High Performance Reconfigurable Architectures for Bioinformatics and Computational Biology Applications

Server Kasap

A thesis submitted for the degree of Doctor of Philosophy.

The University of Edinburgh.

September 2010

Abstract

The field of Bioinformatics and Computational Biology (BCB), a relatively new discipline which spans the boundaries of Biology, Computer Science and Engineering, aims to develop systems that help organise, store, retrieve and analyse genomic and other biological information in a convenient and speedy way. This new discipline emerged mainly as a result of the Human Genome project which succeeded in transcribing the complete DNA sequence of the human genome, hence making it possible to address many problems which were impossible to even contemplate before, with a plethora of applications including disease diagnosis, drug engineering, bio-material engineering and genetic engineering of plants and animals; all with a real impact on the quality of the life of ordinary individuals.

Due to the sheer immensity of the data sets involved in BCB algorithms (often measured in tens/hundreds of Gigabytes) as well as their computation demands (often measured in Tera-Ops), high performance supercomputers and computer clusters have been used as implementation platforms for high performance BCB computing. However, the high cost as well as the lack of suitable programming interfaces for these platforms still impedes a wider undertaking of this technology in the BCB community. Moreover, with increased heat dissipation, supercomputers are now often augmented with special-purpose hardware (or ASICs) in order to speed up their operations while reducing their power dissipation. However, since ASICs are fully customised to implement particular tasks/algorithms, they suffer from increased development times, higher Non-Recurring-Engineering (NRE) costs, and inflexibility as they cannot be reused to implement tasks/algorithms other than those they have been designed to perform. On the other hand, Field Programmable Gate Arrays (FPGAs) have recently been proposed as a viable alternative implementation platform for BCB applications due to their flexible computing and memory architecture which gives them ASIC-like performance with the added programmability feature.

In order to counter the aforementioned limitations of both supercomputers and ASICs, this research proposes the use of state-of-the-art reprogrammable system-on-chip technology, in the form of platform FPGAs, as a relatively low cost, high performance and reprogrammable implementation platform for BCB applications. This research project aims to develop a sophisticated library of FPGA architectures for bio-sequence analysis, phylogenetic analysis, and molecular dynamics simulation.

Decleration of Originality

I hereby declare that the research reported in this thesis and the thesis itself was composed and originated entirely by myself in the School of Engineering at The University of Edinburgh.

Server Kasap

September 2010

To my family

To my loved ones

To my beautiful island

Acknowledgements

I would like to thank Dr. Khaled Benkrid and Professor Tughrul Arslan for their help and guidance during my research.

Thanks also to Dr. Ahmet Erdogan and to my lab colleagues for their support and valuable suggestions.

Contents

List of Figures

List of Tables

Acronyms and Abbreviations

A/D	Analogue/Digital
ASIC	Application Specific Integrated Circuit
BCB	Bioinformatics and Computational Biology
BLAST	Basic Local Alignment Search Tool
C	Coulombic
CAD	Computer-Aided Design
CPU	Central Processing Unit
DDR	Double Data Rate
DMA	Direct Memory Access
DSP	Digital Signal Processor
EDA	Electronic Design Automation
FHPCA	FPGA High Performance Computing Alliance
FIFO	First-In, First-Output
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GAML	Genetic algorithm for Maximum Likelihood
GPP	General Purpose Processor
HDL	Hardware Description Language
HLL	High Level Language
HPC	High Performance Computing
HTU	Hypothetical Taxonomic Unit
HW/SW	Hardware/Software
I/O	Input/Output
IC	Integrated Circuit
IOB	Input/Output Block
LAMMPS	Large-scale Atomic/Molecular Massively Parallel Simulation
LJ	Lennard-Jones
LUT	Look-Up Table
MAC	Media Access Controller
MAC	Multiply-Accumulate

MC	Monte Carlo
MD	Molecular Dynamics
MGT	Multigigabit Serial Transceiver
ML	Maximum Likelihood
MP	Maximum Parsimony
MPI	Message Passing Interface
NaN	Not a Number
NCBI	National Centre for Biotechnology Information
NRE	Non-Recurring Engineering
OTU	Operational Taxonomic Unit
PAUP	Phylogenetic Analysis Using Parsimony
PE	Processing Element
PLF	Phylogenetic Likelihood Function
PME	Particle-Mesh Ewald
PPPM	Particle-Particle Particle-Mesh
PSI-BLAST	Position Specific Iterated BLAST
PSSM	Position Specific Scoring Matrix
PTK	Parallel Toolkit
QDR	Quad Data Rate
RAM	Random Access Memory
RC	Reconfigurable Computing
SDRAM	Synchronous Dynamic RAM
SGE	Sun Grid Engine
SIMD	Single-Instruction, Multiple-Data
SoC	System on Chip
SoPC	System on a Programmable Chip
SRAM	Static RAM
UPGMA	Unweighted Pair Group Method with Arithmetic Means

Chapter 1

Introduction

1.1 Introduction and Motivation

The field of Bioinformatics and Computational Biology (BCB), a relatively new discipline which spans the boundaries of Biology, Computer Science and Engineering, aims to develop systems that help organise, store, retrieve and analyse genomic and other biological information in a correct and speedy way. Bioinformatics refers to the analysis and the management of biological information while the term computational biology is more often used to address physical and mathematical simulations of biological processes [1].

This new discipline emerged mainly as a result of the Human Genome project which succeeded in transcribing the complete DNA sequence of the human genome, hence making it possible to address many problems which were impossible to even contemplate before, with a plethora of applications including disease diagnosis, drug engineering, bio-material engineering and genetic engineering of plants and animals; all with a real impact on the quality of the life of ordinary individuals. However, the ample amount of genomic information has led to an absolute requirement for specialized BCB tools to analyze a huge volume of data in reasonable time frames.

Due to the sheer immensity of the data sets involved in BCB algorithms (often measured in tens/hundreds of Gigabytes) as well as their computation demands (often measured in Tera-Ops), high performance supercomputers and computer clusters have been used as implementation platforms for high performance BCB computing. However, the high cost as well as the lack of

suitable programming interfaces for these platforms still impedes a wider undertaking of this technology in the BCB community. Moreover, with increased heat dissipation, supercomputers are now often augmented with special-purpose hardware (or ASICs) in order to speed up their operations while reducing their power dissipation. However, since ASICs are fully customised to implement particular tasks/algorithms, they suffer from increased development times, higher Non-Recurring Engineering (NRE) costs, and inflexibility as they cannot be reused to implement tasks/algorithms other than those they have been designed to perform. On the other hand, Field Programmable Gate Arrays (FPGAs) have recently been proposed as a viable alternative implementation platform for BCB applications due to their flexible computing and memory architecture which gives them ASIC-like performance with the added programmability feature.

1.1.1 Genomic Data

Genomic data consist of DNA and protein sequences as well as RNA and gene expression profiles. DNA strings are sequences of *nucleotides* ranging over the 4-letters (i.e. A, C, G and T) alphabet which leads to a 2-bit encoding scheme that is well suited for FPGA hardware in comparison to a 32-bit or 64-bit Von Neumann computer architecture. Length of a DNA sequence varies from a few thousands characters (for a single gene) to several billions characters (for a complete genome). On the other hand, protein sequences are translated from DNA strings on 6 possible reading frames as illustrated in figure 1.1. Therefore, proteins are strings of characters over a generally used 20-letter (i.e. A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y) alphabet of *amino acids*. Their length ranges from a few tens to a few thousands characters.

DNA and protein sequences are stored, organized and indexed in genomic databases. Two well-known databases for protein sequences are SWISS-PROT

Figure 1.1: *Translation of a DNA string to protein strings according to the 6 different reading frames*

and Protein Data Bank while GenBank and EMBL are two examples of DNA databases. These databases containing millions of sequences exchange data on a daily basis and new releases are made every two months to incorporate new genomic data coming from worldwide research institutes. As these biological sequence banks are growing exponentially as illustrated in figure 1.2 for GenBank, performing computation on the growing mass of genomic data becomes more and more challenging [2].

1.2 Objective and Contributions

In order to counter the aforementioned limitations of both supercomputers and ASICs, this thesis proposes the use of state-of-the-art reprogrammable system-on-chip technology, in the form of platform FPGAs, as a relatively low cost, high performance and reprogrammable implementation platform for BCB applications. The research question in this thesis is to assess the viability of FPGAs as a high performance platform for BCB. This research project aims to develop a sophisticated library of FPGA architectures for a number of BCB applications. The applications that we will target are as follows:

- Bio-sequence Analysis
- Phylogenetic Analysis
- Molecular Dynamics Simulation

In BCB, biological sequence alignment is a very common task where subject sequences from a large database are aligned to a query sequence to find similarities between the query sequence and the database sequences [2]. Furthermore, phylogenetic analysis is the investigation of the evolution and relationships among organisms that is widely used in the fields of system biology and comparative genomics [3]. Moreover, Molecular Dynamics (MD) is a deterministic simulation technique often performed to help understand the properties of assemblies of molecules in terms of their structure and the microscopic interactions between them [4]. MD simulations act as a bridge between microscopic length and time scales and the macroscopic world of the laboratory, serving as a complement to conventional experiments.

The results of these case studies are studied to assess the efficacy and efficiency of FPGAs for implementing BCB algorithms. The main contributions of this thesis are summarized in the rest of this section.

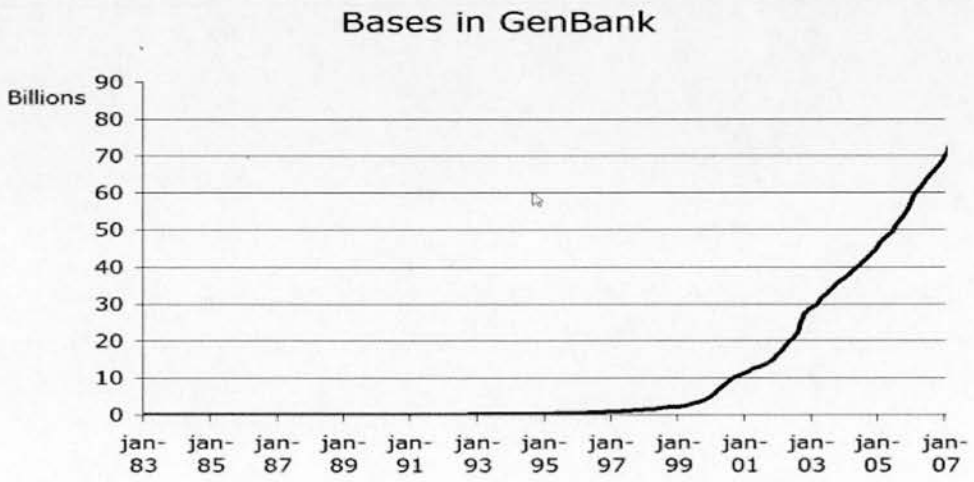


Figure 1.2: *Exponential growth of the GenBank database [11]*

1.2.1 Contributions

A major application of sequence alignment is to infer biological information about a newly discovered sequence from a set of previously annotated sequences, something which is extremely useful in early disease diagnosis and drug engineering [2]. However, sequence alignment is a computationally intensive operation, and with biological sequence databases growing at an exponential rate, desktop computers alone cannot be relied upon to perform this task within acceptable execution times. In the fourth chapter of the thesis, we hence present the first FPGA implementation of the Position Specific Iterated BLAST (PSI-BLAST) [5] which is a heuristic biological sequence alignment algorithm widely used by the BCB community in order to detect distant relationships among query and database sequences. The resulting implementation outperforms an equivalent desktop-based software implementation by at least one order-of magnitude.

Phylogenetic analysis is particularly important in drug and vaccine development [3]. In molecular-based phylogenetic analysis, the relationship between species is estimated by inferring the common history of their genes and then phylogenetic trees are constructed to illustrate evolutionary relationships among genes and organisms. However, phylogenetic tree construction is also a computationally intensive operation and the number of theoretically possible tree topologies grows exponentially with the number of species under consideration. In the fifth chapter of the thesis, the detailed design of a FPGA core for molecular-based phylogenetic analysis with Maximum Parsimony (MP) method [6] and its implementation on the nodes of an FPGA-based supercomputer named Maxwell is hence presented. The resulting implementation outperforms an equivalent desktop-based software implementation (i.e. PAUP) by very high orders-of magnitude.

Carrying out MD simulations of biomolecules provide a molecular picture of the structure and behavior of biological systems such as enzymes, proteins,

DNA strands and membranes. This allows scientists to advance their understanding of biologically important molecules. The MD method has applications in the fields of protein engineering [7], drug design [8] and refinements of structures based on X-ray [9] and NMR experiments [10]. However, biological systems of interest have sizes ranging from a few tens of thousands to millions of atoms and thus performing MD simulation of a biological process, such as protein folding, for a reasonable physical time requires enormous amounts of computational effort and may take years to complete on conventional computers. Therefore, it is mandatory to utilize faster computing platforms. In the sixth chapter of this thesis, we hence present the detailed design and implementation of a MD processor core on the Maxwell FPGA-based supercomputer. This FPGA core parallelises all the necessary operations to compute the non-bonded interactions in the Large-scale Atomic/Molecular Massively Parallel Simulation (LAMMPS) software tool. The timing performance figures of this MD core for the pairwise LJ and short-range Coulombic (via PPPM) interaction computations in the MD simulations of systems with various numbers of atom shows performance gains over the pure software implementation by factors of up to 13 on two nodes of the Maxwell machine

1.3 Thesis Structure

The structure of this thesis is set out as follows:

- Chapter 2 presents fundamentals and characteristics of reconfigurable computing. Furthermore, specific reconfigurable computing architectures are briefly outlined and reconfiguration technology is shortly described. Mapping algorithms to reconfigurable hardware is also introduced before providing various application fields for reconfigurable computing.

- Chapter 3 introduces an important form of reconfigurable logic (i.e. FPGA) that has been widely used in reconfigurable computing, and provides a brief overview of its basic architecture, programming the architecture and additional specialized function resources. Furthermore, an FPGA-based supercomputer named Maxwell is elaborately described towards the end of the chapter.

- Chapter 4 first presents essential background information on the general BLAST algorithm. Then, the design and implementation of a novel FPGA core for Position Specific Iterated BLAST is elaborated. Following this, implementation results are presented and then evaluated comparatively with the performance of equivalent software implementations running on a desktop computer.

- Chapter 5 first presents essential background information on phylogenetic analysis and discusses related prior works in the literature. Furthermore, the MP method for molecular based phylogenetic tree construction is detailed. Then, the design and implementation of a novel FPGA core for the MP method is elaborated. Following this, implementation results are presented and then evaluated comparatively with equivalent software implementations running on a desktop computer.

- Chapter 6 first presents essential background information on MD simulation and discusses related prior works in the literature. Furthermore, LAMMPS MD simulation software is introduced and the general system architecture is explained. Then, the design and implementation of a novel FPGA core for computing the non-bonded interactions in a MD simulation is elaborated. Following this, implementation results are presented and then evaluated comparatively with equivalent pure software implementations.

- Chapter 7 presents the summary and outlines the conclusions of this thesis. Several areas for potential future research are also suggested.

- Appendix A presents the journal papers published and awaiting acceptance as a result of the work carried out in completion of this thesis.

- Appendix B presents the conference papers published and awaiting acceptance as a result of the work carried out in completion of this thesis.

Chapter 2

An Introduction to Reconfigurable Computing: Accelerating Computation with FPGAs

2.1 Introduction

Reconfigurable Computing (RC), the use of programmable logic to accelerate computation, is emerging as a new computing paradigm with the commercial availability of reconfigurable logic [1]. Simply, reconfigurable logic is a special kind of hardware circuit which can be reconfigured into whatever logic the user desires by programming some kind of configuration memory.

Reconfigurable computing is becoming increasingly attractive for many applications since it utilizes hardware that can be adapted for specific computations in each application without compromising performance and hence promises an intermediate trade-off between performance and flexibility. Reconfigurable architectures can exploit fine-grained and coarse-grained parallelism available in applications because of the adaptability feature, thus providing significant performance advantages compared to general purpose architectures, such as microprocessors, particularly for high performance, low volume applications that incorporate inherent instruction-level parallelism. Other advantages of reconfigurable computing include reduced power consumption, improved time-to-market, improved upgradability and reduction in size and hardware count, and hence in cost.

In the rest of this chapter, fundamentals and characteristics of reconfigurable computing is first presented. Then, specific reconfigurable computing architectures are briefly outlined and reconfiguration technology is shortly

described. Mapping algorithms to reconfigurable hardware is also introduced before providing various application fields for reconfigurable computing. Finally, concluding remarks will be presented.

2.2 Fundamentals of Reconfigurable Computing

Reconfigurable computing systems contain coupled microprocessors and reconfigurable devices, such as *Field Programmable Gate Arrays* (FPGAs), where reconfigurable devices are used as coprocessors in the design that are deployed to execute the small portion of the application taking possibly the most of the time for the purpose of acceleration. Obviously, FPGAs can achieve this task only when computations lend themselves to implementation in hardware, subject to the limitations of the currently utilized FPGA chip architecture and the specified data transfer constraints.

A reconfigurable computing architecture composed of a *General-Purpose Processor* (GPP) and some reconfigurable hardware logic is illustrated in figure 2.1 adapted from [12]. In this architecture, the reconfigurable hardware logic executes application-specific, computation intensive, such as the encryption task (C_1) and the image processing task (C_5), while the GPP is used to control the behaviour of these tasks running in the reconfigurable hardware logic and to coordinate external Input/Output (I/O) communications between two processing units. When reconfigurable hardware logic has finished the computation of its task, such as the encryption task C_1, the processor reconfigures that hardware logic to execute another task, such as the signal processing task C_2, as shown in figure 2.1. Furthermore, during this reconfiguration process, the image processing task C_5 continues to execute without any interruption. Note that the reconfigurable computing architecture can also be defined as *Hardware-On-Demand* or general-purpose custom hardware [12].

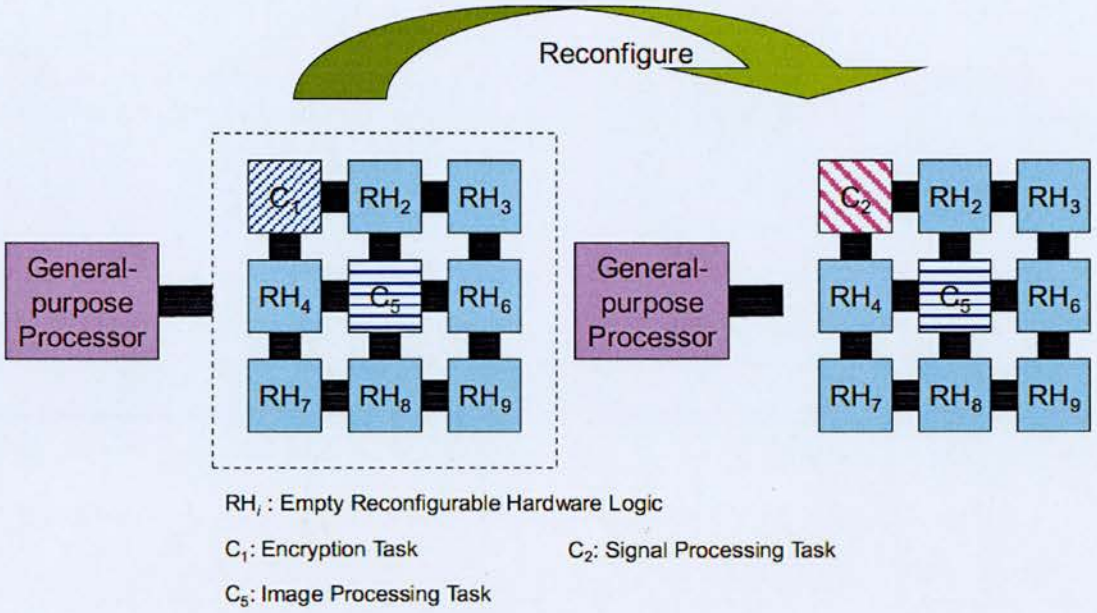


Figure 2.1: *Reconfigurable Computing [12]*

2.3 Characteristics of Reconfigurable Computing

Computing was traditionally classified into general-purpose computing performed by a GPP and application-specific computing performed by an *Application-Specific Integrated Circuit* (ASIC) [12].

A general-purpose computer is a single *Integrated Circuit* (IC), called a microprocessor or GPP, that provide a flexible computing platform by being able to execute a large class of applications with its fixed function components. The same fixed hardware can be used for many applications since they are executed by decoding a stream of instructions from software and operating on data stored in memory. However, the performance achievable by microprocessors is severely limited by the sequential instruction decoding and execution, memory access bottleneck and fixed control architecture.

An ASIC is an IC which integrates several functions or logic control block for a specific application into one single chip. Hence, each ASIC has a fixed

functionality and high performance for a restricted set of applications. However, fixed resource and algorithm architecture of ASICs restrict the flexibility and exclude any upgrades in features and algorithms.

On the other hand, reconfigurable computing has the advantages of both computing systems. In reconfigurable computing, the functions of a system can be altered by configuring a fixed set of logic resources through memory settings, where the fixed set of logic resources include logic blocks, I/O blocks, routing blocks and application-specific blocks, and memory settings can be achieved by the programming of configuration bits that control the functions of these logic resources [12]. The table below compares the characteristics of the three mentioned computing systems.

In table 2.1, it can be observed that reconfigurable computing benefits from both configurable computing resources (called *configware*) and configurable algorithms (called *flowware*). Furthermore, the performance of reconfigurable computing systems is better that general-purpose systems and the cost is smaller than that of ASICs, whereas reconfigurable systems consume more power than ASICs. Moreover, high flexibility is the main advantage of the reconfigurable system while lacking a mature computing model is its main disadvantage. Finally, the Non-Recurring Engineering (NRE) cost of reconfigurable systems which represent the design effort is between that of general-purpose computers and ASICs.

Table 2.1: *Comparison of Computing Architectures [12]*

	General Purpose	ASIC	Reconfigurable Computing
Resources	Fixed	Fixed	Configware
Algorithms	Software	Fixed	Flowware
Performance	Low	High	Medium
Cost	Low	High	Medium
Power	Medium	Low	Medium

Flexibility	High	Low	High
Computing Model	Mature	Mature	Immature
NRE Cost	Low	High	Medium

Therefore, it can be concluded that reconfigurable computing is a trade-off between general-purpose computing and application-specific computing since it aims to achieve a balance among performance, flexibility, cost, power and design effort. Due to the mentioned characteristics of reconfigurable computing, it has been used widely for several years to enhance the performance of many applications in a large variety of domains, as will be described in subsection 2.7.

2.4 Reconfigurable Computing System Architectures

Figure 2.2 adapted from [1] shows that FPGAs form the processing building blocks of reconfigurable computers. Most reconfigurable computing systems are constructed by plugging an accelerator board into the I/O slot of a microprocessor, where the plug-in board typically contains:

- One or more FPGAs,
- Interface logic (i.e. IF) for the FPGAs to communicate with the microprocessor's I/O bus,
- Memory local to the reconfigurable computing board such as *Static Random Access Memory* (SRAM) and/or *Synchronous Dynamic RAM* (SDRAM) with Double Data Rate (DDR) or Quad Data Rate (QDR),
- Analogue/Digital (A/D) converters or serial communication interface to acquire data or communicate over a network.

The reconfigurable computing system can be used in two scenarios to accelerate a computation. In the first scenario, the host sends data to the reconfigurable computing memory subsystem, and then the FPGAs perform compute-intensive operations on the on-board data and write results back into the local memory, which are then retrieved by the host. Another option in this scenario is that FPGAs process data streams acquired externally from the serial I/O interface with high bandwidth capability, and then send processed data to the host processor after some data reduction operations [1].

In the second scenario, reconfigurable system is used as an acceleration component of a supercomputer, where the I/O interface on the FPGA board is used to access to the interconnection network of the supercomputer over which an FPGA communicates with processors and other FPGAs.

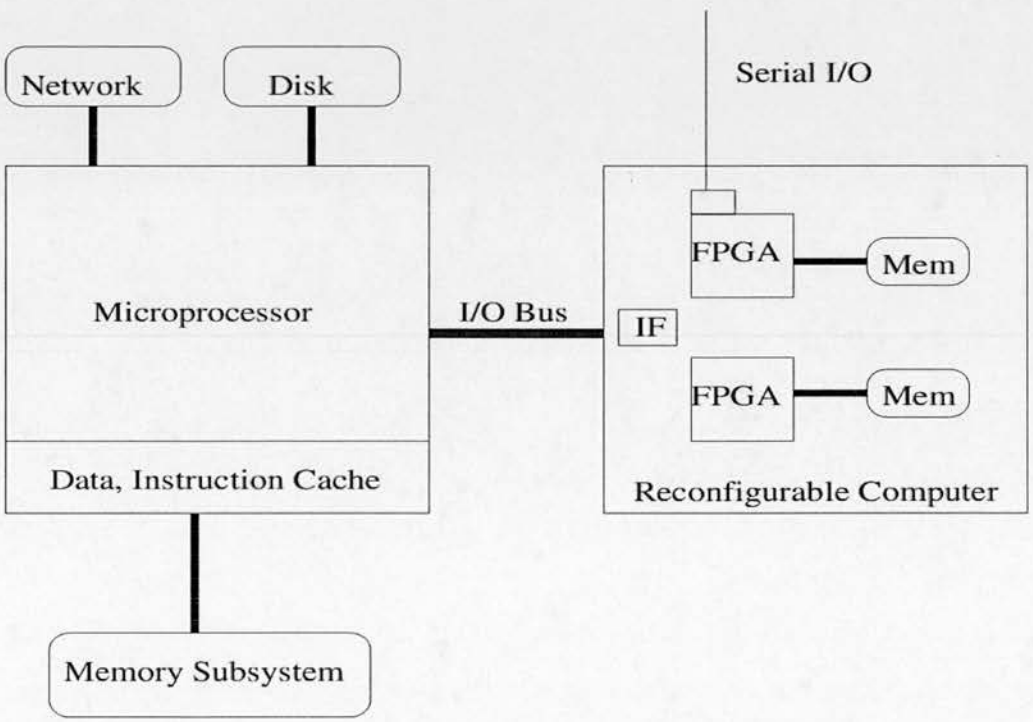


Figure 2.2: *Reconfigurable computing system [1]*

Typically, a reconfigurable system is made up of one or more processors, one or more reconfigurable fabrics, and one or more memories. Reconfigurable

systems are often classified according to the extent the reconfigurable fabric and the *Central Processing Unit* (CPU) are coupled with each other [13]. Five such classifications are presented in figure 2.3.

In figure 2.3 (a), the reconfigurable fabric is in the form of one or more standalone devices, where the processor use its input and output mechanisms to communicate with the reconfigurable fabric. Since the data transfer between the processor and the fabric is relatively slow in this configuration, it only makes sense to use this architecture when the fabric can do a significant amount of processing without requiring any processor intervention. However, it is by far the most commonly used reconfigurable system architecture. On the other hand, figure 2.3 (b) and figure 2.3 (c) show two structures where the reconfigurable fabrics are used in the forms of attached processing unit and co-processor, respectively. Both of these architectures have a lower communication cost compared to the one shown in figure 2.3 (a).

Furthermore, the processor and the fabric is very tightly coupled in the architecture presented in figure 2.3 (d), where the reconfigurable fabric is part of the processor itself to form a reconfigurable functional unit that allows for the creation of custom instructions for the processor. Finally, in figure 2.3 (e), the processor is embedded in the fabric either as a 'hard', or 'soft' core which is implemented using the resources of the reconfigurable fabric itself.

In all of these architectures, the FPGAs serve as coprocessors to the microprocessors where the main application runs on the microprocessors, while the FPGA handle kernels that have a long execution time. These kernels typically incorporate data-parallel overlapped computations that can be efficiently implemented in hardware as fine-grained architectures, such as Single-Instruction, Multiple-Data (SIMD) engines, pipelines and systolic arrays [14].

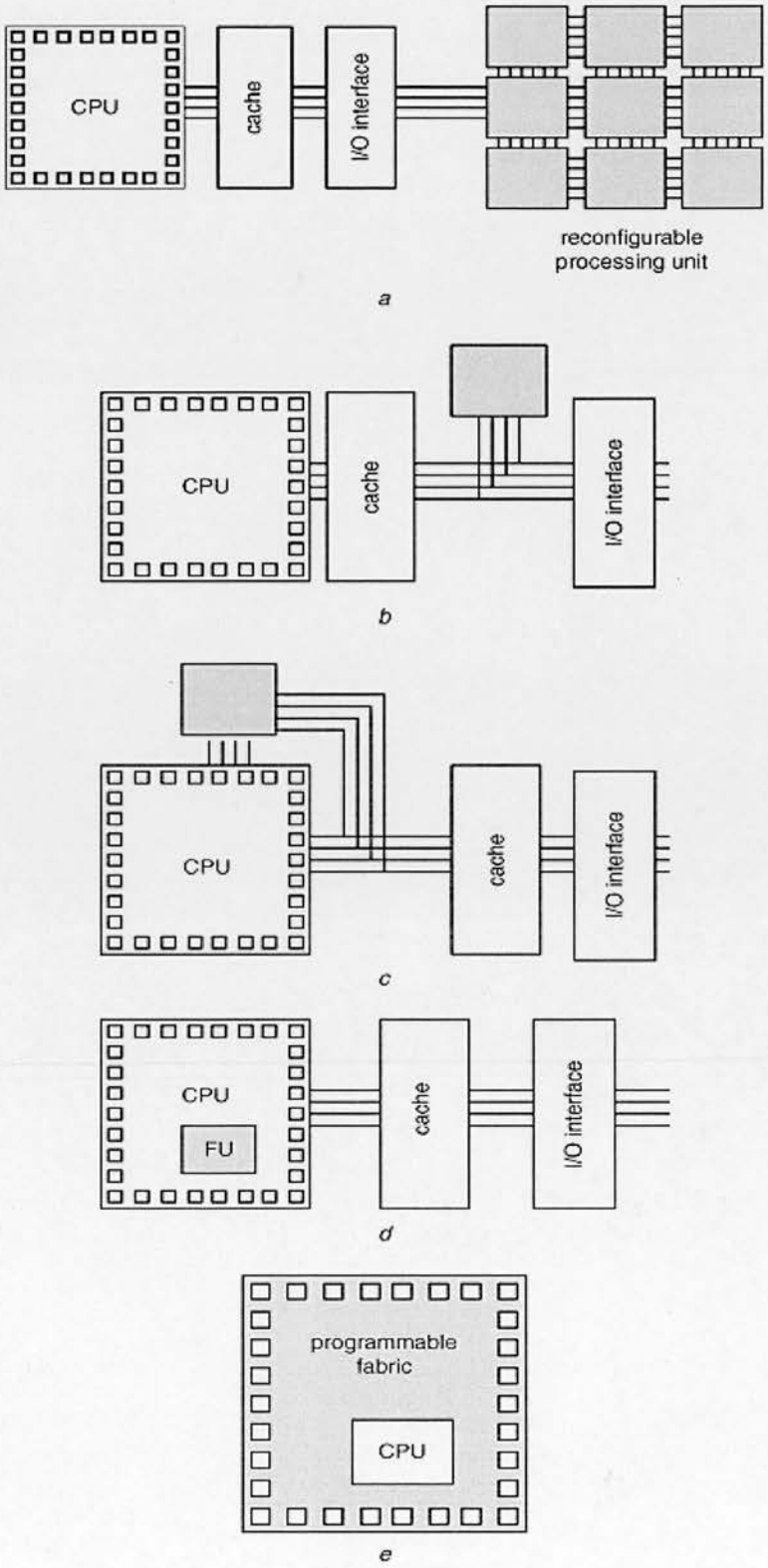


Figure 2.3: *Five classes of reconfigurable computing systems [13]*

2.5 Reconfiguration Technology

Reconfigurable computing has become a popular field due to the advent of FPGAs in the 1980s, which are integrated circuits containing programmable logic components and programmable interconnects. Hardware personality of an FPGA, whose internal structure based on the Xilinx style is shown in figure 2.4, can be completely redefined simply by loading a new configuration analogous to loading new software modules onto a microprocessor or *Digital Signal Processor* (DSP). Hence, an FPGA chip can be reprogrammed to perform a different function unlike ASICs that perform a single specific function for the lifetime of a chip.

The FPGA is a regularly tiled two-dimensional array of logic blocks which can be programmed to duplicate the functionality of basic logic gates or functional intellectual properties. Furthermore, the logic blocks communicate through a programmable interconnection network. FPGAs also include memory elements composed of simple flip-flops or more complete blocks of memories [12]. A detailed description of FPGA will be presented in section 3.

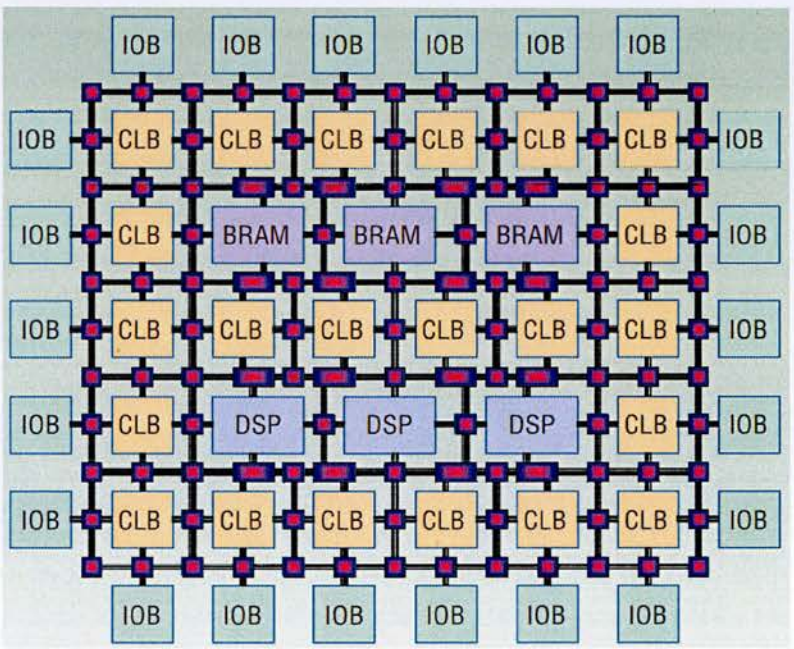


Figure 2.4: *FPGA internal structure based on the Xilinx style [15]*

2.6 Mapping Algorithms to Hardware

Mapping an algorithm onto a collection of configurable logic blocks is a complex task where Hardware Description Languages (HDLs), High Level Languages (HLLs) or schematic entry tools are commonly used to create reconfigurable computing configurations. However, compiling an algorithmic description (either in HDL or HLL) to a reconfigurable device, such as an FPGA, necessitates a long tool chain whose ultimate output, the configuration bitstream, may take hours to be generated.

Designers need Computer-Aided Design (CAD) tools to construct a reconfigurable computing system. For instance, a design analysis tool for architecture design, a synthesis tool for hardware construction, a simulator for hardware behaviour simulation, and a placement and routing tool for circuit layout are required for system design and implementation [12]. Commonly used commercial FPGA and Electronic Design Automation (EDA) tools with various functionalities from different vendors are listed in table 2.2.

Table 2.2: *Commercial Reconfiguration Tools [12]*

Functionality	Tool Name	FPGA/EDA Company
Design Analysis	PlanAhead	Xilinx
FPGA Suite Tools	ISE Foundation	Xilinx
	Quartus	Altera
	FPGA Advantage	Mentor Graphics
FPGA Synthesizer	Synplify Pro	Synplicity
	FPGA Compiler	Synopsys
	Leonardo Spectrum	Mentor Graphics
	Precision Synthesis	Mentor Graphics
Simulator	ModelSim	Mentor Graphics
	NC SIM	Cadence
	Scirocco Simulator	Cadence
	Spexsim	Verisity

	VCS	Synopsys
	Verilog-XL	Cadence

Figure 2.5 illustrates the typical FPGA design flow where the first step is a synthesis process that generates a technology-mapped netlist from HDL descriptions. Then, FPGA-specific tools map these gate-level descriptions onto configurable logic blocks and routes, and perform placing and routing to produce the binary configuration bitstream which can be used to reconfigure the FPGA chip. Furthermore, simulations, timing analyses and other verification methodologies are utilized to validate the results of synthesis, mapping, placing and routing, as can be observed in figure 2.5.

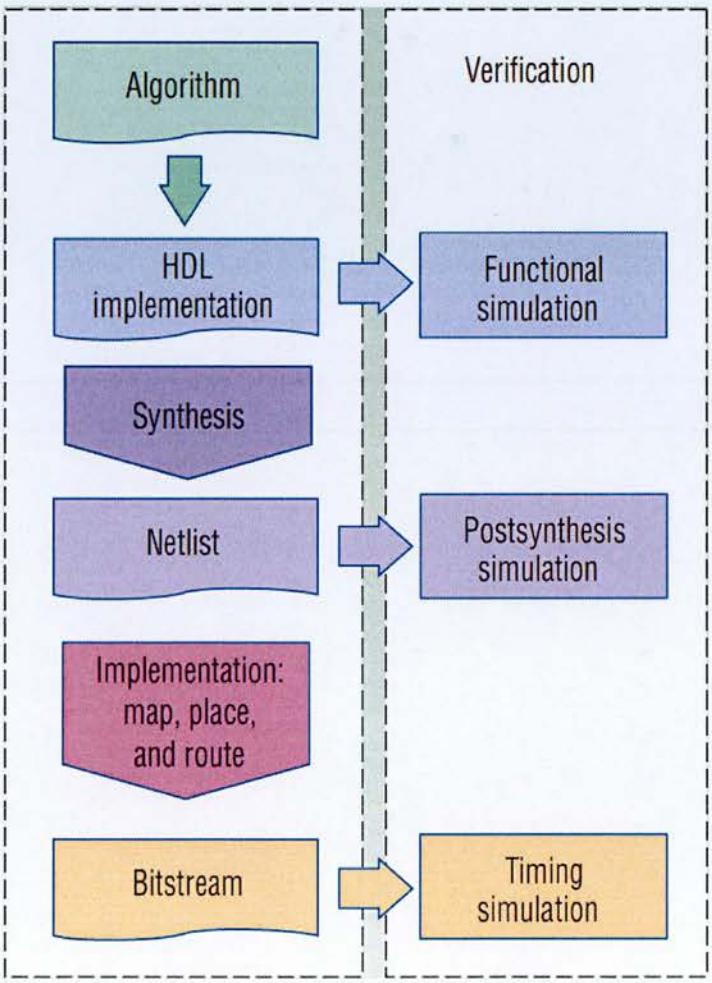


Figure 2.5: *Typical FPGA design flow [15]*

2.7 Reconfigurable Computing Applications

Although there are several challenges associated with converting algorithms into hardware, reconfigurable computing has flourished as a disciple and has been widely applied in a large variety of domains. Reconfigurable computing has enhanced the performance in the following application fields that are characterized by heavy computation tasks and large amounts of data required to be processed, with the additional need for adaptation to the dynamical requirements of the data streams and algorithms [12]:

- Embedded Systems [16]
- System on Chips (SoCs) [17]
- Digital Signal Processing [18]
- Image Processing [19]
- Network Security [20]
- Bioinformatics and Computational Biology [21]
- Supercomputing [22]
- Cryptanalysis [23]
- Boolean Satisfiability problem [24]
- Spacecrafts and Military applications [25]

2.8 Summary

In this chapter, we discussed fundamentals and characteristics of reconfigurable computing. Then, specific reconfigurable computing

architectures and reconfiguration technology were introduced. Later, mapping algorithms to reconfigurable hardware was explained. We concluded this chapter by presenting various application fields for reconfigurable computing.

Chapter 3

Reconfigurable Logic Devices and the Maxwell FPGA-based Supercomputer

3.1 Introduction

Although reconfigurable computing was conceived as early as 1960, the recent developments in reconfigurable computing were made possible by the availability of logic devices which can be easily programmed to perform a large variety of functions [1]. FPGAs were the first significantly available field-programmable devices that achieved enough density to perform significant portions of a computation. Arrays of simple logic functions and memories (e.g. flip-flops) which can be connected through programmable interconnection networks are provided in these chips for the designer.

In the beginning, the FPGA devices from companies such as Xilinx, Altera and Actel provided relatively little logic, however later generations provided enough logic to enable the direct implementation of many computational algorithms in reconfigurable logic devices. Today's FPGAs have reached so high logic densities that they have developed into devices which can be used to build complete Systems on a Programmable Chip (SoPCs) with the provision of specialized function blocks such as embedded SRAM blocks of various sizes, Digital Signal Processing (DSP) blocks, multi-gigabit serial I/O units and embedded microprocessors.

The goal of this chapter is to introduce an important form of reconfigurable logic (i.e. FPGA) that has been widely used in reconfigurable computing, and provide a brief overview of its basic architecture, programming the architecture and additional specialized function resources. Furthermore, an

FPGA-based supercomputer named Maxwell will be elaborately described towards the end of the chapter. Finally, concluding remarks will be presented.

3.2 Field-Programmable Gate Arrays

FPGAs are a family of silicon devices intended for custom hardware implementation with the key property of being reconfigurable for an infinite number of times. Reconfiguring an FPGA means changing its functionality to support a new application or mapping some new piece of hardware, with a new functionality, onto the FPGA chip. This is to say that FPGAs make it possible to have custom-designed, high-density hardware in an integrated circuit, with the added advantage of having the possibility of changing it whenever needed, even while the entire application is still running. Furthermore, the functionality to be implemented on the device is described by a Hardware Description Language (HDL) such as Verilog or VHDL, then software tools provided by the device manufacturer translates this description of the hardware functionality into a configuration file for the device to be downloaded on it.

The flexibility of having both custom and changeable hardware has made FPGA devices popular in a broad range of application fields. For instance, if the FPGA is coupled with a general purpose processor, the most demanding sections of the software can be ported into hardware cores that can notably accelerate the program execution, especially when software sections executed serially on the processor can be ported into hardware that can exploit the parallelism inherently available in the algorithm. This is a reasonable technique in high performance computing, where software kernels taking a long execution time are implemented on an FPGA, leaving the rest of the program to execute on a general purpose processor. For this purpose, the

original program is first profiled to locate its computationally intensive functions which are then translated into hardware using an HDL, and finally communication issues between the processor and the hardware on FPGA are taken into consideration with the goal of eliminating any possible communication bottlenecks [12]. The result is a significant improvement in the execution time of the algorithm. This methodology, commonly referred to as *hardware-software codesign*, is widely applied whenever there is a need for accelerating software running over large sets of data.

3.3 FPGA Architecture

This section will provide a brief overview of FPGA architecture and its distinctive features to clarify how they can implement custom hardware via their reconfiguration. The basic architecture of FPGAs consists of three kinds of components: logic blocks, input/output blocks, and routing resources. The architecture of a generic FPGA is illustrated in figure 3.1 adapted from [12]. As it can be seen, FPGAs incorporate an array of programmable logic blocks that can be interconnected to each other as well as to the programmable I/O blocks through a programmable routing matrix. Furthermore, FPGAs have fine-grained architecture which means that the logic operations are mainly done at the bit or small word level [1]. In other words, FPGAs have a very flexible architecture that can be customized to the very specific needs of an application. For instance, if an application needs a 24-bit adder for a particular operation and a 8-bit adder for another, then the designer can directly implement adders at the desired size rather than use a fixed size (e.g. 32-bit) adder to perform these different precision computations. However, this flexibility have significant costs associated with it in terms of both silicon area and circuit speed as compared to non-programmable, custom silicon implementations due to the large number of transistors and the large amount of wiring needed to provide the fine-grained programmability.

The next three subsections will respectively describe the three main building blocks of an FPGA and provide some typical examples of how they are constructed.

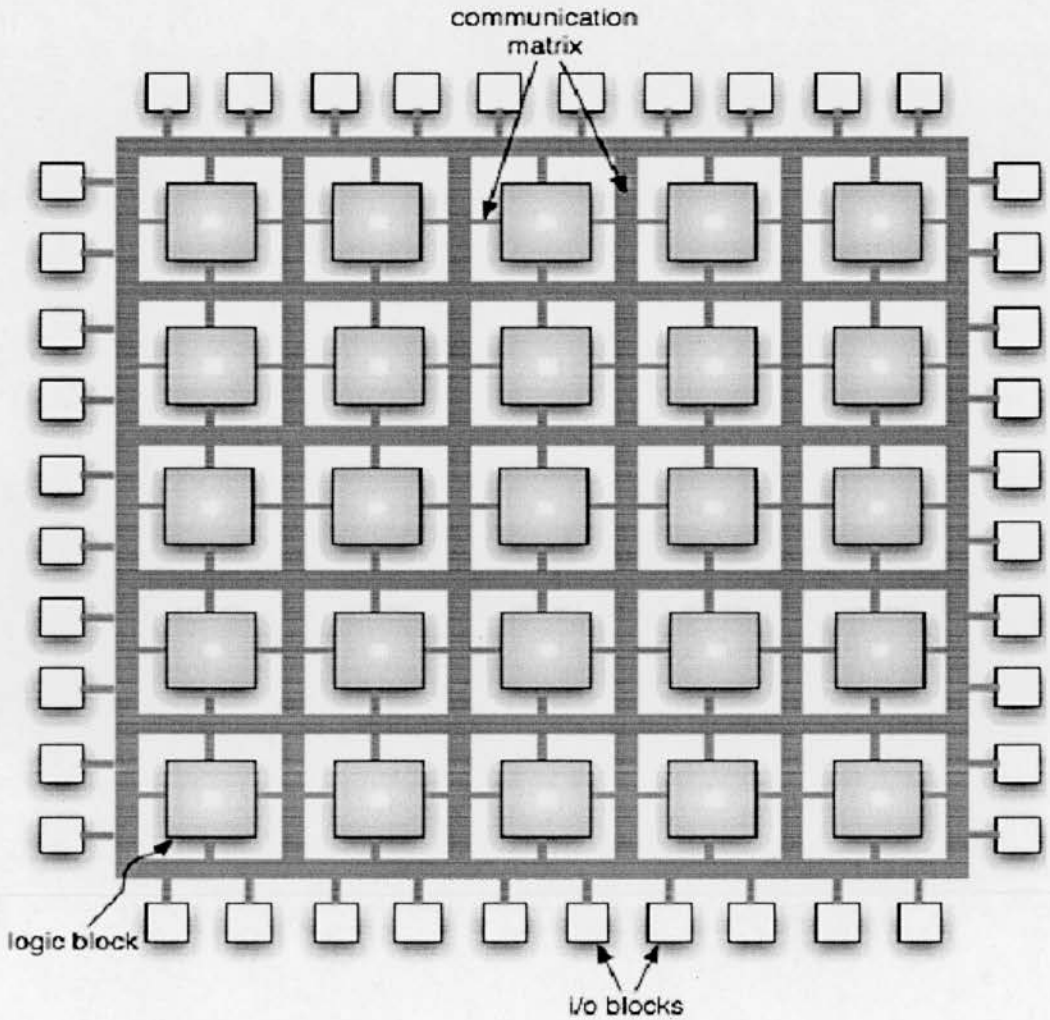


Figure 3.1: *A generic FPGA architecture [12]*

3.3.1 Programmable Logic Blocks

Programmable logic blocks which can be generalized as shown in figure 3.2 are the main components of an FPGA. They generally contain some form of programmable logic, a flip-flop and some fast carry logic to reduce the delay and area costs for implementing carry logic, as illustrated in figure 3.2. The

output of the programmable logic block is selectable between the output of the flip-flop and the output of the combinational logic via a multiplexer which is controlled by some form of configuration memory. Furthermore, configuration memory is used throughout the logic block to control the specific function of each element within the block.

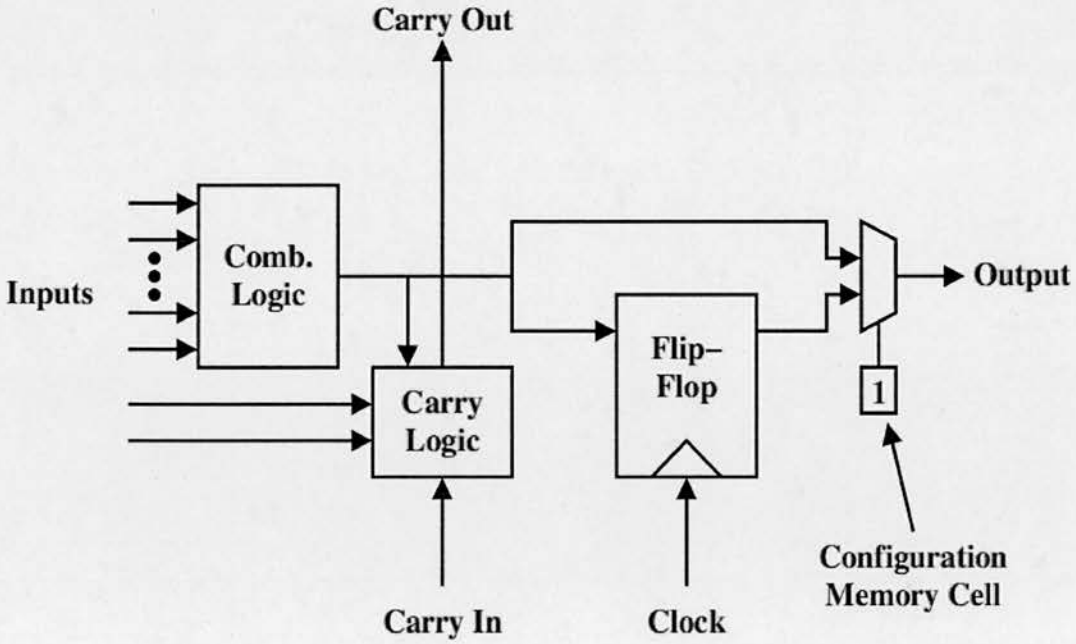


Figure 3.2: *A generic programmable logic block [1]*

The combinational logic portion of the logic block is most commonly realized by a Look-Up Table (LUT) that can implement an arbitrary logic function according to its configuration. The result of the function is stored for every possible combination of the inputs in these devices, for instance a 4-bit LUT will require 16 memory cells to store the function irrespective of its complexity. Therefore, the memory inside the look-up tables is written during the configuration process of an FPGA to implement a desired function.

Many FPGA architectures have the logic blocks clustered together using short-length and fast routing to reduce the delay costs of using

programmable routing, allowing to create larger functions using only the faster routing of the cluster. For this purpose, most recent LUT-based architectures often pair two or more logic blocks into a cluster. Due to the increasing costs of general-purpose programmable routing, the size of these clusters has been gradually augmented to improve the circuit performance.

3.3.2 Programmable I/O Blocks

Input/Output Blocks (IOBs) have the function of interconnecting the signals of the internal logic to a pin of the FPGA package where there is a unique IOB for every I/O pin of the chip package. As shown in figure 3.3, the IOBs contain input buffers for the inputs and tri-state buffers for the outputs. Within the IOB, the input signal, the output signal and the tri-state enable signal can be individually registered or can be left unregistered. Furthermore, the IOBs have their own configuration memory that stores the voltage standards to which the pin should comply, and configures the direction of the communication on it, such that mono-directional links in either way or bidirectional ones can be established efficiently.

3.3.3 Programmable Routing

The programmable routing resources within an FPGA allow the arbitrary connection of logic blocks and I/O blocks. Various forms of routing exist throughout FPGA architecture. While some amount of routing is included within each logic cluster to form larger functions by combining the logic blocks, there is also the global routing architecture of the FPGA external to the logic clusters.

Routing within a logic cluster is used for several purposes. First of all, it determines where the inputs to the logic blocks come from and where the outputs will go. Programmable routing within a cluster also determines how

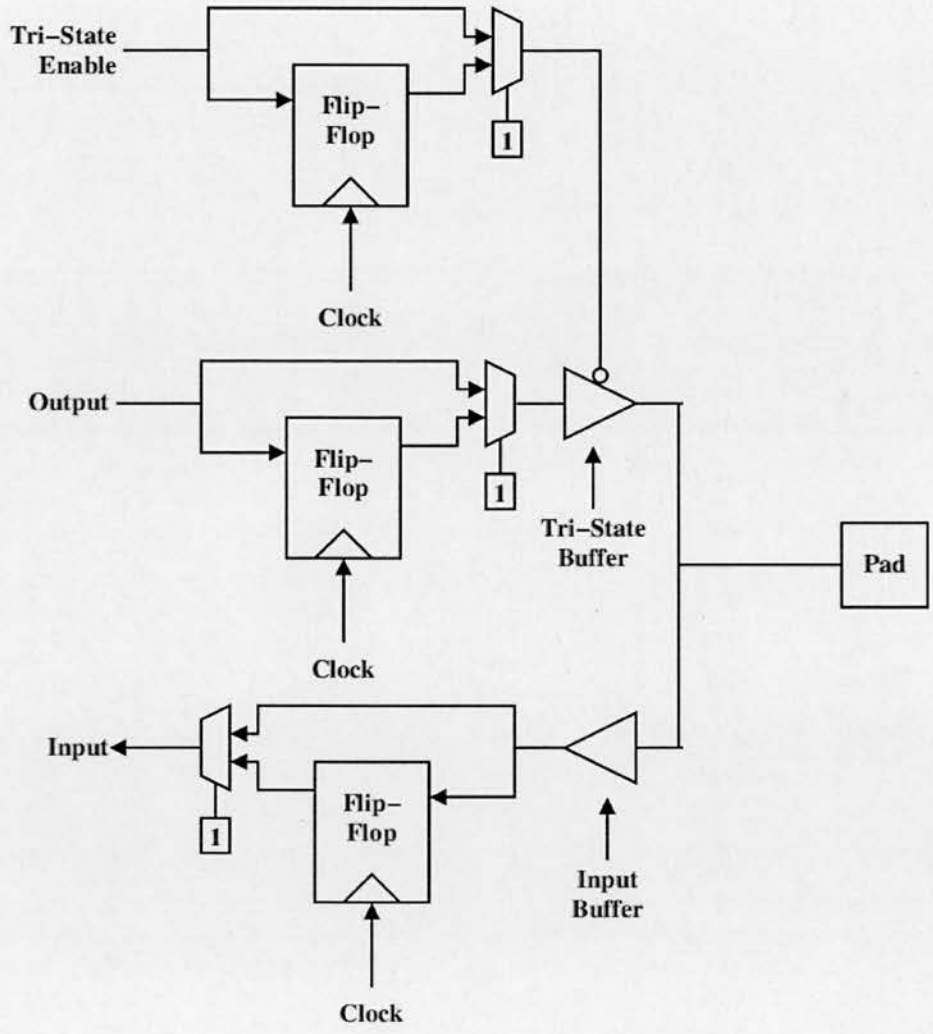


Figure 3.3: *I/O block architecture [1]*

signals propagate through the logic blocks. Furthermore, there exists non-programmable routing which is used for fast carry propagation within the cluster, and extends between clusters to enable wide additions. Finally, the logic blocks can be combined into wider functions by routing within the cluster. All sorts of the internal cluster routing are illustrated in figure 3.4.

There exist several global routing architectures implemented in FPGAs which can be categorized as the island, cellular, long-line and row architectures [1]. The island routing architecture is widely utilized in Xilinx FPGA chips. Hence, this architecture will be briefly described in the rest of this subsection.

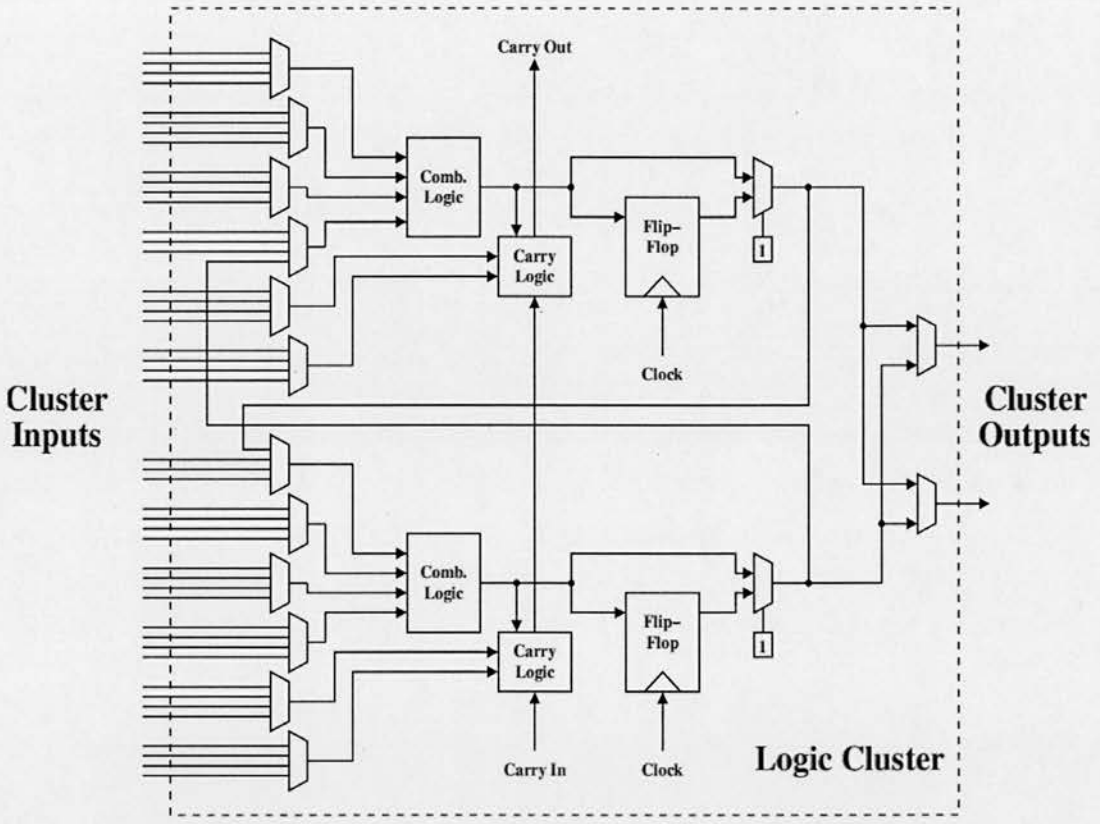


Figure 3.4: *Internal logic cluster routing [1]*

The basic island-style routing architecture is illustrated in figure 3.5. As can be seen, logic clusters are surrounded by segmented both horizontal and vertical, routing channels in this architecture where each cluster connects to the routing through connection boxes and each segment in the routing can be connected to another segment through a switch box. In this type of routing, connections between logic clusters are made through segments in several lengths while local routing between logic clusters are also provided to make the architecture more efficient. Segmented routing is based on lines that can be interconnected using programmable switch matrices. In this kind of routing, there are also wires that cross the entire chip in order to maximize the speed of communication and limit signal skew. Segmented routing offer a reduced power dissipation since resistance and capacity of the interconnection wires are only dependent on the interconnection length between the clusters [12].

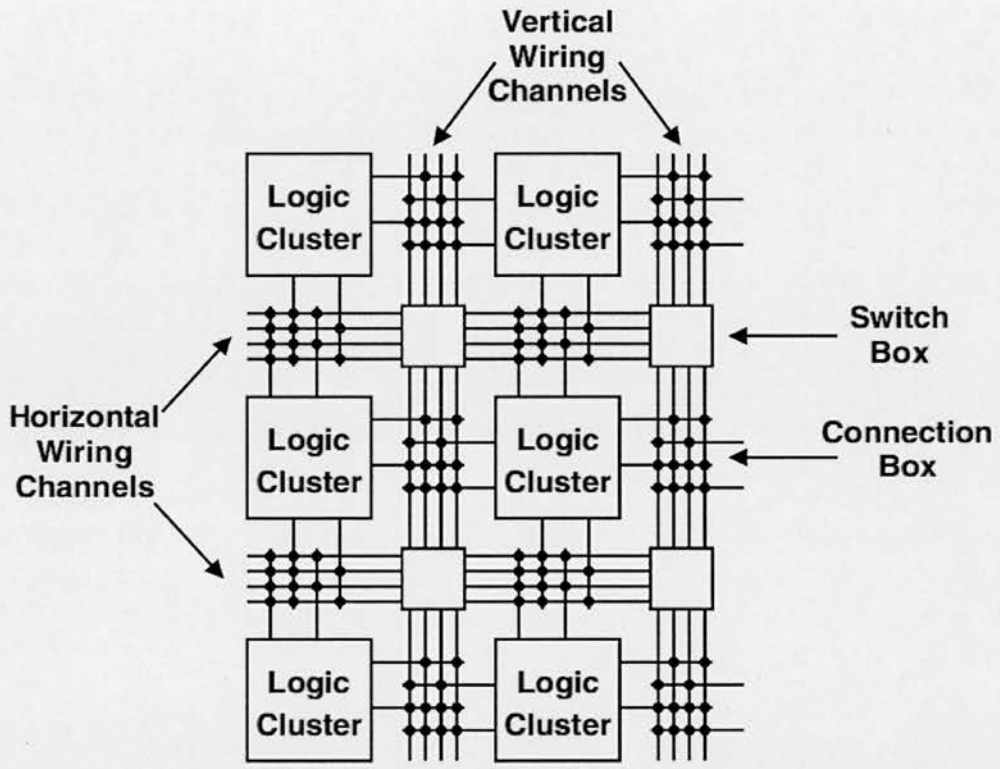


Figure 3.5: *Island routing architecture [1]*

3.4 FPGA Programming

The tree building blocks presented so far interconnect together in the device to create a structure composed of the communication wires and IOBs around a bi-directional array of logic clusters which covers most of the FPGA chip area. The key features of every configurable resource in this structure is controlled by the memory cells attached to them, such that the interconnections among the communication infrastructure are controlled by setting the appropriate bits in the configuration memory, the I/O voltage standards of a IOB are controlled in accordance with the value in its corresponding memory cell, and the functions implemented by LUTs are controlled in the same way. The mentioned configuration memory is made up of SRAM memory elements which are volatile, meaning that an FPGA device loses its configuration when its power is turned off. Generally, an

external machine such as a host processor downloads the configuration on the FPGA via one of the configuration interfaces and then sends a start command to signal that the configuration has been completed. Furthermore, some FPGA boards also have a ROM to store the configuration which can be subsequently downloaded on the FPGA on power up. The file storing the information to be copied over the configuration SRAM memory of the FPGA is called *bitstream* which can be either full or partial depending on the extent of configuration memory addressed in it.

3.5 Additional Resources

Most FPGAs are not composed only of the three components described above, but they have additional programmable resources directly embedded on the die. These resources such as embedded memory, arithmetic units, high-speed serial I/O units and embedded processors have been added due to a frequent need for such resources in FPGA applications. Designers can integrate these resources in their systems so that the need of having to implement all of the desired functionalities on the configurable logic resources is eliminated with the added bonus of enriched functionalities and high speed achievable by pre-made embedded hardware cores. The result is that many recent FPGAs have a more heterogeneous structure than early FPGAs. In the following subsections, the additional resources that have been made available in recent FPGAs will be briefly described one by one.

3.5.1 Embedded Memory

Memory is the basic component of most digital systems. Although flip-flops can be used for memory, they are very inefficient for creating memory of any depth. In Xilinx FPGAs, the LUTs used for logic can operate as synchronous RAMs and dual-ported RAMs. LUTs are better than flip-flops for

implementing deep memories, however most FPGA vendors now include more dense blocks of SRAM that have from hundreds to thousands of bits within the architectures. Generally, the aspect ratio of an embedded RAM can be programmed to make it operate in different modes such as 4096x1, 2048x2, 1024x4 or 512x8 where the aspect ratios are given as depth x width in bits. Furthermore, embedded RAMs in some FPGAs can operate as dual-ported RAMs and First-In, First-Output (FIFO) buffers.

The key benefits of these on-chip memories is the large number of memory ports made available and the aggregate memory bandwidth that is high enough to provide a significant advantage to very parallel applications requiring significant memory bandwidth.

3.5.2 Embedded Arithmetic Units

Many FPGAs have started to include 18x18 multipliers or digital signal processing (DSP) blocks as separate resources in addition to the basic carry logic and adders provided in the logic clusters. The DSP blocks which have a high degree of configurability provide addition/subtraction, multiplication and Multiply-Accumulate (MAC) operations which are very useful for many DSP applications.

3.5.3 High-Speed Serial I/O Units

Since many FPGAs are used in high throughput telecommunication equipments, multi-gigabit serial transceivers are recently added to their architecture as I/O units. These units can perform full-duplex serialization/deserialization functions, encoding/decoding functions and error control. Furthermore, some FPGAs have a number of dedicated Ethernet Media Access Controllers (MACs) to provide a complete solution for Ethernet serial I/O communications.

3.5.4 Embedded Microprocessors

Dedicated microprocessors have also been integrated with the FPGA logic to perform control-intensive functions, so complete embedded system can now be implemented with a single device. For instance, Xilinx produced FPGAs with integrated PowerPC microprocessors which are placed as an island within the FPGA logic, as illustrated in figure 3.6. It can be seen that the processor interfaces to on-chip RAM, but there exists no dedicated processor or peripheral buses which must be implemented using FPGA logic, if desired. This gives the flexibility to define the entire architecture of the embedded system. However, the drawback is that no useful work can be done with the processor if the FPGA logic is not configured in a way.

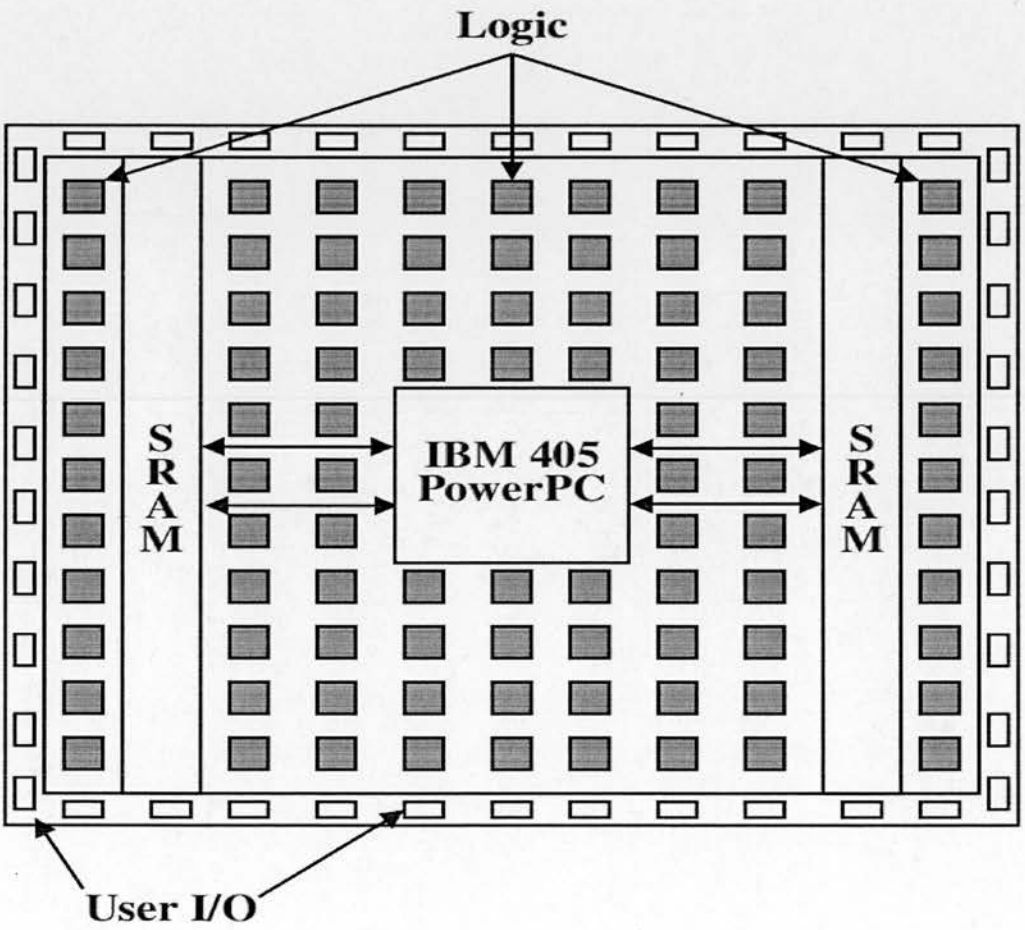


Figure 3.6: *Xilinx style processor integration [1]*

3.6 The Maxwell Supercomputer

Maxwell [26] is an FPGA-based supercomputer developed by the FPGA High Performance Computing Alliance (FHPCA) [27] in Scotland to run computationally demanding applications on an array of FPGAs at low energy budgets. The alliance partners are Algotronix, Alpha Data, Nallatech, the Institute for System Level Integration, Xilinx and EPCC at the University of Edinburgh. The two main goals of the Alliance were to design and build a 64-FPGA supercomputer from commodity parts and plug-in FPGA cards, and to demonstrate its effectiveness for real-world High Performance Computing (HPC) applications. The physical architecture, topology, logical structure and software environment of the Maxwell supercomputer are discussed in subsections 3.6.1, 3.6.2, 3.6.3 and 3.6.4, respectively. Note that there are only a few other FPGA-based supercomputers (e.g. Cray XD1 [33], SGI RC100/200 [34], SRC-6/7 [35]) available in the world.

3.6.1 Physical Architecture

Maxwell is essentially an IBM BladeCentre cluster with FPGA acceleration, which contains 32 blade servers each with one Intel Xeon CPU and two Xilinx Virtex-4 FPGAs where the CPUs are connected to the FPGAs with a standard IBM PCI-X expansion module. Maxwell (pictured in figure 3.7) comprises two 19-inch racks and five IBM BladeCentres, four of which have seven IBM Intel Xeon blades and the fifth has four (hence 32 blades in total). Each blade is a diskless 2.8 GHz Xeon with 1 GB memory. Furthermore, the blades are booted over the network from the head node (Dell server).

The FPGAs in Maxwell are Xilinx Virtex-4 devices of two different types which are mounted on two different types of plug-in PCI card, namely Alpha Data ADM-XRC-4FX [28] and Nallatech H101 [29]. FPGAs on the Alpha Data cards are XCVFX100 parts having 94,896 logic cells, embedded PowerPC

cores and Multigigabit Serial Transceivers (MGTs) (i.e. RocketIO) for off-chip communications, while those on the Nallatech cards are XC4VLX160 having 152,064 logic cells but no embedded processors and MGTs. Furthermore, both types of card connect to the Xeon on a particular blade using a PCI/PCI-X bridge which is capable of 64 bit, 133 MHz operation in PCI-X mode, giving a peak bandwidth of 1064 MB/s.

The 16 blades in Maxwell host 32 Nallatech H101 PCIXM cards, each of which has 16 MB of DDR-II SRAM in four banks delivering a peak bandwidth of 6.2 GB/s, and one 512 MB bank of DDR-II SDRAM delivering a peak bandwidth of 3.2 GB/s. Also, communication links from the Nallatech cards are achieved through a separate communications chip (a Virtex-II Pro FX device with an embedded router core). Hence, each H101 card has four MGT links each with a maximum bandwidth of 2.5 Gb/s.

On the other hand, rest of the 16 blades in Maxwell host 32 Alpha Data ADM-XRC-4FX PMC/PMC-X/XMC cards, each of which has one 16 MB bank of DDR-II SRAM and 1,024 MB of DDR-II SDRAM in four banks delivering a peak bandwidth of 8.4 GB/s. Furthermore, off-chip communication is direct from four RocketIO MGTs on V4FX FPGA devices with a maximum bandwidth of 3.125 Gb/s per link.

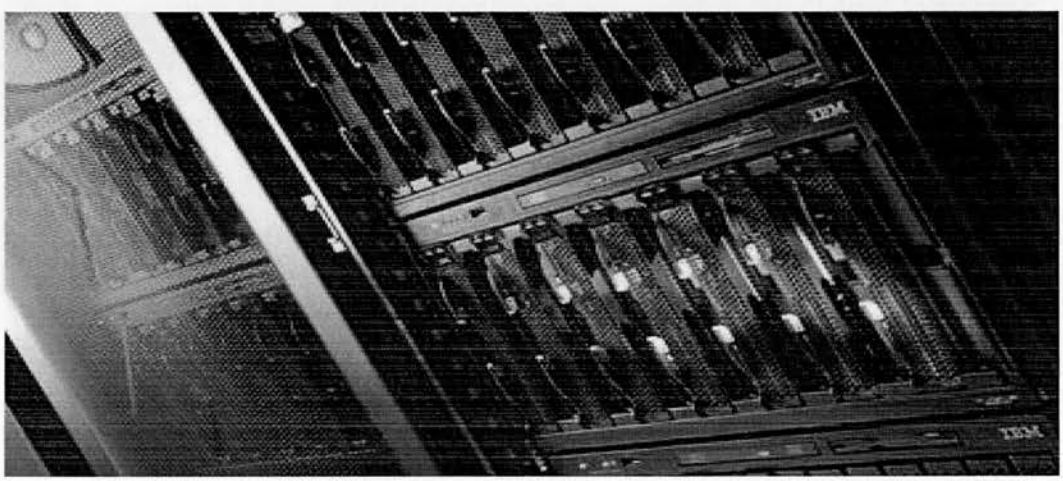


Figure 3.7: *Maxwell-a 64-FPGA supercomputer [31]*

3.6.2 Topology

Maxwell has three independent communications networks for CPU-CPU, CPU-FPGA, and FPGA-FPGA communications. The blade CPUs are networked over gigabit Ethernet through a single 48-way Netgear switch with 40 Gb/s throughput. Thus, CPUs have an all-to-all connectivity. The FPGA network consists of point-to-point links between the MGT connectors of adjacent FPGAs. Each FPGA has 4 MGT links enabling the 64 FPGAs to be connected together in a two-dimensional 8 x 8 torus, as illustrated in figure 3.8. The FPGA pairs hosted on a single CPU form "east-west" pairs in the network. Furthermore, the MGTs are connected with standard HSSDC2 Infiband cables of 50 cm and 100 cm lengths, as shown in figure 3.9. Note that there is no routing logic implemented in FPGAs, so the connections between the FPGAs are purely point-to-point. Finally, the two FPGAs and one CPU on a particular blade can communicate with each other over the PCI bus as mentioned above.

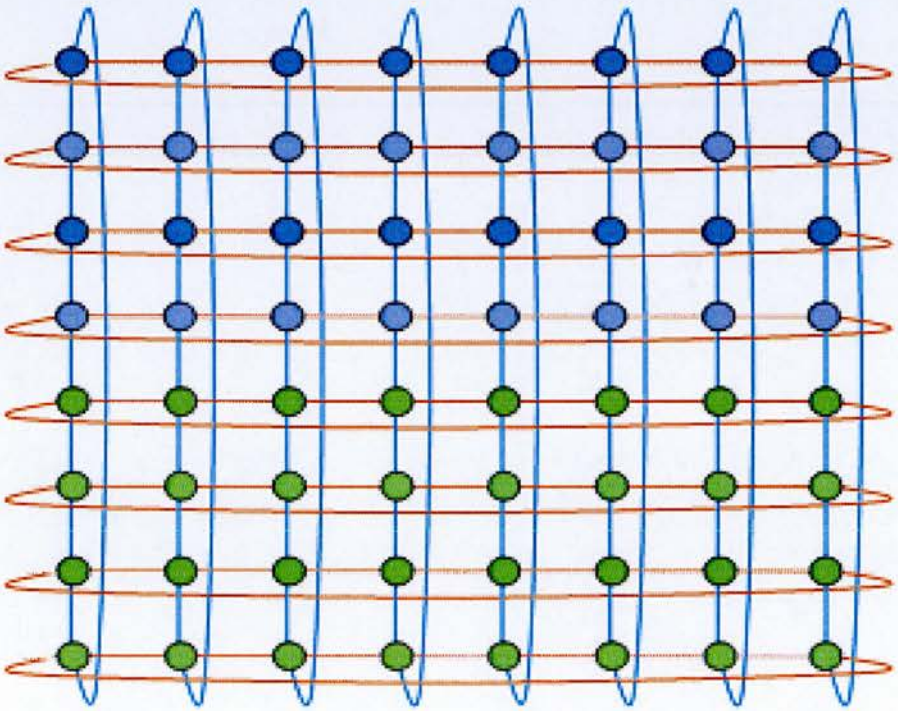


Figure 3.8: *FPGA connectivity in Maxwell [26]*

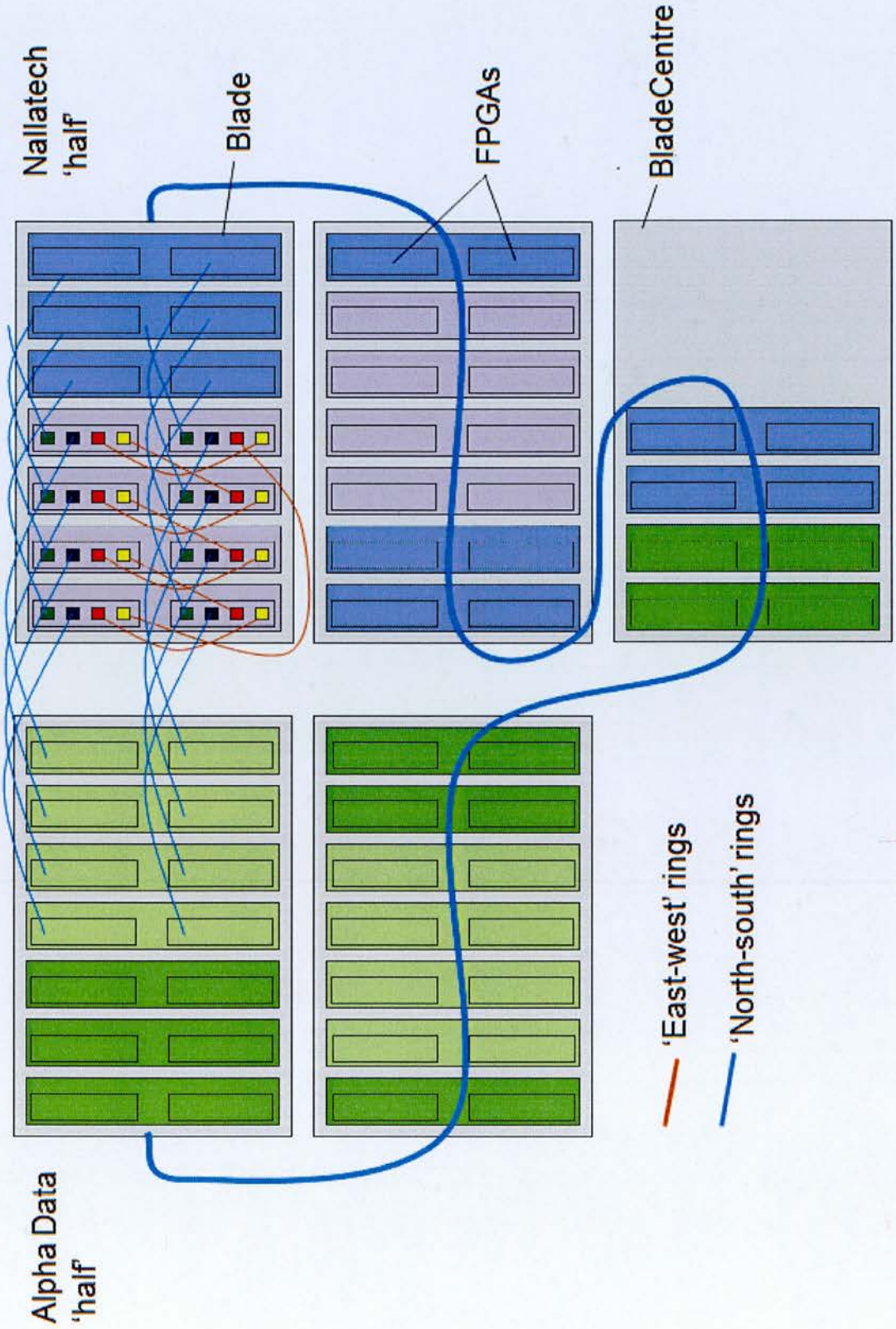


Figure 3.9: *FPGA topology in Maxwell [31]*

3.6.3 Logical Structure

Logically, Maxwell can be regarded as a collection of 64 nodes, where a node is defined as a software process running on a host CPU together with some FPGA acceleration hardware, as illustrated in figure 3.10. In the typical case of 64 nodes configuration, each blade CPU hosts two software processes each of which manages one of the two FPGAs on the blade during runtime. However, it is also possible to have 32 fat nodes where each blade CPU hosts one software process taking care of both of the FPGAs on the blade. Note that logical structure is not set in stone and can be varied per application in the way it would be more beneficial.

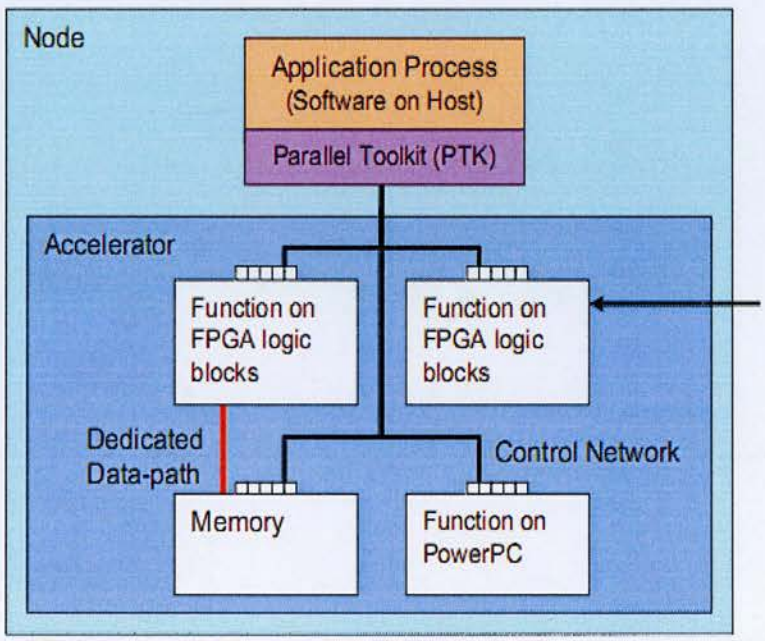


Figure 3.10: *Logical structure of the Maxwell [31]*

3.6.4 Software Environment

The software environment of Maxwell comprises Linux variant CentOS, standard GNU/Linux tools, Sun Grid Engine (SGE) as the batch scheduling system, MPI for inter-process communication and most importantly the

FHPCA Parallel Toolkit (PTK) [30] that forms a bridge from the software process (the application) to the FPGA hardware. As it can be seen in figure 3.11, a node's application process runs on a host CPU and communicates with an FPGA accelerator via the PTK. Essentially, the PTK is a set of practices and infrastructure written mostly in C++ (bash used for scripting tasks) that aims to address acceleration issues such as associating processes with FPGA resources, associating FPGAs with bitstreams, managing contention for FPGA resources within a process and managing code dependencies to facilitate re-use. In other words, it is a set of system level tools to support parallel execution on the Maxwell supercomputer. The PTK comprises a library of C++ classes providing abstract interfaces to FPGA hardware components, classes providing standard ways to configure arbitrary FPGA hardware, and a standard way of launching parallel FPGA jobs [30].

