FPGA-based acceleration of the RMAP short read mapping tool

Pooja N. Mhapsekar
Iowa State University

FPGA-based acceleration of the RMAP short read mapping tool

by

Pooja N. Mhapsekar

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:

Phillip H. Jones, Major Professor

Joseph Zambreno

Srinivas Aluru

Iowa State University

Ames, Iowa

2013

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

First and foremost, the person I would like to thank the most is my major professor, Dr. Phillip Jones. This thesis would not have been possible without his guidance and help. I am grateful to him for motivating me and being instrumental in preventing otherwise disheartening moments for me during the course of this thesis. I cannot thank him enough for the endless patience he has shown in helping me throughout my masters program at Iowa State University. Since a major chunk of the work in this thesis was done off-campus, he took time out from his schedule, even during weekends, to discuss with me the directions for this work through long phone calls. It wouldn't be an overstatement if I said I consider it an honor to have been able to work under him.

Next, I would like to thank my committe members, Dr. Joseph Zambreno and Dr. Srinivas Aluru. Dr. Zambreno has provided me with useful guidance, both during coursework and later while I was working as a teaching assistant for one of his courses. Dr. Aluru and one of his students, Xiao Yang, helped me get acquainted with the field of Bioinformatics, on which this thesis is based. I would also like to thank my two good friends, Avinash Srinivasa and Moinuddin Sayed, for helping me in proof-reading and perfecting the content in this thesis.

I cannot thank my parents enough for having stood by me in every decision I have taken, for encouraging me to do my masters, and for often going out of their way to make sure I had everything I needed to be academically successful. Lastly, I would like to thank all my wonderful friends and my managers at Micron for helping me sail through and bring my thesis to its successful culmination.

ABSTRACT

Bioinformatics is a quickly emerging field. Next generation sequencing technologies are producing data up to several gigabytes per day, making bioinformatics applications increasingly computationally intensive. In order to achieve greater speeds for processing this data, various techniques have been developed. These techniques involve parallelizing algorithms and/or spreading data across many computing nodes composed of devices such as Microprocessors, Graphics Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs).

In this thesis, an FPGA is used to accelerate a bioinformatics application called RMAP, which is used for Short-Read Mapping. The most computationally intensive function in RMAP, the read mapping function, is implemented on the FPGA's reconfigurable hardware fabric. This is a first step in a larger effort to develop a more optimal hardware/software co-design for RMAP.

The Convey HC-1 Hybrid Computing System was used as the platform for development. The short-read mapping functionality of RMAP was implemented on one of the four Xilinx Virtex 5 FPGAs available in the HC-1 system. The RMAP 2.0 software was rewritten to separate the read mapping function to facilitate its porting over to hardware. The implemented design was evaluated by varying input parameters such as genome size and number of reads. In addition, the hardware design was analyzed to find potential bottlenecks. The implementation results showed a speedup of ~5x using datasets with varying number of reads and a fixed reference genome, and ~2x using datasets with varying genome size and a fixed number of reads, for the hardware-implemented short-read mapping function of RMAP.

CHAPTER 1. Introduction

This chapter gives an overview of the developments in the field of Bioinformatics and its applicability to society. It sheds light on the need of computation tools and platforms for processing the data generated by Next Generation Sequencing (NGS) technology. It then outlines the motivation for this work, followed by the contributions and organization of this thesis.

Bioinformatics for society. The fields of Bioinformatics and Computational Biology are growing at a rapid pace. Bioinformatics has been an interface between modern biology and informatics. It involves discovery, development, and implementation of computational algorithms and software tools that facilitate an understanding of biological processes with the goal of serving society in several ways. Bioinformatics aims to play a key role in important areas such as agriculture, bioenergy, and medicine. In agriculture, bioinformatics is being used to identify genes responsible for various plant traits to increase nutritional content, volume and disease resistance of agricultural produce [Jung and Main (2011); Mochida and Shinozaki (2010); Quijadaa et al. (2004a); Paterson et al. (2004); Quijadaa et al. (2004b); Vij et al. (2006); Close (2011)]. Use of Bioinformatics for improving bioenergy producing species for traits such as high biomass yield, environmental stress tolerance and high nutrient is contributing towards increased bioenergy producing species [Shen et al. (2009)]. In the pharmaceutical sector, it can be used in the drug discovery process to custom design drugs and to develop personalized medicine [Hanash (2003); Imming et al. (2003)] and for early diagnosis of diseases such as cancer [Bravo et al. (2012)]. [Ley et al. (2008)] shows how sequencing the DNA of normal skin cells and the DNA of tumours can help identify cancer-initiating mutations that alter the gene sequence of healthy cells.

Need for large scale data processing. Next generation sequencing technologies are capable of generating billions of bytes of data in a single sequencing run, which can take from hours to days depending on the sequencing technology used [Schuster (2008); Mardis (2008)]. This data is often of the form of DNA and RNA base sequences, where each base is typically stored using 2 bits to a byte of memory. For example, the Illumina/Solexa Hiseq system can produce up to 3 billion reads within 1.5 to 11 days [NGS (2012)] and the 454 Life Sciences sequencer can generate a million 100 base sequences a day [Margulies et al. (2005)]. In order to facilitate analysis of this data, computational tools have been developed. However, these tools often cannot keep up with the increasing rate of data generation. New computing approaches are being continuously developed to enable processing and in-depth analysis of such large datasets in a timely manner. These approaches can be categorized as follows: 1) algorithmic [Langmead et al. (2009b); Li et al. (2008b)], 2) porting existing algorithms to emerging computing platforms (such as graphics processors) [Ashwin et al. (2010); Schatz (2009)], 3) designing custom computing platforms [Olson et al. (2012)], and 4) a combination of the above techniques [CeBiTec (2012)]. A common theme among these approaches are the extraction and exploitation of computational parallelism, and/or leveraging statistical properties associated with the dataset.

Motivation. With increasing amounts of data being generated by new generation sequencers, applications can take several hours while executing on a normal PC. One such application is the popular Bioinformatics application for Short Read mapping called RMAP. RMAP is used to accurately map reads generated from the afore mentioned next generation sequencing machines [Smith et al. (2009)]. This work attempts to reduce the turnaround time and improve performance of RMAP, in particular its read mapping function. The read mapping function was chosen for acceleration since it is the most computationally intensive function of RMAP. A custom computing architecture was developed to implement this function in hardware. The design was implemented on a Convey HC-1 computer, which is a hybrid computing platform that tightly couples a standard x86 platform with an FPGA subsystem [Convey (2012)].

Contributions. The primary contributions of this work are:

1) a detailed profiling of the latest version of RMAP (RMAP 2.0) software to determine its most computationally intensive functions,

2) a software-hardware based solution accelerating the basic RMAP 2.0 software functionality

on a hybrid computing platform (HC-1), and

3) a performance analysis of the implemented solution.

Organization. The remainder of this work is organized as follows. Chapter 2 gives a background on some major areas in the field of Bioinformatics and outlines the Short Read Mapping process. It also provides an overview of the computing technology used in this work. Chapter 3 discusses related work in the areas of accelerating bioinformatics algorithms using software techniques and custom computing hardware platforms. Chapter 4 gives an overview of the RMAP algorithm. In Chapter 5, the organization and architecture of the Convey HC-1 system, the platform used for RMAP implementation, is described. Chapter 6 provides an overview of the software-hardware co-design implemented in this work, and Chapter 7 gives architecture-level details of the hardware design. The methodology used for evaluating the hardware architecture is discussed in Chapter 8. Chapter 9 presents analysis of the hardware design and the data obtained from performance experiments. Chapter 10 concludes this thesis and suggests areas of future research.

CHAPTER 2. Background

This section first provides a high-level overview of the field of bioinformatics. The specific area of “Short Read Mapping” is then discussed in greater detail, since the application accelerated in this work is for short read mapping. The section concludes with a brief introduction to the computing technology used for acceleration, called Field Programmable Gate Arrays (FPGAs).

2.1 Bioinformatics

Bioinformatics is the science of storing and maintaining databases of biological information, and developing new and revised techniques to access, process and analyze this information. These techniques combine the power of computers, mathematical algorithms, and statistics to uncover useful information hidden in these databases and obtain a clearer insight into the fundamental biology of organisms. This extracted information has profound impacts on fields as varied as human health, agriculture, the environment, energy and biotechnology.

Terms and definitions. The biological terms used in this thesis are explained below.

- *Genome.* The total genetic material of a given organism.
- *Deoxyribonucleic acid (DNA).* Holds the genetic code for an organism in the arrangement of four bases: Adenine (A), Thymine (T), Guanine (G) and Cytosine (C). These bases are called nucleotides.
- *Chromosome.* Collection of DNA, and proteins that organize an organism’s genome.

- *Reads.* Short sequences made up of the four bases A, T, G, C, extracted from a DNA sample. These sequences can be as large as 400 bases.
- *Coverage.* Number of reads of length x covering a chromosome site.
- *Error rate.* The amount of error induced to account for mutations (i.e. a base randomly changing value) while generating reads.
- *Primary structure.* Linear arrangement of atoms in a molecule and the chemical bonds connecting them.
- *Secondary structure.* Areas of folding or coiling within a molecule.
- *Tertiary structure.* Three-dimensional structure, as defined by the atomic coordinates.

Algorithms based on mathematical and computer science principles have been developed for the bioinformatics areas of sequence alignment, structure prediction, sequence assembly, phylogenetics, system biology, gene prediction, motif finding and comparative genomics [Sequence and Genome Analysis (2004)]. These algorithms have complexities ranging from polynomial time to exponential time. Brief definitions of a few of these bioinformatics areas and the algorithms used for them are given in the following paragraphs.

Sequence Alignment. Is used to determine the similarity and function relatedness between two or more sequences. For example, if a new sequence is obtained from genome sequencing, then the first step is to look for similarities to known sequences found in other organisms. If the function/structure of similar sequences/proteins is known, then it is highly likely that the new sequence corresponds to a sequence/protein with the same function/structure [Sequence and Genome Analysis (2004)]. An optimal alignment is where arrangement of the two sequences is in a way that the number of mutations is minimal [Bioinformatics I (2008)]. Sequence alignments can be global or local. A global alignment is an optimal alignment that includes all characters from each sequence, whereas a local alignment is an optimal alignment that includes only the most similar local region or regions [Krawetz and Womble (2003)]. Dynamic Programming [Eddy (2004)], Hidden Markov Models [Eddy (1995)] and Longest Common Subsequence [Sahoo and Padhy (2009)] are techniques that are often used for developing efficient sequence alignment algorithms [Jones and Pevzner (2004)]. Two widely used alignment algorithms that are based on dynamic programming are the Smith Waterman algorithm [Boukerche et al. (2007)] for local alignment and the Needleman-Wunsch algorithm [Needleman and Wunsch (1970)] for global alignment.

Structural Alignment. Is used to establish similarity between two or more molecule structures by comparing them based on their shape. It is a useful tool when the sequences to be compared have low sequence similarities due to major evolutionary changes [Bourne and Shindyalov (2003)]. This technique is mainly used for protein and to some extent for RNA molecules. DALI [Holm and Sander (1993)] and Combinatorial Extension (CE) [Bourne and Shindyalov (1998)] are two standard protein structural comparison methods. SETTER (SEcondary sTructure-based TERtiary Structure Similarity) [Hoksza and Svozil (2012)] is a program used for RNA structural comparison.

Structure Prediction. Is a technique used to determine the secondary and tertiary structure of protein (amino acids) or nucleic acid sequences (DNA, RNA) on the basis of their primary structure [Sequence and Genome Analysis (2004)]. Structure prediction can provide meaningful insights into the nature of protein or nucleic acid structures and their functional mechanisms. SFOLD [Ding and Lawrence (2003)] is a statistical tool for RNA secondary structure prediction. Chou-Fousman [Chou and Fasman (1974)] and GOR [Garnier et al. (1996)] are two statistical based methods used for protein structure prediction.

Sequence Assembly. Is the process of placing fragments of DNA that have been sequenced into their correct position within a chromosome. SHARCGS [Dohm et al. (2007)] and SSAKE [Warren et al. (2007)] are examples of Greedy graph based assemblers. Newbler [Margulies et al. (2005)] and Celera [Myers et al. (2000)] are examples of Overlap-Layout-Consensus based assemblers. Abyss [Simpson et al. (2009)] and Velvet [Zerbino and Birney (2008)] are examples of de Bruijn Graph based assemblers.

Phylogenetics. Is the study of relatedness among organisms through their morphological (form and structure of organisms) or molecular characteristics. Phylogenetic analysis helps biologists in making predictions about fossils, learning about evolution of complex features, and making predictions about poorly-studied species [W. E. Stein (1987); Labeda et al. (2012)].

Methods for phylogenetic analysis include statistical methods such as Maximum Parsimony [Swofford and Begle (1993)], Bayesian Inference [Yang and Rannala (1997)] and Maximum Likelihood [Yang (2007)] and Genetic Algorithm-based methods [Helaers and Milinkovitch (2010)].

2.2 Short Read Mapping

Short Reads. Reads are generated through sequencing, which is the process of determining the order of nucleotides in a DNA fragment. A sample is fed into the sequencing machines and the sequences generated are fragments read from a longer DNA molecule present in the sample [NGS (2012)]. These fragment sequences are called reads. Since they are typically 25-400 bases in length, they are termed as short reads. Before the advent of New Generation Sequencing technology, these reads were produced using capillary electrophoresis (CE)-based Sanger sequencing [Sanger and Coulson (1975)] method, which was slow and expensive [Schuster (2008)]. Now, with new generations of DNA sequencers, billions of reads can be generated rapidly and inexpensively [Schuster (2008)].

Short Read Mapping. Is when a collection of short reads are individually sequence aligned (i.e. mapped) to a known genome called a reference genome [Trapnell and Salzberg (2009)]. The mapping of reads onto the reference genome can be based on different criteria. For example, exact mapping will only align a read to the reference genome if an alignment is found that has no mismatches. If, however, mutations are to be considered, then an approximate alignment can be performed that allows some threshold of mismatches. Another reason for allowing for approximate mapping is to account for inserted or deleted (indel) bases by allowing gaps in the alignment of a read to the reference genome.

Challenges of mapping short reads. The two main challenges posed by short read mapping are as follows.

1) *Resources and speed.* When mapping involves a large reference genome, for example the human genome which contains 3 billion nucleotides and an equally large number of short reads, the mapping problem can become computationally intensive. This results in increased usage of CPU and memory resources leading to long execution times on standard processors [Trapnell and Salzberg (2009)].

2) *Accuracy.* A genome can have repeats, which are multiple copies of the same base sequences across the genome. If a read was extracted from a repeat location, then it becomes difficult for the mapping software to decide which of the locations the read belongs to. In addition, mutations and sequencing errors can also affect the accuracy of the mapping [Trapnell and Salzberg (2009)].

