system implementation was validated for a number of system sizes, with 2-million cells 2D LBM grid being the largest. Performance model shows how the FPGA enabled PC cluster is the preferred multiple FPGA organisation over the multiple FPGA based PC setup. Latency hiding as suggested by [56] has been the premise through out our multiple FPGA based systems, and was fully exploited for PC cluster based system as well. To validate the successful implementation of

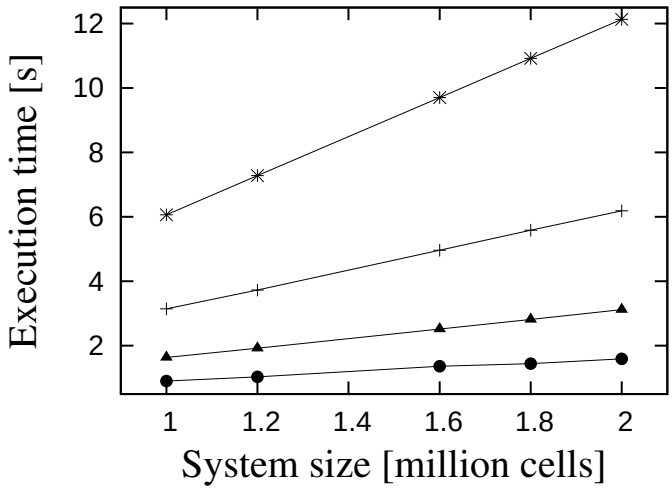


Figure 7.7: *Execution time as a function of system size. Each line represents the execution times for computing the specified system sizes for 10 iterations. Stars represent one, pluses represent two, triangles represent four, and circles represent eight FPGA implementation setups respectively.*

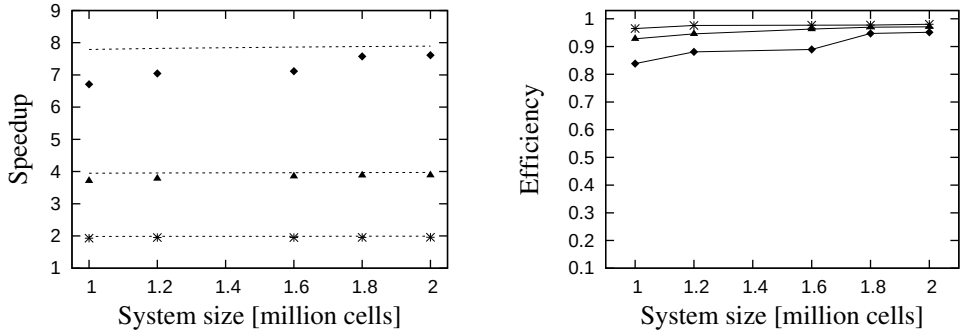


Figure 7.8: *Speedups and the efficiencies as a function of system size. Stars represent two, triangles represent four, and squares represent eight FPGA based implementations respectively. In the speedup figure (towards left), broken line represents the model.*

using latency hiding technique, and to track the interrupt driven interactions between the host processes and the attached FPGAs, a system profiling was performed for every single CA iteration computation using software timers. Firstly, latency hiding being in sync with the performance model and secondly, the time to update and download boundary data by the host processes (over the PCI interface and the Ethernet/MPI network) being negligible compared to the time for computing bulk cells, indicates a two dimensional decomposition system investigation.

Chapter 8

Performance Predictions of CA Computations on Advanced FPGA Computing Resources

Performance modeling and evaluation of FPGA based CA accelerators have been the main focus of this research, with real number based lattice Boltzmann computations as the prime candidate. Real number based computations are not only FPGA logic resource hungry but also place constraints on the data transfer interface between the FPGA and the external components like memory banks. Confining such a resource hungry application within the available FPGA resources is always a challenge. Starting from a single to multiple FPGA based implementations, previous chapters have demonstrated the successful implementations of our proposed performance model. Being aware of the strengths and weaknesses of our model, in this chapter, we take the liberty of using the model to predict CA implementations using advanced computing resources and CA algorithms. In the absence of advanced computing resources, pencil and paper based computations are always an option. This chapter focuses on performance predictions for advanced computing resources and CA algorithms.

8.1 FPGA with 61 PEs

A single FPGA based accelerator as presented in Chapter 5, is either IO bound or compute bound for a given set of parameters as specified by Equation (3.9). And these values are defined by the available FPGA device specification. For example, updating a single LBM cell PE requires 674 FPGA core clock-cycles, a number of cells read by the FPGA in parallel from the attached on-board memory bank $k=\frac{1}{10}$, on an average the FPGA takes 1.1 core clock-cycles to read a single word (64-bits) from a source memory bank, and design implementations on a FPGA board with memory banks run at 150MHZ and

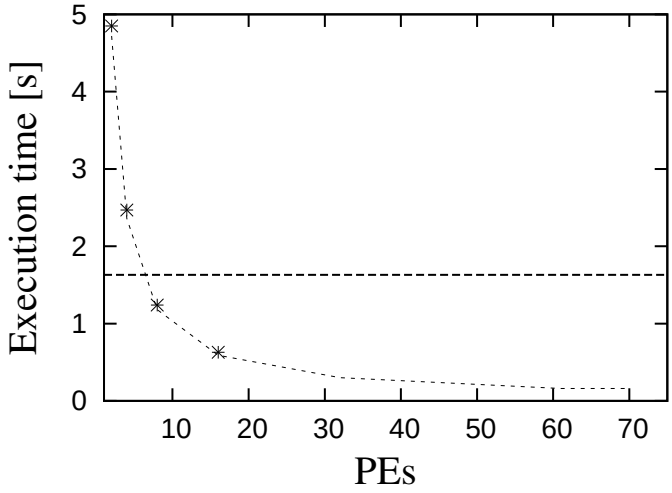


Figure 8.1: *Single FPGA based performance prediction for computing 512 iterations of the 64x64 LBM D2Q9 grid. Stars represent the measurements similar to as shown in Figure 5.6 and curve is the model. Broken line represents the Fortran implementation.*

180MHz respectively. For these values, the accelerator stays compute bound up to a maximum of 61 PEs, however, the available logic using the FX140 FPGA only allowed a maximum of 16 PE implementation. In this section we look into the performance of FPGA based D2Q9 LBM implementations using the maximum number of PEs, that is, 61.

8.1.1 Single FPGA based D2Q9 LBM Implementation

Similar to Figure 5.6, a performance prediction for a single FPGA based implementation, extended to a maximum possible of 61 PE implementations is shown in Figure 8.1. FX140 chip resources allowed a maximum of 16 PE implementation which achieved a speedup of 2.3 as compared with the Fortran implementation. If the chip resources could accommodate 61 PE implementation, the performance would further improve by a factor close to four and also by an order of speedup as compared to Fortran implementation. Beyond 61 PEs, implementation is no longer compute bound and therefore execution time stays constant with overall decrease in the efficiency.

As shown in Figure 8.1, a 64x64 D2Q9 lattice computed for 512 iterations using a 61 PEs implementation, reaches a theoretical peak performance of 13.10 MLUPS (million lattice-site updates per second).

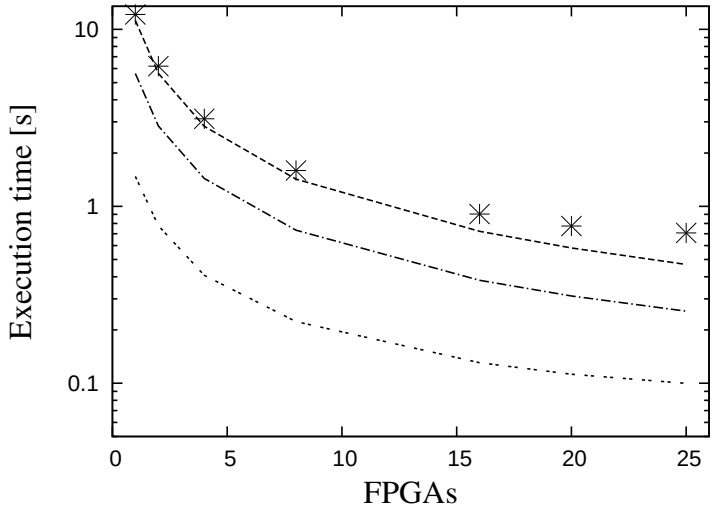


Figure 8.2: *FPGA enabled PC cluster based performance prediction for computing 10 iterations of the 1000x2000 D2Q9 LBM grid. Top most broken curve represents the model and the stars are the measurements as shown in Figure 7.4 for an 8PE per FPGA implementation. The middle and bottom curves are the model for a 16PE and 61PE per FPGA implementation respectively.*

8.1.2 FPGA enabled PC Cluster Based D2Q9 LBM Implementation

Multiple FPGA based implementations were run using the Maxwell machine which includes FX100 FPGAs. Since the available logic resources on FX100 are less when compared to FX140 FPGA, therefore, implementations tested on Maxwell allowed a maximum of 8 PEs per FPGA. Had Maxwell's FPGAs been FX140, then a 16 PE per FPGA implementation would have been possible (same as our single FPGA based implementations using FX140, that allowed a maximum of 16 PE implementation), and a theoretical maximum of 61 PEs per FPGA using advanced chips of the future. Based on the three specified flavors of FPGA hardware, performance for the multiple FPGA enabled PC cluster implementation as a function of FPGAs for a given system size of 1000x2000 cells is as shown in Figure 8.2: top most curve represents the 8 PEs per FX100 FPGA implementation; middle curve represents the 16 PE per FX140 FPGA implementation that would improve the performance by a factor of 2; and the bottom curve represents the theoretical maximum of 61 PEs per FPGA that would further improve the overall system performance by a factor of 4.7 for a 25 FPGAs based setup.

As shown in Figure 8.2, a 2 million D2Q9 lattice computed for 10 iterations, using 25 FPGA implementation each with 61 PEs, reaches a theoretical peak performance of 200.4 MLUPS.

8.2 3D LBM Performance Prediction

For a 3D LBM implementation, let us consider the D3Q19 lattice Boltzmann model. The number of floating points to represent a D3Q19 cell, and the required number of operations to update this cell, almost double when compared to that of a D2Q9 cell. To represent a cell state for a D3Q19 model using our D2Q9 based implementation strategy, requires twenty 64-bit numbers (18 for a given cells neighbours, another representing the particle at rest and an extra redundant one similar to D2Q9 implementation). This implies $k=\frac{1}{20}$, 1350 FPGA core clock-ticks to update a cell (D2Q9 cell update takes 674 clock-ticks). A single PE implementation for a D3Q19 model would remain more or less similar to that of a D2Q9 implementation as shown in Figure 5.2, however, the size of the register bank needs to be larger and the state machine that drives the PE data path should have comparatively more states. In terms of the hardware resources, the main compute core PE remains same as D2Q9 model, that is, floating point cores for sum, multiplication and division remain the same and rest can be ignored for the purpose of simplicity. Therefore, a single FPGA based implementation for the above stated numbers, stays compute bound for a maximum of 61 PEs using Equation (3.9), the same as D2Q9 implementation. In the following section we look into the performance prediction of such systems.

8.2.1 Single FPGA based Implementation

For a single FPGA based D3Q19 model implementation, the execution time for the compute bound case is similar to the D2Q9 model as specified by Equation (3.10). Using the given performance model, we predict the execution time for varying system sizes of D3Q19 model computed over 256 iterations. In the D2Q9 model implementation, we could only manage to achieve 16 PEs on a FX140 chip, though theoretically the system can include 61 PEs to stay compute bound. Based on this, we have predicted the performance model for the D3Q19 implementation. Figure 8.3(left), represents the execution time as a function of system size for a D2Q9 model computed over 256 iterations using 16 and 61 PEs respectively. Similarly, the D3Q19 model is as shown on the right-hand-side of the same figure. The figures demonstrate that for a particular system size, the execution time for D3Q19 model is almost double to that of the D2Q9 model implementation.