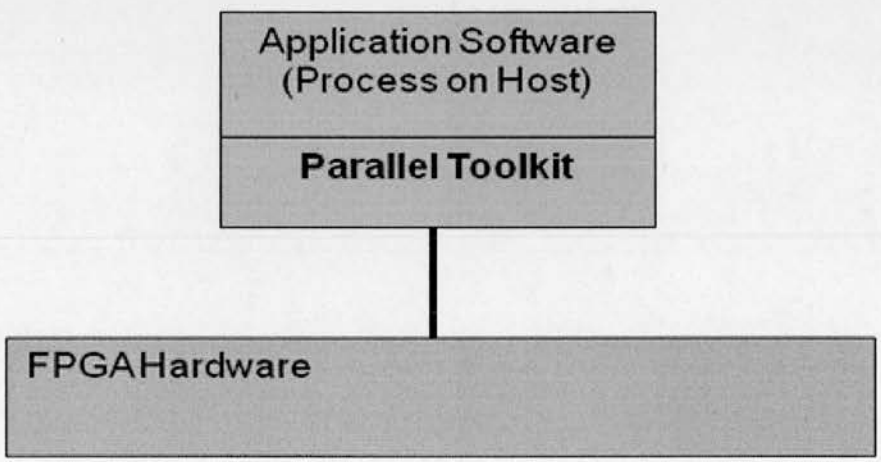


Figure 3.11: *FHPCA Parallel Toolkit [32]*

The approach to designing the PTK began with the assumed starting position of an existing parallel application running entirely in software over multiple processors with inter-process communication handled by the standard Message-Passing Interface (MPI) library. The basic assumption is that the fully parallelized code is not delivering the desired performance. Therefore,

the primary aim in building the PTK was to provide a way to transform the starting code into a hardware-accelerated parallel application. Custom configuration of the FPGA would then deliver the desired performance if the "hotspot" functions of the code are migrated from software to FPGA hardware. The steps of the basic PTK strategy to accelerate an application are listed below as bullet points, referring to figure 3.12 [30]:

- Identify code 'hotspot' function F
- Design corresponding hardware function F'
- F' bitstream programmed into accelerator
- F accelerated by replacing its innards with a call out to F'
- At runtime, F copies relevant input data to memory component on the accelerator before invoking F'
- F' processes data directly from local memory
- Once F' signals completion, F copies output data back to the host
- Execution of F' may involve external communication with neighbouring nodes to implement the message-passing model of parallel computation commonly used in HPC applications (this happens using FPGA-to-FPGA dataflow over MGTs).

The architecture of the PTK as implemented on the Maxwell supercomputer is shown in figure 3.13 where the vendor-neutral *Config File* is a repository of information used in configuring a given machine for a given application and the *PTK Launcher* is a script for launching accelerated applications. On the other hand, *Accelerator* classes which are mainly responsible for configuring hardware model all the relevant FPGA acceleration hardware for a given

implementation of a given algorithm for a given node. Furthermore, *Component* classes and *Hard Data Structure* classes which are the vendor-neutral but application-specific abstract interfaces model functional and data resources on the FPGA hardware, respectively. Moreover, *Allocator* classes serve requests for Components and Hard Data Structures made by the application. Finally, *Runner* classes which are at the level above the Components and Hard Data Structures hide the need to request resources and present simple-case interface to the application code above.

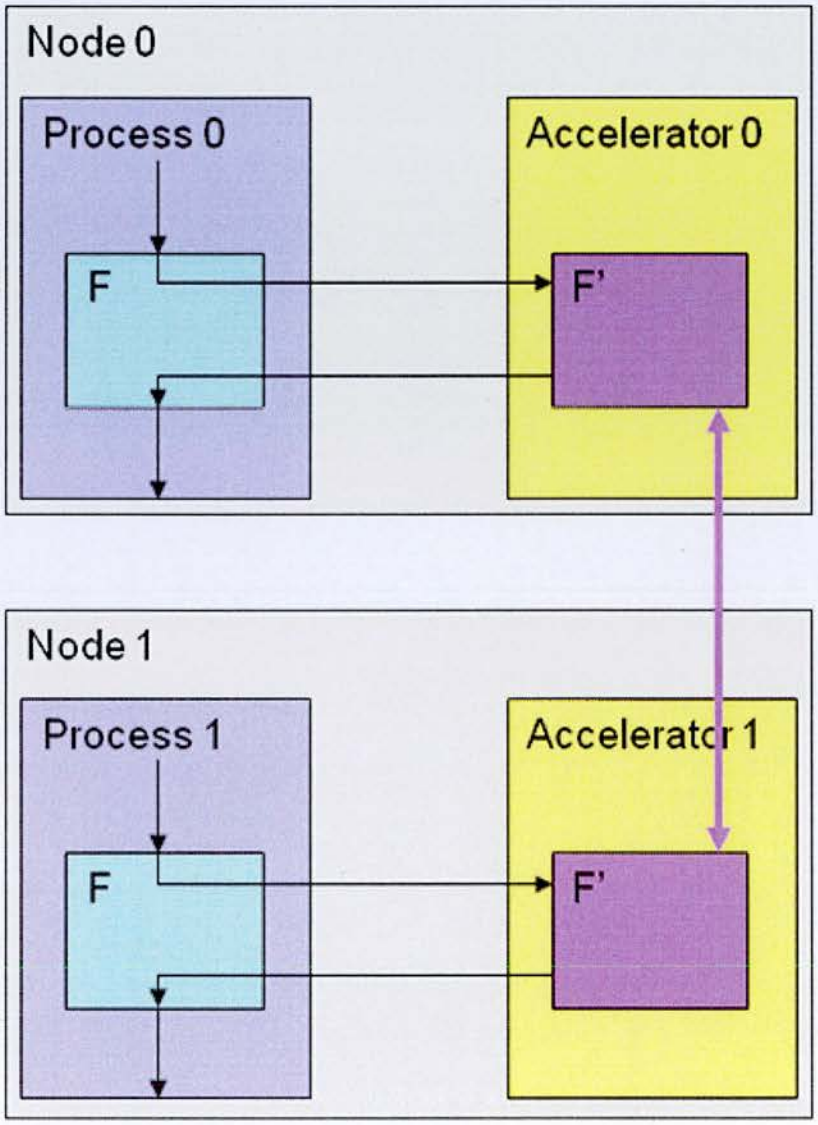


Figure 3.12: *Basic PTK acceleration strategy [32]*

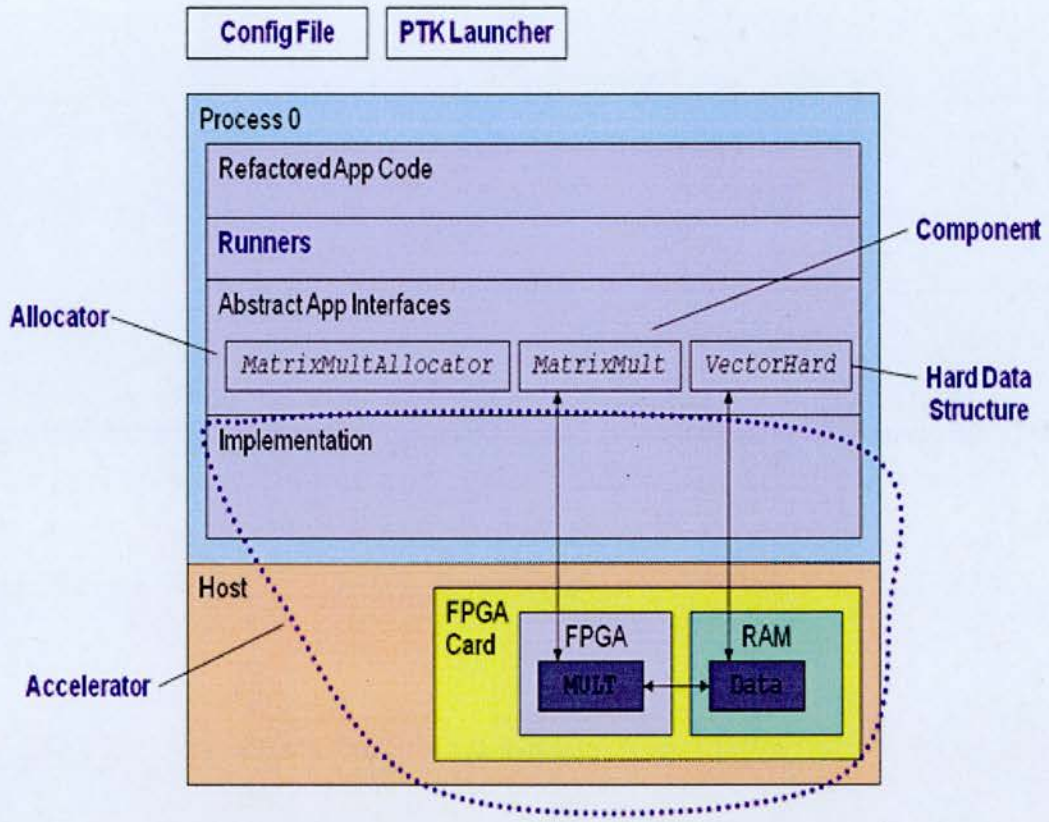


Figure 3.13: *Parallel Toolkit architecture [31]*

3.7 Summary

In this chapter, Field-Programmable Gate Arrays that are widely used in reconfigurable computing were introduced, and then its basic architecture, programming the architecture and additional specialized function resources within the architecture were briefly explained. We concluded this chapter by shortly describing an FPGA-based supercomputer named Maxwell which was used as our hardware implementation platform for two case studies as will be explained in chapters 5 and 6.

Chapter 4

High Performance Position Specific Iterated BLAST Implementation on a Reconfigurable Hardware

4.1 Introduction

In Bioinformatics and Computational biology, biological sequence alignment is a very common task where subject sequences from a large database are aligned to a query sequence to find similarities between the query sequence and the database sequences [2]. A major application of sequence alignment is to infer biological information about a newly discovered sequence from a set of previously annotated sequences. For instance, if a new sequence is found to be similar to a known cancerous sequence, then information regarding the functionality of the new sequence can be inferred, something which is extremely useful in early disease diagnosis and drug engineering. Furthermore, the study of evolutionary development and history of species is essentially based on biological sequence alignment [2] [36].

However, sequence alignment is a computationally intensive operation, and with biological sequence databases growing at an exponential rate [2], desktop computers alone cannot be relied upon to perform this task within acceptable execution times. The utilization of a faster computing platform is hence mandatory. Field Programmable Gate Arrays (FPGAs) have been recently proposed as an efficacious and efficient implementation platform for sequence alignment algorithms, thanks to their flexible computing and memory architecture which gives them ASIC-like performance with the added programmability feature [37] [38] [39].

There are various biological sequence alignment algorithms some of which are exhaustive and give optimal alignments (e.g. Needleman-Wunsch [40], Smith-Waterman [41]) and some of which are heuristic and give sub-optimal alignments (e.g. FASTA[42], BLAST[43]). In this chapter, we concentrate on Basic Local Alignment Search Tool (BLAST) which is a local alignment algorithm. Although it is heuristic which means that it produces local alignments which are not always optimal, it is much faster than ordinary exhaustive dynamic programming algorithms.

The design and FPGA implementation of the most complicated variant of BLAST named Position Specific Iterated BLAST (PSI-BLAST) [5] is presented in this chapter. Although this is the first FPGA implementation of PSI-BLAST reported in the literature, there have been several efforts before to accelerate the basic BLAST algorithm on reconfigurable hardware [51] [55] [56] [57]. Prior to this work, we designed and implemented BLAST with the two-hit method [53] and Gapped BLAST [54] which are the two simpler variants of the basic BLAST algorithm compared to the PSI-BLAST.

An FPGA-platform-independent language, namely Handel-C language [46] was used to capture the design. So, the proposed design can be ported across a number of FPGA architectures (e.g. from Xilinx or Altera). However, since Handel-C is a very high level language for hardware description, the achieved clock frequency for the final FPGA design is low compared to the case where a conventional hardware description language such as Verilog or VHDL was used.

The remainder of this chapter will first present essential background information on the general BLAST algorithm. Then, the design and implementation of the proposed FPGA core for Position Specific Iterated BLAST will be elaborated. Following this, implementation results are presented and then evaluated comparatively with the performance of

equivalent software implementations running on a desktop computer. Finally, conclusions are laid out with plans for future work.

4.2 Background – Essentials of the PSI-Blast Algorithm

Biological sequences evolve through mutation, selection and random genetic drift [44]. Mutation, in particular manifests itself through 3 main processes which are as follows:

- Substitution of residues: Residue A in the sequence is substituted by another residue B.
- Insertion of residues: New residues are inserted into the sequence.
- Deletion of residues: Existing residues in the sequence are deleted.

Insertions and deletions result in *gaps* which are taken into consideration when aligning biological sequences. The degree of alignment of biological sequences is measured by a score which is obtained by the summation of score terms of each aligned pair of residues with possible gap penalty terms. Score terms for each aligned residue pair are obtained from probabilistic models which are stored in scoring or substitution matrices such as BLOSUM50 [2]. The latter is a 20x20 matrix for protein sequence residues. On the other hand, gap penalties depend on the length of the gap and are independent of gap residues. There are two main types of gap penalties:

- Linear gap penalty: The cost of a gap of length g is given by the following linear function where d is the gap penalty:

$$\text{Penalty}(g) = -g*d \quad (4.1)$$

- Affine gap penalty: A constant penalty d is given for opening a new gap while a linear and smaller penalty e is given for subsequent gap extensions. The cost function of the affine gap penalty is hence given by the following affine equation:

$$\text{Penalty}(g) = -d-(g-1)*e \qquad (4.2)$$

BLAST stands for Basic Local Alignment Tool. It is developed on the ideas of FASTA. It is used for searching both protein and DNA sequence databases for sequence similarities. It is a heuristic local alignment algorithm which approximates the dynamic programming Smith-Waterman algorithm. Since it is a heuristic algorithm, the local alignment it produces is not always optimal. However, it is much faster than the Smith-Waterman algorithm. As a result, BLAST and its variants are some of the most widely used sequence search tools.

The central idea of the BLAST algorithm is that a statistically significant alignment is likely to contain high-scoring pairs of aligned words. BLAST first finds these high scoring pairs of aligned words and then extends them to the real alignment. These words are k-residues long where k is different for DNA and protein sequences. The default k values for DNA and protein sequences are 11 and 3 respectively [5]. There are 3 basic steps of BLAST:

- Pre-processing the query sequence: All k-long words in the query sequence are extracted. Then, words that are similar to these are found. We call the overall results the k-words.

- Scanning the subject sequences: All the subject sequences in the database are scanned one by one for matches with the obtained k-words.

- Extension of the matches: All matches in the subject sequences are extended to form local alignments between the query sequence and related subject sequences in the database.

In subsections 4.2.1-4.2.3, all basic steps of the BLAST algorithm mentioned above will be explained in more detail. It is worth mentioning at this stage that the aforementioned basic steps belong to the original BLAST algorithm. However, several variants of the original algorithm have been devised over the years with the aim of increasing its sensitivity while keeping run-times at minimum. All of these variants include the 3 basic steps of the original algorithm, with the addition of new steps. In this chapter, we discuss three of these variants, namely: BLAST with two-hit method, Gapped BLAST and PSI-BLAST which are detailed in subsections 4.2.4, 4.2.5 and section 4.3, respectively.

4.2.1 Step 1: Pre-processing the Query Sequence

An example protein sequence which has 9 residues (or amino acids) is shown below:

LVNRKPVVP

In this first step, we take the query sequence and chop it into overlapping k-words as illustrated below for the query sequence shown above, with k = 3:

Word 0: LVN
Word 1: VNR
Word 2: NRK
Word 3: RKP

Word 4: KPV

Word 5: PVV

Word 6: VVP

As it can be seen, there are 7 words extracted from the query sequence which are 3 residues long. In general, the number of words extracted equals (m-k) + 1 where m is the number of residues in the query sequence. After this, words similar to each of these extracted words are found through the usage of specific scoring (substitution) matrix. An example scoring matrix for protein residues (BLOSUM50) is shown below.

	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W
C	**9**	-1	-1	-3	0	-3	-3	-3	-4	-3	-3	-3	-3	-1	-1	-1	-1	-2	-2	-2
S	-1	**4**	1	-1	1	0	1	0	0	0	-1	-1	0	-1	-2	-2	-2	-2	-2	-3
T	-1	1	**4**	1	-1	1	0	1	0	0	0	-1	0	-1	-2	-2	-2	-2	-2	-3
P	-3	-1	1	**7**	-1	-2	-1	-1	-1	-1	-2	-2	-1	-2	-3	-3	-2	-4	-3	-4
A	0	1	-1	-1	**4**	0	-1	-2	-1	-1	-2	-1	-1	-1	-1	-1	-2	-2	-2	-3
G	-3	0	1	-2	0	**6**	-2	-1	-2	-2	-2	-2	-2	-3	-4	-4	0	-3	-3	-2
N	-3	1	0	-2	-2	0	**6**	1	0	0	-1	0	0	-2	-3	-3	-3	-3	-2	-4
D	-3	0	1	-1	-2	-1	1	**6**	2	0	-1	-2	-1	-3	-3	-4	-3	-3	-3	-4
E	-4	0	0	-1	-1	-2	0	2	**5**	2	0	0	1	-2	-3	-3	-3	-3	-2	-3
Q	-3	0	0	-1	-1	-2	0	0	2	**5**	0	1	1	0	-3	-2	-2	-3	-1	-2
H	-3	-1	0	-2	-2	-2	1	1	0	0	**8**	0	-1	-2	-3	-3	-2	-1	2	-2
R	-3	-1	-1	-2	-1	-2	0	-2	0	1	0	**5**	2	-1	-3	-2	-3	-3	-2	-3
K	-3	0	0	-1	-1	-2	0	-1	1	1	-1	2	**5**	-1	-3	-2	-3	-3	-2	-3
M	-1	-1	-1	-2	-1	-3	-2	-3	-2	0	-2	-1	-1	**5**	1	2	-2	0	-1	-1
I	-1	-2	-2	-3	-1	-4	-3	-3	-3	-3	-3	-3	-3	1	**4**	2	1	0	-1	-3
L	-1	-2	-2	-3	-1	-4	-3	-4	-3	-2	-3	-2	-2	2	2	**4**	3	0	-1	-2
V	-1	-2	-2	-2	0	-3	-3	-3	-2	-2	-3	-3	-2	1	3	1	**4**	-1	-1	-3
F	-2	-2	-2	-4	-2	-3	-3	-3	-3	-3	-1	-3	-3	0	0	0	-1	**6**	3	1
Y	-2	-2	-2	-3	-2	-3	-2	-3	-2	-1	2	-2	-2	-1	-1	-1	-1	3	**7**	2
W	-2	-3	-3	-4	-3	-2	-4	-4	-3	-2	-2	-3	-3	-1	-3	-2	-3	1	2	**11**

Figure 4.1: *The Blosum50 scoring (substitution) matrix*

Words which score at least user-defined threshold value T with the scoring matrix when aligned with the words extracted from the query sequence are regarded to be similar to these extracted words. Similar words for each extracted word are found and then recorded with the location address of the corresponding extracted word in the query sequence tagged to them. This

process is illustrated below with the first extracted word shown above (i.e. LVN) using the Blosum50 scoring matrix for the case where T is 12:

Word 0: L V N
4 + 4 + 6 = 14
Query word 1: L V N

Word 0: L V N
2 + 4 + 6 = 12
Query word 2: M V N

Word 0: L V N
4 + 4 + 1 = 9
Query word 3: L V S

Query word 1 and query word 2 score 14 and 12 respectively when aligned with the first extracted word (LVN) from the query sequence. Since score values are over 12, query word 1 and query word 2 are recorded with the location address of the first extracted word in the query sequence, which is 0. However, query word 3 is discarded since it scores less than 12 when aligned with the extracted word. All recorded similar words are used in step 2 of the BLAST algorithm.

4.2.2 Step 2: Scanning the Subject Sequences

In this step, all subject sequences in the database are scanned one by one to find the possible exact matches of the query words which were recorded in step 1. Each match is referred to as a hit or hotspot. Each hit is recorded in a list for the third step of the BLAST algorithm with the identity of the

corresponding query word and the location address where the hit occurred in the subject sequence. Considering the fact that current databases contains tens of thousands of subject sequences and that each subject sequence comprises hundreds/thousands of residues, it is obvious that this sequence database scanning process is a massively time consuming task.

4.2.3 Step 3: Extension of the Matches

In this last step of the basic BLAST algorithm, we utilize the list of matches (hits) obtained in step 2 to form local alignments between the query sequence and the subject sequences in the database. Each entry in the list of hits contains the location address of a match in the subject sequence and the location address of the corresponding query word in the query sequence. Starting from these 2 location addresses, each of the hits in the list is extended on the query and corresponding subject sequence in both directions without allowing any gaps. In this extension, pairs of residues along the query and subject sequence are scored with a scoring matrix (e.g. BLOSUM50). This process is illustrated in figure 4.2 with the following subject sequence:

GVCRRPLKC

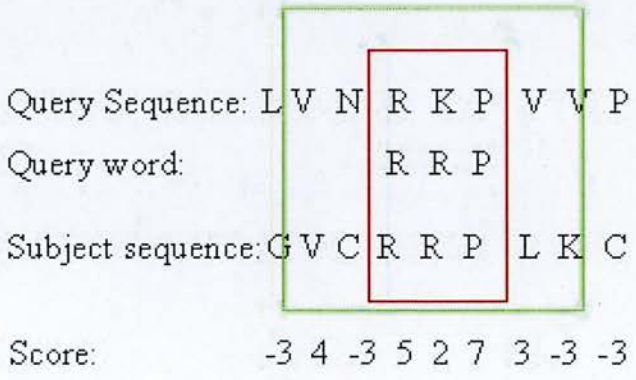


Figure 4.2: *Step 3: Extension of the matches*

In figure 4.2, the red box shows a hit where query word RRP is matched in the subject sequence. The query word RRP is similar to RKP word in the query sequence. The green box shows the extension which started from the edges of the red box. As the extension proceeds in a 1 residue pair at a time in both directions and without allowing for any gaps, pairs of residues along the extension are scored using a scoring matrix (BLOSUM50 in our case). These score terms are added up after each extension step and the extension is terminated when this total score falls a user-defined cut-off distance below the best total score obtained so far. Then, the extension goes back to its state which yielded the highest total score. As a result of this extension step, the related subject sequence is locally aligned to the query sequence (without gaps).

4.2.4 BLAST with Two-Hit Method

The third step of the BLAST algorithm, i.e. the extension of the matches on the query and subject sequences, generally accounts for a very high percentage of the BLAST algorithm's execution time. Hence, the two-hit method was devised to reduce the time spent in this extension step. The central idea of the two-hit method is to start extension only when there are two non-overlapping hits on the same diagonal within distance A of each other. This is illustrated in figure 4.3 where only two non-overlapping hits on the same diagonal line which are close enough to each other are extended.

In other words, if the distance between any two non-overlapping hits on the subject sequence is equal to the distance between the locations of the corresponding query words in the query sequence, then ungapped extension is triggered in both directions starting from both hits. The rest of the process is the same as explained in subsection 4.2.3 above and the result is a local ungapped alignment of the query and subject sequences. This process is illustrated in figure 4.4 where A is equal to 5.

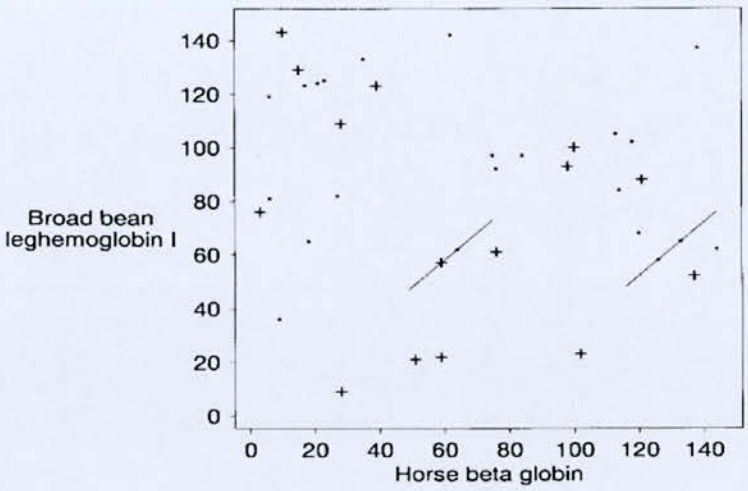


Figure 4.3: *Ungapped extension of the two close hits on the sane diagonal lines [5]*

In figure 4.4, the red boxes show two non-overlapping hits on the query and subject sequences within a distance of 4. Since the distance between the query words in the query sequence is equal to the distance between the two hits on the subject sequence, and since this distance between the two hits is less than 5, and bigger than 2, ungapped extension is started from the edges of the left and right hand sides of the red boxes respectively (see the green box in figure 4.4).

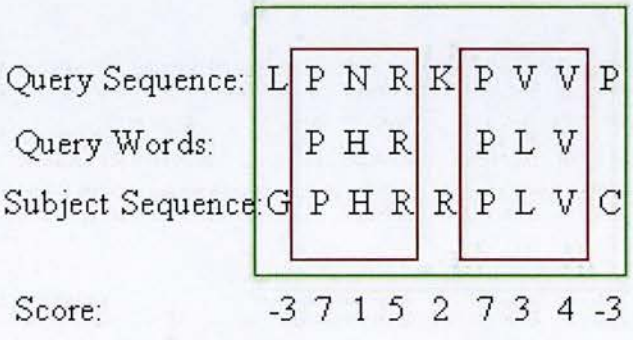


Figure 4.4: *Extension with the two-hit method*

To maintain the sensitivity of the general algorithm, the threshold value T used in the query pre-processing step of the algorithm is reduced. Hence, the

number of query words recorded in this step will increase. As a result, while scanning the subject sequences in step 2 we will potentially find more hits than before. However, only a small fraction of these hits will have an associated second hit. Therefore, ungapped extension will be triggered less frequently compared to the case in the original BLAST algorithm. The total execution time of BLAST is thus reduced.

4.2.5 Gapped BLAST

Gapped BLAST is an advancement of BLAST with the two-hit method, which is faster and gives better alignments and alignment scores. In addition to the steps outlined above, gapped alignment is triggered in gapped BLAST if local ungapped alignment obtained as a result of ungapped extension has a sufficiently high score. If this is the case, the central pair of the local ungapped alignment is used as a seed from which the gapped alignment is run both backwards and forwards, as illustrated in figure 4.5. The gapped alignment algorithm utilized in Gapped BLAST is a modified version of the Needleman-Wunsch algorithm where the alignment is pruned when alignment scores fall a user-defined cut-off distance below the best score so far. The Needleman-Wunsch algorithm with linear and affine gap models is explained in subsections 4.2.6 and 4.2.7 below, respectively. The necessary modifications of the original Needleman-Wunsch algorithm needed in the Gapped BLAST algorithm are explained in subsection 4.2.8.

4.2.6 The Needleman-Wunsch Algorithm with the Linear Gap Model

The Needleman-Wunsch algorithm is a dynamic programming algorithm which finds optimal global gapped alignment between two sequences [40]. In Gapped BLAST, however, it is used for local alignment purposes, after a

slight modification as will be explained in subsection 4.2.8 below. In this section, we will present the original Needleman-Wunsch algorithm where a linear gap model is assumed.

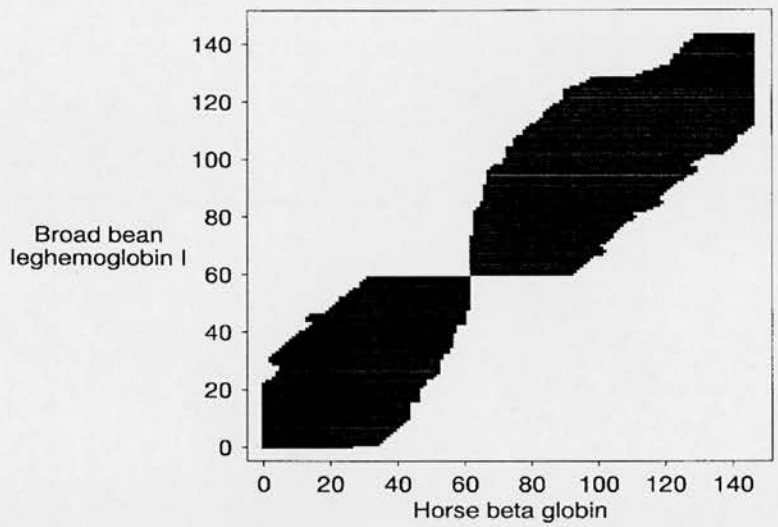


Figure 4.5: *Gapped alignment started from the central pair of the local ungapped alignment in both directions [5]*

Assuming we have two sequences $X = x_1x_2....x_M$ and $Y = y_1y_2.....y_N$, whose lengths are M and N respectively, a dynamic programming score matrix F is built where each cell F (i, j) represents the best alignment between $x_1x_2....x_i$ segment of X and $y_1y_2.....y_j$ segment of Y. The boundary cells of Matrix F are set by the following set of equations:

$$F(0, 0) = 0 \quad (4.3)$$

$$F(i, 0) = -i*d \text{ where } i=1, 2....M \quad (4.4)$$

$$F(0, j) = -j*d \text{ where } j=1, 2....N \quad (4.5)$$

Equation 4.6 is used to compute the values of each of the remaining cells of matrix F:

$$F(i,j) = max\begin{Bmatrix} F(i-1,j-1)+s(x_i,y_j) \\ F(i-1,j)-d \\ F(i,j-1)-d \end{Bmatrix} \quad (4.6)$$

Here, we aim to find best alignment between $x_1x_2....x_i$ and $y_1y_2.....y_j$ given the best alignment between $x_1x_2....x_{i-1}$ and $y_1y_2.....y_{j-1}$ (i.e. F(i-1, j-1)), between $x_1x_2....x_{i-1}$ and $y_1y_2.....y_j$ (i.e. F(i-1, j)) and between $x_1x_2....x_i$ and $y_1y_2.....y_{j-1}$ (i.e. F(i, j-1)). There are three alternatives:

- An alignment between x_i and y_j: In this case, the new score F(i, j) is F(i-1, j-1) + $s(x_i, y_j)$ where $s(x_i, y_j)$ is the scoring matrix score for x_i and y_j.
- An alignment between x_i and a gap in Y: In this case, the new score F(i, j) is F (i-1, j)-d where d is the user-defined gap penalty.
- An alignment between a gap in X and y_j: In this case, the new score F(i, j) is F (i, j-1)-d where d is the user-defined gap penalty.

One of these three alternatives (see figure 4.6) yields the largest score and is the best alignment between $x_1x_2....x_i$ and $y_1y_2.....y_j$.

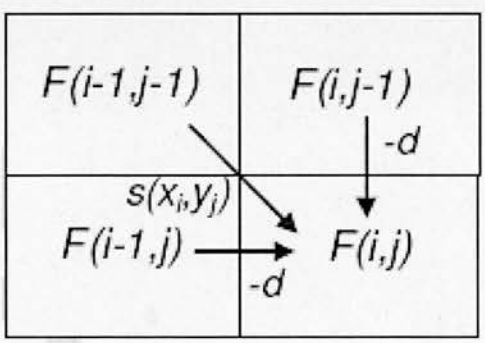


Figure 4.6: *Illustration of the Needleman-Wunsch dynamic programming equations*

Note that a pointer to the cell from which F (i, j) was derived (i.e. above, left, above-left) is stored in each cell. Once the value of the last cell of matrix F (i.e. F (M, N)) is computed, the best global alignment between X and Y is obtained by tracking back from this cell, using the aforementioned pointers, and applying the following procedure:

- If cell (i, j) was derived from cell (i-1, j-1), the pair of symbols x_i and y_j is added to the front of the current alignment.
- If cell (i, j) was derived from cell (i-1, j), x_i and a gap in Y are added to the front of the current alignment.
- If cell (i, j) was derived from cell (i, j-1), a gap in X and y_j are added to the front of the current alignment.

This is illustrated in figure 4.7 for 2 protein sequences. In this figure, the trace-back starts from F(M, N) = F(7, 10) and moves backward to the cell from which the current cell was derived until F(0,0) is reached, while applying the aforementioned procedure at every step of the trace-back. The resulting global alignment of these 2 sequences can be seen at the bottom of figure 4.7.

4.2.7 The Needleman-Wunsch Algorithm with the Affine Gap Model

The Needleman-Wunsch algorithm with the affine gap model is similar to the one with the linear gap model. However, in this case, we have three new matrixes namely I_z, I_x and I_Y to compute. The following equations are used to compute the values of I_z, I_x and I_Y where d is the user-defined penalty associated with the gap opening and e is the user-defined penalty associated with the gap extension:

$$I_Z(i,j) = max\begin{Bmatrix} I_Z(i-1,j-1)+s(x_i,y_j) \\ I_X(i-1,j-1)+s(x_i,y_j) \\ I_Y(i-1,j-1)+s(x_i,y_j) \end{Bmatrix} \quad (4.7)$$

$$I_X(i,j) = \max\begin{Bmatrix} I_Z(i-1,j)-d \\ I_X(i-1,j)-e \end{Bmatrix} \quad (4.8)$$

$$I_Y(i,j) = \max\begin{Bmatrix} I_Z(i,j-1)-d \\ I_Y(i,j-1)-e \end{Bmatrix} \quad (4.9)$$

The values of the dynamic programming matrix cells F(i, j) are equal to the maximum of I_Z *(i, j)*, I_X *(i, j)* and I_Y *(i, j)* as shown in equation 4.10.

$$F(i,j) = max\begin{Bmatrix} I_Z(i,j) \\ I_X(i,j) \\ I_Y(i,j) \end{Bmatrix} \quad (4.10)$$

Note that the pointer to the above-left cell is stored in the cell if *F(i, j)* is set to equal I_Z *(i, j)* whereas the pointer to the left cell is stored if *F(i, j)* is set to equal I_x*(i, j)*. Finally, the pointer to the above cell is stored if *F(i, j)* is set to equal I_Y(i, j).

4.2.8 Modified Needleman-Wunsch algorithm

The Needleman-Wunsch algorithm presented above is used for finding global gapped alignments between two sequences. Gapped BLAST however requires some modifications to the original Needleman-Wunsch algorithm. First, no computations are done for the dynamic programming matrix cells which are adjacent to cells whose F(i, j) values are a certain cut-off value

below the highest cell value computed so far. Second, the trace-back procedure may start at any cell which has the highest value F(i, j) among all the cells, rather than bottom rightmost cell. In this way, we have a local gapped alignment at the end of the trace-back procedure.

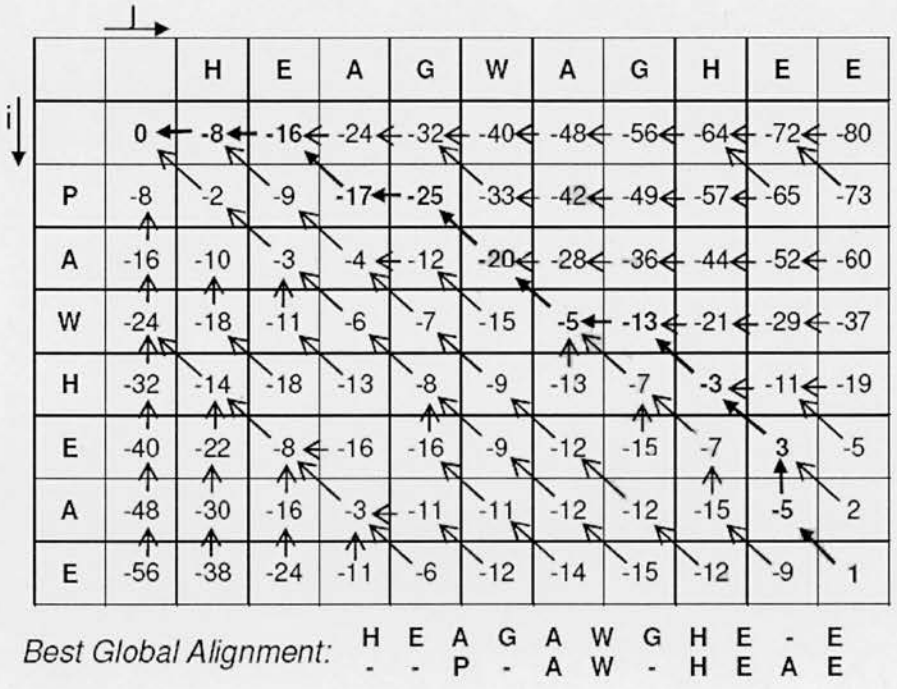


Figure 4.7: *Illustration of the Needleman-Wunsch algorithm*

4.3 Position Specific Iterated BLAST (PSI-BLAST)

PSI-BLAST is a profile (or motif) based search method which is more sensitive than Gapped BLAST at detecting distant relationships among query and database sequences. It can identify additional related database sequences that might otherwise be missed by Gapped BLAST. In essence, PSI-BLAST is iterative Gapped BLAST. It consists of following main steps:

1. A database search is conducted with Gapped BLAST using a query sequence and a scoring matrix (BLOSUM 50).

2. All of the subject sequences with local alignment score higher than a specific threshold value are identified and then a multiple alignment of the segments of these high scoring subject sequences and the query sequence is performed. This multiple sequence alignment is detailed to some extent in subsection 4.3.1.

3. A profile called PSSM (Position Specific Scoring Matrix) is abstracted from the aforementioned multiple sequence alignment. A PSSM is a matrix with n rows and m columns where n is the size of the alphabet (n=20 for protein sequences) and m is the length of the query sequence. More information regarding PSSM and its construction from multiple sequence alignment is presented in subsection 4.3.2.

4. Gapped BLAST is iterated using the obtained PSSM instead of the query sequence itself and the substitution matrix with the aim of identifying a higher number of related database sequences. The way PSSM is utilized in Gapped BLAST is explained in section 4.4 below.

5. After the second iteration, PSSM is updated by taking newly discovered distant relative database sequences into account through steps 2 and 3. This new PSSM is utilized in subsequent Gapped BLAST iteration.

6. Iterations of Gapped BLAST continue until no more new related database sequences are discovered.

4.3.1 Multiple Sequence Alignment

After each iteration of Gapped BLAST, the high scoring segments of subject sequences and the query sequence are multiply aligned. The query sequence is used as a template for constructing the multiple alignment. That is to say that each subject sequence segment is first pairwisely and globally aligned to

the query sequence and then all these obtained alignments are compiled to form a multiple alignment M. An example multiple alignment M is shown in figure 4.8 with the query sequence at the top.

Columns of M that involve gap characters inserted into the query sequence are ignored so that M has the same length as the query sequence. Figure 4.9 shows the multiple alignment M of figure 4.8 trimmed to the length of the query sequence. The PSSM matrix is constructed from the trimmed multiple alignment M as will be explained in the next subsection.

```
AASSLDELVALCKRRGFIF---QSSE------IYGG---L-QGVYD-YGPLGVELKNNLK

-------VVNTLERRLFYI---PSFK------IYSG---V-AGLFD-YGPPGCAIKSNV-
----------------------QSFA------IYGG---V-TGLYD-FGPMG----ANML
-AVAREALVDLCRRRHFLSGTPQQLS------TAAL---L-SGCHARFGPLGVELRKNLA
----------LIKRRFFYD---QSFSMTSRFAIYGG---I-TGQFD-FGPMGCALKSNMI
-----EALLEICQRRHFLS---GSKQ------QLSRDSLL-SGCHPGFGPLGVELRKNLA
--S---KLESTLRRRFFYT---PSFE------IYGG---V-SGLFD-LGPPGCQLQNNLI
-----EQLESVLRGRFFYA---PAFD------LYGG---V-SGLYD-YGPPGCAFQNNII
```

Figure 4.8: *The multiple sequence alignment M*

```
AASSLDELVALCKRRGFIFQSSEIYGGLQGVYDYGPLGVELKNNLK

-------VVNTLERRLFYIPSFKIYSGVAGLFDYGPPGCAIKSNV-
--------------------QSFAIYGGVTGLYDFGPMG----ANML
-AVAREALVDLCRRRHFLSQQLSTAALLSGCHAFGPLGVELRKNLA
----------LIKRRFFYDQSFSIYGGITGQFDFGPMGCALKSNMI
-----EALLEICQRRHFLSGSKQQLSRLSGCHPFGPLGVELRKNLA
--S---KLESTLRRRFFYTPSFEIYGGVSGLFDLGPPGCQLQNNLI
-----EQLESVLRGRFFYAPAFDLYGGVSGLYDYGPPGCAFQNNII
```

Figure 4.9: *The trimmed multiple sequence alignment M*

4.3.2 Construction of Position Specific Scoring Matrix (PSSM)

A PSSM is a motif descriptor which includes a weight (score) for each residue occurring at each position along the motif. It is a 20 by m matrix for

protein sequences where m is the length of the motif. The 20 rows of each column specify the probability of finding each of the 20 amino acids at that position in the motif. Figure 4.10 shows an example PSSM for a motif of length 8. The M_{jk} element of the PSSM is the score for the j^{th} amino acid at the k^{th} position of the motif.

A	-1	-2	-1	0	-1	-2	0	-2
R	5	0	5	-2	1	-3	-2	0
N	0	6	0	0	0	-3	0	1
D	-2	1	-2	-1	0	-3	-1	-1
C	-3	-3	-3	-3	-3	-2	-3	-3
Q	1	0	1	-2	5	-3	-2	0
E	0	0	0	-2	2	-3	-2	0
G	-2	0	-2	6	-2	-3	6	-2
H	0	1	0	-2	0	-1	-2	8
I	-3	-3	-3	-4	-3	0	-4	-3
L	-2	-3	-2	-4	-2	0	-4	-3
K	2	0	2	-2	1	-3	-2	-1
M	-1	-2	-1	-3	0	0	-3	-2
F	-3	-3	-3	-3	-3	6	-3	-1
P	-2	-2	-2	-2	-1	-4	-2	-2
S	-1	1	-1	0	0	-2	0	-1
T	-1	0	-1	-2	-1	-2	-2	-2
W	-3	-4	-3	-2	-2	1	-2	-2
Y	-2	-2	-2	-3	-1	3	-3	2
V	-3	-3	-3	-3	-2	-1	-3	-3

Figure 4.10: *A Position Specific Scoring Matrix (PSSM)*

The PSSM can be constructed from the multiple alignment M described in subsection 4.3.1 above. The first step of PSSM matrix construction is the reduction of each column of M to form the columns of matrix M_C (motif under consideration). To construct each column C of M_C, the set R of sequences that contribute a residue in column C of M are identified as illustrated in figure 4.11 for the third column of the multiple alignment M shown at the top of the figure.

```
AASSLDELVALCKRRGFIFQSSEIYGGLQGVYDYGPLGVELKNNLK
---------VVNTLERRLFYIPSFKIYSGVAGLFDYGPPGCAIKSNV-
-----------------------QSFAIYGGVTGLYDFGPMG----ANML
-AVAREALVDLCRRRHFLSQQLSTAALLSGCHAFGPLGVELRKNLA
----------LIKRRFFYDQSFSIYGGITGQFDFGPMGCALKSNMI
-----EALLEICQRRHFLSGSKQQLSRLSGCHPFGPLGVELRKNLA
---S---KLESTLRRRFFYTPSFEIYGGVSGLFDLGPPGCQLQNNLI
-----EQLESVLRGRFFYAPAFDLYGGVSGLYDYGPPGCAFQNNII
```

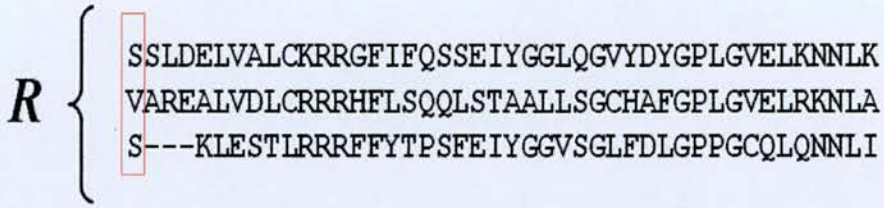


Figure 4.11*: Reduction of one column of the multiple alignment M*

We use the data-dependent pseudo-count method proposed in [5] to calculate the values of PSSM elements from M_C. In it, the PSSM score for the j^{th} amino acid at the k^{th} position (M_{jk}) is computed as shown in equation 4.11, where P_{jk} is the frequency of residue j at the k^{th} position of the M_C matrix and P_j is the background frequency of residue j. Background frequencies of residues are derived from large and carefully selected sets of alignments [52].

$$M_{jk} = log\left(\frac{P_{jk}}{P_j}\right) \quad (4.11)$$

The following equation is used to compute P_{jk}:

$$P_{jk} = \frac{\alpha * f_{jk} + \beta * h_{jk}}{\alpha + \beta} \quad (4.12)$$

where f_{jk} and h_{jk} are the observed frequency and pseudo-count frequency of residue j at position k of M_C, respectively. α and β are the relative weights given to the observed and pseudo-count frequency residues, respectively. In equation 4.12, α is equal to N_C-1, where N_C is the total number of different residue types, including gaps, observed in the columns of M_C, whereas β is set to the default value of 7. The value of h_{jk} in equation 4.12 is set to depend on the observed residue frequencies via a scoring matrix S_{ij} (see equation 4.13) where λ is a natural scale for S_{ij} [5].

$$h_{jk} = p_j \sum_{i=1}^{20} f_{ik} e^{\lambda S_{ij}} \qquad (4.13)$$

As stated above, in PSI-BLAST, it is this obtained PSSM matrix that is used instead of the query sequence and original substitution matrix (e.g. BLOSUM50) in subsequent database search iterations, with the aim of identifying a higher number of related database sequences. The process of database search and PSSM generation is iterated until no more new related database sequences are discovered.

4.4 Hardware Implementation

Figure 4.12 shows a hardware architecture which implements the PSI-BLAST algorithm. Each block in the architecture implements one step of the algorithm as described in the above sections, except for the pre-processing query sequence step and construction of the PSSM which are implemented by high level application software running on the host computer. The architecture consists of 8 *HitFinderTwoHit* blocks, 2 *UngappedExtender* blocks and 1 *GappedExtender block* all of which are running in parallel. There are also 8 32K x 5 bits subject sequence memories each of which holds a number of subject sequences. Note that each subject sequence memory belongs to one *HitFinderTwoHit* block. Each *HitFinderTwoHit* block is composed of 5 *HitFinder* blocks and 1 *TwoHitMethod* block. Each *HitFinder* block implements step 2 outlined in subsection 4.2.2 and scans its assigned subject sequence memory to find exact matches of the query words in the subject sequences. Each *TwoHitMethod* block performs the two-hit method procedure on hits coming from the 5 *HitFinder* blocks which are in the same *HitFinderTwoHit* block as the *TwoHitMethod* block. Besides these, each *UngappedExtender* block implements step 3 mentioned in subsection 4.2.3 and extends the two hits

found by its 4 allocated *TwoHitMethod* blocks without allowing gaps, in order to obtain local ungapped alignments. Finally, a single *GappedExtender* block implements the modified Needleman-Wunsch algorithm to produce local gapped alignments from local ungapped alignments obtained in 2 *UngappedExtender* blocks. Note that the number of *HitFinder*, *TwoHitMethod*, *UngappedExtender* and *GappedExtender* blocks in the architecture was chosen after empirical study with real biological data in order to balance the loading between design blocks.

The high level application software and all of the blocks which constitute the architecture shown in figure 4.12 are detailed in the following subsections.

4.4.1 High Level Application Software

Figure 4.13 shows the organization of the proposed PSI-BLAST FPGA implementation. Application software running on the host has many duties, the most important of which is the query sequence pre-processing as

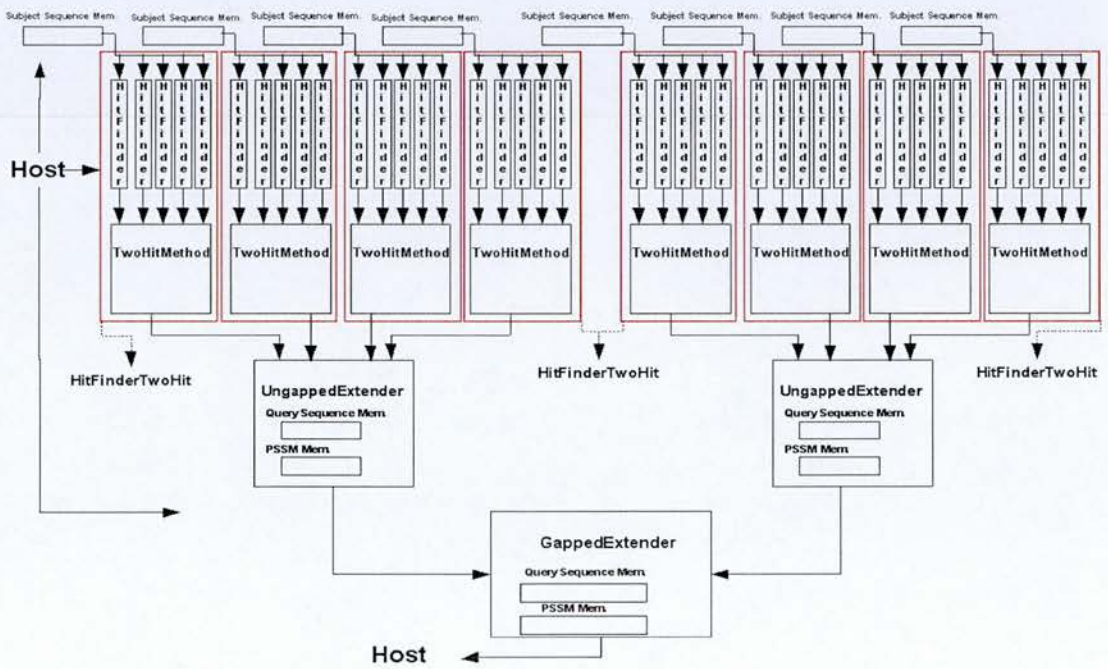


Figure 4.12: *Hardware architecture for the PSI-BLAST algorithm*

explained in section 4.2.1. In brief, the application software finds 3 letter long query words which score at least threshold value T with a scoring matrix when aligned with words extracted from the query sequence. However, in case when there is a constructed PSSM, the application software finds 3 letters long query words which score at least threshold value T when aligned with the PSSM. Then, the location address of each of these query words in the query sequence is placed at a vacant position in an upper word list and a lower word list pair depending on the 2 most significant letters and 2 least significant letters of the query word, respectively (see subsection 4.4.2 below for more detail on our hit finding implementation). Note that there are 5 upper word and lower word list pairs.

As it can be seen in figure 4.13, we produced various FPGA configuration bit files for different threshold and cut-off value parameters. The first task of the application software is to pick the proper bit file, depending on the user-supplied algorithm parameters, from a database of FPGA configurations and load it on to the FPGA chip. Afterwards, the application software runs the hardware configuration in 4 modes. In mode 1, the application software sends one of the 5 upper word and lower word list pairs to each of the 5 *HitFinder* blocks in every *HitFinderTwoHit* block. In mode 2, a number of subject sequences are sent to the 8 available subject sequence memories on FPGA, depending on the subject sequence lengths. In mode 3, the application software sends a query sequence to the FPGA to be stored in the query sequence memories within the 2 *UngappedExtender* blocks and the single *GappedExtender* block. Finally, the execution of the hardware configuration is launched in mode 4. After some time, the FPGA starts sending the high scoring subject sequences to host with their alignment scores. By repeating these steps several times for different subject sequences, we can align all subject sequences in a sequence database to the query sequence (or to PSSM when we have a constructed one).

When all subject sequences are aligned, segments of subject sequences which have a local alignment score higher than a specific threshold value are multiply aligned with the query sequence as explained in subsection 4.3.1 by the application software. Then, the application software constructs the PSSM matrix from the multiple sequence alignment as explained in subsection 4.3.2 above.

After the construction of PSSM, application software iterates all the aforementioned steps to perform a new Gapped BLAST operation. However, in subsequent iteration, the application software sends the PSSM constructed matrix instead of the query sequence to the FPGA, in mode 3, to be stored in the PSSM memories within the 2 *UngappedExtender* blocks and the single *GappedExtender* block. These Gapped BLAST iterations continue until no more new high scoring subject sequences are found.

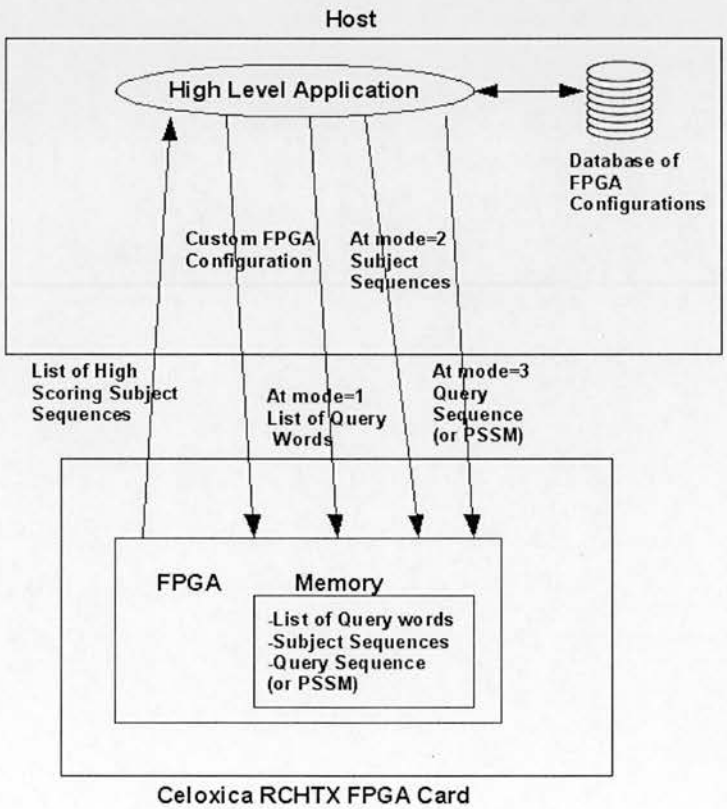


Figure 4.13: *Organization of the proposed PSI-BLAST system*

4.4.2 HitFinder Block

Figure 4.14 shows a simplified inner structure of a *Hitfinder* block. The architecture of this block is a modified version of of an architecture presented in [51]. Indeed, as opposed to [51] we added the positions of the query words in the query sequence into the memory content of the Hit Finder to increase the sensitivity of the hit finding process. Furthermore, the proposed design implements the two-hit method (detailed in the next subsection) which is not the case in the implementation reported in [51]. Lastly, the proposed core includes a unit for gapped alignment for the purpose of implementing Gapped BLAST in contrast to [51] which just implements the original BLAST.

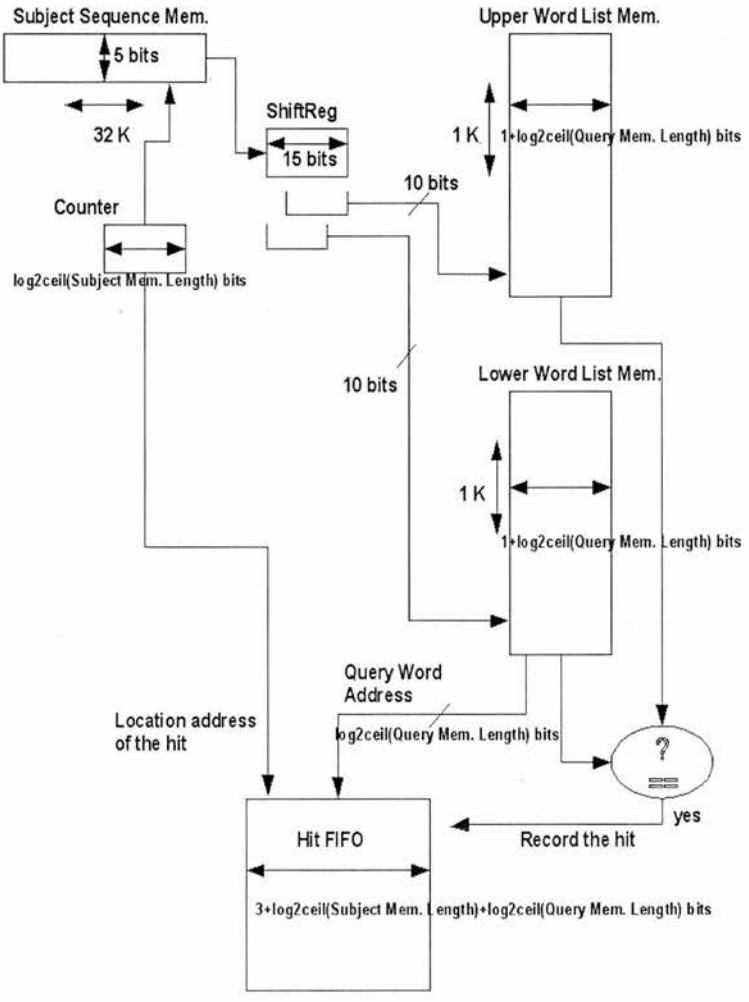


Figure 4.14: *Simplified inner structure of the Hitfinder block*

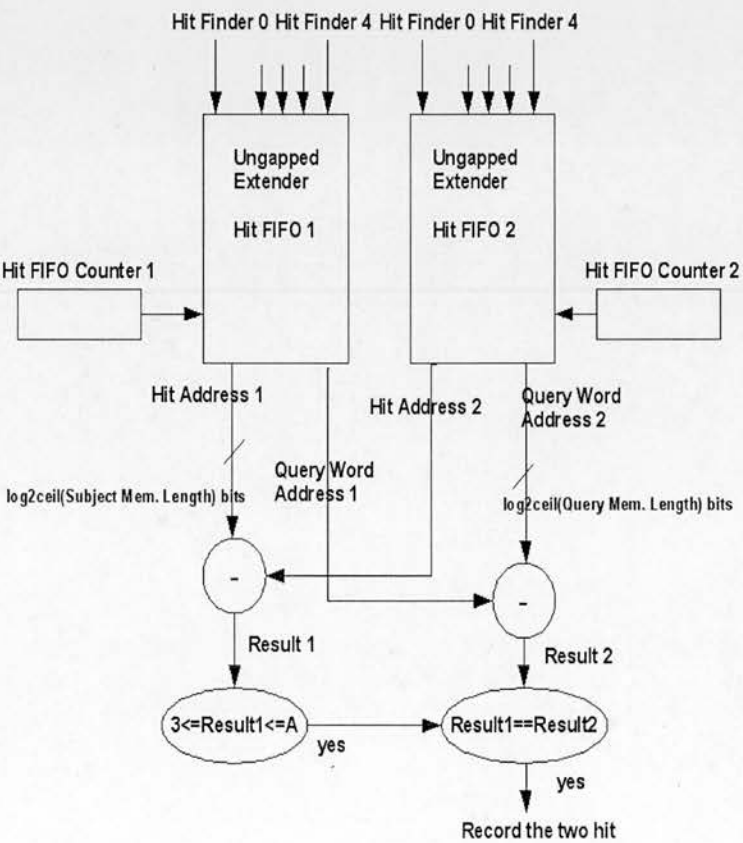
The major aim of this block is to scan each three letter long word of the subject sequences in order to find exact matches of the query words, as explained in subsection 4.2.2. It is comprised of an upper word list memory, a lower word list memory, a shift register, a FIFO buffer and some control logic. Note that every *Hitfinder* block is assigned to a subject sequence memory whose address register (*Counter*) is unique in the *HitFinderTwoHit* block.

At every clock cycle, 5-bit long residues of a subject sequence are shifted into the shift register (*ShiftReg*) from the assigned subject sequence memory and the address register of the subject sequence memory is incremented by one. The shift register is 15 bits long and hence it can hold 3 subject sequence residues at the same time. At every clock cycle, the 10 most significant bits and the 10 least significant bits of the shift register content are used as addresses for the upper word list memory and the lower word list memory respectively (see figure 4.14). If the resulting outputs of these memories are valid entries and are equal to each other, this means that a three-letter long word of the subject sequence which is currently held in the shift register matches exactly a query word whose location address in query sequence is given in the outputs of the word list memories. In this case, we have a hit condition which needs to be recorded for the following steps of the algorithm. Hence, we register the address of the query word in the query sequence and the location address of the hit in the subject sequence to a FIFO buffer named *Hit FIFO* with 3 control bits. These entries to *Hit FIFO* are processed by the *TwoHitMethod* block assigned to the *Hitfinder* block (see figure 4.12).

4.4.3 TwoHitMethod Block

Figure 4.15 shows a simplified inner structure of the *TwoHitMethod* block. Its aim is to find two non-overlapping hits on the same diagonal within distance

A of each other as explained in subsection 4.2.4 above. In this architecture, there are two FIFOs of the same length and same width namely *Hit FIFO 1* and *Hit FIFO 2* to which the same hit entries from the *Hit FIFOs* of the 5 *Hitfinder* blocks (which belong to the same *HitFinderTwoHit* block) are stored one by one in turn starting from the *Hit FIFO* in the first *Hitfinder* block. The processing of hit entries commences when there are more than two hit entries in the FIFOs. For instance, the a^{th} hit entry of *Hit FIFO 1* and b^{th} hit entry of *Hit FIFO 2* are taken and the hit addresses of these entries are subtracted from each other. If the result is less than 3, we continue with the processing of the a^{th} hit entry in *Hit FIFO 1* and $(b+1)^{th}$ hit entry in *Hit FIFO 2* in the next clock cycle. On the other hand, if the result is bigger than threshold value A, we continue with the processing of the $(a+1)^{th}$ hit entry in *Hit FIFO 1* and $(a+2)^{th}$ hit entry in *Hit FIFO 2* in the next clock cycle.



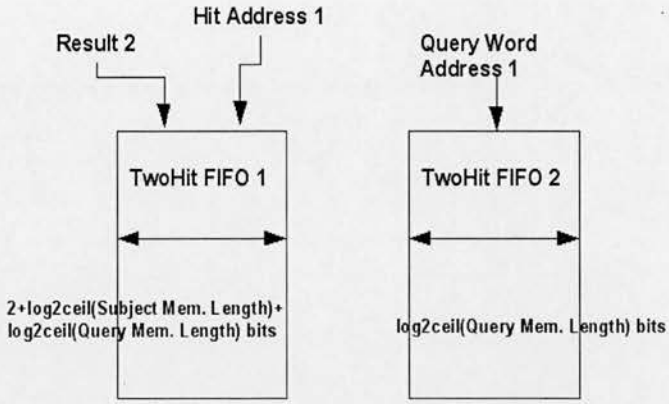


Figure 4.15: *Simplified inner structure of the TwoHitMethod block*

However, if the result of this subtraction is between 3 and threshold value A inclusive, we subtract the query word addresses in the hit entries. If the second subtraction result is not equal to the first one, this means that the two hits are not on the same diagonal, and hence we continue with the processing of the a^{th} hit entry in *Hit FIFO 1* and $(b+1)^{th}$ hit entry in *Hit FIFO 2* in the next clock cycle. If the two results are the same, however, this means that we have two close enough non-overlapping hits on the same diagonal which need to be recorded for the subsequent steps of the algorithm. The two hit cases are recorded to two FIFOs namely *TwoHit FIFO1* and *TwoHit FIFO* 2. The address of the first hit and the distance between the two hits (Result 2 in figure 4.15) are stored in *TwoHit FIFO1* with 2 control bits, whereas the address of the first query word is stored in *TwoHit FIFO* 2. These two-hit entries to the *TwoHit FIFOs* are subsequently processed by the assigned *UngappedExtender* block.

4.4.4 UngappedExtender Block

The *UngappedExtender* block implements the ungapped extension step of the Gapped BLAST algorithm as explained in subsection 4.2.3 above. Each of the two *UngappedExtender* blocks read *Twohit FIFOs* of its 4 assigned *TwoHitMethod* blocks in turn. When the *UngappedExtender* block detects a

two-hit entry in the *Twohit FIFOs* of one *TwoHitMethod* block, the hit address of the first hit, the address of the first query word in the query sequence and the distance between the two hits are all extracted from that entry to compute the start (seed) points of the outward ungapped extension in both directions, on both query sequence (or PSSM) and related subject sequence. Note that the first residue pair of the first hit and the last residue pair of the second hit are the seed points of the outward ungapped extension on the query sequence (or PSSM) and related subject sequence. Afterwards, the inward ungapped extension starts from one seed point and ends at the other seed point where the residue pairs along the extension are scored against a scoring matrix, with the intermediate scores accumulated. However, in case when there is a constructed PSSM, the subject sequence residues along the inward extension are scored against the PSSM. When the inward ungapped extension ends, the outward ungapped extension is launched in both directions. Here again, the residue pairs along the extension are scored against a scoring matrix, with the intermediate score terms accumulated, and added up with the total score obtained from the inward ungapped extension. Again, the subject sequence residues along the outward extension are scored against the PSSM if there is a constructed PSSM. The outward ungapped extension terminates either when the currently computed grand total score falls a certain cut-off value below the highest grand total score obtained so far, or when the extension reaches end of the query sequence (or PSSM) or subject sequence in either direction. In this case, the ungapped extension retracts to its previous state which yielded the highest grand total score. If this highest grand total score exceeds a certain threshold value, the end points of this high scoring ungapped extension in both directions on both query sequence (or PSSM) and subject sequence are registered to two *UngappedResult FIFOs* to be read and processed by the single *GappedExtender* block for the purpose of gapped alignment.

4.4.5 GappedExtender block

The *GappedExtender* block implements the gapped alignment step using the modified Needleman-Wunsch algorithm with the affine gap model. Here, only the gapped alignment score is computed. The final alignment, i.e. with trace-back, is not done on FPGA because of its excessive memory requirement. The *GappedExtender* block reads *UngappedResult FIFOs* of the two *UngappedExtender* blocks in turn to obtain the edge points of the high scoring ungapped alignments produced by these two *UngappedExtender* blocks. These edge points are used to compute the central point of the ungapped alignment from which the gapped alignment on the query sequence (or PSSM) and related subject sequence is launched in both directions. In other words, the seed point of the gapped alignment is computed. Figure 4.16 shows one of the two linear systolic arrays in the *GappedExtender* block which run independently in parallel to perform the modified Needleman-Wunsch algorithm on each side of the seed residue pair. This architecture is deduced from the data dependency graph of the Needleman-Wunsch algorithm as presented in section 4.2 above [45].

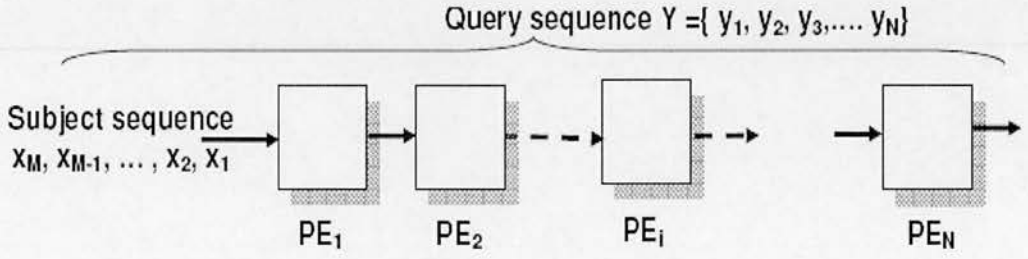


Figure 4.16: *Linear systolic array for the gapped alignment*

The linear systolic array consists of pipelined basic Processing Elements (PEs) each of which performs the dynamic programming equations presented in subsection 4.2.7 above. Before the operation of the array, the query sequence residues at one side of the seed point are shifted through the array. At the end of this shift, each PE holds one query residue. However, in

case when there is a constructed PSSM, columns of PSSM at one side of the seed point are shifted through the array. Following this, the subject sequence residues at the same side of the seed point are shifted systolically through the array during which each PE generates the value of one dynamic programming matrix cell every clock cycle. However, the direction of the cell from which the current value has been derived is not saved since trace-back will not be performed in hardware. Each PE generates one column of the dynamic programming matrix after M cycles where M is equal to the number of subject sequence residues. However, each PE is one cycle behind its predecessor PE due to the fact that computations in PE_{i+1} depend on the computation results in PE_i. Figure 4.17 illustrates the execution of the equations of the original Needleman-Wunsch algorithm on the linear array architecture where diagonal lines cross the matrix cells of dynamic programming matrix whose values are computed at the t^{th} clock cycle.

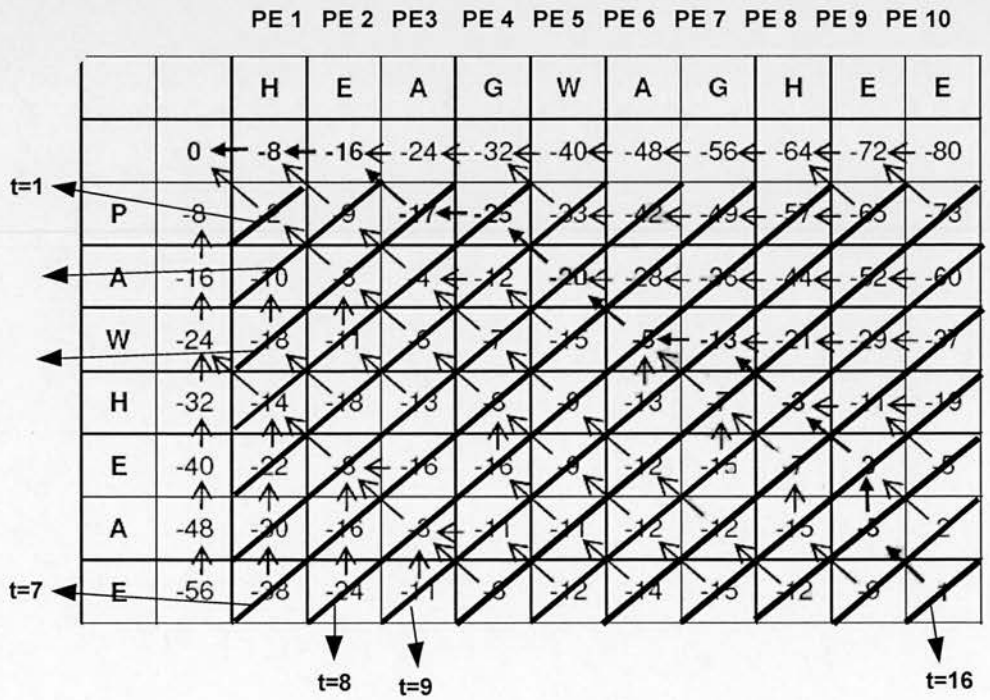


Figure 4.17: *Illustration of the execution of the original Needleman-Wunsch algorithm on the linear systolic array architecture*

The linear array architecture keeps record of the maximum value in the dynamic programming matrix at each PE, calculating its *maximum-so-far* value and broadcasting it to the next PE. The gapped extension in the linear array architecture terminates when the end of the query sequence (or PSSM) or subject sequence is reached in either side, or when the current result in PE_1 is a certain cut-off value below its *maximum-so-far*. Once both of the linear array architectures in the *GappedExtender* block terminate, their maximum values are added up to obtain the score of the gapped alignment. If this score exceeds certain threshold value, the location of the subject sequence in the related subject sequence memory is sent to the host to allow for the subject sequence to be truly aligned with the query sequence (or PSSM) by the high level application software running on the host.

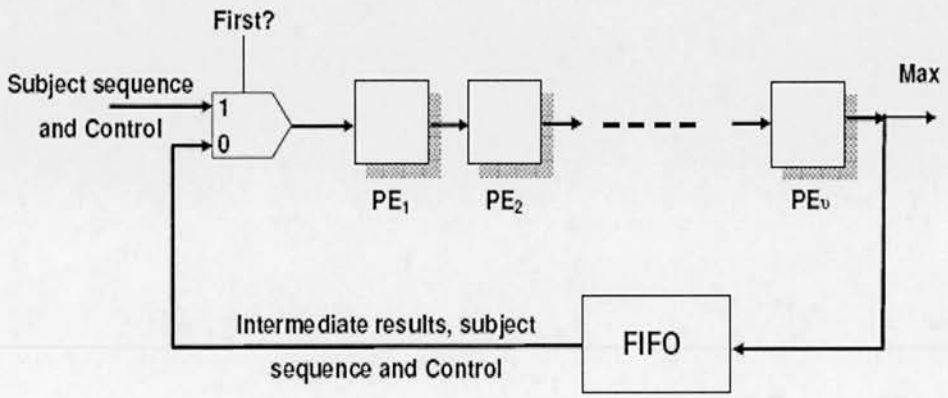


Figure 4.18: *Partitioning and mapping of the modified Needleman-Wunsch algorithm on a fixed size systolic array*

Note that number of PEs in the linear array architectures should be equal to the number of residues in the query sequence (or PSSM) in order to correctly implement the modified Needleman-Wunsch algorithm. However, considering the amount of resources in today's FPGAs, this is sometimes impossible since there could be hundreds or even thousands of residues in the query sequence. To solve this problem, the algorithm is partitioned into small alignment steps which are mapped onto a fixed size linear systolic

array as shown in figure 4.18 above [47] [48]. In this architecture, the alignment process is performed in a number of passes depending on the length of the query sequence (or PSSM), where a FIFO is used to store intermediate results and subject sequence residues from each pass before they are fed back to the input of the array for the next pass. In the proposed implementation, each of the linear arrays in the *GappedExtender* block has 4 processing elements. This could be extended at will, resource permitting.

4.5 Implementation Results

The proposed PSI-BLAST design was captured in the Handel C language to which no specific resource inference or placement constraints were applied. Hence, it can be directly targeted to a variety of FPGA platforms (e.g. Xilinx and Altera FPGAs). The resulting core was compiled into EDIF by Agility's DK5 SP2 suite from which FPGA bitstreams were generated using Xilinx ISE9.2 tool.

A real hardware implementation of the core was achieved on a Celoxica RCHTX FPGA board [50] which has a Xilinx Virtex 4 (xc4vlx160ff1148-11) FPGA and off-chip memory fitted on it. In the proposed implementation, however, the off-chip memory was not used. The operation of the core was tested on the Swiss-Prot protein sequence database [49] with various query protein sequences.

We have also implemented the PSI-BLAST algorithm in C in order to compare the proposed hardware implementation with a pure software implementation. Table 4.1 presents timing performance figures of both hardware and software implementations for one PSI-BLAST iteration for 9 random query protein sequences of various lengths. Note that FPGA execution times listed in the table include the time spent on the host for

query (or PSSM) pre-processing and multiple global alignment which is in the order of milliseconds.

Furthermore, all PSI-BLAST iterations take approximately the same amount of time. So the total execution time for PSI-BLAST is equal to the number of iterations required for the specific query sequence multiplied by the given time value in the table for that query sequence. Note that iterations for a given query sequence continue depending on the preferences of the user or until PSSMs constructed at the end of each iteration start to converge.

The FPGA hardware was clocked at 15 MHz only and the software implementation was executed on an Intel Centrino Duo 2.2 GHz PC with 2 GB RAM. Furthermore, the same threshold and cut-off values were used in both hardware and software implementations at every step of the algorithm. Note that the results from the hardware implementation were the same as those of the reference software code.

As it can be seen from table 4.1, the proposed FPGA implementation results in substantial speed-up compared to software, ranging from 20x to 44x (the speed-up figure depends on the query sequence). The reason behind this high speed-up figure of the FPGA implementation, despite the huge difference in clock frequency, is due to the high level of process parallelism on FPGA as well as deep pipelining. On the other hand, the low clock frequency of the FPGA design was owing to the use of a High Level Language (HLL), namely Handel-C. However, the advantage of using a HLL is reduced design time. Besides this, running the FPGA hardware at 15 MHz has the added bonus of resulting in a much lower power consumption compared to the software implementation.

Although there are several BLAST accelerators reported in the literature [51] [55] [56] [57], these did not implement PSI-BLAST. Hence, comparison with them is not applicable. Furthermore, National Centre for Biotechnology

Information (NCBI) [58] provides a public domain version of BLAST that includes PSI-BLAST functionality. The online utility of NCBI does not report execution time. Moreover, using the NCBI PSI-BLAST software code proved problematic and it was not achievable to get the timing results from it within this work's time frame.

Table 4.1: *Timing performance figures of the hardware and software implementations for one PSI-BLAST iteration for 9 random protein sequences queried in Swiss-Prot protein sequence database*

Query Sequence	No of Residues in Query Sequence	No of Query words	FPGA Execution time (sec)	Software execution time (sec)	FPGA Speed-up
No. 1	111	116	4.47	93.59	20.93
No. 2	214	98	5.04	133.97	26.58
No. 3	368	136	4.37	139.47	31.92
No. 4	459	263	5.94	214.48	36.11
No. 5	565	137	5.80	184.54	31.82
No. 6	635	140	5.45	197.52	36.24
No. 7	746	117	6.93	237.31	34.23
No. 8	864	240	7.12	315.37	44.29
No. 9	985	53	5.46	198.23	36.31

4.6 Conclusions

The detailed design and FPGA implementation of the PSI-BLAST algorithm has been presented in this chapter. The architecture of the proposed implementation is composed of various blocks each of which performs a specific step of the algorithm in parallel. Furthermore, the FPGA core is parameterized in terms of the sequence lengths, scoring matrix, gap penalties and cut-off and threshold values. The resulting implementation outperforms an equivalent desktop-based software implementation by at least one order-

of magnitude. Moreover, it was designed in the Handel-C language which makes it FPGA-platform-independent. As a result, the same core can be ported to other FPGA architectures from different vendors. However, one disadvantage of using Handel-C was that the achieved clock frequency for the FPGA design was relatively low for a Virtex 4 FPGA chip.

Future work for this case study includes a multi-threaded implementation of various flavours of BLAST (including the PSI-BLAST algorithm) and other sequence analysis algorithms with a web interface that allows users to submit queries remotely to an FPGA-based server. Note that Verilog HDL instead of Handel-C will be used for the rest of the case studies in this thesis since the company owning the development tools for the Handel-C (i.e. Celoxica) sold its software because of the financial troubles which resulted in the demise of Handel-C language.

Chapter 5

High Performance Phylogenetic Analysis with Maximum Parsimony on a FPGA Parallel Computer

5.1 Introduction

Phylogenetic analysis is the investigation of the evolution and relationships among organisms that is widely used in the fields of system biology and comparative genomics [3]. It is particularly important in drug and vaccine development. In molecular based phylogenetic analysis, the relationship between species is estimated by inferring the common history of their genes and then phylogenetic trees are constructed to illustrate evolutionary relationships among genes and organisms [3] [6].

However, phylogenetic tree construction is a computationally intensive operation and desktop computers alone cannot be relied upon to perform this task within acceptable execution times. This is because the number of theoretically possible tree topologies grows exponentially with the number of species under consideration. For instance, it takes over 30 hours to construct the phylogenetic tree for 12 species on a 2.2 GHZ Intel Centrino Duo machine with 2 GB of RAM. Hence, it is mandatory to utilize faster computing platforms such as Field Programmable Gate Arrays (FPGAs). These have indeed been recently proposed as an efficacious and efficient implementation platform also for phylogenetic analysis due to their flexible computing and memory architecture which gives them ASIC-like performance with the added programmability feature [59] [60] [61] [62] [63] [64] [65] [66][67].Hence, we chose FPGAs over ASICs because of their reconfigurability feature and shorter development time which results in lower Non-Recurring Engineering (NRE) costs.

There are various phylogenetic tree construction and phylogenetic analysis methods using different strategies. In this chapter, we concentrate on the Maximum Parsimony (MP) method which is one of the most widely used and most accurate tree construction method [6].The design and implementation of the FPGA core for parsimony analysis employing Sankoff's dynamic programming algorithm is presented in this chapter. Systolic array architecture was selected in the proposed design due to its several benefits for the design. First of all, systolic structures have inherently massive, local, parallelism potential at both coarse and fine-grain levels. Coarse-grain parallelism is through the number of parallel processing elements, whereas the fine-grain parallelism is achieved in each processing element. This is the main reason behind the accomplished high speed-up values. Furthermore, since only the processing element at the border of the array can communicate with the host, communications in the architecture are mostly local (i.e. between and within the processing elements). Hence, communication paths have short delays resulting in high clock frequencies and consequently, high throughput. Moreover, systolic architectures can be easily implemented on FPGAs as demonstrated in the literature.

A real hardware implementation of the designed core was achieved on the nodes of an FPGA supercomputer, named Maxwell, which consists of 64 Virtex-4 FPGA chips. To our knowledge, this is the first FPGA implementation of this method for nucleotide sequence data ever reported in the literature. FPGA implementations of other phylogenetic analysis methods and different molecular data have been reported in the past, however, as described in more detail in section 5.3.

The remainder of this chapter will first present essential background information on phylogenetic analysis and then discuss related prior works in the literature. Following this, the Maximum Parsimony (MP) method for molecular based phylogenetic tree construction will be detailed. Furthermore,

the design and implementation of the proposed FPGA core for the MP method will be elaborated. Following this, implementation results are presented and then evaluated comparatively with equivalent software implementations running on a desktop computer. Finally, conclusions are laid out with plans for future work.

5.2 Phylogenetic Analysis

Evolution and relationships among organisms can be investigated in different ways. Although morphology is the classic method of estimating relationships, continuously growing molecular information such as nucleotide or amino acid sequences can also be utilized to infer evolutionary relatedness.

Molecular-based phylogenetic analysis estimates the relationship between species by inferring the common history of their genes through comparing homologous sites with each other. For this reason, sequences under investigation are aligned by some specific algorithms so that homologous sites form columns in the alignment. These alignments are used to construct phylogenetic trees which illustrate evolutionary relationships among genes and organisms.

5.2.1 Phylogenetic Trees

Diagrams depicting the relationship of species resemble the structure of a tree. Hence, they are called phylogenetic trees. There are two types of phylogenetic tree, rooted or unrooted. Rooted phylogenetic trees are drawn with a root to the left. Figure 5.1 shows an example rooted phylogenetic tree where the root node is indicated. It can be seen that phylogenetic trees are strictly bifurcated (binary).

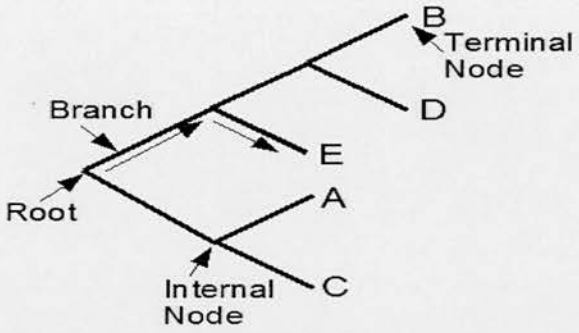


Figure 5.1: *Rooted phylogenetic tree*

Phylogenetic trees have some number of External (Terminal) nodes which are often called Operational Taxonomic Units (OTUs). OTUs represent existing taxa (i.e. a group of one or more organisms). For instance, B, D, E, A and C are all terminal nodes in the phylogenetic tree shown in figure 5.1. Also, phylogenetic trees have some number of internal nodes which are called Hypothetical Taxonomic Units (HTUs). HTUs represent hypothetical ancestors of OTUs. Nodes other than root and terminal nodes are internal nodes in phylogenetic tree as shown in figure 5.1. Furthermore, the lines between the nodes are branches. The branching pattern is called the topology of the tree. Figure 5.2 shows an example unrooted phylogenetic tree.

An unrooted phylogenetic tree does not indicate the direction of evolution process as seen in figure 5.2 since it is not known which node represents the ancestor of all OTUs. However, in a rooted tree, there is a root node which leads to the common ancestor of all OTUs in it.