Algorithms targeting these issues are being actively worked upon.

Short read mapping algorithms. To overcome the challenges posed in mapping short reads, short read mapping algorithms are designed to have a low memory footprint, good mapping speed, high accuracy and sensitivity. The most commonly used algorithmic method for short read mapping are indexing-based solutions [Pevzner and Waterman (1995)], which attempt to find subsequences of each read that match perfectly to a location in the reference genome and evaluate only these reads with the genome [Smith et al. (2009); Lin et al. (2008)]. RMAP, an indexing-based solution, maps 8 million reads per hour to the human genome, allowing two mismatches, at full sensitivity [Smith et al. (2009)]. Another common method is the Burrows-Wheeler transform [Burrows and Wheeler (1994); Langmead et al. (2009b); Li and Durbin (2009)]. The Burrows-Wheeler transform starts with a large list of possible locations to which the read could align and iteratively reduces the list to a small set of locations. Bowtie, based on Burrows-Wheeler transform, aligns short DNA sequences (reads) to the human genome at a rate of over 25 million 35-bp reads per hour on a normal workstation [Bowtie (2012)]. However, these tools are run sequentially on general purpose processors, which creates a bottleneck for high performance. Hence, computing platforms for parallel execution of mapping algorithms are needed.

2.3 Field Programmable Gate Array (FPGA)

Traditional general purpose processors tend to be inherently sequential. Thus they do not efficiently support extracting the parallelism in an application. Field Programmable Gate Arrays (FPGAs) on the other hand have fine-grained parallelism, which allows them to explicitly realize parallelism that traditional processors cannot. An FPGA is an integrated circuit designed to be configured by a customer or a designer after manufacturing. Hence the name

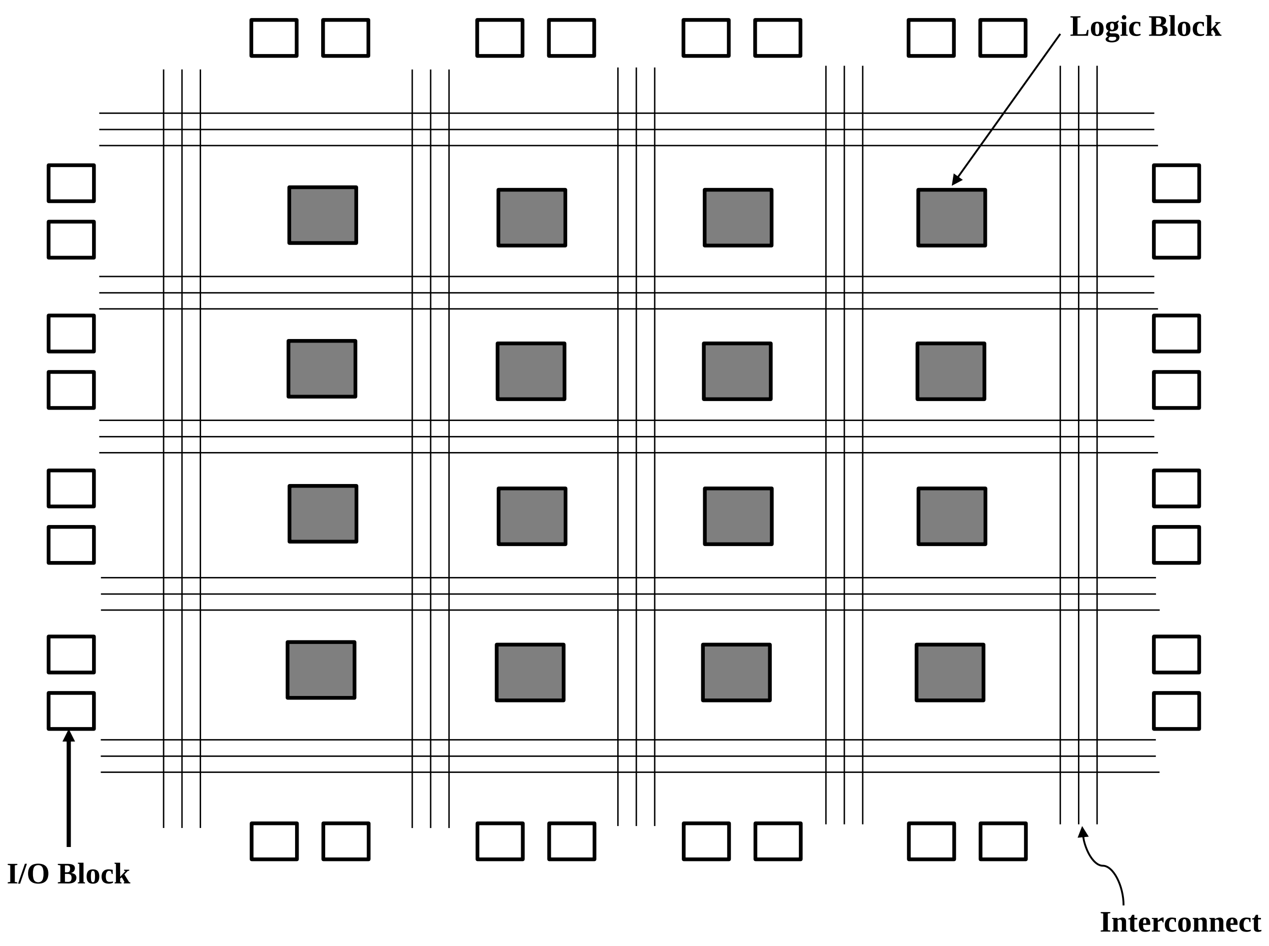


Figure 2.1: FPGA fabric with CLBs, interconnect and I/O blocks.

"field-programmable". An FPGA is a sea of reconfigurable logic and programmable routing for realizing digital circuits.

Figure 2.1 shows the FPGA fabric with configurable logic blocks, interconnects, and I/O pins. The basic components of an FPGA are [Xilinx (2012)]:

1) *Configurable Logic Blocks (CLBs).* These are the programmable logic blocks in an FPGA. Every CLB has a configurable look up table (LUT), which can be configured to implement combinational logic functions. They also contain memory elements, which when coupled with the LUTs can be used to implement registers for sequential logic.

2) *I/O Blocks.* The Input/Output blocks make it possible to connect the FPGA resources to elements outside the FPGA. These are divided into banks and each bank supports a particular I/O standard.

3) *Interconnect/Routing.* These are the wiring resources of the FPGA that connect the logical blocks, I/O blocks and other resources. Within each CLB, the logic elements are connected using interconnects. The interconnects also connect CLBs to each other and to I/O blocks to implement larger functions.

FPGAs also contain additional features, such as digital signal processing (DSP) blocks and large memories (i.e. Block RAMs) with low latency access.

CHAPTER 3. Related Work

This chapter is divided into three parts. The first part discusses accelerating bioinformatics applications, followed by the use of FPGAs for accelerating such applications. The second part focuses on Short Read Mapping acceleration techniques. The last part goes on to explain the related work done in accelerating RMAP.

3.1 Accelerating Bioinformatics Applications

Bioinformatics tools involve intensive computing leading to long application run-times. Techniques have been proposed in [Smith et al. (2009); Gálvez et al. (2010); Schatz et al. (2007); Mount (2004); Zhang et al. (2007)] to address this issue and accelerate applications. [Smith et al. (2009)] uses cache optimization techniques to reduce the memory footprint of the software and spaced-seed filtration of reads to reduce the amount of data involved in read mapping. In [Gálvez et al. (2010)], a sequence alignment algorithm called FASTLSA [Driga et al. (2003)] is implemented on the Tile64 processor by splitting computation across the 64 cores of the processor. A speedup of up to 20x is achieved over the non-parallelized version. MUMmerGPU [Schatz et al. (2007)] is an application from NVIDIA's Tesla Bio Workbench. In MUMmerGPU, the Compute Unified Device Architecture (CUDA) programming language was used to develop a Graphics Processing Unit (GPU) implementation for multiple sequence alignment of sequences against a reference sequence stored as a suffix tree. The multiple sequences are processed in parallel using the highly parallel architecture of the GPU to achieve a speedup of 3.5x in total application time over MUMmerGPU running on a high-end CPU.

Due to their inherent parallelism and reconfigurability, the use of FPGAs for accelerating bioinformatics applications is increasing. In [Chen et al. (2009)], FPGA hardware is used to accelerate the filtration stage of BLASTN [BLAST (2012)], a widely used sequence alignment tool for DNA sequences. An efficient bloom filter, a space-efficient hashing data structure, based architecture is implemented on the FPGA to replace the software-based filtration stage of BLASTN. In [Kasap et al. (2008a)], the authors present the first FPGA-based core implementation for accelerating the Gapped BLAST sequence alignment tool. Their results show substantial speedup compared to software only implementation, ranging from 20x to 44x. The XtremeData XD1000 [XtremeData (2006)], a hybrid FPGA computing platform, has been used in [Zhang et al. (2007)] to accelerate the Smith-Waterman algorithm [Boukerche et al. (2007)], for both DNA and protein sequences. The Altera FPGA coprocessor of the platform implements a multistage processing element (PE) design having 384 PEs in a systolic array [Kung and Leiserson (1978)]. The design attains a speed-ups 185x and 250x for DNA and protein sequences respectively, as compared to executing on the 2.2GHz AMD64 Opteron host processor of the XD1000 platform.

3.2 Short Read Mapping Acceleration

Next generation sequencers are generating reads at a rate that is overwhelming current short-read mapping tools. Techniques have been developed to accelerate these tools, employing both algorithmic as well as novel computing platform.

Algorithmic techniques. Given the large size of read sets, comparing each read with each chromosome position for mapping is not feasible. Hence, short read mapping algorithms filter poor matching reads before performing a full comparison. They use filtering strategies such as Burrow-Wheelers transform [Burrows and Wheeler (1994)], masking with seeds for hash table indexing (spaced seed approach) [Pevzner and Waterman (1995)], suffix array [Abouelhoda et al. (2004)] and non-deterministic automata matching [Holub and Melichar (1999)]. For example, RMAP [Smith et al. (2009)], ZOOM [Lin et al. (2008)], SHRiMP [Rumble et al. (2009)], MAQ [Li et al. (2008a)] and RazerS [D et al. (2009)] use a spaced seed approach for filtering. Tools like Bowtie [Langmead et al. (2009b)] and BWA [Li and Durbin (2009)] use the Burrow-Wheelers transform method for filtering, while Segemehl [Hoffmann et al. (2009)] uses a suffix array technique. PatMaN (Pattern Matching in Nucleotide databases), a fast, short read alignment tool implements non-deterministic automata matching [K et al. (2008)].

Computing platforms. [Liu et al. (2012); Langmead et al. (2009a); Kasap et al. (2008b); Olson et al. (2012)] present the use of new computing platforms for the acceleration of short read mapping applications. [Liu et al. (2012)] implement SOAP3 using multiple cores in a GPU to achieve speedups of at least 7.5x and 20x, as compared to BWA and Bowtie respectively, while aligning millions of short reads to a human genome. Crossbow [Langmead et al. (2009a)] is a hadoop [Hadoop (2012)] based short read aligner that uses a cloud computing based approach for short read mapping. It combines the speed of Bowtie and mapping accuracy of SOAPsnp [Li et al. (2009)]. It aligns reads with 38-fold coverage of the human genome under 3 hours on a 320-core cluster from Amazon's Elastic Compute Cloud (EC2) [Amazon EC2 (2012)] service. In [Kasap et al. (2008b)], an FPGA-based custom design implements an exact mapping algorithm using a brute force approach to compare short sequences in parallel to a reference genome from genome databases. For performance analysis, 100,000 reads of length 50bp taken from a chromosome of the human genome with 222,389,117 base pairs is used. Results indicate high sensitivity to genetic variations in short reads, as compared to Bowtie and Maq with computation speed similar to that of Bowtie. Another FPGA-based approach is used in [Olson et al. (2012)] to accelerate the algorithm used by the BFAST mapping software [Li et al. (2008a)]. A 250x speed up versus the original BFAST software and a 31x speedup against Bowtie is reported. Shepard is an exact matching short read aligner tool implemented on the Convey HC-1 hybrid-core reconfigurable computing system. For exact matching, Shepard is hundreds of thousands of times faster than SOAP2 or Bowtie, and about 60 times faster than the GPU-implemented SOAP3 [Nelson et al. (2012)].

3.3 RMAP Acceleration

[Ashwin et al. (2010)] and [Schatz (2009)] present a GPU-based and cloud-based implementation, respectively, for improving the performance of RMAP. The RMAP algorithm is presented in Chapter 4. GPU-RMAP [Ashwin et al. (2010)] accelerates the latest version of RMAP [Smith et al. (2009)] by implementing its genome-mapping function on a GPU. For the mapping function, the input reference chromosome is divided across multiple GPU threads.

The short reads are passed to each GPU thread, and all threads access the same binary tree made from the reads for performing binary search. The results of the mapping function are then passed to the CPU for further processing. With respect to the CPU implementation, GPU-RMAP shows a speed-up of 14.5x for the mapping function and 9.6x for the overall application. CloudBurst [Schatz (2009)], a parallel short read mapping algorithm, is modeled after an earlier version of RMAP [Smith et al. (2008)]. It uses the open-source Hadoop implementation of MapReduce to execute the design on multiple compute nodes of a cluster. CloudBurst running on a 24-core configuration is up to 30 times faster than RMAP running on a single core. When run on a 96-core Amazon EC2 cloud cluster, a speed-up of 105x was observed.

In this work, the short-read mapping function of the latest version of RMAP is implemented on Convey's HC-1 hybrid-core system. It is the first FPGA-based solution for accelerating a part of the RMAP software.

CHAPTER 4. RMAP

In this work, the latest version of the RMAP short read mapping application [Smith et al. (2008)] is accelerated. This chapter gives an overview of RMAP's features and describes its spaced-seed filtration technique in detail. The RMAP algorithm for carrying out short read mapping is then outlined along with a description of each phase of the computation. This is followed by detailed profiling and performance analysis of the application.

4.1 Overview

RMAP is a short read mapping tool introduced by Smith et al. (2008) to accurately map short reads generated from next generation sequencers. RMAP was originally designed to map reads from Illumina sequencers and can map reads of length ranging from 25-50 bases. Two important features of RMAP that improve its mapping accuracy are the consideration of information in 3' ends of longer reads and the use of quality scores for improving mapping accuracy. In addition, RMAP supports weight-matrix matching and wild-card matching modes using quality scores for improving accuracy. The idea of a quality score is to assign a probability for each base at a given position in a read, and based on this assign a score for the read position itself [Ewing et al. (1998)]. This score is used to determine whether the read position should be considered for mapping, based on a cutoff value which is adjustable by the user.