As shown in Figure 8.3(left), a 262144 cell D2Q9 lattice computed for 256 iterations using a 61 PEs implementation, reaches a theoretical peak performance of 13.55 MLUPS. And a similar sized D3Q19 lattice computed for 256 iterations using a 61 PEs implementation, reaches a theoretical peak performance of 6.77 MLUPS.

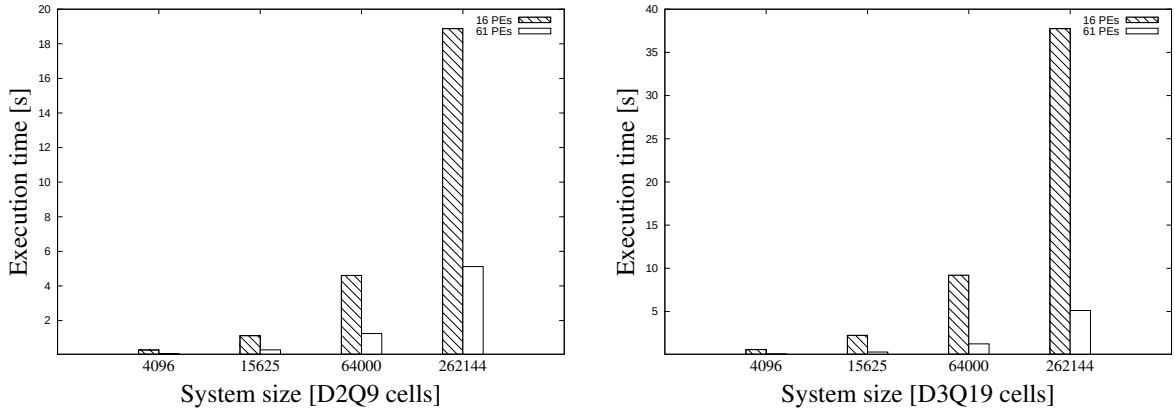


Figure 8.3: *Performance prediction comparison for a FPGA-enabled-PC based D2Q9 and D3Q19 LBM model. Execution time as a function of system size for a single FPGA based implementation. (Left) Each data point represents the execution time to compute 256 iterations of a D2Q9 LBM grid for a given system size with 16 and 61PEs respectively. (Right) Each data point represents the execution time to compute 256 iterations of a D3Q19 LBM grid for a given system size using 16 and 61 PEs respectively.*

8.2.2 FPGA enabled PC Cluster Implementation

For a FPGA enabled PC cluster based D3Q19 implementation, we consider a 3D rectangular problem domain of size 2.04 million cells, decomposed along the longest dimension into F equal sized chunks (1-D domain decomposition setup similar to the 2D implementation as explained in Chapter 7) and computed for 256 iterations using F FPGAs. Each iteration computation is a three-step process similar to the D2Q9 implementation strategy as explained in Chapter 7, that is, starting with boundary cells, followed by bulk cells and finally by the on-board memory update (see Section 7.1 for details). However, the boundary processing, bulk processing, and the boundary download, as specified in Section 7.1, for a D2Q9 implementation differs from D3Q19 implementation as shown below:

8.2.2.1 Boundary Processing

Each FPGA starts with computing boundary cells and can be specified as

$$T_{\mathrm{b}} \;=\; \tau_{\mathrm{r}} + \left\{ \frac{2xz}{p} \right\} \tau_{\mathrm{c}} \tag{8.1}$$

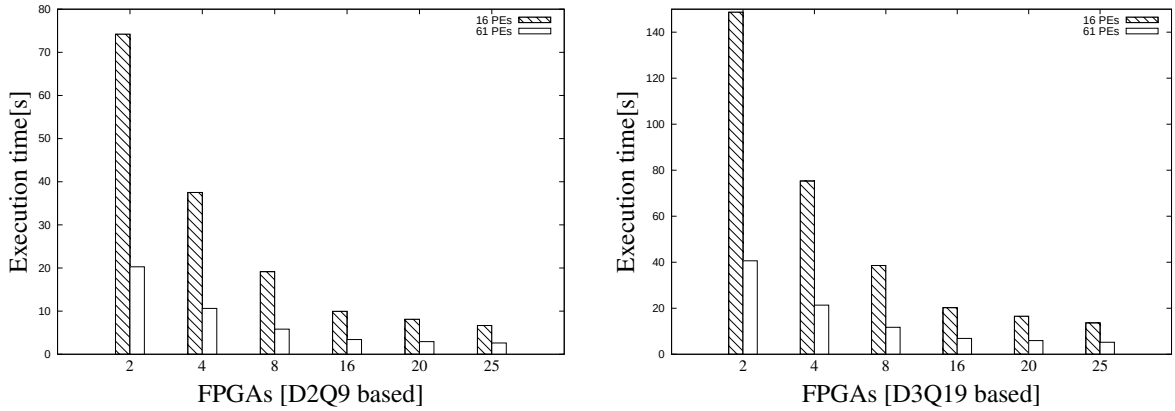


Figure 8.4: *Performance prediction comparison for a FPGA-enabled-cluster based D2Q9 and D3Q19 LBM model. Execution time as a function of FPGAs for a FPGA enabled PC cluster implementation. (Left) Each data point represents the execution time to compute 256 iterations of a 2.04 million cells D2Q9 LBM grid using 16 and 61PEs respectively. (Right) Each data point represents the execution time to compute 256 iterations of a 2.04 million cells D3Q19 LBM grid using 16 and 61 PEs respectively.*

8.2.2.2 Bulk Processing

After the boundary cells, each FPGA computes the bulk cells and in parallel the respective host is informed to start the boundary cells update process around the neighbouring FPGAs. The time to compute the bulk cells and to update the boundary cells is specified as

$$T_{\mathrm{k}} = \frac{1}{p}\left\{\frac{N}{F} - 2xz\right\}\tau_{\mathrm{c}} + \left\{\frac{p}{k} - 1\right\}\tau_{\mathrm{r}} + \tau_{\mathrm{w}} \tag{8.2}$$

$$T_{\mathrm{u}} = 2xz\left\{\tau_{\mathrm{d}} + \tau_{\mathrm{b}} + \tau_{\mathrm{m}}\right\} \tag{8.3}$$

8.2.2.3 Boundary Download

Finally upon the completion of the next iteration computation, the host process downloads the updated boundary data to its connected FPGA modules source bank. The time to download boundary cells (T_{d}) is $2xz\tau_{\mathrm{d}}$. The sum of the above mentioned time durations results in the overall execution time for the FPGA enabled PC cluster implementation.

$$T_{\mathrm{F}} = T_{\mathrm{b}} + \max\left\{T_{\mathrm{u}}, T_{\mathrm{k}}\right\} + T_{\mathrm{d}}. \tag{8.4}$$

Comparing the execution times of D2Q9 and the D3Q19 model as a function of FPGAs as shown in Figure 8.4 respectively, we notice, a system size of 2.04 million cells computed

over 256 generations using 16 and 61PE respectively. The three dimensional setup takes twice the time as compared to the two dimensional setup.

As shown in Figure 8.4, a 2.04 million D2Q9 lattice computed for 256 iterations, using 25 FPGA implementation each with 61 PEs, reaches a theoretical peak performance of 200.23 MLUPS. And a similar sized D3Q19 lattice computed for 256 iterations, using 25 FPGA implementation each with 61 PEs, reaches a theoretical peak performance of 100.11 MLUPS.

8.3 Summary

Using pencil and paper approach we deduced the theoretical limits of our model implementations and their performance relative to the existing implementations. For a single FPGA based implementation, improving the number of PEs from 16 to a theoretical maximum of 61, improves the overall performance by a factor close to four. A theoretical peak performance of 13 MLUPS is expected for such system implementations. For a multiple FPGA enabled PC cluster implementation, improving the number of PEs per FPGA from eight to a theoretical maximum of 61, improves the overall performance of D2Q9 implementation by a factor close to five, with an expected theoretical peak performance of 200 MLUPS. Secondly, by porting our two-dimensional performance model to a three-dimensional D3Q19 LBM implementation, a linear increment in terms of execution time is identified. FPGA logic resources required for D3Q19 PE implementation would largely resemble D2Q9 PE in terms of floating point units and control logic, though a larger register file would be required to store and compute the higher number of computations.

Chapter 9

Manycore Paradigm – What, Why and How?

The previous chapters demonstrated the performance modeling and evaluation of CAs with hundreds of processing elements computing millions of cells. This chapter focuses on running massively parallel systems capable of crunching numbers at a rate of 10^{18} operations per second. The chapter starts with a literature review of the manycore paradigm- a requirement to reach the exaflops scale [2, 67, 88]. Some of the important publications reviewed in this chapter include - a report on DARPA IPTO sponsored ExaScale study [67], a report on parallel computing research compiled in December 2006 by a multidisciplinary group of researchers from Berkeley over a span of two years [5], and work presented by 50 leading experts from academia and industry at EDF organised workshop on using million cores systems [2].

Within the framework of the literature review, we analyse our FPGA based CA implementations and draw parallels between the two. Most importantly we are keen to find out how our work contributes to the existing research. Following are some of the main highlights of the literature review as presented in the following sections:

- The traditional model of exploring hardware and software as a single-domain research initiative in isolation can no longer address the current challenges [67].
- Much can be learned by examining the success of parallelism at the extremes of computing spectrum like high performance computing [5].
- Our ability to scale up applications to millions of processors, or even port conventional codes to a few dozen cores is almost non-existent [67].
- Widening, mitigating, or eliminating the von Neumann bottleneck must be the thrust of research [61, 67].

9.1 A Marriage of Convenience – von Neumann and Moore

Computer simulation is the key high performance computing technique to find solutions to global challenges [49] ranging from weather forecast to terrorist threats [15]. Running HPC simulations demand improvements in computational accuracy and speed, and it is no surprise that the major milestones in the high performance computing are pushed by the emergence of faster computer systems. In general, when the aggregate performance of a computer system first crosses a threshold of 10^{3k} operations per second, for some k, it is defined as a major milestone [67]. Billion floating point operations per second or Gigascale (10^9) was first achieved by Cray-2 in 1985 [108], the next milestone Terascale (10^{12}) was reached almost a decade later in 1997 by the Intel ASCI Red system at Sandia National Laboratory and the most recent milestone of Petascale (10^{15}) was achieved by Roadrunner at Los Alamos National Laboratory in 2008 [6]. Considering the time line with previous milestones and assuming the continued acceleration of progress, Exascale (10^{18}) computing is expected to be around in 2015 [67].

Consistent advances in the CMOS technology have been one of the main building blocks behind these milestones. Technological advancement has resulted in smaller logic and shorter signaling distances within the processor cores, and has been primarily used to enable higher processor clock frequencies, resulting in faster processor computations. Also known as the famous Moore's law, this has been interpreted as the doubling of the transistor density per unit area on a chip every 18-24 months. Moore's law has also been misinterpreted as doubling of performance of the general purpose computer (von Neumann architecture) every 24 months.

The von Neumann architecture, a term coined in 1945 [114], refers to a sequential computer architecture with a processing and a memory unit where memory stores both instructions and the data. Predominantly, it is a sequential thread execution model, where a program is a sequence of instructions that are executed logically one at a time, until completion. This architecture has been the dominant computing paradigm ever since and what we also call as general purpose computing [114].

9.2 The Three Walls

Till 2000, with each turn of the Moore's Law, processors kept getting faster. Simultaneously, in terms of access time and data bandwidth, memory has not been able to keep up with the improvements in processor clock rates. As a consequence, modern computers end up spending most of its time moving data rather than processing the same. This

limited data transfer rate between the processing unit and the memory unit, also known as the von Neumann bottleneck [114], has been one of the main performance limiters of the von Neumann architecture ever since its inception.