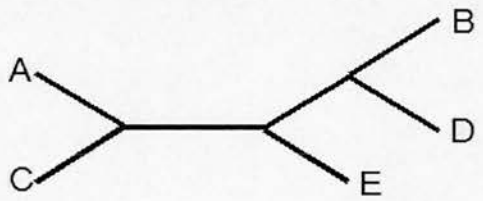


Figure 5.2: *Unrooted phylogenetic tree*

In figure 5.1, arrows indicate the direction of evolution from root to terminal node E for instance. Note that an unrooted phylogenetic tree can be rooted

with a method named outgroup rooting if a set of the most distantly related OTUs (i.e. outgroup) can be formed. Otherwise, the midpoint rooting method can be utilized. Both of these methods are described in detail in [3].

5.2.2 Methods to Reconstruct Phylogenetic Trees

There are various methods to generate phylogenetic trees from nucleotide acid sequence alignments in molecular data based phylogenetic analysis. All of these methods use certain evolutionary assumptions. If these assumptions apply to the date set, the methods perform well.

These methods can be grouped in one way according to whether they use discrete character states or pairwise distance matrices. Character-state methods regard each position in the aligned sequences as a character and the nucleotides and amino acids at that position as states. All characters are compared separately and independently from each other. One advantage of these methods is that they can reconstruct the character state of the internal nodes which represent ancestral taxa.

On the other hand, distance-matrix methods produce a pairwise distance matrix and then infer relationships of the OTUs from that matrix. Although distance-matrix methods cannot reconstruct the character state of ancestral nodes like character-state methods, they are much less computer-intensive, and hence faster.

Molecular based phylogenetic analysis methods can also be grouped according to whether they consider all possible trees or cluster OTUs stepwise to obtain the single best tree. Exhaustive-search methods evaluate all theoretically possible tree topologies for a given number of OTUs using a certain criteria and choose the best one as true phylogeny. One advantage of these methods is that it is possible to assess the confidence in the best tree obtained by comparing it with the second best tree. However, the number of

possible trees grows exponentially as the number of taxa increases. Hence, these methods require very high computing power.

On the other hand, stepwise-clustering constructs a single tree by following specific clustering algorithms. Hence, these methods can cope with large numbers of OTUs. However, there is no way to estimate the confidence in correctness of a tree obtained since only one tree is produced in these methods.

Table 5.1 below lists phylogenetic tree construction and phylogenetic analysis methods classified according to the strategy they use. Note that most of the distance-matrix methods utilize stepwise clustering to construct the best tree whereas all character-state methods search the tree space exhaustively to find the best tree.

Table 5.1: *Classified phylogenetic tree construction and phylogenetic analysis methods*

	Exhaustive Search	**Stepwise Clustering**
Character State	Maximum Parsimony (MP)[68] Maximum Likelihood (ML) [69]	
Distance Matrix	Fitch-Margoliash[70]	UPGMA [71] Neighbour-Joining (NJ) [72]

In this work, a discrete character method widely used in molecular phylogenetic analysis, namely the MP method, was employed to find the

best phylogenetic tree for a given number of taxa where all theoretically possible tree topologies are evaluated. There are some faster heuristic approaches to this method, however, which attempt to heuristically find optimal solutions to the best tree topology problem [3]. Although these approaches have shorter run times in software, they are approximate and hence do not guarantee to find the best tree topology. With faster implementation platforms, however, this compromise need not take place, and that is why we have chosen to accelerate the MP method with exhaustive search on FPGA hardware in this work.

5.3 Prior Work

Although the FPGA implementation of the MP phylogenetic tree construction for nucleotide sequence data has never been reported in the literature, there exist some papers discussing the hardware implementations of the other phylogenetic analysis methods for different types of molecular data. For instance, [59], [60] and [61] describe the design of FPGA-based coprocessor architecture to accelerate the reconstruction of MP phylogenies for gene-arrangement data. The design performs a parallelized version of the breakpoint median computation which is the most time consuming component of the reconstruction. Reference [59] reports that the breakpoint median hardware core achieves a 1005x speed-up over the related desktop software solely for the computation and a 417x speed-up when the architecture is used to accelerate the entire reconstruction procedure.

Moreover, [62], [63] and [64] present high performance FPGA implementations for tackling the tree evaluation process for nucleotide sequences under the Maximum Likelihood (ML) criterion in order to speed-up the tree reconstruction. Reference [64] proposes a Hardware/Software (HW/SW) system for solving the tree reconstruction problem using the

Genetic algorithm for Maximum Likelihood (GAML) approach which yields speed-up of 30x to 100x compared to a software solution. Furthermore, [62] extends this HW/SW co-design to a more powerful embedded computing platform to achieve much faster computation speed for phylogeny inference. Also, the FPGA logic design is based on the idea of partial likelihood this time to improve the tree likelihood evaluation process.

On the other hand, [65] presents the application of Custom Computing techniques to speed up the Unweighted Pair Group Method with Arithmetic Means (UPGMA), which is the oldest and simplest method used to generate phylogenetic trees from distance data, by a factor of 100 against equivalent software running on a desktop computer, for nucleotide sequences. The paper reports on the conducted experiments and discusses how custom computing techniques can be utilized to accelerate the performance of phylogenetic analysis algorithms on high-performance computing engines.

Finally, recent papers [66] and [67] present an architecture which computes the Phylogenetic Likelihood Function (PLF) through the implementation of a massive floating point arithmetic unit using a large number of DSP blocks in Xilinx FPGAs [76]. PLF is the most time-consuming kernel of all ML based programs for the reconstruction of evolutionary relationships. The architecture presented in [66] is reported to achieve speed-ups ranging from 1.6 up to 7.2 compared to a general purpose computer running a highly optimized and parallelized software implementation of the PLF.

5.4 Maximum Parsimony

The MP method is one of the most widely used discrete character method in molecular phylogenetic analysis [6]. It operates on a character-state matrix which is typically an aligned set of DNA or protein sequences where the

states are the nucleotides (i.e. A, C, G, and T) for DNA sequences and letters of 20 amino acids for protein sequences.

The MP method operates by defining an objective function which returns a score for any input tree topology. This tree score is used to rank all possible trees according to the chosen optimality criterion to find the optimal tree topologies. The parsimony objective function and an algorithm to solve it will be discussed in subsections 5.4.1 and 5.4.2, respectively. Searching the tree space to find the optimal trees under the parsimony criterion will be detailed in subsection 5.4.3. Following that, subsection 5.4.4 will shortly present a software tool for phylogenetic inference named PAUP.

5.4.1 Parsimony Analysis

Parsimony criterion is the number of character changes required to explain all nodes of a tree at every sequence position for a given set of aligned sequences. The total amount of character change required by any given tree is called the length of that tree. In parsimony analysis, the aim is to find the tree topologies with the smallest length. Calculating the length of a given tree will be explained and illustrated later in this subsection.

An unrooted binary tree for T taxa contains T-2 internal nodes, 2T-3 branches and T terminal nodes representing sequences of taxa. The length L of an arbitrarily chosen tree r under parsimony criterion is given by the following equation where l_j is the length for single site j:

$$L(r) = \sum_{j=1}^{N} l_j \quad (5.1)$$

In (5.1), N is the number of sites in the sequence alignment and l_j corresponds to the minimum amount of character change implied by a reconstruction

where a character-state x_{ij} is assigned to each node i for each site j. Note that character-state assignment of the terminal nodes is fixed by the input sequences of T taxa. Equation 5.2 shows the calculation of l_j.

$$l_j = \sum_{k=1}^{2T-3} C_{a(k),b(k)} \quad (5.2)$$

In (5.2), a(k) and b(k) represent the states assigned to the nodes at either end of branch k whereas c_{xy} is the cost of change from state x to state y. There are various cost schemes which can be represented as a cost matrix that assigns a cost for the change between each pair of character states. Two cost matrices for nucleotide data are shown in figure 5.3. The matrix at the left hand side of figure 5.3 assigns equal cost of 1 if the nucleotides are different or 0 if they are the same. On the other hand, the right hand side matrix assigns a greater cost to transversions than to transitions. An important point is that cost matrices are symmetric meaning that c_{xy} is equal to c_{yx}. As a consequence, the length of a tree is the same regardless of the position of the root. Therefore, the search among tree space can be conducted over unrooted trees rather than rooted trees.

	A	C	G	T
A	0	1	1	1
C	1	0	1	1
G	1	1	0	1
T	1	1	1	0

	A	C	G	T
A	0	4	1	4
C	4	0	4	1
G	1	4	0	4
T	4	1	4	0

Figure 5.3: *Two possible cost matrices*

Although there are various algorithms to determine l_j, it would be helpful to illustrate the calculation of tree length for one site by evaluating all possible r^{T-2} character-state reconstructions where r is the number of states (r = 4 for DNA sequences or r =20 for protein sequences). As an example, we will

consider the alignment of 4 DNA sequences (i.e. K, L, M, and N) shown in figure 5.4 assuming that the lengths for the first j-1 sites have been calculated and the length of site j is going to be calculated next.

```
      J
K  ...ACGTCGT...
L  ...TGCCTGA...
M  ...CGTCGTC...
N  ...GCTACTG...
```

Figure 5.4: *Example alignment of 4 nucleotide sequences*

The tree topology that will be evaluated in our example is shown in figure 5.5. The number of character-state reconstructions to be evaluated for the two internal nodes is $4^{(4-2)} = 16$. The lengths implied by four of these reconstructions under equal cost scheme are shown in figure 5.6. Note that the minimum length obtained from one of these 16 possible combinations of state assignments will be the tree length and the associated reconstruction will determine the default states of the two internal nodes at site j. Note that the last reconstruction in figure 5.6 corresponds to the optimal case of all 16 possible reconstructions with the minimum length of 2.

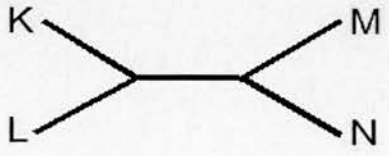


Figure 5.5: *Tree topology to be evaluated as an example*

The brute-force method used in this example can be applied to every site in the sequence alignment to obtain the minimum lengths and summing these lengths will give the total length. However, there is a need for better ways to determine the minimum lengths since the evaluation of r^{T-2} reconstructions will take considerable amount of time and storage when the number of taxa

under consideration grows. For this purpose, we will employ a straightforward dynamic programming algorithm namely Sankoff's algorithm [73] which is illustrated in subsection 5.4.2 below.

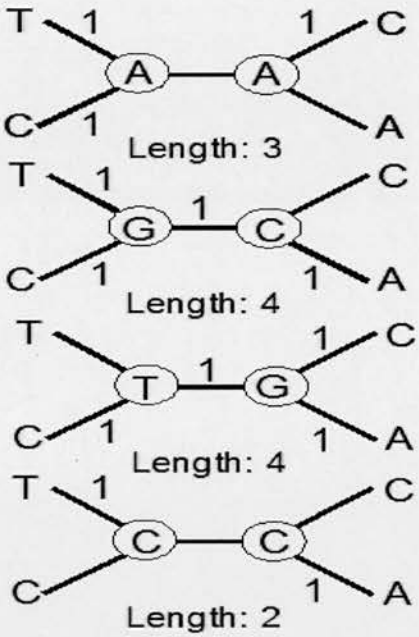


Figure 5.6: *Four possible combinations of state assignments to the two internal nodes and the resulting lengths*

5.4.2 Sankoff's Algorithm

Dynamic programming algorithms operate by solving a set of subproblems and then assembling those solutions to find an optimal solution for the whole problem. In the case of Sankoff's algorithm, the best length achievable for each subtree is determined given each of the possible state assignments to each node while moving from the tips toward the root of the tree. An optimal length for the full tree is obtained when the root is reached.

Sankoff's algorithm operates on conditional subtree length vectors which are depicted by rectangular boxes in the tree shown in figure 5.7. It can be seen that for each node i, there is an associated conditional subtree vector S_i

containing the minimum possible lengths s_{ik} of the subtree descending from node i if it is assigned state k. Working from the tips toward the root, the algorithm proceeds by filling in the vector at each node based on the values assigned to the pair of vectors above the regarding node. Note that for the terminal nodes, vectors are initialized to 0 for the states actually observed in the sequence alignment or to infinity otherwise. The algorithm will be illustrated in hardware implementation section to ease the comprehension. An important point is that for symmetric cost matrices, an unrooted tree can be arbitrarily rooted to determine the minimum tree length in this algorithm.

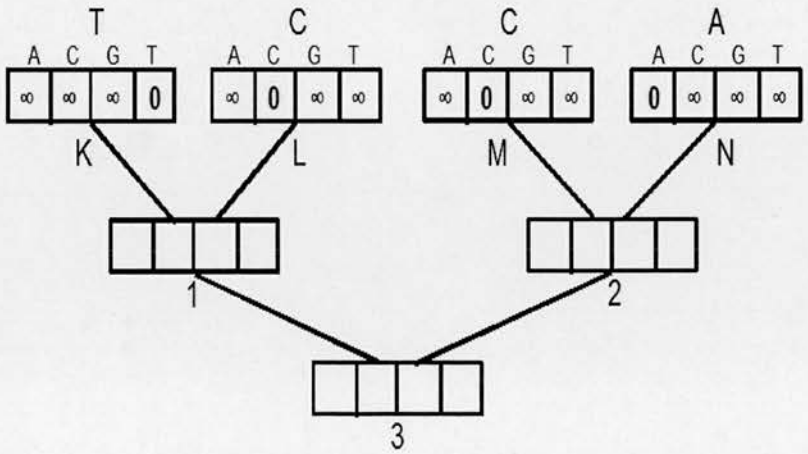


Figure 5.7: *An example tree topology with conditional subtree length vectors for each node*

5.4.3 Searching for Optimal Trees

Since the length of a tree under parsimony criterion can be calculated using Sankoff's algorithm, the search over tree space can now be started to find the optimal tree. However, there is a need for an algorithm to generate all possible trees to be evaluated under parsimony criterion. Such an algorithm recursively adds the n^{th} taxon in a stepwise manner to all possible trees

containing the first n-1 taxa until all T taxa have been joined. This algorithm will be illustrated for five taxa in the following part of this subsection.

We begin with the only tree for the first three taxa and then connect the fourth taxon to each of the three branches on this tree to generate all 3 possible unrooted trees for the first four taxa. This process is illustrated in figure 5.8.

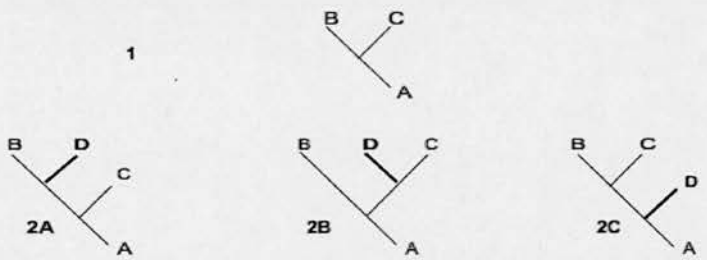


Figure 5.8: *Generation of all 3 possible unrooted trees for the first four taxa*

Furthermore, we connect the fifth taxon to each branch (5 branches per tree) on each of these 3 trees to yield all 15 possible unrooted trees for the five taxa as shown in figure 5.9.

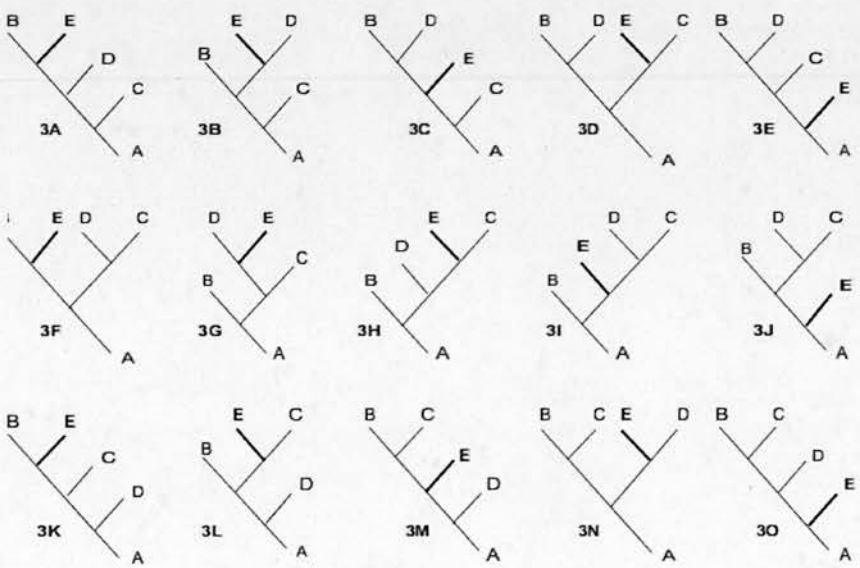


Figure 5.9: *Generation of all 15 possible unrooted trees for the five taxa*

The number of possible trees grows by a factor increasing by two with each additional taxon as expressed in equation 5.3 below where B(t) is the number of unrooted trees for t taxa. Table 5.2 shows the number of possible unrooted trees for a given number of taxa.

$$B(t) = \prod_{i=3}^{t} (2i - 5) \quad (5.3)$$

Table 5.2: *Number of possible unrooted trees for up to 12 taxa*

Number of Taxa	Number of Unrooted Trees
3	1
4	3
5	15
6	105
7	945
8	10395
9	135135
10	2027025
11	34459425
12	654729075

5.4.4 PAUP Software Tool

PAUP (Phylogenetic Analysis Using Parsimony) [74] is a phylogenetic analysis program using NEXUS format for input data files. It includes support for the maximum parsimony, maximum likelihood and distance methods as well as some additional capabilities. Details of PAUP can be found in its user manual, command reference manual and quick start tutorial in [75].

Several versions of PAUP are available with support for a full graphical user interface (Macintosh), a partial graphical user interface (Microsoft Windows),

and a command-line only interface (Unix/Linux and Microsoft Windows console). The Macintosh interface allows for the execution of commands via menus and command line whereas the Windows and Unix/Linux interfaces are almost entirely command-line driven. Some menu functions are available in the Windows interface.

Although there is no evidence to suggest that PAUP is the most efficient software tool available in the area of phylogenetic analysis, it is widely used by the Bioinformatics community [3]. Hence, we are going to compare the proposed FPGA hardware implementation with PAUP software.

5.5 Hardware Implementation

Sankoff's algorithm requires calculation of the conditional subtree length vector for every internal node in a tree. The algorithm will be illustrated first in this section based on the tree shown in figure 5.7 using the cost matrix shown at the right hand side of figure 5.3. We start with the calculation of the vector values of node 1 (see figure 5.7). For each element k of this vector, the costs associated with each of the four possible state assignments to each of the child nodes K and L and the cost needed to reach these states from state k (obtained from the cost matrix shown at the right hand side of figure 5.3) are considered. For node 1, these calculations are simple since it is ancestral to two terminal nodes. Hence, only one state needs to be considered for each child node. For example, the minimum length of the subtree descending from node 1 assuming that state A is assigned to node 1 is equal to the sum of the cost of a change from A to T in the left branch and the cost of a change from A to C in the right branch ($s_{1A} = c_{AT} + c_{AC} = 4 + 4 = 8$). In the same manner, s_{1C} is the sum of c_{CT} (left branch) and c_{CC} (right branch) giving the value of 1. Continuing like this, we obtain the entire conditional subtree length vector for node 1 as shown in figure 5.10. With the same procedure,

we compute the elements of the vector for node 2 (see figure 5.7) as shown in figure 5.11. On the other hand, calculations for node 3 (see figure 5.7) are more complicated since we must consider each of the four state assignments to each of the child nodes 1 and 3 for each state k at its node. Figure 5.12 shows the computation of the conditional subtree length vector for node 3.

The conditional vector S_3 contains the minimum possible lengths for the full tree given each of the four possible state assignments to the root. The minimum of these tree lengths is the tree length we seek, which is 5 in our case as can be seen in figure 5.12. Note that different rooting of the tree in figure 5.7 would yield the same length.

This algorithm provides a way to calculate the minimum tree length for any character on any tree under any cost scheme. The total length of a given tree can be computed by repeating the mentioned procedure for each character in the sequence alignment and then adding up all of the obtained minimum lengths for the characters which can be multiplied beforehand by different weights depending on the importance of the characters in the alignment.

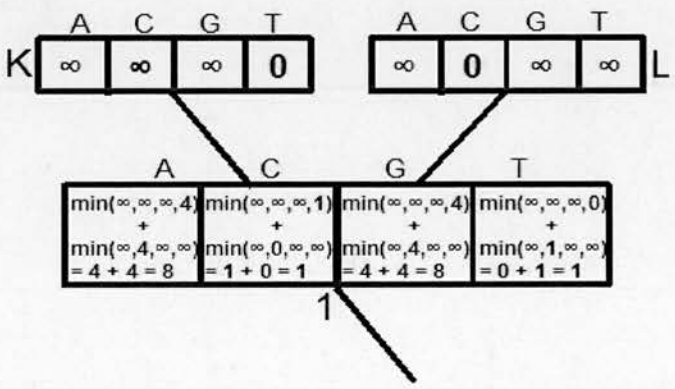


Figure 5.10: *Calculation of the conditional subtree length vector for node 1*

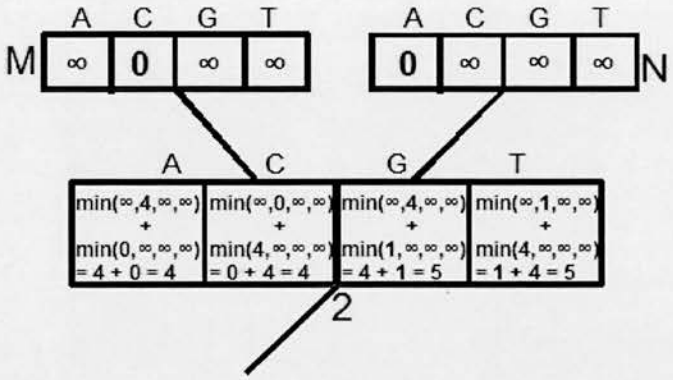


Figure 5.11: *Calculation of the conditional subtree length vector for node 2*

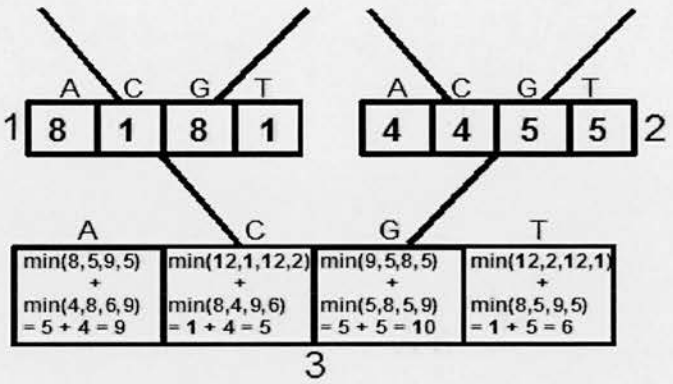


Figure 5.12: *Calculation of the conditional subtree length vector for node 3*

Figure 5.13 shows the hardware architecture which computes the subtree length vectors of the nucleotides (i.e. A, C, G, and T). In this architecture, registers *LregA*, *LregC*, *LregG* and *LregT* represent the elements of the vector of the left hand side upper node (e.g. node 1 in figure 5.12) whereas registers *RregA*, *RregC*, *RregG* and *RregT* represent the elements of the vector of the right hand side upper node (e.g. node 2 in figure 5.12).

Each of these registers are added up with three different specific cost values (i.e. C_{A2C}, C_{A2G}, C_{A2T}, C_{C2G}, C_{C2T}, C_{G2T}) to obtain three subscores (e.g. S_A2C_L, S_A2G_L, S_A2T_L for *LregA*) and then each register and its associated three subscores (e.g. S_C2A_L, S_G2A_L, S_T2A for *LregA*) are inputted to the combinational block *Min* to find the minimum of these values

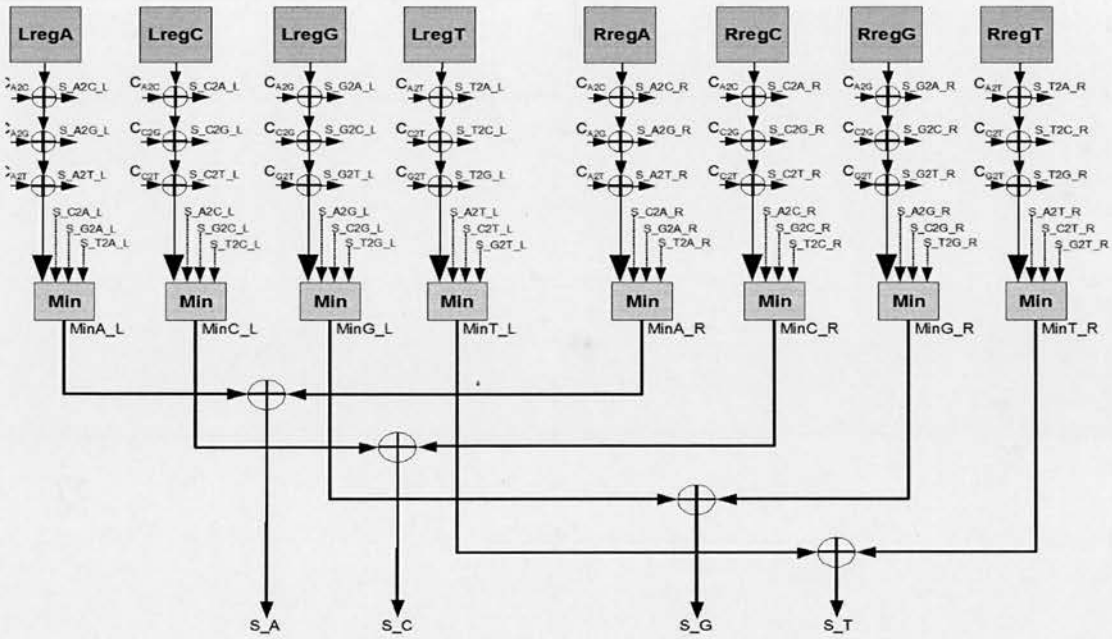


Figure 5.13: *Simplified hardware architecture for the conditional subtree length vector calculation of the nucleotide sequence*

MinX_Y (X=A, C, G or T and Y=L or R). Furthermore, two minimum values for each nucleotide (e.g. MinA_L and MinA_R for A) are added to obtain the scores for each nucleotide (i.e. S_A, S_C, S_G, S_T) which are the elements of the vector of the target node (e.g. node 3 in figure 5.12).

5.5.1 Parallel Implementation of Sankoff's Algorithm

An important point is that some of these node vectors can be computed at the same time. For example, in the 10-taxa tree shown in figure 5.14, computations for nodes on the same line can be done in parallel. Vectors of nodes on different lines are computed consecutively starting from the first line until the root node is reached. FPGAs can take advantage of this parallelism of Sankoff's algorithm to accelerate it by computing several node vectors concurrently. For the tree topology in figure 5.14, FPGA hardware would calculate 2, 4 and 2 node vectors in parallel in the first, second and third clock cycles, respectively. In the fourth clock cycle, a vector for the root

node would be calculated to obtain the minimum length (score) of the tree. Hence, the score of the tree is computed in four clock cycles in total rather than the nine cycles required in the case of sequential node calculations. Note that the tree under consideration should be rooted in a way so that the left and right subtrees of the rooted tree will have almost the same number of taxa to maximize the available parallelism (i.e. tree balancing).

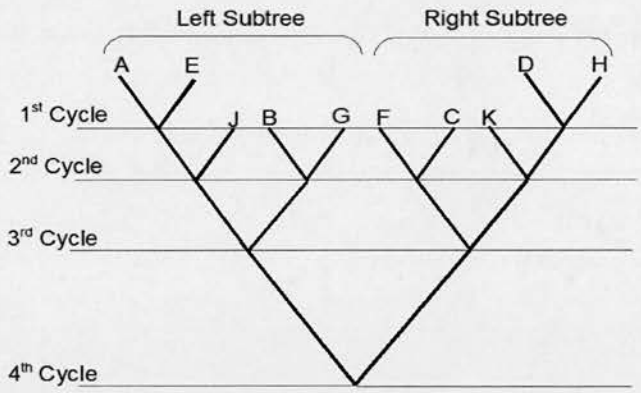


Figure 5.14: *Tree topology illustrating the parallelism of Sankoff's algorithm*

The hardware architecture which computes node vector values (see figure 5.13) can be used within a linear systolic array to implement the complete Sankoff's algorithm in a parallel manner as explained in subsection 5.5.2 below. Also, subsection 5.5.3 elaborates on the inner structure of the processing element constituting this array.

5.5.2 Linear Systolic Array Implementation of Sankoff's Algorithm

Figure 5.15 shows a linear systolic array which implements Sankoff's algorithm. It is composed of several processing elements PE_i, each of which contains a number of sub-elements with similar architecture as shown in figure 5.13, in order to compute node vector values. Each *PE* (processing element) calculates the score of a different tree topology in parallel independently from each other. Hence, the total number of *PE*s is equal to

the number of theoretically possible tree topologies for the given number of taxa.

The architecture in figure 5.15 also comprises two *FIFO*s and an *FSM*. The *Input FIFO* is fed by high level application software running on the host computer with cost matrix, tree topology and sequence alignment data in respective order. Concurrently, the linear array reads the *Input FIFO* to first get the values of the chosen cost matrix which are then shifted through the processing elements within the array. Following this, tree topology vectors whose number is equal to the number of possible tree topologies are read and shifted through the array independently to configure each *PE* to operate on one specific tree topology.

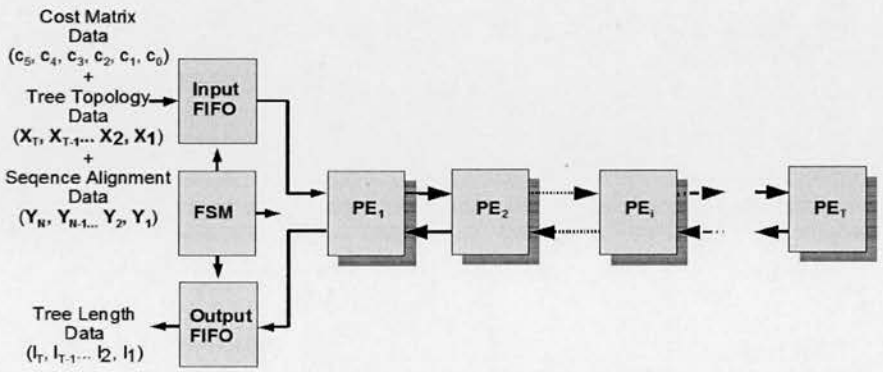


Figure 5.15: *Linear systolic array for Sankoff's algorithm*

Finally, nucleotide vectors composed of nucleotides at one site of the sequence alignment (e.g. site j in figure 5.4) under consideration are read and shifted through the array one by one so as to enable the processing elements to compute the scores for that specific alignment site for all tree topologies in parallel. When the first *PE* finishes its operation for one nucleotide vector, another vector is read and shifted through the array until there is no more nucleotide vector left in the *Input FIFO*. When every *PE* is done with the last nucleotide vector, the total tree scores computed by accumulating the score of each alignment site during the whole process in each *PE* are shifted

backwards through the array into the *Output FIFO* to be read by the application software. The *FSM* coordinates all of these operations of the *PEs*, *Input FIFO* and *Output FIFO* in accordance with control data coming from the application software running on the host.

As mentioned before, the number of *PEs* in the linear array is equal to the number of possible tree topologies. However, considering the amount of resources in today's FPGAs, this is not always feasible since there could be hundreds or even thousands of theoretically possible tree topologies for a given number of taxa as seen in table 5.2. To solve this problem, the algorithm is partitioned into small steps which are mapped onto a fixed size linear systolic array as shown in figure 5.16 [47] [48].

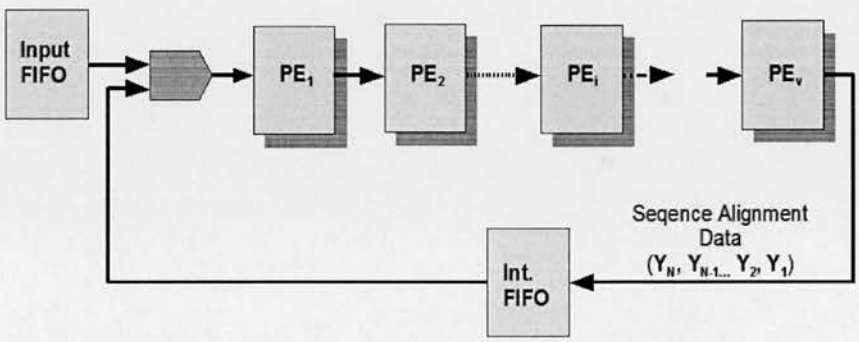


Figure 5.16: *Partitioning and mapping of Sankoff's algorithm on a fixed size systolic array*

In this architecture, the tree evaluation process is performed in numerous iterations (or passes) for each set of tree topologies. Obviously, the number of iterations depends on the number of possible tree topologies. The additional *FIFO* in this architecture is used to store the sequence alignment data shifted in the first pass which will be read by the array in the next passes when the time comes for shifting all nucleotide vectors through the array. On the other hand, the *Input FIFO* is read to obtain a new set of tree topology vectors at each pass while there is no need to read cost matrix data after the first pass.

5.5.3 Architecture of a Processing Element

Figure 5.17 shows the simplified inner structure of a processing element which is mainly composed of *DpathL* and *DpathR* blocks. Data read from the *Input FIFO* is shifted through the array via linked *Data* registers in the *PE*s as illustrated in figure 5.15 to be used by *DpathL* and *DpathR* blocks which implement Sankoff's algorithm on the left and right subtrees (see figure 5.14) of a tree topology, respectively. In the architecture, the score of the right branch computed by the *DpathR* is inputted to the *DpathL* for the calculation of the 4 elements of the root node vector which are then inputted to the *Min* block to find the minimum of them. The minimum value is the score of the tree at a specific site of the sequence alignment (e.g. site j in figure 5.4) under consideration. This score is multiplied by the *Weight* register which holds the weight of that site within the alignment and then, the obtained result is added to the *TotScore* register which will hold the total score of the tree topology when the computations for the last site in the alignment is finished in the *PE*. The value of the *TotScore* registers which are linked to each other are shifted backwards into the *Output FIFO* as illustrated in figure 5.15 when every *PE* in the array is done with the computation of the total score of its assigned tree topology.

Figure 5.18 shows the simplified inner structure of the *DpathL* block which contains one *DpathUnitL* block and three *DpathUnitR* blocks. *DpathUnitL* and *DpathUnitR* blocks are responsible from conditional node vector calculation (see figure 5.13). Each *DpathUnitR* has 4 data inputs two of which are coming from outside the *DpathUnitL* and *DpathUnitR* blocks, one of which is coming from its *Min & Add Op.* block (whose inner structure is shown in figure 5.13) and last of which is coming from the *Min & Add Op.* block of the right hand side neighbour *DpathUnitR* block. On the other hand, *DpathUnitL* has 5 data inputs four of which are like those of *DpathUnitR* and the fifth one (input R) is coming from outside the *DpathL*. Also, the *DpathUnitL* and *DpathUnitR*

blocks have control inputs C_x that determine which data inputs will be registered. For example, if C_c is asserted, input C will be stored in *Lreg_1* in the next cycle. With various combinations of these control signals, *DpathL* can compute conditional vectors of multiple nodes (up to 4 nodes) in various topological forms at the same cycle. Furthermore, *DpathUnitL* block of the *DpathL* is employed to compute the root node vector of the tree under consideration using its input R coming from *DpathR* (see figure 5.17).

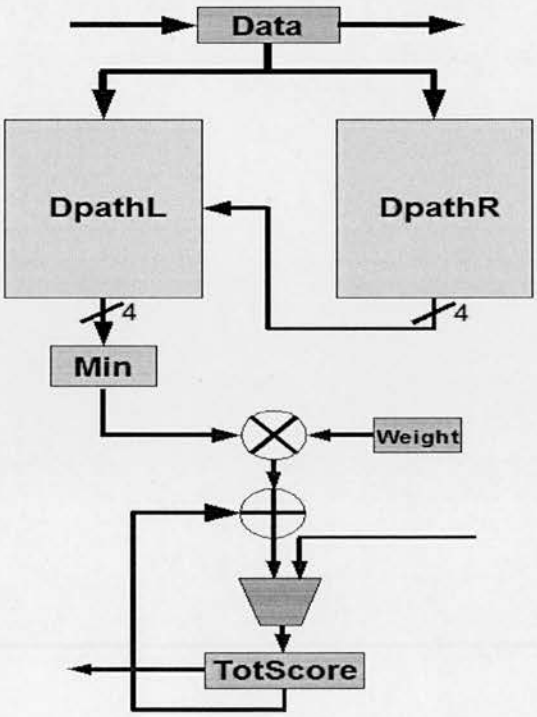


Figure 5.17: *Simplified inner structure of a processing element (annotated numbers represent number of words)*

Note that *DpathR* has a similar structure to that of *DpathL* but it has one less *DpathUnitR* block. So, it can compute conditional vectors of up to 3 nodes concurrently. *DpathL* will be explained more in detail next in this subsection.

DpathL block incorporates four arrays of registers (*CostReg*, *TreeStructReg*, *TaxaOrderReg* and *ResidueReg*) which are fed by the *Data* register shown in figure 5.17. *CostReg* stores the values of the cost matrix which are used within

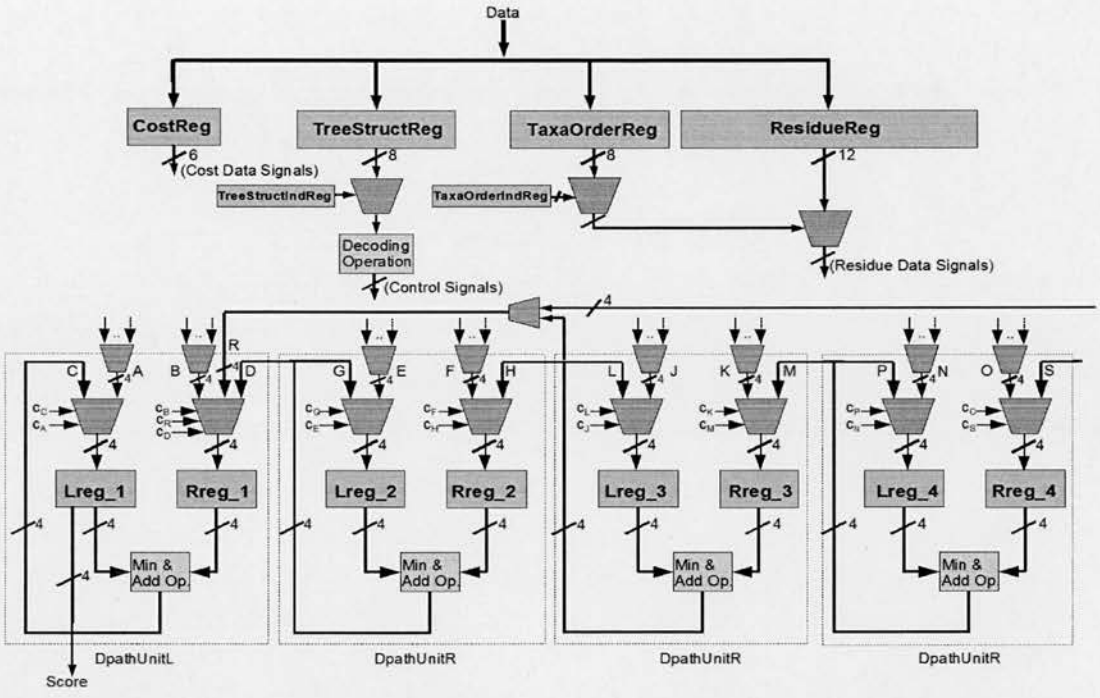


Figure 5.18: *Simplified inner structure of the DpathL block (annotated numbers represent number of words)*

Min & Add Op. block of each *DpathUnitL* and *DpathUnitR* blocks while *TreeStructReg* contains control configurations for the specific tree topology. *TreeStructReg* is decoded to obtain appropriate control signals for all multiplexers in the datapath with the help of *TreeStructIndReg* incrementing by one at every cycle. Furthermore, *ResidueReg* keeps the nucleotides of the specific site in the sequence alignment a set of which is applied to the data inputs of the *DpathUnitL* and *DpathUnitR* blocks (i.e. inputs A, B, E, F, J, K, N, and O) appropriately at every cycle. The applied set of nucleotides is determined by the *TaxaOrderReg* with the help of *TaxaOrderIndReg* which is incremented every cycle by some value depending on the current control configuration in *TreeStructReg*. Note that the values of *TreeStructReg* and *TaxaOrderReg* at a time constitute a tree topology vector whereas contents of *ResidueReg* are obviously nucleotide vectors (see subsection 5.5.1).

With their architecture, *DpathL* and *DpathR* can process any subtree topology with up to 8 and 6 taxa, respectively. So, the most complicated subtree

topologies *DpathL* and *DpathR* can handle are the ones shown in the upper and lower halves of figure 5.19, respectively. Finally, a processing element in the linear array (see figure 5.15) can support a tree topology with at most 12 taxa.

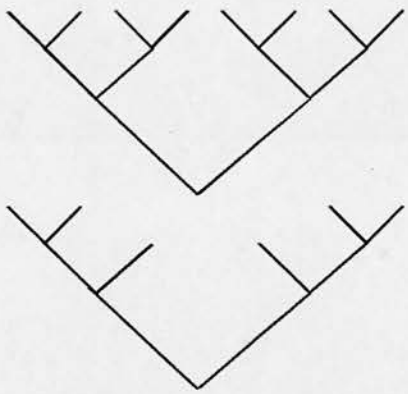


Figure 5.19: *Most complicated subtree topologies supported by DpathL (upper one) and DpathR (lower one)*

5.6 Implementation Results

The MP method was implemented on the Alpha Data nodes of the Maxwell machine with the array architecture shown in figure 5.16, where the number of the processing elements was 20. The proposed design was captured in Verilog hardware description language which was then synthesized, placed, and routed by Xilinx ISE9.2 tool. FPGA bitstreams were also generated by the same tool while ModelSim was employed to test the core with a number of testbenches. The clock frequency of the FPGAs was set to 70 MHz. Note that only one FPGA bitstream is used to configure the FPGAs regardless of the number of taxa under consideration. Another important point is that the time it takes to load the bitstream to FPGA is in milliseconds (ms) whereas the computation time on FPGAs is in seconds (s). Hence, FPGA configuration time does not affect the overall computation time that much. A high level

application process was built using the FHPCA Parallel Toolkit (PTK) and run on the host CPUs. Its main duty was to write the cost matrix data, tree topology data and sequence alignment data to the input FIFO and then read the scores of the tree topologies from the output FIFO of the FPGAs with Direct Memory Access (DMA) transfers (see figure 5.15). On the other hand, a small C program was written to construct the tree topology data for various numbers of taxa.

Table 5.3 presents the performance figures of the proposed hardware implementation for the MP method for up to 12 nucleotide sequences on one node of the Maxwell machine (each node has a Xilinx Virtex-4 XC4VFX100 FPGA [76]). It was assumed that the cost of changes from a purine (A or G) to pyrimidine (C or T) is two times the cost of changes from a purine to a purine and pyrimidine to a pyrimidine. The length of the nucleotide sequences was 898 where a two times higher weight was applied to the changes occurring at the first position of the codons compared to the second and third positions.

Table 5.3: *Timing performance figures of the hardware implementation for the MP method on one node of the Maxwell machine*

No. of Taxa	No. of Trees	No. of Iterations	Min. Score	Average Time (s)
4	3	1	778	1.420
5	15	1	927	1.421
6	105	6	1124	1.423
7	945	48	1361	1.425
8	10395	520	1396	1.430
9	135135	6757	1488	1.480
10	2027025	101352	1587	2.255
11	34459425	1722972	1898	3.446
12	654729075	32736454	2230	5.893

Each row in table 5.3 is associated with some number of taxa given in the first column where the second column presents the number of unrooted tree topologies to be searched for that specific number of taxa. Furthermore, the third column gives the number of iterations required by the hardware core considering the number of available PEs to complete the processing of all trees (see subsection 5.5.1) while the fourth column shows the score of the most parsimonious tree found during the exhaustive tree search. Finally, the fifth column gives the average time in seconds taken by the hardware core to complete its operation for each number of taxa.

For comparative purposes, table 5.4 below shows the timing figures of the PAUP software execution configured to operate in the same way as the hardware implementation, with the same nucleotide sequences. The software version was run on a 2.2 GHZ Intel Centrino Duo machine with 2 GB of RAM running Windows XP operating system. Note that results obtained by the hardware implementation were identical to those of PAUP.

Table 5.4: *Timing performance figures of the PAUP software for the MP method*

No. of Taxa	No. of Trees	Average Time (s)
4	3	0.02
5	15	0.02
6	105	0.02
7	945	0.16
8	10395	0.67
9	135135	7.84
10	2027025	241
11	34459425	5852
12	654729075	127325

Figure 5.20 plots the timing performance results of the FPGA and software implementations shown in tables 5.3 and 5.4 with a logarithmic scale. As it can be seen, for low numbers of taxa, PAUP operates faster than the FPGA hardware implementation. However, the latter becomes much faster as the number of taxa increases. Note that both plots in figure 5.20 show an exponentially increasing curve which is obviously much sharper for the software solution for the MP method.

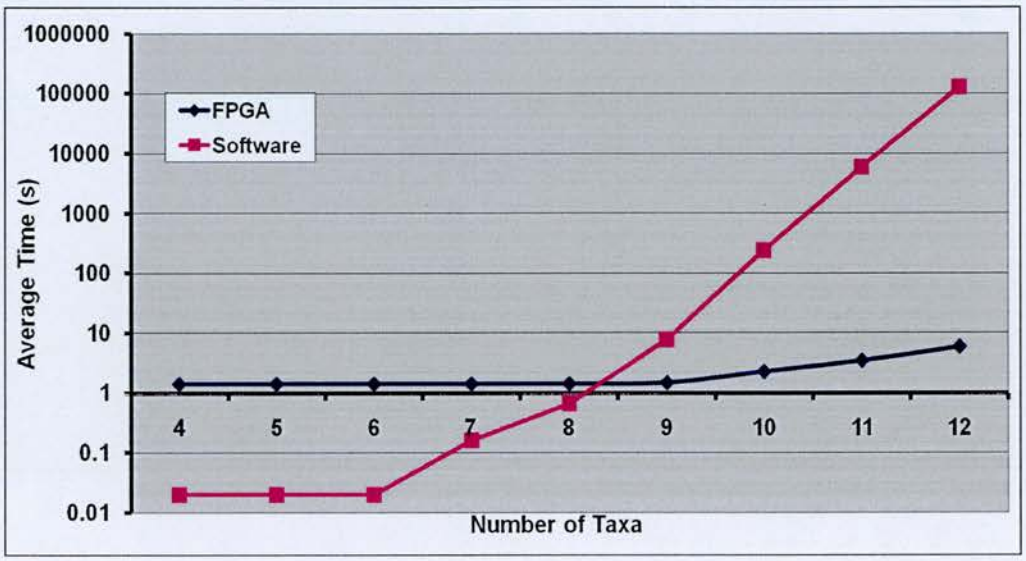


Figure 5.20: *Timing performance plot of the FPGA and software solutions for the MP method (scale is logarithmic)*

Table 5.5 below provides the speed-up values of the hardware implementation on 1 node over the software implementation (i.e. PAUP) for various numbers of taxa. It is obvious that hardware core outperforms PAUP hugely when the number of taxa is over 8 with the speed-up values reaching 21606x for the 12-taxa case.

Table 5.5: *Software (PAUP) versus 1-node hardware implementation speed-up values*

No. of Taxa	FPGA Speed-Up
9	5
10	110.1
11	1698.2
12	21606.1

Tables 5.6, 5.7 and 5.8 below show the timing figures of the FPGA implementation for the MP method on 2, 4, 8 nodes of the Maxwell machine, respectively, where the tree topologies for a given number of taxa are shared and distributed among the specified number of nodes by the master node among the CPUs of the nodes using MPI [77]. Furthermore, the third columns in these tables present the maximum number of iterations required by the hardware core on a node, while the fourth columns give the total time taken to complete the whole process including the collection of the tree scores from each node by the master node. It can be noticed that average times taken are decreasing as the number of utilized nodes increases although the overhead of distributing tree topology data and collecting results may surpass the gain from the parallel operation of the nodes in the case of a low number of taxa. The effects of this communication overhead on the efficiency and scalability of the proposed design over multiple nodes is graphically represented in figure 5.21 with the timing values for each given number of taxa as presented in tables 5.6, 5.7 and 5.8.

Table 5.6: *Timing performance figures of the hardware implementation of the MP method on two nodes of the Maxwell machine*

No. of Taxa	Total No. of Iterations	No. of Iterations per Node	Average Time (s)
7	48	24	1.423
8	520	260	1.428
9	6757	3379	1.460
10	101352	50676	1.930
11	1722972	861486	2.926
12	32736454	16368227	4.911

Table 5.7: *Timing performance figures of the hardware implementation of the MP method on four nodes of the Maxwell machine*

No. of Taxa	Total No. of Iterations	No. of Iterations per Node	Average Time (s)
7	48	12	1.415
8	520	130	1.423
9	6757	1690	1.443
10	101352	25338	1.780
11	1722972	430743	2.584
12	32736454	8184114	4.256

Table 5.8: *Timing performance figures of the hardware implementation of the MP method on eight nodes of the Maxwell machine*

No. of Taxa	Total No. of Iterations	No. of Iterations per Node	Average Time (s)
7	48	6	1.418
8	520	65	1.425
9	6757	845	1.433
10	101352	12669	1.690
11	1722972	215372	2.412

12	32736454	4092057	3.928

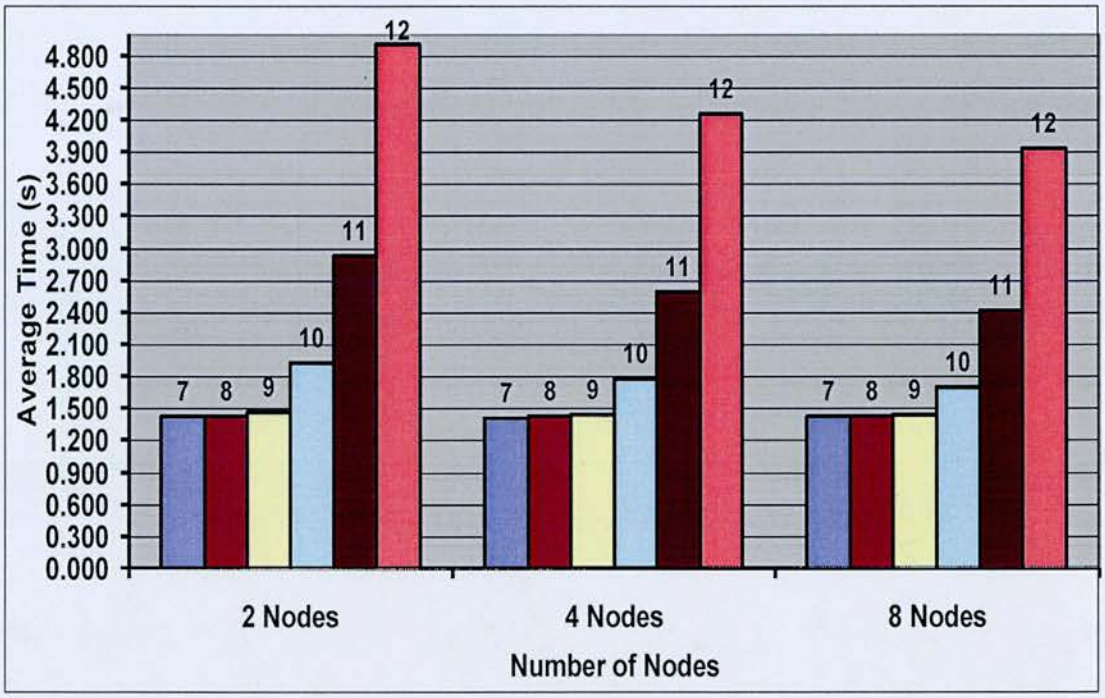


Figure 5.21: *Scaling performance of the hardware core on multiple nodes of the Maxwell for given numbers of taxa*

Finally, table 5.9 below provides the speed-up values of the hardware implementation on 2 nodes, 4 nodes, and 8 nodes over the software implementation for various numbers of taxa up to 12. Note that the poor scaling in performance when using multiple nodes (as can be seen in figure 5.21) is due to the fixed communication latency between the host computer and corresponding FPGA. This is a generic problem which will always occur when the ratio of the computation time on FPGA hardware to the communication time between host and FPGA becomes low.

Table 5.9: *Software (PAUP) versus 2-nodes/4-nodes/8-nodes hardware implementations speed-up values*

No. of Taxa	FPGA Speed-up with 2 Nodes	FPGA Speed-up with 4 Nodes	FPGA Speed-up with 8 Nodes
9	5.4	5.4	5.5
10	124.9	135.4	142.6
11	2000	2264.7	2426.2
12	25929.3	29916.6	32414.7

5.7 Conclusions

In this chapter, the detailed FPGA implementation of the Maximum Parsimony method for molecular phylogenetic analysis on the nodes of the Maxwell FPGA supercomputer has been presented. This is the first FPGA implementation of this method for nucleotide sequence data reported in the literature to our knowledge. The hardware architecture is a linear systolic array composed of 20 processing elements each of which performing Sankoff's algorithm for a different tree topology in parallel. This array computes the scores of all tree topologies for a given number of taxa in several iterations.

The currently supported maximum number of taxa is 12 but this number can be easily improved by cascading more *DpathUnit*s in *DpathL* and *DpathR* blocks. Furthermore, the resulting implementation outperforms an equivalent desktop-based software implementation (i.e. PAUP) by very high orders-of magnitude. The speed-up values achieved by the hardware implementation on a single node of Maxwell can reach up to 21606x for the 12-taxa case while implementations on several nodes can yield even higher values. The reasons behind this very high speed-up are essentially twofold: the first is the coarse-grain parallelism among processing elements, since

each *PE* processes a different tree topology in parallel with other PEs, and second is the fine-grain parallelism achieved in each processing element, as conditional vectors of several nodes on a specific level of the tree topology under consideration are computed concurrently (see figure 5.14).

As a short-term future goal for this case study, we plan to extend and improve the presented architecture to be able to support computations for unlimited number of taxa by incorporating a reconfigurable router into the design. On the other hand, we plan to design a web-based interface for the proposed design through which bioinformaticians can submit their sequences online for high performance phylogenetic tree construction on Maxwell FPGA based supercomputer as a long-term future goal for the case study.

Chapter 6

Parallel Processor Design for Molecular Dynamics Simulations on a FPGA Parallel Computer

6.1 Introduction

Computer simulations are carried out to understand the properties of assemblies of molecules in terms of their structure and the microscopic interactions between them [4]. They act as a bridge between microscopic length and time scales and the macroscopic world of the laboratory, serving as a complement to conventional experiments. Carrying out simulations on computers that are either difficult or impossible in the labarotory enables us to learn something new, something that can not be found out in other ways.

There are two main families of simulation techniques; Molecular Dynamics (MD) and Monte Carlo (MC)-based simulations. There are also several hybrid techniques which combine features from both. MD is a deterministic simulation technique whereas simulation results from MC simulations are stochastic. Furthermore, MD can provide the dynamic properties of the simulated system as well as the static properties, as opposed to MC.

In MD, the time evolution of a set of interacting atoms modelled with classical mechanics is followed by integrating their Newtonian equations of motion. MD simulations of biomolecules provide a molecular picture of the structure and behaviour of biological systems such as enzymes, proteins, DNA strands and membranes. This allows scientists to advance their understanding of biologically important molecules. The MD method has applications in the fields of protein engineering [7], drug design [8] [78] and refinements of structures based on X-ray [9] and NMR experiments [10].

However, biological systems of interest have sizes ranging from a few tens of thousands to millions of atoms. Performing MD simulation of a biological process, such as protein folding, for a reasonable physical time requires enormous amounts of computational effort and may take years to complete on conventional computers. Therefore, it is mandatory to utilize faster computing platforms.

Special-purpose computers for the acceleration of MD simulation gathered growing interest lately [89]. Field Programmable Gate Arrays (FPGAs) in particular have recently been proposed as a viable alternative implementation platform for MD simulation due to their flexible computing and memory architecture which gives them ASIC-like performance with the added programmability feature. Therefore, we chose FPGAs over ASICs as they offer reprogrammability, shorter development times and lower Non-Recurring Engineering (NRE) costs.

There are several MD simulation software tools. However, software for MD simulation can spend very high percentange of the total computation time in calculating the non-bonded interactions among particles because the computational complexity of the evaluation of non-bonded potentials or forces is quadratic. Therefore, we can accelerate MD simulation by porting the calculation of the non-bonded interactions from software to FPGAs since non-bonded interactions lend themselves to be easily calculated in parallel. On the other hand, the remaining MD calculation, which is complex but only consumes a very limited percentage of the total computation time, can be left to software running on a host computer. Our ultimate goal is to design and implement a MD simulation sytem that will allow scientists to simulate a biomolecular system within a reasonable time frame and obtain useful information of a biological system.

The design and implementation of an FPGA core that parallelises all the necessary operations to compute the non-bonded interactions in the Large-scale

Atomic/Molecular Massively Parallel Simulation (LAMMPS) software tool is explained in this chapter. The proposed MD processor core is comprised of 4 identical pipelines working independently in parallel to evaluate the non-bonded potentials, forces and virials acting on a particle from all of the other particles in the simulated molecular system. A real hardware implementation of the designed core was achieved on the nodes of an FPGA-based supercomputer, called Maxwell, which consists of 64 Virtex-4 FPGA chips. Implementing the proposed FPGA core on multiple nodes of Maxwell allowed us to produce a special-purpose parallel machine for the hardware acceleration of MD simulations. This machine is highly scalable, yielding higher computational power with the additional Maxwell nodes.

The remainder of this chapter will first present essential background information on MD simulation and then discuss related prior works in the literature. Subsequently, LAMMPS MD simulation software will be introduced and then the general system architecture will be explained. Furthermore, the design and implementation of the proposed FPGA core for computing the non-bonded interactions in a MD simulation will be elaborated. Following this, implementation results are presented and then evaluated comparatively with equivalent pure software implementations. Finally, conclusions are laid out with possible extensions of this implementation.

6.2 Molecular Dynamics Simulation

Molecular Dynamics is commonly used for the simulation of the structural, thermodynamic and transport properties of large biological systems on a diverse range of timescales. In MD simulations, atoms in the system are treated as classical particles and are subject to covalent bond, Van der Waals and Coulomb forces from other particles. During a time-step of the MD simulation, forces are computed and accumulated on each atom due to its interac-

tion with other atoms, and positions and velocities of atoms are updated by integrating the Newtonian equations of motion.

6.2.1 Molecular Interactions

In MD simulations of biological systems, the potential for a particle i, Φ_i, is modelled as follows:

$$\Phi_i = \Phi_i^B + \sum_{j \neq i} \epsilon_{ab} \left\{ \left(\frac{\sigma_{ab}}{|\boldsymbol{r}_{ji}|} \right)^{12} - \left(\frac{\sigma_{ab}}{|\boldsymbol{r}_{ji}|} \right)^{6} \right\} + q_i \sum_{j \neq i} \frac{q_j}{|\boldsymbol{r}_{ji}|} \quad (6.1)$$

where $\mathbf{r}_{ji}$ is a vector from the particle j to i and q_i is the charge of the particle i. Also, σ_{ab} is a length parameter and ε_{ab} is an energy parameter where a and b denote the atom types of particles. The first term Φ_i^B is the bonded potential due to interactions within the topology of the molecules:

$$\Phi_i^B = \sum_{bonds} K_b (r - r_0)^2 + \sum_{angles} K_\theta (\theta - \theta_0)^2 + \sum_{dihedrals} K_{\phi_p} [1 + d_p \cos(n_p \phi)] \quad (6.2)$$

Bonded potential is written here as sums over simple harmonic 2-body (bond), 3-body (angle) and 4-body (dihedral) interactions although other potential models could also be used. On the other hand, the last 2 terms in (6.1) are the non-bonded potential due to interactions between all pairs of atoms in the system. Note that the forces exerted on the particle i, $\mathbf{f}_i$, are obtained by taking the gradient of (6.1) with respect to the position of the particle.

The second term in (6.1), which describes van der Walls interaction, is the Lennard-Jones (LJ) potential characterized by parameters σ_{ab} and ε_{ab}. If we

take the gradient of this potential, an LJ force $\mathbf{f}_i^{LJ}$ can be expressed as:

$$\boldsymbol{f}_i^{LJ} = \sum_{j \neq i} \frac{\epsilon_{ab}}{\sigma_{ab}^2} \left\{ 12 \left(\frac{\sigma_{ab}}{|\boldsymbol{r}_{ji}|} \right)^{14} - 6 \left(\frac{\sigma_{ab}}{|\boldsymbol{r}_{ji}|} \right)^{8} \right\} \boldsymbol{r}_{ji} \quad (6.3)$$

On the other hand, the third term on the right hand side of (6.1) is the Coulombic (C) potential, and the corresponding Coulombic force $\mathbf{f}_i^C$ is expressed as:

$$\boldsymbol{f}_i^C = q_i \sum_{j \neq i} \left(\frac{q_j}{|\boldsymbol{r}_{ji}|^3} \right) \boldsymbol{r}_{ji} \quad (6.4)$$

The computational complexity of evaluating Φ_i^B is $O(1)$ since only few particles are covalently bonded to the i^{th} particle. However, the computation time to evaluate nonbonded potential or force functions is $O(N)$ for each particle where N is the number of particles in the simulated system. Hence, the computational complexity to evaluate the functions for all particles in the system is $O(N^2)$. Accelerating these evaluations is therefore the prime target for the design of the proposed MD core. Note that the proposed MD processor core will be able to deal with an arbitrary potential or force function although only the LJ and Coulombic interactions are mentioned in this section.

6.2.2 Cutoff Convention

The simplest method for reducing the computation time is the cutoff convention. Contributions from particles outside a certain cutoff radius r_c are ignored in this method and hence, the time complexity is reduced to $O(N)$. For instance, since LJ force and potential decrease rapidly with increasing distance (refer to (6.3)), the sum over j can be truncated within the determined

cutoff distance so that only a few neighbours of atom i contribute rather than all N. This does not affect the results in most cases provided that the particles are well separated with respect to an appropriate value of r_c.

In contrast, the Coulombic interaction is long-range which means it decreases slowly with an increase of distance (refer to (6.4)). Hence, evaluating Coulombic force as a truncated sum over neighbours rather than as a full sum introduces large inaccuracies [79]. On the other hand, applying the latter method is problematic in periodic systems (briefly mentioned in subsection 6.2.4 below). Consequently, other methods are often used for the evaluation of Coulombic force and potential. One of these methods, namely the Ewald method, is discussed in subsection 6.2.5 and the one used by the LAMMPS software is explained in subsection 6.4.1.

6.2.3 Virials

Virials represent the effect of mutual interaction of particles on the pressure in the system. The virial $\mathbf{v}_i$ on the particle i can be calculated with the following equation where T denotes the transpose of the vector:

$$\boldsymbol{v}_i = \sum_{j \neq i} \boldsymbol{f}_{ji} \boldsymbol{r}_{ji}^{T} \quad (6.5)$$

Note that the time complexity of this operation for all particles is $O(N^2)$ since it is $O(N)$ for each particle. The proposed MD processor incorporates the computation of all components of each virial.

6.2.4 Periodic Boundary Conditions

MD simulations are generally performed under periodic boundary conditions where the original simulation cell is deemed to be surrounded by its 26

image cells [4]. Then, minimum image convention should be adopted in the calculations of pairwise interactions. This means that a force exerted on the particle i from the particle j is only to be calculated for the real particle j or nearest image of it to the particle i.

6.2.5 Ewald Method

In the cases where periodic boundary conditions apply and hence, electrically charged particles exist periodically, Coulombic forces can be calculated precisely by the Ewald method [80]. Force $\mathbf{f}_i^C$ is split into the sum of two rapidly converging series in the Ewald method as follows:

$$\boldsymbol{f}_i^C = \boldsymbol{f}_i^r + \boldsymbol{f}_i^m \quad (6.6)$$

where $\mathbf{f}_i^r$ is the real space sum and $\mathbf{f}_i^m$ is the reciprocal space sum. The real space sum is given in (6.7) where the positive parameter α is taken to be an appropriate value so that the $\mathbf{f}_i^r$ converges rapidly.

$$\boldsymbol{f}_i^r = \frac{q_i}{4\pi\epsilon_0}\sum_j q_j \left\{\frac{2\alpha}{\sqrt{\pi}} exp\left(-\alpha^2|\boldsymbol{r}_{ji}|\right) + \frac{erfc\left(\alpha|\boldsymbol{r}_{ji}|\right)}{|\boldsymbol{r}_{ji}|}\right\}\frac{1}{|\boldsymbol{r}_{ji}|^2}\boldsymbol{r}_{ji} \quad (6.7)$$

In (6.7), *erfc* is the complementary error function which is defined as:

$$erfc(x) = 1 - \frac{2}{\sqrt{\pi}}\int_0^x exp\,(-t^2)\,dt \quad (6.8)$$

The proposed MD processor can evaluate $\mathbf{f}_i^r$ according to (6.7) whose computation time is $O(N^2)$ for all particles since it is $O(N)$ for each particle. On the other hand, the computation of $\mathbf{f}_i^m$ is left to the software running on a host

processor in the proposed implementation.

6.2.6 Time Integration

There are various kinds of integrators to integrate Newtonian equations of motion, such as Verlet algorithm [81], Beeman algorithm [82], and multiple time-step algorithms [83]. One of the simplest and most popular algorithms for the time integration of the positions and velocities of particles is the Verlet algorithm which is expressed as the following two equations:

$$\boldsymbol{v}\left(t + {}^{\delta t}/_{2}\right) = \boldsymbol{v}\left(t - {}^{\delta t}/_{2}\right) + \delta t \boldsymbol{a}(t) \quad (6.9)$$

$$\boldsymbol{r}(t + \delta t) = \boldsymbol{r}(t) + \delta t \boldsymbol{v}\left(t + {}^{\delta t}/_{2}\right) \quad (6.10)$$

where **r** (t), **v** (t) and **a**(t) are the position, velocity and acceleration vectors of a particle at time t, respectively and δt denotes the chosen size of each time-step. Note that the acceleration of a particle at a time-step is computed by the Newton's second law of motion:

$$\boldsymbol{a}_i = \frac{\boldsymbol{f}_i}{m_i} \quad (6.11)$$

6.3 Prior Work

Improving the performance of MD simulation software with fast computation algorithms or parallel algorithms such as atom decomposition, force decomposition and spatial decomposition was the primary focus of prior research on accelerating MD simulations. There exist a number of sophisticated MD software packages including GROMACS [84], [85], NAMD [86], [87] and LAMMPS [88]. In next section, the LAMMPS tool (a highly parallel MD si-

mulator) will be introduced. However, since these software packages are limited by the the performance of a general purpose processor, some research has turned to special-purpose and application-specific hardware acceleration of the MD simulation. The main target of this new research topic is to speed-up the most computationally intensive portion of the MD simulation computation, namely the non-bonded interactions.

MD-GRAPE [89], [90] is one of the most prominent hardware acceleration systems for MD simulations. It uses a fourth order polynomial with 1024 piece to approximate the calculation of the force or potential where the coefficients determine which force or potential is calculated. MD-GRAPE which has a peak speed of 4.2 Gflops only accelerates the computation of the force and potential while leaving the rest of the MD simulation to a host processor. MD engine [91] was also a special-purpose computer for MD simulation which had similar system architecture to that of the MD-GRAPE system, where the host computer communicates with the special-purpose parallel machine that computes the non-bonded interactions.The MD engine system consists of 76 individual processors named MODEL each of which calculates both the Lennard-Jones and Coulombic interactions. The system can perform the simulation 50 times faster than an equivalent software implementation running on a Sun Ultra-2 200 MHz machine.

All of the aforementioned special-purpose hardware platforms for MD simulation were implemented using ASIC technology. However, hardware development in this way can take up several years before the application is fully implemented. On the other hand, recent advances have made FPGAs a viable platform for accelerating MD simulations with substantial performance gains. Therefore, recent academic research has attempted to implement special-purpose computers for MD simulation using FPGAs.

Prior reserach on FPGA-based MD simulations have concentrated on accelerating different parts of the MD simulation. One of them mapped the posi-

tion and velocity update to FPGA [92] while most of them computed LJ and Coulombic interactions of each time-step on FPGA [93], [94], [95], [96], [110]. On the other hand, only few ones moved all tasks in MD simulation onto FPGA [97], [98].

6.4 LAMMPS MD Simulation Software

LAMMPS is a classical molecular dynamics code written in C++, which stands for Large-scale Atomic/Molecular Massively Parallel Simulator [99]. It was developed at Sandia National Laboratories under the US department of Energy as a freely-available, open-source code, distributed under the terms of the GNU public license.

LAMMPS runs on single-processor machine although it was designed to run most efficiently on parallel computers supporting the MPI message-passing library, for instance on distributed- or shared-memory parallel machines and Beowulf-style clusters. LAMMPS can model atomic, polymeric, biological, metallic, granular and coarse-grained systems with only a few particles up to millions or billions using a variety of force fields and boundary conditions. However, it was designed to be easily modified or extended with new capabilities, such as new force fields, atom types or boundary condtions.

LAMMPS partitions the simulation domain into small 3D subdomains with spatial decomposition techniques on parallel machines. Each subdomain is assigned to a processor, and processors communicate and store ghost atom information for atoms that border their subdomain. By subdividing the physical volume among processors, most computations become local and communication is minimized so that optimal N/P scaling of the overall calculation can be achieved on P processors. Hence, the spatial-decomposition method is clearly the best algorithmic choice in comparison with atom decomposition and force decomposition methods both of which do not scale well to

large numbers of processors. Note that systems with uniform particle density are most efficiently simulated by LAMMPS on parallel machines.

In the simplest sense, LAMMPS integrates Newton's equation of motion for particles interacting via short- or long-range forces. It utilizes neighbour lists to keep track of the nearby particles for each particle so that the short-range, non-bonded potentials and forces for all particles are computed efficiently using cutoff convention (see subsection 6.2.2) with time complexity of $O(N)$. As atoms move, these lists are reformed at every few time-steps using both owned and ghost atoms.

There are several ways to enable the quick calculation of the Coulombic interactions by avoiding the all-pairs $O(N^2)$ computation. Approximate techniques include multipole methods [100], [101] scaling as N, Ewald summation (see subsection 6.2.5) scaling as $N^{3/2}$ and Particle-Particle Particle-Mesh (PPPM) method [102] scaling as N log $(N)^{1/2}$. PPPM which is a variant of Particle-Mesh Ewald (PME) method [103] is the method used by LAMMPS for the Coulombic computations due to its higher computational efficiency relative to other methods, particularly in parallel setting, as described in subsection 6.4.1.

Papers [104], [105] elaborate on the technical details of the algorithms used in LAMMPS.

6.4.1 PPPM Method

A detailed comparison of Ewald, multipole and PPPM methods shows that in addition to being less complex to implement, PPPM is the fastest for systems of any reasonable size [106]. The basic idea of PPPM is to replace the point charge Coulombic term in (6.1) with an equivalent expression for extended charges centered on the original atomic positions. Hence, Coulombic potential is now expressed as follows:

$$\Phi_i^C = \frac{q_i}{4\pi\epsilon_0}\sum_{j\neq i}\frac{q_j}{|r_{ji}|}erfc\left(\frac{G|r_{ji}|}{\sqrt{2}}\right) + \iint\frac{\hat{p}_i(r)\hat{p}_i(r')}{|r-r'|}\,drdr' - \frac{G}{\sqrt{2\pi}}q_i^2 \quad (6.12)$$

where $\hat{p}_i(r)$ is the Gaussian density that represents an extended charge and is given as follows:

$$\hat{p}_i(r) = q_i\left(\frac{G^2}{\pi}\right)^{3/2} exp(-G^2(r-r_i)^2) \quad (6.13)$$

The first term in (6.12) is the usual Coulombic potential multiplied by a complementary error function which forces it to go to nearly zero at a user-specified cutoff distance r_c, where G is determined by the accuracy criterion. Thus, this term is the short-range portion of the Coulombic interaction and is computed in LAMMPS at the same time as van der Waals interactions as a sum over nearby particles utilising neighbour lists. On the other hand, the second term in (6.12) is the Coulombic potential due to the interaction of the extended charges whereas the last term is a constant.

6.5 System Architecture

The proposed special-purpose parallel machine for MD simulations is a reconfigurable hardware accelerator plugged into a number of host CPUs. Figure 6.1 illustrates the basic process flow in the proposed machine for each time-step. LAMMPS MD simulation software (see section 6.4) running on a general purpose computer first initialises the simulation environment, calculates the less time-consuming bonded interactions and builds a neighbour list for each particle i in the simulated system, which includes all nearby j particles within a certain radius of the particle i. Then, for each neighbour list, software on host CPUs first broadcast the coordinates and electric charge of

the particle i to the reconfigurable hardware and subsequently the coordinates and electric charge of each j particle in the neighbour list as well as the interaction parameters and the cutoff distances for the specific pairs of i and j particles are sent to the reconfigurable hardware one by one as shown in figure 6.1. Following this, the proposed parallel machine computes all non-bonded forces, virials and potentials acting on each particle i due to the j particles in its neigbour list and then, send these pairwise values back to the host CPU. Software running on the host use the force values to calculate the acceleration of each particle in the simulated system by (6.11) and then integrates Newtonian equations of motion (see subsection 6.2.6) by an integration technique to update the velocity and position values of all particles at the current time-step. Software also adds-up pairwise potential and virial values to compute the total per-atom potentials and virials, respectively. The total potential energy and pressure in the simulated system at the current time-step are also calculated by accumulating these potential and virial values, respectively. Note that a new C++ class was written for the LAMMPS software using the FHPCA Parallel Toolkit (PTK) to be able to co-operate with the reconfigurable hardware for MD simulations in the way explained above. Another important point is that all transfers between host CPU and reconfigurable hardware are done with Direct Memory Access (DMA) method.

Figure 6.2 shows the system connection diagram of the proposed special-purpose parallel machine for MD simulations. Two processes of LAMMPS software run on each Intel Xeon CPU while an instance of the proposed MD processor core resides in each user FPGA. Actually, a software process running on a host CPU and a hardware core in a user FPGA form a Maxwell node as described in subsection 3.7.3. The number of utilized Maxwell nodes where LAMMPS processes communicate with each other by Message Passing Interface (MPI) [77] can be easily configured as desired.

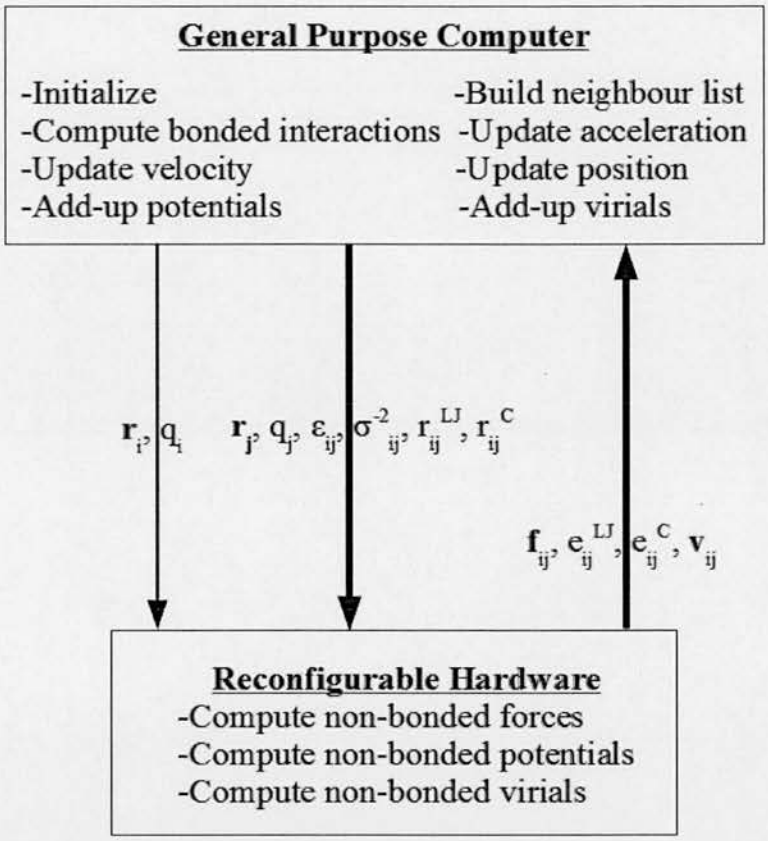


Figure 6.1: *Basic structure of the proposed special-purpose parallel machine for Molecular Dynamics simulations*

Each Xeon CPU on PCI-X bus connects to two user FPGAs through bridge/control FPGAs mediating communication between the 32-bit wide PCI-X bus operating at 133 MHz and the 32-bit wide local buses of the user FPGAs operating at 80 MHz, as shown in figure 6.2. Furthermore, user FPGAs in the proposed MD machine are of Xilinx Virtex-4 FX-100 type whereas smaller FPGAs bridging PCI-X and local buses are of Xilinx Virtex-4 LX25 type. On the other hand, four 256 MB DDR2 SDRAMs are connected to each user FPGA. The physical width and depth of the SDRAMs are 32 bits and 64M words, respectively while the logical width of the SDRAMs is 128 bits. Note that the MD processor core in a user FPGA runs at 150 MHz although the logic interfacing the user FPGA to the local bus it is connected to runs at 80 MHz.

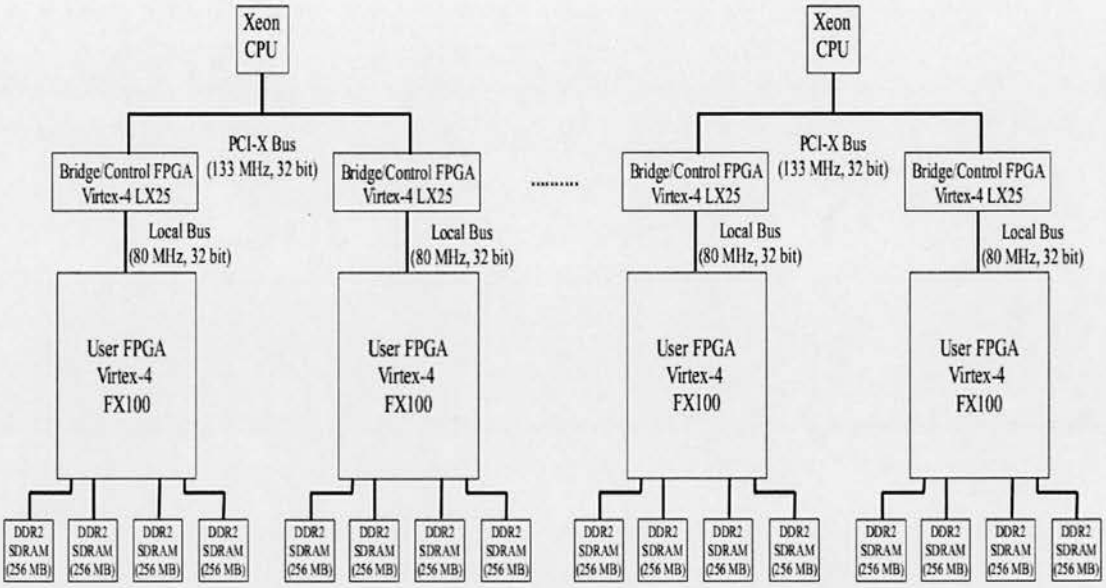


Figure 6.2: *Sytem connection diagram of the proposed special-purpose parallel machine for Molecular Dynamics simulations*

Figure 6.3 shows the block diagram of the proposed MD processor core. As it can be seen, the proposed MD core incorporates 4 identical MD pipelines which are working independently in parallel to evaluate the non-bonded potentials, forces and virials acting on a particle from each of the particles in the neighbour list of that particle. Each MD processor is associated with one of the SDRAM banks connected to the user FPGA. Furthermore, the LAMMPS process running on a host CPU transfers the simulation-related data mentioned above to the allocated first region of each SDRAM bank to be processed by the relevant MD processor. When these incoming transfers complete, the software process signals each MD processor to start its operation of reading data from its associated SDRAM bank through the use of an input buffer and then writing the evaluated potential, force and virial values back to the allocated second region of the associated SDRAM bank through the use of an output buffer, all under the control of a Finite State Machine (FSM) as shown in figure 6.3. When the MD processor is done with its operation, it signals the software process to transfer its computed values back from the relevant SDRAM bank for further processing as explained above. Moreover,

two identical function coefficient memories shown in the figure 6.3 store the coefficients of the interpolation used to evaluate a number of functions as will be detailed later in the next section in conjunction with the inner architecture and operation flow of the proposed designed MD processor.

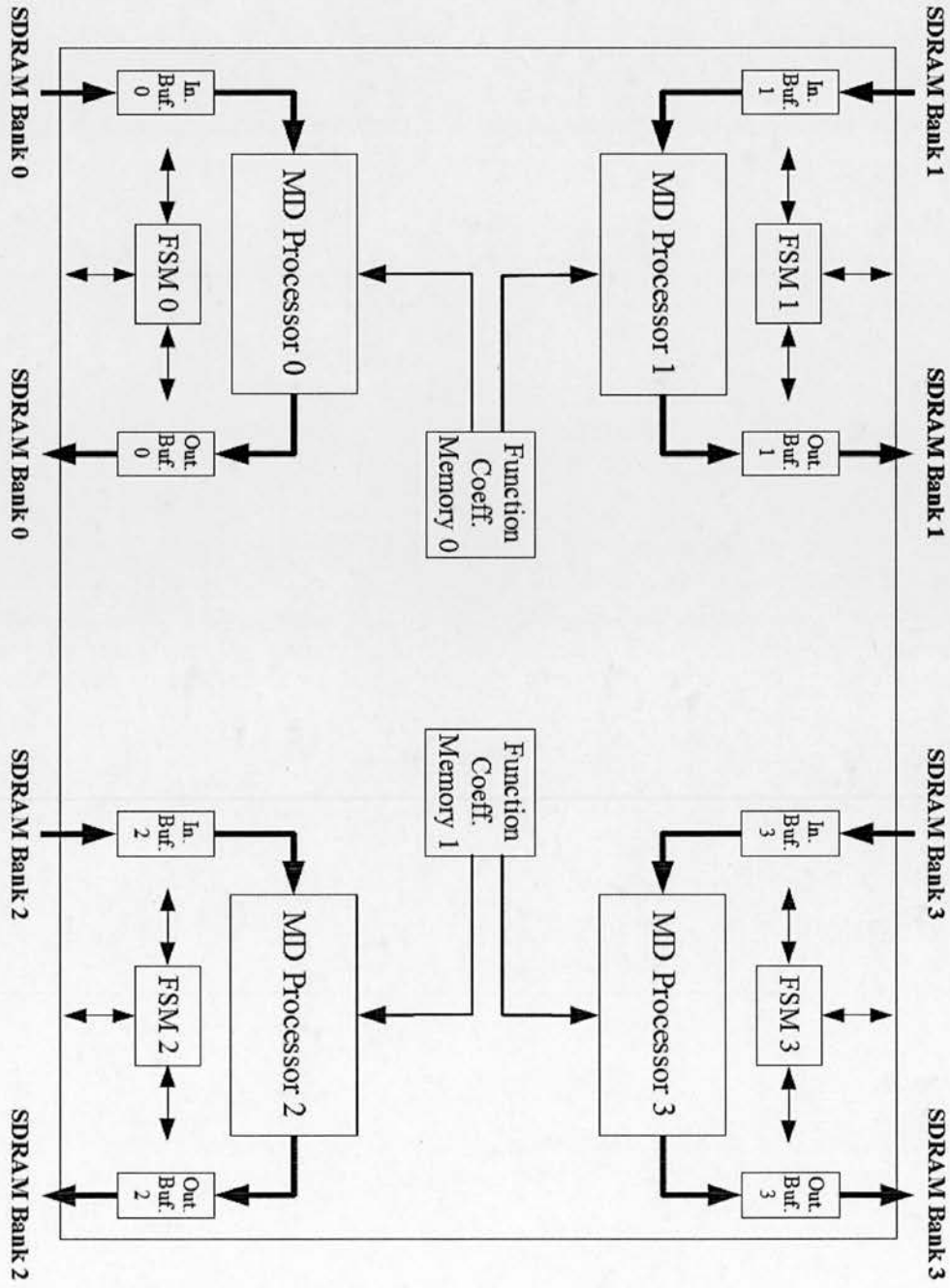


Figure 6.3: *Block diagram of the proposed Molecular Dynamics processor core*

6.6 Design of Molecular Dynamics Processor

In the proposed design, internal format of the numbers used is the IEEE standard single precision (i.e. 32-bit wide) floating-point as shown in figure 6.4 (a). All data are handled in this format. Hence, single precision floating-point arithmetic units are utilized throughout the proposed MD processor. Several pipelined floating-point multipliers and adders/subtractors obtained from [107] are incorporated in the proposed design whose operation latencies are 4 and 6 clock cycles, respectively, as explained in [108]. These arithmetic units do not support denormalized numbers and NaN ("Not a Number") to minimize the required hardware resources and realize high operation speed by simplifying the circuitry.