The inputs to RMAP are a set of reference genomes and a set of short reads. The output is composed of three pieces of mapping information for each read that maps on to the reference genome: 1) the best mapping location (site) on the chromosome, 2) the corresponding map score, and 3) strand of the reference genome (i.e. the chromosome considered in its original form (forward strand) or reverse-complement form). The genome file is in FASTA format, while the reads file can be in FASTA, FASTAQ, or PRB format. FASTAQ and PRB include quality score information for each base in the read set.

RMAP features. The latest version of RMAP supports mapping three types of reads: normal reads, paired-end reads and bi-sulphite treated reads. The same mapping algorithm is used for all three read types, which is outlined in Section 4.2. RMAP supports three mapping modes: 1) mapping with mismatches, 2) mapping using wildcards, and 3) mapping using weight-matrix matching. The user can set mapping parameters such as:

1) read width
2) number of seeds
3) number of allowable mismatches
4) number of best maps allowed
5) redirection of ambiguous reads to a file
6) use of quality scores
7) use of wildcard matching mode

This latest version of RMAP uses spaced-seed filtration technique, giving it improved accuracy over its predecessor. It is also structured to have better cache performance.

Spaced-seed filtration. The purpose of spaced-seed is to reduce the number of reads compared to any given location in the reference genome. A spaced seed is a pattern of 0's and 1's that is masked with the set of reads input to RMAP. The reads whose bases match in the 1's location are grouped together. When a section of the reference genome is processed, only the group of reads that match the section masked with the seed are considered for a full (base by base) comparison against the genome. The spaced seed technique helps in improving the specificity (i.e. the fraction of the sequences predicted as matches that really are true matches) of the algorithm by eliminating non-matching reads with the help of the spaced structure.

4.2 Algorithm

While the primary function of RMAP is the mapping of reads onto a reference genome, pre-processing of inputs and post-processing of results are also important aspects of the algorithm. The RMAP algorithm can be divided into the following steps, as illustrated in Figure 4.1.

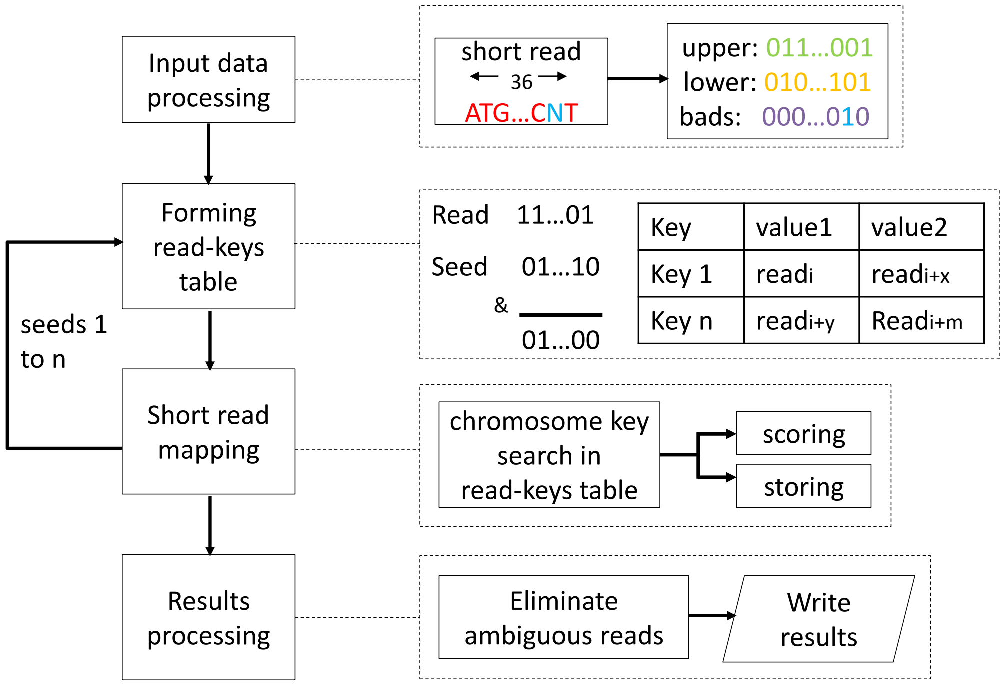


Figure 4.1: RMAP Algorithm.

1) **Input data processing**. Reading chromosomes and reads from input files. Decomposing the reads into “upper”, “lower” and “bads” structures.
2) **Forming a read-keys table**. Masking the reads with a seed; the masked values form keys of the read-keys table.
3) **Short read mapping**. Processing the reference genome chromosomes to pass as keys to be searched in the read-keys table. Splitting them into “upper”, “lower” and “bads” data for scoring against reads. Performing a search for each chromosome key, scoring the chromosome with the reads at the matched location and storing them.
4) **Results processing**. Eliminating ambiguous reads from the stored mapping results and printing results to a file.

Step 2 and 3 are executed for each seed, and are performed to map against the forward and reverse complement of the reference genome. Thus steps 2 and 3 are performed twice for each seed. Step 4 is performed once after processing the first seed. This is to filter out the “bad reads”, reducing the number of reads processed by the remaining seeds. It is also run after all the seeds have been processed to eliminate ambiguous reads from the mapping results of all seeds. These steps are explained in greater detail in the following section.

4.3 RMAP Modules

The following describes the main phases of RMAP, and reviews the data structures used and the various computations involved in each phase.

1) **Input data processing.** Chromosomes are read from the input file in the form of strings and are stored in a vector; this data represents the forward strand of the chromosome. The vector corresponding to the reverse-complement chromosome strand is created by taking the complement of each base, starting from the last base in the forward chromosome strand. Reads, each 36 bases wide, are read from the input file and encoded in a two bit structure, with A, C, G and T bases corresponding to 00, 01, 10 and 11 respectively. Each encoded base gets divided into "upper" - upper bit of the two bit encoded structure, "lower" - lower bit of the two bit encoded structure, and "bads" - which is '1' when a base is unusable for mapping. Thus each read is composed of a data structure having "upper", "lower" and "bads" data. Each read has a name associated with it which is read from the input file and stored in another vector. This indicates the position of the read in the chromosome from where it was extracted. For each read, a maximum of two sets with the three pieces of mapping information can be stored. Both the sets are stored in a vector of size equal to the number of reads. Thus, each read is associated with "upper", "lower", "bads", "score of the first best map", "score of the second best map (equal to first)", "mapping location on the chromosome" and "strand of the chromosome".

2) **Read-keys table formation.** The read-keys table is used for exact matching of read-keys and chromosome keys to filter out the non-matching reads. The seeds used in RMAP are 64-bit wide and a read-keys table is formed for each seed. Each read is 36-base wide (72 bits after encoding), hence it is truncated to 32 bases to form a 64 bit read-word. Each 64-bit read-word is then masked (i.e. bitwise 'AND'ed) with the seed forming a read-key. Each read-key is associated with a value which is the index of the read from which it was formed. The read-keys and the associated read indices are stored as pairs in a vector called "seed-keys table". Many of the reads when masked with the seed may result in the same read-key.

The vector container is sorted to group duplicate read-keys using an internal sort function provided by the STL vector library. The read indices associated with these duplicate read-keys are grouped together, resulting in a single read-key being associated with a set of read indices. An unordered map data structure (read-keys table) is then formed with the read-keys acting as keys and the starting and last index of a duplicate read set acting as a pair of values.

3) **Short read mapping.** This module identifies reads mapping to a particular chromosome site and scores the mapping. The chromosome bases stored in the chromosome string vector are converted to the two-bit encoded structure, which is then processed to form chromosome "upper", "lower", "bads" and "chromosome word" in a manner similar to the reads. The chromosome word is masked with the seed in use to form a chromosome key and is searched in the read-keys table for possible matches with the read-keys. For a matched chromosome key, the pair of read indices associated with it is fetched. The reads corresponding to these indices are then retrieved, and scored by comparing their "upper", "lower" and "bads" data with that of the chromosome section (associated with the matched chromosome key). A score is considered valid if it is less than a user defined threshold, which specifies the maximum mismatches allowed while mapping. If a score is found to be valid, the score, location of mapping on the chromosome, and the strand of the chromosome are stored for that read in the best mapping information vector described earlier. The second set of mapping information for a read is updated if the first set of information is already recorded and the current mapping score for that read is equal to the score in the first set.

4) **Results processing.** This is the last phase of RMAP, wherein the best mapping information vector is traversed to look for ambiguous reads. If a read, with both of its best mapping information recorded, maps to two different chromosome sites with the same score, then it is considered ambiguous and is eliminated. After eliminating the ambiguous reads, the best maps information for the non-ambiguous reads are written to an output file.

Table 4.1: RMAP parameters.

Parameter	Value
mapping mode	mapping based on mismatches
chromosome strand	forward
read length	36
maximum mismatches	3
seeds	1
best mapping informationfor each read	2

4.4 RMAP Profiling

Using the RMAP parameters specified in Table 4.1, several performance profiling experiments were conducted. These experiments helped determine the most computationally intensive function across various input datasets. *Gprof (a Linux GNU profiling tool)* was used to profile RMAP. Upon profiling, the *map_reads* function was found to be most computationally intensive. The processing time of this function was found to increase with an increase in problem size. For mapping 260 Million reads on to the complete human genome, which is the largest problem size shown here, it was found to take 90% of the total time. All profiling experiments were performed on the HC-1 host processor, described in Chapter 5.

Profiling and analysis. RMAP was profiled using three input datasets and the execution times of the five most computationally intensive functions were recorded. Figures 4.2, 4.3 and 4.4 show the results of these experiments. These results are also presented in the table shown in Figure 4.6. For each experiment, a read coverage of 40 was used. As can be seen, as the size of chromosome and number of reads increases (i.e. input size increases), the portion of RMAP's execution time spent in the *map_reads* function increases.

Another experiment was performed using the human genome, which has 3 billion bases, with a read coverage of 3 (Figure 4.5). The results indicate that 90% of the total execution time for this dataset is spent in *map_reads*. For large datasets, *map_reads* is clearly the most computationally intensive function in RMAP, and thereby an ideal candidate for acceleration. The human genome generated by the 1000genomes project [1000 Genomes (2010)] was used for this profiling and it can be downloaded from http://www.1000genomes.org/data.

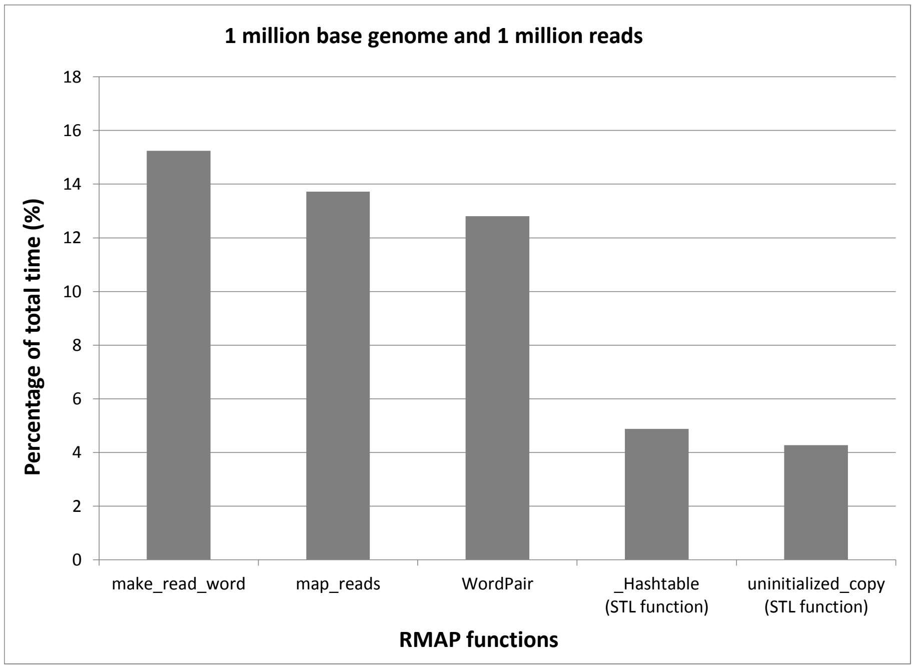


Figure 4.2: Five most computation intensive functions of RMAP for 1 million base genome with 1 million reads.

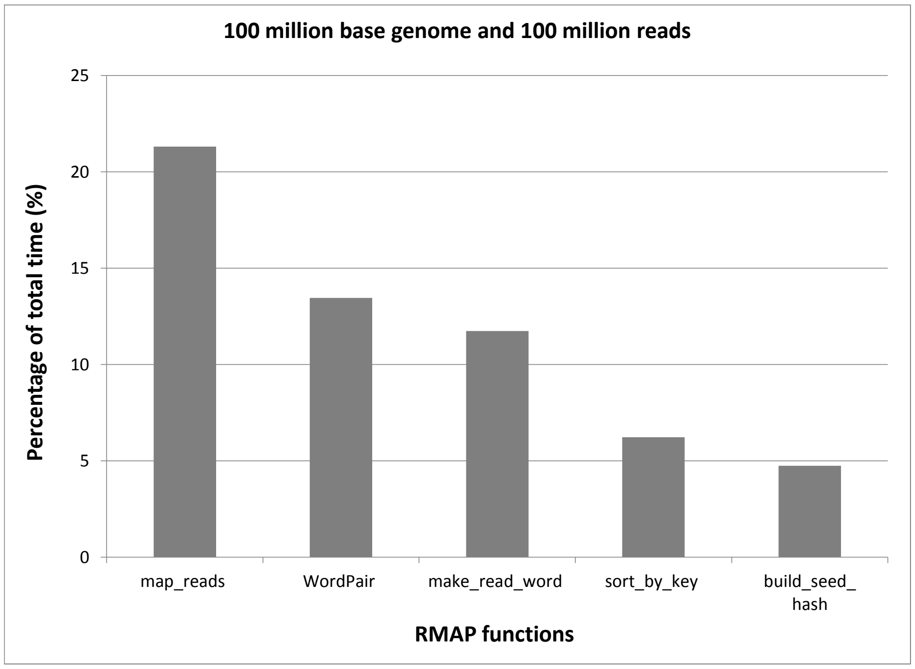


Figure 4.3: Five most computation intensive functions of RMAP for 100 million base genome with 100 million reads.