To get around this memory wall, a great deal of complexity was pushed within the processor architecture. And in an attempt to keep most of the relevant data for a program close to the processing logic and not violate sequential execution model [67], architecture include deep memory hierarchies also known as multi-level caches. For example, one of the existing fastest super computers, the IBM BG/L [107] can accomplish four floating-point mathematical operations in one clock cycle but requires around 100 such cycles to fetch each operand from the memory [67]. Even worse, since 2004, advances in the uniprocessor performance also dropped down to about 20% per year due to: power dissipation also called power wall and cooling restrictions; limited instruction-level parallelism (ILP) left to exploit; and almost unchanged memory latency [68]. As a consequence the three walls (power, memory and ILP) forced the computer industry to change its course from a uniprocessor to a multiple cores per processor systems; with Intel cancelling its high performance uniprocessor project, and joining IBM and Sun Microsystems in envisioning high performance microprocessors based on multicore processors [5, 54, 68].

9.3 The von Neumann Architecture and Multicore Bond

Moore's Law is still going strong; and in an attempt to exploit improved feature-size and logic density, modern processors also include multiple cores per die called multicore. Other than increasing the number of functional units per chip or the spatial efficiency, multicore architecture also limits the energy consumption per operation and constrains the uniprocessor complexity growth [98].

Most of the current popular multicore processors are based on symmetric multiprocessor (SMP) architecture [113] where the few available powerful cores share the same memory in a symmetric manner, and each core is a conventional processor, executing as per the von Neumann model. This trend of simply replicating conventional processor cores on the same die, by the main stream processor industry, faces some serious limitations. These limitations include situations wherein, a) workloads with multiple sequential threads would benefit but how will individual tasks become faster [5], b) the complexity within the processor architecture is further increased, with multiple cores attempting to share the same memory, and collaborate by explicitly executing separate parts of a program in parallel [67], and c) a single shared memory will end up as a bottleneck as the

number of cores increases and the paradigm is expected to reach a dead end with the number of cores reaching sixteen or so [68].

Moreover, [5] states, "Switching from sequential to modestly parallel computing will make programming much more difficult without rewarding this greater effort with a dramatic improvement in power-performance. Hence, multicore is unlikely to be the ideal answer." And the more challenging tasks with multicore that remain at large include: effectively exploiting multiple-thread parallelism; parallel computing and programming models; aggravated memory wall; slow and flatten rate of pin growth or the interface between the chip to the outside world; and the mechanisms required for efficient inter-process coordination like synchronisation, mutual exclusion and context switching.

Although software and hardware architects are still in the infancy of multicore era, multicore have consolidated their place as being the building blocks of the fastest current supercomputer called Roadrunner [6, 112]. Roadrunner is a hybrid system built out of two different multicore processor architectures, dual-core Opteron processor from AMD and PowerXCell 8i- a cell processor from IBM, reaching the Petascale milestone at Los Alamos National Laboratory in May 2008. One of the proposed machine and programming model for the era of multicore chips are described in [62].

9.4 Exascale Computing

Chips with hundreds of cores are already in the market, for example, 130nm process based single chip with 188 RISC cores from Cisco [5]. More recently, Nvidia announced its latest Fermi GPU [80] with 512 cores assumed to reach a peak performance of 0.5 to 1 teraflops [44]. Evidently Moore's Law is pushing the idea of having manycores per chip and bringing the supercomputing capabilities to everyday devices [53]. This growth of cores per chip is in line with what is required to reach Exascale [88]. [67] describes Exascale systems as, "A 2015 data center sized capacity system is one whose goal would be to allow roughly 1,000X the production of results from the same applications that run in 2010 on the Petascale systems of the time." And to achieve this 1000X increase in the aggregate computational rates, [67] explains that either by improving the computational speed of a single program by three orders of magnitude or have the capability to concurrently run 1000 such jobs at the same time. It is not only about improving the computational capabilities but also the need to reduce the system sizes that perform current level of computing into far smaller packages [67].

9.4.1 Why?

With reference to Exascale systems and related applications, [67] writes, "there is a continuing need to run critical applications at much higher rates of performance than using Petascale machines and such applications are evolving in more complex entities when compared to those of the Terascale era." It further points out that it is not only the classical scientific and engineering applications but also the applications, requirements related to the internet boom in general that demand such systems.

The enormous amount of data moved around the internet via applications like internet commerce, sharing multimedia content, social networking, blogging, search, news, 24/7 connectivity to internet over mobile devices, and many more have pushed the limits of required computing infrastructure both in terms of real-time performance and efficiency. Supporting internet in general requires computing resources and infrastructure at the speed rivaling those of any supercomputer. Situations demanding immediate attention include: power requirements, storage space, physical space, speed etc. Considering such computing requirements in general, [67] mentions, "thus in a real sense the need for advanced computing has grown significantly beyond the need for just flops", and to recognise this fact [67] introduces the use of terms Gigascale, Terascale, Petascale, etc to reflect such systems.

Several studies and workshops, like [2, 67], discuss questions related to manycore systems and the Exascale computing in details. Such studies clearly indicate that the HPC community aims to reach Exascale milestone by 2015 [67].

9.4.2 How?

A serious discussion concerning the Petascale computing was published around 1994 [99] and only after a span of 16 years, in 2008 the first Petascale milestone was actually reached [112]. However, how the system would evolve in term of the technologies, architectures and programming models was known and this was similar to the Terascale machines which were remarkably foreseen early [67]. On the contrary, another 1000X performance would require altogether a different set of tools both in hardware and software as is evident from some of the following comments from industry experts, "To achieve Exascale performance, single, dual or even quad core systems frequency and ops/cycle might improve by 2-4X in next the 8-10 years, only opportunity for dramatic performance is in number of compute engines" [103] and b) "all that is needed to reach Exascale is a 5-10 TF/chip and 100-200K of such chips" [88].

The message is clear that the manycore systems are required to reach the Exascale [103], and we have to look for methods on how to map and glue hardware and software

together guaranteeing overall system efficiency both in terms of solving and specifying the problem. Before we hit the road towards Exascale computing, let us consider some of the key challenges and how these are different compared to the Terascale era.

9.4.3 Challenges are Different

Previously Moore's Law played a significant role in terms of doubling the processor performance every two years, but this is no longer valid as it has hit the power wall; and has essentially resulted in flattening of the clock rates ever since. This also marks the end of dependence on the performance improvements with a single threaded model and pushing the needs to have processor architecture exploit concurrency and locality [67, 88]. If parallelism is the only mechanism in silicon to increase overall system performance then how to exploit parallelism remains a challenge [67]. Towards the software side, the challenge is to find enough concurrency in the applications [88].

Another challenge is to design a fault tolerant system. Smaller feature sizes, design, manufacturing, and low-energy operation requirements, are likely to suffer from higher failure rates due to higher thermal and quantum error rates [120]. Each new process generation, as a result, rapidly increases permanent and transient faults within the chip and therefore, designing a chip with irregular structures becomes increasingly difficult [120]. Hence it is no surprise that fault tolerant design has been cited as one of the major challenges of the future technologies that are expected to mitigate a rapid reduction of yield and reliability [5]. Consequently, [49] calls for a paradigm shift where focus should to be more on fault tolerant software rather than on eliminating failures. Related challenges include: scalability issues with monitoring and notification [49]; development of scalable and naturally fault tolerant parallel algorithms [49]; identifying the class of problems that can be made scalable and self-healing [49]; the type of cooperation between the cores both within and outside a chip [70]; and the overall hardware and software resiliency [88].

Thirdly, right from the start, memory technologies were unable to keep pace with increase in processing speed. With manycore systems the situation gets worse, limiting performance of future designs [23, 88]. How to limit the aggravated memory wall, remains a challenge.

Designing a modern processor itself has become a mammoth challenge. Processor design using latest technologies is quite an expensive and sophisticated job that only few companies can afford. Therefore, the traditional model of exploring hardware and software as a single-domain research initiative in isolation can no longer address the current challenges [67].

Scalability is yet another challenge. There are no clues on how to scale up applications to the millions of processors and even porting conventional programs to a few dozen cores is almost non-existent [67].

With respect to the unique challenges we are facing with the manycore paradigm, [67] concludes that the current technology trends are simply insufficient and to bring alternatives in-line requires explicit direction and support in new research, otherwise a wait of another 16 years will not guarantee the emergence of Exascale systems.

9.4.4 Dos and Don'ts

In the face of the manycore challenges, some of the recommendations by the industry and academia experts are as follows:

Think beyond von Neumann architecture: Academia and industry based manycore focus groups like [5, 61, 67] expect the solutions to be beyond the nature of von Neumann architecture. [61] suggest to look for new computation models that either widen or mitigate the von Neumann bottlenecks and system that provide self-healing and trustworthy hardware and software. [5] call for models that are completely not von Neumann in nature and that exploit Moore's Law in a way different from what has been done historically.

Multicore is unlikely to be the ideal answer: Focus on the manycore instead of multicore systems [5]. UCB review [5] announce a desperate need for new solutions for parallel software and hardware, instead of relying on evolutionary methods to address problems of parallelism via multicore solutions. [5] goes further to write, "It is unwise to presume that multicore architecture and programming models suitable for 2 to 32 processors can incrementally evolve to serve manycore systems of 1000s of processors", and concludes: evolutionary approaches to parallel hardware and software are not going to scale beyond 16 and 32 processors; and switching from sequential to modestly parallel computing techniques, used with multicore, make programming difficult.

Learn from the extremes of computing: The years of experience gained from embedded and high performance computing are highlighted as the launching pads. [5] points out that embedded and high performance computing, the two extremes of the computing spectrum, are both concerned with challenges like power, hardware utilisation, feature size, unauthorised access and viruses. Therefore much can be learned by examining the success of parallelism there. Plus much can be learned from MPI based implementation [5] as MPI will remain where HPC applications may become a set of VMs [13].

Use Dwarfs as stand-ins: To understand and learn how to build manycore systems, [5] recommend using 13 benchmark dwarfs (categories of applications that would target

Dwarf	Description
Dense Linear Algebra	Dense datasets
Sparse Linear Algebra	Data with many zeros
Spectral Methods	FFT
N-Body Methods	Interactions between many discrete points
Structured Grids	Regular grid
Unstructured Grids	Irregular grid
Monte Carlo	Embarrassingly parallel computation on subsets of data
Combinational Logic	Simple operations on massive amounts of data, e.g., encryption
Graph Traversal	Indirect lookups in search
Dynamic Programming	Problem solving via solving simpler sub-problems
Backtrack and Branch-and-Bound	Global optimisation for huge search spaces
Graphical Models	Bayesian Networks and Hidden Markov Models
Finite State Machines	Interconnected States as used in parsing; embarrassingly sequential

Table 9.1: *13 Dwarfs. A benchmark suite from [5], that captures a pattern of computation and communication common to a class of important applications that can help draw broader conclusion about the parallel computing requirements of the future. See [5] for full details.*

manycore systems as shown in Table 9.1, see [5] for details) as a promising approach and using FPGA based system emulators for rapid design space explorations.

Some of the notable initiatives towards Exascale computing include; Chapel- A new parallel language developed by Cray for HPCS [88] and Corona- Optically enabled 256 core processor (10 TFLOP 3D chip) [13]

9.5 Limits of CMOS and Beyond

The existing CMOS technologies feature size of 45nm with an integration capacity of 8 billion transistors is expected to reach 8nm with 256 billion transistors by 2018 [16, 17, 18]. Pushing the limits of CMOS, these numbers translate to having a chip with thousands of simple processors in the near future [53]. With billions of nanoscale electronic devices per square centimeter, the challenge is to design a system that addresses issues like fault tolerance, the three walls, and the power performance ratio as presented in the previous sections. Such requirements have dictated a shift towards scalable and highly parallel microprocessor architectures. Architectures like [12] are actively being studied and some seek to explore different information processing and computing approaches like the nanoscale dynamics systems [2, 120].