SDRAM banks in the proposed MD machine were partitioned into two regions. The first region of a SDRAM bank was allocated to data transfers from the host CPU while the second region was allocated to data transfers from the designated MD processor on a user FPGA, as mentioned in section 6.5. Figure 6.4 (b) shows the layout of a memory portion in the first region of a SDRAM bank storing the coordinates $\mathbf{r}_i = (r_x, r_y, r_z)$ and electric charge q_i of a particle i whereas figure 6.4 (c) shows the layout of another memory portion in the first region of a SDRAM bank storing the coordinates $\mathbf{r}_j = (r_x, r_y, r_z)$ and electric charge q_j of a j particle, as well as the interaction parameters ε_{ij}, $\sigma^{-2}{}_{ij}$ and the cutoff distances for both Lennard-Jones $r^{LJ}{}_{ij}$ and Coulombic $r^{C}{}_{ij}$ interactions, all pertaining to the particular pair of i and j particles. Since the logical width of the memory banks is 128 bits, the layouts in figure 6.4 (b) and (c) occupy the space of 1 and 2 logical words, respectively. Furthermore, the layout of a memory portion in the second region of a SDRAM bank storing the force $\mathbf{f}_{ij} = (f_x, f_y, f_z)$, Lennard-Jones potential $e^{LJ}{}_{ij}$, Coulombic potential $e^{C}{}_{ij}$ and virial $\mathbf{v}_{ij} = (v_x^2, v_y^2, v_z^2, v_{xy}, v_{xz}, v_{yz})$ values computed for the specific pair of i and j particles is displayed in figure 6.4 (d) which takes up the space of 3 logical words in a memory bank.

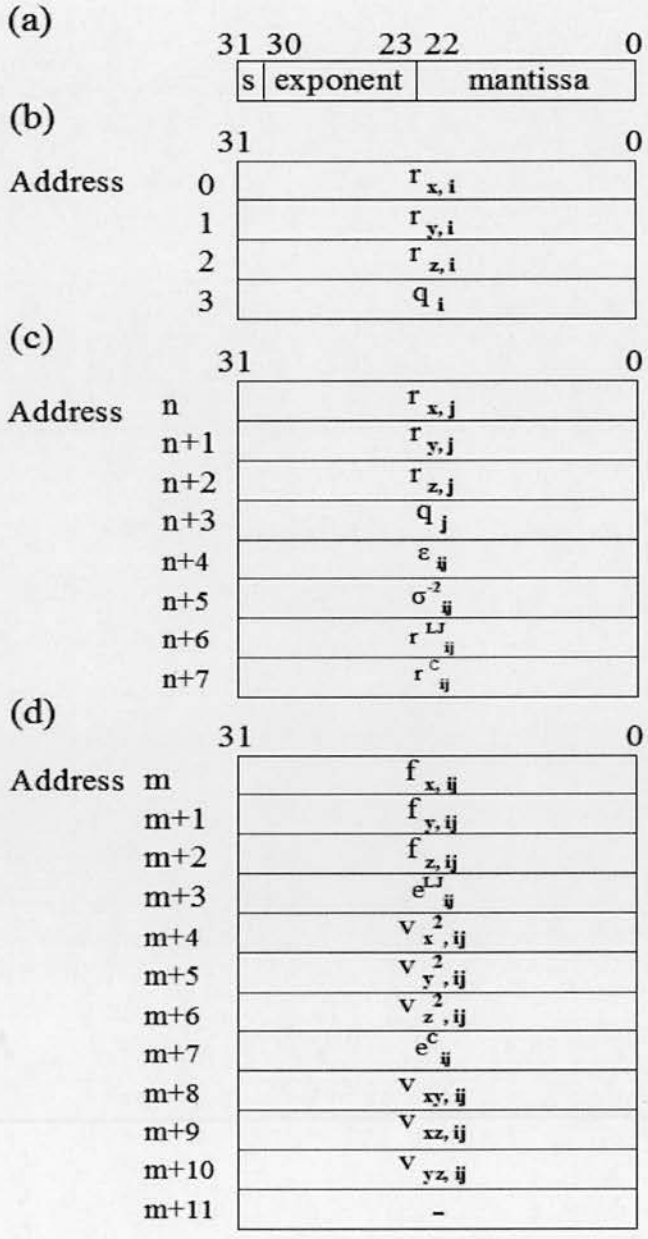


Figure 6.4: *(a) Internal format of the numbers used in the proposed design (b) Layout of a memory portion in the first region of a SDRAM bank storing the coordinates and electric charge of a particle i (c) Layout of another memory portion in the first region of a SDRAM bank storing the coordinates and electric charge of a j particle as well as the interaction parameters and the cutoff distances for the particular pair of i and j particles (d) Layout of a memory portion in the second region of a SDRAM bank storing the force, potential and virial values computed for the specific pair of i and j particles*

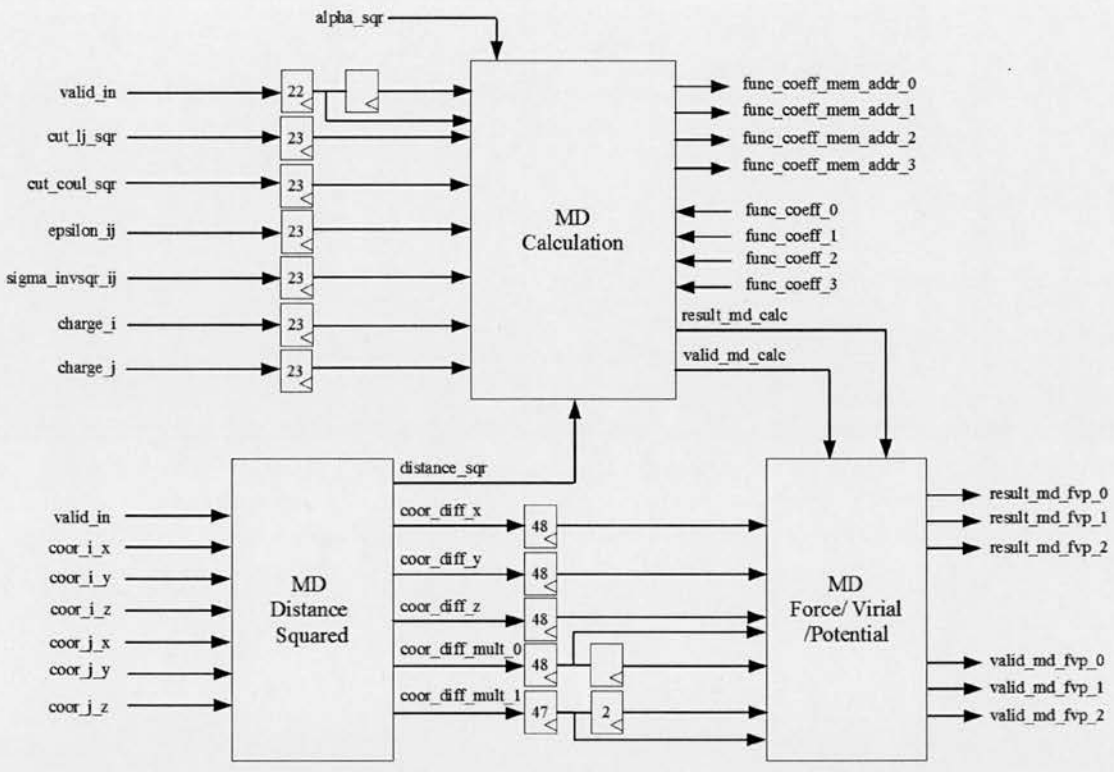


Figure 6.5: *Functional block diagram of a Molecular Dynamics processor*

Figure 6.5 above shows the functional block diagram of the proposed MD processor. It contains a pipeline comprised of three major functional units, namely *MD Squared Distance* unit, *MD Calculation* unit and *MD Force/Virial/Potential* unit in the order presented. The MD processor whose operating frequency is 150 MHz calculates the non-bonded interactions in the simulated molecular system as stated above. The detailed architectures and operations of the three functional units in the MD processor will be described in subsections 6.6.1, 6.6.2 and 6.6.3, respectively.

6.6.1 MD Distance Squared Unit

Figure 6.6 shows a simplified pipeline architecture of the first functional unit in the proposed MD processor, *MD Squared Distance* unit whose primary duty is to calculate the squared distance between an i particle and the j particles in the neighbour list of that i particle. When a MD processor is triggered by

the host CPU, it begins to transfer simulation data of i and j particles into its input buffer (see figure 6.3) from the first region of its associated SDRAM bank. Input buffers in the proposed design make use of double buffering so as to enhance the efficiency of the data transfers from a memory bank and hence, increase the operation speed of the MD processors. Note that each of these double buffers has a width of 256 bits, so two logical words are transferred one by one from a SDRAM bank to make up one word of the buffer.

When an input buffer is completely full with data, the *MD Squared Distance* unit starts to read one word from the buffer every four clock cycles. If it is detected that the read word contains the coordinates and electric charge of an i particle, those coordinates are registered separately in the unit (not shown in figure 6.6) whereas the charge value is shifted towards the *MD Calculation* unit in a register array as shown in figure 6.5. On the other hand, if the read word contains data related to a j particle, coordinate values in that word are registered and then pushed into the pipeline with the *valid_in* signal asserted for four clock cycles, while the rest of the data in the word (see figure 6.4 (c)) are seperately shifted towards the *MD Calculation* unit in five register arrays.

When the coordinate values of a j particle enters the pipeline, three floating-point subtractors are used to compute the coordinate differences between the registered i particle and that j particle in three dimensions, $d_{ij} = (d_x, d_y, d_z)$, independently in parallel. These coordinate differences are shifted separately towards the end of the unit in three register arrays to be passed to the *MD Force/Virial/Potential* unit. Furthermore, two floating-point multipliers compute the two squared coordinate differences in x and y dimensions, d_x^2, d_y^2, and the squared coordinate difference in z dimension, d_z^2, in different clock cycles through the use of the three multiplexers whose control signal values are determined depending on the value of the 2-bit counter *valid_cnt* which increments by one with the high value of the delayed *valid_in* signal as shown in figure 6.6. Table 6.1 shows how the values of the control signals for

the multiplexers vary depending on the value of the counter *valid_cnt*. With these control signal values, two floating-point multipliers also compute the following products of the coordinate differences: d_xd_y, d_xd_z and d_yd_z in addition to d_x^2, d_y^2 and d_z^2 in an order dictated by the multiplexers. Moreover, all of these coordinate difference products are shifted towards the end of the unit in two register arrays to be passed to the *MD Force/Virial/ Potential* unit for further computations.

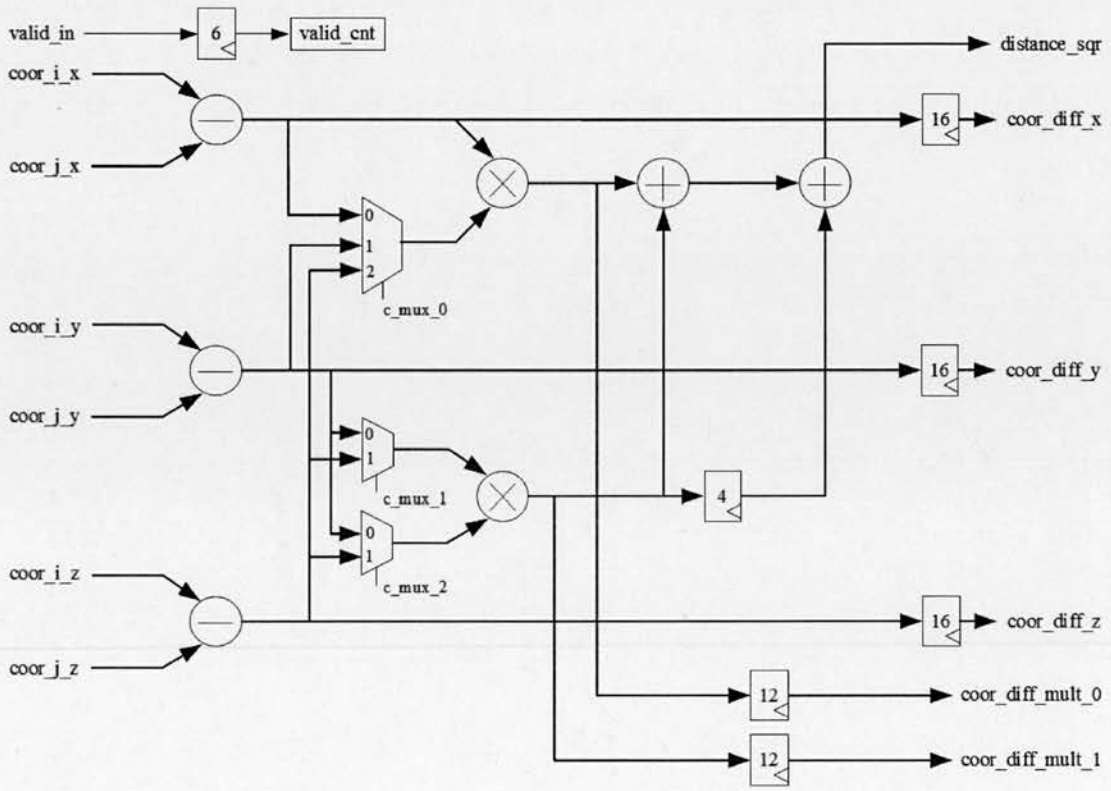


Figure 6.6: *Simplified pipeline architecture of the MD Distance Squared unit*

Finally, the first floating-point adder in the unit computes the sum of the squared coordinate differences in x and y dimensions, $d_x^2 + d_y^2$, while the second floating-point adder adds the squared coordinate difference in z dimension, d_z^2, to this sum to calculate the squared distance, $r^2_{ij} = d_x^2 + d_y^2 + d_z^2$, between a pair of i and j particles. Furthermore, this value is passed to

the *MD Calculation* unit to evaluate several functions of distance. Note that the pipeline latency of the *MD Squared Distance* unit is 22 clock cycles.

Table 6.1: *Control signal values for the three multiplexers in the MD Distance Squared unit*

valid_cnt	0	1	2	3
c_mux_0	0	1	2	X
c_mux_1	0	0	1	X
c_mux_2	0	1	1	X

6.6.2 MD Calculation Unit

Figure 6.7 shows the simplified pipeline architecture of the second and largest functional unit in the proposed MD processor, *MD Calculation* unit whose primary duty is to calculate and separately multiply the first two terms in (6.14) and (6.15), and both terms in (6.16) and (6.17). The multiplied terms are then passed to the *MD Force/Virial/Potential* unit for the computations of the pairwise forces, virials and potentials due to both Lennard-Jones and Coulombic interactions between an i particle and the j particles in the neighbour list of that i particle. Note that only the short-range portion of the Coulombic interactions is computed in the proposed MD processor.

$$\boldsymbol{f}_{ij}^{LJ} = \frac{\epsilon_{ij}}{\sigma_{ij}^2} \cdot g_1\left(\frac{r_{ij}^2}{\sigma_{ij}^2}\right) . \boldsymbol{r}_{ij} \quad (6.14)$$

$$\boldsymbol{f}_{ij}^{C} = q_i q_j \cdot g_2(\alpha^2 r_{ij}^2) . \boldsymbol{r}_{ij} \quad (6.15)$$

$$\Phi_{ij}^{LJ} = \epsilon_{ij} . g_3\left(\frac{r_{ij}^2}{\sigma_{ij}^2}\right) \quad (6.16)$$

$$\Phi_{ij}^{C} = q_i q_j . g_4\left(\alpha^2 r_{ij}^2\right) \quad (6.17)$$

When the squared distance between a pair of i and j particles is passed to the *MD Calculate* unit by the *MD Squared Distance* unit, this value is registered to be valid for four clock cycles with the asserted *valid_in* signal. Then, two floating-point comparators in the unit compare the registered squared distance with the specified cutoff distances for the Lennard-Jones and Coulombic interactions, respectively. If the squared distance happens to be bigger than the cutoff distance for any interaction, then the forces, virials and potentials due to that interaction are set to be zero for the particular pair of i and j particles. Furthermore, utilizing the values shifted into the unit as shown in figure 6.5, the first terms in (6.14), (6.15), (6.16) and (6.17) are computed to be available at the output of the multiplexer *mux_3* in four consecutive clock cycles through the use of the floating-point multiplier *mult_0* and the two multiplexers, *mux_0* and *mux_1*, shown in figure 6.7. Control signal values of the mentioned multiplexers and the multiplexer *mux_2* are determined depending on the value of the 2-bit counter *valid_cnt* as shown in Table 6.2.

Table 6.2: *Control signal values for the four multiplexers in the MD Calculator unit*

valid_cnt	**0**	**1**	**2**	**3**
c_mux_0	0	1	0	1
c_mux_1	0	1	0	1
c_mux_2	0	1	0	1
c_mux_3	0	0	1	0

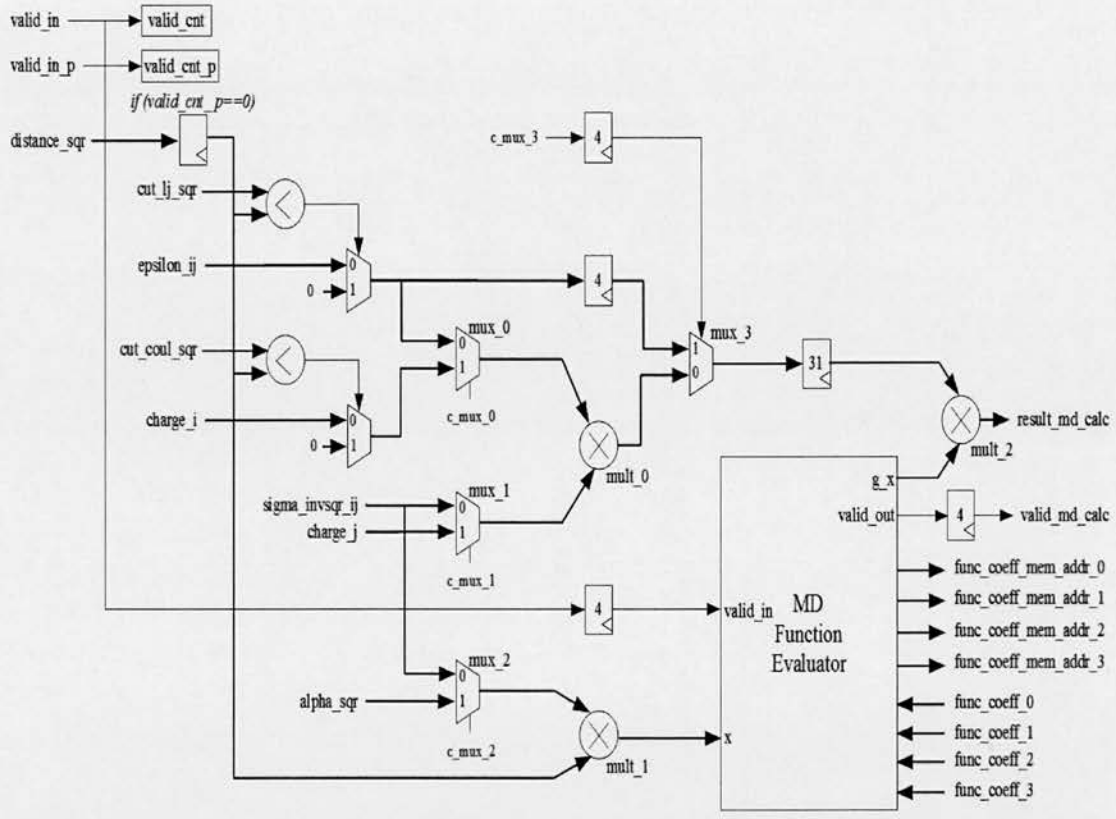


Figure 6.7: *Simplified pipeline architecture of the MD Calculation unit*

On the other hand, the floating-point multiplier *mult_1* computes the arguments of the functions $g_1(x)$, $g_2(x)$, $g_3(x)$ and $g_4(x)$, which are in the second terms of (6.14), (6.15), (6.16) and (6.17), respectively, in four consecutive clock cycles through the use of the multiplexer *mux_2*. These computed arguments are then passed to the *MD Function Evaluator* unit to be evaluated in their corresponding function in a pipelined manner with a latency of 31 clock cycles. The *MD Function Evaluator* unit will be elaborated in subsection 6.6.2.1. Moreover, the functions $g_1(x)$, $g_2(x)$, $g_3(x)$ and $g_4(x)$ are expressed below:

$$g_1(x) = 48x^{-7} - 24x^{-4} \quad (6.18)$$

$$g_2(x) = erfc(\sqrt{x})x^{-3/2} + exp(-x)x^{-1} \quad (6.19)$$

$$g_3(x) = 4x^{-6} - 4x^{-3} \quad (6.20)$$

$$g_4(x) = erfc(\sqrt{x})x^{-1/2} \quad (6.21)$$

Finally, the floating-point multiplier *mult_2* multiplies the delayed output of the multiplexer *mux_3* and the output of the *MD Function Evaluator* unit, and hence gets the first two terms in (6.14) and (6.15) and both terms in (6.16) and (6.17) multiplied in 4 consecutive clock cycles for a pair of i and j particles. These results are then sent to the *MD Force/Virial/Potential* unit with the asserted *valid_md_calc* signal for further processing. Note that the pipeline latency of the *MD Calculator* unit is 40 clock cycles.

6.6.2.1 MD Function Evaluation Unit

Figure 6.9 shows the simplified pipeline architecture of a functional unit in the proposed *MD Calculation* unit, namely *MD Function Evaluation* unit which evaluates the functions $g_1(x)$, $g_2(x)$, $g_3(x)$ and $g_4(x)$, which are expressed respectively in (6.18), (6.19), (6.20) and (6.21), consecutively in four clock cycles using the piecewise third-order polynomial interpolation. When the argument x enters the unit with the asserted *valid_in* signal, it is decomposed into two numbers, x_{addr} and x_{interp}, as shown in figure 6.8.

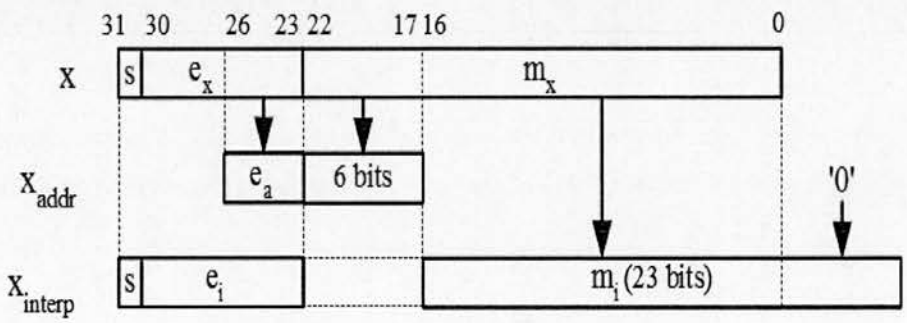


Figure 6.8: *The operation required for the polynomial interpolation with a look-up table to evaluate the functions $g_1(x)$, $g_2(x)$, $g_3(x)$ and $g_4(x)$*

The floating-point number x_{addr} represents the argument x in a range $[2^{-5}, 2^{11})$ using 10 bits, 4 for the exponent and 6 for the mantissa. The latter is the copy of the most significant bits of the mantissa of x, namely m_x. The unit calculates the value of $x - x_{addr} \cdot 2^{-e_a + e_x}$, where e_a and e_x are the exponents of x_{addr} and x, respectively, and normalizes it in a 32-bit floating-point number x_{interp}. Actually, the exponent of x_{interp}, namely e_i, is equal to e_x-7-n, where n is the number of leading zeros in the 17 least significant bits of x. Then, a function g(x) can be approximated with x_{interp} as follows, where c_3, c_2, c_1 and c_0 are the coefficients of the piecewise polynomial interpolation:

$$g(x) = \left(\left(c_3 x_{interp} + c_2 \right) x_{interp} + c_1 \right) x_{interp} + c_0 \quad (6.22)$$

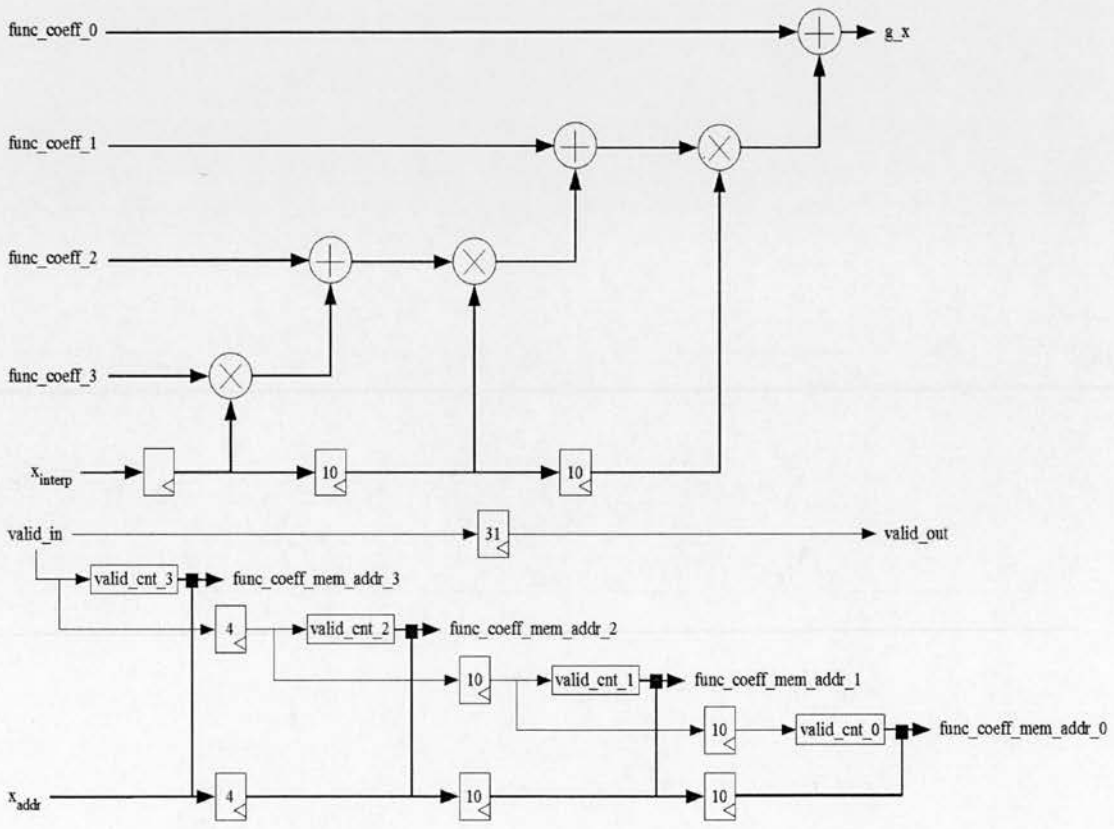


Figure 6.9: *Simplified pipeline architecture of the MD Function Evaluator unit*

A set of the quadruples of the coefficients (i.e. a look-up table) is stored in the

two identical *Function Coefficients* memories shown in figure 6.3. Note that each *Function Coefficients* memory serves two separate MD processors with its dual port. These memories are comprised of four sub-memories storing one of the four piecewise interpolation coefficients (i.e. c_3, c_2, c_1, c_0) for four functions (i.e. $g_1(x)$, $g_2(x)$, $g_3(x)$, $g_4(x)$) in four separate regions. Each sub-memory is a 4096 x 32 bits Block RAM with an address width of 12 bits. The layout of a sub-memory in the *Function Coefficients* memory is shown in figure 6.10.

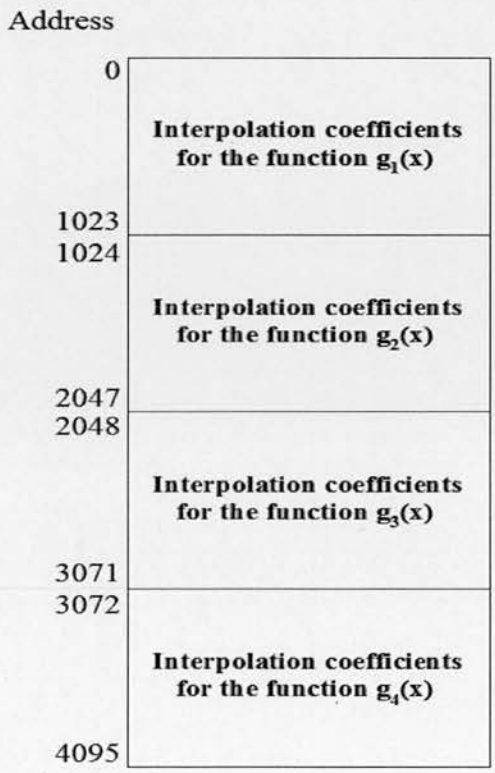


Figure 6.10: *The layout of a sub-memory in the Function Coefficients memory storing one of the four piecewise interpolation coefficients for the functions $g_1(x)$, $g_2(x)$, $g_3(x)$ and $g_4(x)$ in four separate regions*

Furthermore, four differently delayed values of the 10-bit number x_{addr} are used as part of the addresses for accessing the four sub-memories individually, as shown in figure 6.9. On the other hand, the two most significant bits of the four addresses, coming respectively from the four 2-bit counters in the

unit, are used to select one out of four tables (functions) stored in the sub-memories. With this configuration, *MD Function Evaluation* unit evaluates the functions $g_1(x)$, $g_2(x)$, $g_3(x)$ and $g_4(x)$ for a pair of i and j particles in four consecutive clock cycles with a latency of 31 clock cycles.

6.6.3 MD Force/Virial/Potential Unit

Figure 6.11 shows the simplified pipeline architecture of the third and final functional unit in the proposed MD processor, *MD Force/Virial/Potential* unit whose primary duty is to compute the pairwise virials and forces acting on an i particle due to both Lennard-Jones and Coulombic interactions with the j particles in the neighbour list of that i particle. In four consecutive clock cycles, the multiplied first two terms of (6.14), (6.15), (6.16) and (6.17) are pushed one by one into the unit by the *MD Calculation* unit with the *valid_in* signal asserted for four clock cycles. Obviously, these four products pertain to the LJ force, Coulombic force, LJ potential and Coulombic potential for a pair of i and j particles, respectively.

The first and second values entering the unit are registered separately in the first two clock cycles under the control of the 2-bit counter *valid_cnt*, which increments by the high value of the *valid_in* signal, as shown in figure 6.11. These registered values are then added up by the floating-point adder in the unit and the result is passed to the three floating-point multipliers. On the other hand, the third and fourth values are buffered respectively in the synchronous write, asynchronous read buffers *fifo_elj* and *fifo_ecb* in the third and fourth clock cycles under the control of the counter *valid_cnt*. These bufferings last until the end of the operation of these multipliers, as will be explained later.

Furthermore, the three floating-point multipliers compute all three components of the total pairwise force $\mathbf{f}_{ij}$ (i.e. f_x, f_y, f_z) and six components of the

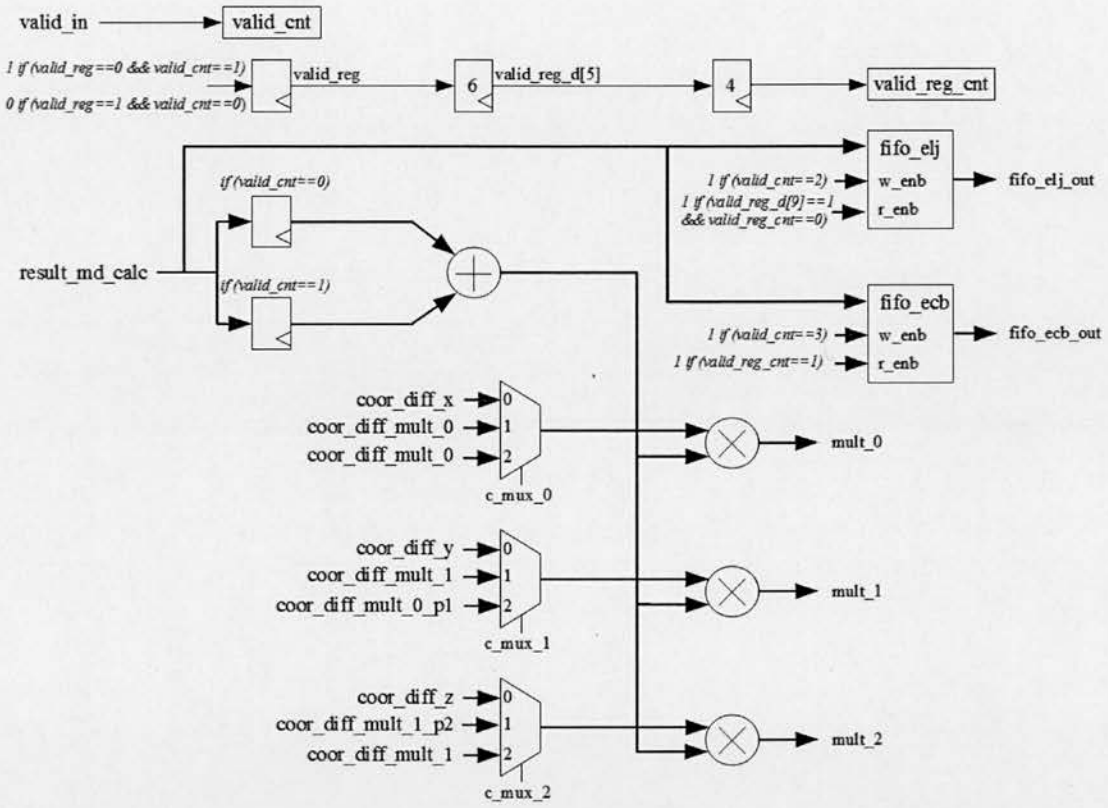


Figure 6.11: *Simplified pipeline architecture of the MD Force/Virial/Potential unit*

pairwise virial $\mathbf{v}_{ij}$ (i.e. v_x^2, v_y^2, v_y^2, v_{xy}, v_{xz}, v_{yz}) in parallel in three consecutive clock cycles by multiplying the output of the floating-point adder with the coordinate differences (i.e. d_x, d_y, d_z) and the coordinate difference products (i.e. d_x^2, d_y^2, d_z^2, d_xd_y, d_xd_z, d_yd_z) ,which are shifted into the pipeline from the *MD Squared Distance* unit for a pair of i and j particles, through the use of three multiplexers, as shown in figure 6.11. In the first clock cycle, the output of the adder is multiplied by the coordinate differences, d_x, d_y and d_z, to calculate the components of the total pairwise force, f_x, f_y, f_z, whereas the adder output is multiplied by the following coordinate difference products: d_x^2, d_y^2 and d_z^2 to compute the following three components of the pairwise virial: v_x^2, v_y^2 and v_y^2 in the second clock cycle. Finally, in the third clock cycle, the following coordinate difference products: d_xd_y, d_xd_z and d_yd_z are multiplied by the output of the adder to calculate the following three components of the

pairwise virial: v_{xy}, v_{xz} and v_{yz}. Note that the control signal values of the three multiplexers in the unit are determined depending on the value of the 2-bit counter *valid_cnt_2* (not shown in figure 6.11) which counts up to three with the high value of the *valid_reg_d[9]* signal. Table 6.3 shows how the values of the control signals for the multiplexers vary depending on the value of the counter *valid_cnt_2*.

Table 6.3: *Control signal values for the three multiplexers in the MD Force/Virial/Potential unit*

valid_cnt_2	**0**	**1**	**2**
c_mux_0	0	1	2
c_mux_1	0	1	2
c_mux_2	0	1	2

As the multipliers finish their operations, their outputs *mult_0*, *mult_1* and *mult_2* are concatenated into a word which is written to the output buffer of the MD processor in three consecutive clock cycles, as shown in figure 6.3. Furthermore, the pairwise LJ potential e^{LJ} in the corresponding location of the *fifo_elj* buffer and the pairwise Coulombic potential e^{C} in the corresponding location of the *fifo_ecb* buffer are incorporated into that word in the first and second clock cycles, respectively, extending its width to 128 bits. When an output buffer is completely full with data, its content is flushed into the second region of the associated SDRAM bank. Layout of a memory portion in the second region of a SDRAM bank is shown in figure 6.4 (d). Moreover, the 128-bit output buffers in the proposed design make use of double buffering so as to enhance the efficiency of the data transfers to a memory bank and hence, increase the operation speed of the MD processors. Note that the pipeline latency of the *MD Force/Virial/Potential* unit is 12 clock cycles.

6.7 Implementation Results

Molecular Dynamics simulations were implemented on the Alpha Data nodes of the Maxwell machine with the MD processor cores shown in figure 6.3, each of which incorporating four MD processors working independently in parallel with a total pipeline latency of 74 clock cycles. The proposed MD core was written in Verilog language while the interfaces of the user FPGA with the local bus and the DDR2 SDRAM banks were provided by the Alpha Data in the VHDL language. The design was then synthesized, placed, and routed by the Xilinx ISE 11.5 tool. FPGA bitstreams were also generated by the same tool while the ModelSim tool was employed to test the MD core with a number of testbenches. Note that there is only one FPGA bitstream used to configure all FPGAs in the MD machine regardless of the number of atoms in the simulated system. Furthermore, MATLAB tool was used to compute the piecewise polynomial interpolation coefficients for the evaluation of the several functions needed, as explained in subsection 6.6.2.1.

The clock frequency of the user FPGAs for the local bus interface was set to be 80 MHz whereas the clock frequency for the MD core was set to be 150 MHz. Due to this clock frequency of the MD core, the clock frequency for the DDR2 SDRAM banks was 300 MHz. For benchmark purposes, an all-atom Rhodopsin protein in solvated lipid bilayer was simulated with the Lennard-Jones forces, and the Coulombic forces via PPPM (particle-particle particle mesh), incorporating SHAKE constraints. This model contains counter-ions and a reduced amount of water to make a 32K atom system. The details of the simulation are as follows:

- 32,000 atoms for one time-step
- LJ and Coulombic force cutoff of 10.0 Angstroms

- Neighbor skin of 2.0 Angstroms
- Average neighbors per atom = 372 atoms
- NVT time integration

Table 6.4 presents the timing performance figures of the LAMMPS software for the pairwise LJ and short-range Coulombic interaction computations of the above mentioned Rhodopsin protein system on two nodes of the Maxwell machine (i.e. two software processes running on one host Intel Xeon CPU). The protein system was replicated in X, Y or Z dimensions to achieve the simulation of systems with up to 256,000 atoms, as presented in table 6.4.

Table 6.4: *Timing figures of the LAMMPS software for the pairwise interaction computations on two Maxwell nodes*

No. of Atoms	No. of Bonds	No. of Angles	No. of Dihedrals	Computation Time (s)
32000	27723	40467	56829	5.051342
64000	55446	80934	113658	9.911402
128000	110892	161868	227316	20.407846
256000	221784	323736	454632	40.746851

For comparative purposes, table 6.5 below shows the timing performance figures of the MD machine configured to operate in the same way as the pure software implementation on two nodes of the Maxwell machine (i.e. two software processes running on one host Intel Xeon CPU and MD core instances on two Xilinx Virtex-4 XC4VFX100 FPGAs [76]). Note that the timing figures presented in table 6.5 do not include the I/O communication costs occurring during the data transfers between a host CPU and SDRAM banks.

Table 6.5: *Timing figures of the MD machine for the pairwise interaction computations on two Maxwell nodes*

No. of Atoms	No. of Bonds	No. of Angles	No. of Dihedrals	Computation Time (s)
32000	27723	40467	56829	0.379309
64000	55446	80934	113658	0.785921
128000	110892	161868	227316	1.674183
256000	221784	323736	454632	3.130863

Figure 6.12 plots the timing performance results of the pure software implementation and the MD machine for the pairwise interaction computations on two nodes of the Maxwell, as shown in tables 6.4 and 6.5, respectively. As it can be seen, at all atom sytems, the MD machine operates faster than the pure software implementation. Note that both plots in figure 6.12 show a quadratically increasing curve which is obviously much sharper for the pure software solution for the MD simulations.

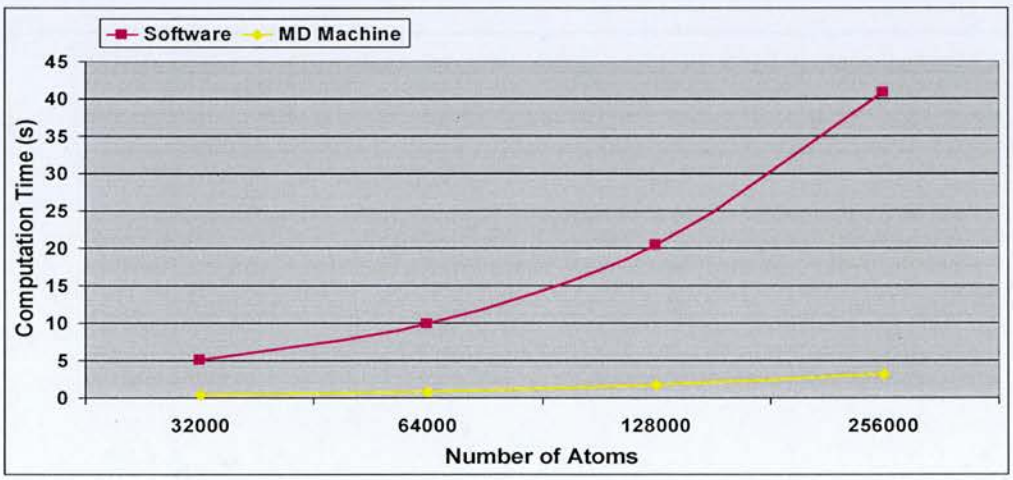


Figure 6.12: *Timing performance plot of the LAMMPS software and the MD machine for the pairwise interaction computations on two nodes of the Maxwell*

Table 6.6 below provides the speed-up values of the MD machine over the pure software implementation (i.e. LAMMPS) for the pairwise interaction computations of the systems with various numbers of atoms on two Maxwell nodes. Note that the MD machine outperforms the pure software implementation by 12x-13x.

Table 6.6: *LAMMPS versus MD machine speed-up values for the interaction computations on two Maxwell nodes*

No. of Atoms	MD Machine Speed-Up
32000	13.32
64000	12.61
128000	12.19
256000	13.01

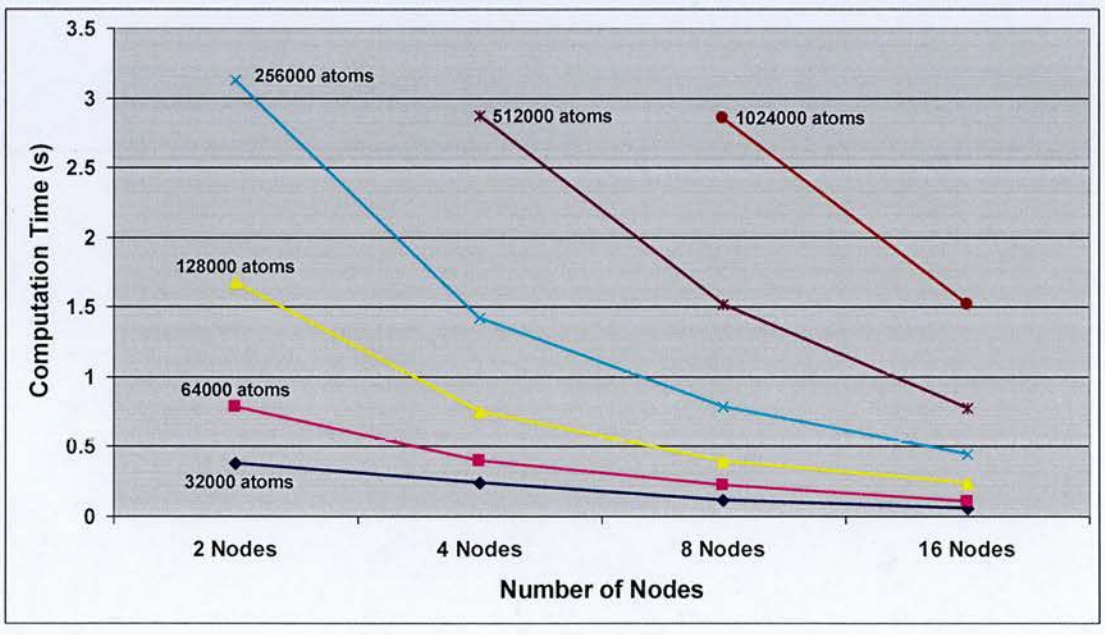


Figure 6.13: *Scaling performance of the MD machine on different numbers of nodes of the Maxwell for the given numbers of atoms*

Table 6.7 shows the timing performance figures of the MD machine on two Maxwell nodes, this time including the I/O communication costs of the data transfers between a host CPU and SDRAM banks, as opposed to table 6.5. As it can be seen, I/O communication times account for over 96 percent of the total time. Due to this very high cost of the communication, the overall timing performance of the MD machine is poor compared to the pure software implementation for all atom systems (refer to table 6.4). Although multithreading, where each of the four existing threads deals with its assigned MD processor in the MD core, was utilised in the software process to take advantage of the direct memory transfers (DMA), the total time could not be reduced to a desired level because of the very poor data bandwidth between the host CPU and SDRAM banks. Nonetheless this limitation is not conceptual but rather dependent on the hardware platform targeted in this implementation. This communication bottleneck can be significantly resolved by integrating FPGA boards tighter into the host systems. In this way, FPGAs will have high-speed access to host memory through, for instance, AMD's Hypertransport, Intel's Quick Path Interconnect or SGI's NumaLink which offer bandwidths ranging from 15 to 25.6 GB/s, thus reducing communication overheads by at least 100x in comparison to our currently used communication link. This would result in performance gains of the MD machine in overall over the pure software implementation by almost same factors listed in table 6.6.

Table 6.7: *Timing figures of the MD machine including I/O communication costs on two Maxwell nodes*

No. of Atoms	Total Time (s)	I/O Comm. Time (s)	Percent. of I/O Comm.
32000	11.202145	10.822836	% 96.61

64000	22.298593	21.512672	% 96.48
128000	44.749425	43.075242	% 96.26
256000	89.581439	86.450576	% 96.50

Tables 6.8, 6.9 and 6.10 below show the comparative timing figures of the pure software implementation and the MD machine for the pairwise interaction computations of the Rhodopsin protein systems with up to over two million atoms on 4, 8 and 16 nodes of the Maxwell machine, respectively. As it can be seen, the MD machine speed-up values for the pairwise interaction computations range from 10x to 14x. In addition, the efficiency and scalability of the MD machine on different numbers of nodes of the Maxwell is graphically represented in figure 6.13 with the timing values for the given numbers of atoms, as presented in tables 6.5, 6.8, 6.9 and 6.10. Note that the computational power of the proposed MD machine increases highly with the increasing number of the Maxwell nodes utilized.

Table 6.8: *Comparative timing figures of the LAMMPS software and the MD machine on four Maxwell nodes*

No. of Atoms	SW Comp. Time (s)	MD Machine Comp. Time (s)	MD Machine Speed-Up
32000	2.467384	0.237942	10.37
64000	4.959632	0.398379	12.45
128000	10.006767	0.748793	13.36
256000	20.08099	1.416626	14.18
512000	39.751763	2.869737	13.85

Table 6.9: *Comparative timing figures of the LAMMPS software and the MD machine on eight Maxwell nodes*

No. of Atoms	SW Comp. Time (s)	MD Machine Comp. Time (s)	MD Machine Speed-Up
32000	1.202016	0.118279	10.16
64000	2.444738	0.220909	11.07
128000	4.856528	0.392149	12.38
256000	9.857229	0.781985	12.61
512000	19.563335	1.516907	12.90
1024000	40.567838	2.854312	14.21

Table 6.10: *Comparative timing figures of the LAMMPS software and the MD machine on sixteen Maxwell nodes*

No. of Atoms	SW Comp. Time (s)	MD Machine Comp. Time (s)	MD Machine Speed-Up
32000	0.610724	0.05701	10.71
64000	1.211133	0.111407	10.87
128000	2.406081	0.240549	10.00
256000	4.96922	0.441922	11.24
512000	9.783473	0.774302	12.64
1024000	19.683968	1.516273	12.98
2048000	40.287101	2.891285	13.93

Table 6.11 shows the resource utilization of the MD core in a user FPGA. Note that the total number of the floating-point adder/subtractors in the MD core is 36 while the total number of the floating-point multipliers is 44. In

addition, 8 floating-point comparators are also utlized in the proposed design. Furthermore, the floating-point adder/subtractors and comparators were entirely implemented in the slice logic. On the other hand, the floating-point multipliers were partially implemented in the DSP48 blocks on the FPGA. However, since each multiplier requires 4 DSP48 blocks and the total number of the DSP48 blocks in the user FPGA is just 160, it was only possible to map 40 of the multipliers to the DSP48 blocks while the rest of them were entirely implemented in the user logic.

The accuracy in the computations was sufficient enough to carry out stable MD simulations but the accuracy could be improved if the single extended precision (i.e. width of 40-bit) was used for the floating-point numbers inside the design rather than the single precision (i.e. width of 32-bit) [109]. However, this precision increase would require higher amounts of slice logic and DSP48 blocks to implement the floating-point arithmetic units utilised in the MD core. Unfortunately, currently used Xilinx Virtex-4 XC4VFX100 FPGA chips can not accommodate any higher resource demand as can be clearly seen in table 6.11.

Furthermore, it is reported by [109] that the evaluation of the functions, which involves the piecewise third-order polynomial interpolation with a look-up table, requires a key with a width of at least 15 bits (see subsection 6.6.2.1). However, even 1 bit increase in the width of the used key would require doubling the size of the utilized Function Coefficients memories (see figure 6.3). It is also impossible to realize the usage of 15-bit wide key considering the amount of Block RAMs available in the currently used Virtex-4 FX100 FPGA chip (refer to table 6.11).

Table 6.11: *Resource utilization of the MD core in a user FPGA*

	Used	Available	Utilization
Number of Occupied Slices	39,880	42,176	% 94
Total Number of 4 Input LUTs	69,622	84,352	% 82
Number of Slice Flip Flops	43,021	84,352	% 51
Number of FIFO16/RAMB16s	280	376	% 74
Number of DSP48s	160	160	% 100

6.8 Conclusions

The design and implementation of a FPGA core, namely MD core, carrying out all the necessary operations to compute the non-bonded interactions in a MD simulation with the purpose of accelerating the LAMMPS MD software was presented in this chapter. The proposed MD processor core comprised of 4 identical pipelines working independently in parallel to evaluate the non-bonded potentials, forces and virials was implemented on the nodes of a FPGA-based supercomputer, named Maxwell, which consists of 64 Virtex-4 FPGA chips. This implementation allowed us to produce a special-purpose parallel machine for the hardware acceleration of the MD simulations. This machine yields higher computational power with the additional Maxwell nodes, making it highly scalable.

The timing performance figures of the MD machine for the pairwise LJ and short-range Coulombic (via PPPM) interaction computations in the MD

simulations of the solvated Rhodopsin protein systems with various numbers of atom show performance gains over the pure software implementation by factors of up to 13 on two nodes of the Maxwell machine. These MD machine speed-up values for the pairwise interaction computations were also maintained on different numbers of Maxwell nodes. However, the overall timing performance of the MD machine is worse than the pure software implementation due to the very high I/O communication costs of the data transfers between a host CPU and SDRAM banks. This case stems from the very poor data bandwidth between a host CPU and SDRAM banks which is a limitation caused by the hardware platform targeted in this implementation (i.e. Maxwell FPGA-based supercomputer).

Nonetheless, if FPGA boards are integrated tighter into the host systems through, for instance, AMD's Hypertransport, Intel's Quick Path Interconnect or SGI's NumaLink, the bandwidth of the I/O communications would be greatly enhanced up to 25.6 GB/s, thus yielding much lower communication costs (i.e. up to 100x reduction in comparison with our currently used communication link). This would result in performance gains of the MD machine in overall over the pure software implementation. On the other hand, the accuracy of the computations could be improved if the number of slices and DSP48 blocks available in the user FPGA (i.e. Xilinx Virtex-4 XC4VFX100) was higher. Furthermore, wider DSP48 blocks and larger block RAMs would also help to enhance the computation accuracy. Solving the aforementioned concerns with a better hardware implementation platform is the major plan for the future of this work.

Chapter 7

Summary and Conclusions

7.1 Introduction

This thesis proposes the use of state-of-the-art reprogrammable system-on-chip technology, in the form of platform FPGAs, as a relatively low cost, high performance and reprogrammable implementation platform in order to cope with the sheer immensity of the data sets involved in BCB algorithms (often measured in tens/hundreds of Gigabytes) as well as their computation demands (often measured in Tera-Ops). The research question in this thesis was to assess the viability of FPGAs as a high performance platform for BCB. Therefore, the aim of this research was to develop a sophisticated library of FPGA architectures for bio-sequence analysis, phylogenetic analysis, and molecular dynamics simulation. The results of these case studies were then studied to assess the viability of FPGAs as an alternative implementation platform for BCB applications.

This final chapter will first summarize the work presented in each of the previous chapters and then draw together the conclusions reached from results presented in chapters 4, 5 and 6. Potential ideas for future research extensions are also discussed.

7.2 Thesis Summary

Reconfigurable Computing (RC) is emerging as a new computing paradigm with the commercial availability of reconfigurable logic which is a special kind of hardware circuit that can be reconfigured into whatever logic the

user desires by programming some kind of configuration memory. Chapter 2 presented fundamentals and characteristics of reconfigurable computing, the use of programmable logic to accelerate computation. Furthermore, specific reconfigurable computing architectures were briefly outlined together with the reconfiguration technology. Moreover, mapping algorithms to reconfigurable hardware was introduced and various application fields for reconfigurable computing were highlighted.

FPGAs were the first significantly available field-programmable devices that achieved enough density to perform significant portions of a computation. Arrays of simple logic functions and memories (e.g. flip-flops) which can be connected through programmable interconnection networks are provided in these chips for the designer. Chapter 3 introduced FPGAs that has been widely used in reconfigurable computing, and provided a brief overview of its basic architecture, programming the architecture and additional specialized function resources. Furthermore, an FPGA-based supercomputer named Maxwell, our hardware implementation platform for two case studies as explained in chapters 5 and 6, was briefly described towards the end of the chapter.

Aligning subject sequences from a large biological database to a query sequence to find similarities between the query sequence and the database sequences is a very common task in BCB. However, sequence alignment is a computationally intensive operation, and desktop computers alone cannot be relied upon to perform this task within acceptable time periods since biological sequence databases are growing at an exponential rate. To address this problem, chapter 4 presented an FPGA implementation of the Position Specific Iterated BLAST (PSI-BLAST) which is a heuristic biological sequence alignment algorithm widely used by the BCB community in order to detect distant relationships among query and database sequences. Subsequently, implementation results were presented and then evaluated comparatively

with the performance of equivalent software implementations running on a desktop computer. It was seen that our implementation outperformed an equivalent desktop-based software implementation by at least one order-of magnitude.

Phylogenetic analysis, investigation of the evolution and relationships among organisms, is widely used in the fields of system biology and comparative genomics. In molecular-based phylogenetic analysis, the relationship between species is estimated by inferring the common history of their genes and then phylogenetic trees are constructed to illustrate evolutionary relationships among genes and organisms. However, phylogenetic tree construction which is a computationally intensive operation takes a very long time on conventional processors due to the number of theoretically possible tree topologies growing exponentially with the number of species under consideration. Hence, chapter 5 presented the detailed design of a FPGA core for molecular-based phylogenetic analysis with Maximum Parsimony (MP) method and its implementation on the nodes of an FPGA-based supercomputer named Maxwell. Subsequently, implementation results were presented and then evaluated comparatively with equivalent software implementations running on a desktop computer. It was observed that our implementation outperformed an equivalent desktop-based software implementation (i.e. PAUP) by very high orders-of magnitude.

Molecular Dynamics (MD), a deterministic simulation technique often performed to help understand the properties of assemblies of molecules in terms of their structure and the microscopic interactions between them, act as a bridge between microscopic length and time scales and the macroscopic world of the laboratory, serving as a complement to conventional experiments. However, biological systems of interest have sizes ranging from a few tens of thousands to millions of atoms and thus performing MD

simulation of a biological process for a reasonable physical time requires enormous amounts of computational effort, taking years to complete on conventional computers. To address this problem, chapter 6 presented the detailed design of a MD processor core which parallelises all the necessary operations to compute the non-bonded interactions in the LAMMPS software tool and its implementation on the nodes of the Maxwell FPGA-based supercomputer. Subsequently, implementation results were presented and then evaluated comparatively with equivalent pure software implementations on Maxwell nodes. Note that our implementation showed performance gains over the pure software implementation by factors of up to 13 on two nodes of the Maxwell machine

7.3 Evaluation and Conclusions

The detailed FPGA implementation of the PSI-BLAST algorithm was presented in chapter 4. The architecture of this FPGA core which is parameterized in terms of the sequence lengths, scoring matrix, gap penalties and cut-off and threshold values was composed of various blocks each of which performing a specific step of the algorithm in parallel. The resulting implementation outperformed an equivalent desktop-based software implementation by at least one order-of magnitude. Furthermore, the core was designed in the Handel-C language, thus making it FPGA-platform-independent. This means that the same core can be ported to all other FPGA architectures from different vendors. However, the drawback of using Handel-C was that the achieved clock frequency for the FPGA design was relatively low for a Virtex-4 FPGA chip. Note that using Verilog HDL in place of Handel-C would improve this frequency drastically.

In chapter 5, the detailed FPGA implementation of the Maximum Parsimony method for molecular phylogenetic analysis on the nodes of the Maxwell

FPGA-based supercomputer was presented. The architecture of this core was a linear systolic array composed of 20 processing elements each of which performing Sankoff's algorithm for a different tree topology in parallel. In several iterations, this array computes the scores of all tree topologies for a given number of taxa where the currently supported maximum number of taxa is 12 but this number can be easily improved by cascading more hardware blocks. The resulting implementation outperformed an equivalent desktop-based software implementation (i.e. PAUP) by very high orders-of magnitude. For instance, the speed-up values achieved on a single node of Maxwell could reach up to 21606x for the 12-taxa case while implementations on several nodes could yield even higher values.

The design and implementation of a FPGA core carrying out all the necessary operations to compute the non-bonded interactions in a MD simulation with the purpose of accelerating the LAMMPS MD software was presented in this chapter 6. Our MD processor core comprised of 4 identical pipelines working independently in parallel to evaluate the non-bonded potentials, forces and virials was implemented on the nodes of the Maxwell FPGA-based supercomputer which enabled us to produce a special-purpose parallel machine for the hardware acceleration of the MD simulations. The timing performance figures of this MD machine for the pairwise LJ and short-range Coulombic (via PPPM) interaction computations in the MD simulations of systems with various numbers of atom showed performance gains over the pure software implementation by factors of up to 13 on two nodes of the Maxwell machine. However, the overall timing performance of the MD machine was worse than the pure software implementation due to the very high I/O communication costs of the data transfers between a host CPU and SDRAM banks as a result of the very poor data bandwidth between a host CPU and SDRAM banks. Therefore, there is a need for much better coupling of CPU and FPGA boards.

7.4 Future Work

Future work for each case study is set out as follows:

- Future work for the first case study can be a multi-threaded implementation of various flavours of BLAST and other sequence analysis algorithms with a web interface that allows users to submit queries remotely to an FPGA-based server.

- As to the second case study, we plan to extend and improve the existing architecture to be able to support computations for unlimited number of taxa by incorporating a reconfigurable router into the design. Furthermore, we plan to design a web-based interface for our design through which scientists can submit their sequences online for high performance phylogenetic tree construction on an FPGA-based server.

- With regard to the third case study, if FPGA boards are integrated tighter into the host systems through, for instance, AMD's Hypertransport, Intel's Quick Path Interconnect or SGI's NumaLink, the bandwidth of the I/O communications would be greatly enhanced up to 25.6 GB/s, resulting in performance gains of the MD machine in overall over the pure software implementation. On the other hand, the accuracy of the computations could be improved if the number of slices and DSP48 blocks available in the used FPGA chips was higher. Furthermore, wider DSP48 blocks and larger block RAMs would also help to enhance the computation accuracy. Solving the aforementioned concerns with a better hardware implementation platform is the major plan for the future of this research.

In addition, we plan to conduct rigorous evaluation of power consumption and assessment of associated costs (e.g. development time, purchase prices) for all our designed and implemented FPGA cores. Furthermore,

backward/forward compatibility and portability of our FPGA cores can be analyzed in future. Finally, exploration of more sophisticated algorithms for each case study that can yield better FPGA implementation results is in our agenda for future work.

References

[1] M. Gokhale and P. S. Graham, "Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays", *Springer*, 2005.

[2] R. Durbin, S. Eddy, A. Krogh and G. Mitchison, "Biological Sequence Analysis: Probabilistic Models for Proteins and Nucleic Acids", *Cambridge University Press*, Cambridge, UK, 1998.

[3] M. Salemi and A. M. Vandamme, "The Phylogenetic Handbook: A Practical Approach to DNA and Protein Phylogeny", *Cambridge University Press*, 2003.

[4] M. P. Allen and D. J. Tildesley, "Computer Simulation of Liquids", *Oxford University Press*, 1987.

[5] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, D. J. Lipman, "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs, Nucleic Acid Research", *Oxford Journals*, vol. 25, no. 17, pp. 3389-3402, 1997.

[6] K. K. Kidd and L. A. Sgaramella-Zonta, "Phylogenetic analysis: Concepts and methods", *American Journal of Human Genetics*, vol. 23, pp. 235-252, 1971.

[7] G. D. Fasman, "Prediction of Protein Structure and the Principles of Protein Conformations", *Plenum Press*, New York, 1989.

[8] Z. R. Wasserman and C. N. Hodge, "Fitting an inhibitor into the active site of thermolysin: A molecular dynamics case study", *J. Proteins: Structure, Function, and Bioinformatics*, vol. 24, no. 2, pp. 227-237, Feb. 1996.

[9] D. I. Liao, E. Silverton, Y. J. Seok, B. R. Lee, A. Peterkofsky and D. R. Davies, "The first step in sugar transport: crystal structure of the amino terminal domain of enzyme I of the E. coli PEP: sugar phosphotransferase system and a model of the phosphotransfer complex with HPr.", *J. Structure*, vol. 4, no. 7, pp. 861-872, July 1996.

[10] N. L. Greenbaum, I. Radhakrishnan, D. J. Patel and D. Hirsh, "Solution

structure of the donor site of a trans-splicing RNA", *J. Structure*, vol. 4, no. 6, pp. 725-733, June 1996.

[11] NCBI, "GenBank Statistics", available at http://www.ncbi.nlm.nih.gov/genbank/genbankstats.html, Oct. 2010.

[12] P. A. Hsiung, M. D. Santambrogio and C. H Huang, "Reconfigurable System Design and Verification", *Taylor & Francis Group*, LLC, 2009.

[13] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk and P. Y. K. Cheung, "Reconfigurable computing: architectures and design methods", *IEE Proc. Comput. Digit. Tech.*, vol. 152, no. 2, pp. 193-207, March 2005.

[14] T. E. Ghazawi, E. E. Araby, M. Huang, K. Gaj, V. Kindratenko and D. Buell, "The Promise of High-Performance Reconfigurable Computing", *IEEE Computer*, vol. 41, no. 2, pp. 69-76, Feb. 2008.

[15] D. Buell, T. E. Ghazawi, K. Gaj and V. Kindratenko, "High-Performance Reconfigurable Computing", *IEEE Computer*, vol. 40, no. 3, pp. 23-27, March 2007.

[16] R. Anthony, A. Rettberg, D. Chen, I. Jahnich, G. de Boer and C. Ekelin, "Towards a dynamically reconfigurable automotive control system architecture", *Embedded System Design: Topics, Techniques and Trends*, vol. 231, pp. 71-84, May 2007.

[17] T. Geng, L. Liu, S. Yin, M. Zhu, W. Jia and S. Wei, "Parallel implementation of computing-intensive decoding algorithms of H.264 on reconfigurable SoC", *Proc. IEEE Int. Symp. Circuits and Systems*, pp. 1153-1156, May 2010.

[18] P. Graham and B. Nelson, "FPGA-based sonar processing", *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, pp. 201 - 208, 1998.

[19] M.A. Vega-Rodriguez, J.M. Sanchez-Perez and J.A. Gomez-Pulido, "Real time image processing with reconfigurable hardware", *Proc. IEEE Int. Conf. on Electronics, Circuits and Systems*, pp. 213-216, Sep. 2001.

[20] A. Dandalin, V. Prasanna and J. Rolim, "An adaptive crypto-graphic engine for IPSec architectures", *Proc. IEEE Symp. FPGAs for Custom Computing Machines*,

pp. 132-141, April 2000.

[21] D. Lavenier, S. Guyetant, S. Derrien and S. Rubini, "A reconfigurable parallel disk system for filtering genomic banks", *Proc. Int. Conf. Engineering of Reconfigurable Systems and Algorithms*, pp. 153-166, 2003.

[22] T. Hamada, T. Fukushige, A. Kawai and J. Makino, "Prograpе-1: A programmable special-purpose computer for many-body simulations", *Proc. IEEE Int. Symp. FPGAs for Custom Computing Machines*, pp. 256–257, April 1998.

[23] A. Hodjat and I. Verbauwhede, "A 21.54 gbits/s fully pipelined AES processor on FPGA", Proc. *IEEE Symp. FPGAs for Custom Computing Machines*, pp. 308-309, April 2004.

[24] I. Skilarova and A. Ferrari, "Reconfigurable hardware SAT solvers: A survey of systems", *IEEE Trans. Computers*, vol. 53, no. 11, pp. 1449-1461, November 2004.

[25] B.J. LaMeres and C. Gauer, "Dynamic reconfigurable computing architecture for aerospace applications", *Proc. IEEE Conf. Aerospace*, pp.1-6, March 2009.

[26] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, R.Smart, A. Cantle, R. Chamberlain and G. Genest, "Maxwell—a 64 FPGA supercomputer", *Proc. NASA/ESA Conf. Adaptive Hardware Systems*, pp. 287-294, 2007.

[27] FHPCA, Edinburgh, U.K., "The FHPCA website", available at http://www.fhpca.org, Sep. 2010.

[28] Alpha Data Ltd., Edinburgh, U.K., "ADM-XRC-4FX Datasheet", available at http://www.alphadata.co.uk./adm-adm-xrc-4fx.html, May 2007.

[29] Nallatech Ltd., Glasgow, U.K., "H100 Series Datasheet", available at http://www.nallatech.com/meadiLibrary/images/english/5595.pdf, May 2007.

[30] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, R. Smart, A. Cantle, R. Chamberlain and G. Genest, "The FPGA HPC alliance parallel toolkit", *Proc. NASA/ESA Conf. Adaptive Hardware Systems*, pp. 301-310, 2007.

[31] FHPCA, Edinburgh, U.K., "PowerPoint presentation", available at http://www.fhpca.org/download/MRSC07-Mar07.ppt, Mar. 2007.

[32] FHPCA, Edinburgh, U.K., "PowerPoint presentation", available at http://www.fhpca.org/download/ParallelToolkit-general.ppt, Mar. 2007.

[33] Cray Inc., Seattle, WA, USA, available at http://www.cray.com, Sep. 2010.

[34] SGI Inc., Fremont, CA, USA, available at http://www.sgi.com, Sep. 2010.

[35] SRC Computers Inc., Colorado Springs, CO, USA, available at http://www.srccomp.com, Sep. 2010.

[36] J. Hein, "A New Methodology that simultaneously aligns and reconstructs ancestral sequences for any number of homologous sequences, when a phylogeny is given", *J. Molecular Biology*, vol. 6, pp. 649-668, 1989.

[37] D. T. Hoang, "Searching genetic databases on Splash 2", *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, pp. 185-191, 1993.

[38] M. Gokhale, B. Holmes and K. Iobst, "Processing in memory: The Terasys massively parallel PIM array", *Computer*, vol. 28, no. 4, pp. 23-31, April 1995.

[39] TimeLogic Corporation, "Decypher Scalable, High Performance Biocomputing Solutions", available at http://www.timelogic.com, May 2008.

[40] S. Needleman and C. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two sequences", *J. Molecular Biology*, vol. 48, no. 3, pp. 443-453, 1970.

[41] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences", *J. Molecular Biology*, vol. 147, pp. 195-197, 1981.

[42] W. R. Pearson and D. J. Lipman, "FASTA: Improved tools for biological sequence comparison", *Proc. National Academy of Sciences*, pp. 2444-2448, 1988.

[43] S. F. Altschul, W. Gish, W. Miller, E. W. Myers and D. J. Lipman, "Basic Local Alignment Search Tool", *J. Molecular Biology*, vol. 215, pp. 403-410, 1990.

[44] G. A. Harrison, J. M. Tanner, D. R. Pilbeam and P. T. Baker, "Human Biology: An introduction to human evolution, variation, growth, and adaptability", *Oxford Science Publications*, 1988.

[45] E. Chow, T. Hunkapiller, J. Peterson and M. S. Waterman, "Biological Information Signal Processor", *Proc. Application-Specific Systems, Architectures, and Processors*, pp. 144-160, 1991.

[46] Agility Plc, "The Handel-C Language Reference Manual", available at http://www.agilityds.com, Jan. 2008.

[47] S. Y. Kung, "VLSI Array Processors", *Prentice-Hall*, 1988.

[48] D. I. Moldovan and J. A. B. Fortes, "Partitioning and mapping of algorithms into fixed size systolic arrays", *IEEE Trans. Computers*, vol. 35, no. 1, pp. 1-12, January, 1986.

[49] B. Boeckmann, A. Bairoch, R. Apweiler, M. C. Blatter, A. Estreicher, E. Gasteiger, M. J. Martin, K. Michoud, C. O'Donovan, I. Phan, S. Pilbout and M. Schneider, "The SWISS-PROT protein knowledgebase and its supplement TrEMBL", *Nucleic Acids Research*, vol. 31, pp. 365-370, 2003.

[50] Celoxica Plc, "RCHTX FPGA Board Reference Manual", available at http://www.celoxica.com, March 2008.

[51] E. Sotiriades and A. Dollas, "A General Reconfigurable Architecture for the BLAST Algorithm", *J. VLSI Signal Processing*, vol. 48, pp. 189–208, 2007.

[52] Y. K. Yu, J. C. Wootton and S. F. Altschul, "The compositional adjustment of amino acid substitution matrices", *PNAS*, vol. 100, no 26, pp. 15688-15693, December, 2003.

[53] S. Kasap, K. Benkrid and Y. Liu, "High performance FPGA-based core for BLAST sequence alignment with the two-hit method", *Proc. IEEE Int. Conf. Bioinformatics and Bioengineering*, Oct. 2008.

[54] S. Kasap, K. Benkrid and Y. Liu, "Design and Implementation of an FPGA-based Core for Gapped BLAST Sequence Alignment with the Two-Hit Method", *IAENG Engineering Letters*, vol. 16, no. 3, pp. 443-452, Aug. 2008.

[55] M. C. Herbordt, J. Model, B. Sukhwani, Y. Gu and T. V. Court, "Single pass streaming BLAST on FPGAs", *Parallel Computing*, vol. 33, no. 10-11, pp. 741-756, 2007.

[56] A. Jacob, J. Lancaster, J. Buhler, B. Harris and R. D. Chamberlain, "Mercury BLASTP: Accelerating Protein Sequence Alignment", *ACM Trans. Reconfigurable Technology and Systems*, vol. 1, no. 2, pp. 1-44, June 2008.

[57] S. Datta, P. Beeraka and R. Sass, "RCBLASTn: Implementation and Evaluation of the BLASTn Scan Function", *Proc. IEEE Int. Symp. Field-Programmable Custom Computing Machines*, pp. 88-95, April 5-7 2009.

[58] National Centre for Biotechnology Information (NCBI), available at http://www.ncbi.nlm.nih.gov/, Sep. 2010.

[59] J. D. Bakos and P. E. Elenis, "Special-Purpose Architecture for Solving the Breakpoint Median Problem", *IEEE Trans. VLSI Systems*, vol. 16, no. 12, pp. 1666-1676, 2008.

[60] J. D. Bakos, P. E. Elenis and J. Tang, "FPGA Acceleration of Phylogeny Reconstruction for Whole Genome Data", *Proc. IEEE Int. Conf. Bioinformatics and Bioengineering*, pp. 888-895, Oct. 2007.

[61] J. D. Bakos, "FPGA Acceleration of Gene Rearrangement Analysis", *Proc. IEEE Int. Symp. Field-Programmable Custom Computing Machines*, pp. 85-94, April 2007.

[62] T. S. T. Mak and K. P. Lam, "Embedded Computation of Maximum-Likelihood Phylogeny Inference Using Platform FPGA", *Proc. IEEE Computational Systems Bioinformatics Conf.*, pp. 512-514, Aug. 2004.

[63] T. S. T. Mak and K. P. Lam, "FPGA-based Computation for Maximum Likelihood Phylogenetic Tree Evaluation", *Proc. Field-Programmable Logic and Applications Conf.*, 2004.

[64] T. S. T. Mak and K. P. Lam, "High Speed GAML-based Phylogenetic Tree Reconstruction Using HW/SW Codesign", *Proc. IEEE Computational Systems Bioinformatics Conf.*, pp. 470-473, Aug. 2003.

[65] J. P. Davis, S. Akella and P. H. Waddell, "Accelerating phylogenetics computing on the desktop: Experiments with executing UPGMA in programmable logic", *Proc. IEEE Conf. Engineering in Medicine and Biology Society*, pp. 2864-2868, Sept. 2004.

[66] N. Alachiotis, E. Sotiriades, A. Dollas and A. Stamatakis, "A Reconfigurable Architecture for the Phylogenetic Likelihood Function", *Proc. IEEE Conf. Field Programmable Logic and Applications*, pp. 674-678, Aug. 2009.

[67] N. Alachiotis, E. Sotiriades, A. Dollas and A. Stamatakis, "Exploring FPGAs for Accelerating the Phylogenetic Likelihood Function", *Proc. IEEE Workshop High Performance Computational Biology*, pp. 1-8, May 2009.

[68] W. M. Fitch, "Toward defining the course of evolution: Minimum change for a specific tree topology", *Systematic Zoology*, vol. 20, pp. 406-416, 1971.

[69] J. Felsentein, "Evolutionary trees from DNA sequences: a maximum likelihood approach", *J. Molecular Evolution*, vol. 17, pp. 368-376, 1981.

[70] W. M. Fitch and E. Margoliash, "Construction of phylogenetic trees", *Science*, vol. 155, pp. 279-284, 1967.

[71] J. S. Farris, "Estimating phylogenetic trees from distance matrices", *American Nature*, vol. 106, pp. 645-668, 1970.

[72] N. Saitou and N. Nei, "The neighbour-joining method: A new method for reconstructing phylogenetic trees", *Molecular Biology & Evolution*, vol. 4, pp. 406-425, 1987.

[73] D. Sankoff and P. Rousseau, "Locating the vertices of a Steiner tree in an arbitrary metric space", *Mathematical Programming*, vol. 9, pp. 240-246, 1975.

[74] D. L. Swofford, "PAUP*: Phylogenetic analysis using parsimony (* and other methods)", *Sinauer Associates Inc.*, 2002.

[75] Download website for PAUP, available at http://paup.csit.fsu.edu/downl.html, Aug. 2008.

[76] Xilinx Inc., San Jose, CA, "Virtex-4 datasheets", available at http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/ind

ex.htm, May 2007.

[77] Argonne National Lab, Argonne, IL, "MPI manual", available at http://www.unix.mcs.anl.gov/mpi/www/www3/MPI_W time.html, Jan. 2009.

[78] N. R. Taylor and M. Itzstein, "A structural and energetics analysis of the binding of a series of N-acetylneuraminic-acid-based inhibitors to influenza virus sialidase", *J. Computer-Aided Molecular Design*, vol. 10, no. 3, pp. 233-246, June 1996.

[79] D. C. Rapaport, "The Art of Molecular Dynamics Simulation", *Cambridge University Press*, New York, 2004.

[80] P. P. Ewald, "Evaluation of optical and electrostatic lattice potentials", *Annals of Physics Leipzig*, vol. 64, pp. 253-287, 1921.

[81] L. Verlet, "Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules", *Physical Review*, vol. 159, no. 1, pp. 98-103, July 1967.

[82] D. Beeman, "Some Multistep Methods for Use in Molecular Dynamics Calculations", *J. Computational Physics*, vol. 20, pp. 130-139, Feb. 1976.

[83] M.E. Tuckerman, G.J. Martyna and B.J. Berne, "Reversible multiple time-scale molecular dynamics", *J. Chemical Physics*, vol. 97, no. 3, pp. 1990-2001, March 1992.

[84] D. V. D. Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark and H. J. Berendsen, "GROMACS: Fast, flexible, and free", *J. Computational Chemistry*, vol. 26, no. 16, pp. 1701-1718, Oct. 2005.

[85] GROMACS-4.0.7, "Download website for GROMACS 4.0.7", available at http://www.gromacs.org, Dec. 2009.

[86] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale and K. Schulten, "Scalable molecular dynamics with NAMD", *J. Computational Chemistry*, vol. 26., no. 16, pp. 1781-1802, Oct. 2005.

[87] NAMD-2.7b2, "Download website for NAMD 2.7b2", available at

http://www.ks.uiuc.edu/Research/namd, Nov. 2009.

[88] LAMMPS, "Download website for LAMMPS", available at http://lammps.sandia.gov, Jan. 2010.

[89] F. Toshiyuki, T. Makoto, M. Junichiro, E. Toshikazu and S. Duiichiro, "A Highly Parallelized Special-Purpose Computer for Many-Body Simulations with an Arbitrary Central Force: MD-GRAPE", *Astrophysical J.*, vol. 468, pp. 51-61, Sep. 1996.

[90] Y. Komeiji, M. Uebayasi, R. Takata, A. Shimizu, K. Itsukashi and M. Taiji, "Fast and Accurate Molecular Dynamics Sumulation of a Protein Using a Special-Purpose Computer", *J. Computational Chemistry*, vol. 18, no. 12, pp. 1546-1563, Sep. 1997.

[91] S. Toyoda, H. Miyagawa, K. Kitamura, T. Amisaki, E. Hashimoto, H. Ikeda, A. Kusumi and N. Miyakawa, "Development of MD Engine: High-Speed Accelerator with Parallel Processor Design for Molecular Dynamics Simulations", *J. Computational Chemistry*, vol. 20, no.2, pp. 185-199, 1999.

[92] C. Wolinski, F. Trouw and M. B. Gokhale, "A preliminary study of molecular dynamics on reconfigurable computers", *Proc. Int. Conf. Engineering Reconfigurable Systems and Algorithms*, pp. 304-307, June 2003.

[93] R. Scrofano and V. K. Prasanna, "Computing Lennard-Jones potentials and forces with reconfigurable hardware", *Proc. Int. Conf. Engineering Reconfigurable Systems and Algorithms*, June 2004.

[94] R. Scrofano, M. B. Gokhale, F. Trouw and V. K. Prasanna, "Accelerating Molecular Dynamics Simulations with Reconfigurable Computers", *IEEE Trans. on Parallel and Distributed Systems*, vol. 19, no. 6, pp. 764-778, June 2008.

[95] Y. Gu, T. VanCourt and M. C. Herbordt, "Improved interpolation and system integration for FPGA-based molecular dynamics simulations", *Proc. Int. Conf. Field Programmable Logic and Applications*, pp. 21-28, Aug. 2006.

[96] Y. Gu, T. VanCourt and M. C. Herbordt, "Explicit design of FPGA-based coprocessors for short-range force computations in molecular dynamics

simulations", *Elsevier Parallel Computing*, vol. 34, no. 4, pp. 261-277, May 2008.

[97] N. Azizi, I. Kuon, A. Egier, A. Darabiha and P. Chow, "Reconfigurable Molecular Dynamics Simulator", *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*, pp. 197-206, Apr. 2004.

[98] Y. Gu, T. VanCourt and M. C. Herbordt. "Accelerating molecular dynamics simulations with configurable circuits", *Proc. Int. Conf. Field Programmable Logic and Applications*, pp. 475-480, Aug. 2005.

[99] LAMMPS, "LAMMPS manual", available at http://lammps.sandia.gov/doc/Manual.html, Jan. 2010.

[100] L. Greengard and V. Rokhlin, "A Fast Algorithm for Particle Simulations", *J. Computational Physics*, vol. 73, no.2, pp. 325-348, Dec. 1987.

[101] H. Q. Ding, N. Karasawa and W. A. Goddard, "Atomic level simulations on a million particles: The cell multipole method for Coulomb and London nonbond interactions", *J. Chemical Physics*, vol. 97, no. 6, pp. 4309-4315, Sep. 1992.

[102] R. W. Hockney and J. W. Eastwood, "Computer Simulation Using Particles", *Adam Hilger*, 1988.

[103] T. Darden, D. York and L. Pedersen, "Particle mesh Ewald: An N·log(N) method for Ewald sums in large systems", *J. Chemical Physics*, vol. 98, no. 12, pp. 10089-10092, June 1993.

[104] S. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics", *J. Computational Physics*, vol. 117, no. 1, pp. 1-19, Mar. 1995.

[105] S. J. Plimpton, R. Pollock and M. Stevens, "Particle-Mesh Ewald and rRESPA for Parallel Molecular Dynamics Simulations", *Proc. SIAM Conf. Parallel Processing for Scientific Computing*, Mar. 1997.