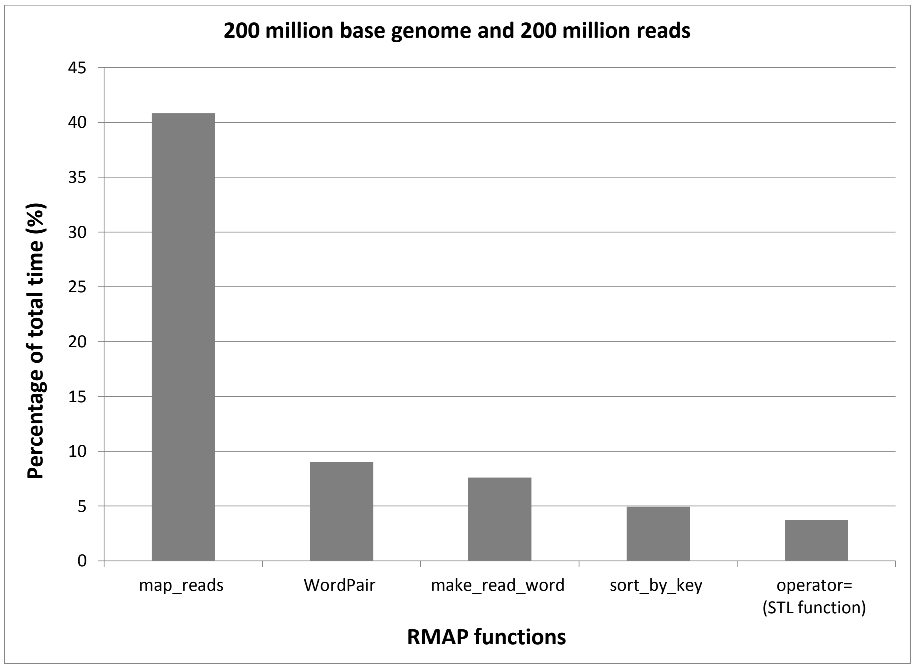


Figure 4.4: Five most computation intensive functions of RMAP for 200 million base genome with 200 million reads.

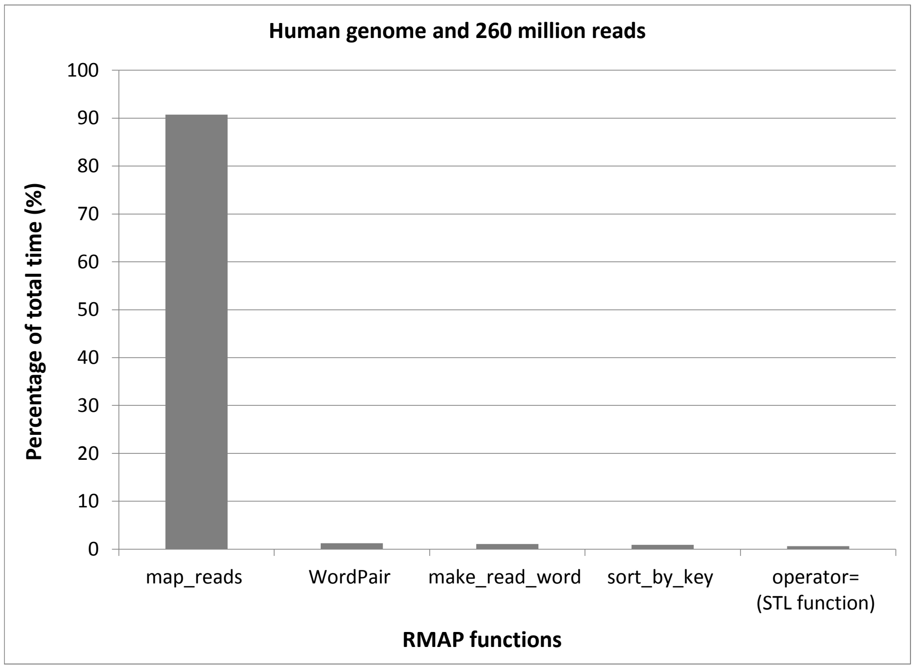
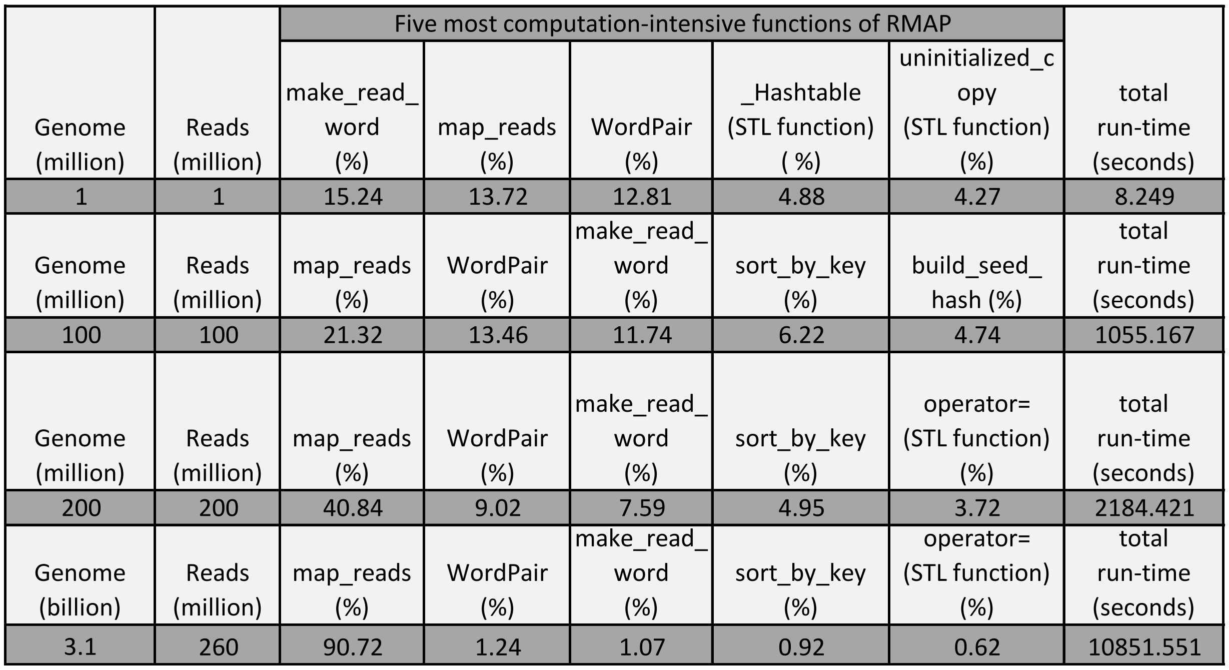


Figure 4.5: Five most computation intensive functions of RMAP for 3 billion base genome (human genome) with 260 million reads (coverage 3).



		Five most computation-intensive functions of RMAP					
Genome (million)	Reads (million)	make_read_word (%)	map_reads (%)	WordPair (%)	_Hashtable (STL function) (%)	uninitialized_copy (STL function) (%)	total run-time (seconds)
1	1	15.24	13.72	12.81	4.88	4.27	8.249
Genome (million)	Reads (million)	map_reads (%)	WordPair (%)	make_read_word (%)	sort_by_key (%)	build_seed_hash (%)	total run-time (seconds)
100	100	21.32	13.46	11.74	6.22	4.74	1055.167
Genome (million)	Reads (million)	map_reads (%)	WordPair (%)	make_read_word (%)	sort_by_key (%)	operator= (STL function) (%)	total run-time (seconds)
200	200	40.84	9.02	7.59	4.95	3.72	2184.421
Genome (billion)	Reads (million)	map_reads (%)	WordPair (%)	make_read_word (%)	sort_by_key (%)	operator= (STL function) (%)	total run-time (seconds)
3.1	260	90.72	1.24	1.07	0.92	0.62	10851.551

Figure 4.6: Data showing the five most computation-intensive functions of RMAP for the four datasets used.

CHAPTER 5. Platform

This chapter gives an overview of the Convey HC-1 system [Convey (2012)], the computing platform used in this work. It describes the architecture of the platform, lists some of its features, and its applicability for this work.

5.1 HC-1 system

System organization. The Convey HC-1 system is a hybrid computer containing two processor architectures: a 2.4 GHz, eight core Intel host processor and a reconfigurable coprocessor based on FPGA technology, as shown in Figure 5.1. It has a two socket motherboard, with the host processor in one socket and the coprocessor in the other. Both the host processor and the coprocessor share the same global memory space. However, the physical memory is laid out in such a way that both the host processor and coprocessor are associated with certain regions of this memory space (256 GB and 64 GB respectively) to which they have the fastest access. Within an application, both use the same virtual address to access a particular physical location. The coprocessor is typically used for implementing custom instruction sets which are

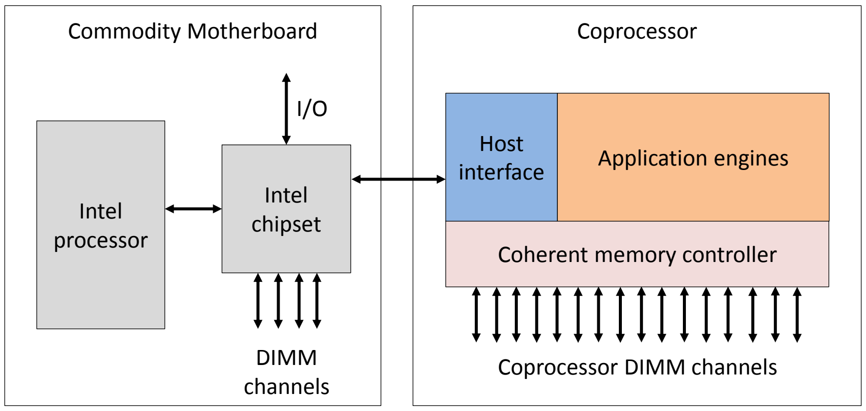


Figure 5.1: Convey HC-1 system organization.

highly optimized for the application being accelerated.

Coprocessor organization. The coprocessor architecture has three major components: 1) Application Engine Hub (AEH), 2) Application Engines (AEs), and 3) Memory Controllers (MCs). The coprocessor organization is shown in Figure 5.2.

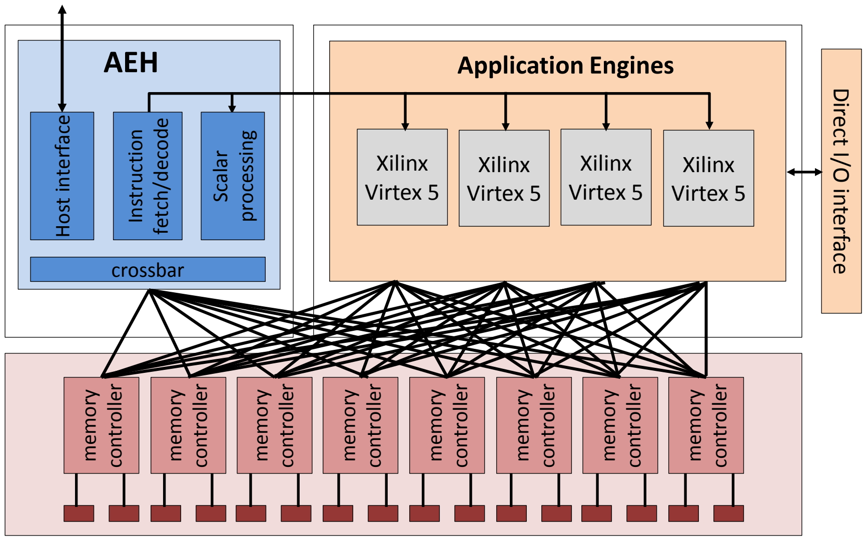


Figure 5.2: Coprocessor organization.

1) **Application Engine Hub (AEH).** This acts as an interface between the host processor and the AEs. It has an instruction fetch-and-decode unit, and processes scalar instructions. It processes data fetch requests from the host processor to the coprocessor memory by routing these requests to the MCs.

2) **Application Engines (AEs).** These contain the four reconfigurable units of the Convey system. Each unit is a Xilinx Virtex 5 LX330, FF1760 FPGA. The AEs are used to implement custom designs for accelerating an applications. The AEs are connected to the AEH by a command bus that transfers opcodes and scalar operands. They are also connected to memory controllers via a high bandwidth network of point-to-point links.

3) **Memory Controllers (MCs).** The coprocessor architecture has eight memory controllers (MCs), each supporting two DDR2 memory channels connected to the global memory space, providing an aggregate of 80GB/sec bandwidth with scather-gather DIMMs. The MCs translate the virtual addresses to physical addresses for the AEs.

Memory Controller Interface. The eight memory controller interfaces connect the AEs directly to the memory controllers, and each MC physically connects to 1/8 of the coprocessor memories. The AEs and MCs are linked by a 300 MHz bus. However, in order to ease timing in the FPGA, the 300 MHz interface is converted into two 150 MHz memory ports (odd and even) to/from the AE personality.

Personality. Is a term used to refer to the set of instructions implemented by the design running on the coprocessor. These instructions can be seen as an extension to the host processor instruction set. The HC-1 system is capable of switching between multiple personalities to execute different types of code, but only one personality can be active on the coprocessor at any given time. Each personality is associated with an ID or a personality number and using these IDs, the system can load different personalities on the AEs and switch between them. A Personality is installed on the coprocessor using a bitfile (generated by Xilinx tool chain) which contains the coprocessor instruction set for that personality.

Coprocessor programming model. When an application is implemented on the coprocessor of the HC-1, it contains both host processor and coprocessor instructions and control is switched between the two using cache coherence hardware to minimize latency. Before the application is started, the personality for that application is installed on the coprocessor. When the application starts running on the host processor, a block of instructions is dispatched to the coprocessor. These instructions initialize the coprocessor and load arguments passed by the host processor into the coprocessor registers. While the coprocessor is executing, the host processor can continue its operation or wait for the coprocessor to finish, depending upon the application's requirements.

Performance monitor tool. The HC-1 system provides a performance monitor tool for the design implemented on the coprocessor. This tool lists for each AE, the number of clock cycles it runs, the load/store count for each MC, stalls encountered during load/store by each MC, and memory bandwidth utilization. A performance monitor flag "PERFMON" needs to be set while generating the bit file to enable the tool for performance data collection.

5.2 Applicability of HC-1 platform for RMAP.

The Convey HC-1 system can provide good performance and ease of implementation for RMAP, by virtue of some of its salient features which are listed below.

- **Ease of HW/SW co-design.** Based on the results from Section 4.4, the reads mapping function is found to be the most computationally intensive function in RMAP. The hybrid-core computing feature of the HC-1 system helps in separating the reads mapping function from the rest of the software, and implementing it on the coprocessor, in addition to providing a highly abstract interface for transferring control between the two. The reads mapping function implemented on the coprocessor can thus be invoked as just another function from the RMAP code running on the host processor.

- **Shared global memory space.** The mapping function implemented on the coprocessor is memory intensive, especially the *Chroms key search unit* described in Chapter 7. By placing the arrays accessed by the coprocessor in the physical memory region (referred to as coprocessor memory) to which it has the fastest access, the memory access latencies can be greatly reduced. Also, the Convey operating system and compilers provide mechanisms to allocate data in the appropriate region(s) of memory, both statically as well as by migrating data at runtime. For faster software processing, data can be allocated in the host memory initially and then can be moved to the coprocessor memory if it is being accessed by the coprocessor.