[5] states that the trade off between power and performance is going to be the most important factors across the entire spectrum of current and future system applications. Therefore, the current trend is to go manycore with a hope that it will lead us to the Promised Land. Benefits of having a massively parallel architecture with smaller processing units include: ability to perform dynamic voltage scaling and power control at a fine-grain level (and therefore easier to shut down in the face of defects); and easy to

predict their performance and power characteristics [5]. Another advantage of having a massively parallel architecture is its inherent regularity, regularity within the hardware simplifies the diagnoses and isolation of faulty nodes and therefore simplifies fault tolerant implementations. Other advantages include ease of design and functional verification, and an energy efficient way to achieve performance [5].

Assuming availability of manycore systems as inevitable, one would like to know how its processing units, memory and the interconnect network are going to evolve. Before summarising what current technology literature recommends, let us consider some scenarios based on our own experience of massively parallel application implementations as presented in previous chapters in an attempt to get some feel about the deployment of such systems in the future.

9.5.1 Manycore Systems and CA Simulations

Few things are clear about manycore chips of the future: hundreds of simple cores per chip; massively parallel hardware and software setup; and scalable architecture for optimal power-performance ratio. Plus features like, the regular structure for feasible fabrication of chip, configurable processing units tailored to suit application requirements, and using 13 benchmark dwarfs as stand-ins for parallel applications of the future, make evaluation of such systems more realistic and easy. As expected, lot of research in programming models needs to be carried out in terms of how to parallelise conventional sequential codes to massively parallel paradigm. For now we have one less thing to think about as CAs are inherently parallel with structured grids (also among the list of dwarfs, see Table 9.1 for details), and at the same time are a benchmark application that would target manycore systems of the future. Our Maxwell based LBM implementation with tens of processing units computing millions of cells in parallel also has much in common with manycore architectures of the future [5, 10]. Therefore, first let us consider the Maxwell system and find out how much performance we can squeeze out of it, based on our performance model implementation.

9.5.1.1 Squeezing Maxwell

First some assumptions, all of the Maxwell's 64-FPGAs are of the same type (see Appendix 10 for exact details), that is, each FPGA runs 61 LBM PEs (existing multi-FPGA based implementation could only accommodate 8 out of the algorithm's theoretical maximum of 61 PEs), and in aggregate all FPGAs on-board memory banks store around 214 million LBM cells (each FPGA comes attached with a source memory bank of size 256 Mbytes and each cell size is 80 bytes, see Chapter 5 for details). With this available

k (cells read in parallel)	0.1	0.2	0.5
p (PEs per FPGA)	61	123	307
GFLOPS (150 MHz clock freq)	1.43	2.86	7.16
GFLOPS (250 MHz clock freq)	2.39	4.77	10.9
GFLOPS (500 MHz clock freq)	4.77	9.55	23.9
GFLOPS (1.0 GHz clock freq)	9.55	19.1	47.7
GFLOPS (1.5 GHz clock freq)	14.3	28.6	71.6

Table 9.2: *A 64-FPGA parallel system based D2Q9 LBM performance prediction as shown in Equation (7.4). Based on a combination of different values for the given system parameters like number of PEs per FPGA, FPGA and on-board memory interface, and clock frequency. The performance for a particular clock frequency and specified in GFLOPS represents a theoretical maximum a single FPGA chip can achieve. Note, single FPGA performance is a part of 64-FPGA D2Q9 LBM simulations size of 214 million cells.*

hardware running at 150 MHz, a 1-D domain decomposition of 1000x214400 D2Q9 LBM lattice, spreads over 64 FPGA, takeing 0.25 seconds to compute a single iteration based on the performance model in Equation (7.4). Each LBM cell update includes 38 sums, 27 multiplications and 10 divisions, and applying Dongarra's metrics (where sum or a multiplication is a single FLOP and division counts as four FLOPS) as explained in [69], takes 105 FLOP. Therefore, a single iteration computation spread over 64 FPGAs in parallel, performs 22.5 billion FLOP in 0.25 seconds, in other words a peak performance of 90 GFLOPS. To simplify things, lets say a peak performance of 1.4 GFLOPS per FPGA. Up to this point the only unreal thing about the given numbers is 61 PEs per FPGA. Theoretically, this is within the limits of the model, but it is difficult to achieve considering logic capacity of the current FPGAs.

Moving ahead, lets start pushing the given set of parameters of the model. Clock frequency can be increased say up to 1.5 GHz as this is a standard driving frequency for conventional laptop cores. Memory bandwidth can also be increased say from 64-pin to 320-pin interface (each for read and write operation). Increase in data-pin interface also results in increase of theoretical maximum for PEs per FPGA.

As shown in Table 9.2, the core clock frequency when improved by one order combined with five times improved memory bandwidth, results in the improvement of the overall performance per FPGA chip from 1.4 to 71.6 GFLOPS. However, a FPGA with 307 PEs and running at 1.5 GHz is quite distant from the current generation of FPGAs. Fitting 16 PEs per FPGA, as presented in the previous chapters, has been quite a challenge especially for the design to meet signal propagation timing requirements. One of the main hurdles faced during the FPGA based LBM implementation has been the long wire lengths

running between a PE and the memory bank interfaces. With the increase in PEs per FPGA the wire lengths increase proportionally, and the design required more registering around signals both to boost signal strength and meet design timing constraints. This also reduced the driving clock frequency of the circuit. Other than that, the main limitations of this layout is the maximum possible number of PE for a given set of parameters, especially the cell update time. With improved cell update time (that is, consuming less than 675 core clock cycles to update a cell), the theoretical maximum of PE goes down, making this a major limitation. What is desired, is a layout with as many computationally efficient PEs as possible; and not to forget we are looking at thousand cores chips in future and application needs to be scaled accordingly in order to achieve the desired power-performance efficiency.

9.5.1.2 One-to-one Mapping

Let us consider a pure CA hardware implementation to get around the performance model limitation with the number of PEs per chip. Since we assume availability of thousand core chips and our application of interest being inherently parallel algorithm with local connectivity requirements, why not consider a one-to-one mapping?

To make calculations simple, let us focus on a single chip implementation and find out the possibilities of how to improve overall performance. Consider, a 32x32 D2Q9 lattice size with solid or bounce-back boundary conditions, 32-bit single precision floating points for state representations and a 1024 core processor platform. With these assumptions, a one-to-one mapping requires: each cell directly connects to eight of its neighbours; each core has its own floating point unit; and an IO interface for global communication requirements. One of the main strength of this mapping is the uniform hardware structure with even distribution of power-performance ratio. Cell connectivity confined to local neighbours means shorter wire lengths other than the global signals for synchronisation and control. With direct cell-to-cell connectivity resulting in short signal lengths, cores can be run at higher clock frequency. One of the main disadvantages of having one-to-one mapping is the local cell-to-cell data width requirements. Here each cell is to send and receive a 32-bit number from eight of its neighbours and that sums to numerous wires per cell. The situation becomes worse when a one-to-one mapping setup spreads over multiple manycore chips. One solution is to have a bit serial data transfer among cells.

Let us consider some numbers, 105 FLOP for the collision, 675 core clocks ticks to compute a collision, another 100 core clock ticks to exchange data across neighbouring cells (propagation part) since this is a one-to-one mapping, and a clock frequency of 150 MHz. 675 cycles to compute 105 FLOP per cell running at 150 MHz is based on a) our

Core clock ticks to compute a cell	675	338	168
Core clock ticks to exchange boundary	100	100	100
GFLOPS (150 MHz clock freq)	20.8	36.8	60.2
GFLOPS (250 MHz clock freq)	34.7	61.4	100.0
GFLOPS (1.0 GHz clock freq)	146.6	261.8	431.7
GFLOPS (2.5 GHz clock freq)	378.8	677.9	1117.6

Table 9.3: *A 1024 cores processor. D2Q9 LBM lattice size of 1024 cells mapped to 1024 core processor with required cell-to-cell local connectivity. Based on a combination of different values for the given system parameters like core clock ticks to update a cell and cell-to-cell data exchange. The performance for a particular clock frequency and specified in GFLOPS represents a theoretical maximum that a single FPGA chip can achieve.*

LBM implementations and b) considering a bit serial data transfer between cells also at 150 MHz consuming 100 clock cycles.

As shown in Table 9.3, to reach a teraFLOPS scale would require a 1024 core chip running at 2.5 GHz with each core (floating point units inclusive) computing collision (105 FLOP) and data transfer (propagation) in 268 clock ticks. And to reach a target of 5-10 teraFLOPS per chip, the only option is to increase the number of cores per chips by an equal portion. Another question would be, how to use 1024 core chip for lattice size larger

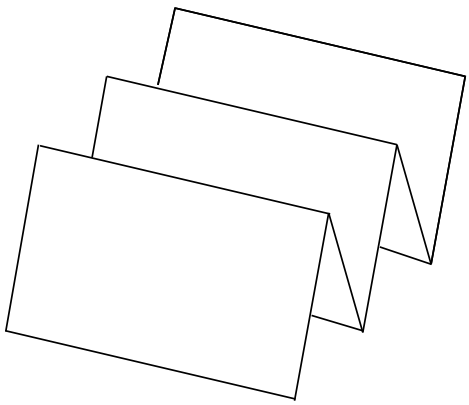


Figure 9.1: *Zigzag shaped: Two dimensional rectangular lattice.*

than the available cores? Considering again the previous situation of direct mapping of 32^2 lattice to 1024 core processor, let us name 32^2 as the base grid. Now if we have a lattice that is l times base grid and imagine the lattice layout in a zigzag form as shown in Figure 9.1, a direct mapping is again possible using 1024 core processor. As long as the cores have sufficient memory, each lattice layer is mapped onto cores with each core storing and processing l cells each belonging to the corresponding layer. As expected, cores start by computing the first layer of the zigzag, followed by the second and so on.

Another advantage with this layout is the ease of implementing wrap-around boundary conditions as well.

The same strategy can be applied to three-dimensional CAs as well, with lattice visualised as a stack of two-dimensional planes and each plane mapped to a manycore chip. Again the main requirement is the core capacity to hold the required number of cells and inter-cellular routing.

9.6 Manycores of Future

Parallelism, uniform structure, processor hungry supercomputing application, reconfigurable processing elements and interconnects, and scalability are some of the common elements that stand out when drawing parallels between the manycore review and our FPGA based CA implementations work. Based on these findings let us summarise the various types of manycore chip systems especially in terms of their processing units, memory and interconnects, that are to be expected in the future.

Starting with the processing units or cores, one would like to find out if these are going to be of same size or heterogeneous, what would be their optimal size and design, and so on. [5] believes that the building blocks of the future microprocessors are likely to be, "simple, modestly pipelined, floating point units, vectors, and SIMD processing elements." And as discussed in previous sections, both size and design parameters of processing elements would be determined by the application under consideration. Identical or heterogeneous processing elements both come with their respective pros and cons. Simple identical elements offer ease of design and implementation in hardware while different sized processing units improve spatial chip efficiency and the parallel speedup by reducing the runtime for the less parallel codes [5]. Additionally, a thousand core chip both in terms of hardware and software in itself is a complex structure to deal with, and including heterogeneous cores would further make things difficult. One approach to include heterogeneity could be to have a chip composed of simple and uniform cores, and then have some of the cores, in aggregate, perform a specific task as and when required. Reconfigurability within the chip can be considered to address the on-demand aggregation of cores to perform specific functions. However, including reconfigurable units also elevate the possibilities of using the available logic space for a variety of different application domains, for example, our D2Q9 LBM PE, with little modifications, can be reconfigured to compute D3Q19 LBM cells.

The memory wall has been the major challenge ever since and [5] report this as the main obstacle for almost half of their identified dwarfs. Manycore designs mean much higher number of instruction executions (or MIPS) per chip and therefore, more memory

bandwidth requirements. The current trends with memory capacity and speed suggest the memory wall getting worse [5]. And considering this MIPS explosion, situation demands innovations beyond traditional von Neumann architecture where the main memory is assumed to be a separate entity connected via standard interfaces. Keeping in view our experience with CA implementation, one of the main limitations of our D2Q9 LBM computation model, as presented in previous chapters, was also due to memory layout. Having a one giant memory block as an input source bank to multiple PEs processing cells limited the data transfer rates and also imposed the limit on the number of PEs within the design. For a thousand core chip this design is not scalable and therefore a model based on distributed memory within the chip is required. Hence, some important class of computations like stencil based operations has a very regular and entirely local memory access. Such applications can benefit from innovations in memory designs and lead the way towards novel designs.