[106] E. L. Pollock and J. Glosli, "Comments on P3M, FMM, and the Ewald method for large periodic Coulombic systems", *Computer Physics Communications*, vol. 95, no. 2, pp. 93-110, June 1996.

[107] OpenCores website, "Floating Point Adder and Multiplier", available at

http://opencores.org/project,fpuvhdl, Nov. 2009.

[108] G. Marcus, P. Hinojosa, A. Avila and J. N. Flores, "A Fully Synthesizable Single-Precision, Floating-Point Adder/Subtractor and Multiplier in VHDL for General and Educational Use", *Proc. IEEE Int. Caracas Conf. Devices, Circuits and Systems*, pp. 319-323, Nov. 2004.

[109] T. Amisaki, T. Fujiwara, A. Kusumi, H. Miyagawa and K. Kitamura, "Error evaluation in the design of a special-purpose processor that calculates nonbonded forces in molecular dynamics simulations", *J. Computational Chemistry*, vol. 16, no. 9, pp. 1120-1130, Sep. 1995.

[110] M. Chiu and M.C. Herbordt, "Efficient particle-pair filtering for acceleration of molecular dynamics simulation", *Proc. Int. Conf. Field Programmable Logic and Applications*, pp. 345-352, Aug. 2009.

Appendix A

Journal Publications

1. **S. Kasap**, K. Benkrid and Y. Liu, "Design and Implementation of an FPGA-based Core for Gapped BLAST Sequence Alignment with the Two-Hit Method", *IAENG Engineering Letters*, vol. 16, no. 3, pp. 443-452, August 2008.
2. **S. Kasap** and K. Benkrid, "High Performance Phylogenetic Analysis with Maximum Parsimony on Reconfigurable Hardware", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Digital Object Identifier: 10.1109/TVLSI.2009.2039588, accepted November 2009, *expected to be published by the end of 2010.*
3. **S. Kasap** and K. Benkrid, "Parallel Processor Design and Implementation for Molecular Dynamics Simulations on a FPGA Parallel Computer", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, submitted July 2010, *under review.*

Design and Implementation of an FPGA-based Core for Gapped BLAST Sequence Alignment with the Two-Hit Method

Server Kasap, Khaled Benkrid and Ying Liu
The University of Edinburgh, School of Electronics and Engineering,
Mayfield Road, Edinburgh EH9 3JL, Scotland, UK
(s.kasap,k.benkrid,y.liu)@ed.ac.uk

Abstract--**This paper presents the design and implementation of the first FPGA-based core for Gapped BLAST sequence alignment with the two-hit method, ever reported in the literature. Gapped BLAST with two hit is a heuristic biological sequence alignment algorithm which is very widely used in the Bioinformatics and Computational Biology world. The architecture of the core is parameterized in terms of sequence lengths, match scores, gap penalties and cut-off, and threshold values. It is composed of various blocks each of which performs one step of the algorithm in parallel. This results in high performance and efficient FPGA implementations, which easily outperform equivalent software implementations by one order of magnitude or more. Furthermore, the core was captured in an FPGA-platform-independent language, namely the Handel-C language, to which no specific resource inference or placement constraints were applied. Hence, the core can be ported to different FPGA families and architectures.**

Index Terms—**FPGA, Gapped BLAST, Sequence Alignment, Two-Hit method**

I. Introduction

Biological sequence alignment is a widespread operation in the world of Bioinformatics and Computational biology (BCB) where sequence databases are searched to find similar sequences to a query sequence, by aligning each subject sequence in the database to the query sequence [1]. The main aim of this operation is to obtain information about a newly discovered biological sequence (i.e. Protein, DNA or RNA) from other known sequences (stored in the database). For instance, if a new sequence is similar to a known sequence representing a cancerous gene, then information pertained to the functionality of the new sequence can be inferred, which is useful in early disease diagnosis and drug engineering. Besides this, study of evolutionary development and history of species can be done through biological sequence alignment [1] [2].

Biological sequence alignment is a computationally intensive operation and with exponentially growing sequence databases (see Figure 1) this task cannot be achieved by desktop computer systems within realistic execution times. Hence, there is a need for faster computing platforms to cope with this growth. Recently, Field Programmable Gate Arrays (FPGAs) have been proposed as a high performance reconfigurable hardware platform for sequence alignment algorithms [3] [4] [5]. Indeed, FPGAs are capable of providing high speed-ups compared to general purpose processors with the convenience of reprogrammability, which makes them an attractive platform to accelerate biocomputing applications.

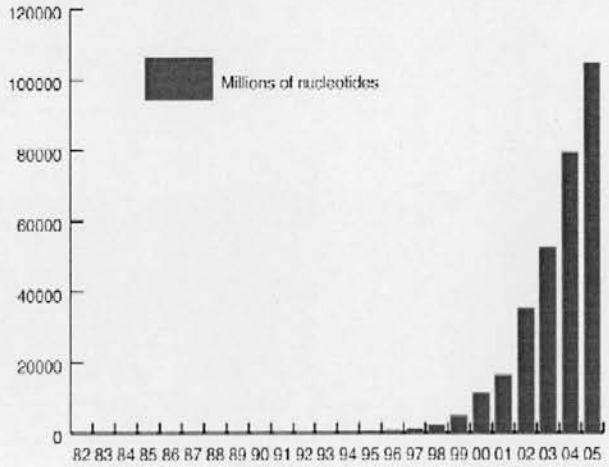


Figure 1. Exponential growth of biological sequence databases over years [1]

There are various biological sequence alignment algorithms some of which are exhaustive and give optimal alignments (e.g. Needleman-Wunsch [6], Smith-Waterman [7]) and some of which are heuristic and give sub-optimal alignments (e.g. FASTA[8], BLAST[9]). In this paper, we concentrate on Basic Local Alignment Search Tool (BLAST) which is a local alignment algorithm. Although it is heuristic, in the sense that it produces local alignments which are not always optimal, it is much faster than ordinary exhaustive dynamic programming algorithms. The design and implementation of a variant of BLAST, namely Gapped BLAST with two-hit method [10], is presented in this paper. The design is captured in the Handel-C language [13] which is a FPGA-platform-independent language, making our design portable across a number of FPGA architectures (e.g. Xilinx, Altera). The remainder of this paper will first present essential background information on the general BLAST algorithm. Then, the design and implementation of our FPGA core for Gapped BLAST with the two-hit method will be detailed. After that, comparative timing performance evaluation of our core against equivalent desktop software is presented. Finally, conclusions are laid out with plans for future work.

II. Background

Biological sequences evolve through mutation, selection and random genetic drift [11]. Mutation, in particular manifests itself through 3 main processes which are as follows:

- Substitution of residues: Residue A in the sequence is substituted by another residue B.
- Insertion of residues: New residues are inserted into the sequence.
- Deletion of residues: Existing residues in the sequence are deleted.

Insertions and deletions result in *gaps* which are taken into consideration when aligning biological sequences. The degree of alignment of biological sequences is measured by a score which is obtained by the summation of score terms of each aligned pair of residues with possible gap penalty terms. Score terms for each aligned residue pair are obtained from probabilistic models which are stored in score or substitution matrices such as BLOSUM50 [1]. The latter is a 20x20 matrix for protein sequence residues. On the other hand, gap penalties depend on the length of the gap and are independent of gap residues. There are two main types of gap penalties:

- Linear gap penalty: The cost of a gap of length g is given by following linear function:

$$\text{Penalty}(g) = -g*d$$

- Affine gap penalty: A constant penalty is given for opening a new gap while a linear and smaller penalty is given for subsequent gap extensions. The cost function of the affine gap penalty is hence given by the following affine equation:

$$\text{Penalty}(g) = -d-(g-1)*e$$

BLAST stands for Basic Local Alignment Tool. It is developed on the ideas of FASTA.BLAST is used for searching both protein and DNA sequence databases for sequence similarities. It is a heuristic local alignment algorithm which approximates the dynamic programming Smith-Waterman algorithm. Since it is a heuristic algorithm, the local alignment it produces is not always optimal. However, it is much faster than the Smith-Waterman algorithm. As a result, BLAST and its variants are some of the most widely used sequence search tools.

The central idea of the BLAST algorithm is that a statistically significant alignment is likely to contain a high-scoring pair of aligned words. BLAST first finds these high scoring pairs of aligned words and then extends them to the real alignment. These words are k-residues long where k is different for DNA and protein sequences. The default k values for DNA and protein sequences are 11 and 3 respectively. There are 3 basic steps of BLAST:

- Pre-processing the query sequence: All k-long words in the query sequence are extracted. Then, words that are similar to these are found. We call the overall results the k-words.
- Scanning the subject sequences: All the subject sequences in the database are scanned one by one for matches with the obtained k-words.
- Extension of the matches: All matches in the subject sequences are extended to form local alignments between the query sequence and related subject sequences in the database.

In subsections II.A-II.C, all basic steps of the BLAST algorithm mentioned above will be explained in more detail.

It is worth mentioning at this stage that the aforementioned basic steps belong to the original BLAST algorithm. However, several variants of the original algorithm have been devised over the years with the aim of increasing its sensitivity while keeping run-times at minimum. All of these variants include the 3 basic steps of the original algorithm, with the addition of new steps. In this paper, we discuss two of these variants, namely BLAST with two-hit method, and Gapped BLAST, which are in subsections II.D and II.E respectively.

A. Step 1: Pre-processing the Query Sequence

An example protein sequence which has 9 residues (or amino acids) is shown below:

LVNRKPVVP

In this first step, we take the query sequence and chop it into overlapping k-words as illustrated below for the query sequence shown above, with k = 3:

Word 0: LVN

Word 1: VNR

Word 2: NRK

Word 3: RKP

Word 4: KPV

Word 5: PVV

Word 6: VVP

As it can be seen, there are 7 words extracted from the query sequence which are 3 residues long. In general, the number of words extracted equals (m-k) + 1 where m is the number of residues in the query sequence. After this, words similar to each of these extracted words are found through the usage of specific scoring matrix. An example scoring matrix for protein residues (Blosum50) is shown below in figure 2.

	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W
C	**9**	-1	-1	-3	0	-3	-3	-3	-4	-3	-3	-3	-3	-1	-1	-1	-1	-2	-2	-2
S	-1	**4**	1	-1	1	0	1	0	0	0	-1	-1	0	-1	-2	-2	-2	-2	-2	-3
T	-1	1	**4**	1	-1	1	0	1	0	0	0	-1	0	-1	-2	-2	-2	-2	-2	-3
P	-3	-1	1	**7**	-1	-2	-1	-1	-1	-1	-2	-2	-1	-2	-3	-3	-2	-4	-3	-4
A	0	1	-1	-1	**4**	0	-1	-2	-1	-1	-2	-1	-1	-1	-1	-1	-2	-2	-2	-3
G	-3	0	1	-2	0	**6**	-2	-1	-2	-2	-2	-2	-2	-3	-4	-4	0	-3	-3	-2
N	-3	1	0	-2	-2	0	**6**	1	0	0	-1	0	0	-2	-3	-3	-3	-3	-2	-4
D	-3	0	1	-1	-2	-1	1	**6**	2	0	-1	-2	-1	-3	-3	-4	-3	-3	-3	-4
E	-4	0	0	-1	-1	-2	0	2	**5**	2	0	0	1	-2	-3	-3	-3	-3	-2	-3
Q	-3	0	0	-1	-1	-2	0	0	2	**5**	0	1	1	0	-3	-2	-2	-3	-1	-2
H	-3	-1	0	-2	-2	-2	1	1	0	0	**8**	0	-1	-2	-3	-3	-2	-1	2	-2
R	-3	-1	-1	-2	-1	-2	0	-2	0	1	0	**5**	2	-1	-3	-2	-3	-3	-2	-3
K	-3	0	0	-1	-1	-2	0	-1	1	1	-1	2	**5**	-1	-3	-2	-3	-3	-2	-3
M	-1	-1	-1	-2	-1	-3	-2	-3	-2	0	-2	-1	-1	**5**	1	2	-2	0	-1	-1
I	-1	-2	-2	-3	-1	-4	-3	-3	-3	-3	-3	-3	-3	1	**4**	2	1	0	-1	-3
L	-1	-2	-2	-3	-1	-4	-3	-4	-3	-2	-3	-2	-2	2	2	**4**	3	0	-1	-2
V	-1	-2	-2	-2	0	-3	-3	-3	-2	-2	-3	-3	-2	1	3	1	**4**	-1	-1	-3
F	-2	-2	-2	-4	-2	-3	-3	-3	-3	-3	-1	-3	-3	0	0	0	-1	**6**	3	1
Y	-2	-2	-2	-3	-2	-3	-2	-3	-2	-1	2	-2	-2	-1	-1	-1	-1	3	**7**	2
W	-2	-3	-3	-4	-3	-2	-4	-4	-3	-2	-2	-3	-3	-1	-3	-2	-3	1	2	**11**

Figure 2. The Blosum50 scoring matrix

Words which score at least threshold value T with the scoring matrix when aligned with the words extracted from the query sequence are regarded to be similar to these extracted words. Similar words for each extracted word are found and then recorded with the location address of the corresponding extracted word in the query sequence tagged to them. This process is illustrated below with the first extracted word shown above (i.e. LVN) using the Blosum50 scoring matrix for the case where T is 12:

Word 0: L V N
4 + 4 + 6 = 14
Query word 1: L V N

Word 0: L V N
2 + 4 + 6 = 12
Query word 2: M V N

Word 0: L V N
4 + 4 + 1 = 9
Query word 3: L V S

Query word 1 and query word 2 score 14 and 12 respectively when aligned with the first extracted word (LVN) from the query sequence. Since score values are over or equal to 12, query word 1 and query word 2 are recorded with the location address of the first extracted word in the query sequence, which is 0. However, query word 3 is discarded since it scores less than 12 when aligned with the extracted word. All recorded similar words are used in step 2 of the BLAST algorithm.

B. Step 2: Scanning the subject sequences

In this step, all subject sequences in the database are scanned one by one to find the possible exact matches of the query words which were recorded in step 1. Each match is referred to as hit or hotspot. Each hit is recorded in a list for the third step of the BLAST algorithm with the identity of the corresponding query word and the location address where the hit occurred in the subject sequence. Considering the fact that current databases contains tens of thousands of subject sequences and that each subject sequence comprises hundreds/thousands of residues, it is obvious that this sequence database scanning process is a massively time consuming task.

C. Step 3: Extension of the matches

In this last step of the basic BLAST algorithm, we utilize the list of matches (hits) obtained in step 2 to form local alignments between the query sequence and the subject sequences in the database. Each entry in the list of hits contains the location address of a match in the subject sequence and the location address of the corresponding query word in the query sequence. Starting from these 2 location addresses, each of the hits in the list is extended on the query and corresponding subject sequence in both directions without allowing any gaps. In this extension, pairs of residues along the query and subject sequence are scored with a scoring matrix (e.g. Blasoum50). This process is illustrated in figure 3 with the following subject sequence:

GVCRRPLKC

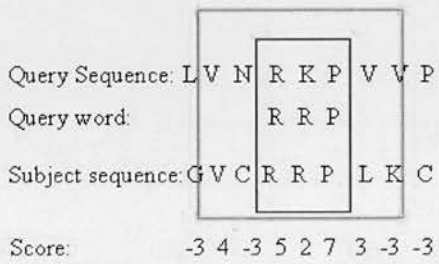


Figure 3. Step 3: Extension of matches

In figure 3, the red box shows a hit where query word RRP is matched in the subject sequence. The query word RRP is similar to RKP word in the query sequence. The green box in figure 3 shows the extension which started from the edges of the red box. As the extension proceeds in a 1 residue pair at a time in both directions and without allowing for any gaps, pairs of residues along the extension are scored using a scoring matrix (BLOSUM50 in our case). These score terms are added up after each extension step and the extension is terminated when this total score falls a certain cut-off distance below the best total score obtained so far. Then, the extension goes back to its state which yielded the highest total score. As a result of this extension step, the related subject sequence is locally aligned to the query sequence (without gaps).

D. BLAST with two-hit method

The third step of the BLAST algorithm, i.e. the extension of the matches on the query and subject sequences, generally accounts for a very high percentage of the BLAST algorithm's execution time. Hence, the two-hit method was devised to reduce the

time spent in this extension step. The central idea of the two-hit method is to start extension only when there are two non-overlapping hits on the same diagonal within distance A of each other. This is illustrated in figure 4 where only two non-overlapping hits on the same diagonal line which are close enough to each other are extended.

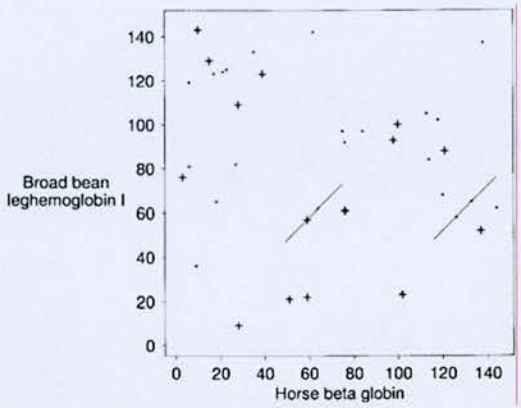


Figure 4. Ungapped extension of two close hits on the sane diagonal lines [10]

In other words, if the distance between any two non-overlapping hits on the subject sequence is equal to the distance between the locations of the corresponding query words in the query sequence, then ungapped extension is triggered in both directions starting from both hits. The rest of the process is the same as explained in subsection II.C and the result is a local ungapped alignment of the query and subject sequences. This process is illustrated in figure 5 where A is equal to 5.

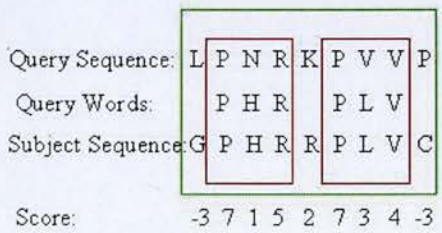


Figure 5. Extension with the two-hit method

In figure 5, the red boxes show two non-overlapping hits on the query and subject sequences within a distance of 4. Since the distance between the query words in the query sequence is equal to the distance between the two hits on the subject sequence, and since this distance between the two hits is less than 5, and bigger than 2, ungapped extension is started from the edges of the left and right hand sides of the red boxes respectively (see the green box in figure 5).

To maintain the sensitivity of the general algorithm, the threshold value T used in the query pre-processing step of the algorithm is reduced. Hence, the number of query words recorded in this step will increase. As a result, while scanning the subject sequences in step 2 we will potentially find more hits than before. However, only a small fraction of these hits will have an associated second hit. Therefore, ungapped extension will be triggered less frequently compared to the case in the original BLAST algorithm. The total execution time of BLAST is thus reduced.

E. Gapped BLAST

Gapped BLAST is an advancement of BLAST with the two-hit method, which is faster and gives better alignments and alignment scores. In addition to the steps outlined above, gapped alignment is triggered in gapped BLAST if local ungapped alignment obtained as a result of ungapped extension has a sufficiently high score. If this is the case, the central pair of the local ungapped alignment is used as a seed from which the gapped alignment is run both backwards and forwards, as illustrated in figure 6. The gapped alignment algorithm utilized in Gapped BLAST is a modified version of the Needleman-Wunsch algorithm where the alignment is pruned when alignment scores fall a certain cut-off distance below the best score so far. The Needleman-Wunsch algorithm with linear and affine gap models is explained in subsections II.F and II.G below, respectively. The necessary modifications of the original Needleman-Wunsch algorithm needed in the Gapped BLAST algorithm are explained in subsection II.H.

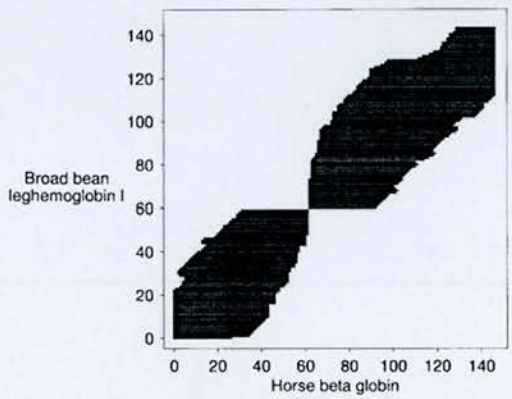


Figure 6. Gapped alignment started from the central pair of the local ungapped alignments in both directions [10]

F. The Needleman-Wunsch algorithm with the Linear Gap Model

The Needleman-Wunsch algorithm is a dynamic programming algorithm which finds optimal global gapped alignment between two sequences [6]. In Gapped BLAST, however, it is used for local alignment purposes, after a slight modification as will be explained in subsection II.H below. In this section, we will present the original Needleman-Wunsch algorithm where a linear gap model is assumed.

Assuming we have two sequences $X = x_1x_2....x_M$ and $Y = y_1y_2.....y_N$, whose lengths are M and N respectively, a dynamic programming score matrix F is built where each cell F (i, j) represents the best alignment between $x_1x_2....x_i$ segment of X and $y_1y_2.....y_j$ segment of Y.

The boundary cells of Matrix F are set by the following set of equations:

$$F(0, 0) = 0 \quad (1)$$

$$F(i, 0) = -i*d \text{ where } i=1, 2....M \quad (2)$$

$$F(0, j) = -j*d \text{ where } j=1, 2....N \quad (3)$$

The following equation is used to compute the values of each of the remaining cells of matrix F:

$$F(i,j)=max\begin{Bmatrix} F(i-1,j-1)+s(x_i,y_j) \\ F(i-1,j)-d \\ F(i,j-1)-d \end{Bmatrix} \quad (4)$$

Here, we aim to find best alignment between $x_1x_2....x_i$ and $y_1y_2.....y_j$ given the best alignment between $x_1x_2....x_{i-1}$ and $y_1y_2.....y_{j-1}$ (i.e. F(i-1, j-1)), between $x_1x_2....x_{i-1}$ and $y_1y_2.....y_j$ (i.e. F(i-1, j)) and between $x_1x_2....x_i$ and $y_1y_2.....y_{j-1}$ (i.e. F(i, j-1)). There are three alternatives:

- An alignment between x_i and y_j: In this case, the new score F(i, j) is F(i-1, j-1) + $s(x_i, y_j)$ where $s(x_i, y_j)$ is the scoring matrix score for x_i and y_j.
- An alignment between x_i and a gap in Y: In this case, the new score F(i, j) is F (i-1, j)-d where d is the gap penalty.
- An alignment between a gap in X and y_j: In this case, the new score F(i, j) is F (i, j-1)-d where d is the gap penalty.

One of these three alternatives (see figure 7) yields the largest score and is the best alignment between $x_1x_2....x_i$ and $y_1y_2.....y_j$.

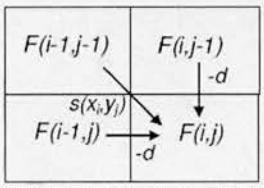


Figure 7. Illustration of the Needleman-Wunsch dynamic programming equations

Note that a pointer to the cell from which F (i, j) was derived (i.e. above, left, above-left) is stored in each cell. Once the value of the last cell of matrix F (i.e. F (M, N)) is computed, the best global alignment between X and Y is obtained by tracking back from this cell, using the aforementioned pointers, and applying the following procedure:

- If cell (i, j) was derived from cell (i-1, j-1), the pair of symbols x_i and y_j is added to the front of the current alignment.
- If cell (i, j) was derived from cell (i-1, j), x_i and a gap in Y are added to the front of the current alignment.
- If cell (i, j) was derived from cell (i, j-1), a gap in X and y_j are added to the front of the current alignment.

This is illustrated in figure 8 for 2 protein sequences. In this figure, the trace-back starts from F(M, N) = F(7, 10) and moves backward to the cell from which the current cell was derived until F(0, 0) is reached, while applying the aforementioned procedure at every step of the trace-back. The resulting global alignment of these 2 sequences can be seen at the bottom of figure 8.

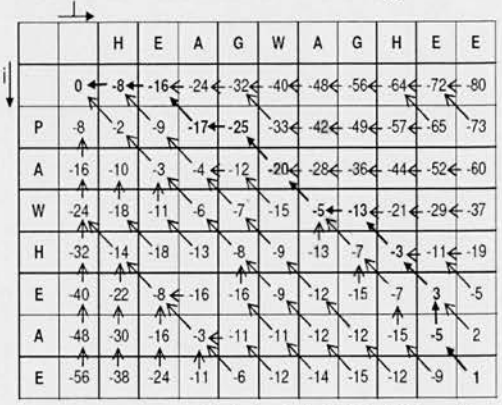


Figure 8. Illustration of the Needleman-Wunsch algorithm

G. The Needleman-Wunsch algorithm with the Affine Gap Model

The Needleman-Wunsch algorithm with the affine gap model is similar to the one with the linear gap model. However, in this case, we have three new matrixes namely I_z, I_x and I_Y to compute. The following equations are used to compute the values of I_z, I_x and I_Y where d is the penalty associated with the gap opening and e is penalty associated with the gap extension:

$$I_z(i,j)=max\begin{Bmatrix} I_z(i-1,j-1)+s(x_i,y_j) \\ I_x(i-1,j-1)+s(x_i,y_j) \\ I_y(i-1,j-1)+s(x_i,y_j) \end{Bmatrix} \quad (5)$$

$$I_x(i,j)=max\begin{Bmatrix} I_z(i-1,j)-d \\ I_x(i-1,j)-e \end{Bmatrix} \quad (6)$$

$$I_y(i,j)=max\begin{Bmatrix} I_z(i,j-1)-d \\ I_y(i,j-1)-e \end{Bmatrix} \quad (7)$$

The values of the dynamic programming matrix cells F(i, j) are equal to the maximum of I_Z *(i, j)*, I_X *(i, j)* and I_Y *(i, j)* as shown in equation 8.

$$F(i,j)=max\begin{Bmatrix} I_z(i,j) \\ I_x(i,j) \\ I_y(i,j) \end{Bmatrix} \quad (8)$$

Note that the pointer to the above-left cell is stored in the cell if *F(i, j)* is set to equal I_Z *(i, j)* whereas the pointer to the left cell is stored if *F(i, j)* is set to equal

$I_x(i, j)$. Finally, the pointer to the above cell is stored if $F(i, j)$ is set to equal $I_Y(i, j)$.

H. Modified Needleman-Wunsch algorithm

The Needleman-Wunsch algorithm presented above is used for finding global gapped alignments between two sequences. Gapped BLAST however requires some modifications to the original Needleman-Wunsch algorithm. First, no computations are done for the dynamic programming matrix cells which are adjacent to cells whose F(i, j) values are a certain cut-off value below the highest cell value computed so far. Second, the trace-back procedure may start at any cell which has the highest value F(i, j) among all the cells, rather than bottom rightmost cell. In this way, we have a local gapped alignment at the end of the trace-back procedure.

III. Hardware Implementation of Gapped BLAST with the Two-Hit Method

Figure 9 shows a hardware architecture which implements gapped BLAST algorithm with the two-hit method. Each block in the architecture implements one step of the algorithm as described in the above sections, except for the pre-processing query sequence step which is implemented by high level application software running on the host computer. The architecture consists of 8 *HitFinderTwoHit* blocks, 2 *UngappedExtender* blocks and 1 *GappedExtender block* all of which are running in parallel. There are also 8 32K x 5 bits subject sequence memories each of which holds a number of subject sequences. Note that each subject sequence memory belongs to one *HitFinderTwoHit* block. Each *HitFinderTwoHit* block is composed of 5 *HitFinder* blocks and 1 *TwoHitMethod* block. Each *HitFinder* block implements step 2 outlined in subsection II.B and scans its assigned subject sequence memory to find exact matches of the query words in the subject sequences. Each *TwoHitMethod* block performs the two-hit method procedure on hits coming from the 5 *HitFinder* blocks which are in the same *HitFinderTwoHit* block as the *TwoHitMethod* block. Besides these, each *UngappedExtender* block implements step 3 mentioned in subsection II.C and extends the two hits found by its 4 allocated *TwoHitMethod* blocks without allowing gaps, in order to obtain local ungapped alignments. Finally, a single *GappedExtender* block implements the modified Needleman-Wunsch algorithm to produce local gapped alignments from local ungapped alignments obtained in 2 *UngappedExtender* blocks.
The high level application software and all of the blocks which constitute the architecture shown in figure 9 are detailed in the following subsections.

A. High Level Application Software

Figure 10 shows the organization of our Gapped BLAST FPGA implementation. Application software running on the host has many duties, the most important of which is the query sequence pre-processing as explained in section II.A. In brief, the application software finds 3 letter long query words which score at least a threshold value T when aligned with words extracted from the query sequence. Then, the location address of each of these query words in the query sequence is placed at a vacant position in an upper word list and a lower word list pair depending on the 2 most significant letters and 2 least significant letters of the query word, respectively. Note that there are 5 upper word and lower word list pairs.

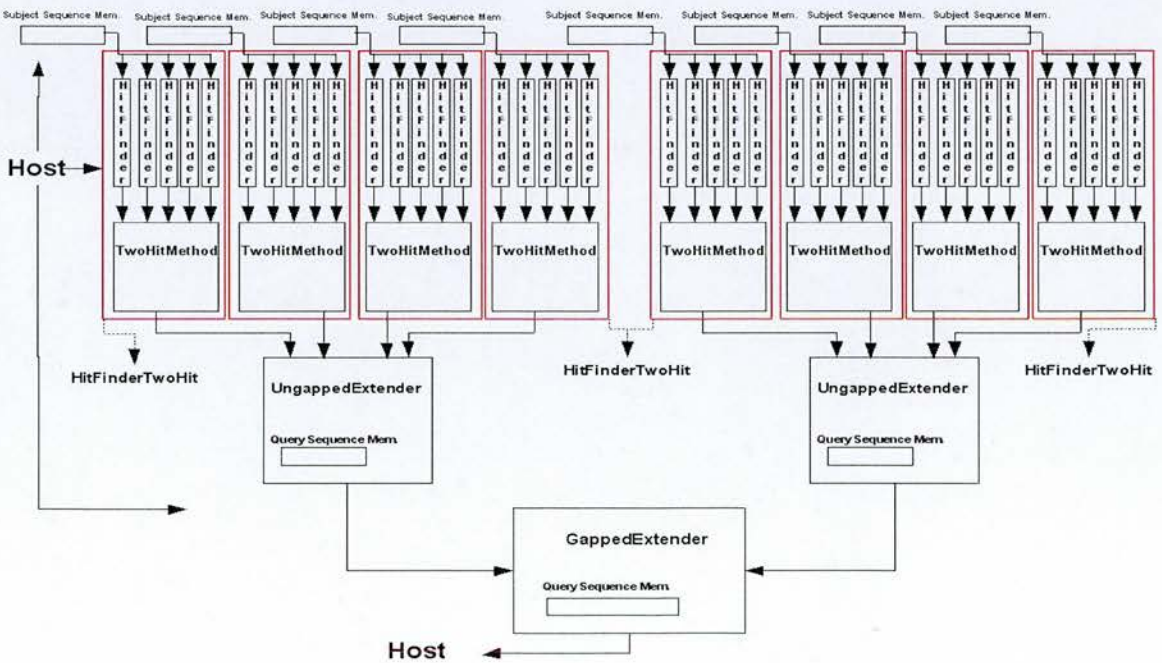


Figure 9. Hardware architecture for the Gapped BLAST algorithm with the two-hit method

As it can be seen in figure 10, there are various FPGA configuration bit files for different threshold and cut-off value parameters. The first task of the application software is to pick the proper bit file, depending on the user-supplied algorithm parameters, from a database of FPGA configurations and load it on to the FPGA chip. Afterwards, the application software runs the hardware configuration in 4 modes. In mode 1, the application software sends one of the 5 upper word and lower word list pairs to each of the 5 *HitFinder* blocks in every *HitFinderTwoHit* block. In mode 2, a number of subject sequences are sent to the 8 available subject sequence memories on FPGA, depending on the subject sequence lengths. In mode 3, the application software sends a query sequence to the FPGA to be stored in memories within the 2 *UngappedExtender* blocks and the single *GappedExtender* block. Finally, the execution of the hardware configuration is launched in mode 4. After some time, the FPGA starts sending the high scoring subject sequences to host with their alignment scores to be printed onto the screen. By repeating these steps several times for different subject sequences, we can align a query sequence to all subject sequences in a sequence database. Note that each iteration is called as "pass".

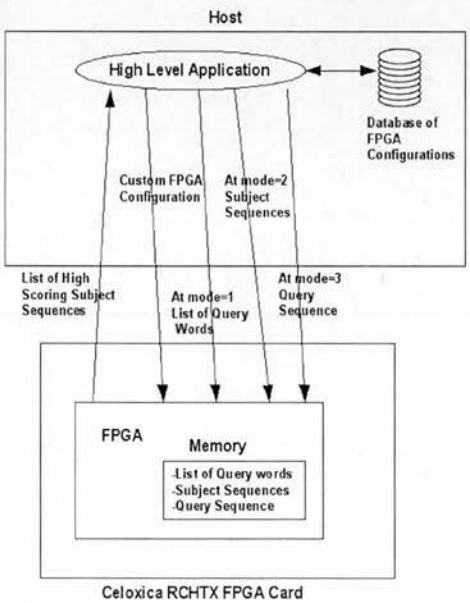


Figure 10. Organization of our Gapped BLAST system

B.

A.

B. HitFinder Block

Figure 11 shows a simplified inner structure of a *Hitfinder* block. The architecture of this block is modified version of one shown in figure 7 of [18].The difference is that as opposed to [18],we added the positions of the query words in the query sequence into the memory content of the Hit Finder to increase the sensitivity of the hit finding process. Furthermore, our design implements the two-hit method (detailed in the next section) which is not the case with [18]. Lastly, our core includes a unit for gapped alignment for the purposes of implementing Gapped BLAST in contrast to [18] which just implements original BLAST.

The major aim of this block is to scan each three letter long word of the subject sequences in order to find exact matches of the query words, as explained in subsection II.B. It is comprised of an upper word list memory, a lower word list memory, a shift register, a FIFO buffer and some control logic. Note that every *Hitfinder* block is assigned to a subject sequence memory whose address register (*Counter*) is unique in the *HitFinderTwoHit* block.

At every clock cycle, 5-bit long residues of a subject sequence are shifted into the shift register (*ShiftReg*) from the assigned subject sequence memory and the address register of the subject sequence memory is incremented by one. The shift register is 15 bits long and hence it can hold 3 subject sequence residues at the same time. At every clock cycle, the 10 most significant bits and the 10 least significant bits of the shift register content are used as addresses for the upper word list memory and the lower word list memory respectively (see figure 11). If the resulting outputs of these memories are valid entries and are equal to each other, this means that a three-letter long word of the subject sequence which is currently held in the shift register matches exactly a query word whose location address in query sequence is given in the outputs of the word list memories. In this case, we have a hit condition which needs to be recorded for the following steps of the algorithm. Hence, we register the address of the query word in the query sequence and the location address of the hit in the subject sequence to a FIFO buffer named *Hit FIFO* with 3 control bits. These entries to *Hit FIFO* are processed by the *TwoHitMethod* block assigned to the *Hitfinder* block (see figure 9).

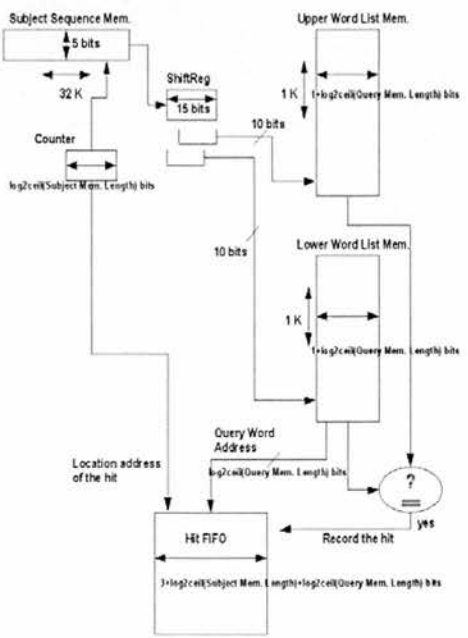


Figure 11. Simplified inner structure of the *Hitfinder* block

C. TwoHitMethod Block

Figure 12 shows a simplified inner structure of the *TwoHitMethod* block. Its aim is to find two non-overlapping hits on the same diagonal within distance A of each other as explained in subsection II.D above. In this architecture, there are two FIFOs of the same length and same width namely *Hit FIFO 1* and *Hit FIFO 2* to which the same hit entries from the *Hit FIFO*s of the 5 *Hitfinder* blocks (which belong to the same *HitFinderTwoHit* block) are stored one by one in turn starting from the *Hit FIFO* in the first *Hitfinder* block. The processing of hit entries commences when there are more than two hit entries in the FIFOs. For instance, the a^{th} hit entry of *Hit FIFO 1* and b^{th} hit entry of *Hit FIFO 2* are taken and the hit addresses of these entries are subtracted from each other. If the result is less than 3, we continue with the processing of the a^{th} hit entry in *Hit FIFO 1* and $(b+1)^{th}$ hit entry in *Hit FIFO 2* in the next clock cycle. On the other hand, if the result is bigger than threshold value A, we continue with the processing of the $(a+1)^{th}$ hit entry in *Hit FIFO 1* and $(a+2)^{th}$ hit entry in *Hit FIFO 2* in the next clock cycle. However, if the result of this subtraction is between 3 and threshold value A inclusive, we subtract the query word addresses in the hit entries. If the second subtraction result is not equal to the first one, this means that the two hits are not on the same diagonal, and hence we continue with the processing of the a^{th} hit entry in *Hit FIFO 1* and $(b+1)^{th}$ hit entry in *Hit FIFO 2* in the next clock cycle. If the two results are the same, however, this means that we have two close enough non-overlapping hits on the same diagonal which need to be recorded for the subsequent steps of the algorithm. The two hit cases are recorded to two FIFOs namely *TwoHit FIFO1* and *TwoHit FIFO* 2. The address of the first hit and the distance between the two hits (Result 2 in figure 12) are stored in *TwoHit FIFO1* with 2 control bits, whereas the address of the first query word is stored in *TwoHit FIFO* 2. These two-hit entries to the *TwoHit FIFOs* are subsequently processed by the assigned *UngappedExtender* block.

D. UngappedExtender Block

The *UngappedExtender* block implements the ungapped extension step of the Gapped BLAST algorithm as explained in subsection II.C above. Each of the two *UngappedExtender* blocks read *Twohit FIFO*s of its 4 assigned *TwoHitMethod* blocks in turn. When the *UngappedExtender* block detects a two-hit entry in the *Twohit FIFOs* of one *TwoHitMethod* block, the hit address of the first hit, the address of the first query word in the query sequence and the distance between the two hits are all extracted from that entry to compute the start (seed) points of the outward ungapped extension in both directions, on both query and related subject sequence. Note that first residue pair of the first hit and the last residue pair of the second hit are the seed points of the outward ungapped extension on the query and related subject sequence,. Afterwards, the inward ungapped extension starts from one start point to the other start point where the residue pairs along the extension are scored against a scoring matrix, with the intermediate scores accumulated. When the inward ungapped extension ends, the outward ungapped extension is launched in both directions. Here again, the residue pairs along the extension are scored, with the intermediate score terms accumulated, and added up with the total score obtained from the inward ungapped extension. The outward ungapped extension terminates either when the currently computed grand total score falls a certain cut-off value below the highest grand total score obtained so far, or when the extension reaches end of the query or subject sequences in either direction. In this case, the ungapped extension retracts to its previous state which yielded the highest grand total score. If this highest grand total score exceeds a certain threshold value, the end points of this high scoring ungapped extension in both directions on both query and subject sequences are registered to two *UngappedResult FIFOs* to be read and processed by the single *GappedExtender* block for the purpose of gapped alignment.

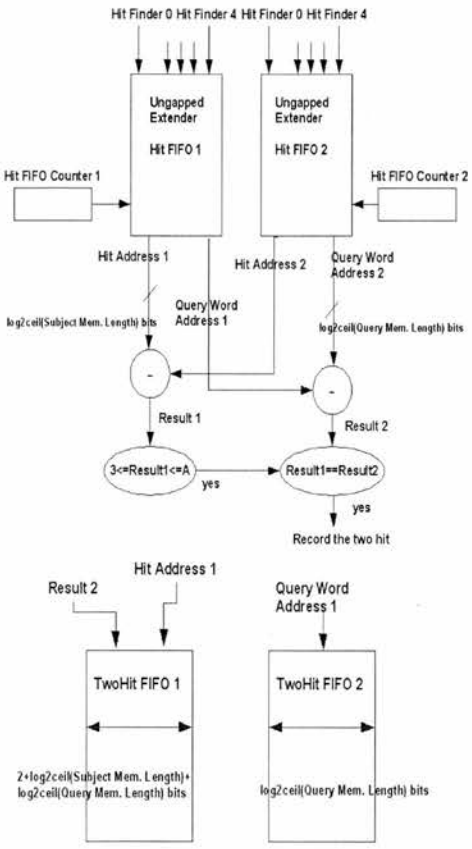


Figure 12. Simplified inner structure of TwoHitMethod block

E. GappedExtender block

The *GappedExtender* block implements the gapped alignment step using the modified Needleman-Wunsch algorithm with the affine gap model. Here, only the gapped alignment score is computed. The final alignment, i.e. with trace-back, is not done on FPGA because of its excessive memory requirement. The *GappedExtender* block reads *UngappedResult FIFOs* of the two *UngappedExtender* blocks in turn to obtain the edge points of the high scoring ungapped alignments produced by these two *UngappedExtender* blocks.

These edge points are used to compute the central residue pair of the ungapped alignment from which the gapped alignment on the query and related subject sequence is launched in both directions. Figure 13 shows one of the two linear systolic arrays in the *GappedExtender* block which run independently in parallel to perform the modified Needleman-Wunsch algorithm on each side of the seed residue pair. This architecture is deducted from the data dependency graph of the Needleman-Wunsch algorithm as presented in section II above [12].

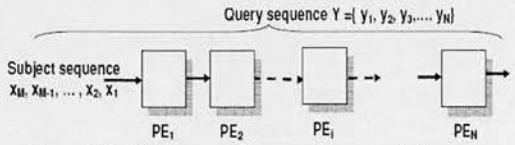


Figure 13. Linear systolic array for gapped alignment

The linear systolic array consists of pipelined basic processing elements (PE) each of which performs the dynamic programming equations presented in subsections II.G above. Before the operation of the array, the query sequence residues at one side of the seed residue pair are shifted through the array. At the end of this shift, each PE holds one query residue. Following this, the subject sequence residues at the same side of the seed residue pair are shifted systolically through the array during which each PE generates value of one dynamic programming matrix cell every clock cycle. However, the direction of the cell from which the current result has been derived is not saved since trace-back will not be performed in hardware. Each PE generates one column of the dynamic programming matrix after M cycles where M is equal to the number of subject sequence residues. However, each PE is one cycle behind its predecessor PE due to the fact that computations in PE_{i+1} depend on the computation results in PE_i. Figure 14 illustrates the execution of the recursive equations of the original Needleman-Wunsch algorithm on the linear array architecture where diagonal lines cross the matrix cells of dynamic programming matrix whose values are computed at the t^{th} clock cycle.

The linear array architecture keeps record of the maximum value in the dynamic programming matrix at each PE, calculating its *maximum-so-far* value and broadcasting it to the next PE. The gapped extension in the linear array architecture terminates when the end of the query or subject sequence is reached in either side, or when the current result in PE_1 is a certain cut-off value below its *maximum-so-far*. Once both of the linear array architectures in the *GappedExtender* block terminate, their maximum values are added up to obtain the score of the gapped alignment. If this score exceeds certain threshold value, the address of the subject sequence in the related subject sequence memory is sent to the host to allow for the subject sequence to be truly aligned with the query sequence by the high level application software running on the host.

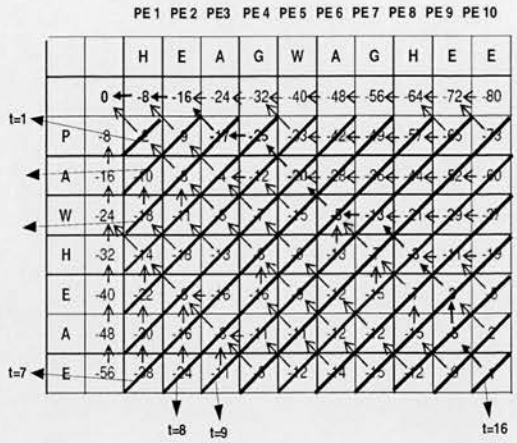


Figure 14. Illustration of the execution of the original Needleman-Wunsch algorithm on the linear systolic array architecture

Note that number of PEs in the linear array architectures should be equal to the number of residues in the query sequence in order to correctly implement the modified Needleman-Wunsch algorithm. However, considering the amount of resources in today's FPGAs, this is impossible since there could be hundreds or even thousands of residues in the query sequence. To solve this problem, the algorithm is partitioned into small alignment steps which are mapped onto a fixed size linear systolic array as shown in figure 15 [14] [15]. In this architecture, the alignment process is performed in a number of passes depending on the length of the query sequence, where a FIFO is used to store intermediate results and subject sequence residues from each pass before they are fed back to the input of the array for the next pass. In our implementation, each of the linear arrays in the *GappedExtender* block has 4 processing elements. This could be extended at will, resource permitting.

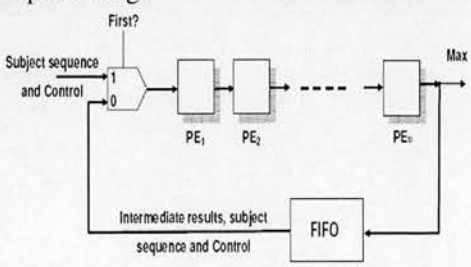


Figure 15. Partitioning and mapping of the modified Needleman-Wunsch algorithm on a fixed size systolic array

IV. Results

Our Gapped BLAST design was captured in the Handel C language to which no specific resource inference or placement constraints were applied. Hence, it can be directly targeted to a variety of FPGA platforms (e.g. Xilinx and Altera FPGAs). The resulting core was compiled into EDIF by Agility's DK5 SP2 suite from which FPGA bitstreams were generated using Xilinx ISE9.2 tool.

The hardware implementation of the core was achieved on a Celoxica RCHTX FPGA board [17] which has a Xilinx Virtex 4 (xc4vlx160ff1148-11) FPGA and off-chip memory fitted on it. In our implementation,

however, the off-chip memory was not used. The operation of the core was tested on the Swiss-Prot protein sequence database [16] with various query protein sequences.

We have also implemented Gapped BLAST with the two-hit method algorithm in C in order compare our hardware implementation with a pure software implementation. Table 1 presents timing performance figures of both hardware and software implementations for 9 random query protein sequences of various lengths searched in the Swiss-Prot database. The FPGA hardware was clocked at 15 MHz. The software implementation was executed on an Intel Centrino Duo 2.2 GHz PC with 2 GB RAM. The same threshold and cut-off values were used in both hardware and software implementations at every step of the algorithm.

As it can be seen from table 1, our FPGA implementation result in substantial speed-up compared to software, ranging from 44x to 20x (the speed-up figure depends on the query sequence). The reason behind this high speed-up figure of the FPGA implementation, despite the huge difference in clock frequency, is due to the high level of process parallelism on FPGA.

Table 1. Timing performance figures of hardware and software implementations for 9 random protein sequences queried in Swiss-Prot protein sequence database

	No of Residues in Query Sequence	No of Query words	FPGA Execution time (sec)	Software execution time (sec)	FPGA Speed-up
1. Query Sequence	111	116	4.45	91.56	20.58
2. Query Sequence	214	98	5.01	131.93	26.34
3. Query Sequence	368	136	4.32	137.42	31.81
4. Query Sequence	459	263	5.88	211.42	35.96
5. Query Sequence	565	137	5.73	181.48	31.67
6. Query Sequence	635	140	5.36	194.45	36.28
7. Query Sequence	746	117	6.83	233.25	34.15
8. Query Sequence	864	240	7.01	311.23	44.40
9. Query Sequence	985	53	5.33	194.12	36.42

V. Conclusion

In this paper, the detailed FPGA implementation of the Gapped BLAST with two-hit method algorithm has been presented. To our knowledge this is the first FPGA implementation of this algorithm ever reported in the literature. The hardware architecture is composed of various blocks each of which performs a specific step of the algorithm in parallel. Moreover, the FPGA core is parameterized in terms of the sequence lengths, match score, gap penalties, cut-off and threshold values. The resulting implementation outperforms an equivalent desktop-based software implementation by at least one order-of magnitude. Furthermore, it was designed in the Handel-C language which makes it FPGA-platform-independent. As a result, the same core can be ported to other FPGA architectures from different vendors.

The work presented in this paper is part of a bigger project which seeks to harness the computational performance and re-configurability features of FPGAs in the field of Bioinformatics and computational biology. Future work includes the extension of this work to the Position Specific Iterated BLAST (PSI-BLAST) algorithm, as well as other sequence analysis techniques based on Hidden Markov Models.

VI. References

[1] Durbin, R., Eddy, S., Krogh, A., and Mitchison, G., 'Biological Sequence Analysis: Probabilistic Models for Proteins and Nucleic Acids', Cambridge University Press, Cambridge UK, 1998

[2] Hein, J. 'A New Methodology that simultaneously aligns and reconstructs ancestral sequences for any number of homologous sequences, when a phylogeny is given'. Journal of Molecular Biology, 6, pp.649-668, 1989

[3] Hoang, D.T. 'Searching genetic databases on Splash 2', in Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, pp. 185-191, 1993.

[4] Gokhale, M. et al. 'Processing in memory: The Terasys massively parallel PIM array', Computer, 28 (4), pp. 23-31, April 1995.

[5] TimeLogic Corporation, 'Decypher Scalable, High Performance Biocomputing Solutions', http://www.timelogic.com

[6] Needleman, S. and Wunsch, C. 'A general method applicable to the search for similarities in the amino acid sequence of two sequences' Journal of Molecular Biology, 48(3), pp.443-453, 1970

[7] Smith, T.F. and Waterman, M.S. Identification of common molecular subsequences. J. Mol. Biol., 147, pp.195-197, 1981

[8] Pearson, W.R. and Lipman, D.J. 'FASTA: Improved tools for biological sequence comparison', Proceedings of the National Academy of Sciences, USA 85, pp. 2444-2448, 1988

[9] Altschul, S. F., Gish, W., Miller, W., Myers, E.W. and Lipman, D.J. 'Basic Local Alignment Search Tool', Journal of Molecular Biology,215, pp. 403-410, 1990

[10] Altschul, S. F., Madden, T. L., Schaffer, A. A., Zhang, J., Zhang, Z., Miller, W., and Lipman, D. J. 'Gapped BLAST and PSI-BLAST: a new generation of protein database search programs', Nucleic Acid Research, Oxford Journals, 25(17), pp. 3389-3402, 1997

[11] Harrison G. A., Tanner, J. M., Pilbeam D. R., and Baker, P. T. 'Human Biology: An introduction to human evolution, variation, growth, and adaptability', Oxford Science Publications, 1988

[12] Chow, E., Hunkapiller, T., Peterson, J., Waterman, M.S. 'Biological Information Signal Processor', Proceedings of Application-Specific Systems, Architectures, and Processors, ASAP ASAP'91, pp. 144-160, 1991

[13] The Handel-C Language Reference Manual, Agility Plc, http://www.agilityds.com

[14] Kung, S. Y. 'VLSI Array Processors', Prentice-Hall, 1988

[15] Moldovan, D. I. and Fortes, J. A. B. 'Partitioning and mapping of algorithms into fixed size systolic arrays', IEEE Transactions on Computers, 35(1), pp. 1-12, January, 1986

[16] Boeckmann, B., et al., 'The SWISS-PROT protein knowledgebase and its supplement TrEMBL' in 2003 Nucleic Acids Research, Vol.31, pp. 365-370, 2003

[17] RCHTX FPGA Board Reference Manual, Celoxica Plc, http://www.celoxica.com

[18] Sotiriades, E.,Dollas,'A General Reconfigurable Architecture for the BLAST Algorithm', Journal of VLSI Signal Processing 48, 189–208, 2007

High Performance Phylogenetic Analysis With Maximum Parsimony on Reconfigurable Hardware

Server Kasap and Khaled Benkrid, *Senior Member, IEEE*

***Abstract*—We present in this paper the detailed field-programmable gate-array (FPGA) design of the Maximum Parsimony method for molecular-based phylogenetic analysis and its implementation on the nodes of an FPGA supercomputer called Maxwell. This is the first FPGA implementation of this method for nucleotide sequence data reported in the literature. The hardware architecture consists in a linear systolic array composed of 20 processing elements each of which performing Sankoff's algorithm for a different tree topology in parallel. This array computes the scores of all theoretically possible trees for a given number of taxa in several iterations. The currently supported maximum number of taxa is 12 but this number can be easily increased. Furthermore, the resulting implementation outperforms an equivalent desktop-based software implementation (using phylogenetic analysis using parsimony software) by several orders of magnitude. The speed-up values achieved by the hardware implementation on a single node of the Maxwell machine can reach up to four orders of magnitude for the 12-taxa case while implementations on several Maxwell nodes can yield even higher speed-ups. This is achieved through harnessing both coarse-grain and fine-grain parallelism available in the algorithm and corresponding hardware implementation platform.**

***Index Terms*—Field-programmable gate array (FPGA), high performance computing, maximum parsimony, phylogenetic analysis, reconfigurable hardware.**

I. Introduction

PHYLOGENETIC analysis is the investigation of the evolution and relationships among organisms that is widely used in the fields of system biology and comparative genomics [1]. It is particularly important in drug and vaccine development. In molecular-based phylogenetic analysis, the relationship between species is estimated by inferring the common history of their genes and then phylogenetic trees are constructed to illustrate evolutionary relationships among genes and organisms [1], [2].

However, phylogenetic tree construction is a computationally intensive operation and desktop computers alone cannot be relied upon to perform this task within acceptable execution times. This is because the number of theoretically possible tree topologies grows exponentially with the number of species under consideration. For instance, it takes over 10 hours to construct the phylogenetic tree for 12 species. Hence, it is mandatory to utilize faster computing platforms such as field-programmable gate arrays (FPGAs). These have indeed been recently proposed as an efficacious and efficient implementation platform for phylogenetic analysis due to their flexible computing and memory architecture which gives them application-specific integrated circuit (ASIC)-like performance with the added programmability feature [3]–[11]. Hence, we chose FPGAs over ASICs because of their reconfigurability feature and shorter development time which results in lower nonrecurring engineering (NRE) costs.

There are various phylogenetic tree construction and phylogenetic analysis methods using different strategies. In this paper, we concentrate on the maximum parsimony (MP) method which is one of the most widely used and most accurate tree construction method [2]. The design and implementation of the FPGA core for parsimony analysis employing Sankoff's dynamic programming algorithm is presented in this paper. Systolic array architecture was selected in our design due to its several benefits for our design. First of all, systolic structures have inherently massive, local, parallelism potential at both coarse and fine-grain levels. Coarse-grain parallelism is through the number of parallel processing elements, whereas the fine-grain parallelism is achieved in each processing element. This is the main reason behind the accomplished high speed-up values. Furthermore, since only the processing element at the border of the array can communicate with the host, communications in the architecture are mostly local (i.e., between and within the processing elements). Hence, communication paths have short delays resulting in high clock frequencies and consequently, high throughput. Moreover, systolic architectures can be easily implemented on FPGAs as demonstrated in the literature.

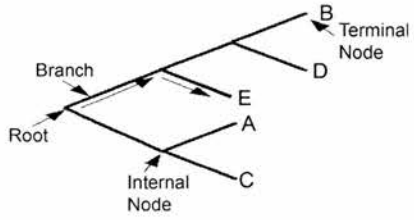
A real hardware implementation of the designed core was achieved on the nodes of an FPGA supercomputer, named Maxwell, which consists of 64 Virtex-4 FPGA chips. To our knowledge, this is the first FPGA implementation of this method for nucleotide sequence data ever reported in the literature. FPGA implementations of other phylogenetic analysis methods and different molecular data have been reported in the past, however, as described in more detail in Section III.

The remainder of this paper will first present essential background information on phylogenetic analysis and then discuss related prior works in the literature. Following this, the MP method for molecular-based phylogenetic tree construction will be detailed. After that, the architecture of the Maxwell FPGA supercomputer will be illustrated. Then, the design and implementation of our FPGA core for the MP method will be elaborated. Following this, implementation results are presented and then evaluated comparatively with equivalent software implementations running on a desktop computer. Finally, conclusions are laid out with plans for future work.

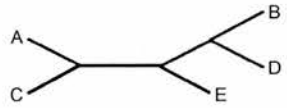
Manuscript received September 06, 2009; revised November 16, 2009.

The authors are with the School of Engineering, The University of Edinburgh, Edinburgh, EH9 3JL Scotland, U.K. (e-mail: s.kasap@ed.ac.uk; benkrid@led.ac.uk).

Digital Object Identifier 10.1109/TVLSI.2009.2039588



ig. 1. Rooted phylogenetic tree.



ig. 2. Unrooted phylogenetic tree.

II. Phylogenetic Analysis

Evolution and relationships among organisms can be investigated in different ways. Although morphology is the classic method of estimating relationships, continuously growing molecular information such as nucleotide or amino acid sequences can also be utilized to infer evolutionary relatedness.

Molecular-based phylogenetic analysis estimates the relationship between species by inferring the common history of their genes through comparing homologous sites with each other. For this reason, sequences under investigation are multiply aligned by some specific algorithms so that homologous sites form columns in the alignment. These alignments are used to construct phylogenetic trees which illustrate evolutionary relationships among genes and organisms.

A. Phylogenetic Trees

Diagrams depicting the relationship of species resemble the structure of a tree. Hence, they are called phylogenetic trees. There are two types of phylogenetic tree: rooted or unrooted. Rooted phylogenetic trees are drawn with a root to the left. Fig. 1 shows an example rooted phylogenetic tree where the root node is indicated. It can be seen that phylogenetic trees are strictly bifurcated (binary).

Phylogenetic trees have some number of external (terminal) nodes which are often called operational taxonomic units (OTUs). OTUs represent existing taxa (i.e., a group of one or more organisms). For instance, B, D, E, A, and C are all terminal nodes in the phylogenetic tree shown in Fig. 1. Also, phylogenetic trees have some number of internal nodes which are called hypothetical taxonomic units (HTUs). HTUs represent hypothetical ancestors of OTUs. Nodes other than root and terminal nodes are internal nodes in phylogenetic tree as shown in Fig. 1. Furthermore, the lines between the nodes are branches. The branching pattern is called the topology of the tree. Fig. 2 shows an example unrooted phylogenetic tree.

An unrooted phylogenetic tree does not indicate the direction of evolution process as seen in Fig. 2 since it is not known which node represents the ancestor of all OTUs. However, in a rooted tree, there is a root node which leads to the common ancestor of all OTUs in it.

In Fig. 1, arrows indicate the direction of evolution from root to terminal node E for instance. Note that an unrooted phylogenetic tree can be rooted with a method named outgroup rooting if a set of the most distantly related OTUs (i.e., outgroup) can be formed. Otherwise, the midpoint rooting method can be utilized. Both of these methods are described in detail in [1].

B. Methods to Reconstruct Phylogenetic Trees

There are various methods to generate phylogenetic trees from nucleotide acid sequence alignments in molecular data based phylogenetic analysis. All of these methods use certain evolutionary assumptions. If these assumptions apply to the date set, the methods perform well.

These methods can be grouped in one way according to whether they use discrete character states or pairwise distance matrices. Character-state methods regard each position in the aligned sequences as a character and the nucleotides and amino acids at that position as states. All characters are compared separately and independently from each other. One advantage of these methods is that they can reconstruct the character state of the internal nodes which represent ancestral taxa.

On the other hand, distance-matrix methods produce a pairwise distance matrix and then infer relationships of the OTUs from that matrix. Although distance-matrix methods can not reconstruct the character state of ancestral nodes like character-state methods, they are much less computer-intensive, and hence faster.

Molecular-based phylogenetic analysis methods can also be grouped according to whether they consider all possible trees or cluster OTUs stepwise to obtain the single best tree. Exhaustive-search methods evaluate all theoretically possible tree topologies for a given number of OTUs using a certain criteria and choose the best one as true phylogeny. One advantage of these methods is that it is possible to assess the confidence in the best tree obtained by comparing it with the second best tree. However, the number of possible trees grows exponentially as the number of taxa increases. Hence, these methods require very high computing power.

On the other hand, stepwise-clustering constructs a single tree by following specific clustering algorithms. Hence, these methods can cope with large numbers of OTUs. However, there is no way to estimate the confidence in correctness of a tree obtained since only one tree is produced in these methods.

Table I lists phylogenetic tree construction and phylogenetic analysis methods classified according to the strategy they use. Note that most of the distance-matrix methods utilize stepwise clustering to construct the best tree whereas all character-state methods search the tree space exhaustively to find the best tree.

In this work, a discrete character method widely used in molecular phylogenetic analysis, namely the MP method, was employed to find the best phylogenetic tree for a given number of taxa where all theoretically possible tree topologies are evaluated. There are some faster heuristic approaches to this method, however, which attempt to heuristically find optimal solutions to the best tree topology problem [1]. Although these approaches have shorter run times in software, they are approximate and hence do not guarantee to find the best tree topology. With faster implementation platforms, however, this

TABLE I
CLASSIFIED PHYLOGENETIC TREE CONSTRUCTION AND PHYLOGENETIC ANALYSIS METHODS

	Exhaustive Search	Stepwise Clustering
Character State	Maximum Parsimony (MP)[12] Maximum Likelihood (ML) [13]	
Distance Matrix	Fitch-Margoliash[14]	UPGMA [15] Neighbour-Joining (NJ) [16]

ompromise need not take place, and that is why we have hosen to accelerate the MP method with exhaustive search on PGA hardware in this work.

III. PRIOR WORK

Although the FPGA implementation of the MP phylogenetic ee construction for nucleotide sequence data has never been eported in the literature, there exist some papers discussing ne hardware implementations of the other phylogenetic anal-sis methods for different types of molecular data. For instance, 3]–[5] describe the design of FPGA-based coprocessor archi-ecture to accelerate the reconstruction of MP phylogenies for ene-arrangement data. The design performs a parallelized ver-ion of the breakpoint median computation which is the most me consuming component of the reconstruction. Reference [3] eports that the breakpoint median hardware core achieves a 005× speed-up over the related desktop software solely for the omputation and a 417× speed-up when the architecture is used o accelerate the entire reconstruction procedure.

Moreover, [6]–[8] present high performance FPGA im-lementations for tackling the tree evaluation process for ucleotide sequences under the maximum likelihood (ML) riterion in order to speed-up the tree reconstruction. Reference 8] proposes a hardware/software (HW/SW) system for solving ne tree reconstruction problem using the genetic algorithm or maximum likelihood (GAML) approach which yields peed-up of 30× to 100× compared to a software solution. urthermore, [6] extends this HW/SW codesign to a more owerful embedded computing platform to achieve much faster omputation speed for phylogeny inference. Also, the FPGA ogic design is based on the idea of partial likelihood this time o improve the tree likelihood evaluation process.

On the other hand, [9] presents the application of custom omputing techniques to speed up the unweighted pair group ethod with arithmetic means (UPGMA), which is the oldest nd simplest method used to generate phylogenetic trees from istance data, by a factor of 100 against equivalent software unning on a desktop computer, for nucleotide sequences. The aper reports on the conducted experiments and discusses how ustom computing techniques can be utilized to accelerate the erformance of phylogenetic analysis algorithms on high-per-ormance computing engines.

Finally, recent papers [10] and [11] present an architecture hich computes the phylogenetic likelihood function (PLF) through the implementation of a massive floating point arithmetic unit using a large number of digital signal processing (DSP) blocks in Xilinx FPGAs [28]. PLF is the most time-consuming kernel of all ML-based programs for the reconstruction of evolutionary relationships. The architecture presented in [10] is reported to achieve speed-ups ranging from 1.6 up to 7.2 compared to a general purpose computer running a highly optimized and parallelized software implementation of the PLF.

IV. MAXIMUM PARSIMONY

The MP method is one of the most widely used discrete character method in molecular phylogenetic analysis [2]. It operates on a character-state matrix which is typically an aligned set of DNA or protein sequences where the states are the nucleotides (i.e., A, C, G, and T) for DNA sequences and symbols of 20 amino acids for protein sequences.

The MP method operates by defining an objective function which returns a score for any input tree topology. This tree score is used to rank all possible trees according to the chosen optimality criterion to find the optimal tree topologies. The parsimony objective function and an algorithm to solve it will be discussed in Sections IV-A and IV-B, respectively. Searching the tree space to find the optimal trees under the parsimony criterion will be detailed in Section IV-C. Following that, Section IV-D will shortly present a software tool for phylogenetic inference named phylogenetic analysis using parsimony (PAUP).

A. Parsimony Analysis

Parsimony criterion is the number of character changes required to explain all nodes of a tree at every sequence position for a given set of aligned sequences. The total amount of character change required by any given tree is called the length of that tree. In parsimony analysis, the aim is to find the tree topologies with the smallest length. Calculating the length of a given tree will be explained and illustrated later in this subsection.

An unrooted binary tree for T taxa contains $T-2$ internal nodes, $2T-3$ branches and T terminal nodes representing sequences of taxa. The length L of an arbitrarily chosen tree r under parsimony criterion is given by the following equation, where l_j is the length for single site j:

$$L(\boldsymbol{r}) = \sum_{j=1}^{N} \boldsymbol{l}_j. \tag{1}$$

In (1), N is the number of sites in the sequence alignment and l_j corresponds to the minimum amount of character change implied by a reconstruction where a character-state x_{ij} is assigned to each node i for each site j. Note that character-state assignment of the terminal nodes is fixed by the input sequences of T taxa. Equation (2) shows the calculation of l_j

$$\boldsymbol{l}_j = \sum_{k=1}^{2T-3} \boldsymbol{C}_{a(k),b(k)}. \tag{2}$$

In (2), $a(k)$ and $b(k)$ represent the states assigned to the nodes at either end of branch k whereas c_{xy} is the cost of change from state x to state y. There are various cost schemes which can be represented as a cost matrix that assigns a cost for the change

	A	C	G	T
A	0	1	1	1
C	1	0	1	1
G	1	1	0	1
T	1	1	1	0

	A	C	G	T
A	0	4	1	4
C	4	0	4	1
G	1	4	0	4
T	4	1	4	0

Fig. 3. Two possible cost matrices.

```
      J
K  ...ACGTCGT...
L  ...TGCCTGA...
M  ...CGTCGTC...
N  ...GCTACTG...
```

Fig. 4. Example alignment of four nucleotide sequences.

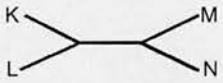


Fig. 5. Tree topology to be evaluated as an example.

etween each pair of character states. Two cost matrices for nu-leotide data are shown in Fig. 3. The matrix at the left hand side f Fig. 3 assigns equal cost of 1 if the nucleotides are different r 0 if they are the same. On the other hand, the right hand side natrix assigns a greater cost to transversions than to transitions. An important point is that cost matrices are symmetric meaning hat c_{xy} is equal to c_{yx}. As a consequence, the length of a tree s the same regardless of the position of the root. Therefore, the earch among tree space can be conducted over unrooted trees ather than rooted trees.

Although there are various algorithms to determine l_j, it vould be helpful to illustrate the calculation of tree length or one site by evaluating all possible $r^T - 2$ character-state econstructions where r is the number of states ($r = 4$ for DNA equences or $r = 20$ for protein sequences). As an example, ve will consider the alignment of 4 nucleotide sequences (i.e., K, L, M, and N) shown in Fig. 4 assuming that the lengths for he first $j - 1$ sites have been calculated and the length of site j s going to be calculated next.

The tree topology that will be evaluated in our example is hown in Fig. 5. The number of character-state reconstructions o be evaluated for the two internal nodes is $4^{(4-2)} = 16$. The engths implied by four of these reconstructions under equal cost cheme are shown in Fig. 6. Note that the minimum length ob-ained from one of these 16 possible combinations of state as-ignments will be the tree length and the associated reconstruc-ion will determine the default states of the two internal nodes t site j. Note that the last reconstruction in Fig. 6 corresponds o the optimal case of all 16 possible reconstructions with the ninimum length of 2.

The brute-force method used in this example can be applied o every site in the sequence alignment to obtain the minimum engths and summing these lengths will give the total length. Iowever, there is a need for better ways to determine the min-mum lengths since the evaluation of $r^T - 2$ reconstructions will ake considerable amount of time and storage when the number f taxa under consideration grows. For this purpose, we will em-loy a straightforward dynamic programming algorithm namely ankoff's algorithm [17] which is illustrated in Section IV-B.

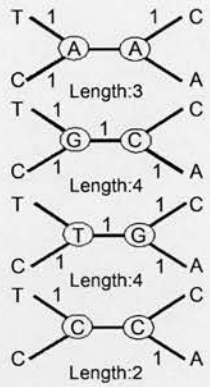


Fig. 6. Four possible combinations of state assignments to the two internal nodes and the resulting lengths.

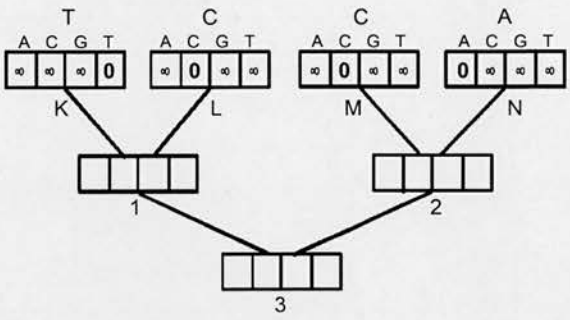


Fig. 7. Example tree topology with conditional subtree length vectors for each node.

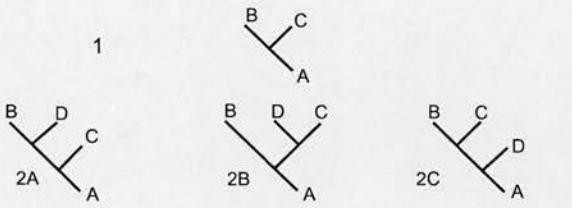
B. Sankoff's Algorithm

Dynamic programming algorithms operate by solving a set of subproblems and then assembling those solutions to find an optimal solution for the whole problem. In the case of Sankoff's algorithm, the best length achievable for each subtree is determined given each of the possible state assignments to each node while moving from the tips toward the root of the tree. An optimal length for the full tree is obtained when the root is reached.

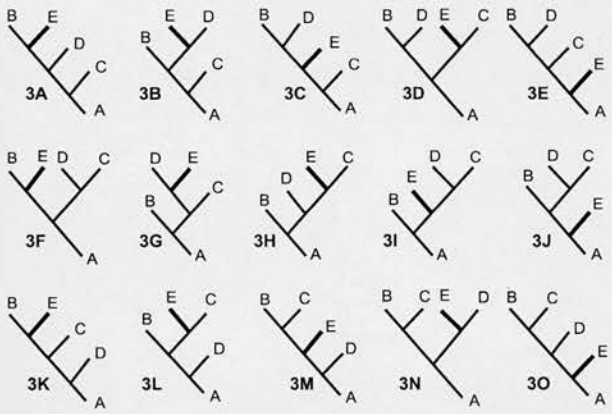
Sankoff's algorithm operates on conditional subtree length vectors which are depicted by rectangular boxes in the tree shown in Fig. 7. It can be seen that for each node i, there is an associated conditional subtree vector S_i containing the minimum possible lengths s_{ik} of the subtree descending from node i if it is assigned state k. Working from the tips toward the root, the algorithm proceeds by filling in the vector at each node based on the values assigned to the pair of vectors above the regarding node. Note that for the terminal nodes, vectors are initialized to 0 for the states actually observed in the sequence alignment or to infinity otherwise. The algorithm will be illustrated in hardware implementation section to ease the comprehension. An important point is that for symmetric cost matrices, an unrooted tree can be arbitrarily rooted to determine the minimum tree length in this algorithm.

C. Searching for Optimal Trees

Since the length of a tree under parsimony criterion can be calculated using Sankoff's algorithm, the search over tree space can now be started to find the optimal tree. However, there is a



ig. 8. Generation of all three possible unrooted trees for the first four taxa.



ig. 9. Generation of all 15 possible unrooted trees for the five taxa.

TABLE II
NUMBER OF POSSIBLE UNROOTED TREES FOR UP TO 12 TAXA

Number of Taxa	Number of Unrooted Trees
3	1
4	3
5	15
6	105
7	945
8	10395
9	135135
10	2027025
11	34459425
12	654729075

eed for an algorithm to generate all possible trees to be evalated under parsimony criterion. Such an algorithm recursively dds the nth taxon in a stepwise manner to all possible trees ontaining the first $n-1$ taxa until all T taxa have been joined. his algorithm will be illustrated for five taxa in the following art of this subsection.

We begin with the only tree for the first three taxa and then onnect the fourth taxon to each of the three branches on this ree to generate all three possible unrooted trees for the first four axa. This process is illustrated in Fig. 8.

Furthermore, we connect the fifth taxon to each branch (five ranches per tree) on each of these three trees to yield all 15 ossible unrooted trees for the five taxa as shown in Fig. 9.

The number of possible trees grows by a factor increasing by wo with each additional taxon as expressed in (3), where $B(t)$ s the number of unrooted trees for t taxa. Table II shows the umber of possible unrooted trees for a given number of taxa

$$B(t) = \prod_{i=3}^{t} (2\mathrm{i} - 5). \tag{3}$$

D. PAUP

PAUP [18] is a phylogenetic analysis program using NEXUS format for input data files. It includes support for the maximum parsimony, maximum likelihood and distance methods as well as some additional capabilities. Details of PAUP can be found in its user manual, command reference manual, and quick start tutorial in [19].

Several versions of PAUP are available with support for a full graphical user interface (Macintosh), a partial graphical user interface (Microsoft Windows), and a command-line only interface (Unix/Linux and Microsoft Windows console). The Macintosh interface allows for the execution of commands via menus and command line whereas the Windows and Unix/Linux interfaces are almost entirely command-line driven. Some menu functions are available in the Windows interface.

Although there is no evidence to suggest that PAUP is the most efficient software tool available in the area of phylogenetic analysis, it is widely used by the Bioinformatics community [1]. Hence, we are going to compare our FPGA hardware implementation with PAUP software.

V. OUR IMPLEMENTATION PLATFORM: THE MAXWELL MACHINE

Maxwell [21] is an FPGA-based supercomputer developed by the FPGA high performance computing alliance (FHPCA) in Scotland [20] to run computationally demanding applications on an array of FPGAs at low energy budgets. Its physical architecture, logical structure and software environment are discussed in Sections V-A–V-C, respectively.

A. Physical Architecture

Maxwell comprises two 19-in racks and five IBM blade centres, four of which have seven IBM Intel Xeon blades and the fifth has four (32 blades in total). The blades are booted over the network from the head node (Dell server). Furthermore, each blade is a diskless 2.8 GHz Xeon with 1 GB memory which hosts 2 Xilinx Virtex4 FPGAs through a PCI-X expansion module. Thus, Maxwell comprises 64 FPGAs having 512 or 1024 MB off-chip memory and four MGT Rocket IO connectors which run at 2.5 Gb/s. Furthermore, the FPGAs are mounted on two different PCI-X card types, namely Alpha Data ADM-XRC-4FX [23] and NallatechHR101 [24]. Both types of cards connect to the Xeon on a particular blade using a PCI-X bridge which is capable of 64 bit, 133 MHz operation meaning a peak bandwidth of 600 MB/s.

Maxwell has three independent communications networks for CPU-CPU, CPU-FPGA, and FPGA-FPGA communication. The blade CPUs are networked over gigabit Ethernet through a single 32-way Netgear switch with 40 Gb/s throughput. Thus, CPUs have an all-to-all connectivity. The FPGA network consists of point-to-point links between the MGT connectors of adjacent FPGAs. Each FPGA has 4 RocketIO links enabling the 64 FPGAs to be connected together in a 2-D 8×8 torus as illustrated in Fig. 10. Finally, FPGAs and CPUs are connected to PCI-X interfaces.

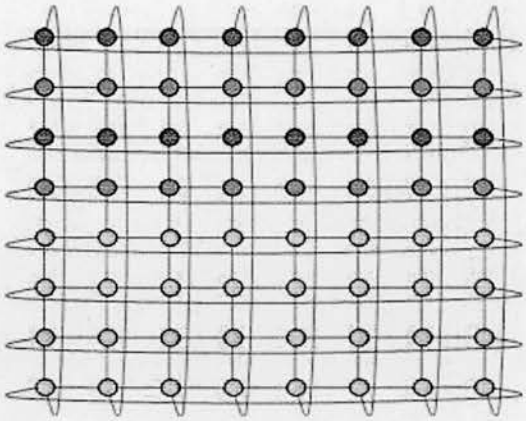


Fig. 10. FPGA connectivity in Maxwell [21].

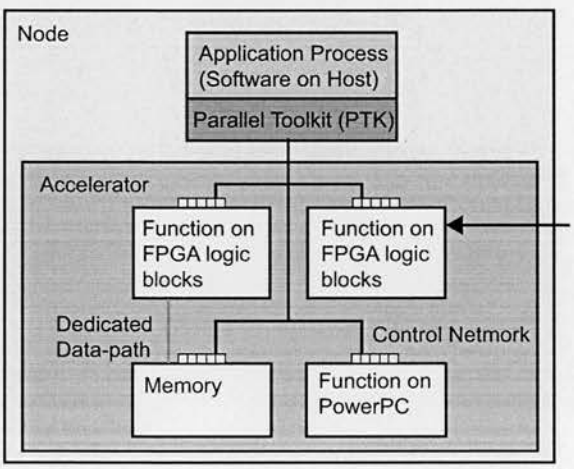


Fig. 11. Logical structure of the Maxwell [25].

B. Logical Structure

Logically, Maxwell can be regarded as a collection of 64 nodes, where a node is defined as a software process running on a host CPU together with some FPGA acceleration hardware as illustrated in Fig. 11. In the typical case of 64 nodes configuration, each blade CPU host two software processes each of which manages one of the two FPGAs in the blade.

C. Software Environment

The software environment of Maxwell comprises Linux variant CentOS, standard GNU/Linux tools, Sun Grid Engine (SGE) as the batch scheduling system, MPI for inter-process communication and most importantly the FHPCA Parallel Toolkit (PTK) [22] that forms a bridge from the application process to the FPGA process (see Fig. 11). Essentially, the PTK is a set of practices and infrastructure written mostly in C++ that aims to address acceleration issues such as associating processes with FPGA resources, associating FPGAs with bitstreams, managing contention for FPGA resources within a process and managing code dependencies to facilitate reuse.

VI. Hardware Implementation

Sankoff's algorithm requires calculation of the conditional subtree length vector for every internal node in a tree. The algorithm will be illustrated first in this section based on the tree shown in Fig. 7 using the cost matrix shown on the right-hand side of Fig. 3. We start with the calculation of the vector values of node 1 (see Fig. 7). For each element k of this vector, the costs associated with each of the four possible state assignments to each of the child nodes K and L and the cost needed to reach these states from state k (obtained from the cost matrix shown on the right-hand side of Fig. 3) are considered. For node 1, these calculations are simple since it is ancestral to two terminal nodes. Hence, only one state needs to be considered for each child node. For example, the minimum length of the subtree descending from node 1 assuming that state A is assigned to node 1 is equal to the sum of the cost of a change from A to T in the left branch and the cost of a change from A to C in the right branch ($s_{1A} = c_{AT} + c_{AC} = 4 + 4 = 8$). In the same manner, s_{1C} is the sum of c_{CT} (left branch) and c_{CC} (right branch) giving

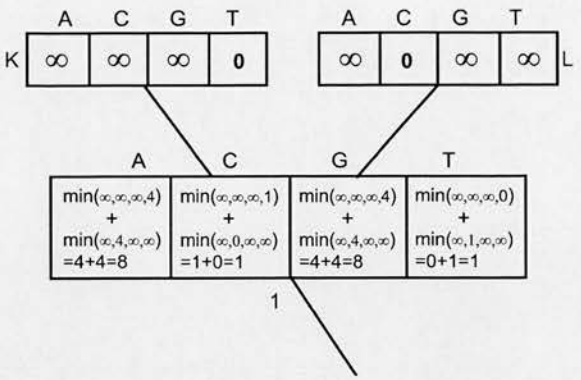


Fig. 12. Calculation of the conditional subtree length vector for node 1.

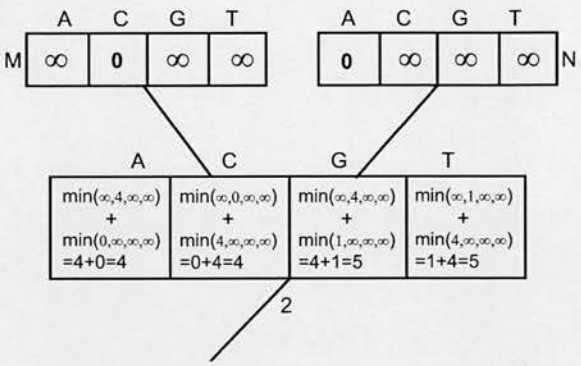


Fig. 13. Calculation of the conditional subtree length vector for node 2.

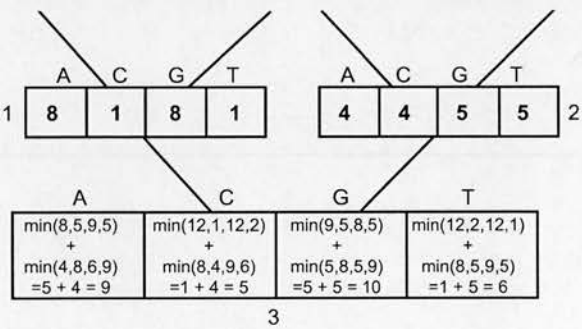
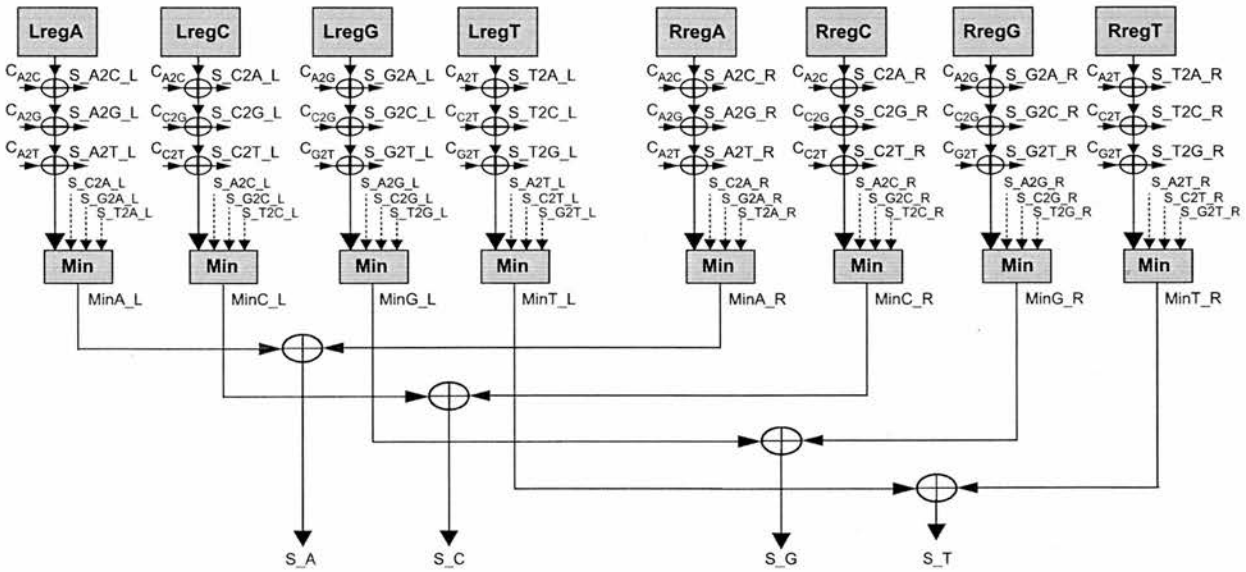


Fig. 14. Calculation of the conditional subtree length vector for node 3.



ig. 15. Simplified hardware architecture for the conditional subtree length vector calculation of the nucleotide sequence.

ne value of 1. Continuing like this, we obtain the entire condi-onal subtree length vector for node 1 as shown in Fig. 12. With ne same procedure, we compute the elements of the vector for ode 2 (see Fig. 7) as shown in Fig. 13. On the other hand, cal-ulations for node 3 (see Fig. 7) are more complicated since we ust consider each of the four state assignments to each of the hild nodes 1 and 3 for each state k at its node. Fig. 14 shows the omputation of the conditional subtree length vector for node 3.