- **Parallel memory access.** The Convey system provides eight memory controllers, each having an even and odd port, effectively making it a total of 16 memory controllers for the coprocessor memory. All of these controllers can work in parallel to service data read and write requests. In this work, data fetch requests for some of the arrays (described in Section 7.4.2) are placed in the coprocessor region of memory. By allocating these arrays such that they are on different memory controllers, high bandwidth parallel data fetch can be performed.

- **Crossbar switch, fetch request read order and write complete interface.** The user does not have to worry about ensuring that FPGA memory requests are sent to the appropriate MCs. The "crossbar switch" feature abstracts the user from the MC layer, and routes the requests to the correct MCs. Also, the "MC read order" feature ensures that data is received in the order in which it was requested, helping to ease implementing designs where inorder processing is required. The "write complete interface" indicates completion of a write to memory. This prevents the hardware design from reading stale data, which is important for the read mapping function.

CHAPTER 6. Architecture Overview

This chapter describes how RMAP is laid out on the hybrid architecture of the Convey HC-1 system. RMAP is implemented as a HW/SW co-designed solution, with the software running on the Host processor and the custom hardware on the Coprocessor of the Convey system, as shown in Figure 6.1. The software running on the host processor (software unit) calls the coprocessor (hardware unit) via a coprocessor function call and provides the required data as input (arguments) by means of load instructions (described in low level assembly) which move the data into coprocessor registers. After the hardware completes execution, the results of the hardware unit are passed back to the software unit for post processing. Details of both units are provided next.

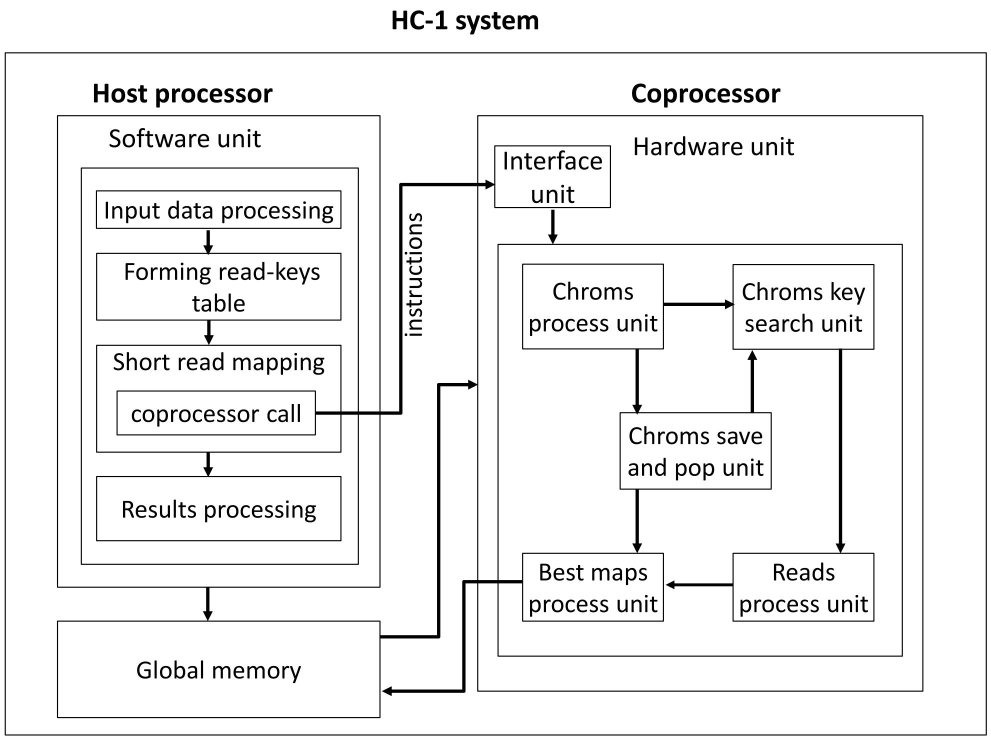


Figure 6.1: RMAP Software-Hardware Architecture.

6.1 Software Unit

The software component of RMAP executes on the host processor of the HC-1 system. This section describes the functionality of the software unit and the differences with respect to the original RMAP functionality detailed in Section 4.2.

The basic RMAP code (Chapter 4) was modified to facilitate ease of porting the read mapping function to the FPGA on the convey coprocessor. The code was redesigned while retaining the basic functionality, and transforming complex data structures such as Standard Template Library (STL) containers into arrays. This simplified allocating memory and passing addresses to the hardware design. An unordered map data structure, used for building the "read-keys table", is one of the data structures that was replaced with a two dimensional array in this modified version of RMAP.

The following lists the modules in the RMAP code and describes how this modified version differs from the one described in Section 4.2.

1) **Input data processing.** The chromosomes that are read from a file are stored in a character array instead of a vector. As in the original code, the reads are read from the input reads file and are separated into "upper", "lower" and "bads", and stored in separate arrays (reads upper array, reads lower array and reads bads array respectively). The names for each read are stored in another array. The vector for storing the two sets of best mapping information is replaced by a set of arrays. Two arrays (score1 and score2) are created to store the first and second set of scores and another two (chrom ID1 and chrom ID2) for storing the sites of best mapping. The strand is common for the two sets of mapping information, and is hence stored in a single array.

2) **Forming a read-keys table.** The seeds are read from an input seed file to avoid using complex data structures associated with seed generation. Only one seed is used for this implementation. In the modified code, the "seed-keys table" is a two column, two dimensional array with read-keys (reads masked with seed) in the first column and read indices in the second. This table is sorted using merge sort algorithm, so as to group similar read-keys together. These sorted read-keys, and the first and last read indices associated with each

read-key (as a result of sorting) are put in a new two dimensional array with three columns. The first column contains the read-keys, and the second and third columns contain the first and last read indices respectively. This two dimensional array acts as the "read-keys table" replacing the unordered map structure used in the original RMAP code.

3) **Short read mapping.** This is the section of RMAP chosen for acceleration, and is implemented as an FPGA-based custom hardware design on the coprocessor. It is invoked from the software via a coprocessor function call. The arguments to this coprocessor function are: chromosome array starting address, chromosome array ending address, read-keys table starting address, read-keys table ending address, reads upper array base address, reads lower array base address, reads bads array base address, score1 base address, score2 base address, chrom ID1 base address, chrom ID2 base address, strand array base address, current seed and max mismatches. The output of the hardware unit is the best maps information for each read. This information is stored in arrays that were allocated during the "Input data processing" step of RMAP.

4) **Results processing.** This part works similar to that in the original RMAP code. The only difference is that the processing of best mapping information for identifying and eliminating ambiguous reads involves arrays as opposed to the vectors used in RMAP.

6.2 Hardware Unit

The hardware unit implements the mapping function of RMAP on one of the Application Engines (AEs) of the HC-1 coprocessor. This section provides an overview of the modules in the hardware design for the mapping function. The software unit uses a coprocessor function call to pass arguments to and start the hardware engine. The arguments passed to the hardware unit are listed in Section 6.1.

The following gives a brief summary of the modules within the hardware unit. Chapter 7 provides further details for each module.

Chroms process unit. This unit processes the input chromosomes that were stored in a character array by the software unit. It has the following sub-units: *Chroms fetch unit*, *Chroms split and shift unit* and *Chroms key process unit*. The *Chroms fetch unit* performs the function of fetching chromosomes from the array. The *Chroms split and shift unit* encodes each fetched chromosome base (8 bits), separates the encoded base into “upper” bit, “lower” bit, and “bads” bit, and combines this data for the incoming chromosome bases to form three 36-bit wide “upper”, “lower”, and “bads”, which is stored in registers. The chromosomes’ upper and lower are passed to the *Chroms key process unit* to form 64-bit chromosome keys. The output of *Chroms key process unit* is sent to the *Chroms key search unit*. The chromosomes upper, lower and bads are also sent to the *Chroms save and pop unit* for on-chip storage into Block RAMs (BRAMs).

Chroms save and pop unit. This unit manages the data sent by the *Chroms process unit* for storage. The sub-units of this unit are: *Free list controller*, *Chroms save unit* and *Chroms pop unit*.

The *Chroms save unit* receives data to be stored in BRAMs from the *Chroms process unit*. It saves this data to BRAMs based on the addresses received from the *Free list controller*, which maintains a list of addresses available in the BRAMs. The *Chroms pop unit* passes the data stored in the BRAMs to *Best maps process unit* based on the output of *Chroms key search unit* and the control signals received from the *Best maps process unit*.

Chroms key search unit. This unit performs a search of the chromosome keys in the “read-keys table”. The chromosome keys received from the *Chroms process unit* are searched in the “read-keys table” to look for matches with the read-keys stored in the table. The *Keysearcher* unit, a sub-unit, performs a binary search for each incoming chromosome key. If there is a match, then the match location is passed to the *Reads process unit* to fetch the reads data corresponding to the match location. The search ID corresponding to the match is passed to the *Chroms save and pop unit* for retrieving the chromosome data. The reads data and chromosome data are sent to the *Best maps process unit* for scoring.

Reads process unit. This unit fetches reads data for the read-keys that matched with the chromosome keys. The *Get reads IDs unit* and *Get reads data unit* are sub-units.

The match locations stored by the *Chroms key search unit* are read by the *Get reads IDs unit*, and these locations are used as addresses to fetch the reads IDs corresponding to each match location. After the read IDs are received, the *Get reads data unit* uses these read IDs as indexes and sends fetch requests for read's "upper", "lower", "bads", "score1", and "score2" data associated with each read ID. Before sending a fetch request for this data, the corresponding read ID is checked in the *Content Addressable Memory unit* (CAM), which is part of the *Best maps process unit* to ensure the requested read ID is not currently being scored. If the read ID is not in the CAM, then the fetch request is sent and that read ID is added to the CAM. The fetched read's "upper", "lower", "bads", "score1" and "score2" data are stored in fifos and the data from them is used by *Best maps process unit*.

Best maps process unit. This unit identifies the best maps for the reads and stores the best mapping information in memory. The *Scoring unit*, *Best maps store unit* and *Content addressable memory unit* (CAM) are sub-units. This unit retrieves the read's "upper", "lower" and "bads" that was fetched by the *Reads process unit*. It also requests the chromosome's "upper", "lower", "bads" and "chromosome ID" corresponding to the matched chromosome key from the *Chroms save and pop unit*, and sends it to the *Scoring unit* for calculating the score of the match. The *Scoring unit* checks for mismatches between the read data and the chromosome data and provides the number of mismatches as a score. If the score is within an acceptable range, then the score, the chromosome ID and the strand for that read are written to memory. Once these writes complete, the corresponding read ID entry for that read is removed from the CAM.

The output of the hardware unit is the best maps information for every read ID whose corresponding read-key matched with a chromosome key. The data written to main memory for each read ID are its mapping score, location on the chromosome, and the strand of the chromosome.

CHAPTER 7. Hardware Design Implementation

The hardware design performs the mapping of chromosomes with short reads. The mapping process in hardware is divided into five stages as shown in Figure 7.1:

- Chroms Process Unit
- Chroms Save and Pop Unit
- Chroms Key Search Unit
- Reads Process Unit
- Best Maps Process Unit

The chromosomes fetched from memory are processed in the *Chroms process unit* and are sent to *Chroms save and pop unit* to store the chromosome data in BRAMs. The processed chromosome data is also passed to the *Chrom key search unit* to look for chromosome keys in the read keys table. As the search for each chromosome key is completed, the *Reads process unit* works to fetch the reads data associated with the matched read-key; the *Chroms save and pop unit* fetches the chromosome data associated with the matched chromosome key. The reads data and chromosome data are then passed to the *Best maps process unit* for scoring and storing results.

The remainder of this chapter describes further details of the modules that compose the hardware unit.

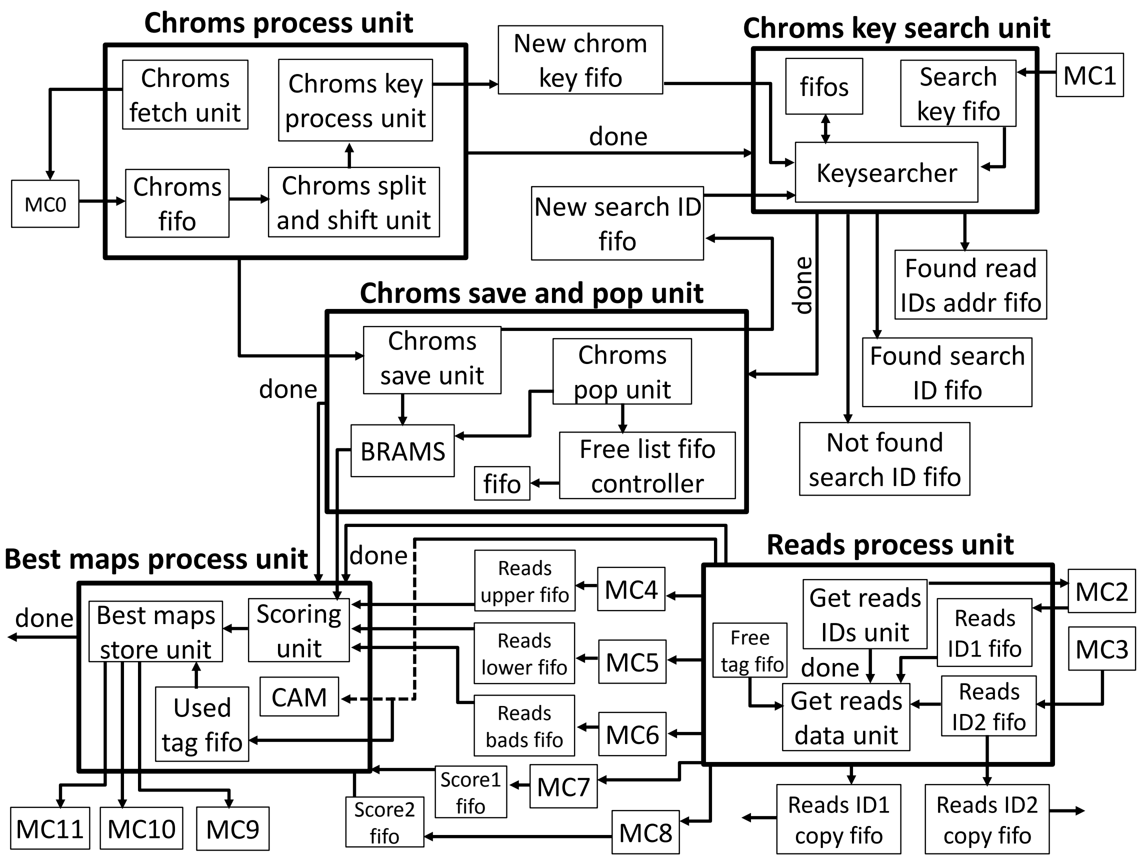


Figure 7.1: Overview of the read mapping function hardware design.