Finally, the interconnects, [70] point out, a random communication pattern on a very large number of cores leads to enormous bisection bandwidth yielding complexity, latency and reliability issues, and simultaneously applications that scale to 10^6 cores with structured communications. Again, it is the application requirements that ask for innovations with how to interconnect hundreds of available cores. Currently, multicore systems employ buses or crossbar switches between the cores and the cache banks and such solutions are not scalable to systems with thousands of cores [5, 113]. Therefore, manycore systems require on-chip connectivity that scales linearly with the system size and at the same time prevents its complexity from dominating the overall costs. Ability to configure the wiring topology between the cores that meets the application communication requirements is an essential requirement in order to have a scalable architecture. Hence, one of the straight forward approaches is to use the flexibility of reconfigurable computing that allows the configuration of the on-chip network topology.

From the above discussion and the various listed desirable features of the machine in future, one thing that is common to most of the elements is the flexibility, not only in terms of software but hardware as well. Whether it's the processor internal structure, memory layout and the access pattern or the interconnect network, all demand to be quite flexible and customisable as per the application requirements. Without doubt, the ability to configure hardware is going to be an integral part of the manycore systems but in what proportion, that would only evolve with time. The current status with reconfigurable devices or FPGA technology in particular, is making rapid progress [104] and doubling in capacity every 18 months. FPGAs, have the capacity of millions of gates and memory bits, and also provide the ease and the speed of reconfiguration like modifying software. They are also narrowing down the concept of having a processor and the memory as

separate entities. FPGAs have recently demonstrated their strength in HPC applications. FPGAs when compared to an equivalent ASIC processor in terms of flexibility, large scale production volumes, and low cost development both in terms of time and cost, suggests their use not only as prototyping or system emulation but also as flexible processors within the manycore system. No doubt that we are very soon to enter the manycore era, however, formulating a power and performance efficient computation is more or less an open question. [5] conclude their report with, "we do not claim to have resolved these questions. Rather our point is that resolution of these questions is certain to require significant research and experimentation". Plus questions ranging right from the micro to the macro level of manycore systems are around, for example, comparing processor to a transistor like a basic building block or like a NAND gate in a standard-cell library [5]. Question about the evolution of manycore processing or the dominant computer architecture of the future in general [120].

As the hunt for answers continues, it would be interesting to investigate a chip with hundreds of simple and uniform cores, each core flexible enough to perform a function either on its own or as a part of a larger unit that is aggregation of cores. This would require cores to be configurable to function as a processing unit, or a memory unit, or as an interconnect conduit.

Chapter 10

Summary and Conclusions

Cellular automata (CA) are inherently decentralised spatially extended systems that are capable of modeling the behaviour of complex systems from nature with a high degree of efficiency and robustness. Field Programmable Gate Arrays (FPGA) are inherently parallel commodity chips that are an ideal platform to investigate the hardware realisation of CA systems. More recently, reconfigurable devices like FPGAs have been integrated into high performance computing (HPC) systems for direct hardware execution of computationally intensive algorithms and thus an opportunity to exploit extreme performance potential.

FPGA based CA accelerators are not new to the scientific community, however mapping CA systems from an abstract concept to the hardware logic remains a major challenge. The availability of more advanced FPGAs and the lack of a problem solving environment targeting such systems, makes it necessary to get a better understanding of the underlying mapping process.

FPGA implementations are still compared to yesteryears writing of computer programs in assembly code, which by no means are as simple and straight forward like edit-compile-run-debug cycle of conventional programming. And to make things tough, system design and implementation are carried out using number of independent tools with majority of them being proprietary. In general, with FPGA based implementation, one gets lost with the minute details of the implementation and unable to see the big picture. Finally in the end it's the performance of the overall system and not the fine technical details that count. Though a lot of research is being carried out to develop tools to target FPGAs via conventional high level programming languages, see for example [82] and [24], but as of now there have been no clear breakthroughs. See [105] for an overview of the latest developments with high level language tools targeting FPGAs. Availability of a problem solving environment with pre-cooked engines, glued together at the run time via high level programming constructs, is highly desirable.

Hence the main focus of this work has been to offer a disciplined look into the issues and requirements of mapping CA algorithms to the physical realities of special purpose hardware. There are many trade-offs to consider when mapping such systems that include both the available FPGA resources and the CA algorithm's execution time. The most important aspect is to fully understand the behaviour of the specified CA algorithm in terms of its execution times which are either compute bound or I/O bound. In this work, the strengths and weaknesses of various hardware architectures employed to achieve the speedups ranging from single to a multiple FPGA based implementations are discussed in detail. This includes performance modeling for each of the implementations and determining the optimal values for the various key parameters that control the overall system implementation.

Performance modeling is a highly popular and commonly used technique in HPC [46]. In contrast, there is dearth of published work that demonstrates the application of performance modeling techniques in HPC using special purpose hardware. Instead designers seem to rely on the use of subset or incremental implementations of their application domain as a benchmark for the overall performance. Therefore, if the overall system implementation is going to be a productive process, is determined by these preliminary results. This methodology is by no mean foolproof with the reason being, the lack of understanding of the overall system and its behaviour.

In this thesis we demonstrate how the traditional technique of performance modeling can be applied to application implementations using FPGAs. This methodology provides a tool to efficiently analyse the proposed system in its entirety for the optimal implementations. Starting with single FPGA based CA implementations, a generic model is presented (Chapter 3). One of the key features of this model is its scalability to accommodate huge CA sizes. Based on the generic model, a methodology is presented to categorise a specified CA algorithm as compute bound or I/O bound. The model is validated for both I/O bound (Chapter 4) and compute bound (Chapter 5) two-dimensional CA algorithms ranging from a bit-state based Game of Life to a floating point based Lattice Boltzmann method. It is demonstrated how this methodology helps to predict the performance of running CA algorithms on specific FPGA hardware and how to determine optimal values for the various parameters that control the mapping process. We find that our model predictions are accurate within 7%. For a single FPGA based D2Q9 Lattice Boltzmann method implementation, an overall speedup of 2.3 is achieved as compared to a general purpose FORTRAN implementation.

As per our performance model, as long as the implementation stays compute bound, porting a D2Q9 Lattice Boltzmann method implementation to a multiple FPGA based implementation was feasible. Consequently, progressing from a single to a dual FPGA

based CA implementation; dual FPGA based implementation (Chapter 6) achieved a speedup close to 1.8 as compared to a single FPGA based implementation.

Progressing further, a multiple FPGA enabled PC cluster implementations are presented (Chapter 7). For a floating-point based CA implementation, parallised and distributed over Maxwell – a 64 FPGA Supercomputer, demonstrated a full parallel system implementation with potentially hundreds of processing elements computing a CA lattice. For a 2-million cells 2D LBM accelerator with periodic boundary conditions, performance model showed how the FPGA enabled PC cluster is the preferred multiple FPGA organisation over the multiple FPGA based PC setup. Latency hiding as the premise through out our multiple FPGA based systems, was fully exploited for our PC cluster based system.

Hence, we presented the application of performance modeling to HPC using special purpose hardware. This demonstrated a cost model for the computation and communication time of the overall FPGA based system implementation. Plus we successfully validated our models with FPGA based implementations that ranged from a single to multiple FPGA based computations. With this we achieved the successful completion of our first two objectives as presented in chapter 1.

Finally, we analysed our experimental work within the framework of the manycore literature review, also specified as our third objective in chapter 1. This work contributes to the novel idea of investigating CAs as a potential parallel processing paradigm on multicore architectures. The emergence of multicore architectures and the era of rolling out hundreds of cores per die sometime in near future, might have triggered the evolution of von Neumann architectures towards a parallel processing paradigm. The capability to have hundreds of cores per die is exciting, however, how optimally we are able to utilise such a resource remains a challenge. Again, the challenge is to best map an application to the underlying manycore architecture in order to identify the various key parameters that control the overall system performance.

In chapter 9, a literature review is presented to understand the current status with manycore systems. Parallelism, uniform structure, processor hungry supercomputing application, reconfigurable processing elements and interconnects, and scalability are some of the common elements that stood out when drawing parallels between the manycore review and our FPGA based CA implementations work. Based on these findings we have summarised the various types of manycore chip systems especially in terms of their processing units, memory, and interconnects that are to be expected in the future. For all the various listed desirable features of the machine in future, one thing that is common to most of the elements is the flexibility, not only in terms of software but hardware as well. The processor internal structure, memory layout and the access pattern or the interconnect network, all elements demand to be quite flexible and customisable as per the

application requirements. Without doubt, the ability to configure hardware is going to be an integral part of the manycore systems but in what proportion, that would only evolve with time. Reconfigurable devices like FPGAs making rapid progress with doubling of capacity every 18 months, is narrowing down the concept of having a processor and the memory as separate entities. These trends suggest FPGA devices playing a significant role in the computing device of the future. The review concludes with a set of open questions, ranging from the micro to the macro level of manycore systems and the evolution of manycore processing, that demand significant research and experimentation. As the hunt for answers continues, for future work, we believe investigating the following is of great interest

- Investigate a chip with hundreds of simple and uniform cores, each core flexible enough to perform a function either on its own or as a part of a larger unit, that is, aggregation of cores. This would require cores to be configurable to function as a processing unit, or a memory unit, or as an interconnect conduit.
- The inherent parallelism – a common denominator among CA systems, multicore architectures, and the FPGA chips – demands a need to investigate and explore the possibility of a future computing platform that might be somewhere along the road, where these three independent concepts intersect.

Appendix

Spartan-3 Board

The Spartan-3 starter kit board [118] from Xilinx has one XC3S200 FPGA, and two 512KB SRAM banks. The two SRAM banks are not independent as both of them are addressed using common pins from the FPGA. A high-level block diagram of the said board is shown in Figure 10.1. The Spartan-3 board runs with a maximum clock frequency of 50MHz and the available asynchronous SRAM chips on board have access time $\leq$ 10ns. Some of the relevant components and features available on this board are as following:

- 200,000-gate Xilinx Spartan-3 XC3S200 FPGA
  - 4,320 logic cells
  - Twelve 18K-bit block RAMs(216K bits)
  - Twelve 18x18 hardware multipliers
  - Up to 173 user-defined I/O signals
- 1M-byte of Asynchronous SRAM
  - Two independent 256Kx16 10 ns SRAM arrays
  - Individual chip select per device
- RS-232 Serial Port, that uses straight-through serial cable to connect to computer serial port
- 50 MHx crystal oscillator clock source

ADMXRC-FX140 Board

The ADM-XRC-4FX PCI Mezzanine board [3] from Alfa-data [38] is a Xilinx Virtex-4 FX140 [119] [117] based PMC with four independent 256MB DDR2-SDRAM banks. Each DDR2 bank can be accessed independently from the FPGA (user logic) and via the PCI interface from the host machine. A high-level block diagram of the said board is shown in Figure 10.2. Some of the relevant components and features available on this board are as following:

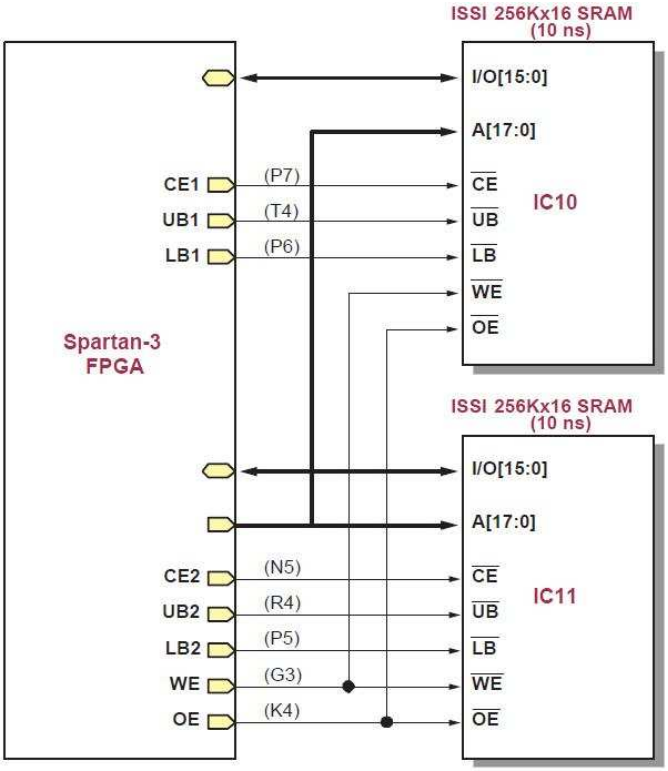


Figure 10.1: *Spartan-3 board: FPGA to SRAM connections. Image taken from [118]*

- Virtex-4 FX140 FPGA
  - 142128 logic cells
  - 552 18K-bit block RAMs(9936K bits)
  - 192 DSP slices
  - 2 PowerPC processor blocks and other I/O blocks
- Four independent 64Mx32 DDR2 SDRAM (1GB total)
- High performance PCI and DMA controllers
- Programmable user clock between 31.25- to 625-MHz

The ADM-XRC-4FX board comes with a software development kit [4] including drivers, header and library files that support a C or C++ program running in the host machine to communicate directly with the board. Alpha-Data also provided an ADM-XRC-4FX Co-Processor Development Kit, used for FPGA-based HPC implementations. Additional code is provided for board initialization and selection, control of the programmable clocks and handling of FPGA configuration files.