The conditional vector S_3 contains the minimum possible engths for the full tree given each of the four possible state as-ignments to the root. The minimum of these tree lengths is the ee length we seek, which is five in our case as can be seen in ig. 14. Note that different rooting of the tree in Fig. 7 would ield the same length.

This algorithm provides a way to calculate the minimum tree ength for any character on any tree under any cost scheme. The otal length of a given tree can be computed by repeating the nentioned procedure for each character in the sequence align-nent and then adding up all of the obtained minimum lengths or the characters which can be multiplied beforehand by dif-erent weights depending on the importance of the characters in ne alignment.

Fig. 15 shows the hardware architecture which computes the ubtree length vectors of the nucleotides (i.e., A, C, G, and T). n this architecture, registers *LregA*, *LregC*, *LregG* and *LregT* epresent the elements of the vector of the left hand side upper ode (e.g., node 1 in Fig. 14) whereas registers *RregA*, *RregC*, *RregG* and *RregT* represent the elements of the vector of the ght-hand side upper node (e.g., node 2 in Fig. 14).

Each of these registers are added up with three different spe-ific cost values (i.e., C_{A2C}, C_{A2G}, C_{A2T}, C_{C2G}, C_{C2T}, C_{G2T}) o obtain three subscores (e.g., S_A2C_L, S_A2G_L, S_A2T_L or *LregAk*) and then each register and its associated three sub-cores (e.g., S_C2A_L, S_G2A_L, S_T2A for *LregA*) are in-utted to the combinational block *Min* to find the minimum of nese values MinX_Y (X = A, C, G, or T and Y = L or). Furthermore, two minimum values for each nucleotide (e.g., MinA_L and MinA_R for A) are added to obtain the scores for each nucleotide (i.e., S_A, S_C, S_G, S_T) which are the elements of the vector of the target node (e.g., node 3 in Fig. 14).

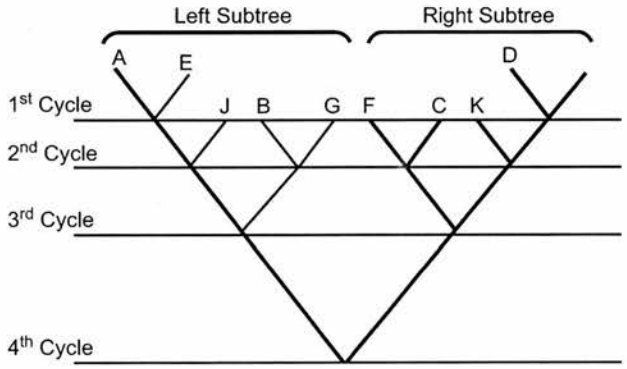


Fig. 16. Tree topology illustrating the parallelism of Sankoff's algorithm.

A. Parallel Implementation of Sankoff's Algorithm

An important point is that some of these node vectors can be computed at the same time. For example, in the 10-taxa tree shown in Fig. 16, computations for nodes on the same line can be done in parallel. Vectors of nodes on different lines are computed consecutively starting from the first line until the root node is reached. FPGAs can take advantage of this parallelism of Sankoff's algorithm to accelerate it by computing several node vectors concurrently. For the tree topology in Fig. 16, FPGA hardware would calculate 2, 4, and 2 node vectors in parallel in the first, second, and third clock cycles, respectively. In the fourth clock cycle, a vector for the root node would be calculated to obtain the minimum length (score) of the tree. Hence, the score of the tree is computed in four clock cycles in total rather than the nine cycles required in the case of sequential node calculations. Note that the tree under consideration should be rooted in a way so that the left and right subtrees of the rooted

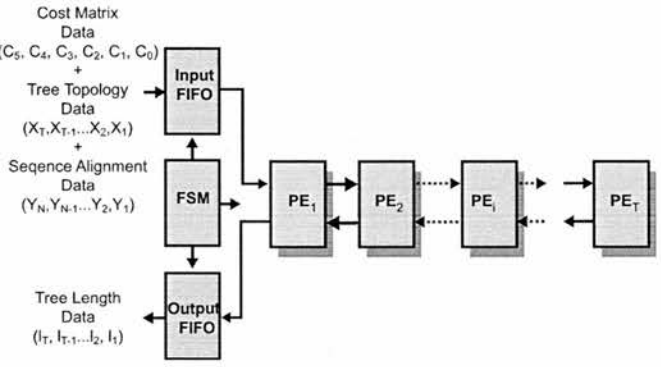


Fig. 17. Linear systolic array for Sankoff's algorithm.

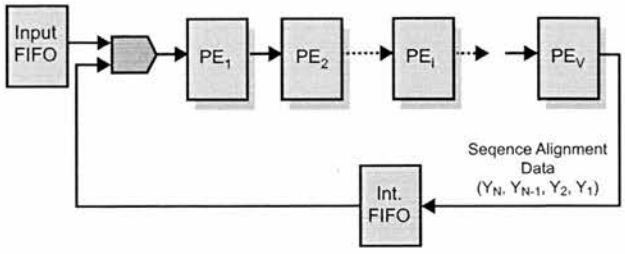


Fig. 18. Partitioning and mapping of Sankoff's algorithm on a fixed size systolic array.

ree will have almost the same number of taxa to maximize the vailable parallelism (i.e., tree balancing).

The hardware architecture which computes node vector alues (see Fig. 15) can be used within a linear systolic array o implement the complete Sankoff's algorithm in a parallel nanner as explained in Section VI-B. Also, Section VI-C elaborates on the inner structure of the processing element onstituting this array.

3. *Linear Systolic Array Implementation of Sankoff's Algorithm*

Fig. 17 shows a linear systolic array which implements Sankoff's algorithm. It is composed of several processing elements PE_i, each of which contains a number of sub-elements with similar architecture as shown in Fig. 15, in order to compute node vector values. Each PE (processing element) calculates the score of a different tree topology in parallel independently from each other. Hence, the total number of PEs s equal to the number of theoretically possible tree topologies or the given number of taxa.

The architecture in Fig. 17 also comprises two *FIFO*s and an *FSM*. The *Input FIFO* is fed by high level application software unning on the host computer with cost matrix, tree topology nd sequence alignment data in respective order. Concurrently, he linear array reads the *Input FIFO* to first get the values of he chosen cost matrix which are then shifted through the processing elements within the array. Following this, tree topology vectors whose number is equal to the number of possible tree opologies are read and shifted through the array independently o configure each PE to operate on one specific tree topology.

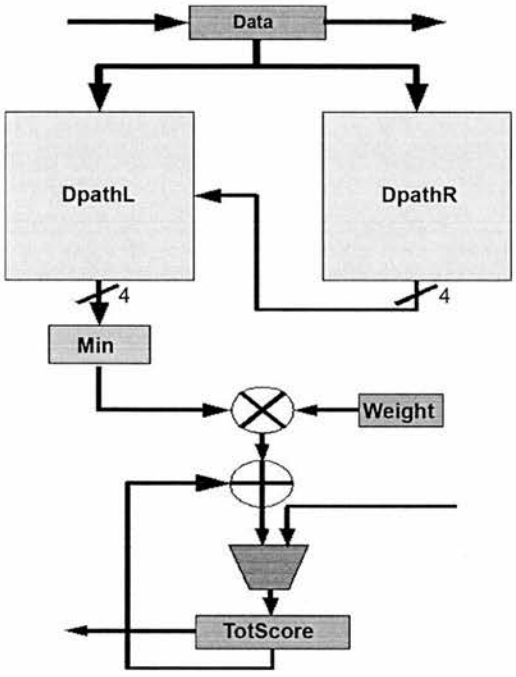
Finally, nucleotide vectors composed of nucleotides at one ite of the sequence alignment (e.g., site j in Fig. 4) under consideration are read and shifted through the array one by one so as o enable the processing elements to compute the scores for that pecific alignment site for all tree topologies in parallel. When he first PE finishes its operation for one nucleotide vector, another vector is read and shifted through the array until there is no nore nucleotide vector left in the *Input FIFO*. When every PE s done with the last nucleotide vector, the total tree scores computed by accumulating the score of each alignment site during he whole process in each PE are shifted backwards through the array into the *Output FIFO* to be read by the application software. The *FSM* coordinates all of these operations of the PEs, *Input FIFO* and *Output FIFO* in accordance with control data coming from the application software running on the host.

As mentioned before, the number of PEs in the linear array is equal to the number of possible tree topologies. However, considering the amount of resources in today's FPGAs, this is not always feasible since there could be hundreds or even thousands of theoretically possible tree topologies for a given number of taxa as seen in Table II. To solve this problem, the algorithm is partitioned into small steps which are mapped onto a fixed size linear systolic array as shown in Fig. 18 [26], [27] .

In this architecture, the tree evaluation process is performed in numerous iterations (or passes) for each set of tree topologies. Obviously, the number of iterations depends on the number of possible tree topologies. The additional *FIFO* in this architecture is used to store the sequence alignment data shifted in the first pass which will be read by the array in the next passes when the time comes for shifting all nucleotide vectors through the array. On the other hand, the *Input FIFO* is read to obtain a new set of tree topology vectors at each pass while there is no need to read cost matrix data after the first pass.

C. *Architecture of a Processing Element*

Fig. 19 shows the simplified inner structure of a processing element which is mainly composed of *DpathL* and *DpathR* blocks. Data read from the *Input FIFO* is shifted through the array via linked *Data* registers in the PEs as illustrated in Fig. 17 to be used by *DpathL* and *DpathR* blocks which implement Sankoff's algorithm on the left and right subtrees (see Fig. 16) of a tree topology, respectively. In the architecture, the score of the right branch computed by the *DpathR* is inputted to the *DpathL* for the calculation of the four elements of the root node vector which are then inputted to the *Min* block to find the minimum of them. The minimum value is the score of the tree at a specific site of the sequence alignment (e.g., site j in Fig. 4) under consideration. This score is multiplied by the *Weight* register which holds the weight of that site within the alignment and then, the obtained result is added to the *TotScore* register which will hold the total score of the tree topology when the computations for the last site in the alignment is finished in the PE. The value of the *TotScore* registers which are linked to each other are shifted backwards into the *Output FIFO* as illustrated Fig. 17 when every PE in the array is done with the computation of the total score of its assigned tree topology.



ig. 19. Simplified inner structure of a processing element (annotated numbers epresent number of words).

Fig. 21 shows the simplified inner structure of the *DpathL* lock which contains one *DpathUnitL* block and three *DpathU-itR* blocks. *DpathUnitL* and *DpathUnitR* blocks are respon-ible from conditional node vector calculation (see Fig. 15). ach *DpathUnitR* has 4 data inputs two of which are coming om outside the *DpathUnitL* and *DpathUnitR* blocks, one of hich is coming from its *Min & Add Op.* block (whose inner tructure is shown in Fig. 15) and last of which is coming om the *Min & Add Op.* block of the right-hand side neighbor *DpathUnitR* block. On the other hand, *DpathUnitL* has five ata inputs four of which are like those of *DpathUnitR* and the fth one (input R) is coming from outside the *DpathL*. Also, he *DpathUnitL* and *DpathUnitR* blocks have control inputs C_x that determine which data inputs will be registered. For ex-mple, if C_c is asserted, input C will be stored in *Lreg_1* in the ext cycle. With various combinations of these control signals, *DpathL* can compute conditional vectors of multiple nodes (up o four nodes) in various topological forms at the same cycle. urthermore, *DpathUnitL* block of the *DpathL* is employed to ompute the root node vector of the tree under consideration sing its input R coming from *DpathR* (see Fig. 19).

Note that *DpathR* has a similar structure to that of *DpathL* ut it has one less *DpathUnitR* block. So, it can compute condi-onal vectors of up to three nodes concurrently. *DpathL* will be xplained more in detail next in this subsection.

DpathL block incorporates four arrays of registers (*CostReg*, *reeStructReg*, *TaxaOrderReg*, and *ResidueReg*) which are ed by the *Data* register shown in Fig. 19. *CostReg* stores the alues of the cost matrix which are used within *Min & Add p.* block of each *DpathUnitL* and *DpathUnitR* blocks while *reeStructReg* contains control configurations for the specific ee topology. *TreeStructReg* is decoded to obtain appropriate ontrol signals for all multiplexers in the datapath with the

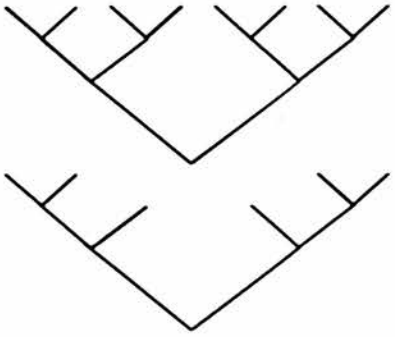


Fig. 20. Most complicated subtree topologies supported by DpathL (upper one) and DpathR (lower one).

help of *TreeStructIndReg* incrementing by one at every cycle. Furthermore, *ResidueReg* keeps the nucleotides of the specific site in the sequence alignment a set of which is applied to the data inputs of the *DpathUnitL* and *DpathUnitR* blocks (i.e., inputs A, B, E, F, J, K, N, and O) appropriately at every cycle. The applied set of nucleotides is determined by the *TaxaOrderReg* with the help of *TaxaOrderIndReg* which is incremented every cycle by some value depending on the current control configuration in *TreeStructReg*. Note that the values of *TreeStructReg* and *TaxaOrderReg* at a time constitute a tree topology vector whereas contents of *ResidueReg* are obviously nucleotide vectors (see Subsection VI-A).

With their architecture, *DpathL* and *DpathR* can process any subtree topology with up to 8 and 6 taxa, respectively. So, the most complicated subtree topologies *DpathL* and *DpathR* can handle are the ones shown in the upper and lower halves of Fig. 20, respectively. Finally, a processing element in the linear array (see Fig. 17) can support a tree topology with at most 12 taxa.

VII. Implementation Results

The MP method was implemented on the Alpha Data nodes of the Maxwell machine with the array architecture shown in Fig. 18, where the number of the processing elements was 20. Our design was captured in Verilog hardware description language which was then synthesized, placed, and routed by Xilinx ISE9.2 tool. FPGA bitstreams were also generated by the same tool while ModelSim was employed to test the core with a number of testbenches. The clock frequency of the FPGAs was set to 70 MHz. Note that only one FPGA bitstream is used to configure the FPGAs regardless of the number of taxa under consideration. Another important point is that the time it takes to load the bitstream to FPGA is in milliseconds (ms) whereas the computation time on FPGAs is in seconds (s). Hence, FPGA configuration time does not affect the overall computation time that much. A high level application process was built using the FHPCA Parallel Toolkit (PTK) and run on the host CPUs. Its main duty was to write the cost matrix data, tree topology data and sequence alignment data to the input FIFO and then read the scores of the tree topologies from the output FIFO of the FPGAs with Direct Memory Access (DMA) transfers (see Fig. 17). On the other hand, a small C program was written to construct the tree topology data for various numbers of taxa.

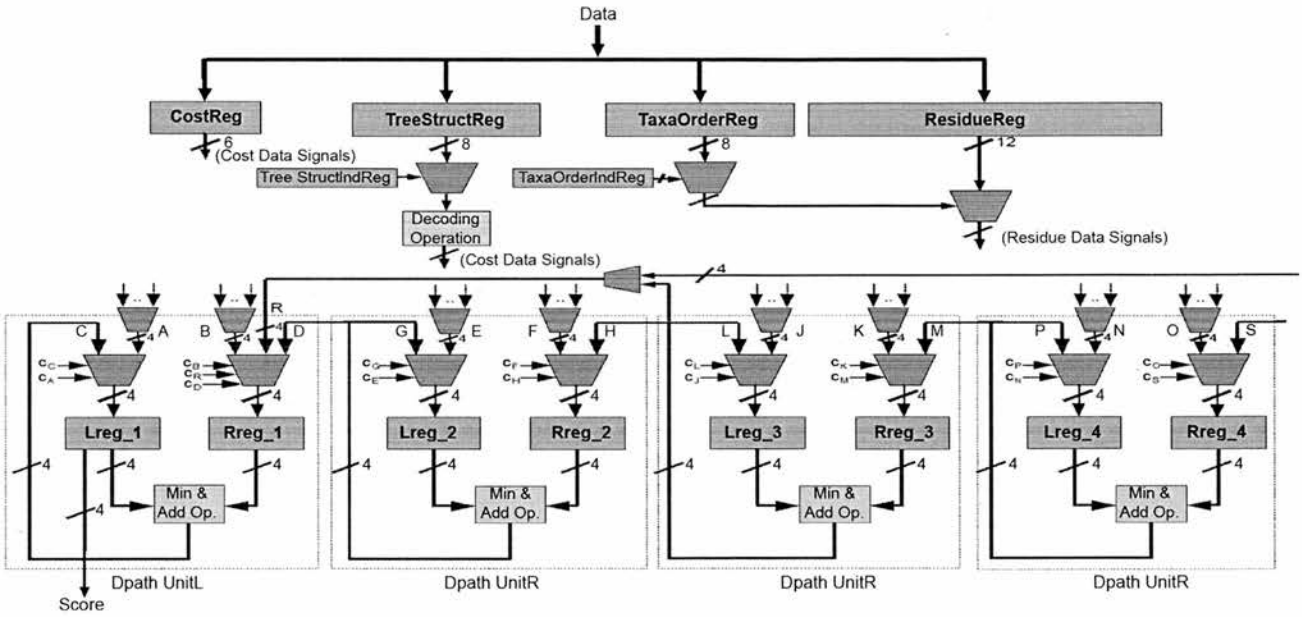


Fig. 21. Simplified inner structure of the DpathL block (annotated numbers represent number of words).

TABLE III
TIMING PERFORMANCE FIGURES OF THE HARDWARE IMPLEMENTATION FOR THE MP METHOD ON ONE NODE OF THE MAXWELL MACHINE

No. of Taxa	No. of Trees	No. of Iterations	Min. Score	Average Time (s)
4	3	1	778	1.420
5	15	1	927	1.421
6	105	6	1124	1.423
7	945	48	1361	1.425
8	10395	520	1396	1.430
9	135135	6757	1488	1.480
10	2027025	101352	1587	2.255
11	34459425	1722972	1898	3.446
12	654729075	32736454	2230	5.893

Table III presents the performance figures of our hardware implementation for the MP method for up to 12 nucleotide sequences on one node of the Maxwell machine (each node has a Xilinx Virtex-4 XC4VFX100 FPGA [28]). It was assumed that the cost of changes from a purine (A or G) to pyrimidine (C or T) is two times the cost of changes from a purine to a purine and pyrimidine to a pyrimidine. The length of the nucleotide sequences was 898 where a two times higher weight was applied to the changes occurring at the first position of the codons compared to the second and third positions.

Each row in Table III is associated with some number of taxa given in the first column where the second column presents the number of unrooted tree topologies to be searched for that specific number of taxa. Furthermore, the third column gives the number of iterations required by the hardware core considering the number of available PEs to complete the processing of all trees (see Section VI-A) while the fourth column shows the score of the most parsimonious tree found during the exhaustive tree search. Finally, the fifth column gives the average time in seconds taken by the hardware core to complete its operation for each number of taxa.

For comparative purposes, Table IV below shows the timing figures of the PAUP software execution configured to operate in the same way as the hardware implementation, with the same nucleotide sequences. The software version was run on a 2.2 GHz Intel Centrino Duo machine with 2 GB of RAM running Windows XP operating system. Note that results obtained by the hardware implementation were identical to those of PAUP.

TABLE IV
TIMING PERFORMANCE FIGURES OF THE PAUP SOFTWARE FOR THE MP METHOD

No. of Taxa	No. of Trees	Average Time (s)
4	3	0.02
5	15	0.02
6	105	0.02
7	945	0.16
8	10395	0.67
9	135135	7.84
10	2027025	241
11	34459425	5852
12	654729075	127325

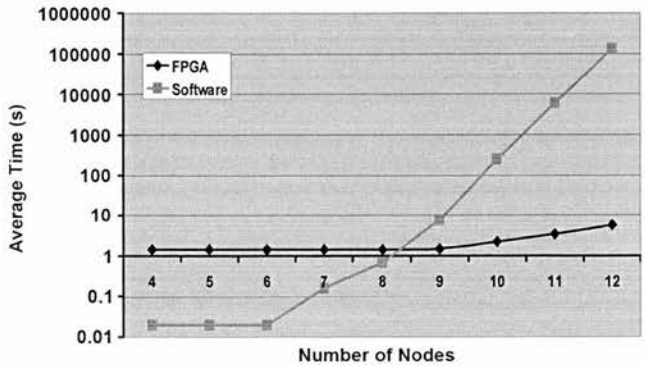
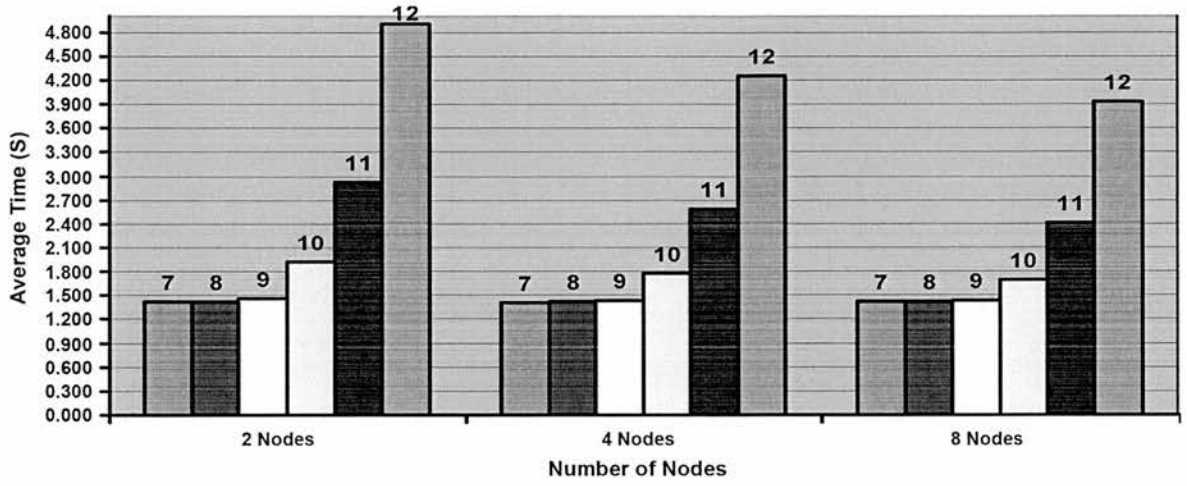


Fig. 22. Timing performance plot of the FPGA and software solutions for the MP method (scale is logarithmic).

Fig. 22 plots the timing performance results of the FPGA and software implementations shown in Tables III and IV with a logarithmic scale. As it can be seen, for low numbers of taxa, PAUP



ig. 23. Scaling performance of the hardware core on multiple nodes of the Maxwell for given numbers of taxa.

TABLE V
SOFTWARE (PAUP) VERSUS 1-NODE HARDWARE IMPLEMENTATION SPEED-UP VALUES

No. of Taxa	FPGA Speed-Up
9	5
10	110.1
11	1698.2
12	21606.1

perates faster than the FPGA hardware implementation. How-ver, the latter becomes much faster as the number of taxa in-reases. Note that both plots in Fig. 22 show an exponentially ncreasing curve which is obviously much sharper for the soft-vare solution for the MP method.

Table V below provides the speed-up values of the hardware mplementation on 1 node over the software implementation PAUP) for various numbers of taxa. It is obvious that hard-vare core outperforms PAUP hugely when the number of taxa is ver 8 with the speed-up values reaching 21606× for the 12-taxa ase.

Tables VI–Table VIII below show the timing figures of the PGA implementation for the MP method on 2, 4, 8 nodes of the Maxwell machine, respectively, where the tree topologies for a iven number of taxa are shared and distributed among the spec-ied number of nodes by the master node among the CPUs of the odes using MPI [29]. Furthermore, the third columns in these ables present the maximum number of iterations required by the ardware core on a node, while the fourth columns give the total me taken to complete the whole process including the collec-on of the tree scores from each node by the master node. It can e noticed that average times taken are decreasing as the number f utilized nodes increases although the overhead of distributing ee topology data and collecting results may surpass the gain rom the parallel operation of the nodes in the case of a low umber of taxa. The effects of this communication overhead on ne efficiency and scalability of our design over multiple nodes s graphically represented in Fig. 23 with the timing values for ach given number of taxa as presented in Tables VI–VIII.

Finally, Table IX below provides the speed-up values of the ardware implementation on 2, 4, and 8 nodes over the software

TABLE VI
TIMING PERFORMANCE FIGURES OF THE HARDWARE IMPLEMENTATION OF THE MP METHOD ON 2 NODES OF THE MAXWELL MACHINE

No. of Taxa	Total No. of Iterations	No. of Iterations per Node	Average Time (s)
7	48	24	1.423
8	520	260	1.428
9	6757	3379	1.460
10	101352	50676	1.930
11	1722972	861486	2.926
12	32736454	16368227	4.911

TABLE VII
TIMING PERFORMANCE FIGURES OF THE HARDWARE IMPLEMENTATION OF THE MP METHOD ON FOUR NODES OF THE MAXWELL MACHINE

No. of Taxa	Total No. of Iterations	No. of Iterations per Node	Average Time (s)
7	48	12	1.415
8	520	130	1.423
9	6757	1690	1.443
10	101352	25338	1.780
11	1722972	430743	2.584
12	32736454	8184114	4.256

TABLE VIII
TIMING PERFORMANCE FIGURES OF THE HARDWARE IMPLEMENTATION OF THE MP METHOD ON EIGHT NODES OF THE MAXWELL MACHINE

No. of Taxa	Total No. of Iterations	No. of Iterations per Node	Average Time (s)
7	48	6	1.418
8	520	65	1.425
9	6757	845	1.433
10	101352	12669	1.690
11	1722972	215372	2.412
12	32736454	4092057	3.928

implementation for various numbers of taxa up to 12. Note that the poor scaling in performance when using multiple nodes (as can be seen in Fig. 23) is due to the fixed communication latency between the host computer and corresponding FPGA. This is a generic problem which will always occur when the ratio of

TABLE IX
SOFTWARE (PAUP) VERSUS 2-NODES/4-NODES/8-NODES HARDWARE IMPLEMENTATIONS SPEED-UP VALUES

No. of Taxa	FPGA Speed-up with 2 Nodes	FPGA Speed-up with 4 Nodes	FPGA Speed-up with 8 Nodes
9	5.4	5.4	5.5
10	124.9	135.4	142.6
11	2000	2264.7	2426.2
12	25929.3	29916.6	32414.7

he computation time on FPGA hardware to the communication ime between host and FPGA becomes low.

VIII. CONCLUSION

In this paper, the detailed FPGA implementation of the Max- mum Parsimony method for molecular phylogenetic analysis on the nodes of the Maxwell FPGA supercomputer has been presented. This is the first FPGA implementation of this method for nucleotide sequence data reported in the literature to our knowledge. The hardware architecture is a linear systolic array composed of 20 processing elements each of which performing Sankoff's algorithm for a different tree topology in parallel. This array computes the scores of all tree topologies for a given number of taxa in several iterations.

The currently supported maximum number of taxa is 12 but his number can be easily improved by cascading more *DpathU- nits* in *DpathL* and *DpathR* blocks. Furthermore, the resulting mplementation outperforms an equivalent desktop-based soft- ware implementation (PAUP) by very high orders-of magnitude. The speed-up values achieved by the hardware implementation on a single node of Maxwell can reach up to 21 606× for the 12-taxa case while implementations on several nodes can yield even higher values. The reasons behind this very high speed-up are essentially twofold: the first is the coarse-grain parallelism among processing elements, since each PE processes a dif- ferent tree topology in parallel with other PEs, and second is he fine-grain parallelism achieved in each processing element, as conditional vectors of several nodes on a specific level of the tree topology under consideration are computed concurrently (see Fig. 16).

The work presented in this paper is part of a bigger project which aims to harness the computational performance and re- configurability features of FPGAs in the field of bioinformatics and computational biology. As a short-term future goal, we plan to extend and improve the presented architecture to be able to support computations for unlimited number of taxa by incorpo- rating a reconfigurable router into the design. On the other hand, we plan to design a web-based interface for our design through which bioinformaticians can submit their sequences online for high performance phylogenetic tree construction on Maxwell FPGA-based supercomputer as a long-term future goal.

REFERENCES

[1] M. Salemi and A. M. Vandamme, *The Phylogenetic Handbook: A Practical Approach to DNA and Protein Phylogeny*. Cambridge, U.K.: Cambridge University Press, 2003.

[2] K. K. Kidd and L. A. Sgaramella-Zonta, "Phylogenetic analysis: Concepts and methods," *Amer. J. Human Genetics*, vol. 23, pp. 235–252, 1971.

[3] J. D. Bakos and P. E. Elenis, "Special-purpose architecture for solving the breakpoint median problem," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 12, pp. 1666–1676, Dec. 2008.

[4] J. D. Bakos, P. E. Elenis, and J. Tang, "FPGA acceleration of phylogeny reconstruction for whole genome data," in *Proc. IEEE Bioinform. Bioeng. Conf.*, 2007, pp. 888–895.

[5] J. D. Bakos, "FPGA acceleration of gene rearrangement analysis," in *Proc. Field-Program. Custom Comput. Mach.*, 2007, pp. 85–94.

[6] T. S. T. Mak and K. P. Lam, "Embedded computation of maximum-likelihood phylogeny inference using platform FPGA," in *Proc. IEEE Computational Syst. Bioinform. Conf.*, 2004, pp. 512–514.

[7] T. S. T. Mak and K. P. Lam, "FPGA-based computation for maximum likelihood phylogenetic tree evaluation," presented at the Field-Program. Logic Appl. Conf., Leuven, Belgium, 2004.

[8] T. S. T. Mak and K. P. Lam, "High speed GAML-based phylogenetic tree reconstruction using HW/SW codesign," in *Proc. IEEE Computational Syst. Bioinform. Conf.*, 2003, pp. 470–473.

[9] J. P. Davis, S. Akella, and P. H. Waddell, "Accelerating phylogenetics computing on the desktop: Experiments with executing UPGMA in programmable logic," *Proc. IEEE Eng. Med. Biol. Soc.*, pp. 2864–2868, 2004.

[10] N. Alachiotis, E. Sotiriades, A. Dollas, and A. Stamatakis, "A reconfigurable architecture for the phylogenetic likelihood function," presented at the IEEE Conf. Field Program. Logic Appl., Prague, Czech Republic, 2009.

[11] N. Alachiotis, E. Sotiriades, A. Dollas, and A. Stamatakis, "Exploring FPGAs for accelerating the phylogenetic likelihood function," presented at the IEEE Workshop High Perform. Computational Biol., Rome, Italy, May 2009.

[12] W. M. Fitch, "Toward defining the course of evolution: Minimum change for a specific tree topology," *Systematic Zool.*, vol. 20, pp. 406–416, 1971.

[13] J. Felsentein, "Evolutionary trees from DNA sequences: A maximum likelihood approach," *J. Mol. Evol.*, vol. 17, pp. 368–376, 1981.

[14] W. M. Fitch and E. Margoliash, "Construction of phylogenetic trees," *Science*, vol. 155, pp. 279–284, 1967.

[15] J. S. Farris, "Estimating phylogenetic trees from distance matrices," *Amer. Nature*, vol. 106, pp. 645–668, 1970.

[16] N. Saitou and N. Nei, "The neighbour-joining method: A new method for reconstructing phylogenetic trees," *Mol. Biol. Evol.*, vol. 4, pp. 406–425, 1987.

[17] D. Sankoff and P. Rousseau, "Locating the vertices of a steiner tree in an arbitrary metric space," *Math. Progr.*, vol. 9, pp. 240–276, 1975.

[18] D. L. Swofford, *PAUP*: Phylogenetic Analysis Using Parsimony (* and Other Methods), Version 4.0b10*. Sunderland, MA: Sinauer Associates Inc., 2002.

[19] PAUP, Sunderland, MA, "Download website for PAUP," 2009. [Online]. Available: http://paup.csit.fsu.edu/downl.html

[20] FHPCA, Edinburgh, U.K., "The FHPCA website," 2009. [Online]. Available: http://www.fhpca.org

[21] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, R. Smart, A. Cantle, R. Chamberlain, and G. Genest, "Maxwell—a 64 FPGA supercomputer," presented at the NASA/ESA Conf. Adapt. Hardw. Syst., Edinburgh, U.K., 2007.

[22] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, R. Smart, A. Cantle, R. Chamberlain, and G. Genest, "The FPGA HPC alliance parallel toolkit," presented at the NASA/ESA Conf. Adapt. Hardw. Syst., Edinburgh, U.K., 2007.

[23] Alpha Data Ltd., Edinburgh, U.K., "ADM-XRC-4FX Datasheet," May 2007. [Online]. Available: http://www.alphadata.co.uk./adm-adm-xrc-4fx.html

[24] Nallatech Ltd., Glasgow, U.K., "H100 Series Datasheet," May 2007. [Online]. Available: http://www.nallatech.com/meadiLibrary/images/english/5595.pdf

[25] FHPCA, Edinburgh, U.K., "PowerPoint presentation," 2007. [Online]. Available: http://www.fhpca.org/download/MRSC07-Mar07.ppt

[26] S. Y. Kung, *VLSI Array Processors*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[27] D. I. Moldovan and J. A. B. Fortes, "Partitioning and mapping of algorithms into fixed size systolic arrays," *IEEE Trans. Comput.*, vol. 35, no. 1, pp. 1–12, Jan. 1986.

[28] Xilinx Inc., San Jose, CA, "Virtex-4 datasheets," May 2007. [Online]. Available: http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm

[29] Argonne National Lab, Argonne, IL, "MPI manual," 2009. [Online]. Available: http:/www.unix.mcs.anl.gov/mpi/www/www3/MPI_Wtime.html

Server Kasap (S'07) received the B.Sc. degree in electrical and electronical engineering from the Middle East Technical University, Turkey, in 2006 and the M.Sc. degree with distinction in system level integration from the University of Edinburgh, U.K., in 2007, where he is currently pursuing the Ph.D. degree from the System Level Integration Research Group, the School of Engineering.

His research interests include high performance reconfigurable architectures for bioinformatics and molecular biology applications, high performance omputing, reconfigurable computing and hardware/software codesign.

Khaled Benkrid (SM'06) received the "Ingenieur d'Etat" degree in electronics engineering with distinction from Ecole Nationale Polytechnique d'Alger, Algeria, Alger, and the Ph.D. degree in computer science and an Executive MBA with distinction from Queen's University Belfast, Belfast, U.K.

With over ten years experience in FPGA hardware design, he has authored over 70 publications in major international journals and conference papers in the areas of high performance reconfigurable computing and electronic design automation with applications in digital signal processing, communication systems, bioinformatics and computational biology, financial computing, and scientific computing in general.

Dr. Benkrid is a Chartered U.K. Engineer. He has served as conference chair, programme chair, program committee member, and session chair at many international conferences on parallel and reconfigurable hardware, VLSI design, and computer science. He is the holder of various research grants from the U.K. government, the EU, and Industry, and is a recognized expert reviewer by the European Commission.

Parallel Processor Design and Implementation for Molecular Dynamics Simulations on a FPGA Parallel Computer

Server Kasap and Khaled Benkrid

Abstract—The design and implementation of a FPGA core that parallelises all the necessary operations to compute the non-bonded interactions in a MD simulation with the purpose of accelarating the LAMMPS MD software is presented in this paper. Our MD processor core comprised of 4 identical pipelines working independently in parallel to evaluate the non-bonded potentials, forces and virials was implemented on the nodes of a FPGA-based supercomputer. Implementing our FPGA core on multiple nodes of Maxwell allowed us to produce a special-purpose parallel machine for the hardware acceleration of MD simulations. The timing performance figures of this machine for the pairwise LJ and short-range Coulombic (via PPPM) interaction computations in the MD simulations of the solvated Rhodopsin protein systems with various numbers of atom show performance gains over the pure software implementation by factors of up to 13 on two nodes of the Maxwell machine. Furthermore, our MD machine is highly scalable, yielding higher computational power with the additional Maxwell nodes.

Index Terms—Molecular Dynamics Simulation, Arithmetic and Logic Structures, Logic Design, Reconfigurable Hardware, Register-Transfer-Level Implementation, Special-Purpose and Application-Based Systems, Performance of systems.

1 Introduction

Computer simulations are carried out to understand the properties of assemblies of molecules in terms of their structure and the microscopic interactions between them [1]. They act as a bridge between microscopic length and time scales and the macroscopic world of the laboratory, serving as a complement to conventional experiments. Carrying out simulations on computers that are either difficult or impossible in the labarotory enables us to learn something new, something that can not be found out in other ways.

There are two main families of simulation techniques; Molecular Dynamics (MD) and Monte Carlo (MC)-based simulations. There are also several hybrid techniques which combine features from both. MD is a deterministic simulation technique whereas simulation results from MC simulations are stochastic. Furthermore, MD can provide the dynamic properties of the simulated system as well as the static properties, as opposed to MC.

In MD, the time evolution of a set of interacting atoms modelled with classical mechanics is followed by integrating their Newtonian equations of motion. MD simulations of biomolecules provide a molecular picture of the structure and behaviour of biological systems such as enzymes, proteins, DNA strands and membranes. This allows scientists to advance their understanding of biologically important molecules. The MD method has applications in the fields of protein engineering [2], drug design [3], [4] and refinements of structures based on X-ray [5] and NMR experiments [6].

However, biological systems of interest have sizes ranging from a few tens of thousands to millions of atoms. Performing MD simulation of a biological process, such as protein folding, for a reasonable physical time requires enormous amounts of computational effort and may take years to complete on conventional computers. Therefore, it is mandatory to utilize faster computing platforms.

Special-purpose computers for the acceleration of MD simulation gathered growing interest lately [17]. Field Programmable Gate Arrays (FPGAs) in particular have recently been proposed as a viable alternative implementation platform for MD simulation due to their flexible computing and memory architecture which gives them ASIC-like performance with the added programmability feature. Therefore, we chose FPGAs over ASICs as they offer reprogrammability, shorter development times and lower nonrecurring engineering (NRE) costs.

There are several MD simulation software tools. However, software for MD simulation can spend very high percentange of the total computation time in calculating the non-bonded interactions among particles because the computational complexity of the evaluation of non-bonded potentials or forces is quadratic. Therefore, we can accelerate MD simulation by porting the calculation of the non-bonded interactions from software to FPGAs since non-bonded interactions lend themselves to be easily calculated in parallel. On the other hand, the remaining MD calculation, which is complex but only consumes a very limited percentage of the total computation time, can be left to software running on a host computer. Our ultimate goal is to design and implement a MD simulation sytem that will allow scientists to simulate a biomolecular system within a reasonable time frame and obtain useful information of a biological system.

The design and implementation of an FPGA core that parallelises all the necessary operations to compute the non-bonded interactions in the Large-scale Atomic/Molecular Massively Parallel Simulation (LAMMPS) software tool is explained in this paper. Our MD processor core is comprised of 4 identical pipelines working independently in parallel to evaluate the non-

Authors are with the The University of Edinburgh, King's Buildings, Mayfield Road, Edinburgh, EH9 3JL, Scotland, UK (e-mail: s.kasap@ed.ac.uk; k.benkrid@led.ac.uk).

bonded potentials, forces and virials acting on a particle from all of the other particles in the simulated molecular system. A real hardware implementation of the designed core was achieved on the nodes of an FPGA-based supercomputer, called Maxwell, which consists of 64 Virtex-4 FPGA chips. Implementing our FPGA core on multiple nodes of Maxwell allowed us to produce a special-purpose parallel machine for the hardware acceleration of MD simulations. This machine is highly scalable, yielding higher computational power with the additional Maxwell nodes.

The remainder of this paper will first present essential background information on MD simulation and then discuss related prior works in the literature. Subsequently, LAMMPS MD simulation software will be introduced. After that, our implementation platform (the Maxwell FPGA-based supercomputer) will be illustrated and the general system architecture will be explained. Furthermore, the design and implementation of our FPGA core for computing the non-bonded interactions in a MD simulation will be elaborated. Following this, implementation results are presented and then evaluated comparatively with equivalent pure software implementations. Finally, conclusions are laid out with plans for future work.

2 Molecular Dynamics Simulation

Molecular Dynamics is commonly used for the simulation of the structural, thermodynamic and transport properties of large biological systems on a diverse range of timescales. In MD simulations, atoms in the system are treated as classical particles and are subject to covalent bond, Van der Waals and Coulomb forces from other particles. During a time-step of the MD simulation, forces are computed and accumulated on each atom due to its interaction with other atoms, and positions and velocities of atoms are updated by integrating the Newtonian equations of motion.

2.1 Molecular Interactions

In MD simulations of biological systems, the potential for a particle i, Φ_i, is modelled as follows:

$$\Phi_i = \Phi_i^B + \sum_{j \neq i} \epsilon_{ab} \left\{ \left(\frac{\sigma_{ab}}{|\boldsymbol{r}_{ji}|} \right)^{12} - \left(\frac{\sigma_{ab}}{|\boldsymbol{r}_{ji}|} \right)^{6} \right\} + q_i \sum_{j \neq i} \frac{q_j}{|\boldsymbol{r}_{ji}|} \quad (1)$$

where $\mathbf{r}_{ji}$ is a vector from the particle j to i and q_i is the charge of the particle i. The first term Φ_i^B is the bonded potential due to interactions within the topology of the molecules and is expressed as:

$$\Phi_i^B = \sum_{bonds} K_b (r - r_0)^2 + \sum_{angles} K_\theta (\theta - \theta_0)^2 + \sum_{dihedrals} K_{\phi_p} \left[1 + d_p \cos(n_p \phi) \right] \quad (2)$$

Bonded potential is written here as sums over simple harmonic 2-body (bond), 3-body (angle) and 4-body (dihedral) interactions although other potential models could also be used. On the other hand, the last 2 terms in (1) are the non-bonded potential due to interactions between all pairs of atoms in the system. Note that the forces exerted on the particle i, $\mathbf{f}_i$, are obtained by taking the gradient of (1) with respect to the position of the particle.

The second term in (1), which describes van der Walls interaction, is the Lennard-Jones (LJ) potential characterized by a length parameter σ_{ab} and an energy parameter ε_{ab} where a and b denote the two atom types of particles. If we take the gradient of this potential, an LJ force $\mathbf{f}_i^{LJ}$ can be expressed as:

$$\boldsymbol{f}_i^{LJ} = \sum_{j \neq i} \frac{\epsilon_{ab}}{\sigma_{ab}^2} \left\{ 12 \left(\frac{\sigma_{ab}}{|\boldsymbol{r}_{ji}|} \right)^{14} - 6 \left(\frac{\sigma_{ab}}{|\boldsymbol{r}_{ji}|} \right)^{8} \right\} \boldsymbol{r}_{ji} \quad (3)$$

The third term on the right hand side of (1) is the Coulombic (C) potential, and the corresponding Coulombic force $\mathbf{f}_i^C$ is expressed as:

$$\boldsymbol{f}_i^C = q_i \sum_{j \neq i} \left(\frac{q_j}{|\boldsymbol{r}_{ji}|^3} \right) \boldsymbol{r}_{ji} \quad (4)$$

The computational complexity of evaluating Φ_i^B is $O(1)$ since only few particles are covalently bonded to the i^{th} particle. However, the computation time to evaluate non-bonded potential or force functions is $O(N)$ for each particle where N is the number of particles in the simulated system. Hence, the computational complexity to evaluate the functions for all particles in the system is $O(N^2)$. Accelerating these evaluations is therefore the prime target for the design of our MD core. Note that our MD processor core will be able to deal with an arbitrary potential or force function although only the LJ and Coulombic interactions are mentioned in this section.

2.2 Cutoff Convention

The simplest method for reducing the computation time is the cutoff convention. Contributions from particles outside a certain cutoff radius r_c are ignored in this method and hence, the time complexity is reduced to $O(N)$. For instance, since LJ force and potential decrease rapidly with increasing distance (refer to (3)), the sum over j can be truncated within the determined cutoff distance so that only a few neighbours of atom i contribute rather than all N. This does not affect the results in most cases provided that the particles are well separated with respect to an appropriate value of r_c.

In contrast, the Coulombic interaction is long-range which means it decreases slowly with an increase of distance (refer to (4)). Hence, evaluating Coulombic force as a truncated sum over neighbours rather than as a full sum introduces large inaccuracies [7]. On the other hand, applying the latter method is problematic in periodic systems (briefly mentioned in subsection 2.4 below). Consequently, other methods are often used for the evaluation of Coulombic force and potential. One of these methods, namely the Ewald method, is discussed in subsection 2.5 and the one used by the LAMMPS software is explained in subsection 4.1.

2.3 Virials

Virials represent the effect of mutual interaction of particles on the pressure in the system. The virial $\mathbf{v}_i$ on the particle i can be calculated with the following equation where T denotes the transpose of the vector:

$$\boldsymbol{v}_i = \sum_{j \neq i} \boldsymbol{f}_{ji} \boldsymbol{r}_{ji}^T \quad (5)$$

Note that the time complexity of this operation for all particles is $O(N^2)$ since it is $O(N)$ for each particle. Our MD processor incorporates the computation of all components of each virial.

2.4 Periodic Boundary Conditions

MD simulations are generally performed under periodic boundary conditions where the original simulation cell is deemed to be surrounded by its 26 image cells [1]. Then, minimum image convention should be adopted in the calculations of pairwise interactions. This means that a force exerted on the particle i from the particle j is only to be calculated for the real particle j or nearest image of it to the particle i.

2.5 Ewald Method

In the cases where periodic boundary conditions apply and hence, electrically charged particles exist periodically, Coulombic forces can be calculated precisely by the Ewald method [8]. Force $\mathbf{f}_i^C$ is split into the sum of two rapidly converging series in the Ewald method as follows:

$$\boldsymbol{f}_i^C = \boldsymbol{f}_i^r + \boldsymbol{f}_i^m \quad (6)$$

where $\mathbf{f}_i^r$ is the real space sum and $\mathbf{f}_i^m$ is the reciprocal space sum. The real space sum is given in (7) where the positive parameter α is taken to be an appropriate value so that the $\mathbf{f}_i^r$ converges rapidly.

$$\boldsymbol{f}_i^r = \frac{q_i}{4\pi\epsilon_0} \sum_j q_j \left\{ \frac{2\alpha}{\sqrt{\pi}} exp\left(-\alpha^2 |\boldsymbol{r}_{ji}|\right) + \frac{erfc\left(\alpha |\boldsymbol{r}_{ji}|\right)}{|\boldsymbol{r}_{ji}|} \right\} \frac{1}{|\boldsymbol{r}_{ji}|^2} \boldsymbol{r}_{ji} \quad (7)$$

In (7), erfc is the complementary error function which is defined as:

$$erfc(x) = 1 - \frac{2}{\sqrt{\pi}} \int_0^x exp\,(-t^2)\, dt \quad (8)$$

Our MD processor can evaluate $\mathbf{f}_i^r$ according to (7) whose computation time is $O(N^2)$ for all particles since it is $O(N)$ for each particle. On the other hand, the computation of $\mathbf{f}_i^m$ is left to the software running on a host processor in our implementation.

2.6 Time Integration

There are various kinds of integrators to integrate Newtonian equations of motion, such as Verlet algorithm [9], Beeman algorithm [10], and multiple time-step algorithms [11]. One of the simplest and most popular algorithms for the time integration of the positions and velocities of particles is the Verlet algorithm which is expressed as the following two equations:

$$\boldsymbol{v}\left(t + \delta t/2\right) = \boldsymbol{v}\left(t - \delta t/2\right) + \delta t \boldsymbol{a}(t) \quad (9)$$

$$\boldsymbol{r}(t + \delta t) = \boldsymbol{r}(t) + \delta t \boldsymbol{v}\left(t + \delta t/2\right) \quad (10)$$

where $\mathbf{r}$ (t), $\mathbf{v}$ (t) and $\mathbf{a}$(t) are the position, velocity and acceleration vectors of a particle at time t, respectively and δt denotes the chosen size of each time-step. Note that the acceleration of a particle at a time-step is computed by the Newton's second law of motion:

$$\boldsymbol{a}_i = \frac{\boldsymbol{f}_i}{m_i} \quad (11)$$

3 Prior Work

Improving the performance of MD simulation software with fast computation algorithms or parallel algorithms such as atom decomposition, force decomposition and spatial decomposition was the primary focus of prior research on accelerating MD simulations. There exist a number of sophisticated MD software packages including GROMACS [12], [13], NAMD [14], [15] and LAMMPS [16]. In next section, the LAMMPS tool (a highly parallel MD simulator) will be introduced. However, since these software packages are limited by the the performance of a general purpose processor, some research has turned to special-purpose and application-specific hardware acceleration of the MD simulation. The main target of this new research topic is to speed-up the most computationally intensive portion of the MD simulation computation, namely the non-bonded interactions.

MD-GRAPE [17], [18] is one of the most prominent hardware acceleration systems for MD simulations. It uses a fourth order polynomial with 1024 piece to approximate the calculation of the force or potential where the coefficients determine which force or potential is calculated. MD-GRAPE which has a peak speed of 4.2 Gflops only accelerates the computation of the force and potential while leaving the rest of the MD simulation to a host processor. MD engine [19] was also a special-purpose computer for MD simulation which had system architecture similar to that of the MD-GRAPE system, where the host computer communicates with the special-purpose parallel machine that computes the non-bonded interactions.The MD engine system consists of 76 individual processors named MODEL each of which calculates both the Lennard-Jones and Coulombic interactions. The system can perform the simulation 50 times faster than an equivalent software implementation running on a Sun Ultra-2 200 MHz machine.

All of the aforementioned special-purpose hardware platforms for MD simulation were implemented using ASIC technology. However, hardware development in this way can take up several years before the application is fully implemented. On the other hand, recent advances have made FPGAs a viable platform for accelerating MD

simulations with substantial performance gains. Therefore, recent academic research has attempted to implement special-purpose computers for MD simulation using FPGAs.

Prior reserach on FPGA-based MD simulations have concentrated on accelerating different parts of the MD simulation. One of them mapped the position and velocity update to FPGA [20] while most of them computed LJ and Coulombic interactions of each time-step on FPGA [21], [22], [23], [24], [46]. On the other hand, only few ones moved all tasks in MD simulation onto FPGA [25], [26].

4 LAMMPS MD Simulation Software

LAMMPS is a classical molecular dynamics code written in C++, which stands for Large-scale Atomic/Molecular Massively Parallel Simulator [27]. It was developed at Sandia National Laboratories under the US department of Energy as a freely-available, open-source code, distributed under the terms of the GNU public license.

LAMMPS runs on single-processor machine although it was designed to run most efficiently on parallel computers supporting the MPI message-passing library, for instance on distributed- or shared-memory parallel machines and Beowulf-style clusters. LAMMPS can model atomic, polymeric, biological, metallic, granular and coarse-grained systems with only a few particles up to millions or billions using a variety of force fields and boundary conditions. However, it was designed to be easily modified or extended with new capabilities, such as new force fields, atom types or boundary condtions.

LAMMPS partitions the simulation domain into small 3D subdomains with spatial decomposition techniques on parallel machines. Each subdomain is assigned to a processor, and processors communicate and store ghost atom information for atoms that border their subdomain. By subdividing the physical volume among processors, most computations become local and communication is minimized so that optimal N/P scaling of the overall calculation can be achieved on P processors. Hence, the spatial-decomposition method is clearly the best algorithmic choice in comparison with atom decomposition and force decomposition methods both of which do not scale well to large numbers of processors. Note that systems with uniform particle density are most efficiently simulated by LAMMPS on parallel machines.

In the simplest sense, LAMMPS integrates Newton's equation of motion for particles interacting via short- or long-range forces. It utilizes neighbour lists to keep track of the nearby particles for each particle so that the short-range, non-bonded potentials and forces for all particles are computed efficiently using cutoff convention (see subsection 2.2) with time complexity of $O(N)$. As atoms move, these lists are reformed at every few time-steps, taking into consideration both owned and ghost atoms, with the utilization of a certain threshold radius (i.e. r_c + an offset) to determine the neighbouring particles for a particle.

There are several ways to enable the quick calculation of the Coulombic interactions by avoiding the all-pairs $O(N^2)$ computation. Approximate techniques include multipole methods [28], [29] scaling as N, Ewald summation (see subsection 2.5) scaling as $N^{3/2}$ and particle-particle particle-mesh method (PPPM) [30] scaling as N log $(N)^{1/2}$. PPPM which is a variant of particle-mesh Ewald (PME) method [31] is the method used by LAMMPS for the Coulombic computations due to its higher computational efficiency relative to other methods, particularly in parallel setting, as described in subsection 4.1.

Papers [32], [33] elaborate on the technical details of the algorithms used in LAMMPS.

4.1 PPPM Method

A detailed comparison of Ewald, multipole and PPPM methods shows that in addition to being less complex to implement, PPPM is the fastest for systems of any reasonable size [34]. The basic idea of PPPM is to replace the point charge Coulombic term in (1) with an equivalent expression for extended charges centered on the original atomic positions. Hence, Coulombic potential is now expressed as follows:

$$\Phi_i^C = \frac{q_i}{4\pi\epsilon_0}\sum_{j\neq i}\frac{q_j}{|\boldsymbol{r}_{ji}|}erfc\left(\frac{G|\boldsymbol{r}_{ji}|}{\sqrt{2}}\right) + \iint\frac{\hat{p}_i(r)\hat{p}_i(r')}{|r-r'|}drdr' - \frac{G}{\sqrt{2\pi}}q_i^2 \quad (12)$$

where $\hat{p}_i(r)$ is the Gaussian density that represents an extended charge and is given as follows:

$$\hat{p}_i(r) = q_i\left(\frac{G^2}{\pi}\right)^{3/2} exp(-G^2(r-r_i)^2) \quad (13)$$

The first term in (12) is the usual Coulombic potential multiplied by a complementary error function which forces it to go to nearly zero at a user-specified cutoff distance r_c, where G is determined by the accuracy criterion. Thus, this term is the short-range portion of the Coulombic interaction and is computed in LAMMPS at the same time as van der Waals interactions as a sum over nearby particles utilising neighbour lists. On the other hand, the second term in (12) is the Coulombic potential due to the interaction of the extended charges whereas the last term is a constant.

5 The Maxwell Supercomputer

Maxwell [35] is an FPGA based supercomputer developed by the FPGA High Performance Computing Alliance (FHPCA) in Scotland [36] to run computationally demanding applications on an array of FPGAs at low energy budgets. Its physical architecture, logical structure and software environment are briefly discussed in subsections 5.1, 5.2 and 5.3, respectively.

5.1 Physical Architecture

Maxwell comprises two 19-inch racks and five IBM blade centres, four of which have seven IBM Intel Xeon

blades and the fifth has four (32 blades in total). The blades are booted over the network from the head node (Dell server). Furthermore, each blade is a diskless 2.8 GHz Xeon with 1 Gbyte memory which hosts 2 Xilinx Virtex-4 FPGAs through a PCI-X expansion module. Thus, Maxwell comprises 64 FPGAs having 512 or 1024 MB off-chip memory and four MGT Rocket IO connectors which can run at 2.5 Gb/s. Furthermore, the FPGAs are mounted on 2 different types of plug-in PCI card, namely Alpha Data ADM-XRC-4FX [37] and Nallatech HR101 [38]. Both types of card connect to the Xeon on a particular blade using a PCI/PCI-X bridge which is capable of 64 bit, 133 MHz operation in PCI-X mode, giving a peak bandwidth of 1064 MB/s.

Maxwell has three independent communications networks for CPU-CPU, CPU-FPGA, and FPGA-FPGA communications. The blade CPUs are networked over gigabit Ethernet through a single 48-way Netgear switch with 40 Gb/s throughput. Thus, CPUs have an all-to-all connectivity. The FPGA network consists of point-to-point links between the MGT connectors of adjacent FPGAs. Each FPGA has 4 RocketIO links enabling the 64 FPGAs to be connected together in a two-dimensional 8 x 8 torus as illustrated in fig. 1. Finally, FPGAs and CPUs can communicate with each other over the PCI bus as mentioned above.

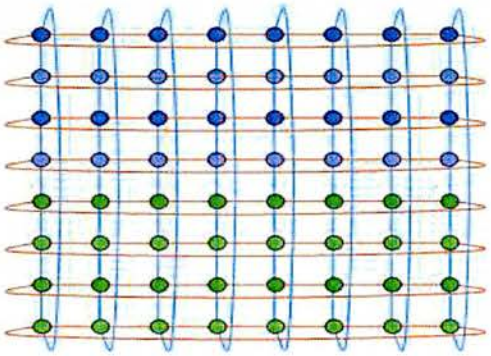


Fig. 1. FPGA connectivity in Maxwell [35]

5.2 Logical Structure

Logically, Maxwell can be regarded as a collection of 64 nodes, where a node is defined as a software process running on a host CPU together with some FPGA acceleration hardware as illustrated in fig. 2. In the typical case of 64 nodes configuration, each blade CPU hosts two software processes each of which manages one of the two FPGAs on the blade.

5.3 Software Environment

The software environment of Maxwell comprises Linux variant CentOS, standard GNU/Linux tools, Sun Grid Engine (SGE) as the batch scheduling system, MPI for inter-process communication and most importantly the FHPCA Parallel Toolkit (PTK) [39] that forms a bridge from the application process to the FPGA process (see fig. 2). Essentially, the PTK is a set of practices and infrastructure written mostly in C++ that aims to address acceleration issues such as associating processes with FPGA resources, associating FPGAs with bitstreams, managing contention for FPGA resources within a process and managing code dependencies to facilitate re-use.

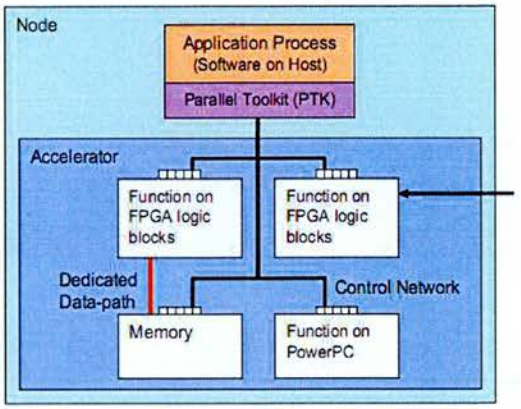


Fig. 2. Logical structure of the Maxwell [40]

6 System Architecture

Our special-purpose parallel machine for MD simulations is a reconfigurable hardware accelerator plugged into a number of host CPUs. Fig. 3 illustrates the basic process flow in our machine for each time-step. LAMMPS MD simulation software (see section 4) running on a general purpose computer first initialises the simulation environment, calculates the less time-consuming bonded interactions and builds a neighbour list for each particle i in the simulated system, which includes all nearby j particles within a certain radius of the particle i. Then, for each neighbour list, software on host CPUs first broadcast the coordinates and electric charge of the particle i to the reconfigurable hardware and subsequently the coordinates and electric charge of each j particle in the neighbour list as well as the interaction parameters and the cutoff distances for the specific pairs of i and j particles are sent to the reconfigurable hardware one by one as shown in fig. 3. Following this, our parallel machine computes all non-bonded forces, virials and potentials acting on each particle i due to the j particles in its neigbour list and then, send these pairwise values back to the host CPU. Software running on the host use the force values to calculate the acceleration of each particle in the simulated system by (11) and then integrates Newtonian equations of motion (see subsection 2.6) by an integration technique to update the velocity and position values of all particles at the current time-step. Software also adds-up pairwise potential and virial values to compute the total per-atom potentials and virials, respectively. The total potential energy and pressure in the simulated system at the current time-step are also calculated by accumulating these potential and virial values, respectively. Note that a new C++ class was written for the LAMMPS software using the FHPCA Parallel Toolkit (PTK) to be able to cooperate with the reconfigurable hardware for MD simulations in the way explained above. Another important point is that all transfers between host CPU and reconfigurable hardware are done with Direct Memory Access (DMA) method.

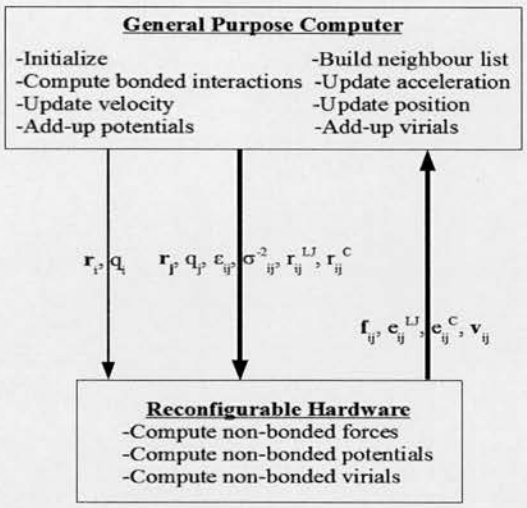


Fig. 3. Basic structure of our special-purpose parallel machine for Molecular Dynamics simulations

Fig. 4 shows the system connection diagram of our special-purpose parallel machine for MD simulations. Two processes of LAMMPS software run on each Intel Xeon CPU while an instance of our MD processor core resides in each user FPGA. Actually, a software process running on a host CPU and a hardware core in a user FPGA form a Maxwell node as described in subsection 5.2. The number of utilized Maxwell nodes where LAMMPS processes communicate with each other by Message Passing Interface (MPI) [41] can be easily configured as desired.

Each Xeon CPU on PCI-X bus connects to two user FPGAs through bridge/control FPGAs mediating communication between the 32-bit wide PCI-X bus operating at 133 MHz and the 32-bit wide local buses of the user FPGAs operating at 80 MHz, as shown in fig. 4. Furthermore, user FPGAs in our MD machine are of Xilinx Virtex-4 FX-100 type whereas smaller FPGAs bridging PCI-X and local buses are of Xilinx Virtex-4 LX25 type. On the other hand, four 256 MB DDR2 SDRAMs are connected to each user FPGA. The physical width and depth of the SDRAMs are 32 bits and 64M words, respectively while the logical width of the SDRAMs is 128 bits. Note that the MD processor core in a user FPGA runs at 150 MHz although the logic interfacing the user FPGA to the local bus it is connected to runs at 80 MHz.

Fig. 5 shows the block diagram of our MD processor core. As it can be seen, our MD core incorporates 4 identical MD pipelines which are working independently in parallel to evaluate the non-bonded potentials, forces and virials acting on a particle from each of the particles in the neighbour list of that particle. Each MD processor is associated with one of the SDRAM banks connected to the user FPGA. Furthermore, the LAMMPS process running on a host CPU transfers the simulation-related data mentioned above to the allocated first region of each SDRAM bank to be processed by the relevant MD processor. When these incoming transfers complete, the software process signals each MD processor to start its operation of reading data from its associated SDRAM bank through the use of an input buffer and then writing the evaluated potential, force and virial values back to the allocated second region of the associated SDRAM bank through the use of an output buffer, all under control of a Finite State Machine (FSM) as shown in fig. 5. When the MD processor is done with its operation, it signals the software process to transfer its computed values back from the relevant SDRAM bank for further processing as explained above. Moreover, two identical function coefficient memories shown in the fig. 5 store the coefficients of the interpolation used to evaluate a number of functions as will be detailed later in the next section in conjunction with the inner architecture and operation flow of our designed MD processor.

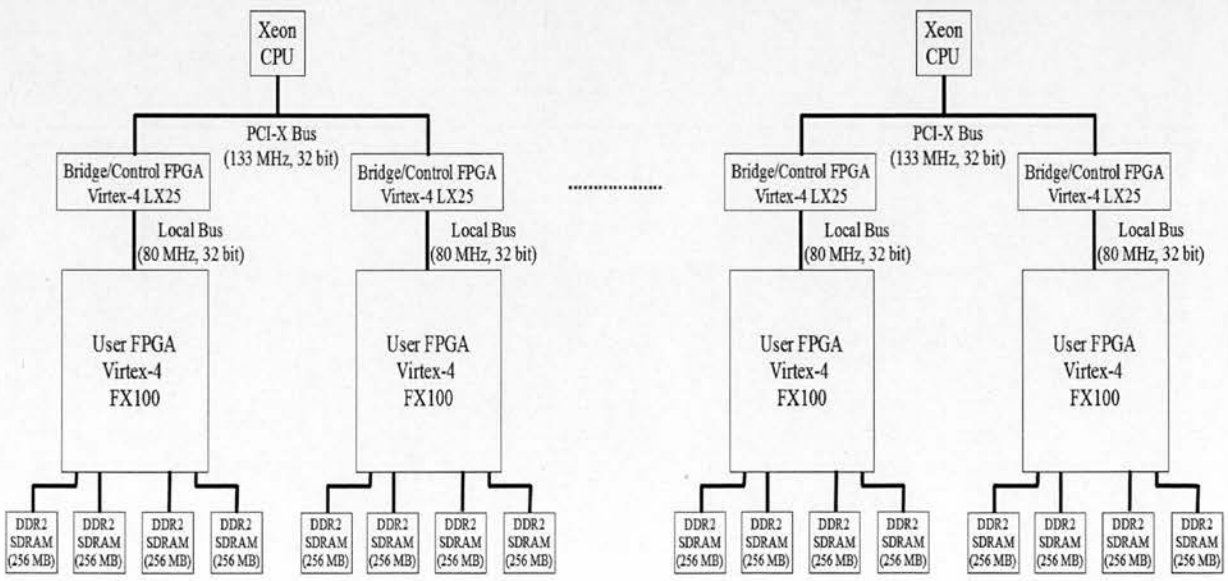


Fig. 4. Sytem connection diagram of our special-purpose parallel machine for Molecular Dynamics simulations

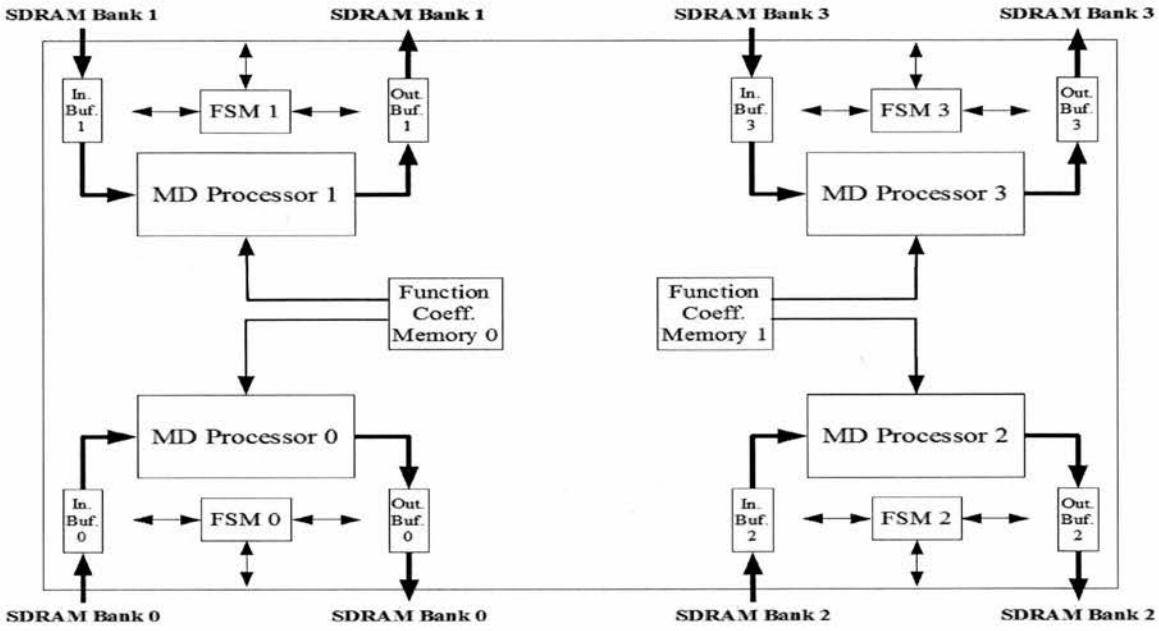


Fig. 5. Block diagram of our Molecular Dynamics processor core

7 Design of Molecular Dynamics Processor

In our design, internal format of the numbers used is the IEEE standard single precision (i.e. 32-bit wide) floating-point as shown in fig. 6 (a). All data are handled in this format. Hence, single precision floating-point arithmetic units are utilized throughout our MD processor. Several pipelined floating-point multipliers and adders/subtractors obtained from [42] are incorporated in our design whose operation latencies are 4 and 6 clock cycles, respectively, as explained in [43]. These arithmetic units do not support denormalized numbers and NaN ("not a number") to minimize the required hardware resources and realize high operation speed by simplifying the circuitry.

SDRAM banks in our MD machine were partitioned into two regions. The first region of a SDRAM bank was allocated to data transfers from host CPU while the second region was allocated to data transfers from the designated MD processor on a user FPGA, as mentioned in section 6. Fig. 6 (b) shows the layout of a memory portion in the first region of a SDRAM bank storing the coordinates $\mathbf{r}_i = (r_x, r_y, r_z)$ and electric charge q_i of a particle i whereas fig. 6 (c) shows the layout of another memory portion in the first region of a SDRAM bank storing the coordinates $\mathbf{r}_j = (r_x, r_y, r_z)$ and electric charge q_j of a j particle, as well as the interaction parameters ε_{ij}, σ^{-2}_{ij} and the cutoff distances for both Lennard-Jones r^{LJ}_{ij} and Coulombic r^{C}_{ij} interactions, all pertaining to the particular pair of i and j particles. Since the logical width of the memory banks is 128 bits, the layouts in fig. 6 (b) and (c) occupy the space of 1 and 2 logical words, respectively. Furthermore, the layout of a memory portion in the second region of a SDRAM bank storing the force $\mathbf{f}_{ij} = (f_x, f_y, f_z)$, Lennard-Jones potential e^{LJ}_{ij}, Coulombic potential e^{C}_{ij} and virial $\mathbf{v}_{ij} = (v_x^2, v_y^2, v_z^2, v_{xy}, v_{xz}, v_{yz})$ values computed for the specific pair of i and j particles is displayed in fig. 6 (d) which takes up the space of 3 logical words in a memory bank.

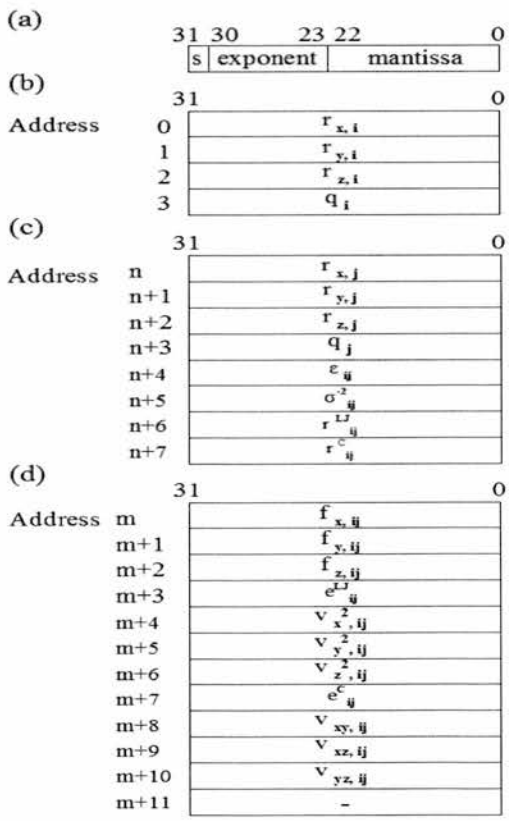


Fig. 6. (a) Internal format of the numbers used in our design (b) Layout of a memory portion in the first region of a SDRAM bank storing the coordinates and electric charge of a particle i (c) Layout of another memory portion in the first region of a SDRAM bank storing the coordinates and electric charge of a j particle as well as the interaction parameters and the cutoff distances for the particular pair of i and j particles (d) Layout of a memory portion in the second region of a SDRAM bank storing the force, potential and virial values computed for the specific pair of i and j particles

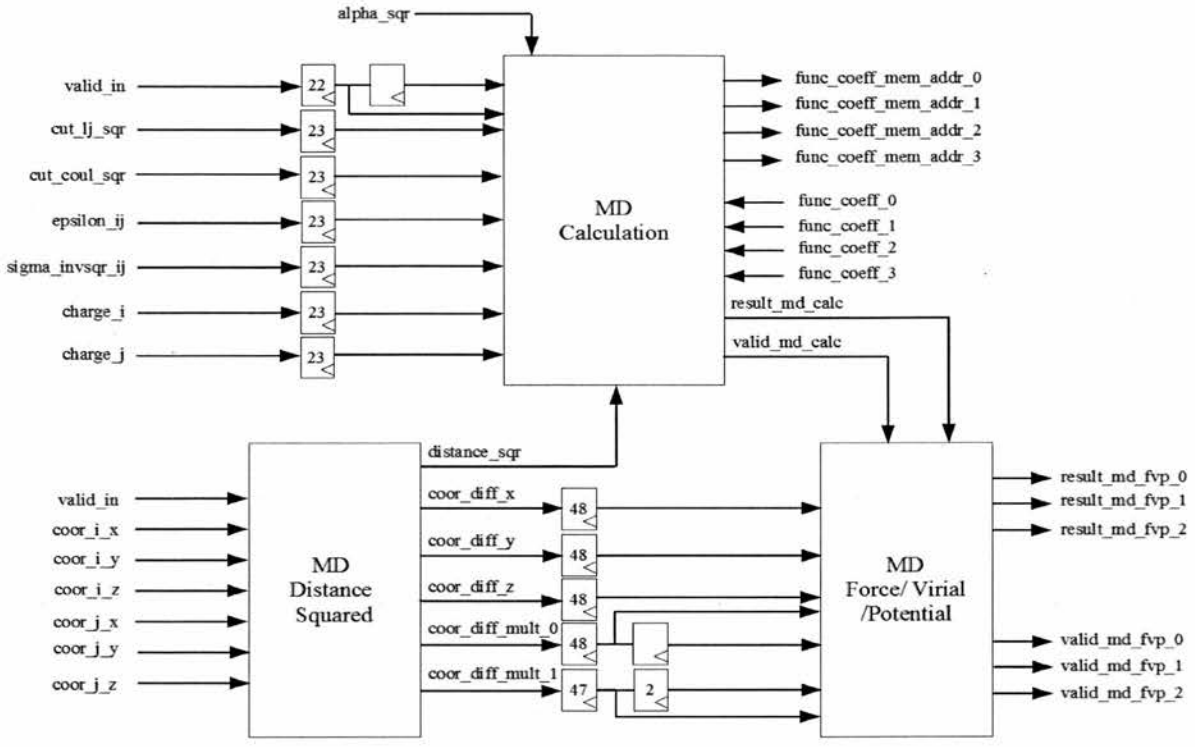


Fig. 7. Functional block diagram of a Molecular Dynamics processor

Fig. 7 above shows the functional block diagram of our MD processor. It contains a pipeline comprised of three major functional units, namely *MD Squared Distance* unit, *MD Calculation* unit and *MD Force/Virial/Potential* unit in the order presented. MD processor whose operating frequency is 150 MHz calculates the non-bonded interactions in the simulated molecular system as stated above. The detailed architectures and operations of the three functional units in the MD processor will be described in subsections 7.1, 7.2 and 7.3, respectively.

7.1 MD Distance Squared Unit

Fig. 8 shows a simplified pipeline architecture of the first functional unit in our MD processor, *MD Squared Distance* unit whose primary duty is to calculate the squared distance between an i particle and the j particles in the neighbour list of that i particle. When a MD processor is triggered by the host CPU, it begins to transfer simulation data of i and j particles into its input buffer (see fig. 5) from the first region of its associated SDRAM bank. Input buffers in our design make use of double buffering so as to enhance the efficiency of the data transfers from a memory bank and hence, increase the operation speed of the MD processors. Note that each of these double buffers has a width of 256 bits, so two logical words are transferred one by one from a SDRAM bank to make up one word of the buffer.

When an input buffer is completely full with data, the *MD Squared Distance* unit starts to read one word from the buffer every four clock cycles. If it is detected that the read word contains the coordinates and electric charge of an i particle, those coordinates are registered separately in the unit (not shown in fig. 8) whereas the charge value is shifted towards the *MD Calculation* unit in a register array as shown in fig. 7. On the other hand, if the read word contains data related to a j particle, coordinate values in that word are registered and then pushed into the pipeline with the *valid_in* signal asserted for four clock cycles, while the rest of the data in the word (see fig. 6 (c)) are seperately shifted towards the *MD Calculation* unit in five register arrays.

When the coordinate values of a j particle enters the pipeline, three floating-point subtractors are used to compute the coordinate differences between the registered i particle and that j particle in three dimensions, $\mathbf{d}_{ij} = (d_x, d_y, d_z)$, independently in parallel. These coordinate differences are shifted separately towards the end of the unit in three register arrays to be passed to the *MD Force/Virial/Potential* unit. Furthermore, two floating-point multipliers compute the two squared coordinate differences in x and y dimensions, d_x^2, d_y^2, and the squared coordinate difference in z dimension, d_z^2, in different clock cycles through the use of the three multiplexers whose control signal values are determined depending on the value of the 2-bit counter *valid_cnt* which increments by one with the high value of the delayed *valid_in* signal as shown in fig. 8. Table 1 shows how the values of the control signals for the multiplexers vary depending on the value of the counter *valid_cnt*. With these control signal values, two floating-point multipliers also compute the following products of the coordinate differences: $d_x d_y$, $d_x d_z$ and $d_y d_z$ in addition to d_x^2, d_y^2 and d_z^2 in an order dictated by the multiplexers. Moreover, all of these coordinate difference products are shifted towards the end of the unit in two register arrays to be passed to the *MD Force/Virial/ Potential* unit for further computations.

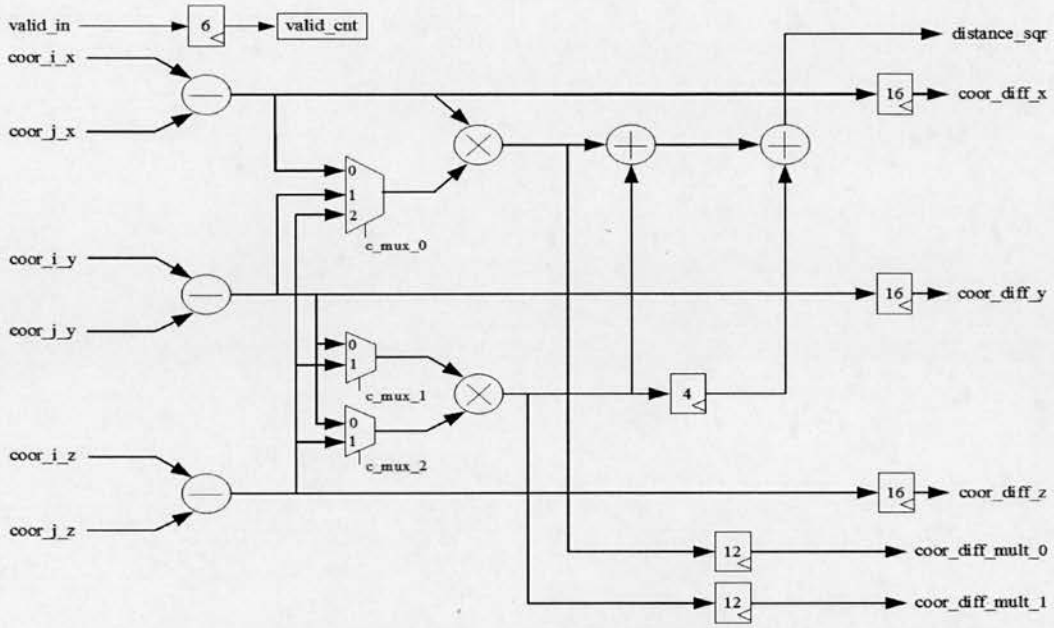


Fig. 8. Simplified pipeline architecture of the MD Distance Squared unit

TABLE 1
CONTROL SIGNAL VALUES FOR THE THREE MULTIPLEXERS IN THE MD DISTANCE SQUARED UNIT

valid_cnt	0	1	2	3
c_mux_0	0	1	2	X
c_mux_1	0	0	1	X
c_mux_2	0	1	1	X

Finally, the first floating-point adder in the unit computes the sum of the squared coordinate differences in x and y dimensions, $d_x^2 + d_y^2$, while the second floating-point adder adds the squared coordinate difference in z dimension, d_z^2, to this sum to calculate the squared distance, $r_{ij}^2 = d_x^2 + d_y^2 + d_z^2$, between a pair of i and j particles. Furthermore, this value is passed to the *MD Calculation* unit to evaluate several functions of distance. Note that the pipeline latency of the *MD Squared Distance* unit is 22 clock cycles.

7.2 MD Calculation Unit

Fig. 9 shows the simplified pipeline architecture of the second and largest functional unit in our MD processor, *MD Calculation* unit whose primary duty is to calculate and separately multiply the first two terms in (14) and (15), and both terms in (16) and (17). The multiplied terms are then passed to the *MD Force/Virial/Potential* unit for the computations of the pairwise forces, virials and potentials due to both Lennard-Jones and Coulombic interactions between an i particle and the j particles in the neighbour list of that i particle. Note that only the short-range portion of the Coulombic interactions is computed in our MD processor.

$$\boldsymbol{f}_{ij}^{LJ} = \frac{\epsilon_{ij}}{\sigma_{ij}^2} \cdot g_1\left(\frac{r_{ij}^2}{\sigma_{ij}^2}\right) \cdot \boldsymbol{r}_{ij} \quad (14)$$

$$\boldsymbol{f}_{ij}^{C} = q_i q_j \cdot g_2\left(\alpha^2 r_{ij}^2\right) \cdot \boldsymbol{r}_{ij} \quad (15)$$

$$\Phi_{ij}^{LJ} = \epsilon_{ij} \cdot g_3\left(\frac{r_{ij}^2}{\sigma_{ij}^2}\right) \quad (16)$$

$$\Phi_{ij}^{C} = q_i q_j \cdot g_4\left(\alpha^2 r_{ij}^2\right) \quad (17)$$

When the squared distance between a pair of i and j particles is passed to the *MD Calculate* unit by the *MD Squared Distance* unit, this value is registered to be valid for four clock cycles with the asserted *valid_in* signal. Then, two floating-point comparators in the unit compare the registered squared distance with the specified cutoff distances for the Lennard-Jones and Coulombic interactions, respectively. If the squared distance happens to be bigger than the cutoff distance for any interaction, then the forces, virials and potentials due to that interaction are set to be zero for the particular pair of i and j particles. Furthermore, utilizing the values shifted into the unit as shown in fig. 7, the first terms in (14), (15), (16) and (17) are computed to be available at the output of the multiplexer *mux_3* in four consecutive clock cycles through the use of the floating-point multiplier *mult_0* and the two multiplexers, *mux_0* and *mux_1*, shown in fig. 9. Control signal values of the mentioned multiplexers and the multiplexer *mux_2* are determined depending on the value of the 2-bit counter *valid_cnt* as shown in Table 2.