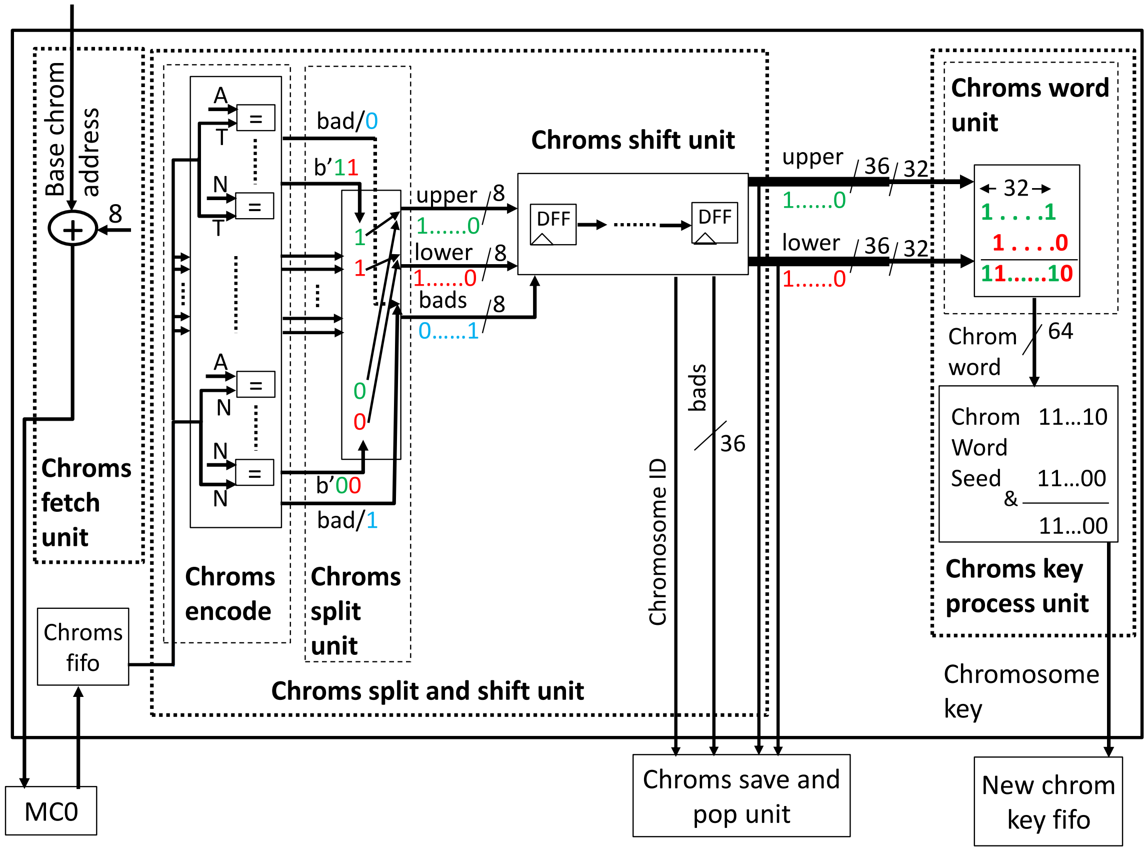


Figure 7.2: Chroms Process Unit.

7.1 Chroms Process Unit

Chromosomes are read from memory, and are then processed to form the chromosomes "upper", "lower", "bads" and chromosome key, which are sent to *Chroms save and pop unit* and *Chroms key search unit*. Figure 7.2 shows the architecture of this unit. It is composed of 3 sub-units: 1) Chroms fetch, 2) Chroms split and shift, and 3) Chroms key process.

7.1.1 Chroms Fetch

The chromosome bases that were stored in an array by the software unit are fetched from memory. The base address of the array is incremented and data is fetched from each address till it reaches the end of the array.

The chromosomes are fetched in quad words containing 8 bases. The fetch requests are made to memory controller 0 (mc0) as shown in Figure 7.2. The fetch address is incremented by 8 after every fetch; it starts with the base address of the array and keeps incrementing till the address of the last chromosome base. The data from the memory controller is stored in *Chroms fifo*. The unit stalls further fetch of chromosome bases, until some of the previously fetched data stored in *Chroms fifo* is read.

7.1.2 Chroms Split and Shift

This unit reads the chromosome quad words from *Chroms fifo* and processes them in its internal blocks listed below:

- Chroms encode unit
- Chroms split unit
- Chroms shift unit

Chroms Encode. This unit converts each chromosome base from an ASCII 8-bit representation to 2 bits. A/a, C/c, G/g, T/t ASCII representations are encoded as 00, 01, 10 and 11 respectively. If a base is none of the 8 alphabets mentioned above, then a flag called “bad bit” is set for that particular base. The output of the *Chroms encode unit* is sent to the *Chroms split unit.*

Chroms Split. Each of the encoded input bases, which are 2-bit wide, are split into their upper and lower bit. The most significant bit of each 2-bit input is packed in to an 8-bit signal designated as “upper”. Similarly the least significant bit of the 2-bit input is packed in to another 8-bit signal called “lower”. In Figure 7.2, the green colour bit represents upper while the red colour bit represents lower, with the blue colour representing the bad bit. The bad bit for each base is packed in to another 8-bit signal. These one byte signals are then passed to the *Chroms shift unit.*

Chroms Shift. This unit combines each one-byte signal formed in the *Chroms split unit*, with incoming bytes to form a 36-bit wide signal. Once the first 36 bits of each of the upper, lower and bads signals are formed, a data valid signal is sent out from this unit along with the data to the *Chroms key process unit* and the *Chroms save and pop unit*. A counter is incremented for every bit shifted in after the first 35. This count acts as the chromosome ID for each 36-bit wide chromosome data created. Figure 7.2 represents this unit as a series of shift registers.

7.1.3 Chroms Key Process

This unit receives the lower 32 bits of the 36-bit wide chromosome “upper” and “lower”. These two 32-bit wide signals are combined to form a 64-bit chromosome word in the *Chroms word unit*. Figure 7.2 illustrates how the green bits of chromosome “upper” signal are combined with the red bits of the chromosome “lower” signal to form the 64-bit chromosome word. This chromosome word is masked with the seed received from the host processor to form a chromosome key. Each of the keys generated are pushed into a fifo. This key will be searched for in the “read-keys table” to look for matches with the read-keys.

7.2 Chroms Save and Pop Unit

The purpose of this unit is to manage the flow of Chromosome data from *Chroms process unit* to *Best maps process unit* through the BRAMs. This unit receives chromosome data from the *Chroms shift and split unit*, and stores this data in Block RAMs, as shown in Figure 7.3. There are four Block RAMs used in the design for storing chromosomes’ “upper”, “lower” and “bads” and chromosome IDs. Each block RAM has 512 slots for storing data. Data is stored in BRAMs if any of these 512 slots are free. Addresses corresponding to the free slots are provided by the *Free list controller unit* and these addresses act as search IDs for chromosome key search. Once a search is complete, the corresponding search ID becomes free and can be reused. Also, the data stored in the BRAMs corresponding to this search ID are passed to the *Best maps process unit* for scoring.

7.2.1 Free List Controller

This unit controls the addresses of the free slots available in the Block RAMs to save chromosome data. These addresses are stored in the *Free list fifo* shown in Figure 7.3. The

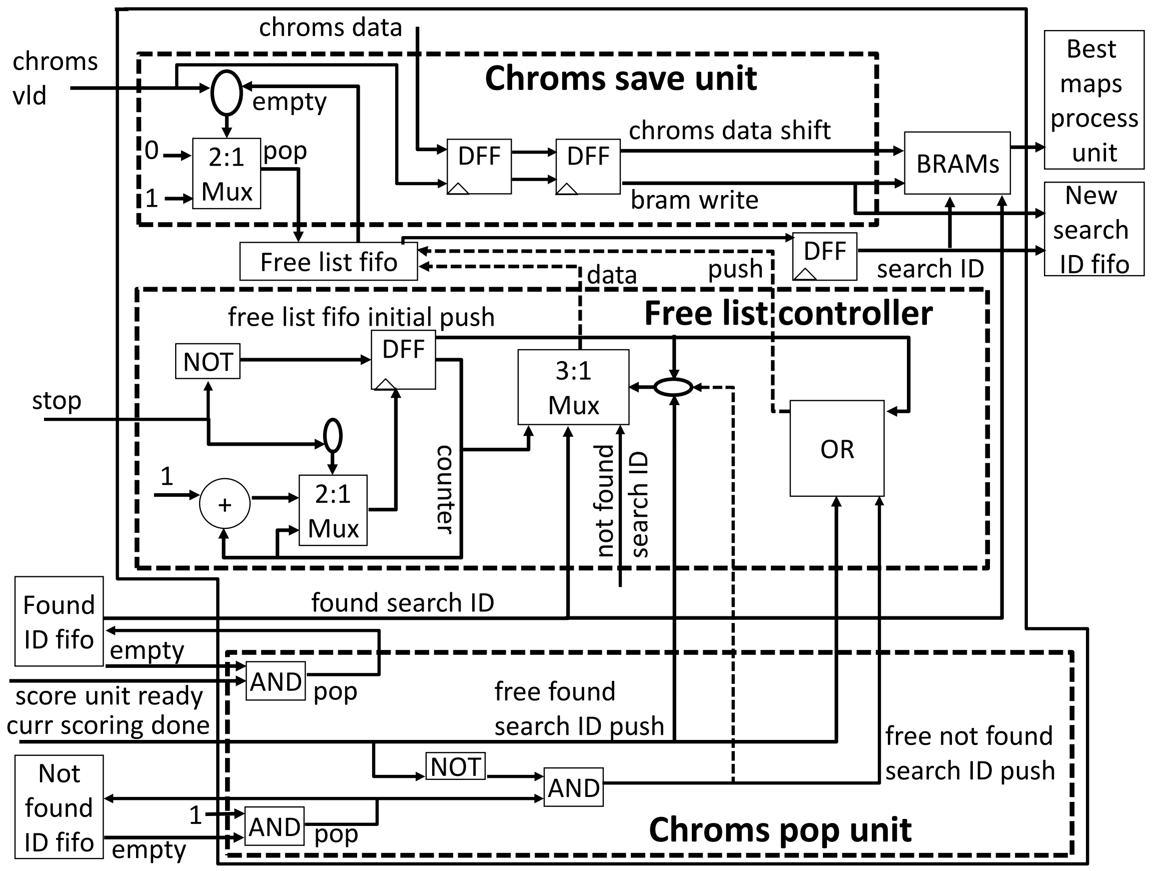


Figure 7.3: Chroms Save and Pop Unit.

Free list controller initially fills the fifo with 512 addresses, starting from 1 to 512. When all 512 addresses are added to the fifo, “stop” signal is issued to stop filling the fifo with more addresses. When data to be stored in BRAM arrives, an address is popped from the *Free list fifo* by the *Chroms save unit* and the data is stored at that BRAM’s location. This unit receives a control signal from the *Chroms pop unit* to push freed addresses (search IDs) back onto the *Free list fifo*. When a search corresponding to a search ID is not successful (i.e. no match in the read-keys table), then the “free not found search ID push” signal is raised by the *Chroms pop unit* and the ID is pushed into the *Free list fifo*. If a search is successful, then the “free found search ID push” signal is raised and the ID is pushed into the *Free list fifo* once the *Best maps process unit* is ready to score the chromosome key associated with the ID. As seen in Figure 7.3, there are three different signals which can push data into the *Free list fifo*, a three input multiplexer decides which of these is selected at any given time.

7.2.2 Chroms Save

When the “chroms vld” flag is high, this unit checks for available addresses in the *Free list fifo*. If an address is available, then it is popped and the chromosome data is stored at that location in BRAM. The BRAM address popped from the *Free list fifo* is also pushed onto the *New search ID fifo* to be passed to *Chroms key search unit*. This address acts as the search ID for the search unit.

7.2.3 Chroms Pop

This unit checks for search IDs for which the associated chromosome key search has completed and sends a control signal to the *Free list controller* to reinsert them in to the free list pool. When search IDs associated with an unsuccessful search are available, they are popped from the *Not found search ID fifo*. A request is sent to the *Free list controller* to push them into the *Free list fifo*. Similarly, search IDs associated with a successful search are checked for their availability in the *Found search ID fifo*. Once these search IDs become available, they are popped only if the *Best maps process unit* is ready to score the corresponding chromosome key and read key.

7.3 Chroms Key Search Unit

This unit performs a major aspect of the read mapping process. It implements a binary search of the read-keys table to determine if a given chromosome key can be found. Each chromosome key has a search ID (obtained from the *Chroms save and pop unit*) associated with it, which helps in tracking the search process for that key.