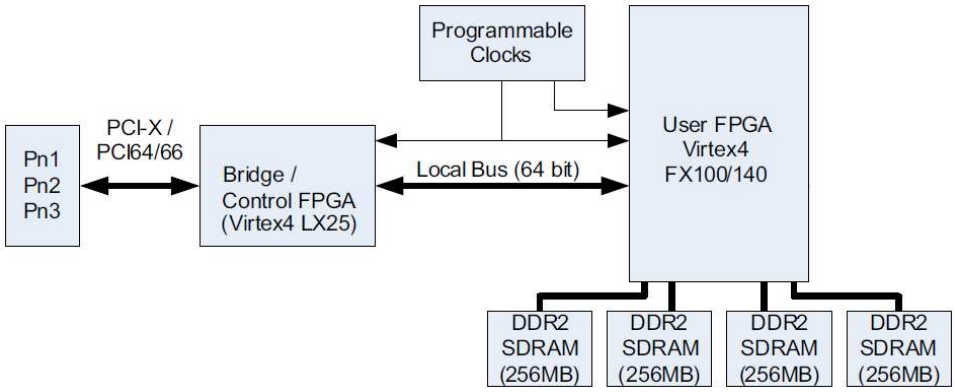


Figure 10.2: *ADM-XRC-4FX PCI Mezzanine card from Alpha-data. Image taken from [3]*

Maxwell 64-FPGA Supercomputer

Maxwell is a 64-FPGA supercomputer [8] from EPCC at the University of Edinburgh, Scotland. Essentially it is an IBM BladeCentre cluster with FPGA acceleration. Each blade server includes one Intel Xeon dual-core processor and two FPGAs. The host CPU and attached FPGA are connected with a standard IBM PCI-X expansion module. The blades are connected over gigabit Ethernet through a switch with 40 Gb/s throughput. In addition to the standard gigabit Ethernet network for CPUs communication, an FPGA network is also provided. The FPGA network consists of point-to-point links between the MGT connectors of adjacent FPGAs, thus forming a two-dimensional torus (see [8] for model details). The idea of having two networks as [8] quotes, "use Ethernet network purely as a control network and to perform parallel communications over RocketIO".

The FPGAs in Maxwell are Xilinx Virtex-4 devices but half of them are FX and rest LX type. FX type come on an Alpha-data [38] supplied PCI expansion card called ADM-XRC fitted with a Xilinx Virtex-4 FX100 FPGA and four independent DDR2 SDRAM banks making a total of 1GByte on-board memory. The other LX type come on an Nallatech [79] supplied card fitted with a Xilinx Virtex-4 LX160 FPGA, SRAM and SDRAM banks. Each vendor supplied FPGA-board comes with its own development kit that enable the communication between the host software process and the FPGA-board resources like memory banks and FPGA core. Secondly, a set of hardware library that provide the necessary interface between the user define logic and the FPGA-board resources like memory, various types of I/O devices.

Since our single FPGA-based D2Q9 LBM CA was implemented using ADM-XRC-FX140 an Alpha-data PCI card, therefore, porting to multiple FPGA system using a similar card (ADM-XRC FX100 PCI card) from Alfa-data made things comfortable. The ADM-XRC FX100 is similar to FX140 board as explained in Appendix 10 except the

FPGA chip itself. This boards comes with a Virtex-4 FX100 that has lower logic density as compared to the FX140 type. This difference in available logic also explains why we were able to implement more LBM PEs using FX140 FPGA (see Chapter 5 for details) as compared to FX100 FPGAs. Other than the FPGA chip rest of the available on-board resources like memory and DMA controllers are same for both boards [3]. Some of the relevant components and features available on ADM-XRC-FX100 board are as following:

- FX100 FPGA
  - 94896 logic cells
  - 376 18K-bit block RAMs(6768K bits)
  - 160 DSP slices
  - 2 PowerPC processor blocks and other I/O blocks
- Four independent 64Mx32 DDR2 SDRAM (1GB total)
- High performance PCI and DMA controllers
- Programmable user clock between 31.25- to 625-MHz

The FX100 based ADM-XRC-4FX board from Alpha-data comes with a software development kit [4] as explained in Appendix 10. Additionally, Parallel toolkit (PTK)- a high-level configuration and APIs are supplied by FPGA HPC Alliance [9] [45]. PTK comprises a library of C++ classes providing a) abstract interfaces to application components implemented using FPGA hardware, b) loading or configuring FPGA with bitstreams, and a standard way of launching parallel FPGA jobs. PTK and its API are explained in [9].

Acronyms and Symbols

ASIC	Application specific integrated circuit
BRAM	Block RAM
CA	Cellular automata
CB	Compute block
CE	Compute engine
CLB	Configurable logic block
COTS	Commercial off the shelf
DDR	Double data rate
DRAM	Dynamic RAM
DSP	Digital signal processor
FDTD	Finite difference time domain
FPGA	Field programmable gate array
HPC	High performance computing
HPRC	High performance reconfigurable computing
LBM	Lattice Boltzmann method
LGCA	Lattice Gas Cellular Automata
LUT	Lookup table
MLUPS	Million lattice-site updates per second
PCI	Peripheral component interconnect
PE	Processing element
PMC	PCI mezzanine card
RAM	Random access memory
SDRAM	Synchronous dynamic RAM
SRAM	Static RAM

T_{pc}	CA execution time on a PC
T_{fpga}	CA execution time on a FPGA
T_{ft}	CA execution time on a FPGA-enabled PC
T_{pre}	Time to pre-process a CA lattice by a host machine
T_{pos}	Time to post-process a CA lattice by a host machine
T_s	Time to download a CA lattice from the host PC to the FPGA on-board memory
T_r	Time to upload a CA lattice from the FPGA on-board memory to the host PC

T_{fo}	Total overhead time ($= T_{pre}+T_s+T_r+T_{pos}$)
p	Number of PEs running in a FPGA
k	Number of CA cell FPGA reads in parallel from source memory bank
N	Number of CA lattice cells
x	Number of CA lattice columns
y	Number of CA lattice rows
n	Number of compute blocks (pipeline depth) within a FPGA
g	Number of CA iterations
w	Buffer size within a compute block
F	Number of FPGAs
τ_{r}	FPGA time to read k-cells from the on-board memory bank
τ_{w}	FPGA time to write k-cells to a on-board memory bank
τ_{c}	FPGA time to update a cell
τ_{d}	Host PC time to download a cell to a FPGA on-board memory
τ_{m}	Time to update a boundary cell across two neighbouring FPGAs
T_{c}	FPGA execution time for a compute bound computation
T_{i}	FPGA execution time for an I/O bound computation
T_1	FPGA execution time to compute a single CA iteration
T_2	Execution time to compute a single CA iteration using a dual FPGA-enabled system
T_{f}	Execution time to compute a CA iteration using a F FPGA-enabled PC
T_{F}	Execution time to compute a CA iteration using a F FPGA-enabled PC cluster

Bibliography

[1] Workshop on Computation in Nanoscale Dynamical Systems. Website, 2006. www.fena.org/downloads_public/CNDSWorkshop051115.pdf.

[2] Workshop on Simulating the Future - Using one Million Cores and Beyond. Website, sep 2008. http://rd.edf.com/edf-fr-accueil/edf-recherche–developpement/poubelle/september-22-24–simulating-the-future-600098.html.

[3] Alpha-data. ADM-XRC-4FX PCI Mezzanine Card User Guide. User guide, Alpha-data, Edinburgh, UK, . www.alpha-data.com/pdfs/adm-xrc-4fxusermanual.pdf.

[4] Alpha-data. ADM-XRC SDK User Guide. User guide, Alpha-data, Edinburgh, UK, . www.alpha-data.com/pdfs/admxrcsdk.pdf.

[5] Krste Asanovic, Ras Bodik, Bryan Catanzaro, Joseph Gebis, Parry Husbands, Kurt Keutzer, David Patterson, William Plishker, John Shalf, Samuel Williams, and Katherine Yelick. The landscape of parallel computing research: a view from Berkeley. Study report, Electrical Engineering and Computer Sciences, University of California at Berkeley. www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf.

[6] Kevin Barker, Kei Davis, Adolfy Hoisie, Darren Kerbyson, Mike Lang, Scott Pakin, and Jose Sancho. Entering the petaflop era: the architecture and performance of Roadrunner. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[7] Michael Barr. A Reconfigurable Computing Primer. *Multimedia Systems Design*, pages 44–47, 1998.

[8] Rob Baxter, Stephen Booth, Mark Bull, Geoff Cawood, James Perry, Mark Parsons, Alan Simpson, Arthur Trew, Andrew McCormick, Graham Smart, Ronnie Smart, Allan Cantle, Richard Chamberlain, and Gildas Genest. Maxwell - a 64 FPGA Supercomputer. *Journal of the American Helicopter Society*, 0, 2007.

[9] Rob Baxter, Stephen Booth, Mark Bull, Geoff Cawood, James Perry, Mark Parsons, Alan Simpson, Arthur Trew, Andrew McCormick, Graham Smart, Ronnie Smart, Allan Cantle,

Richard Chamberlain, and Gildas Genest. The FPGA High-Performance Computing Alliance Parallel Toolkit. In *AHS '07: Second NASA/ESA Conference on Adaptive Hardware Systems*, pages 301–310. IEEE XPLORE, 2007.

[10] Francesco Belletti, Maria Cotallo, Andrés Cruz, Luis Antonio Fernandez, Antonio Gordillo-Guerrero, Marco Guidetti, Andrea Maiorano, Filippo Mantovani, Enzo Marinari, Victor Martin-Mayor, Antonio Muñoz-Sudupe, Denis Navarro, Giorgio Parisi, Sergio Perez-Gaviro, Mauro Rossi, Juan Jesús Ruiz-Lorenzo, Sebastiano Fabio Schifano, Daniele Sciretti, Alfonso Tarancon, Raffaele (lele) Tripiccione, José Luis Velasco, David Yllanes, and Gianpaolo Zanier. Janus: An FPGA-Based System for High-Performance Scientific Computing. *Computing in Science and Engineering*, 11(1), .