TABLE 2
CONTROL SIGNAL VALUES FOR THE FOUR MULTIPLEXERS IN THE MD CALCULATOR UNIT

valid_cnt	0	1	2	3
c_mux_0	0	1	0	1
c_mux_1	0	1	0	1
c_mux_2	0	1	0	1
c_mux_3	0	0	1	0

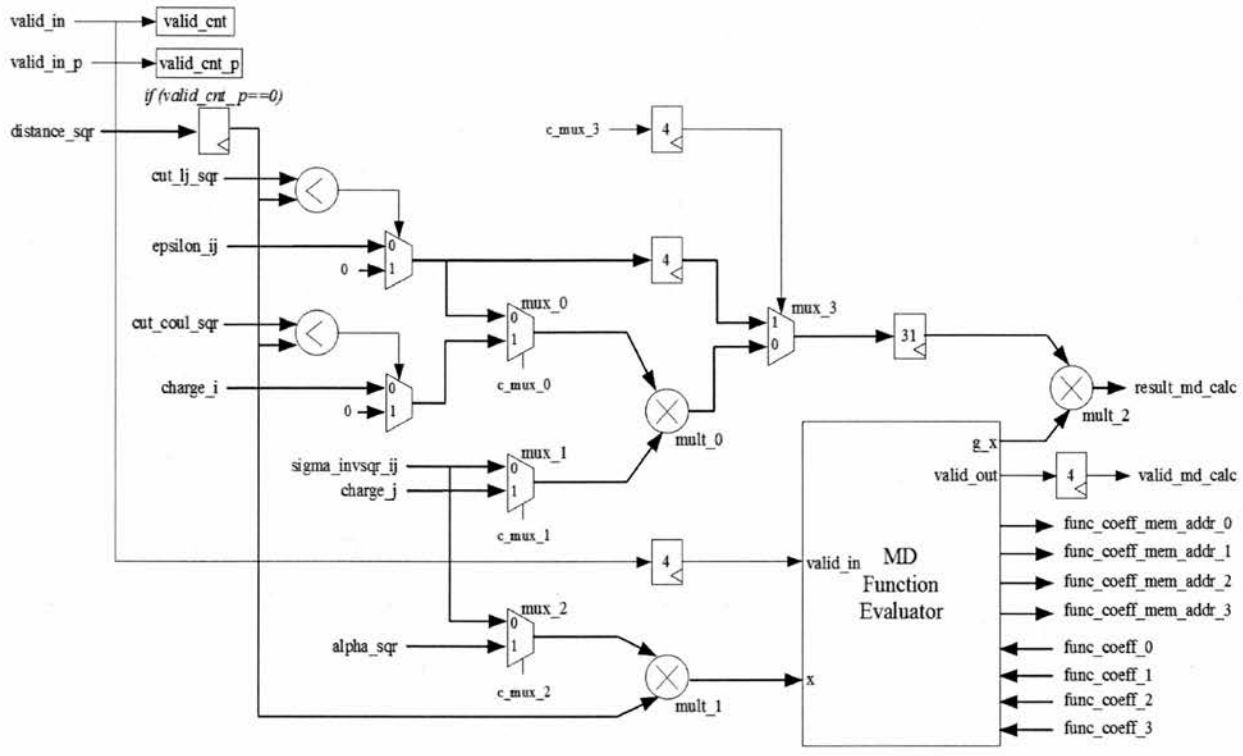


Fig. 9. Simplified pipeline architecture of the MD Calculation unit

On the other hand, the floating-point multiplier *mult_1* computes the arguments of the functions $g_1(x)$, $g_2(x)$, $g_3(x)$ and $g_4(x)$, which are in the second terms of (14), (15), (16) and (17), respectively, in four consecutive clock cycles through the use of the multiplexer *mux_2*. These computed arguments are then passed to the *MD Function Evaluator* unit to be evaluated in their corresponding function in a pipelined manner with a latency of 31 clock cycles. The *MD Function Evaluator* unit will be elaborated in subsection 7.2.1. Moreover, the functions $g_1(x)$, $g_2(x)$, $g_3(x)$ and $g_4(x)$ are expressed below:

$$g_1(x) = 48x^{-7} - 24x^{-4} \quad (18)$$

$$g_2(x) = erfc(\sqrt{x})x^{-3/2} + exp(-x)x^{-1} \quad (19)$$

$$g_3(x) = 4x^{-6} - 4x^{-3} \quad (20)$$

$$g_4(x) = erfc(\sqrt{x})x^{-1/2} \quad (21)$$

Finally, the floating-point multiplier *mult_2* multiplies the delayed output of the multiplexer *mux_3* and the output of the *MD Function Evaluator* unit, and hence gets the first two terms in (14) and (15) and both terms in (16) and (17) multiplied in 4 consecutive clock cycles for a pair of i and j particles. These results are then sent to the *MD Force/Virial/Potential* unit with the asserted *valid_md_calc* signal for further processing. Note that the pipeline latency of the *MD Calculator* unit is 40 clock cycles.

7.2.1 MD Function Evaluation Unit

Fig. 11 shows the simplified pipeline architecture of a functional unit in our *MD Calculation* unit, namely *MD Function Evaluation* unit which evaluates the functions $g_1(x)$, $g_2(x)$, $g_3(x)$ and $g_4(x)$, which are expressed respectively in (18), (19), (20) and (21), consecutively in four clock cycles using the piecewise third-order polynomial interpolation. When the argument x enters the unit with the asserted *valid_in* signal, it is decomposed into two numbers, x_{addr} and x_{interp}, as shown in fig. 10.

The floating-point number x_{addr} represents the argument x in a range $[2^{-5}, 2^{11})$ using 10 bits, 4 for the exponent and 6 for the mantissa. The latter is the copy of the most significant bits of the mantissa of x, namely m_x. The unit calculates the value of $x - x_{addr} \cdot 2^{-e_a + e_x}$, where e_a and e_x are the exponents of x_{addr} and x, respectively, and normalizes it in a 32-bit floating-point number x_{interp}. Actually, the mantissa of x_{interp}, namely e_i, is equal to e_x-7-n, where n is the number of leading zeros in the 17 least significant bits of x. Then, a function g(x) can be approximated with x_{interp} as follows, where c_3, c_2, c_1 and c_0 are the coefficients of the piecewise polynomial interpolation:

$$g(x) = \left((c_3 x_{interp} + c_2)x_{interp} + c_1\right)x_{interp} + c_0 \quad (22)$$

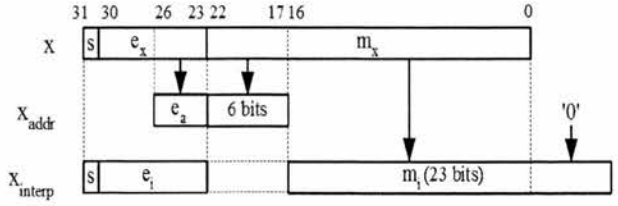


Fig. 10. The operation required for the polynomial interpolation with a look-up table to evaluate the functions $g_1(x)$, $g_2(x)$, $g_3(x)$ and $g_4(x)$.

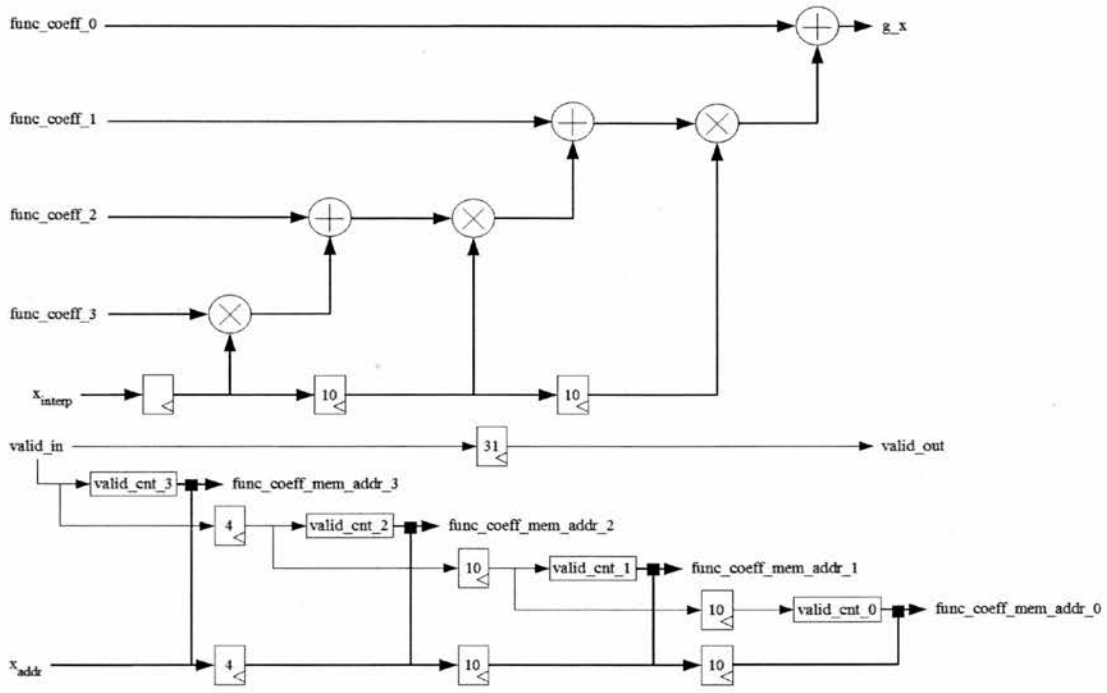


Fig. 11. Simplified pipeline architecture of the MD Function Evaluator unit

A set of the quadruples of the coefficients (i.e. a look-up table) is stored in the two identical *Function Coefficients* memories shown in fig. 5. Note that each *Function Coefficients* memory serves two separate MD processors with its dual port. These memories are comprised of four sub-memories storing one of the four piecewise interpolation coefficients (i.e. c_3, c_2, c_1, c_0) for four functions (i.e. $g_1(x)$, $g_2(x)$, $g_3(x)$, $g_4(x)$) in four separate regions. Each sub-memory is a 4096 x 32 bits Block RAM with an address width of 12 bits. The layout of a sub-memory in the *Function Coefficients* memory is shown in fig. 12.

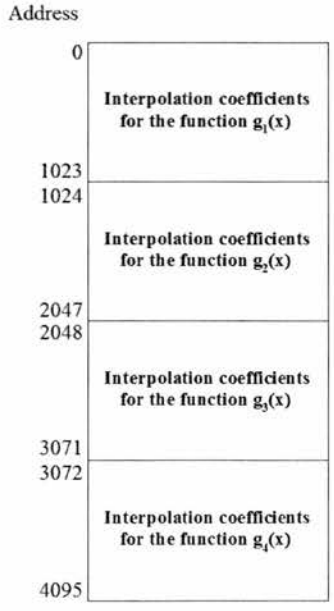


Fig. 12. The layout of a sub-memory in the Function Coefficients memory storing one of the four piecewise interpolation coefficients for the functions $g_1(x)$, $g_2(x)$, $g_3(x)$ and $g_4(x)$ in four separate regions.

Furthermore, four differently delayed values of the 10-bit number x_{addr} are used as part of the addresses for accessing the four sub-memories individually, as shown in fig. 11. On the other hand, the two most significant bits of the four addresses, coming respectively from the four 2-bit counters in the unit, are used to select one out of four tables (functions) stored in the sub-memories. With this configuration, *MD Function Evaluation* unit evaluates the functions $g_1(x)$, $g_2(x)$, $g_3(x)$ and $g_4(x)$ for a pair of i and j particles in four consecutive clock cycles with a latency of 31 clock cycles.

7.3 MD Force/Virial/Potential Unit

Fig. 13 shows the simplified pipeline architecture of the third and final functional unit in our MD processor, *MD Force/Virial/Potential* unit whose primary duty is to compute the pairwise virials and forces acting on an i particle due to both Lennard-Jones and Coulombic interactions with the j particles in the neighbour list of that i particle. In four consecutive clock cycles, the multiplied first two terms of (14), (15), (16) and (17) are pushed one by one into the unit by the *MD Calculation* unit with the *valid_in* signal asserted for four clock cycles. Obviously, these four products pertain to the LJ force, Coulombic force, LJ potential and Coulombic potential for a pair of i and j particles, respectively.

The first and second values entering the unit are registered separately in the first two clock cycles under the control of the 2-bit counter *valid_cnt*, which increments by the high value of the *valid_in* signal, as shown in fig. 13. These registered values are then added up by the floating-point adder in the unit and the result is passed to the three floating-point multipliers. On the other hand, the third and fourth values are buffered respectively in the synchronous write, asynchronous read buffers *fifo_elj* and

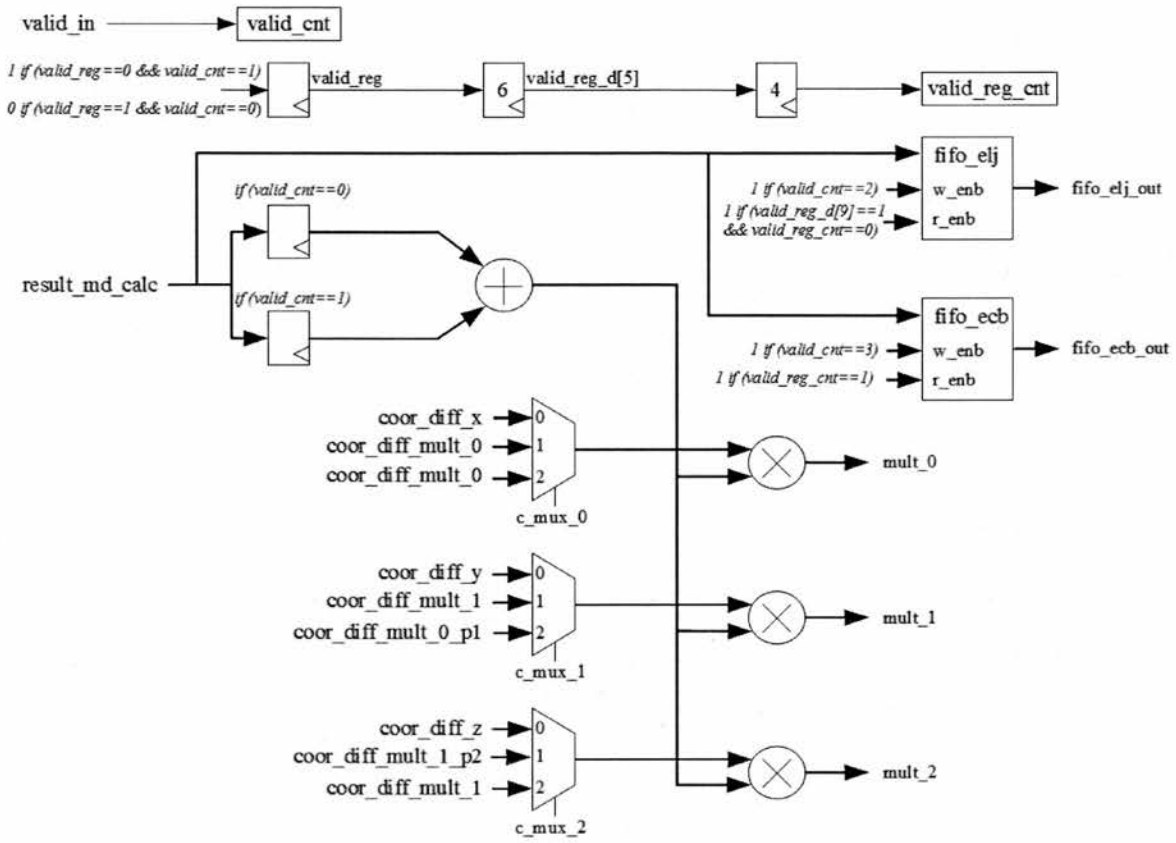


Fig. 13. Simplified pipeline architecture of the MD Force/Virial/Potential unit

fifo_ecb in the third and fourth clock cycles under the control of the counter *valid_cnt*. These bufferings last until the end of the operation of these multipliers, as will be explained later.

Furthermore, the three floating-point multipliers compute all three components of the total pairwise force $\mathbf{f}_{ij}$ (i.e. f_x, f_y, f_z) and six components of the pairwise virial $\mathbf{v}_{ij}$ (i.e. v_x^2, v_y^2, v_y^2, v_{xy}, v_{xz}, v_{yz}) in parallel in three consecutive clock cycles by multiplying the output of the floating-point adder with the coordinate differences (i.e. d_x, d_y, d_z) and the coordinate difference products (i.e. d_x^2, d_y^2, d_z^2, d_xd_y, d_xd_z, d_yd_z) ,which are shifted into the pipeline from the *MD Squared Distance* unit for a pair of i and j particles, through the use of three multiplexers, as shown in fig. 13. In the first clock cycle, the output of the adder is multiplied by the coordinate differences, d_x, d_y and d_z, to calculate the components of the total pairwise force, f_x, f_y, f_z, whereas the adder output is multiplied by the following coordinate difference products: d_x^2, d_y^2 and d_z^2 to compute the following three components of the pairwise virial: v_x^2, v_y^2 and v_y^2 in the second clock cycle. Finally, in the third clock cycle, the following coordinate difference products: d_xd_y, d_xd_z and d_yd_z are multiplied by the output of the adder to calculate the following three components of the pairwise virial: v_{xy}, v_{xz} and v_{yz}. Note that the control signal values of the three multiplexers in the unit are determined depending on the value of the 2-bit counter *valid_cnt_2* (not shown in fig. 13) which counts up to three with the high value of the *valid_reg_d[5]* signal. Table 3 shows how the values of the control signals for the multiplexers vary depending on the value of the counter *valid_cnt_2*.

TABLE 3
CONTROL SIGNAL VALUES FOR THE THREE MULTIPLEXERS IN THE MD FORCE/VIRIAL/POTENTIAL UNIT

valid_cnt_2	0	1	2
c_mux_0	0	1	2
c_mux_1	0	1	2
c_mux_2	0	1	2

As the multipliers finish their operations, their outputs *mult_0*, *mult_1* and *mult_2* are concatenated into a word which is written to the output buffer of the MD processor in three consecutive clock cycles, as shown in fig. 5. Furthermore, the pairwise LJ potential e^{LJ} in the corresponding location of the *fifo_elj* buffer and the pairwise Coulombic potential e^{C} in the corresponding location of the *fifo_ecb* buffer are incorporated into that word in the first and second clock cycles, respectively, extending its width to 128 bits. When an output buffer is completely full with data, its content is flushed into the second region of the associated SDRAM bank. Layout of a memory portion in the second region of a SDRAM bank is shown in fig. 6 (d). Moreover, the 128-bit output buffers in our design make use of double buffering so as to enhance the efficiency of the data transfers to a memory bank and hence, increase the operation speed of the MD processors. Note that the pipeline latency of the *MD Force/Virial/Potential* unit is 12 clock cycles.

8 Implementation Results

Molecular Dynamics simulations were implemented on the Alpha Data nodes of the Maxwell machine with the MD processor cores shown in fig. 5, each of which incorporating four MD processors working independently in parallel with a total pipeline latency of 74 clock cycles. Our MD core was written in Verilog language while the interfaces of the user FPGA with the local bus and the DDR2 SDRAM banks were provided by the Alpha Data in the VHDL language. The design was then synthesized, placed, and routed by the Xilinx ISE 11.5 tool. FPGA bitstreams were also generated by the same tool while the ModelSim tool was employed to test the MD core with a number of testbenches. Note that there is only one FPGA bitstream used to configure all FPGAs in the MD machine regardless of the number of atoms in the simulated system. Furthermore, MATLAB tool was used to compute the piecewise polynomial interpolation coefficients for the evaluation of the several functions needed, as explained in subsection 7.2.1.

The clock frequency of the user FPGAs for the local bus interface was set to be 80 MHz whereas the clock frequency for the MD core was set to be 150 MHz. Due to this clock frequency of the MD core, the clock frequency for the DDR2 SDRAM banks was 300 MHz. For benchmarking purposes, an all-atom Rhodopsin protein in solvated lipid bilayer was simulated with the Lennard-Jones forces, and the Coulombic forces via PPPM (particle-particle particle mesh), incorporating SHAKE constraints. This model contains counter-ions and a reduced amount of water to make a 32K atom system. The details of the simulation are as follows:

- 32,000 atoms for one time-step
- LJ and Coulombic force cutoff of 10.0 Angstroms
- Neighbor skin of 2.0 Angstroms
- Average neighbors per atom = 372 atoms
- NVT time integration

Table 4 presents the timing performance figures of the LAMMPS software for the pairwise LJ and short-range Coulombic interaction computations of the above mentioned Rhodopsin protein system on two nodes of the Maxwell machine (i.e. two software processes running on one host Intel Xeon CPU). The protein system was replicated in X, Y or Z dimensions to achieve the simulation of systems with up to 256,000 atoms, as presented in table 4.

TABLE 4
Timing Figures of the LAMMPS Software for the Pairwise Interaction Computations on Two Maxwell Nodes

No. of Atoms	No. of Bonds	No. of Angles	No. of Dihedrals	Computation Time (s)
32000	27723	40467	56829	5.051342
64000	55446	80934	113658	9.911402
128000	110892	161868	227316	20.407846
256000	221784	323736	454632	40.746851

For comparative purposes, table 5 below shows the timing performance figures of the MD machine configured to operate in the same way as the pure software implementation on two nodes of the Maxwell machine (i.e. two software processes running on one host Intel Xeon CPU and MD core instances on two Xilinx Virtex-4 XC4VFX100 FPGAs [44]). Note that the timing figures presented in table 5 do not include the I/O communication costs occurring during the data transfers between a host CPU and SDRAM banks.

TABLE 5
Timing Figures of the MD Machine for The Pairwise Interaction Computations on Two Maxwell Nodes

No. of Atoms	No. of Bonds	No. of Angles	No. of Dihedrals	Computation Time (s)
32000	27723	40467	56829	0.379309
64000	55446	80934	113658	0.785921
128000	110892	161868	227316	1.674183
256000	221784	323736	454632	3.130863

Fig. 14 plots the timing performance results of the pure software implementation and the MD machine for the pairwise interaction computations on two nodes of the Maxwell, as shown in tables 4 and 5, respectively. As it can be seen, at all atom sytems, the MD machine operates faster than the pure software implementation. Note that both plots in fig. 14 show a quadratically increasing curve which is obviously much sharper for the pure software solution for the MD simulations.

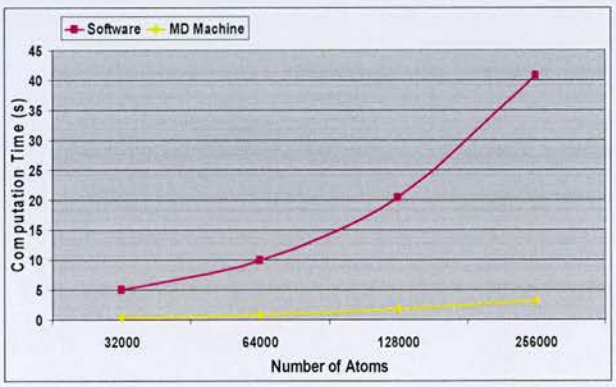


Fig. 14. Timing performance plot of the LAMMPS software and the MD machine for the pairwise interaction computations on two nodes of the Maxwell

Table 6 below provides the speed-up values of the MD machine over the pure software implementation (LAMMPS) for the pairwise interaction computations of the systems with various numbers of atoms on two Maxwell nodes. Note that the MD machine outperforms the pure software implementation by 12x-13x.

TABLE 6
LAMMPS Versus MD Machine Speed-Up Values for the Interaction Computations on Two Maxwell Nodes

No. of Atoms	MD Machine Speed-Up
32000	13.32
64000	12.61
128000	12.19
256000	13.01

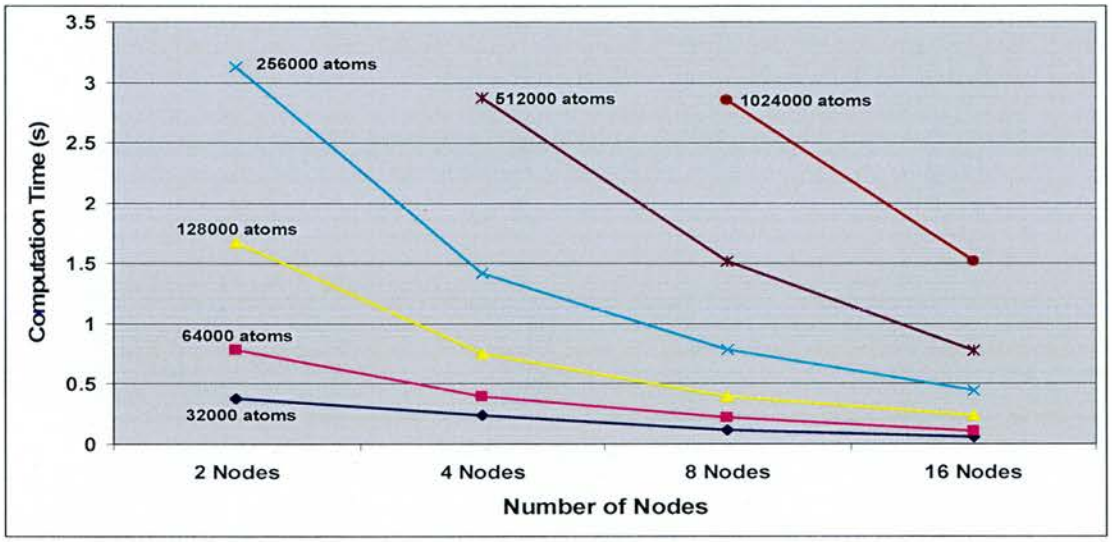


Fig. 15. Scaling performance of the MD machine on different numbers of nodes of the Maxwell for the given numbers of atoms

Table 7 shows the timing performance figures of the MD machine on two Maxwell nodes, this time including the I/O communication costs of the data transfers between a host CPU and SDRAM banks, as opposed to table 5. As it can be seen, I/O communication times account for over 96 percent of the total time. Due to this very high cost of the communication, the overall timing performance of the MD machine is poor compared to the pure software implementation for all atom systems (refer to table 4). Although multithreading, where each of the four existing threads deals with its assigned MD processor in the MD core, was utilised in the software process to take advantage of the direct memory transfers (DMA), the total time could not be reduced to a desired level because of the very poor data bandwidth between the host CPU and SDRAM banks. Nonetheless this limitation is not conceptual but rather dependent on the hardware platform targeted in this implementation. This communication bottleneck can be significantly resolved by integrating FPGA boards tighter into the host systems. In this way, FPGAs will have high-speed access to host memory through, for instance, AMD's Hypertransport, Intel's Quick Path Interconnect or SGI's NumaLink which offer bandwidths ranging from 15 to 25.6 GB/s, thus reducing communication overheads by at least 100x in comparison to our currently used communication link. This would result in performance gains of the MD machine in overall over the pure software implementation by almost same factors listed in table 6.

TABLE 7
TIMING FIGURES OF THE MD MACHINE INCLUDING I/O COMMUNICATION COSTS ON TWO MAXWELL NODES

No. of Atoms	Total Time (s)	I/O Comm. Time (s)	Percent. of I/O Comm.
32000	11.202145	10.822836	% 96.61
64000	22.298593	21.512672	% 96.48
128000	44.749425	43.075242	% 96.26
256000	89.581439	86.450576	% 96.50

Tables 8, 9 and 10 below show the comparative timing figures of the pure software implementation and the MD machine for the pairwise interaction computations of the Rhodopsin protein systems with up to over two million atoms on 4, 8 and 16 nodes of the Maxwell machine, respectively. As it can be seen, the MD machine speed-up values for the pairwise interaction computations range from 10x to 14x. In addition, the efficiency and scalability of the MD machine on different numbers of nodes of the Maxwell is graphically represented in fig. 15 with the timing values for the given numbers of atoms, as presented in tables 5, 8, 9 and 10. Note that the computational power of our MD machine increases highly with the increasing number of the Maxwell nodes utilized.

TABLE 8
COMPARATIVE TIMING FIGURES OF THE LAMMPS SOFTWARE AND THE MD MACHINE ON FOUR MAXWELL NODES

No. of Atoms	SW Comp. Time (s)	MD Machine Comp. Time (s)	MD Machine Speed-Up
32000	2.467384	0.237942	10.37
64000	4.959632	0.398379	12.45
128000	10.006767	0.748793	13.36
256000	20.08099	1.416626	14.18
512000	39.751763	2.869737	13.85

TABLE 9
COMPARATIVE TIMING FIGURES OF THE LAMMPS SOFTWARE AND THE MD MACHINE ON EIGHT MAXWELL NODES

No. of Atoms	SW Comp. Time (s)	MD Machine Comp. Time (s)	MD Machine Speed-Up
32000	1.202016	0.118279	10.16
64000	2.444738	0.220909	11.07
128000	4.856528	0.392149	12.38
256000	9.857229	0.781985	12.61
512000	19.563335	1.516907	12.90
1024000	40.567838	2.854312	14.21

TABLE 10
COMPARATIVE TIMING FIGURES OF THE LAMMPS SOFTWARE AND THE MD MACHINE ON SIXTEEN MAXWELL NODES

No. of Atoms	SW Comp. Time (s)	MD Machine Comp. Time (s)	MD Machine Speed-Up
32000	0.610724	0.05701	10.71
64000	1.211133	0.111407	10.87
128000	2.406081	0.240549	10.00
256000	4.96922	0.441922	11.24
512000	9.783473	0.774302	12.64
1024000	19.683968	1.516273	12.98
2048000	40.287101	2.891285	13.93

Table 11 shows the resource utilization of the MD core in a user FPGA. Note that the total number of the floating-point adder/subtractors in the MD core is 36 while the total number of the floating-point multipliers is 44. In addition, 8 floating-point comparators are also utilised in our design. Furthermore, the floating-point adder/subtractors and comparators were entirely implemented in the slice logic. On the other hand, the floating-point multipliers were partially implemented in the DSP48 blocks on the FPGA. However, since each multiplier requires 4 DSP48 blocks and the total number of the DSP48 blocks in the user FPGA is just 160, it was only possible to map 40 of the multipliers to the DSP48 blocks while the rest of them were entirely implemented in the user logic.

The accuracy in the computations was sufficient enough to carry out stable MD simulations but the accuracy could be improved if the single extended precision (i.e. width of 40-bit) was used for the floating-point numbers inside the design rather than the single precision (i.e. width of 32-bit) [45]. However, this precision increase would require higher amounts of slice logic and DSP48 blocks to implement the floating-point arithmetic units utilised in the MD core. Unfortunately, currently used Xilinx Virtex-4 XC4VFX100 FPGA chips cannot accommodate any higher resource demand as can be clearly seen in table 11.

Furthermore, it is reported by [45] that the evaluation of the functions, which involves the piecewise third-order polynomial interpolation with a look-up table, requires a key with a width of at least 15 bits (see subsection 7.2.1). However, even 1 bit increase in the width of the used key would require doubling the size of the utilized Function Coefficients memories (see fig. 5). It is also impossible to realize the usage of 15-bit wide key considering the amount of Block RAMs available in the currently used Virtex-4 FX100 FPGA chip (refer to table 11).

TABLE 11
RESOURCE UTILIZATION OF THE MD CORE IN A USER FPGA

	Used	Available	Utilization
Number of Occupied Slices	39,880	42,176	% 94
Total Number of 4 Input LUTs	69,622	84,352	% 82
Number of Slice Flip Flops	43,021	84,352	% 51
Number of FIFO16/RAMB16s	280	376	% 74
Number of DSP48s	160	160	% 100

9 CONCLUDING REMARKS

The design and implementation of a FPGA core, namely MD core, carrying out all the necessary operations to compute the non-bonded interactions in a MD simulation with the purpose of accelarating the LAMMPS MD software was presented in this paper. Our MD processor core comprised of 4 identical pipelines working independently in parallel to evaluate the non-bonded potentials, forces and virials was implemented on the nodes of a FPGA-based supercomputer, named Maxwell, which consists of 64 Virtex-4 FPGA chips. This implementation allowed us to produce a special-purpose parallel machine for the hardware acceleration of the MD simulations. This machine yields higher computational power with the additional Maxwell nodes, making it highly scalable.

The timing performance figures of the MD machine for the pairwise LJ and short-range Coulombic (via PPPM) interaction computations in the MD simulations of the solvated Rhodopsin protein systems with various numbers of atom show performance gains over the pure software implementation by factors of up to 13 on two nodes of the Maxwell machine. These MD machine speed-up values for the pairwise interaction computations were also maintained on different numbers of Maxwell nodes. However, the overall timing performance of the MD machine is worse than the pure software implementation due to the very high I/O communication costs of the data transfers between a host CPU and SDRAM banks. This case stems from the very poor data bandwidth between a host CPU and SDRAM banks which is a limitation caused by the hardware platform targeted in this implementation (i.e. Maxwell FPGA-based supercomputer).

Nonetheless, if FPGA boards are integrated tighter into the host systems through, for instance, AMD's Hypertransport, Intel's Quick Path Interconnect or SGI's Numa-Link, the bandwidth of the I/O communications would be greatly enhanced up to 25.6 GB/s, thus yielding much lower communication costs (i.e. up to 100x reduction in comparison with our currently used communication link). This would result in performance gains of the MD machine in overall over the pure software implementation. On the other hand, the accuracy of the computations could be improved if the number of slices and DSP48 blocks available in the user FPGA (i.e. Xilinx Virtex-4 XC4VFX100) was higher. Furthermore, wider DSP48 blocks and larger block RAMs would also help to enhance the computation accuracy. Solving the aforementioned concerns with a better hardware implementation platform is the major plan for the future of this project.

REFERENCES

[1] M. P. Allen and D. J. Tildesley, "Computer Simulation of Liq-

uids", Oxford University Press, 1987.

[2] G. D. Fasman, "Prediction of Protein Structure and the Principles of Protein Conformations", Plenum Press, New York, 1989.

[3] Z. R. Wasserman and C. N. Hodge, "Fitting an inhibitor into the active site of thermolysin: A molecular dynamics case study", *J. Proteins: Structure, Function, and Bioinformatics*, vol. 24, no. 2, pp. 227-237, Feb. 1996.

[4] N. R. Taylor and M. Itzstein, "A structural and energetics analysis of the binding of a series of N-acetylneuraminic-acid-based inhibitors to influenza virus sialidase", *J. Computer-Aided Molecular Design*, vol. 10, no. 3, pp. 233-246, June 1996.

[5] D. I. Liao, E. Silverton, Y. J. Seok, B. R. Lee, A. Peterkofsky and D. R. Davies, "The first step in sugar transport: crystal structure of the amino terminal domain of enzyme I of the E. coli PEP: sugar phosphotransferase system and a model of the phosphotransfer complex with HPr.", *J. Structure*, vol. 4, no. 7, pp. 861-872, July 1996.

[6] N. L. Greenbaum, I. Radhakrishnan, D. J. Patel and D. Hirsh, "Solution structure of the donor site of a trans-splicing RNA", *J. Structure*, vol. 4, no. 6, pp. 725-733, June 1996.

[7] D. C. Rapaport, "The Art of Molecular Dynamics Simulation", Cambridge University Press, New York, 2004.

[8] P. P. Ewald, "Evaluation of optical and electrostatic lattice potentials", *Ann. Phys Leipzig*, vol. 64, pp. 253-287, 1921.

[9] L. Verlet, "Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules", *Physical Review*, vol. 159, no. 1, pp. 98-103, July 1967.

[10] D. Beeman, "Some Multistep Methods for Use in Molecular Dynamics Calculations", *J. Computational Physics*, vol. 20, pp. 130-139, Feb. 1976.

[11] M.E. Tuckerman, G.J. Martyna and B.J. Berne, "Reversible multiple time-scale molecular dynamics", *J. Chemical Physics*, vol. 97, no. 3, pp. 1990-2001, March 1992.

[12] D. V. D. Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark and H. J. Berendsen, "GROMACS: Fast, flexible, and free", *J. Computational Chemistry*, vol. 26, no. 16, pp. 1701-1718, Oct. 2005.

[13] GROMACS-4.0.7, "Download website for GROMACS 4.0.7", available at http://www.gromacs.org, Dec. 2009.

[14] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale and K. Schulten, "Scalable molecular dynamics with NAMD", *J. Computational Chemistry*, vol. 26., no. 16, pp. 1781-1802, Oct. 2005.

[15] NAMD-2.7b2, "Download website for NAMD 2.7b2", available at http://www.ks.uiuc.edu/Research/namd, Nov. 2009.

[16] LAMMPS, "Download website for LAMMPS", available at http://lammps.sandia.gov, Jan. 2010.

[17] F. Toshiyuki, T. Makoto, M. Junichiro, E. Toshikazu and S. Duiichiro, "A Highly Parallelized Special-Purpose Computer for Many-Body Simulations with an Arbitrary Central Force: MD-GRAPE", *Astrophysical Journal*, vol. 468, pp. 51-61, Sep. 1996.

[18] Y. Komeiji, M. Uebayasi, R. Takata, A. Shimizu, K. Itsukashi and M. Taiji, "Fast and Accurate Molecular Dynamics Sumulation of a Protein Using a Special-Purpose Computer", *J. Computational Chemistry*, vol. 18, no. 12, pp. 1546-1563, Sep. 1997.

[19] S. Toyoda, H. Miyagawa, K. Kitamura, T. Amisaki, E. Hashimoto, H. Ikeda, A. Kusumi and N. Miyakawa, "Development of MD Engine: High-Speed Accelerator with Parallel Processor Design for Molecular Dynamics Simulations", *J. Computational Chemistry*, vol. 20, no.2, pp. 185-199, 1999.

[20] C. Wolinski, F. Trouw and M. B. Gokhale, "A preliminary study of molecular dynamics on reconfigurable computers", *Proc. International Conf. Engineering Reconfigurable Systems and Algorithms*, June 2003.

[21] R. Scrofano and V. K. Prasanna, "Computing Lennard-Jones potentials and forces with reconfigurable hardware", *Proc. International Conf. Engineering Reconfigurable Systems and Algorithms*, June 2004.

[22] R. Scrofano, M. B. Gokhale, F. Trouw and V. K. Prasanna, "Accelerating Molecular Dynamics Simulations with Reconfigurable Computers", *IEEE Trans. on Parallel and Distributed Systems*, vol. 19, no. 6, pp. 764-778, June 2008.

[23] Y. Gu, T. VanCourt and M. C. Herbordt, "Improved interpolation and system integration for FPGA-based molecular dynamics simulations", *Proc. International Conf. Field Programmable Logic and Applications*, pp. 21-28, 2006.

[24] Y. Gu, T. VanCourt and M. C. Herbordt, "Explicit design of FPGA-based coprocessors for short-range force computations in molecular dynamics simulations", *Elsevier Parallel Computing*, vol. 34, no. 4, pp. 261-277, May 2008.

[25] N. Azizi, I. Kuon, A. Egier, A. Darabiha and P. Chow, "Reconfigurable Molecular Dynamics Simulator", *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*, pp. 197-206, Apr. 2004.

[26] Y. Gu, T. VanCourt and M. C. Herbordt. "Accelerating molecular dynamics simulations with configurable circuits", *Proc. International Conf. Field Programmable Logic and Applications*, pp. 475-480, Aug. 2005.

[27] LAMMPS, "LAMMPS manual", available at http://lammps.sandia.gov/doc/Manual.html, Jan. 2010.

[28] L. Greengard and V. Rokhlin, "A Fast Algorithm for Particle Simulations", *J. Computational Physics*, vol. 73, no.2, pp. 325-348, Dec. 1987.

[29] H. Q. Ding, N. Karasawa and W. A. Goddard, "Atomic level simulations on a million particles: The cell multipole method for Coulomb and London nonbond interactions", *J. Chemical Physics*, vol. 97, no. 6, pp. 4309-4315, Sep. 1992.

[30] R. W. Hockney and J. W. Eastwood, "Computer Simulation Using Particles, Adam Hilger, 1988.

[31] T. Darden, D. York and L. Pedersen, "Particle mesh Ewald: An N·log(N) method for Ewald sums in large systems", *J. Chemical Physics*, vol. 98, no. 12, pp. 10089-10092, June 1993.

[32] S. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics", *J. Computational Physics*, vol. 117, no. 1, pp. 1-19, Mar. 1995.

[33] S. J. Plimpton, R. Pollock, M. Stevens, "Particle-Mesh Ewald and rRESPA for Parallel Molecular Dynamics Simulations", *Proc. SIAM Conf. Parallel Processing for Scientific Computing*, Mar. 1997.

[34] E. L. Pollock and J. Glosli, "Comments on P3M, FMM, and the Ewald method for large periodic Coulombic systems", *Computer Physics Communications*, vol. 95, no. 2, pp. 93-110, June 1996.

[35] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, R.Smart, A. Cantle, R. Chamberlain and G. Genest, "Maxwell—a 64 FPGA supercputer", *Proc. NASA/ESA Conf. Adaptive Hardware Systems*, pp. 287-294, 2007.

[36] FHPCA, Edinburgh, U.K., "The FHPCA website", available at http://www.fhpca.org, 2010.

[37] Alpha Data Ltd., Edinburgh, U.K., "ADM-XRC-4FX Datasheet",

available at http://www.alphadata.co.uk./adm-adm-xrc-4fx.html, May 2007.

[38] Nallatech Ltd., Glasgow, U.K., "H100 Series Datasheet", available at http://www.nallatech.com/meadiLibrary/images/english/5595.pdf, May 2007.

[39] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, R. Smart, A. Cantle, R. Chamberlain and G. Genest, "The FPGA HPC alliance parallel toolkit", *Proc. NASA/ESA Conf. Adaptive Hardware Systems*, pp.301-310, 2007.

[40] FHPCA, Edinburgh, U.K., "PowerPoint presentation", available at http://www.fhpca.org/download/MRSC07-Mar07.ppt, Mar. 2007.

[41] Argonne National Lab, Argonne, IL, "MPI manual", available at http://www.unix.mcs.anl.gov/mpi/www/www3/MPI_Wtime.html, 2009.

[42] OpenCores website, "Floating Point Adder and Multiplier", available at http://opencores.org/project,fpuvhdl, 2009.

[43] G. Marcus, P. Hinojosa, A. Avila and J. N. Flores, "A Fully Synthesizable Single-Precision, Floating-Point Adder/Subtractor and Multiplier in VHDL for General and Educational Use", *Proc. IEEE International Caracas Conf. Devices, Circuits and Systems*, pp. 319-323, Nov. 2004.

[44] Xilinx Inc., San Jose, CA, "Virtex-4 datasheets", available at http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm, May 2007.

[45] T. Amisaki, T. Fujiwara, A. Kusumi, H. Miyagawa and K. Kitamura, "Error evaluation in the design of a special-purpose processor that calculates nonbonded forces in molecular dynamics simulations", *J. Computational Chemistry*, vol. 16, no. 9, pp. 1120-1130, Sep. 1995.

[46] M. Chiu, M.C. Herbordt, "Efficient particle-pair filtering for acceleration of molecular dynamics simulation", *Proc. International Conf. Field Programmable Logic and Applications*, pp. 345-352, Aug. 2009.

Appendix B

Conference Publications

1. **S. Kasap**, K. Benkrid and Y. Liu, "High Performance FPGA-based Core for BLAST Sequence Alignment with the Two-Hit Method", *Proceedings of the 8th IEEE International Conference on Bioinformatics and Bioengineering (BIBE)*, Athens, Greece, October 2008.

2. **S. Kasap**, K. Benkrid and Y. Liu, "A High Performance FPGA-based Implementation of Position Specific Iterated BLAST", *Proceedings of the 17th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, Monterey, California, USA, pp. 249-252, February 2009.

3. Y. Liu, K. Benkrid, A. Benkrid and **S. Kasap**, "An FPGA-Based Web Server for High Performance Biological Sequence Alignment", *Proceedings of the 4th NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, San Francisco, California, USA, pp. 361-368, July 2009.

4. **S. Kasap** and K. Benkrid, "A High Performance FPGA-based Core for Phylogenetic Analysis with Maximum Parsimony Method", *Proceedings of the 8th IEEE International Conference on Field Programmable Technology (FPT)*, Sydney, Australia, pp. 271-277, December 2009.

5. **S. Kasap** and K. Benkrid, "A High Performance Implementation for Molecular Dynamics Simulations on a FPGA-based Parallel Computer", *the 7th ACM International Symposium on Applied Reconfigurable Computing (ARC)*, Belfast, UK, March 2011 (to be submitted by November 2010).

High Performance FPGA-based Core for BLAST Sequence Alignment with the Two-Hit Method

Server Kasap, Khaled Benkrid, *Senior Member, IEEE,* Ying Liu

***Abstract*--This paper presents the design and implementation of a high performance FPGA-based core for BLAST sequence alignment with the two-hit method. BLAST with two-hit is a very widely used heuristic biological sequence alignment algorithm, and this paper is the first reported FPGA implementation of it, to our knowledge. The architecture of our core is parameterized in terms of the sequence lengths, match scores, gap penalties, and cut-off and threshold values. It is composed of various blocks each of which performs one step of the algorithm in parallel with the others. This results in a high performance and efficient FPGA implementation, which outperforms equivalent software implementations by one order of magnitude or more. Real hardware implementations show that our core is 52 times faster than equivalent software implementations, on average. Furthermore, the core was captured in an FPGA-platform-independent language, namely the Handel-C language, to which no specific resource inference or placement constraints were applied. Hence, the same code can be easily ported to different FPGA families and architectures.**

I. Introduction

In Bioinformatics and Computational biology (BCB), biological sequence alignment is a very common task where subject sequences from a large database are aligned to a query sequence to find similarities between the query sequence and the sequences in the database [1]. Obtaining information about a newly discovered biological sequence (i.e. Protein, DNA or RNA) from other known sequences is a major application of this operation. For example, if a new sequence is found to be similar to a known cancerous sequence, then information regarding the functionality of the new sequence can be deduced. This is obviously useful in early disease diagnosis and drug engineering. Furthermore, biological sequence alignment can be utilized in the study of evolutionary development and history of species [1] [2].

Sequence alignment is a computationally intensive operation, however. This is exacerbated by the exponential growth in biological sequence databases. Therefore, desktop computer systems cannot, usually, perform this task within acceptable execution timeframes. Hence, faster computing platforms are required.

Manuscript received July 5, 2008.

Server Kasap is with the University of Edinburgh, Mayfield Road, Edinburgh EH9 3JL, Scotland, UK. He is a PhD student in School of Electronics and Engineering (E-mail: s.kasap@ed.ac.uk).

Khaled Benkrid is with the University of Edinburgh, Mayfield Road, Edinburgh EH9 3JL, Scotland, UK. He is a lecturer in the School of Engineering and Electronics (e-mail: k.benkrid@ed.ac.uk).

Ying Liu is with the University of Edinburgh, Mayfield Road, Edinburgh EH9 3JL, Scotland, UK. She is a PhD student in School of Electronics and Engineering (E-mail: y.liu@ed.ac.uk).

Recently, high performance reconfigurable hardware in the form of Field Programmable Gate Arrays (FPGAs) has been proposed as an efficacious and efficient implementation platform for sequence alignment algorithms [3] [4] [5]. Indeed, their ASIC-like performance coupled with their reprogrammability feature make FPGAs capable of providing high speed-ups compared to general purpose processors, with the added convenience of reprogrammability.

There are various biological sequence alignment algorithms in the literature. Some of these are exhaustive and give optimal alignments (e.g. Needleman-Wunsch [6], Smith-Waterman [7]) and others are heuristic and give sub-optimal alignments (e.g. FASTA [8], BLAST [9]). In this paper, we concentrate on Basic Local Alignment Search Tool (BLAST) which is a heuristic local alignment algorithm. It is much faster than ordinary exhaustive dynamic programming algorithms, although it produces local alignments which are not always optimal. The design and implementation of BLAST with two-hit method [10] (a variant of BLAST) is presented in this paper. To our knowledge, this is the first reported FPGA implementation of this algorithm variant. It results in 52x speed-up over equivalent software implementations on average. Besides, the design was captured in a FPGA-platform-independent language, namely Handel-C language [13], which makes it portable across a number of FPGA architectures (e.g. from Xilinx or Altera).

In the remainder of this paper, essential background information on the general BLAST algorithm will be presented first. Following that, the design and implementation of our FPGA core for BLAST with the two-hit method will be elaborated. After that, timing performance of our core implementation is presented and compared with equivalent software implementation running on desktop computers. Finally, conclusions are laid out with plans for future work.

II. Background

Biological sequences evolve through mutation, selection and random genetic drift [11]. Mutation, in particular manifests itself through 3 main processes:

- Substitution of residues: Residue A in the sequence is substituted by another residue B.
- Insertion of residues: New residues are inserted into the sequence.
- Deletion of residues: Existing residues in the sequence are deleted.

Insertions and deletions result in *gaps* which are taken into consideration when aligning biological sequences. The degree of alignment of biological sequences is measured by a score which is obtained by the summation of score terms of each aligned pair of

residues with possible gap penalty terms. Score terms for each aligned residue pair are obtained from probabilistic models which are stored in score or substitution matrices such as BLOSUM50 [1]. The latter is a 20x20 matrix for protein sequence residues. On the other hand, gap penalties depend on the length of the gap and are independent of gap residues. There are two main types of gap penalties:

- Linear gap penalty: The cost of a gap of length g is given by following linear function:

$$\text{Penalty}(g) = -g*d$$

- Affine gap penalty: A constant penalty is given for opening a new gap while a linear and smaller penalty is given for subsequent gap extensions. The cost function of the affine gap penalty is hence given by the following affine equation:

$$\text{Penalty}(g) = -d-(g-1)*e$$

BLAST stands for Basic Local Alignment Tool. It is developed on the ideas of FASTA. It is used for searching both protein and DNA sequence databases for sequence similarities. It is a heuristic local alignment algorithm which approximates the dynamic programming Smith-Waterman algorithm. Since it is a heuristic algorithm, the local alignment it produces is not always optimal. However, it is much faster than the Smith-Waterman algorithm. As a result, BLAST and its variants are some of the most widely used sequence search tools.

The central idea of the BLAST algorithm is that a statistically significant alignment is likely to contain a high-scoring pair of aligned words. BLAST first finds these high scoring pairs of aligned words and then extends them to the real alignment. These words are k-residues long where k is different for DNA and protein sequences. The default k values for DNA and protein sequences are 11 and 3 respectively. There are 3 basic steps of BLAST:

- Pre-processing the query sequence: All k-long words in the query sequence are extracted. Then, words that are similar to these are found. We call the overall results the k-words.
- Scanning the subject sequences: All the subject sequences in the database are scanned one by one for matches with the obtained k-words.
- Extension of the matches: All matches in the subject sequences are extended to form local alignments between the query sequence and related subject sequences in the database.

In subsections II.A-II.C, all basic steps of the BLAST algorithm mentioned above will be explained in more detail.

It is worth mentioning at this stage that the aforementioned basic steps belong to the original BLAST algorithm. However, several variants of the original algorithm have been devised over the years with the aim of increasing its sensitivity while keeping run-times at minimum. All of these variants include the 3 basic steps of the original algorithm, with the addition of new steps. In this paper, we discuss one of these variants, namely BLAST with two-hit method which is described in subsection II.D.

A. Step 1: Pre-processing the Query Sequence

An example protein sequence which has 9 residues (or amino acids) is shown below:

LVNRKPVVP

In this first step, we take the query sequence and chop it into overlapping k-words as illustrated below for the query sequence shown above, with k = 3:

Word 0: LVN

Word 1: VNR

Word 2: NRK

Word 3: RKP

Word 4: KPV

Word 5: PVV

Word 6: VVP

As it can be seen, there are 7 words extracted from the query sequence which are 3 residues long. In general, the number of words extracted equals (m-k) + 1 where m is the number of residues in the query sequence. After this, words similar to each of these extracted words are found through the usage of specific scoring matrix.

Words which score at least threshold value T with the scoring matrix when aligned with the words extracted from the query sequence are regarded to be similar to these extracted words. Similar words for each extracted word are found and then recorded with the location address of the corresponding extracted word in the query sequence tagged to them. This process is illustrated below with the first extracted word shown above (i.e. LVN) using the BLOSUM50 scoring matrix for the case where T is 12:

Word 0: L V N
4 + 4 + 6 = 14
Query word 1: L V N

Word 0: L V N
2 + 4 + 6 = 12
Query word 2: M V N

Word 0: L V N
4 + 4 + 1 = 9
Query word 3: L V S

Query word 1 and query word 2 score 14 and 12 respectively when aligned with the first extracted word (LVN) from the query sequence. Since score values are over or equal to 12, query word 1 and query word 2 are recorded with the location address of the first extracted word in the query sequence, which is 0. However, query word 3 is discarded since it scores less than 12 when aligned with the extracted word. All recorded similar words are used in step 2 of the BLAST algorithm.

B. Step 2: Scanning the subject sequences

In this step, all subject sequences in the database are scanned one by one to find the possible exact matches of the query words which were recorded in step 1. Each match is referred to as hit or hotspot. Each hit is recorded in a list for the third step of the BLAST algorithm with the identity of the corresponding query word and the location address where the hit occurred in the subject sequence. Considering the fact that current databases contains tens of thousands of subject sequences and that each subject sequence comprises hundreds/thousands of residues, it is obvious that this sequence database scanning process is a massively time consuming task.

C. Step 3: Extension of the matches

In this last step of the basic BLAST algorithm, we utilize the list of matches (hits) obtained in step 2 to form local alignments between the query sequence and the subject sequences in the database. Each entry in the list of hits contains the location address of a match in the subject sequence and the location address of the corresponding query word in the query sequence. Starting from these two location addresses, each of the hits in the list is extended on the query and corresponding subject sequence in both directions without allowing any gaps. In this extension, pairs of residues along the query and subject sequence are scored with a scoring matrix (e.g. BLOSUM50). This process is illustrated in figure 1 with the following subject sequence:

GVCRRPLKC

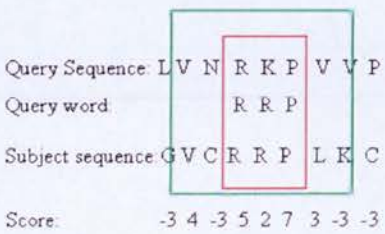


Figure 1. Step 3: Extension of matches

In figure 1, the small box shows a hit where query word RRP is matched in the subject sequence. The query word RRP is similar to RKP word in the query sequence. The big box in figure 1 shows the extension which started from the edges of the small box. As the extension proceeds in a 1 residue pair at a time in both directions and without allowing for any gaps, pairs of residues along the extension are scored using a scoring matrix (BLOSUM50 in our case). These score terms are added up after each extension step and the extension is terminated when this total score falls a certain cut-off distance below the best total score obtained so far. Then, the extension goes back to its state which yielded the highest total score. As a result of this extension step, the related subject sequence is locally aligned to the query sequence (without gaps).

D. BLAST with two-hit method

The third step of the BLAST algorithm, i.e. the extension of the matches on the query and subject sequences, generally accounts for a very high percentage of the BLAST algorithm's execution time. Hence, the two-hit method was devised to reduce the time spent in this extension step. The central idea of the two-hit method is to start the extension only when there are two non-overlapping hits on the same diagonal within distance A of each other. This is illustrated in figure 2 where only two non-overlapping hits on the same diagonal line which are close enough to each other are extended.

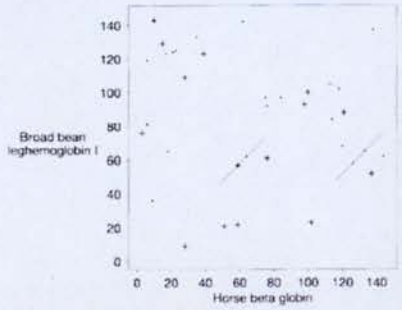


Figure 2. Ungapped extension of two close hits on the same diagonal lines [10]

In other words, if the distance between any two non-overlapping hits on the subject sequence is equal to the distance between the locations of the corresponding query words in the query sequence, then ungapped extension is triggered in both directions starting from both hits. The rest of the process is the same as explained in subsection II.C and the result is a local ungapped alignment of the query and subject sequences. This process is illustrated in figure 3 where A is equal to 5.

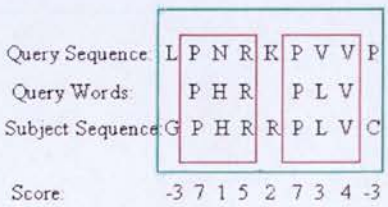


Figure 3. Extension with the two-hit method

In figure 3, the small boxes show two non-overlapping hits on the query and subject sequences within a distance of 4. Since the distance between the query words in the query sequence is equal to the distance between the two hits on the subject sequence, and since this distance between the two hits is less than 5, and bigger than 2, ungapped extension is started from the edges of the left and right hand sides of the small boxes respectively (see the big box in figure 3).

To maintain the sensitivity of the general algorithm, the threshold value T used in the query pre-processing step of the algorithm is reduced. Hence, the number of query words recorded in this step will increase. As a result, while scanning the subject sequences in step 2, we will potentially find more hits than before. However, only a small fraction of these hits will have an associated

second hit. Therefore, ungapped extension will be triggered less frequently compared to the case in the original BLAST algorithm. The total execution time of BLAST is thus reduced.

III. Hardware Implementation of BLAST with the Two-Hit Method

Figure 4 shows our hardware architecture which implements the BLAST algorithm with the two-hit method. Each block in the architecture implements one step of the algorithm as described in the above sections, except for the pre-processing query sequence step which is implemented by high level application software running on a host computer. The architecture consists of 12 *HitFinderTwoHit* blocks, 3 *UngappedExtender* blocks and 1 *Collector* block all of which are running in parallel. There are also 12 32K x 5 bits subject sequence memories each of which holds a number of subject sequences. Note that each subject sequence memory belongs to one *HitFinderTwoHit* block each of which is composed of 5 *HitFinder* blocks and 1 *TwoHitMethod* block. Each *HitFinder* block implements step 2 outlined in subsection II.B and scans its assigned subject sequence memory to find exact matches of the query words in the subject sequences. Each *TwoHitMethod* block performs the two-hit method procedure on hits coming from the 5 *HitFinder* blocks which are in the same *HitFinderTwoHit* block as the *TwoHitMethod* block. Besides these, each *UngappedExtender* block implements step 3 mentioned in subsection II.C and extends the two hits found by its four allocated *TwoHitMethod* blocks without allowing gaps, in order to obtain local ungapped alignments. Finally, a single *Collector* block collects high-scoring local ungapped alignments obtained in 3 *UngappedExtender* blocks and sends their details to the host.

The high level application software and all of the blocks which constitute the architecture shown in figure 4 are detailed in the following subsections.

A. *High Level Application Software*

Figure 5 shows the organization of our FPGA implementation for BLAST with two-hit method. There is application software running on the host computer which has many duties, the most important of which is the query sequence pre-processing as explained in section II.A. Besides running application software, host computer stores sequence database (e.g. Swiss-Prot) which is read as required by application software. In brief, the application software finds 3 letter long query words which score at least a threshold value T when aligned with words extracted from the query sequence. Then, the location address of each of these query words in the query sequence is placed at a vacant position in an upper word list and a lower word list pair depending on the 2 most significant letters and 2 least significant letters of the query word, respectively. Note that there are 5 upper-word and lower-word list pairs.

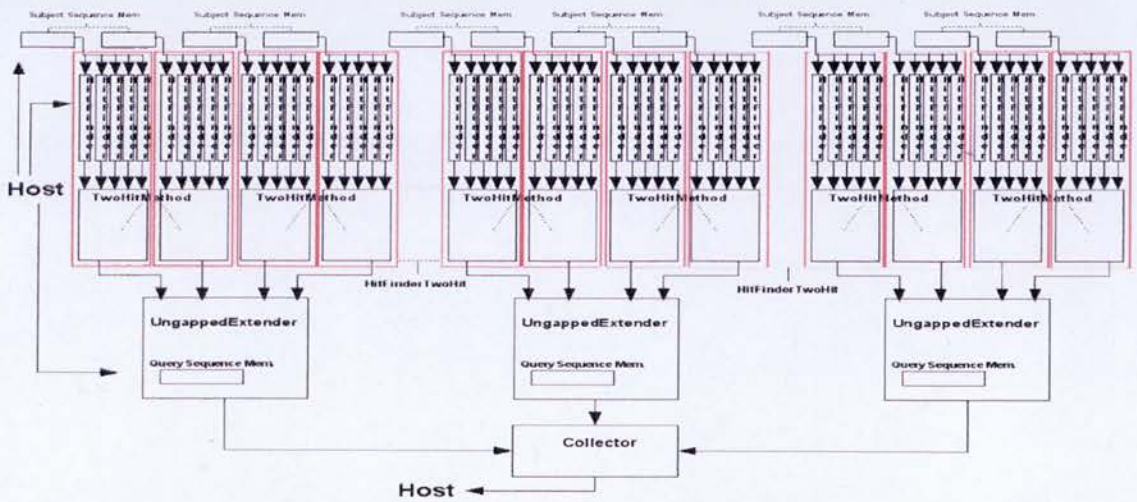


Figure 4. Hardware architecture for the BLAST algorithm with the two-hit method

As it can be seen in figure 5, there are various FPGA configuration bit files for different threshold and cut-off value parameters. The first task of the application software is to pick the proper bit file, depending on the user-supplied algorithm parameters, from a database of FPGA configurations and load it on to the FPGA chip. Afterwards, the application software runs the hardware implementation in 4 modes. In mode 1, the application software sends each of the 5 upper word and lower word list pairs to each of the 5 *HitFinder* blocks in every *HitFinderTwoHit* block. In mode 2, a number of subject sequences read from the database on host are sent to the 12 available subject sequence memories on FPGA, depending on the subject sequence lengths. In mode 3, the application software sends a query sequence to the FPGA to be stored in memories within the 3 *UngappedExtender* blocks. Finally, the execution of the hardware implementation is launched in mode 4.

After some time, the FPGA starts sending the high scoring subject sequences with their alignment scores. Then, application software prints these ungapped alignments onto the screen. This completes the first iteration of the operation. In the following iterations, different set of subject sequences are sent to the FPGA to be processed. Iterations terminate when there is no more subject sequence in the database awaiting to be sent to the FPGA.

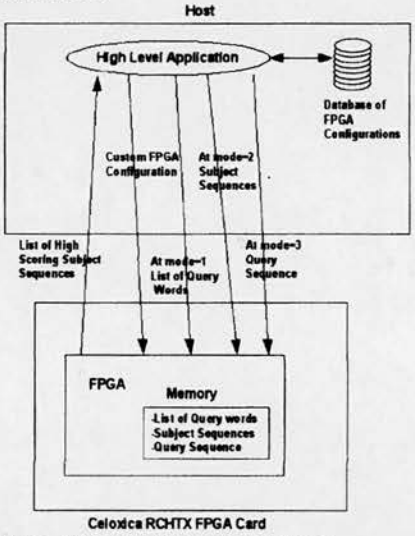


Figure 5. Organization of our BLAST system

B. HitFinder Block

Figure 6 shows a simplified inner structure of a *Hitfinder* block. The architecture of this block is a modified version of the one shown in figure 7 of [18]. The major aim of this block is to scan each three letter long word of the subject sequences in order to find exact matches of the query words, as explained in subsection II.B. It is comprised of an upper word list memory, a lower word list memory, a shift register, a FIFO buffer and some control logic. Note that every *Hitfinder* block is assigned to a subject sequence memory whose address register (*Counter*) is unique in the *HitFinderTwoHit* block.

At every clock cycle, 5-bit long residues of a subject sequence are shifted into the shift register (*ShiftReg*) from the assigned subject sequence memory and the address register of the subject sequence memory is incremented by one. The shift register is 15 bits long and hence it can hold 3 subject sequence residues at the same time. At every clock cycle, the 10 most significant bits and the 10 least significant bits of the shift register content are used as addresses for the upper word list memory and the lower word list memory respectively (see figure 6). If the resulting outputs of these memories are valid entries and are equal to each other, this means that a three-letter long word of the subject sequence which is currently held in the shift register matches exactly a query word whose location address in query sequence is given in the outputs of the word list memories. In this case, we have a hit condition which needs to be recorded for the following steps of the algorithm. Hence, we register the address of the query word in the query sequence and the location address of the hit in the subject sequence to a FIFO buffer named *Hit FIFO* with 3 control bits. These entries to *Hit FIFO* are processed by the *TwoHitMethod* block assigned to the *Hitfinder* block (see figure 4).

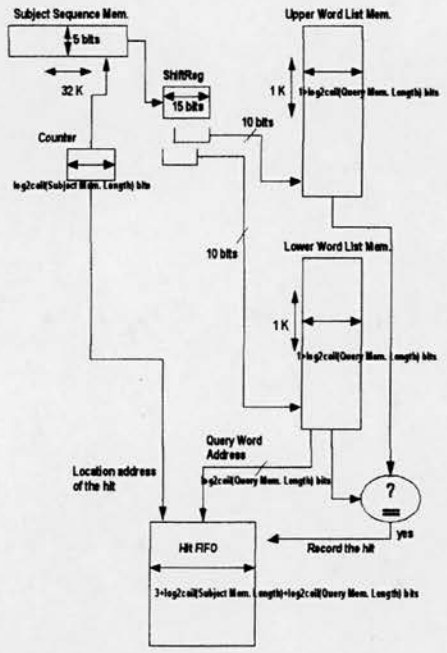


Figure 6. Simplified inner structure of the *Hitfinder* block

C. TwoHitMethod Block

Figure 7 shows a simplified inner structure of the *TwoHitMethod* block. Its aim is to find two non-overlapping hits on the same diagonal within distance A of each other as explained in subsection II.D above. In this architecture, there are two FIFOs of the same length and same width namely *Hit FIFO 1* and *Hit FIFO 2* to which the same hit entries from the *Hit FIFOs* of the 5 *Hitfinder* blocks (which belong to the same *HitFinderTwoHit* block) are stored one by one in turn starting from the *Hit FIFO* in the first *Hitfinder* block. The processing of hit entries commences when there are more than two hit entries in the FIFOs. For instance, the a^{th} hit entry of *Hit FIFO 1* and b^{th} hit entry of *Hit FIFO 2* are taken and the hit addresses of these entries are subtracted from each other. If the result is less than 3, we continue with the processing of the a^{th} hit entry in *Hit FIFO 1* and $(b+1)^{th}$ hit entry in *Hit FIFO 2* in the next clock cycle. On the other hand, if the result is bigger than threshold value A, we continue with the processing of the $(a+1)^{th}$ hit entry in *Hit FIFO 1* and $(a+2)^{th}$ hit entry in *Hit FIFO 2* in the next clock cycle. However, if the result of this subtraction is between 3 and threshold value A inclusive, we subtract the query word addresses in the hit entries. If the second subtraction result is not equal to the first one, this means that the two hits are not on the same diagonal, and hence we continue with the processing of the a^{th} hit entry in *Hit FIFO 1* and $(b+1)^{th}$ hit entry in *Hit FIFO 2* in the next clock cycle. If the two results are the same, however, this means that we have two close enough non-overlapping hits on the same diagonal which need

to be recorded for the subsequent steps of the algorithm. The two hit cases are recorded to two FIFOs namely *TwoHit FIFO1* and *TwoHit FIFO* 2. The address of the first hit and the distance between the two hits (Result 2 in figure 7) are stored in *TwoHit FIFO1* with 2 control bits, whereas the address of the first query word is stored in *TwoHit FIFO* 2. These two-hit entries to the *TwoHit FIFOs* are subsequently processed by the assigned *UngappedExtender* block.

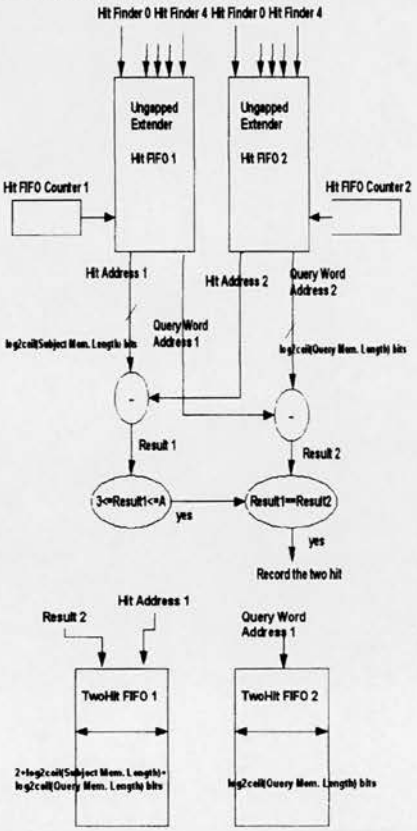


Figure 7. Simplified inner structure of TwoHitMethod block

D. UngappedExtender Block

The *UngappedExtender* block implements the ungapped extension step of the BLAST algorithm as explained in subsection II.C above. Each of the three *UngappedExtender* blocks reads *TwoHit FIFOs* of its four assigned *TwoHitMethod* blocks in turn. When the *UngappedExtender* block detects a two-hit entry in the *Twohit FIFOs* of one *TwoHitMethod* block, the hit address of the first hit, the address of the first query word in the query sequence, and the distance between the two hits are all extracted from that entry to compute the start (seed) points of the outward ungapped extension in both directions, on both query and related subject sequence. Note that first residue pair of the first hit and the last residue pair of the second hit are the seed points of the outward ungapped extension on the query and related subject sequence. Afterwards, the inward ungapped extension starts from one start point to the other start point where the residue pairs along the extension are scored against a scoring matrix, with the intermediate scores accumulated. When the inward ungapped extension ends, the outward ungapped extension is launched in both directions. Here again, the residue pairs along the extension are scored, with the intermediate score terms accumulated, and added up with the total score obtained from the inward ungapped extension. The outward ungapped extension terminates either when the currently computed grand total score falls a certain cut-off value below the highest grand total score obtained so far, or when the extension reaches the end of the query or subject sequences in either direction. In this case, the ungapped extension retracts to its previous state which yielded the highest grand total score. If this highest grand total score exceeds a certain threshold value, the end points of this high scoring ungapped extension in both directions on both query and subject sequences are registered to two *UngappedResult FIFOs* with the score to be read by the single *Collector* block which sends these points as well as the score of the ungapped extension to the host.

IV. Results

Our BLAST design was captured in the Handel C language to which no specific resource inference or placement constraints were applied. Hence, it can be directly targeted to a variety of FPGA platforms (e.g. Xilinx or Altera FPGAs). The resulting core was compiled into EDIF by Agility's DK5 SP2 suite from which FPGA bitstreams were generated using Xilinx ISE9.2 tool.

The hardware implementation of the core was achieved on a Celoxica RCHTX FPGA board [17] which has a Xilinx Virtex 4 (xc4vlx160ff1148-11) FPGA chip and off-chip memory fitted on it. In our implementation, however, the off-chip memory was not used. The operation of the core was tested on the Swiss-Prot protein sequence database [16] with various query protein sequences.

We have also implemented BLAST with the two-hit method algorithm in C in order to compare our hardware implementation with a pure software implementation. Table 1 presents timing performance figures of both hardware and software implementations for 8 random query protein sequences of various lengths searched in the Swiss-Prot database. The FPGA hardware was clocked at 20 MHz. The software implementation was executed on an Intel Centrino Duo 2.2 GHz PC with 2 GB RAM. The same threshold and cut-off values were used in both hardware and software implementations at every step of the algorithm.

Table 1. Timing performance figures of hardware and software implementations for 8 random protein sequences queried in Swiss-Prot protein sequence database

	No of Residues in Query Sequence	No of Query words	FPGA Execution time (sec)	Software execution time (sec)	FPGA Speed-up
1. Query Sequence	111	116	3.49	78.85	22.59
2. Query Sequence	368	136	3.50	137.98	39.42
3. Query Sequence	459	263	3.52	209.84	59.61
4. Query Sequence	565	137	3.45	177.57	51.47

5. Query Sequence	635	140	3.46	179.45	51.86
6. Query Sequence	746	117	3.57	209.25	58.61
7. Query Sequence	864	240	3.52	286.47	81.38
8. Query Sequence	985	53	3.48	197.87	56.86

As it can be seen from table 1, our FPGA implementation results in substantial speed-up compared to software, ranging from 81x to 22x (the speed-up figure depends on the query sequence). Note that the FPGA execution times fluctuate around 3.5 seconds hence showing experimentally that it is predominantly dependent on the size of the database rather than on the size of the query sequence or number of query words.

On average, our core is 52 times faster than equivalent software implementations. The reason behind this high speed-up figure of the FPGA implementation, despite the huge difference in clock frequency, is due to the high level of process parallelism on FPGA. Besides, the complete design, implementation and testing was achieved in less than 5 months by a first year PhD student. This shows that reconfigurable technology can be an efficacious and efficient platform for high performance biological sequence analysis.

V. Conclusion

In this paper, the detailed FPGA implementation of the BLAST algorithm with two-hit method has been presented. This is the first FPGA implementation of this variant of BLAST ever reported in the literature, to our knowledge. The hardware architecture is composed of various blocks each of which performs a specific step of the algorithm in parallel. Moreover, the FPGA core is parameterized in terms of the sequence lengths, match score, gap penalties, cut-off and threshold values. The resulting implementation outperforms equivalent desktop-based software by 52 times on average. Furthermore, our core was designed in the Handel-C language, thus making it FPGA-platform-independent. This means that our core can be ported to other FPGA architectures from different vendors very easily. Finally, it is worth mentioning that the whole design, implementation and testing design took less than 5 person-months to achieve, which shows that FPGAs can be an economic platform for high performance biological sequence alignment.

The work presented in this paper is part of a bigger project where the computational performance and re-configurability features of FPGAs are harnessed in the field of bioinformatics and computational biology. Future work includes the extension of this core to support the Gapped BLAST and Position Specific Iterated BLAST (PSI-BLAST) algorithms.

VI. References

[1] Durbin, R., Eddy, S., Krogh, A., and Mitchison, G., 'Biological Sequence Analysis: Probabilistic Models for Proteins and Nucleic Acids', Cambridge University Press, Cambridge UK, 1998

[2] Hein, J. 'A New Methodology that simultaneously aligns and reconstructs ancestral sequences for any number of homologous sequences, when a phylogeny is given'. Journal of Molecular Biology, 6, pp.649-668, 1989

[3] Hoang, D.T. 'Searching genetic databases on Splash 2', in Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, pp. 185-191, 1993.

[4] Gokhale, M. et al. 'Processing in memory: The Terasys massively parallel PIM array', Computer, 28 (4), pp. 23-31, April 1995.

[5] TimeLogic Corporation, 'Decypher Scalable, High Performance Biocomputing Solutions', http://www.timelogic.com

[6] Needleman, S. and Wunsch, C. 'A general method applicable to the search for similarities in the amino acid sequence of two sequences' Journal of Molecular Biology, 48(3), pp.443-453, 1970

[7] Smith, T.F. and Waterman, M.S. Identification of common molecular subsequences. J. Mol. Biol., 147, pp.195-197, 1981

[8] Pearson, W.R. and Lipman, D.J. 'FASTA: Improved tools for biological sequence comparison', Proceedings of the National Academy of Sciences, USA 85, pp. 2444-2448, 1988

[9] Altschul, S. F., Gish, W., Miller, W., Myers, E.W. and Lipman, D.J. 'Basic Local Alignment Search Tool', Journal of Molecular Biology,215, pp. 403-410, 1990

[10] Altschul, S. F., Madden, T. L., Schaffer, A. A., Zhang, J., Zhang, Z., Miller, W., and Lipman, D. J. 'Gapped BLAST and PSI-BLAST: a new generation of protein database search programs', Nucleic Acid Research, Oxford Journals, 25(17), pp. 3389-3402, 1997

[11] Harrison G. A., Tanner, J. M., Pilbeam D. R., and Baker, P. T. 'Human Biology: An introduction to human evolution, variation, growth, and adaptability', Oxford Science Publications, 1988

[12] Chow, E., Hunkapiller, T., Peterson, J., Waterman, M.S. 'Biological Information Signal Processor', Proceedings of Application-Specific Systems, Architectures, and Processors, ASAP ASAP'91, pp. 144-160, 1991

[13] The Handel-C Language Reference Manual, Agility Plc, http://www.agilityds.com

[14] Kung, S. Y. 'VLSI Array Processors', Prentice-Hall, 1988

[15] Moldovan, D. I. and Fortes, J. A. B. 'Partitioning and mapping of algorithms into fixed size systolic arrays', IEEE Transactions on Computers, 35(1), pp. 1-12, January, 1986

[16] Boeckmann, B., et al., 'The SWISS-PROT protein knowledgebase and its supplement TrEMBL' in 2003 Nucleic Acids Research, Vol.31, pp. 365-370, 2003

[17] RCHTX FPGA PCI Board Reference Manual, Celoxica Plc, http://www.celoxica.com

[18] Sotiriades, E.,Dollas,'A General Reconfigurable Architecture for the BLAST Algorithm', Journal of VLSI Signal Processing 48, 189–208, 2007

A High Performance FPGA-based Implementation of Position Specific Iterated BLAST

Server Kasap
The University of Edinburgh
(+44) 131 650 5592
s.kasap@ed.ac.uk

Khaled Benkrid
The University of Edinburgh
(+44) 131 650 5592
k.benkrid@ed.ac.uk

Ying Liu
The University of Edinburgh
(+44) 131 650 5592
y.liu@ed.ac.uk

School of Electronics and Engineering, Mayfield Road, Edinburgh EH9 3JL, Scotland, UK

ABSTRACT

We present in this paper the first reported FPGA implementation of the Position Specific Iterated BLAST (PSI-BLAST) algorithm. The latter is a heuristic biological sequence alignment algorithm that is widely used in the bioinformatics and computational biology world in order to detect weak homologs. The architecture of our FPGA implementation is parameterized in terms of sequence lengths, scoring matrix, gap penalties and cut-off and threshold values. It is composed of various blmocks each of which performs one step of the algorithm in parallel. This results in high performance implementations, which easily outperform equivalent software implementations by one order of magnitude or more. Furthermore, the core was captured in an FPGA-platform-independent language, namely the Handel-C language, to which no specific resource inference or placement constraints were applied. This makes our core portable across different FPGA families and architectures.

Categories and Subject Descriptors

B.6.0 **[Logic Design]**: General.B.6.1 **[Logic Design]**: Design Styles – *Sequential circuits, Logic arrays, Parallel circuits, Combinational circuits.*

General Terms

Algorithms, Design

1. INTRODUCTION

In Bioinformatics and Computational biology (BCB), biological sequence alignment is a very common task where subject sequences from a large database are aligned to a query sequence to find similarities between the query sequence and the database sequences [1]. A major application of sequence alignment is to infer biological information about a newly discovered sequence from a set of previously annotated sequences. For instance, if a new sequence is found to be similar to a known cancerous sequence, then information regarding the functionality of the new sequence can be inferred, something which is extremely useful in early disease diagnosis and drug engineering. Furthermore, the study of evolutionary development and history of species is essentially based on biological sequence alignment [1] [2]. However, sequence alignment is a computationally intensive operation. Hence, utilization of fast computing platforms is mandatory. Field Programmable Gate Arrays (FPGAs) have been recently proposed as an efficacious and efficient implementation platform for sequence alignment algorithms, thanks to their flexible computing and memory architecture which gives them ASIC-like performance with the added programmability feature [3] [4] [5].

There are various biological sequence alignment algorithms. In this paper, we concentrate on Basic Local Alignment Search Tool (BLAST) [6] which is a local alignment algorithm. It is much faster than ordinary exhaustive dynamic programming algorithms since it is heuristic. Essential background information on the general BLAST algorithm can be found in our other paper [7].Hardware implementation of a variant of BLAST named Position Specific Iterated BLAST (PSI-BLAST) [8] is presented in this paper.

The remainder of this paper will first elaborate on PSI-BLAST algorithm. Then, the design and implementation of our FPGA core for PSI-BLAST will be detailed. Following this, implementation results are presented and then evaluated comparatively with the performance of equivalent software implementations running on a desktop computer. Finally, conclusions are laid out with plans for future work.

2. Position Specific Iterated BLAST (PSI-BLAST)

PSI-BLAST is a profile (or motif) based search method which is more sensitive than Gapped BLAST [8] at detecting distant relationships among query and database sequences. It can identify additional related database sequences that might otherwise be missed by Gapped BLAST. In essence, PSI-BLAST is iterative Gapped BLAST. It consists of following main steps:

1. A database search is conducted with Gapped BLAST using a query sequence and a scoring matrix (BLOSUM 50).
2. All of the subject sequences with local alignment score higher than a specific threshold value are identified and then a multiple alignment of the segments of these high scoring subject sequences and the query sequence is performed. This multiple sequence alignment is detailed to some extent in subsection 2.1.
3. A profile called PSSM (Position Specific Scoring Matrix) is abstracted from the aforementioned multiple sequence alignment. A PSSM is a matrix with n rows and m columns

FPGA'09, February 22–24, 2009, Monterey, California, USA.

where n is the size of the alphabet (n=20 for protein sequences) and m is the length of the query sequence. More information regarding PSSM and its construction from multiple sequence alignment is presented in subsection 2.2.

4. Gapped BLAST is iterated using the obtained PSSM instead of the query sequence itself and the substitution matrix with the aim of identifying a higher number of related database sequences. The way PSSM is utilized in Gapped BLAST is explained in section 3 below.
5. After the second iteration, PSSM is updated by taking newly discovered distant relative database sequences into account through steps 2 and 3.This new PSSM is utilized in subsequent Gapped BLAST iteration.
6. Iterations of Gapped BLAST continue until no more new related database sequences are discovered.

2.1 Multiple Sequence Alignment

After each iteration of Gapped BLAST, the high scoring segments of subject sequences and the query sequence are multiply aligned. The query sequence is used as a template for constructing the multiple alignments. That is to say that each subject sequence segment is first pairwisely and globally aligned to the query sequence and then all these obtained alignments are compiled to form a multiple alignment M.

Columns of M that involve gap characters inserted into the query sequence are ignored so that M has the same length as the query sequence. The PSSM matrix is constructed from the trimmed multiple alignment M as will be explained in the next subsection.

2.2 Construction of Position Specific Scoring Matrix (PSSM)

A PSSM is a motif descriptor which includes a weight (score, probability) for each residue occurring at each position along the motif. It is a 20 by m matrix for protein sequences where m is the length of the motif. The 20 rows of each column specify the probability of finding each of the 20 amino acids at that position in the motif. The M_{jk} element of the PSSM is the score for the j^{th} amino acid at the k^{th} position of the motif.

The PSSM can be constructed from the multiple alignment M described in subsection 2.1 above. The first step of PSSM matrix construction is the reduction of each column of M to form the columns of matrix M_C (motif under consideration). To construct each column C of M_C, the set R of sequences that contribute a residue in column C of M are identified.