The chromosome key and the search ID required to begin a search are read from the *New chrom key fifo* and *New search ID fifo* respectively. If a chromosome key is found in the read-keys table, then its associated search ID and the location where it matched in the read-keys table are stored in FIFOs. The values stored in these FIFOs are used by the *Chroms save and pop unit* and *Read process unit*. This unit receives the base (start) and last (end) address of the read-keys table as input from the software unit. It has 5 fifos and *Keysearcher* unit, as

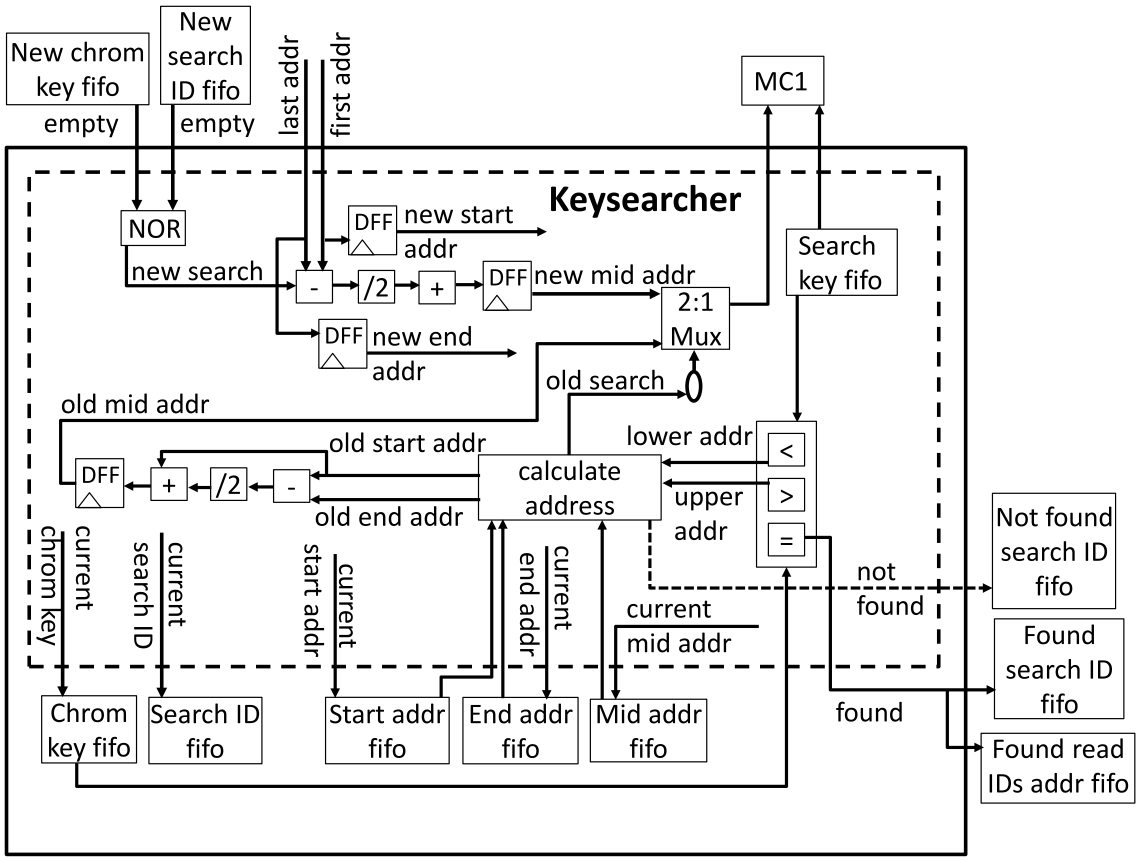


Figure 7.4: Chroms Key Search Unit.

shown in Figure 7.4. The intermediate data during a binary search propogates through these fifos and the *Keysearcher* unit. Figure 7.4 shows the architecture of the chromosome key search hardware.

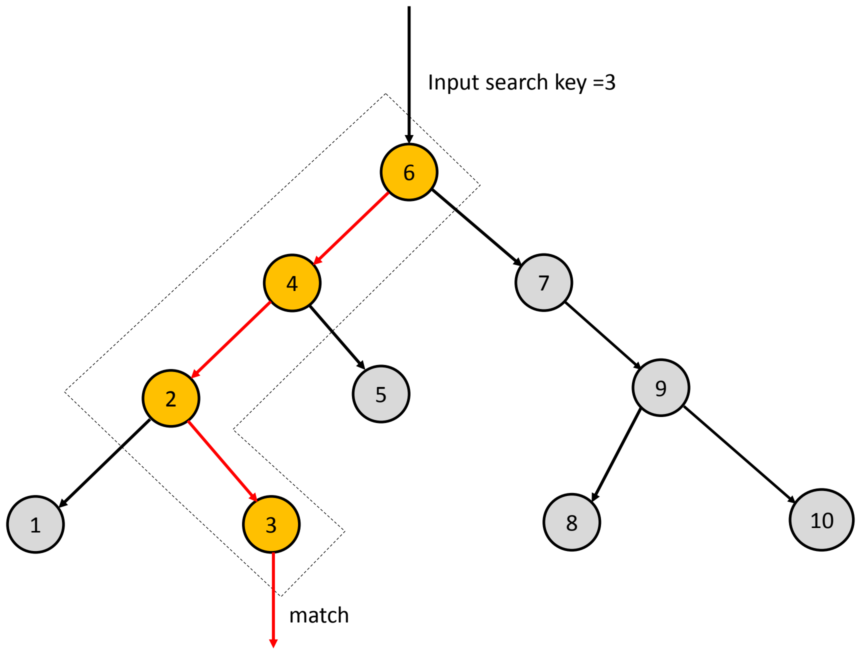


Figure 7.5: Binary search algorithm.

7.3.1 Keysearcher

This unit implements the binary search algorithm shown in Figure 7.5. Since this unit primarily generates memory requests, it can become a bottleneck for the design depending on the size of the read-keys table (i.e. search tree).

Separation of new and old chromosome key search. The *Keysearcher unit* can be divided into two processes. First, the upper portion of Figure 7.4 is responsible for initiating new chromsome key requests. Second, the bottom portion of Figure 7.4 manages chromosome key searches that are in progress. This unit has a read request counter to count the number of key search requests sent to the memory controller. This is to ensure that the five intermediate data fifos and the *Search key fifo* interacting with memory controller 1 (MC1) do not overflow.

New chromosome key search. For every new chromosome key that is searched, it and its associated search ID are popped from the *New chrom key fifo* and *New search ID fifo* respectively. A middle address is calculated using the base (start) and last (end) address of the read-keys table. A data request from this middle address is sent to MC1. Along with this request, the current starting address, middle address, end address, chromosome key and search ID are pushed onto the *Start addr fifo*, *Mid addr fifo*, *End addr fifo*, *Chrom key fifo* and *Search ID fifo* respectively. When the data requested from the middle address is received by MC1, it is pushed into the *Search key fifo*.

Old Chromsome key search. The data in the *Search key fifo* is compared with the chromosome key being searched to check for a match. Based on the results of the comparison, the search is either continued or terminated.

Match. If there is a match, then the middle address for the matched chromosome key obtained from *Mid addr fifo* is the location of the match. This address is written to the *Found read IDs addr fifo*. Also the search ID corresponding to this chromosome key is written to the *Found search ID fifo*.

Not a match. If the data from the *Search key fifo* and the chromosome key do not match and the chromosome key is smaller than the data, then the *calculate address unit* shown in Figure 7.4 selects the lower address range of the read-keys table. If the chromosome key is greater than the search key, then the *calculate address unit* selects the upper address range of the read-keys table.

Search continues. After this address range calculation, if the start address is less than the end address, then the search for the chromosome key is continued and a new middle address is calculated and another request for the same search ID is sent to MC1. Again the current starting address, middle address, end address, chromosome key and search ID are all pushed into their respective FIFOs. This process is repeated until the start address is greater than the end address, or a match is found.

Search ends. If the start address becomes greater than the end address, then the search for the chromosome key is terminated. The search ID is pushed onto the *Not found search ID fifo* by issuing “not found” signal.

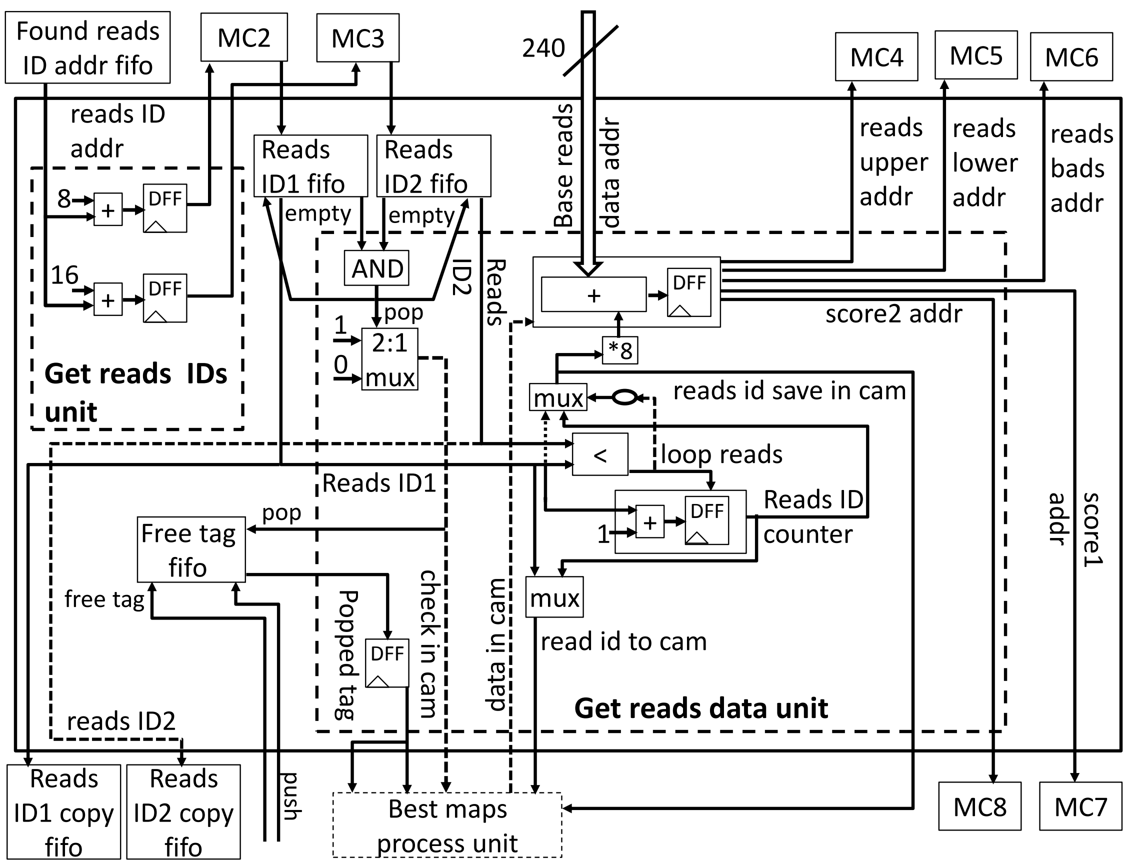


Figure 7.6: Reads Process Unit.

7.4 Reads Process Unit

The function of this unit is to fetch reads data (i.e. reads “upper”, “lower”, “bads”) from memory and send it to the *Best maps process unit* for scoring. It fetches read IDs from memory using the read ID addresses output from the *Chroms key search unit*. Using these read IDs, reads “upper”, “lower” and “bads” data are fetched from memory. Additionally, the two best scores for the read IDs are fetched. A *Free tag fifo*, similar to the *Free list fifo* in the *Chroms save and pop unit*, is filled with free tags. For each read ID, a tag is popped and associated with the read ID. The tag and read ID are sent to the *Best maps process unit*. After the *Best maps process unit* scores the read associated with the read ID and writes the score to memory, the free tag is pushed back onto the *Free tag fifo*. Figure 7.6 shows the architecture of the *Reads process unit*.

7.4.1 Get Reads IDs

This sub-unit fetches two read IDs for each chromosome key and read-key match. The address of the match location is read from the *Found read IDs addr fifo*. This address points to the read-key stored at that address. This address when incremented by 8 is the address of first read ID. The same address incremented by 16 is the address of last read ID. These two read IDs are fetched using memory controller 2 (MC2) and memory controller 3 (MC3) respectively. The read IDs received by MC2 and MC3 are pushed onto *Read ID1 fifo* and *Read ID2 fifo* respectively.

7.4.2 Get Reads Data

This sub-unit fetches the data associated with each read ID. The read IDs are read from *Read ID1 fifo* and *Read ID2 fifo*. The two read IDs indicate the set of reads mapping to the chromosome key, for which there was a match in the read-keys table. The first read ID indicates the beginning of the list of reads to score against the chromosome key. The second read ID indicates the end of the list. Each read ID in the list is used to fetch the "upper", "lower", "bads", and the two best scores of the read associated with it. The two read IDs are pushed onto *Reads ID1 copy fifo* and *Reads ID2 copy fifo* respectively to make a copy of the read IDs to be used by the *Best maps process unit*.

In Figure 7.6, the "base reads data addr" bus specifies the base addresses of 5 arrays that contain the data to be fetched for a read ID. Each read ID in the list is an offset and is added to the 5 base addresses to fetch the data associated with it. The fetch request for "upper", "lower", "bads" and two scores are sent to MC4 to MC8 in that order. This data when fetched from memory are stored in *Reads upper fifo*, *Reads lower fifo*, *Reads bads fifo*, *Score1 fifo* and *Score2 fifo* respectively.

Before sending the data fetch request, each read ID is looked up in the *Content addressable memory unit (CAM)* to determine whether the read ID is being processed. The processing for a read ID involves 1) fetching of data associated with that read ID, 2) scoring the data with the corresponding chromosome data, and 3) storing the best maps data. If the read ID is in

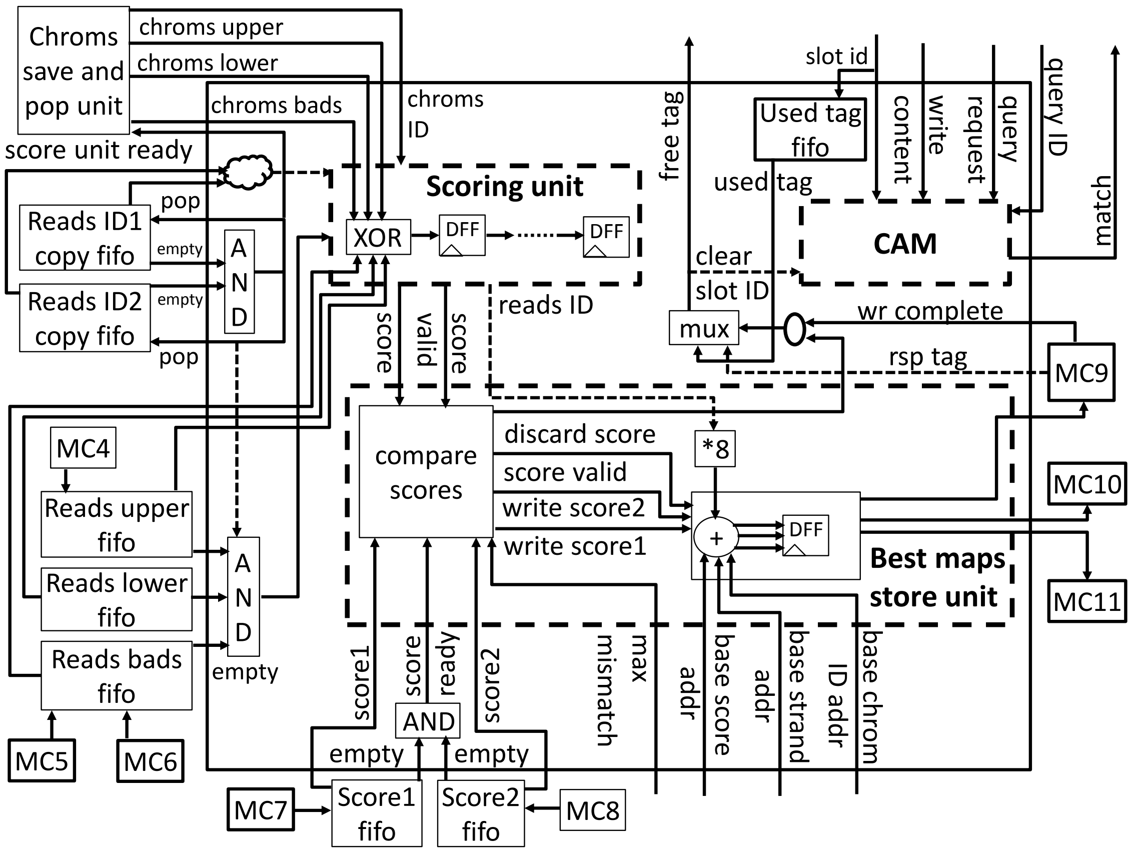


Figure 7.7: Best Maps Process unit.