[11] Francesco Belletti, Filippo Mantovani, Giorgio Poli, Sebastiano Fabio Schifano, R. Tripiccione, Isabel Campos, Andres Cruz Flor, Denis Navarro, Sergio Perez Gaviro, Daniele Sciretti, Alfonso Tarancon, Jos? Luis Velasco, Pedro Tellez, Luis Antonio Fernandez, Victor Martin-Mayor, Antonio Munoz Sudupe, Sergio Jimenez, Andrea Maiorano, Enzo Marinari, and Juan Jesus Ruiz-Lorenzo. Ianus: An Adaptive FPGA Computer. *Computing in Science and Engineering*, 8(1), .

[12] Luca Benini and Giovanni De Micheli. Networks on Chips: A New SoC Paradigm. *Computer*, 35(1).

[13] Ed Benson. Different technology approach to address large scale computing. Website, 2008. http://rd.edf.com/edf-fr-accueil/edf-recherche–developpement/poubelle/september-22-24–simulating-the-future-600098.html.

[14] S. M. Boghosian. Lattice gas and cellular automata. *Future Generation Computing Systems*, 16(0):171–185, 1999.

[15] Jay Boris. The Threat of Chemical and Biological Terrorism: Roles for HPC in Preparing a Response. *Computing in Science and Engg.*, 4(2), 2002.

[16] Shekhar Borkar. Electronics beyond nano-scale CMOS. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 807–808, NY, USA, 2006. ACM.

[17] Shekhar Borkar. Thousand core chips: a technology perspective. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 746–749, NY, USA, 2007. ACM.

[18] Shekhar Borkar. Design Challenges of Technology Scaling. *IEEE Micro*, 19(4), 1999.

[19] Stephen Brown and Zvonko Vranesic. *Fundamentals of Digital Logic with VHDL Design*. McGraw Hill, second edition.

[20] Duncan Buell, Tarek El-Ghazawi, Kris Gaj, and Volodymyr Kindratenko. Guest Editors' Introduction: High-Performance Reconfigurable Computing. *Computer*, 40(3).

[21] Duncan A. Buell and Kenneth L. Pocek. Custom computing machines: an introduction. *Journal of Supercomputing*, 9(3), 1995.

[22] J. M. Buick. *Lattice Boltzmann Methods in Interfacial Wave Modelling*. PhD thesis, University of Edinburgh, 1997.

[23] Bill Camp. Some technology issues for Exascale computing. Website, 2008. http://rd.edf.com/edf-fr-accueil/edf-recherche–developpement/poubelle/september-22-24–simulating-the-future-600098.html.

[24] Allan Cantle. FPGA Accelerated Computing Solutions. In *HPRCTA '08: Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications*, 2008. `http://gladiator.ncsa.illinois.edu/PDFs/hprcta08/session4/cantle.pdf`.

[25] G. Cappuccino and G. Cocorullo. Custom Reconfigurable Computing Machine for High Performance Cellular Automata Processing. In *TechOnLine Publication*, jul 2001.

[26] J. Castillo, Jose L. Bosque, E. Castillo, P. Huerta, and J.I. Martinez. Hardware accelerated montecarlo financial simulation over low cost FPGA cluster. *Parallel and Distributed Processing Symposium, International*, 0, 2009.

[27] Wang Chen. *Acceleration of the 3D FDTD Algorithm in Fixed-point Arithmetic using Reconfigurable Hardware*. PhD thesis, Northeastern University, 2007.

[28] Wang Chen and Miriam Leeser. FDTD: Finite Difference Time Domain Ű A Case Study Using FPGAs. In *Reconfigurable Computing: The Theory and Practice of FPGA-based Computation*.

[29] Wang Chen, Panos Kosmas, Miriam Leeser, and Carey Rappaport. An FPGA implementation of the two-dimensional finite-difference time-domain (FDTD) algorithm. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 213–222, NY, USA, 2004. ACM.

[30] B. Chopard and M. Droz. *Cellular Automata Modeling of Physical Systems*. Cambridge University Press.

[31] Katherine Compton and Scott Hauck. An Introduction to Reconfigurable Computing. *IEEE Computer*, 2000.

[32] Katherine Compton and Scott Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys*, 34:171–210, 2002.

[33] SRC Computers. www.srccomp.com.

[34] DRC Computer Corporation. www.drccomputer.com.

[35] Stephen Craven and Peter Athanas. Examining the viability of FPGA supercomputing. *EURASIP J. Embedded Syst.*, 2007(1), 2007.

[36] R. Culley, A. Desai, S. Gandhi, Wu Shugaung, and K. Tomko. A prototype FPGA finite-difference time-domain engine for electromagnetics simulation. In *MSCS'05: Proceeding of 48th Midwest Symposium on Circuits and Systems*, volume 1, pages 663–666, 2005.

[37] P. F. Curt, J. P. Durbano, M. R. Bodnar, S. Shi, and M. S. Mirotznik. Enhanced Functionality for Hardware-Based FDTD Accelerators. In *ACES'06: Proceeding of 22nd Annual Review of Progress in Applied Computational Electromagnetics*, 2006.

[38] Alpha Data. www.alpha-data.com.

[39] André DeHon and John Wawrzynek. Reconfigurable computing: what, why, and implications for design automation. In *DAC '99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 610–615, NY, USA, 1999. ACM.

[40] A. Deutsch and S. Dormann. *Cellular Automaton Modeling of Biological Pattern Formation*. Birkhauser.

[41] D. Dubbeldam, A. G. Hoekstra, and P. M. A. Sloot. Dynamic structure factor in single- and two-species thermal gbl lattice gas. *Computer Physics Communications*, 129:13–20, 2000.

[42] Tarek El-Ghazawi, Esam El-Araby, Miaoqing Huang, Kris Gaj, Volodymyr Kindratenko, and Duncan Buell. The Promise of High-Performance Reconfigurable Computing. *Computer*, 41(2).

[43] Trenz Electronic. Introduction to FPGA Technology. Website, 2001. `www.techonline.com`.

[44] Michael Feldman. NVIDIA takes GPU computing to the next level. Website, 2009. `www.hpcwire.com/features/NVIDIA-Takes-GPU-Computing-to-the-Next-Level-62800147.html`.

[45] FHPCA. www.fhpca.org.

[46] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0201575949.

[47] Toshiyuki Fukushige, Piet Hut, and Junichiro Makino. High-performance special-purpose computers in science. *Comput. Sci. Eng.*, 1(2), 1999.

[48] Niloy Ganguly, Biplab K Sikdar, Andreas Deutsch, Geoffrey Canright, and P Pal Chaudhuri. A Survey on Cellular Automata. Website, 2003. `www.cs.unibo.it/bison/publications/CAsurvey.pdf`.

[49] AI Geist. The Challenge of continues failure. Website, 2008. http://rd.edf.com/edf-fr-accueil/edf-recherche–developpement/poubelle/september-22-24–simulating-the-future-600098.html.

[50] Maya Gokhale, Christopher Rickett, Justin L. Tripp, Chung Hsu, and Ronald Scrofano. Promises and Pitfalls of Reconfigurable Supercomputing. In *ERSA*, pages 11–20, 2006.

[51] H. A. Gutowitz. *Cellular Automata*. MIT Press, 1990.

[52] Mathias Halbach and Rolf Hoffmann. Implementing Cellular Automata in FPGA Logic. *Parallel and Distributed Processing Symposium, International*, 16, 2004.

[53] Jim Held, Jerry Bautista, and Sean Koehl. From a few cores to many: A tera-scale computing research overview. White paper, Intel, 2006.

[54] John Hennessy and David Patterson. *Computer architecture: a quantitative approach*. 2002.

[55] Martin C. Herbordt, Yongfeng Gu, Tom VanCourt, Josh Model, Bharat Sukhwani, and Matt Chiu. Computing Models for FPGA-Based Accelerators. *Computing in Science and Engineering*, 10(6).

[56] Martin C. Herbordt, Tom VanCourt, Yongfeng Gu, Bharat Sukhwani, Al Conti, Josh Model, and Doug DiSabello. Achieving High Performance with FPGA-Based Computing. *IEEE Computer*, 30, 2007.

[57] H. J. Hilhorst, A. F. Bakker, C. Bruin, A. Compagner, and A. Hoogland. Special purpose computers in physics. *Journal of Statistical Physics*, 34(5-6), 1984.

[58] Mark D. Hill and Michael R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41 (7).

[59] Andrew Ilachinski. *Cellular Automata: A Discrete Universe*. World Scientific, 2001.

[60] Cray Inc. www.cray.com.

[61] Mary Jane Irwin and John Shen. Revitalizing Computer Architecture Research. Study report, Computing Research Association, Monterey Bay, CA, 2005. `http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/exascale_final_report_100208.pdf`.

[62] Chris Jesshope. A Model for the Design and Programming of Multi-cores. *Advances in Parallel Computing - High Performance Computing and Grids in Action*, 2008.

[63] Drona Kandhai. *Large Scale Lattice-Boltzmann Simulations - Computational Methods and Applications*. PhD thesis, University of Amsterdam, 1999.

[64] Randy H. Katz and Gaetano Borriello. *Contemporary Logic Design*. Prentice Hall, second edition.

[65] T. Kobori, T. Maruyama, and T. Hoshino. A Cellular Automata System with FPGA. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 120–129, 2001.

[66] Tomoyoshi Kobori and Tsutomu Maruyama. A High Speed Computation System for 3D FCHC Lattice Gas Model with FPGA. In *FPL '03: Proceedings of International Conference on Field Programmable Logic and Applications*, pages 755–765. IEEE, 2003.

[67] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems. Study report no. 300, DARPA Information Processing Techniques Office, Washington, DC, 2008. `http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/exascale_final_report_100208.pdf`.

[68] E. Kolonis, M. Nicolaidis, D. Gizopoulos, M. Psarakis, J.H. Collet, and P. Zajac. Enhanced self-configurability and yield in multicore grids. *IEEE International On-Line Testing Symposium*, 0, 2009.

[69] Alice Koniges. Parallel Computing Architecture and Resources. Website, 2008. `http://sc08.supercomputing.org/scyourway/conference/view/tut110.html`.

[70] J Lavignon. Which core, how connected, how managed. Website, 2008. http://rd.edf.com/edf-fr-accueil/edf-recherche–developpement/poubelle/september-22-24–simulating-the-future-600098.html.

[71] Eric Lorenz, A.G. Hoekstra, and A. Caiazzo. Lees-Edwards boundary conditions for lattice Boltzmann suspension simulations. *Physical Review E*, 79(3):036706++8, March 2009. doi: 10.1103/PhysRevE.79.036706.

[72] Junichiro Makino, Makoto Taiji, Toshikazu Ebisuzaki, and Daiichiro Sugimoto. GRAPE-4: a one-Tflops special-purpose computer for astrophysical N-body problem. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, pages 429–438, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[73] Norman Margolus. CAM-8: A Computer Architecture Based on Cellular Automata. Website, 1993. `www.ai.mit.edu/people/nhm/cam8.pdf`.

[74] Clive Maxfield. *The Design Warrior's Guide to FPGAs*. Newnes.

[75] Oscar Mencer. ASC: a stream compiler for computing with FPGAs. 2006.

[76] Oscar Mencer, Kuen H Tsoi, Stephen Craimer, Timothy Todman, and Wayne Luk. CUBE: A 512-FPGA cluster. In *SCPL'09: Proceeding of the 5th Southern Conference on Programmable Logic*, 2009.

[77] S. Murtaza, A.G. Hoekstra, and P.M.A. Sloot. Floating point based Cellular Automata simulations using a dual FPGA-enabled system. In *HPRCTA '08: Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications*, pages 1–8. IEEE.

[78] S. Murtaza, A.G. Hoekstra, and P.M.A. Sloot. Compute Bound and I/O Bound Cellular Automata Simulations on FPGA logic. *ACM Transactions on Reconfigurable Technology and Systems*, 1(4), 2009.

[79] Nallatech. www.nallatech.com.

[80] Nvidia.com. Fermi Architecture. Website. `www.nvidia.com`.