We use the data-dependent pseudo-count method proposed in [8] to calculate the values of PSSM elements from M_C. In it, the PSSM score for the j^{th} amino acid at the k^{th} position (M_{jk}) is computed as shown in equation 1, where P_{jk} is the frequency of residue j at the k^{th} position of the M_C matrix and P_j is the background frequency of residue j. Background frequencies of residues are derived from large and carefully selected sets of alignments [9].

$$M_{jk} = \log\left(\frac{P_{jk}}{P_j}\right) \quad (1)$$

The following equation is used to compute P_{jk}:

$$P_{jk} = \frac{\alpha * f_{jk} + \beta * g_{jk}}{\alpha + \beta} \quad (2)$$

where f_{jk} and g_{jk} are the observed frequency and pseudo-count frequency of residue j at position k of M_C, respectively. α and β are the relative weights given to the observed and pseudo-count frequency residues, respectively. In equation 2, α is equal to N_C-1, where N_C is the total number of different residue types, including gaps, observed in the columns of M_C, whereas β is set to the default value of 7. The value of g_{jk} in equation 2 is set to depend on the observed residue frequencies via a scoring matrix S_{ij} (see equation 3) where λ is a natural scale for S_{ij} [8].

$$g_{jk} = p_j \sum_{i=1}^{20} f_{ik} e^{\lambda S_{ij}} \quad (3)$$

As stated above, in PSI-BLAST, it is this obtained PSSM matrix that is used instead of the query sequence and original substitution matrix (e.g. BLOSUM50) in subsequent database search iterations, with the aim of identifying a higher number of related database sequences. The process of database search and PSSM generation is iterated until no more new related database sequences are discovered.

3. HARDWARE IMPLEMENTATION

Figure 2 shows a hardware architecture which implements the PSI-BLAST algorithm. Each block in the architecture implements one step of the algorithm as described in the above sections, except for the pre-processing query sequence step and construction of the Position Specific Scoring Matrix (PSSM) which are implemented by high level application software running on the host computer. The architecture consists of 8 *HitFinderTwoHit* blocks, 2 *UngappedExtender* blocks and 1 *GappedExtender block* all of which are running in parallel. There are also 8 32K x 5 bits subject sequence memories each of which holds a number of subject sequences. Note that each subject sequence memory belongs to one *HitFinderTwoHit* block. Each *HitFinderTwoHit* block is composed of 5 *HitFinder* blocks and 1 *TwoHitMethod* block. Each *HitFinder* block scans its assigned subject sequence memory to find exact matches of the query words in the subject sequences. Each *TwoHitMethod* block performs the two-hit method procedure on hits coming from the 5 *HitFinder* blocks which are in the same *HitFinderTwoHit* block as the *TwoHitMethod* block. Besides these, each *UngappedExtender* block extends the two hits found by its 4 allocated *TwoHitMethod* blocks without allowing gaps, in order to obtain local ungapped alignments. Finally, a single *GappedExtender* block implements the modified Needleman-Wunsch algorithm to produce local gapped alignments from local ungapped alignments obtained in 2 *UngappedExtender* blocks.

The high level application software is explained in subsection 3.1 whereas all of the blocks which constitute the architecture shown in figure 2 are detailed in our other paper [7].

3.1 High Level Application Software

Figure 1 shows the organization of our PSI-BLAST FPGA implementation. Application software running on the host has many duties, the most important of which is the query sequence pre-processing. In brief, the application software finds 3 letter long query words which score at least threshold value T with a scoring matrix when aligned with words extracted from the query sequence. However, in case when there is a constructed PSSM, the application software finds 3 letter long query words which

score at least threshold value T when aligned with the PSSM. Then, the location address of each of these query words in the query sequence is placed at a vacant position in an upper word list and a lower word list pair depending on the 2 most significant letters and 2 least significant letters of the query word, respectively. Note that there are 5 upper word and lower word list pairs.

As it can be seen in figure 1, we produced various FPGA configuration bit files for different threshold and cut-off value parameters. The first task of the application software is to pick the proper bit file, depending on the user-supplied algorithm parameters, from a database of FPGA configurations and load it on to the FPGA chip. Afterwards, the application software runs the hardware configuration in 4 modes. In mode 1, the application software sends one of the 5 upper word and lower word list pairs to each of the 5 *HitFinder* blocks in every *HitFinderTwoHit* block. In mode 2, a number of subject sequences are sent to the 8 available subject sequence memories on FPGA, depending on the subject sequence lengths. In mode 3, the application software sends a query sequence to the FPGA to be stored in the query sequence memories within the 2 *UngappedExtender* blocks and the single *GappedExtender* block. Finally, the execution of the hardware configuration is launched in mode 4. After some time, the FPGA starts sending the high scoring subject sequences to host with their alignment scores. By repeating these steps several times for different subject sequences, we can align all subject sequences in a sequence database to the query sequence (or to PSSM when we have a constructed one).

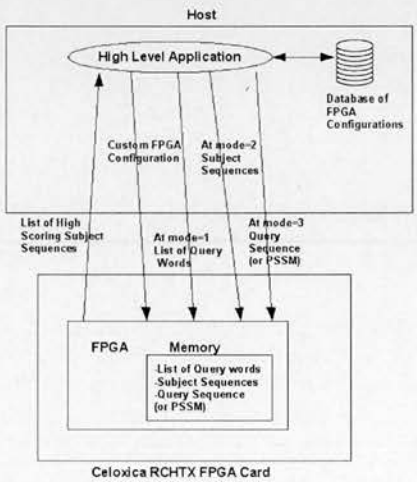


Figure 1. Organization of our Gapped BLAST system

When all subject sequences are aligned, segments of subject sequences which have a local alignment score higher than a specific threshold value are multiply aligned with the query sequence as explained in subsection 2.1 by the application software. Then, the application software constructs the PSSM matrix from the multiple sequence alignment as explained in subsection 2.2 above.

After the construction of PSSM, application software iterates all the aforementioned steps to perform a new Gapped BLAST operation. However, in subsequent iteration, the application software sends the PSSM constructed matrix instead of the query sequence to the FPGA, in mode 3, to be stored in the PSSM memories within the 2 *UngappedExtender* blocks and the single *GappedExtender* block. These Gapped BLAST iterations continue until no more new high scoring subject sequences are found.

4. Results

Our PSI-BLAST design was captured in the Handel C language to which no specific resource inference or placement constraints were applied. Hence, it can be directly targeted to a variety of FPGA platforms. The resulting core was compiled into EDIF by Agility's DK5 SP2 suite from which FPGA bitstreams were generated using Xilinx ISE9.2 tool.

A real hardware implementation of the core was achieved on a Celoxica RCHTX FPGA board [10] which has a Xilinx Virtex 4 (xc4vlx160ff1148-11) FPGA and off-chip memory fitted on it. In our implementation, however, the off-chip memory was not used. The operation of the core was tested on the Swiss-Prot protein sequence database [11] with various query protein sequences.

We have also implemented the PSI-BLAST algorithm in C in order compare our hardware implementation with a pure software implementation. Table 1 presents timing performance figures of both hardware and software implementations for one Gapped BLAST iteration for 9 random query protein sequences of various lengths. Note that all Gapped BLAST iterations take approximately same amount of time. The FPGA hardware was clocked at 15 MHz only and the software implementation was executed on an Intel Centrino Duo 2.2 GHz PC with 2 GB RAM. Furthermore, the same threshold and cut-off values were used in both hardware and software implementations at every step of the algorithm

As it can be seen from table 1, our FPGA implementation result in substantial speed-up compared to software, ranging from 20x to 44x (the speed-up figure depends on the query sequence). The reason behind this high speed-up figure of the FPGA implementation, despite the huge difference in clock frequency, is due to the high level of process parallelism on FPGA.

5. CONCLUSION

In this paper, the detailed FPGA implementation of the PSI-BLAST algorithm has been presented. To our knowledge this is the first FPGA implementation of this algorithm ever reported in the literature. The hardware architecture is composed of various blocks each of which performs a specific step of the algorithm in parallel. Moreover, the FPGA core is parameterized in terms of the sequence lengths, scoring matrix, gap penalties and cut-off and threshold values. The resulting implementation outperforms an equivalent desktop-based software implementation by at least one order-of magnitude. Furthermore, it was designed in the Handel-C language which makes it FPGA-platform-independent. As a result, the same core can be ported to other FPGA architectures from different vendors.

The work presented in this paper is part of a bigger project which seeks to harness the computational performance and re-configurability features of FPGAs in the field of bioinformatics and computational biology. Future work includes a multi-threaded implementation of various flavours of BLAST (including the PSI-BLAST algorithm) and other sequence analysis algorithms with a web interface that allows users to submit queries remotely to an FPGA-based server.

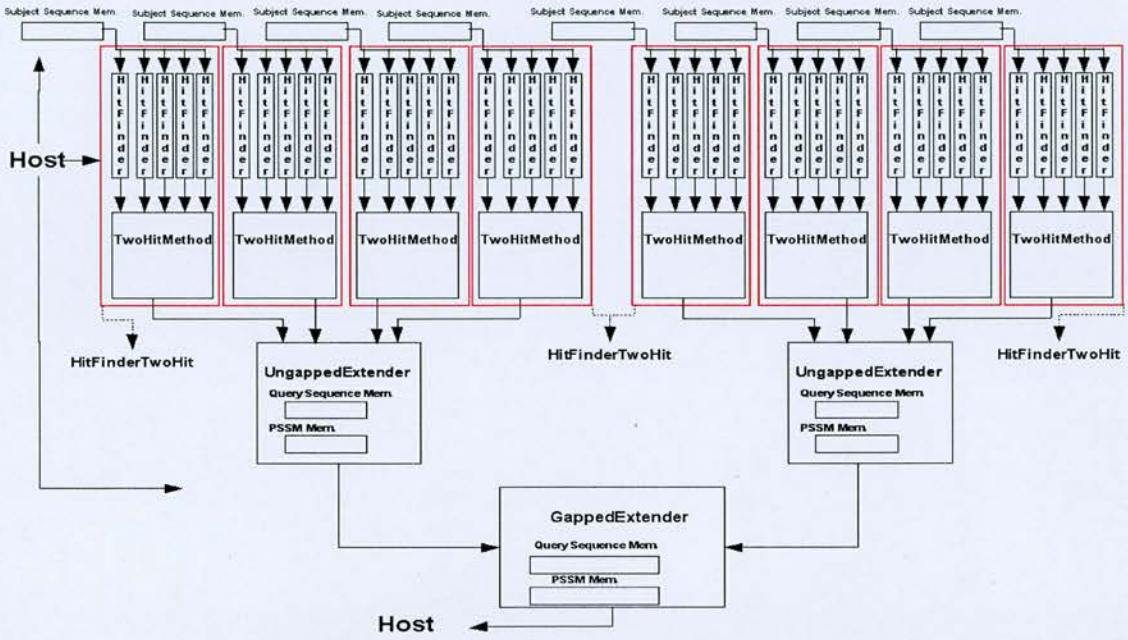


Figure 2. Hardware architecture for the PSI-BLAST algorithm

. Table 1. Timing performance figures of hardware and software implementations for one Gapped BLAST iteration for 9 random protein sequences queried in Swiss-Prot protein sequence database

	No of Residues in Query Sequence	No of Query words	FPGA Execution time (sec)	Software execution time (sec)	FPGA Speed-up
1. Query Sequence	111	116	4.45	91.56	20.58
2. Query Sequence	214	98	5.01	131.93	26.34
3. Query Sequence	368	136	4.32	137.42	31.81
4. Query Sequence	459	263	5.88	211.42	35.96
5. Query Sequence	565	137	5.73	181.48	31.67
6. Query Sequence	635	140	5.36	194.45	36.28
7. Query Sequence	746	117	6.83	233.25	34.15
8. Query Sequence	864	240	7.01	311.23	44.40
9. Query Sequence	985	53	5.33	194.12	36.42

6. REFERENCES

[1] Durbin, R., Eddy, S., Krogh, A., and Mitchison, G., 'Biological Sequence Analysis: Probabilistic Models for Proteins and Nucleic Acids', Cambridge University Press, Cambridge UK, 1998.

[2] Hein, J. 'A New Methodology that simultaneously aligns and reconstructs ancestral sequences for any number of homologous sequences, when a phylogeny is given'. Journal of Molecular Biology, 6, pp.649-668, 1989.

[3] Hoang, D.T. 'Searching genetic databases on Splash 2', in Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, pp. 185-191, 1993.

[4] Gokhale, M. et al. 'Processing in memory: The Terasys massively parallel PIM array', Computer, 28 (4), pp. 23-31, April 1995.

[5] TimeLogic Corporation, 'Decypher Scalable, High Performance Biocomputing Solutions', http://www.timelogic.com.

[6] Altschul, S. F., Gish, W., Miller, W., Myers, E.W. and Lipman, D.J. 'Basic Local Alignment Search Tool', Journal of Molecular Biology,215, pp. 403-410, 1990.

[7] Kasap, S., Benkrid, K., Liu, Y.,' Design and Implementation of an FPGA-based Core for Gapped BLAST Sequence Alignment with the Two-Hit Method', Engineering Letters, Vol. 16, Issue: 3, pp. 443-452,2008.

[8] Altschul, S. F., Madden, T. L., Schaffer, A. A., Zhang, J., Zhang, Z., Miller, W., and Lipman, D. J. 'Gapped BLAST and PSI-BLAST: a new generation of protein database search programs', Nucleic Acid Research, Oxford Journals, 25(17), pp. 3389-3402, 1997.

[9] Yi-Kuo Yu, John C. Wootton, and Stephen F. Altschul, 'The compositional adjustment of amino acid substitution matrices', PNAS, Vol. 100,no 26,pp. 15688-15693, December, 2003.

[10] RCHTX FPGA Board Reference Manual, Celoxica Plc, http://www.celoxica.com.

[11] Boeckmann, B., et al., 'The SWISS-PROT protein knowledgebase and its supplement TrEMBL' in 2003 Nucleic Acids Research, Vol.31, pp. 365-370, 2003.

2009 NASA/ESA Conference on Adaptive Hardware and Systems

An FPGA-Based Web Server for High Performance Biological Sequence Alignment

Ying Liu[1], Khaled Benkrid[1], AbdSamad Benkrid[2] and Server Kasap[1]
[1] *Institute for Integrated Micro and Nano Systems, Joint Research Institute for Integrated Systems, School of Engineering, The University of Edinburgh, The King's Buildings, Mayfield Road, Edinburgh, EH9 3J, UK*
[2] *The Queen's University of Belfast, School of Electronics, Electrical Engineering and Computer Science, University Road, Belfast, BT7 1NN, Northern Ireland, UK*
(y.liu,k.benkrid)@ed.ac.uk

Abstract

This paper presents the design and implementation of the FPGA-based web server for biological sequence alignment. Central to this web-server is a set of highly parameterisable, scalable, and platform-independent FPGA cores for biological sequence alignment. The web server consists of an HTML–based interface, a MySQL database which holds user queries and results, a set of biological databases, a library of FPGA configurations, a host application servicing user requests, and an FPGA coprocessor for the acceleration of the sequence alignment operation. The paper presents a real implementation of this server on an HP ProLiant DL145 server with a Celoxica RCHTX FPGA board. Compared to an optimized pure software implementation, our FPGA-based web server achieved a two order of magnitude speed-up for a pairwise protein sequence alignment application based on the Smith-Waterman algorithm. The FPGA-based implementation has the added advantage of being over 100x more energy-efficient.

1. Introduction

Scanning genome and protein sequence databases is an essential task in molecular biology. Biologists find out the structural and functional similarities between a query sequence and a subject database sequence by scanning the existing genome or protein database sequences, with real world applications in disease diagnosis, drug engineering, bio-material engineering and genetic engineering of plants and animals. There are numbers of biological sequence alignment algorithms with various execution speed/accuracy tradeoffs. Among these, we cite dynamic programming based algorithms [2, 3], heuristic-based algorithms [4, 5], and HMM-based algorithms [6].

The most accurate algorithms for pairwise sequence alignment are exhaustive search dynamic programming (DP)-based algorithms such as the Needleman-Wunsch algorithm [2], and the Smith-Waterman algorithm [3]. The latter is the most commonly used DP algorithm which finds the most similar pair of sub-segments in a pair of biological sequences. However, given that the computation complexity of such algorithms is quadratic with respect to the sequence lengths, heuristics, tailored to general purpose processors, are often introduced to reduce the computation complexity and speed-up bio-sequence database searching. The most commonly used heuristic algorithm for pairwise sequence alignment, for instance, is the BLAST algorithm [4]. In general, however, the quicker the heuristic method is, the worse is the result accuracy. Hence, accurate and fast alignment algorithms need faster computer technologies to keep up with the exponential increase in the sizes of biological databases [1].

Field Programmable Gate Arrays (FPGAs) have been proposed as a candidate technology to solve this problem as they promise the high performance and low power of a dedicated hardware solution while being reprogrammable. Few commercial players are offering real customer solutions for high performance FPGA-based sequence analysis, the most prominent of which are TimeLogic, Progeniq [7, 8] and Mitrionics [9]. TimeLogic, for instance, offers FPGA-based desktop and server solutions for biological sequence analysis applications. Progeniq on the other hand offer mostly small FPGA-based acceleration cards for workstations. Mitrionics offer an FPGA-based server for the BLAST algorithm. Speed-up figures reported by these companies are in the range x20-x80. Nonetheless, these solutions are specific to the hardware and software of choice, and hence do not offer users the flexibility to migrate to other platforms. Moreover, to the best of our knowledge, there is not any academic-based FPGA server solution for biological sequence analysis.

In this paper, we propose a flexible multi-process FPGA-based web server for efficient biological sequence analysis. An FPGA-based web server for pairwise sequence alignment has been realised to demonstrate the

978-0-7695-3714-6/09 $25.00 © 2009 IEEE
DOI 10.1109/AHS.2009.59

benefits of our approach. Central to this server is a highly parameterisable FPGA skeleton for pairwise bio-sequence alignment using dynamic programming algorithms [10].

The remainder of this paper is organised as follows. First, important background information on pairwise bio-sequence alignment algorithms is briefly introduced in section 2. After that, the design of our FPGA-based web server is detailed in section 3. Section 4 then presents a real hardware implementation of a generic DP-based pairwise sequence alignment algorithm on an HP ProLiant DL145 server with a Celoxica RCHTX FPGA board, with detailed implementation results. Finally, conclusions and plans for future work are laid out in section 5.

2. Background

Biological sequences (e.g. DNA or protein sequences) evolve through a process of mutation, selection, and random genetic drift [11]. Mutation, in particular, manifests itself through three main processes, namely: *substitution* of residues (i.e. a residue A in the sequence is substituted by another residue B), *insertion* of new residues, and *deletion* of existing residues. Insertion and deletion are referred to as *gaps*. The gap character "-" is introduced to present the alignment between sequences. There are four ways to indicate the alignment between two sequence s and t as shown below:

(a,a) *denotes a match* $(no\ change\ from\ s\ to\ t)$,
$(a,-)$ *denotes deletion of character* $a\ (in\ s)$,
(a,b) *denotes replacement of* $a\ (in\ s)$ *by* $b\ (in\ t)$,
$(-,b)$ *denotes insertion of chatacter* $b\ (in\ s)$.

For example, an alignment of two sequences s and t (Figure 1) is an arrangement of s and t by position, where s and t can be padded with gap symbols to achieve the same length:

$s: A\ G\ C\ A\ C\ A\ C\ -\ C$
$t: A\ -\ C\ A\ C\ A\ C\ T\ A$

Figure 1. Denotations of the alignment between sequences s and t

(A, A) indicates a match, (G, −) indicates the deletion of G, (−, T) indicates the insertion of T, and (C, A) indicates the replacement of C by A. Gaps should be taken into account when aligning biological sequences.

The most basic pairwise sequence analysis task is to ask if two sequences are related or not, and by how much. It is usually done by first aligning the sequences (or part of sequences) and then deciding whether the alignment is more likely to have occurred because the sequences are related or just by chance. The key issues of the methods are listed below [12]:

- What sorts of alignment should be considered;
- The scoring system used to rank alignments;
- The algorithm used to find optimal (or good) scoring alignments;
- The statistical methods used to evaluate the significance of an alignment score.

The degree of similarity between pairs of biological sequences is measured by a score, which is a summation of odd-log score between pairwise residues in addition to gap penalties. The odd-log scores are based on the statistical likelihood of any possible alignment of pairwise residues, and is often summarised in a substitution matrix (e.g. BLOSUM50, BLOSUM62, PAM). Figure 2 presents a 20 by 20 substitution matrix called BLOSUM50 for amino-acid residues, used for protein sequence alignments.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	5	-2	-1	-2	-1	-1	-1	0	-2	-1	-2	-1	-1	-3	-1	1	0	-3	-2	0
R	-2	7	-1	-2	-4	1	0	-3	0	-4	-3	3	-2	-3	-3	-1	-1	-3	-1	-3
N	-1	-1	7	2	-2	0	0	0	1	-3	-4	0	-2	-4	-2	1	0	-4	-2	-3
D	-2	-2	2	8	-4	0	2	-1	-1	-4	-4	-1	-4	-5	-1	0	-1	-5	-3	-4
C	-1	-4	-2	-4	13	-3	-3	-3	-3	-2	-2	-3	-2	-2	-4	-1	-1	-5	-3	-1
Q	-1	1	0	0	-3	7	2	-2	1	-3	-2	2	0	-4	-1	0	-1	-1	-1	-3
E	-1	0	0	2	-3	2	6	-3	0	-4	-3	1	-2	-3	-1	-1	-1	-3	-2	-3
G	0	-3	0	-1	-3	-2	-3	8	-2	-4	-4	-2	-3	-4	-2	0	-2	-3	-3	-4
H	-2	0	1	-1	-3	1	0	-2	10	-4	-3	0	-1	-1	-2	-1	-2	-3	2	-4
I	-1	-4	-3	-4	-2	-3	-4	-4	-4	5	2	-3	2	0	-3	-3	-1	-3	-1	4
L	-2	-3	-4	-4	-2	-2	-3	-4	-3	2	5	-3	3	1	-4	-3	-1	-2	-1	1
K	-1	3	0	-1	-3	2	1	-2	0	-3	-3	6	-2	-4	-1	0	-1	-3	-2	-3
M	-1	-2	-2	-4	-2	0	-2	-3	-1	2	3	-2	7	0	-3	-2	-1	-1	0	1
F	-3	-3	-4	-5	-2	-4	-3	-4	-1	0	1	-4	0	8	-4	-3	-2	1	4	-1
P	-1	-3	-2	-1	-4	-1	-1	-2	-2	-3	-4	-1	-3	-4	10	-1	-1	-4	-3	-3
S	1	-1	1	0	-1	0	-1	0	-1	-3	-3	0	-2	-3	-1	5	2	-4	-2	-2
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	2	5	-3	-2	0
W	-3	-3	-4	-5	-5	-1	-3	-3	-3	-3	-2	-3	-1	1	-4	-4	-3	15	2	-3
Y	-2	-1	-2	-3	-3	-1	-2	-3	2	-1	-1	-2	0	4	-3	-2	-2	2	8	-1
V	0	-3	-3	-4	-1	-3	-3	-4	-4	4	1	-3	1	-1	-3	-2	0	-3	-1	5

Figure 2. The Blosum50 substitution matrix

The gap penalty depends on the length of gaps and is often assumed independent of the gap residues. There are two types of gap penalties, known as linear gaps and affine gaps. The linear gap is a simple model with constant gap penalty, denoted as:

$$Penalty(g) = -g*d,$$

where g is the length of gaps and d is the constant penalty for each single gap. Affine gaps consist of opening gap penalties and extension gap penalties. The constant penalty value d for opening a gap is normally bigger than the penalty value e of extending a gap. Affine gaps are formulated as:

$$Penalty(g) = -d-(g-1)*e$$

Since it is often the case that a few gaps are as frequent as a single gap, the affine gap model is much more realistic than the linear gap model. The following however presents dynamic programming algorithms in the case of

linear gaps for the sake of clarity. The extension to the case of affine gaps is straightforward [12].

2.1 Dynamic Programming Algorithms

The Needleman-Wunsch (NW) and Smith-Waterman (SW) algorithms are two widely used dynamic programming algorithms for pairwise biological sequence alignment. Needleman-Wunsch is a global alignment algorithm, which is suitable for small sequences, as it aligns the sequences from the beginning to the end. In the case of longer sequences, Needleman-Wunsch introduces too much gap penalty noises that reduce the accuracy of the alignment. Hence, the Smith-Waterman algorithm is used to avoid this problem by looking for similar segments (or subsequences) in sequence pairs (the so-called local alignment problem). In both cases, however, an alignment matrix is computed by a recursion equation with different initial values (see Equation 1 below for the Needleman-Wunsch case, and Equation 2 in the case of the Smith-Waterman algorithm). Here, an alignment between two sequences X = $\{x_i\}$ and Y= $\{y_j\}$ is made.

$$F(i,j)=\max\begin{cases}F(i-1,j-1)+s(x_i,y_j),\\F(i-1,j)-d,\\F(i,j-1)-d.\end{cases} \quad (1)$$

$$F(i,j)=\max\begin{cases}0\\F(i-1,j-1)+s(x_i,y_j),\\F(i-1,j)-d,\\F(i,j-1)-d.\end{cases} \quad (2)$$

From the recursion equations, the alignment score is obtained as the largest value of three alternatives:

- An alignment between x_i and y_j, in which case the new score is $F(i-1,j-1)+s(x_i,y_j)$ where $s(x_i,y_j)$ is the substitution matrix score or entry for residues x_i and y_j.
- An alignment between xi and a gap in Y, in which case the new score is $F(i-1,j)-d$, where d is the gap penalty.
- An alignment between y_j and a gap in X, in which case the new score is $F(i,j-1)-d$, where d is the gap penalty.

The dependency of each cell can be clearly shown in Figure 3. Here, each cell on the diagonal of the alignment matrix is independent of each other, which allows systolic architectures to be introduced to increase the parallelism and speed up the computation of dynamic programming algorithms.

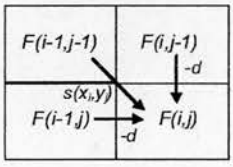


Figure 3. Data dependency of dynamic programming algorithms

As mentioned earlier, affine gap penalties provide a more realistic model of the biological phenomenon of residue insertions and deletions. The affine gap penalty is defined using two constants d and e as follows:

*Penalty(g)=-d-(g-1)*e, where g is the gap length.*

Multiple values of each pair of residue (i,j) need to be computed instead of just one in the affine gap case, with recursive equations similar to the ones for linear gaps, both for local and global alignment [12].

2.2 BLAST

The BLAST algorithm is a heuristic algorithm for pairwise sequence alignment, developed by the National Center for Biotechnology Information (NCBI). The basic idea of NCBI BLAST is to find positions in the subject sequences (the database) which are similar to certain query sequences segments, allowing for insertion, deletion and substitution. These positions are called High-Scoring Pairs (HSPs), which are defined as pairs of aligned segments of sequence pairs that generate an alignment score above a certain threshold T. Starting with these HSPs instead of computing the whole score matrix for pairwise sequences result in substantial computational savings, which makes BLAST searches faster than Smith-Waterman, for instance, on general purpose processors. In general, NCBI BLAST consists of three steps as illustrated in Figure 4 below:

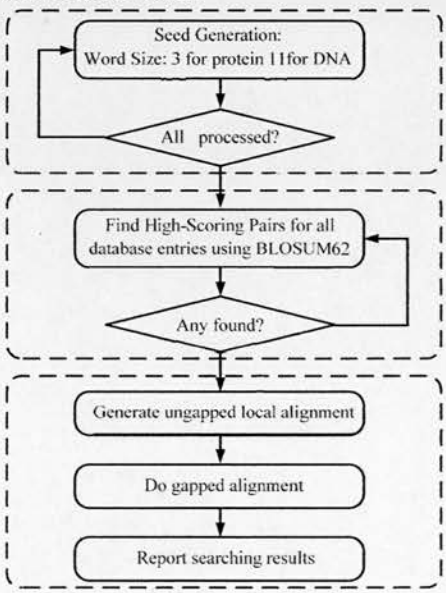


Figure 4 Steps employed by BLAST

Seed generation: A seed is a fragment of fixed length W (W = 3 for protein sequences and W = 11 for DNA sequences) from the query sequence. For a query sequence of length M, the number of seeds generated is M-W+1 as shown in Figure 5.

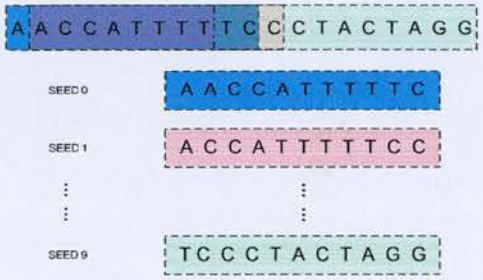


Figure 5. DNA Seed Generation

Hits finding: The second step in the BLAST algorithm is to find matches (potential HSPs) between seeds and the database stream. A substitution matrix is used for proteins in order to obtain the substitution scores for pairwise sequence segments. If the substitution score is above a threshold *T*, the system considers these segments to be High-Scoring Pairs (HSPs).

Hits extension: The final step is to extend the HSPs found in the second step in both directions to complete the alignment. The most widely used implementation of BLAST looks at ungapped alignments only. Nonetheless current versions of NCBI BLAST provide gapped alignments too. The extension process terminates either when it reaches the end of one sequence, or when the score decreases sufficiently to a falloff parameter *F* [13].

NCBI BLAST is able to perform five different similarity searches, namely BLASTn, BLASTp, BLASTx, tBLASTn, and tBLASTx (see Table 1 below).

Table 1. Various searches of BLAST

Search Name	Query Type	Database Type	Translation
BLASTn	Nucleotide	Nucleotide	None
BLASTp	Protein	Protein	None
BLASTx	Nucleotide	Protein	Query
tBLASTn	Protein	Nucleotide	Database
tBLASTx	Nucleotide	Nucleotide	Both

BLASTn is a member of the BLAST program package which searches DNA sequences against DNA database. Due to the high degree of conservation in DNA sequences, the High-Scoring Pairs in the second step of BLASTn are the exact matches between query seeds and database stream. BLASTp is the most widely used program for aligning a Protein sequence against a Protein database. Instead of finding exact matches, it looks for non-identical pairs that generate high similarity scores by using similarity substitution matrices, such as PAM and BLOSUM62. One HSP stands for a pair of similar fragments with a score above the threshold *T*.

The other three members of BLAST searches require translation. A nucleotide sequence can be translated into protein sequences in 6 different frames. Searches are processed against all 6 frames to get the final BLAST result.

3. Our Proposed FPGA-based Web Server for Efficient Biological Sequence Alignment

Figure 6 presents our FPGA-based web server for biological sequence analysis. The web server consists of an HTML based web interface, a MySQL database for storing the queries and results, a list of biological sequence database, a database of FPGA configuration, a host application that services user requests, and FPGA co-processor(s) that accelerate the sequence alignment tasks. The web interface takes all the parameters needed for one unknown query as following:

- *sequence symbol type* i.e. DNA, RNA, or Protein sequences
- *Alignment task* e.g. Smith-Waterman, Needleman-Wunsch, BLASTn, BLASTp.
- *Query sequence*: Here the query sequence length and the alignment task dictate the configuration(s) to be downloaded to the FPGA(s), if need be.
- *match score* i.e. the score attributed to a symbol match. This is supplied in the form of a substitution matrix e.g. BLOSUM matrix.
- *gap penalties*: This could be either linear or affine. The gap penalty is loaded to the FPGA co-processor at run time.
- *match score thresholds*: e.g. the HSPs threshold *T* and the fall-off parameter *F* in the case of the BLAST algorithm.

When the user submits a query, a unique ID is given for checking the result. A MySQL database is used to store all query information with the unique ID into a query list. The host application manages communications between the MySQL database, sequence databases, and FPGA configurations on the one hand, and the FPGA coprocessor(s) on the other hand.

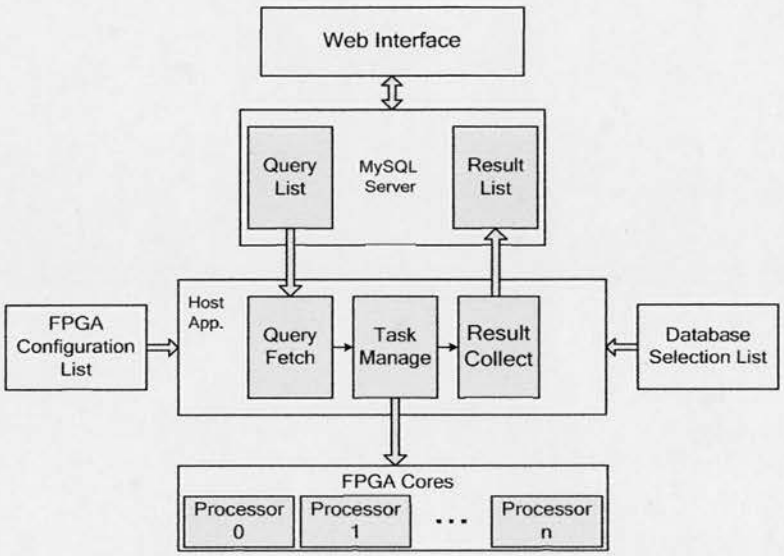


Figure 6. Proposed FPGA-based Web Server for Efficient Biological Sequence Alignment

The FPGA coprocessors are initially configured with the most commonly used configurations by the host application. The subject sequence databases are loaded into host memory. When a query comes, the host application will check if FPGA coprocessors need to be reconfigured. Since many users might be requesting the server at the same time, several processes would run concurrently on the same FPGA chip, and/or across many FPGA chips. Efficient partitioning and scheduling algorithms need to be employed to minimize the average user waiting time. After the configuration of the FPGA co-processor(s), the host application processes the incoming query set before sending it to FPGA coprocessor, and configures the FPGA with query coefficients according to the query sequence submitted. The selected database sequences are sent from the host memory to the FPGA processor(s) for the alignment executions. At the end of processing, alignment results are collected by the host application and stored into a result list in the MySQL database. The users can obtain the results through their unique ID via a web interface to the MySQL database.

4. Real Implementation: an FPGA-based Web Server for DP-based Pairwise Sequence Alignment

In order to demonstrate our proposed FPGA-based web server, we implemented a prototype web server on an HP ProLiant DL145 server with a Celoxica RCHTX FPGA board, and run a generic FPGA skeleton for DP-based pairwise sequence alignment on it. The following first describes our generic skeleton, before implementation results are presented.

4.1 A Generic FPGA Skeleton for DP-based Pairwise Sequence Alignment

Figure 7 presents a generic systolic architecture for DP-based pairwise sequence alignment [14]. Each processing array (PE) in the array consists of control logic, coefficient RAM and computation logic.

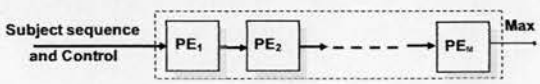


Figure 7. Linear Processor Elements array architecture of pairwise biological sequence alignment with single pass

Given the DNA sequences comprise only four nucleotides, whereas protein sequences comprise fivefold larger variety of sequence characters, it is much easier to detect patterns of sequence similarity between protein sequences than between DNA sequences [13]. Pearson [14, 15, 16] has proven that searches with a protein sequence encoded by a DNA sequence against a DNA

sequence database yield far fewer significant matches than searches using the corresponding protein sequence. Hence, we conducted the web server implementation on Smith-Waterman algorithm for protein sequence and extended it to all the variances of DP based bio-sequence alignment. Figure 8 illustrates the PE architecture/behaviour for a DP pairwise sequence alignment algorithm with linear gap penalty. Control logic separates the different database subject sequence calculations. Coefficient RAMs are run-time reconfigurable according to the query sequence in hand (one column of substitution matrix coefficients are stored in one RAM). Given that each coefficient is 2-bit wide for DNA and 5-bit wide for proteins, and that there are 21 elements in one column, distributed memory on FPGAs is used to implement the coefficient RAMs. The computation logic is tailored to the parameters of dynamic programming algorithms in hand, i.e. sequence type, alignment type, gap penalty, etc.

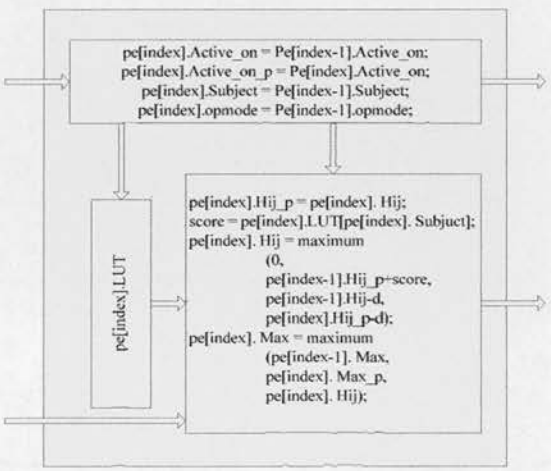


Figure 8. Processor Element for pairwise biological sequence alignment using the Smith-Waterman algorithm

As each processing element copes with one residue from the query sequence, the computation complexity of a single pass linear array of processing elements is reduced from quadratic to linear, as denoted below:

$$O(m, n) = m + n - 1 \quad (4)$$

where $O(m,n)$ represents the number of clock cycles spent on each query sequence with length n aligned to a subject sequence database of m residues.

In the case of longer sequences, due to the possible resource limitation of the FPGA chip in hand, our design can be tailored to cope with this by folding the systolic array and using several alignment passes instead of just one, at the expense of longer processing time (Figure 9). The complexity of the design with multiple passes is denoted as:

$$O(m,n) = m * \text{passNum} + n - 1 \quad (5)$$

where passNum presents the amount of passes that one query needs. passNum is calculate as follows:

$$passNum = \left\lceil \frac{querylength}{PENum} \right\rceil \quad (6)$$

where $querylength$ is the length of the query sequence, and $PENum$ is the maximum number of PEs that can be fitted into the FPGA chip in hand. Hence, the single pass design can be considered as a special case of multiple pass design with $passNum = 1$, which gives the best throughput.

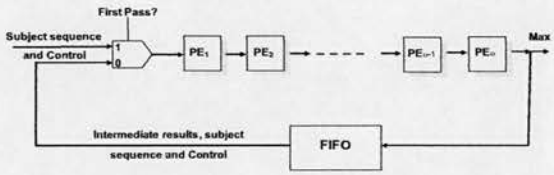


Figure 9. Linear Processor Element (PE) array architecture for pairwise biological sequence alignment with multiple passes

The above described skeleton has been captured in Handel-C, which makes it FPGA-platform independent. Indeed, since our Handel-C description did not use any specific FPGA resource or placement constraints, it can be easily retargeted to a variety of FPGA platforms e.g. from Xilinx and Altera.

Compared to previously reported FPGA-based biological sequence alignment accelerators [18-22], our FPGA-based web server solution has been designed to be platform-independent with a service-based model of operation in mind. Detailed comparison between our implementation and previously reported FPGA-based biological sequence alignment accelerators is presented in [24].

4.2 Implementation Results

As mentioned above, our FPGA-based web server has been targeted to an HP ProLiant DL145, which has an AMD 64bit processor and a Celoxica RCHTX FPGA board. The latter has a Xilinx Virtex 4 LX160-11 FPGA chip on it. All data transfer between the host application and FPGA coprocessor pass through the Hyper-Transport interface and is supported by the DSM library in Handel-C. Celoxica DK5 suite and Xilinx ISE 9.1i were used to compile our Handel-C code into FPGA configurations (bitstreams).

The performance of the resulting FPGA-based web server is illustrated with a single core implementation of the Smith-Waterman algorithm on the Virtex-4 FPGA,

using the Swiss-Prot Protein database as a sequence database. A single processing element of a systolic array implementing the Smith-Waterman algorithm with linear gap penalty and a processing word length of 16 bits, consumes ~110 slices. Consequently, we were able to fit in ~500 PEs on a Xilinx Virtex 4 LX160 -11 FPGA.

We also compared our FPGA implementation with equivalent optimised software implementation running on the Dual-Core AMD Opteron™ processor 2218, and captured in C++. A set of 100 queries of 250 residues is chosen to align against the latest Swiss-Prot database [23] as a test bench. The processing time of a single core FPGA implementation was 188.4 seconds (i.e. 3min 8sec) while it was 28840 seconds (i.e. 8 hours 36sec) for the software implementation. This means that our FPGA implementation outperforms an equivalent software implementation by 150x.

We also performed power consumption measurements for both hardware and software implementations, using a power meter. Factoring the execution time, the total energy consumed by our FPGA-based web server implementation was 2.09 Wh (i.e. 7542 Joule), whereas the software implementation consumed 360.5 Wh (i.e. 1297800 Joule), for the same set of queries. This means that our FPGA-based web server implementation is 172x more energy efficient than the software implementation. This shows that FPGAs offer a high performance and low power platform for biological sequence alignment applications.

It is worth noting that since the computation complexity of the software implementation grows quadratically with the sequence sizes, while it grows linearly in the case of FPGA implementation, the more PEs we can fit on FPGAs, the better the speed up figure we get.

5. Conclusion

In this paper, we have presented the design and implementation of an FPGA-based web server for biological sequence alignment, where FPGA coprocessors are used for the acceleration of sequence alignment tasks. A demonstrator FPGA-based web server was implemented on an HP ProLiant DL145 server with a Celoxica RCHTX FPGA board containing one Xilinx Virtex 4 LX160-11 FPGA chip. Using a highly parameterisable FPGA skeleton for pairwise sequence alignment, our FPGA-based web server implementation outperformed an equivalent optimised software implementation of the Smith-Waterman algorithm by 150x, while consuming 172x less energy. This shows FPGAs to be an efficient and efficacious computing platform for biological sequence alignment applications.

The work presented in this paper is part of a bigger effort by the authors which aims to harness the computational performance and reprogrammability features of FPGAs in the field of Bioinformatics and Computational Biology. Future work includes the extension of the library of biological sequence analysis algorithms implemented on the server including profile HMM searches, various BLAST algorithm variations, and phylogenetic tree construction algorithms.

6. References

[1] Benson, D. A., Karsch-Mizrachi, I., Lipman, D. J., Ostell, J., Rapp, B. A. and Wheeler, D.L. '*Genbank*'. *Nucleic Acids Res.*, **28**, 15-18. 2000

[2] Needleman, S. and Wunsch, C. '*A general method applicable to the search for similarities in the amino acid sequence of two sequences*', J. Mol. Biol., 48(3), (1970), 443-453

[3] Smith, T.F. and Waterman, M.S. '*Identification of common molecular subsequences*'. Journal of Molecular Biology, 147, 195-197, 1981

[4] Altschul, S. F., Gish, W., Miller, W., Myers, E.W. and Lipman, D.J. 'Basic Local Alignment Search Tool', Journal of Molecular Biology, 215, pp. 403-410, 1990

[5] Pearson, W.R. and Lipman, D.J. '*FASTA: Improved tools for biological sequence omparison*', Proceedings of the National Academy of Sciences, USA 85, (1988), pp. 2444-2448.

[6] HMMER user's guide: biological sequence analysis using pro hidden Markov models. http://hmmer.wustl.edu (1998)

[7] Gollery, M., Rector, D., Lindelien, J, *'TLFAM-Pro: A New Prokaryotic Protein Family Database'*, http://www.timelogic.com/ , TimeLogic Corporation,2009

[8] '*BioBoost for Bioinformatics'*, http://www.progeniq.com/, progeniq Private Limited, 2009

[9] *'Mitrion-C Open bio-project'*, http://www.mitrionics.com, mitrionics, 2009

[10] Benkrid, K., Liu, Y., and Benkrid A., '*Design and Implementation of a Highly Parameterised FPGA-Based Skeleton for Pairwise Biological Sequence Alignment'*. FCCM'07, pp. 275-278, 2007

[11] Harrison G. A., Tanner, J. M., Pilbeam D. R., and Baker, P. T. *'Human Biology: An introduction to human evolution, variation, growth, and adaptability'*, Oxford Science Publications, 1988

[12] Durbin, R., Eddy, S., Krogh, A., and Mitchison, G., '*Biological Sequence Analysis: Probabilistic Models for Proteins and Nucleic Acids'*, Cambridge University Press, Cambridge UK, 1998

[13] Kasap, S., Benkrid, K., Liu, Y., *'High performance FPGA-based core for BLAST sequence alignment with the two-hit method'*, BioInformatics and BioEngineering 2008, pp. 1-7, 2008.

[14] Mount D.W., *'Bioinformatics: Sequence and Genome Analysis'*, Cold Spring Harbour Laboratory Press, Cold Spring Harbour, New York, USA, 2001

[15] Pearson W.R., *'Comparison of methods for searching protein sequence databases'*, 1995. *Protein Science e.* **4:** 1150–1160.

[16] Pearson W.R., *'Effective protein sequence comparison'*, 1996, *Methods Enzymol.* **266:** 227–258.

[17] Pearson W.R., *'Flexible sequence similarity searching with the FASTA3 program package'*, 2000, *Methods Mol. Biol.* **132:** 185–219

[18] Yamaguchi, Y., Maruyama, T., and Konagaya, A. *'High Speed Homology Search with FPGAs'*, Proceedings of the Pacific Symposium on Biocomputing, pp.271-282, 2002.

[19] VanCourt, T. and Herbordt, M. C. *'Families of FPGA-Based Algorithms for Approximate String Matching'*, Proceedings of Application-Specific Systems, Architectures, and Processors, ASAP'04, pp. 354-364, 2004

[20] Oliver, T., Schmidt, B. and Maskell, D. *'Hyper customized processors for bio-sequence database scanning on FPGAs'*, Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays

[21] Bojanic, S., Caffarena, G., Pedreira., C., and Nieto-Taladriz, O., *'High Speed Circuits for Genetics Applications'*, Proceedings of the 24th International Conference on Microelectronics (MIEL 2004), Vol. 12, pp. 517-524, 2004

[22] Puttegowda, K., Worek, W., Pappas, N., Dandapani, A., and Athanas, P., *'A Run-Time Reconfigurable System for Gene-Sequence Searching'*, Proceedings of the 16th International Conference on VLSI Design VLSI'03, pp. 561–566,2006.

[23] http://www.uniprot.org/, Uniprot, 2008

[24] K. Benkrid, Y. Liu, A. Benkrid, A Highly Parameterised and Efficient FPGA-Based Skeleton for Pairwise Biological Sequence Alignment, IEEE Transactions on Very Large Scale Integration Systems, v 17, p 561-570, No. 4, 2009

A High Performance FPGA-based Core for Phylogenetic Analysis with Maximum Parsimony Method

Server Kasap and Khaled Benkrid

System Level Integration Group, School of Engineering, the University of Edinburgh

Edinburgh Research Partnership, Institute for Integrated Systems, King's Buildings, Mayfield Road, Edinburgh EH9 3JL, Scotland, UK

(s.kasap, k.benkrid)@ed.ac.uk

***Abstract*—We present in this paper the detailed FPGA design of the Maximum Parsimony method for molecular-based phylogenetic analysis and its implementation on a Xilinx Virtex-4 FPGA chip. This is the first FPGA implementation of this method for nucleotide sequence data ever reported in the literature. The hardware architecture consists of a linear systolic array composed of 20 processing elements each of which performing the Sankoff's algorithm for a different tree topology in parallel. This array computes the scores of all theoretically possible trees for a given number of taxa in several iterations. The currently supported maximum number of taxa is 12 but this number can be easily increased. Furthermore, the resulting implementation outperforms an equivalent desktop-based software implementation (using PAUP software) by several orders of magnitude. The speed-up values achieved by the hardware implementation can reach over 20,000x for the 12-taxa case. This is achieved through harnessing both coarse-grain and fine-grain parallelism available in the algorithm and corresponding hardware implementation platform.**

I. Introduction

Phylogenetic analysis is the investigation of the evolution and relationships among organisms which is widely used in the fields of system biology and comparative genomics [1]. It is particularly vital in drug and vaccine development. In molecular based phylogenetic analysis, the relationship between species is estimated by inferring the common history of their genes and then phylogenetic trees are constructed to illustrate evolutionary relationships among genes and organisms [1] [2].

However, phylogenetic tree construction is a computationally intensive operation and desktop computers alone cannot be relied upon to perform this task within acceptable execution times with the number of theoretically possible tree topologies growing at an exponential rate with the number of species under consideration (e.g. over 30 hours for the case of 12 species). Hence, it is mandatory to utilize faster computing platforms to bring the execution time of this application to a reasonable range. Field Programmable Gate Arrays (FPGAs) have been recently proposed as an efficacious and efficient implementation platform for phylogenetic analysis due to their flexible computing and memory architecture which gives them ASIC-like performance with the added programmability feature [3] [4] [5] [6] [7] [8] [9]. As such, they form the implementation platform of choice in this paper.

There are various phylogenetic tree construction and phylogenetic analysis methods using different strategies. In this paper, we concentrate on the Maximum Parsimony (MP) method which is one of the most widely used and most accurate tree construction method. The design and implementation of an FPGA core for parsimony analysis employing Sankoff's dynamic programming algorithm is presented in this paper. A real hardware implementation of the designed core was achieved on a Xilinx Virtex-4 FPGA chip. To our knowledge, this is the first FPGA implementation of this method for nucleotide sequence data ever reported in the literature.

The remainder of this paper will first present essential background information on the Maximum Parsimony (MP) method for molecular based phylogenetic tree construction. Then, the design and implementation of our FPGA core for the MP method will be elaborated. Following this, implementation results are presented and then evaluated comparatively with equivalent software implementations running on a desktop computer. Finally, conclusions are laid out with plans for future work.

II. Maximum Parsimony

Molecular-based phylogenetic analysis estimates the relationship between species by inferring the common history of their genes through comparing homologous sites with each other. For this reason, sequences under investigation are multiply aligned by some specific algorithms so that homologous sites form columns in the alignment. These alignments are used to construct phylogenetic trees which illustrate evolutionary relationships among genes and organisms.

Diagrams depicting the relationship of species resemble the structure of a tree. Hence, they are called phylogenetic trees. There are two types of phylogenetic tree, rooted or unrooted. Rooted phylogenetic trees are drawn with a root to the left. Figure 1 shows an example unrooted phylogenetic tree. It can be seen that phylogenetic trees are strictly bifurcated (binary).

978-1-4244-4377-2/09/$25.00 © 2009 IEEE

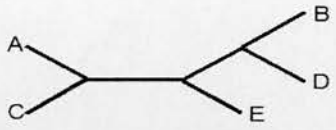


Fig. 1. Unrooted phylogenetic tree

There are various methods to reconstruct phylogenetic trees from nucleotide acid sequence alignments in molecular data based phylogenetic analysis. In this paper, the maximum parsimony (MP) method [10] was employed to find the best phylogenetic tree for a given number of taxa.

The maximum parsimony (MP) method is one of the most widely used discrete character method in molecular phylogenetic analysis [2]. It operates on a character-state matrix which is typically an aligned set of DNA or protein sequences where the states are the nucleotides (i.e. A, C, G, and T) for DNA sequences and symbols of 20 amino acids for protein sequences.

The MP method operates by defining an objective function which returns a score for any input tree topology. This tree score is used to rank all possible trees according to the chosen optimality criterion to find the optimal tree topologies. This process will be discussed in the following subsections .

A. Parsimony Analysis

Parsimony criterion is the number of character changes required to explain all nodes of a tree at every sequence position for a given set of aligned sequences. The total amount of character change required by any given tree is called the length of that tree. In parsimony analysis, the aim is to find the tree topologies with the smallest length. Calculating the length of a given tree will be explained and illustrated later in this subsection.

An unrooted binary tree contains T-2 internal nodes, 2T-3 branches and T terminal nodes representing sequences of T taxa. The length of an arbitrarily chosen tree r under parsimony criterion is given by the following equation:

$$L(r)=\sum_{j=1}^{N} l_j \quad (1)$$

In equation 1, N is the number of sites in the sequence alignment and l_j is the length for single site j which is the minimum amount of character change implied by a reconstruction where a character-state x_{ij} is assigned to each node i for each site j. Note that character-state assignments of the terminal nodes are fixed by the input sequences of T taxa. Equation 2 shows the calculation of l_j.

$$l_j=\sum_{k=1}^{2T-3} C_{a(k),b(k)} \quad (2)$$

In equation 2, a(k) and b(k) represent the states assigned to the nodes at either end of branch k whereas c_{xy} is the cost of change from state x to state y. There are various cost schemes which can be represented as a cost matrix that assigns a cost for the change between each pair of character states. An important point is that cost matrices are symmetric meaning that c_{xy} is equal to c_{yx}. As a consequence, the length of a tree is the same regardless of the position of the root. Therefore, the search among tree space can be conducted over unrooted trees rather than rooted trees.

It is possible to calculate a tree length for one site by evaluating all possible r^{T-2} character-state reconstructions where r is the number of states (r = 4 for DNA sequences or r =20 for protein sequences).However, there is a need for better ways to determine the minimum lengths since the evaluation of r^{T-2} reconstructions will take a considerable amount of time and storage when the number of taxa under consideration grows. For this purpose, we will employ a straightforward dynamic programming algorithm namely Sankoff's algorithm [11] which is illustrated in subsection II-B below.

B. Sankoff's Algorithm

Dynamic programming algorithms operate by solving a set of subproblems and then assembling those solutions to find an optimal solution for the whole problem. In the case of Sankoff's algorithm, the best length achievable for each subtree is determined given each of the possible state assignments to each node, while moving from the tips toward the root of the tree. An optimal length for the full tree is obtained when the root is reached.

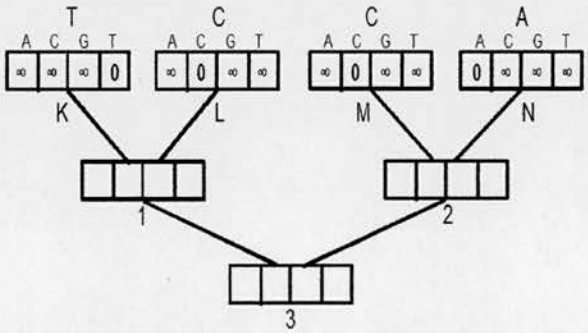


Fig. 2. An example tree topology with conditional subtree length vectors for each node

Sankoff's algorithm operates on conditional subtree length vectors which are depicted by the rectangular boxes in the tree shown in figure 2. It can be seen that for each node i, there is an associated conditional subtree vector $\mathbf{S}_i$ containing the minimum possible lengths s_{ik} of the subtree descending from node i if it is assigned state k. Working from the tips toward the root, the algorithm proceeds by filling in the vector at each node based on the values assigned to the pair of vectors above the regarding node. Note that for the terminal nodes, vectors are initialized to 0 for the states actually observed in the sequence alignment or to infinity otherwise. The algorithm will be illustrated based on the tree shown in figure 2. An important point is that for symmetric cost matrices, an unrooted tree can be arbitrarily rooted to determine the minimum tree length in this algorithm.

We start with the calculation of the vector values of node 1. For each element k of this vector, the costs associated with each of the four possible state assignments to each of the child

nodes K and L and the cost needed to reach these states from state k are considered. For node 1, these calculations are simple since it is ancestral to two terminal nodes. Hence, only one state needs to be considered for each child node. For example, the minimum length of the subtree descending from node 1 assuming that state A is assigned to node 1 is equal to the sum of the cost of a change from A to T in the left branch and the cost of a change from A to C in the right branch ($s_{1A} = c_{AT} + c_{AC} = 4 + 4 = 8$). In the same manner, s_{1C} is the sum of c_{CT} (left branch) and c_{CC} (right branch) giving the value of 1. Continuing like this, we obtain the entire conditional subtree length vector for node 1 as shown in figure 3. With the same procedure, we compute the elements of the vector for node 2 as shown in figure 4. On the other hand, calculations for node 3 are more complicated since we must consider each of the four state assignments to each of the child nodes 1 and 3 for each state k at its node. Figure 5 shows the computation of the conditional subtree length vector for node 3.

The conditional vector $\mathbf{S_3}$ contains the minimum possible lengths for the full tree given each of the four possible state assignments to the root. The minimum of these tree lengths is the tree length we seek, which is 5 in our case as can be seen in figure 5. Note that different rooting of the tree in figure 2 would yield the same length.

This algorithm provides a way to calculate the minimum tree length for any character on any tree under any cost scheme. The total length of a given tree can be computed by repeating the mentioned procedure for each character in the sequence alignment and then adding up all of the obtained minimum lengths for the characters which can be multiplied beforehand by different weights depending on the importance of the characters in the alignment.

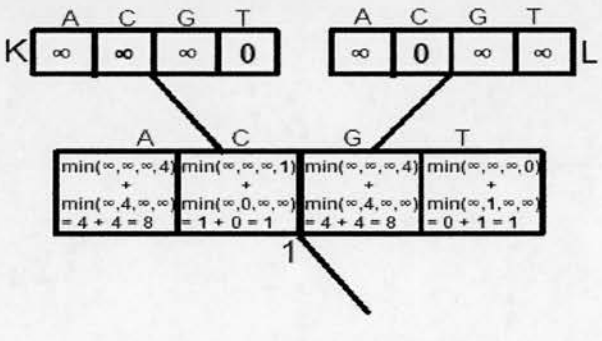


Fig. 3. Calculation of the conditional subtree length vector for node 1

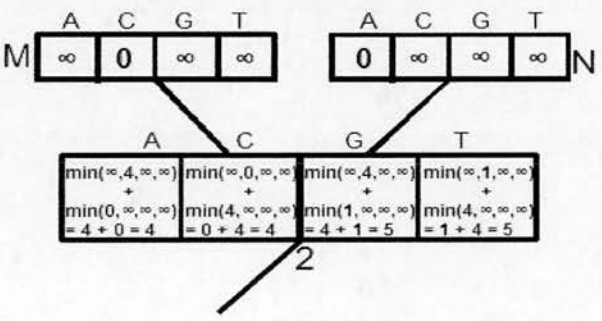


Fig. 4. Calculation of the conditional subtree length vector for node 2

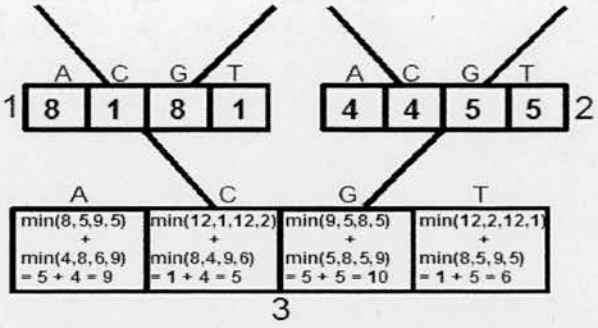


Fig. 5. Calculation of the conditional subtree length vector for node 3

C. *Searching for Optimal Trees*

Since the length of a tree under parsimony criterion can be calculated using Sankoff's algorithm, the search over the tree space can now be started to find the optimal tree. However, there is a need for an algorithm to generate all possible trees to be evaluated under parsimony criterion. Such an algorithm recursively adds the n^{th} taxon in a stepwise manner to all possible trees containing the first n-1 taxa until all T taxa have been joined. The details of this algorithm can be found in [1].

D. *PAUP*

PAUP (Phylogenetic Analysis Using Parsimony) [12] is a phylogenetic analysis program using NEXUS format for input data files. It includes support for the maximum parsimony, maximum likelihood and distance methods as well as some additional capabilities. Details of PAUP can be found in its user manual, command reference manual and quick start tutorial in [13].

III. Hardware Implementation

Sankoff's algorithm requires calculation of the conditional subtree length vector for every internal node in a tree. However, some of these vectors can be computed at the same time. For example, in the 10-taxa tree shown in figure 6, computations for nodes on the same line can be done in parallel. Vectors of nodes on different lines are computed consecutively starting from the first line until the root node is reached. FPGAs can take advantage of this parallelism of Sankoff's algorithm to accelerate it by computing several node vectors concurrently. For the tree topology in figure 6, FPGA hardware would calculate 2, 4 and 2 node vectors in parallel in the first, second and third clock cycles, respectively. In the fourth clock cycle, a vector for the root node would be calculated to obtain the minimum length (score) of the tree. Hence, the score of the tree is computed in four clock cycles in total rather than the nine cycles required in the case of sequential node calculations. An important point is that the tree under consideration should be rooted in a way so that the left and right subtrees of the rooted tree will have almost the same number of taxa to maximize the available parallelism (i.e. tree balancing).

Figure 7 shows the hardware architecture which computes the subtree length vectors of the nucleotides (i.e. A, C, G, and T). In this architecture, registers *LregA*, *LregC*, *LregG* and

LregT represent the elements of the vector of the left hand side upper node (e.g. node 1 in figure 5) whereas registers *RregA*, *RregC*, *RregG* and *RregT* represent the elements of the vector of the right hand side upper node (e.g. node 2 in figure 5).

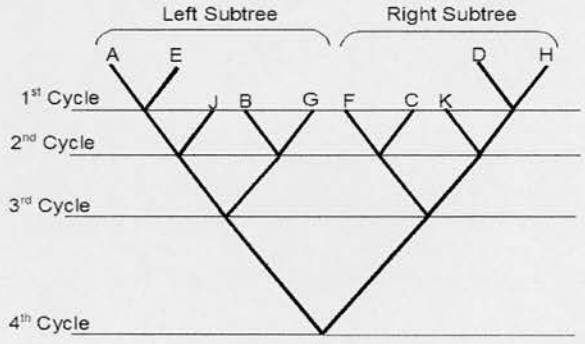


Fig. 6. Tree topology illustrating the parallelism of the Sankoff's algorithm

Each of these registers are added up with three different specific cost values (i.e. C_{A2C}, C_{A2G}, C_{A2T}, C_{C2G}, C_{C2T}, C_{G2T}) to obtain three subscores (e.g. S_A2C_L, S_A2G_L, S_A2T_L for *LregA*) and then each register and its associated three subscores (e.g. S_C2A_L, S_G2A_L, S_T2A for *LregA*) are inputted to the combinational block *Min* to find the minimum of these values MinX_Y (X=A, C, G or T and Y=L or R). Furthermore, two minimum values for each nucleotide (e.g. MinA_L and MinA_R for A) are added to obtain the scores for each nucleotide (i.e. S_A, S_C, S_G, S_T) which are the elements of the vector of the target node (e.g. node 3 in figure 5).

Since we have an architecture which computes the conditional vector, we can use it within a linear systolic array to implement the complete Sankoff's algorithm in FPGA as explained in subsection III-A below. Also, subsection III-B elaborates on the inner structure of the processing element constituting this array.

A. Linear Systolic Array Implementation of the Sankoff's Algorithm

Figure 8 shows a linear systolic array which implements the Sankoff's algorithm. It is composed of several processing elements PE_i, each of which contains a number of sub-elements with similar architecture as shown in figure 7, in order to compute node vector values. Each *PE* calculates the score of a different tree topology in parallel independently from each other. Hence, the total number of *PE*s is equal to the number of theoretically possible tree topologies for the given number of taxa.

The architecture in figure 8 also comprises two *FIFO*s and an *FSM*. The *Input FIFO* is fed by high level application software running on the host computer with cost matrix, tree topology and sequence alignment data in respective order. Concurrently, the linear array reads the *Input FIFO* to first get the values of the chosen cost matrix which are then shifted through the processing elements within the array. Following this, tree topology vectors whose number is equal to the number of possible tree topologies are read and shifted through the array independently to configure each *PE* to operate on one specific tree topology.

Finally, nucleotide vectors composed of nucleotides at one site of the sequence alignment under consideration are read and shifted through the array one by one so as to enable the processing elements to compute the scores for that specific alignment site for all tree topologies in parallel. When the first *PE* finishes its operation for one nucleotide vector, another vector is read and shifted through the array until there is no more nucleotide vector left in the *Input FIFO*. When every *PE* is done with the last nucleotide vector, the total tree scores computed by accumulating the score of each alignment site during the whole process in each *PE* are shifted backwards through the array into the *Output FIFO* to be read by the application software. The *FSM* coordinates all these operations of the *PE*s, *Input FIFO* and *Output FIFO* in

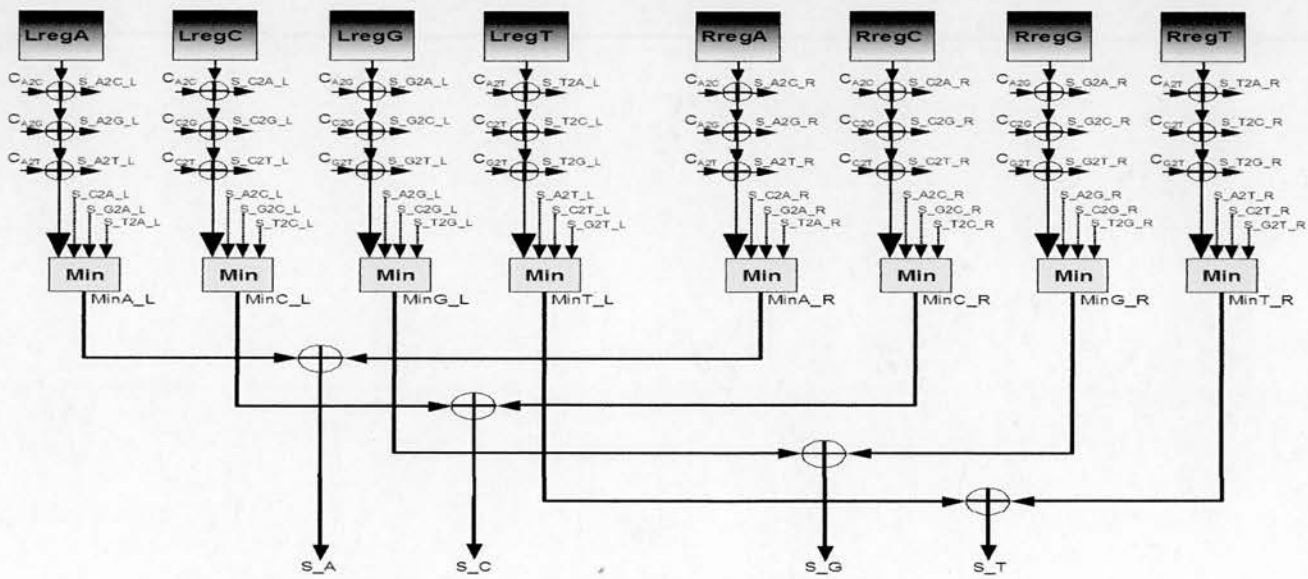


Fig. 7. Simplified hardware architecture for the conditional subtree length vector calculation of the nucleotide sequence

accordance with control data coming from the application software running on the host.

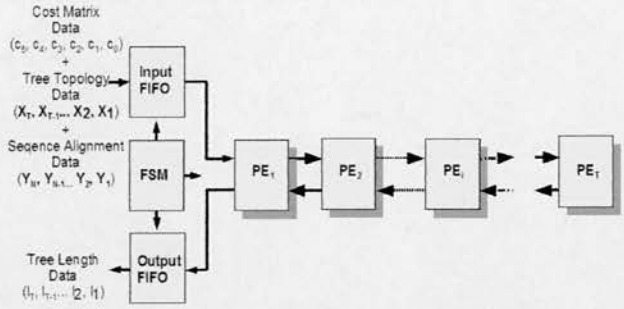


Fig. 8. Linear systolic array for the Sankoff's algorithm

As mentioned before, the number of *PE*s in the linear array is equal to the number of possible tree topologies. However, considering the amount of resources in today's FPGAs, this is not always feasible since there could be hundreds or even thousands of theoretically possible tree topologies for a given number of taxa. To solve this problem, the algorithm is partitioned into small steps which are mapped onto a fixed size linear systolic array [14] [15].

In this architecture, the tree evaluation process is performed in numerous iterations (or passes) for each set of tree topologies. Obviously, the number of iterations depends on the number of possible tree topologies. The additional *FIFO* in this architecture is used to store the sequence alignment data shifted in the first pass which will be read by the array in the next passes when the time comes for shifting all nucleotide vectors through the array. On the other hand, the *Input FIFO* is read to obtain a new set of tree topology vectors at each pass while there is no need to read cost matrix data after the first pass.

B. Architecture of a Processing Element

Figure 9 shows the simplified inner structure of a processing element which is mainly composed of *DpathL* and *DpathR* blocks. Data read from the *Input FIFO* is shifted through the array via linked *Data* registers in the *PE*s as illustrated in figure 8 to be used by *DpathL* and *DpathR* blocks which implement the Sankoff's algorithm on the left and right subtrees (see figure 6) of a tree topology, respectively. In the architecture, the score of the right branch computed by the *DpathR* is inputted to the *DpathL* for the calculation of the 4 elements of the root node vector which are then inputted to the *Min* block to find the minimum of them. The minimum value is the score of the tree at a specific site of the sequence alignment under consideration. This score is multiplied by the *Weight* register which holds the weight of that site within the alignment and then, the obtained result is added to the *TotScore* register which will hold the total score of the tree topology when the computations for the last site in the alignment is finished in the *PE*. The value of the *TotScore* registers which are linked to each other are shifted backwards into the *Output FIFO* as illustrated figure 8 when every *PE* in the array is done with the computation of the total score of its assigned tree topology.

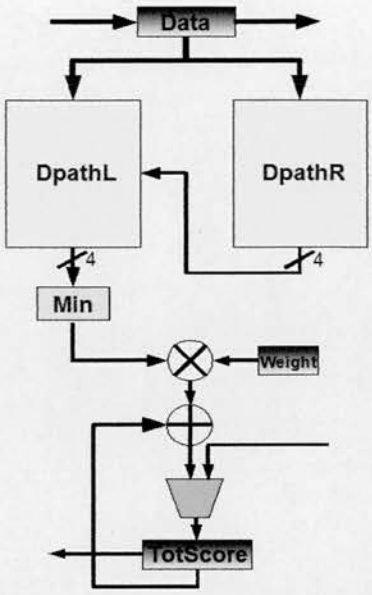


Fig. 9. Simplified inner structure of a processing element (annotated numbers represent number of words)

Figure 10 shows the simplified inner structure of the *DpathL* block which contains one *DpathUnitL* block and three *DpathUnitR* blocks. *DpathUnitL* and *DpathUnitR* blocks are responsible from conditional node vector calculation (see figure 7). Each *DpathUnitR* has 4 data inputs two of which are coming from outside the *DpathUnitL* and *DpathUnitR* blocks, one of which is coming from its *Min & Add Op.* block (whose inner structure is shown in figure 7) and last of which is coming from the *Min & Add Op.* block of the right hand side neighbour *DpathUnitR* block. On the other hand, *DpathUnitL* has 5 data inputs four of which are like those of *DpathUnitR* and the fifth one (input R) is coming from outside the *DpathL*. Also, the *DpathUnitL* and *DpathUnitR* blocks have control inputs C_x that determine which data inputs will be registered. For example, if C_c is asserted, input C will be stored in *Lreg_1* in the next cycle. With various combinations of these control signals, *DpathL* can compute conditional vectors of multiple nodes (up to 4 nodes) in various topological forms at the same cycle. Furthermore, *DpathUnitL* block of the *DpathL* is employed to compute the root node vector of the tree under consideration using its input R coming from *DpathR* (see figure 9).

Note that *DpathR* has a similar structure to that of *DpathL* but it has one less *DpathUnitR* block. So, it can compute conditional vectors of up to 3 nodes concurrently. *DpathL* will be explained more in detail next in this subsection.

DpathL block incorporates four arrays of registers (*CostReg*, *TreeStructReg*, *TaxaOrderReg* and *ResidueReg*) which are fed by the *Data* register shown in figure 9. *CostReg* stores the values of the cost matrix which are used within *Min & Add Op.* block of each *DpathUnitL* and *DpathUnitR* blocks while *TreeStructReg* contains control configurations for the specific tree topology. *TreeStructReg* is decoded to obtain appropriate control signals for all multiplexers in the datapath with the help of *TreeStructIndReg* incrementing by one at every cycle.

Furthermore, *ResidueReg* keeps the nucleotides of the specific site in the sequence alignment a set of which is applied to the data inputs of the *DpathUnitL* and *DpathUnitR* blocks (i.e. inputs A, B, E, F, J, K, N, and O) appropriately at every cycle. The applied set of nucleotides is determined by the *TaxaOrderReg* with the help of *TaxaOrderIndReg* which is incremented every cycle by some value depending on the current control configuration in *TreeStructReg*. Note that the values of *TreeStructReg* and *TaxaOrderReg* at a time constitute a tree topology vector whereas contents of *ResidueReg* are obviously nucleotide vectors (see subsection III-A).

With their architecture, *DpathL* and *DpathR* can process any subtree topology with up to 8 and 6 taxa, respectively. Finally, a processing element in the linear array can support a tree topology with at most 12 taxa.

IV. Implementation Results

The MP method was implemented on an Alpha Data ADM-XRC-4FX [16] PCI-X card with the array architecture detailed lastly in subsection III-A, where the number of the processing elements (PE) was 20. The card has a Xilinx Virtex-4 XC4VFX100 FPGA chip [17] mounted on it. Furthermore, our design was captured in Verilog hardware description language. ModelSim was employed to simulate the core with a number of testbenches, whereas Xilinx ISE9.2 was used to synthesize, place and route the design, and generate the corresponding FPGA bitstream. The clock frequency of the FPGA hardware was set to 70 MHz. Note that only one FPGA bitstream is used to configure the FPGAs regardless of the number of taxa under consideration.

A high level application process running on the host computer was also built whose main duty was to write the cost matrix data, tree topology data and sequence alignment data to the input FIFO and then read the scores of the tree topologies from the output FIFO of the FPGA with Direct Memory Access (DMA) transfers. On the other hand, a small C program was written to construct the tree topology data for various numbers of taxa. Table I below shows the timing performance figures of the FPGA implementation of the MP method for up to 12 nucleotide sequences. It was assumed that the cost of changes from a purine (A or G) to pyrimidine (C or T) is two times the cost of changes from a purine to a purine and pyrimidine to a pyrimidine. The length of the nucleotide sequences was 898 where a two times higher weight was applied to the changes occurring at the first position of the codons compared to the second and third positions.

TABLE I
Timing Performance Figures of the Hardware Implementation for the MP Method

No. of Taxa	No. of Trees	No. of Iterations	Min. Score	Average Time (s)
4	3	1	778	1.420
5	15	1	927	1.421
6	105	6	1,124	1.423
7	945	48	1,361	1.425
8	10,395	520	1,396	1.430
9	135,135	6,757	1,488	1.480
10	2,027,025	101,352	1,587	2.255
11	34,459,425	1,722,972	1,898	3.446
12	654,729,075	32,736,454	2,230	5.893

Each row in Table I is associated with some number of taxa given in the first column where the second column presents the number of unrooted tree topologies to be searched for that specific number of taxa. Furthermore, the third column gives the number of iterations required by the hardware core considering the number of available PEs to complete the processing of all trees (see subsection III-A) while the fourth column shows the score of the most parsimonious tree found during the exhaustive tree search. Finally, the fifth column gives the average time in seconds taken by the hardware core to complete its operation for each number of taxa.

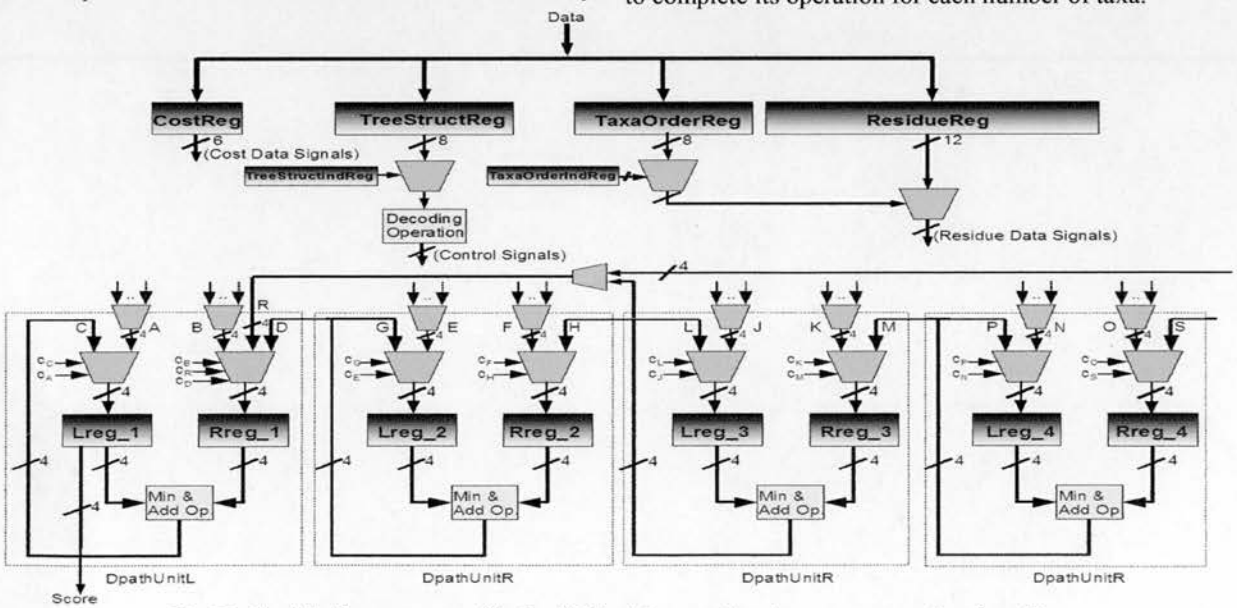


Fig. 10. Simplified inner structure of the DpathL block (annotated numbers represent number of words)

For comparative purposes, Table II below shows the timing figures of the PAUP software execution configured to operate in the same way as the hardware implementation, with the same nucleotide sequences. The software version was executed on a 2.2 GHZ Intel Centrino Duo machine with 2 GB of RAM running Windows XP operating system.

TABLE II
TIMING PERFORMANCE FIGURES OF THE PAUP SOFTWARE FOR THE MP METHOD

No. of Taxa	No. of Trees	Average Time (s)
4	3	0.02
5	15	0.02
6	105	0.02
7	945	0.16
8	10,395	0.67
9	135,135	7.84
10	2,027,025	241
11	34,459,425	5,852
12	654,729,075	127,325

Table III below provides the speed-up values of the hardware implementation over the software implementation (PAUP) for various numbers of taxa. It is obvious that the hardware core outperforms PAUP hugely when the number of taxa is over 8 with speed-up values reaching 21,606x for the 12-taxa case. For smaller numbers of taxa, the overhead of distributing matrix data, tree topology data, and sequence alignment data to the FPGA card from the host, and collecting results back to the host from FPGA, surpasses the gain from the parallel operation of the nodes on FPGAs, which is to be expected.

TABLE III
SOFTWARE (PAUP) VERSUS 1-NODE HARDWARE IMPLEMENTATION SPEED-UP VALUES

No. of Taxa	FPGA Speed-Up
9	5.3
10	110.05
11	1698.2
12	21606.1

V. CONCLUSIONS

In this paper, the detailed FPGA implementation of the Maximum Parsimony method for molecular phylogenetic analysis on a Xilinx Virtex-4 FPGA chip has been presented. This is the first FPGA implementation of this method for nucleotide sequence data ever reported in the literature to our knowledge. The hardware architecture is a linear systolic array composed of 20 processing elements each of which performing the Sankoff's algorithm for a different tree topology in parallel. This array computes the scores of all theoretically possible trees for a given number of taxa in several iterations. The currently supported maximum number of taxa is 12 but this number can be easily improved by cascading more **DpathUnit**s in **DpathL** and **DpathR** blocks. Furthermore, the resulting implementation outperforms an equivalent desktop-based software implementation (PAUP) by several orders of magnitude. The speed-up values achieved by the hardware implementation can reach up to 21,606x for the 12-taxa case. The reasons behind this very high speed-up are essentially twofold: the first is the coarse-grained parallelism among processing elements, since each PE processes a different tree topology in parallel with other PEs, and second is the fine-grained parallelism achieved in each processing element, as conditional vectors of several nodes on a specific level of the tree topology under consideration are computed concurrently (see figure 6).

The work presented in this paper is part of a bigger project which aims to harness the computational performance and re-configurability features of FPGAs in the field of bioinformatics and computational biology. Future work includes extending and improving the presented architecture to be able to support computations of the tree topologies for unlimited number of taxa by incorporating a reconfigurable router into the design.

REFERENCES

[1] Salemi, M. and Vandamme, A. M., *'The Phylogenetic Handbook: A Practical Approach to DNA and Protein Phylogeny'*, Cambridge University Press, 2003.

[2] Kidd, K. K. and Sgaramella-Zonta, L. A., *'Phylogenetic analysis: Concepts and methods'*, American Journal of Human Genetics, 23, pp. 235-252, 1971.

[3] Bakos, J. D. and Elenis, P. E., *'Special-Purpose Architecture for Solving the Breakpoint Median Problem'*, IEEE Trans. VLSI Systems, 16(12), pp. 1666-1676, 2008.

[4] Bakos, J. D., Elenis, P. E. and Tang, J., *'FPGA Acceleration of Phylogeny Reconstruction for Whole Genome Data'*, Proc. of the IEEE Bioinformatics and BioEngineering Conference, 2007.

[5] Bakos, J. D., *'FPGA Acceleration of Gene Rearrangement Analysis'*, Field-programmable Custom Computing Machines, 2007.

[6] Mak, T. S. T. and Lam, K. P., *'Embedded Computation of Maximum-Likelihood Phylogeny Inference Using Platform FPGA'*, Proc. of the IEEE Computational Systems Bioinformatics Conference, pp.512-514, 2004.

[7] Mak, T. S. T. and Lam, K. P., *'FPGA-based Computation for Maximum Likelihood Phylogenetic Tree Evaluation'*, Proc. of the Field-Programmable Logic and Applications Conference, 2004.

[8] Mak, T. S. T. and Lam, K. P., *'High Speed GAML-based Phylogenetic Tree Reconstruction Using HW/SW Codesign'*, Proc. of the IEEE Computational Systems Bioinformatics Conference, pp. 470-473, 2003.

[9] Davis, J. P., Akella, S., and Waddell, P. H., *'Accelerating phylogenetics computing on the desktop: Experiments with executing UPGMA in programmable logic'*, Proc. of the IEEE Eng. Med. Biol. Soc., pp. 2864-2868, 2004.

[10] Fitch, W. M., *'Toward defining the course of evolution: Minimum change for a specific tree topology'*, Systematic Zoology, 20, pp. 406-416, 1971.

[11] Sankoff, D. and Rousseau, P., *'Locating the vertices of a Steiner tree in an arbitrary metric space'*, Math. Progr., 9, pp. 240-276, 1975.

[12] Swofford, D. L., *'PAUP*: Phylogenetic analysis using parsimony (* and other methods)'*, Version 4.0b10, Sinauer Associates Inc., Sunderland, Massachusetts, USA, 2002.

[13] Download website for PAUP, *http://paup.csit.fsu.edu/downl.html.*

[14] Kung, S. Y., *'VLSI Array Processors'*, Prentice-Hall, 1988.

[15] Moldovan, D. I. and Fortes, J. A. B., *'Partitioning and Mapping of algorithms into fixed size systolic arrays'*, IEEE Transactions on Computers, 35(1), pp. 1-12, January, 1986.

[16] ADM-XRC-4FX datasheet, *http://www.alphadata.co.uk/adm-xrc-4fx.html*, Alpha Data ltd., May 2007.

[17] Virtex-4 datasheets, *http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm, Xilinx Inc.*, May 2007.