[81] Y. Pomeau P. Hardy and O. de Pazzis. Time evolution of a two-dimensional model system. I. Invariant states and time correlation functions. *Journal of Mathematical Physics*, 1(0): 1746–1759, 1973.

[82] David Pellerin. Impulse Software-to-FPGA Technology Update. In *HPRCTA '08: Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications*, 2008. `http://gladiator.ncsa.illinois.edu/PDFs/hprcta08/session4/pellerin.pdf`.

[83] P. Rendell. This Is a Turing Machine Implemented in Conway's Game of Life. Website. `www.rendell-attic.org/gol/tm.htm`.

[84] J. P. Rivet and J. P. Boon. *Lattice Gas Hydrodynamics*. Cambridge University Press.

[85] D. H. Rothman and S. Zaleski. *Lattice-Gas Cellular Automata, Simple Models of Complex Hydrodynamics*. Cambridge University Press, Cambridge, 1997.

[86] Kentaro Sano, Oskar Mencer, and Wayne Luk. FPGA-based Acceleration of the Lattice Boltzmann Method. In *PCFD'07: International Conference on Parallel Computational Fluid Dynamics*, 2007.

[87] Ron Sass, William V Kritikos, Andrew G. Schmidt, Srinivas Beeravolu, and Parag Beeraka. Reconfigurable Computing Cluster (RCC) Project: Investigating the Feasibility of FPGA-Based Petascale Computing. In *FCCM '07: Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 127–140, Washington, DC, USA, 2007. IEEE Computer Society.

[88] Steve Scott. Cray's approach to million-core systems. Website, 2008. http://rd.edf.com/edf-fr-accueil/edf-recherche–developpement/poubelle/september-22-24–simulating-the-future-600098.html.

[89] SGI. www.sgi.com.

[90] D. Shand, R. Chamberlain, D. Denning, and E. Lord. A study into implementing the lattice Boltzmann floating point model with reconfigurable computing. In *RSSI'06: Proceedings of The Second Reconfigurable Systems Summer Institute*.

[91] D. Shand, D. Denning, and R. Chamberlain. Lattice Gases - Simple Models of Complex Fluid Dynamics. Website, 2005. White Paper, NT309-0001.

[92] P. M. A. Sloot and A. G. Hoekstra. Modeling Dynamical Systems with Cellular Automata. In *Handbook of Dynamic System Modeling, Editor: Paul A. Fishwick*. CRC Press, 2007.

[93] P. M. A. Sloot and A. G. Hoekstra. Cellular Automata as a Mesoscopic Approach to Model and simulate Complex Systems. In *Lecture Notes in Computer Science: Springer Verlag*.

[94] P. M. A. Sloot, B. Chopard, and A. G. Hoekstra. *Cellular Automata: 6th International Conference on Cellular Automata for Research and Industry, ACRI 2004*. Springer Verlag, oct .

[95] P. M. A. Sloot, J. A. Kaandorp, A. G. Hoekstra, and B. J. Overeinder. Distributed Cellular Automata: Large Scale Simulation of Natural Phenomena. In *Solutions to Parallel and Distributed Computing Problems, Lessons from Biological Sciences*, pages 1–46, 2001.

[96] P. M. A. Sloot, B. J. Overeinder, and A. Schoneveld. Self-organized criticality in simulated correlated systems. *Computer Physics Communications*, 142:76–81, 2001.

[97] William D. Smith and Austars R. Schnore. Towards an RCC-based accelerator for computational fluid dynamics applications. *The Journal of Supercomputing*, 30(3), 2004.

[98] Thomas Sterling. Asymptotic Continuum OS for Exascale Computing. Website, 2008. http://rd.edf.com/edf-fr-accueil/edf-recherche–developpement/poubelle/september-22-24–simulating-the-future-600098.html.

[99] Thomas Sterling, Paul Messina, and Paul H. Smith. *Enabling Technologies for Petaflops Computing*. 1995.

[100] S. Succi. *The Lattice-Boltzmann Equation*. Oxford University Press, Oxford, 2001.

[101] T.J. Todman, G.A. Constantinides, S.J.E Wilton, Oscar Mencer, Wayne Luk, and P.Y.K. Cheung. Reconfigurable computing: architectures and design methods. In *FCCM '07: Proceedings of the IEE Computers and Digital Techniques*, pages 193–207. IEEE Digital Library, 2005.

[102] T. Toffoli and N. Margolus. *Cellular Automata Machines*. MIT Press, 1987.

[103] David Turek. Challenges on the Road to Exascale computing. Website, 2008. http://rd.edf.com/edf-fr-accueil/edf-recherche–developpement/poubelle/september-22-24–simulating-the-future-600098.html.

[104] Keith Underwood. FPGAs vs. CPUs: trends in peak floating-point performance. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 171–180, NY, USA, 2004. ACM.

[105] Jason Villarreal, Adrian Park, Walid Najjar, and Robert Halstead. Designing Modular Hardware Accelerators in C With ROCCC 2.0. In *FCCM '10: The 18th Annual International IEEE Symposium on Field-Programmable Custom Computing Machines*, 2010. `http://www.cs.ucr.edu/~najjar/papers/2010/FCCM-2010.pdf`.

[106] Richard Wain, Ian Bush, Martyn Guest, Miles Deegan, Igor Kozin, and Christine Kitchen. An overview of FPGAs and FPGA programming - Initial experiences at Daresbury. Technical report, CCLRC Daresbury Laboratory, Cheshire, UK. `http://epubs.cclrc.ac.uk/bitstream/1167/DL-TR-2006-010.pdf`.

[107] Wikipedia. Blue Gene. Website, 2009. `www.en.wikipedia.org/wiki/Blue_Gene/P`.

[108] Wikipedia. Cray-2. Website, 2009. `www.en.wikipedia.org/wiki/Cray-2`.

[109] Wikipedia. Conway's Game of Life. Website, 2009. `www.wikipedia.org`.

[110] Wikipedia. Ordnance Discrete Variable Automatic Computer. Website, 2009. `www.en.wikipedia.org/wiki/ORDVAC`.

[111] Wikipedia. Parallel computing. Website, 2009. `www.en.wikipedia.org/wiki/Parallel_computing`.

[112] Wikipedia. Roadrunner. Website, 2009. `www.wikipedia.org`.

[113] Wikipedia. Symmetric multiprocessing. Website, 2009. `www.en.wikipedia.org/wiki/Symmetric_multiprocessing`.

[114] Wikipedia. Von Neumann Architecture. Website, 2009. `www.wikipedia.org`.

[115] Dieter A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models*. Springer.

[116] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 2002.

[117] Xilinx. Virtex-4 FPGA User Guide. Technical report, Xilinx, .

[118] Xilinx. Spartan-3 FPGA Starter Kit Board User Guide. Technical report, Xilinx, .

[119] Xilinx. Virtex-4 Family Overview. Technical report, Xilinx, .

[120] Victor Zhirnov, Ralph Cavin, Greg Leeming, and Kosmas Galatsis. An Assessment of Integrated Digital Cellular Automata Architectures. *IEEE Computer*, 41(1), 2008.

Samenvatting in het Nederlands*

Een Cellulaire Automaat is een inherent discreet, decentraal, en ruimtelijk uitgespreid system dat in staat is om het gedrag van complexe natuurlijke systemen met een hoge graad van nauwkeurigheid en robuustheid te modelleren. Het blijft een grote uitdaging om het abstracte concept van een Cellulaire Automaat af te beelden op computer hardware. De "Field Programmable Gate Array" (FPGA) is een ideaal platform om hardware implementaties van Cellulaire Automaten te bestuderen. Hoog vermogen rekenen (High Performance Computing - HPC) met FPGAs maakt het mogelijk om rekenintensieve algoritmes direct in de hardware uit te voeren en op die manier het extreme potentieel aan rekenkracht van special purpose HPC uit te buiten.

Het belangrijkste doel van dit werk is een disciplinegebonden gezichtspunt op vereisten en problemen die samenhangen met het afbeelden van Cellulaire Automaten op de fysische realiteit van 'special purpose' hardware. Voor - en nadelen van een breed scala aan hardware architecturen, van een enkele to meerdere parallelle FPGAs, worden in detail onderzocht. Het modelleren van de prestaties voor elke implementatie, en het bepalen van de optimale waarde van de meeste belangrijk parameters die de prestatie beïnvloeden is een belangrijk deel van het onderzoek.

Startend met een implementatie op één FGPA is een generiek model ontwikkelend, waarbij één van de belangrijkste grootheden de schaalbaarheid van de implementatie is, zodat enorme grote Cellulaire Automaten bestudeerd kunnen worden. Gebaseerd op dit model worden Cellulaire Automaten gecategoriseerd as rekenbegrensd of i/o-begrensd. Het model is vervolgens gevalideerd voor zowel rekenbegrensde en i/o-begrensde tweedimensionale Cellulaire Automaten (variërend van het enkele bit-state 'Game of Life' tot een floating point gebasseerde Rooster Boltzmann Automaat). De modelvoorspellingen zijn tot 7% nauwkeurig. Voor een implementatie van de D2Q9 Rooster Boltzmann automaat op één FPGA is een speedup gemeten van 2,3 in vergelijking met een Fortran implementatie op een general purpose CPU.

Vervolgens is een implementatie gerealiseerd op twee FPGAs, en in vergelijking met de implementatie op één enkele FPGA is een speedup van 1,8 gerealiseerd. Het prestatie model laat zien dat zolang de implementatie rekenbegrensd is, het goed mogelijk is een

*translation by Dr. Alfons Hoekstra.

D2Q9 Rooster Boltzmann Automaat op meerdere FPGAs te implementeren. Vervolgens is een implementatie gerealiseerd van de floating-point Cellulaire Automaat op Maxwell - een supercomputer met 64 parallelle FPGAs - waarmee het potentieel van special purpose rekening voor HPC is gedemonstreerd. Ook voor deze implementatie is een prestatiemodel ontwikkeld en gevalideerd.

Ten slotte draagt dit werk bij een nieuw idee om te onderzoeken hoe cellulaire automaten als parallel rekenparadigma zouden kunnen functioneren op multicore architecturen. Het inherente parallellisme van Cellulaire Automaten, multicore architecturen en FPGA chips vraagt om dieper onderzoek naar een mogelijke toekomstige rekenplatform op het kruispunt van deze drie concepten.

Publications

1. S. Murtaza, A.G. Hoekstra, and P.M.A. Sloot, "Performance of Cellular Automata Simulations on a FPGA cluster," 2010, Submitted.

2. S. Murtaza, A.G. Hoekstra, and P.M.A. Sloot, "Performance of floating-point based Cellular Automata Simulations using a dual FPGA system," 2009, Submitted.

3. S. Murtaza, A.G. Hoekstra, and P.M.A. Sloot, "Compute Bound and I/O Bound Cellular Automata Simulations on FPGA logic," in *ACM Transactions on Reconfigurable Technology and Systems*, vol. 1, nr. 4, pp. 1–21. ACM, New York, January 2009.

4. S. Murtaza, A.G. Hoekstra, and P.M.A. Sloot, "Floating point based Cellular Automata simulations using a dual FPGA-enabled system," in *Proceeding of the Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA'08)*, held in conjunction with SC08, pp. 1-8. IEEE, Austin, Texas, November 27–29, 2008.

5. S. Murtaza, A.G. Hoekstra, and P.M.A. Sloot, "Performance modeling of 2D Cellular Automata on FPGA," in *17th International Conference on Field Programmable Logic and Applications (FPL'07)*, pp. 74–78. IEEE, Amsterdam, August 27–29, 2007.

6. S. Murtaza, A.G. Hoekstra, and P.M.A. Sloot, "Performance evaluation of FPGA-based Cellular Automata accelerators," in *Proceedings of the Third Annual Reconfigurable Systems Summer Institute (RSSI'07)*, (on line proceedings) +7. Reconfigurable Systems Summer Institute, National Center for Supercomputing Applications, Urbana, Illinois, July 17–20, 2007.