CAM, the fetch request is stalled until the corresponding read ID entry is cleared from CAM.

When a data fetch request is sent, a read ID is added to CAM to indicate the start of processing of that read ID. A free tag is popped from *Free tag fifo* and sent to CAM to assign a tag to the read ID being added to CAM. This free tag is also added to *Used tag fifo* in the *Best maps process unit.*

7.5 Best Maps Process Unit

The best maps processing is performed by 1) scoring the mapped reads' "upper", "lower" and "bads" with the chromosomes' "upper", "lower" and "bads", 2) deciding the best maps based on the score, and 3) storing the result for the best maps, as shown in Figure 7.7.

The *Reads ID1 copy fifo* and *Reads ID2 copy fifo* are popped and "score unit ready" signal is sent to the *Chroms save and pop unit.* Upon receiving the "score unit ready" signal, the

Chroms save and pop unit sends the chromosome data (i.e. upper, lower and bads) from the Block RAMs to the *Scoring unit* for scoring. The Chromosome ID is also provided.

Read ID1 from *Reads ID1 copy fifo* and read ID2 from *Reads ID2 copy fifo* are compared. If they are not same, read ID1 is incremented till its value is equal to read ID2. For each read ID, from read ID1 to read ID2, data is read from *Reads upper fifo*, *Reads lower fifo* and *Reads bads fifo*. This data is passed along with the corresponding read ID to the *Scoring unit*. This logic is encapsulated in the cloud shown in Figure 7.7.

This unit has a *Used tag fifo* in which data is pushed from the *Reads process unit*. It stores the tags for the read IDs which are being processed. The processing of a read ID completes when its best map data are written to memory successfully or when no best map is found.

7.5.1 Scoring

This unit computes the mismatches between a read and a given chromosome location. The read's "upper" and "lower" are compared bit-wise with the chromosome's "upper" and "lower". The "upper", "lower" and "bads" of the read and the chromosome go through a series of logic operations (XOR and OR) as shown in Figure 7.8. The set bits of the resultant signal indicate where mismatches occur between the chromosome and read. These bits are summed to give the score. It takes 7 clock cycles to generate a score. Once the score is calculated, "score valid" signal is sent to the *Best maps store unit*, as shown in Figure 7.7.

7.5.2 Best Maps Store

This sub-unit uses the score computed by the *Scoring unit* to decide whether a mapping is to be considered. If so, then the score, the map location on the chromosome (i.e. the chromosome ID), and the strand of the chromosome are written to memory. Score1 and score2 are popped from *Score1 fifo* and *Score2 fifo* respectively. These two scores correspond to the two best scores associated with the read ID that is being processed. The two best scores are compared with the score computed by the *Scoring unit* (current score), and one of these two scores is updated in memory if the current score is better. This is done in *compare scores*, shown in Figure 7.7.

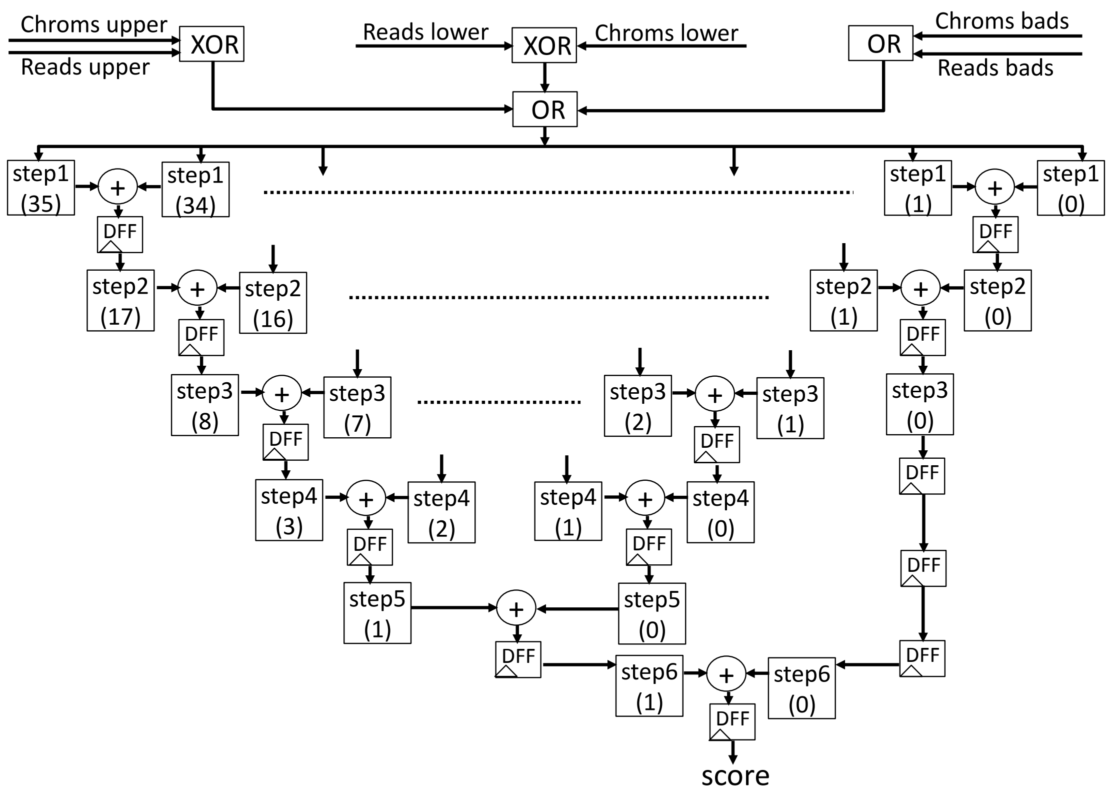


Figure 7.8: Scoring unit.

7.5.3 Content Addressable Memory (CAM)

The CAM is used to prevent stale score values from being fetched for read IDs that are in the process of being scored. The *Reads process unit* fetches data for read IDs only after checking in the CAM to see if the scoring and storing for a previous chromosome key that matched that particular read ID is not in progress. The *Reads process unit* sends a query request for a read ID to this unit. After one clock cycle, the CAM issues a *match* signal indicating whether the read ID is present in the CAM or not. The archtitecture of CAM is shown in Figure 7.9.

The CAM contains 100 slots for storing read IDs for which data processing is currently in progress. The availability of a slot is controlled by a *free tag fifo* located in the *Reads process unit*. It takes the read ID and the tag associated with it as input from the *Reads process unit*. The read ID is stored in a free slot and the tag acts as the slot ID. After a read ID completes processing, the tag associated with that read ID is cleared from the CAM.

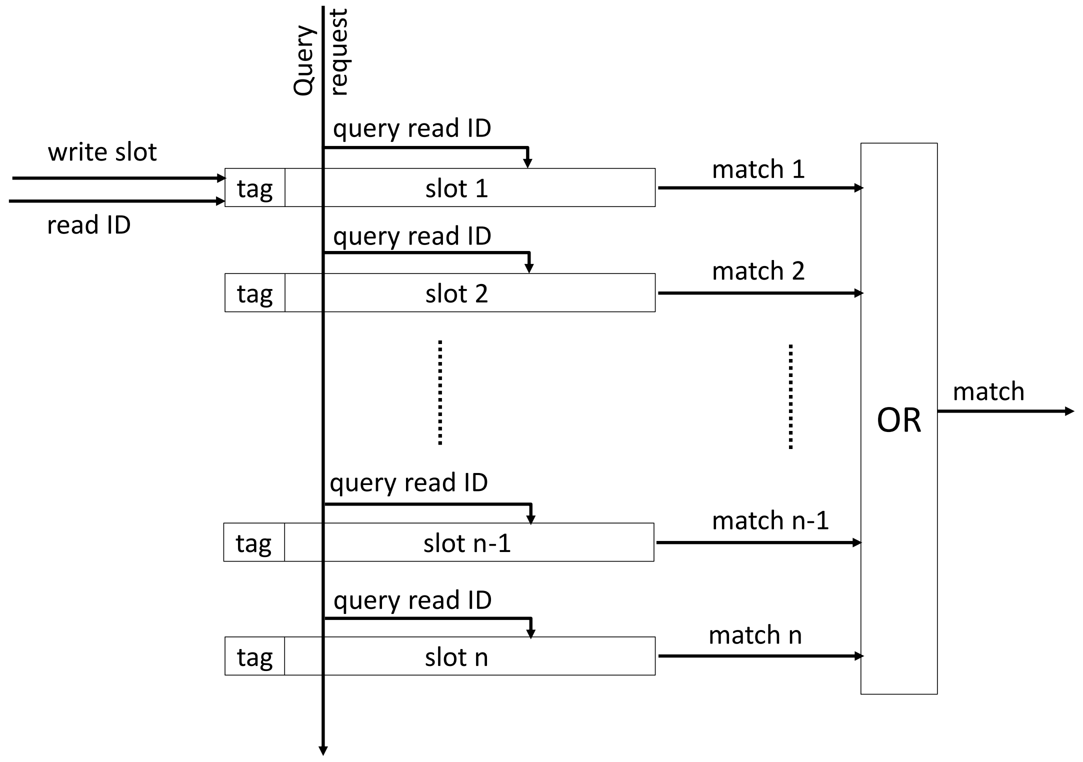


Figure 7.9: Content Addressable Memory unit.

CHAPTER 8. Evaluation Methodology

8.1 Experimental setup

Platform. The HC-1 system (described in Chapter 5) is the platform used for evaluating the design presented in Chapter 6 and 7. The Intel-based host processor on HC-1 runs the software portion of the design, while the hardware portion is implemented on the reconfigurable logic of the coprocessor. The software running on the host processor calls the coprocessor (hardware unit) via a coprocessor function call that provides the required input parameters and initiates coprocessor execution. Figure 6.1 illustrates this setup. The hardware design uses Convey's optional features, such as the "crossbar switch" to provide a simplified memory interface, the "read order queue" to ensure data is received in the order that it was requested, and the "write complete interface" to indicate the completion of a write to memory.

Software. For performing experiments, the original RMAP software (described in Chapter 4) and the modified RMAP software (described in Section 6.1) run the basic RMAP code with the following parameters: 1) mapping based on mismatches, 2) three maximum mismatches, 3) two best maps information for each read, 4) one seed, and 5) read width of 36. Both software versions are compiled using Convey's compiler "cnyCC" for comparable analysis. The chromosome and reads input file are placed in the 250GB host processor RAM, enabling fast file access. In the modified RMAP design, the arrays accessed by the hardware design - 1) chromosome array, 2) read-keys table array, 3) reads upper array, 4) reads lower array, 5) reads bads array, 6) score1 array, 7) score2 array, 8) chrom ID1 array, 9) chrom ID2 array and 10) strand array - are located on the coprocessor side of the memory. The coprocessor memory is organized in 16 banks, with eight memory controllers (MCs) accessing two banks each. The data arrays can typically fall on any of the eight memory controllers. However, for this design, these arrays are aligned on different memory controllers while allocating memory to help balance memory requests across all eight MCs. This results in faster memory access and helps in reducing run-time.

Input dataset generation. The chromosomes and reads files are inputs to RMAP. Both real and synthetic versions of these input datasets were used for the performance evaluation experiments. The synthetic datasets were generated using a genome generator, and a short read generator tool. The genome generator tool generated genomes of a desired length. The short reads generator tool using read width, coverage, error rate and a genome source file as input generated a set of short reads. The real datasets used for performance evaluation were chromosomes comprising the human genome downloaded from `http://hgdownload.cse.ucsc.edu/goldenPath/hg19/chromosomes/`. The short-reads file containing the reads to be mapped on to the chromosomes of the human genome was downloaded from `ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/data/NA06985/sequence_read/`.

8.2 Summary of experiments.

This section describes the experiments conducted for evaluating performance.

Throughput evaluation. In this set of experiments, two input parameters to the design were varied: 1) number of reads, and 2) number of chromosome bases. The experiments were performed for both the original RMAP and the modified RMAP. The time taken by the read mapping function of RMAP implemented in hardware was compared with the time taken by the RMAP software mapping function.

1) *Varying number of reads.* For this experiment, the genome size (i.e. number of chromosome bases) is kept fixed and the number of reads is varied. Synthetic datasets were used for this experiment. The number of reads was varied by changing the parameter 'coverage'. Coverage values of 1, 20, 40, 60, 80 and 100 were used to obtain increasing numbers of reads to map on to a 50 million base long reference genome.

2) *Varying genome size.* For this experiment, the size of the reference genome is varied and a

fixed number of reads is used. A real reference genome and short reads were used in this experiment. Chromosome21, Chromosome18, Chromosome15, Chromosome13 and Chromosome8, from the 23 chromosomes comprising the human genome were used as reference genomes. A fixed set of 41 million, 36-base wide reads were mapped on to each of these reference genomes.

Bottleneck analysis. These experiments identified and evaluated performance bottlenecks in the hardware design.

1) *Varying number of mapping sites on the reference genome.* This experiment was performed for two cases: 1) with no reads mapping, and 2) with many reads mapping on to the reference genome. For case 2, a synthetic genome having 50 million bases, and 55 million short reads (coverage of 40) were used. Case 1 used the same short reads set as case 2. However, all the bases in the genome were replaced by the base 'A', to prevent reads from mapping on to the genome. For both cases, the time taken by the hardware read mapping function was recorded. It should be noted that RMAP, using a reference genome with its bases as 'A', results in a match for all the chromosome keys being searched. The reason is that the read-key of value '0' is added as the first read-key by default to the read-keys table in RMAP. Hence, genome with all A's will have all chromsome keys as '0' and will match with the first read-key. It is determined only after scoring that there are no reads mapping on the genome. However, for this experiment, the modified RMAP did not include '0' as a read-key to the read-keys table resulting in no match for all the chromsome keys. This was done to see effect of such a dataset on the *Chroms key search unit.*

2) *Binary search.* The hardware implemented binary search involves frequently accessing system memory. The root node of the binary search tree is accessed for each chromosome key. This leads to frequent requests to the memory controller containing the root node in this implementation. This experiment was performed to observe the effect of distributing search requests evenly across all 8 memory controllers. The *Keysearcher* unit was modified to send each search request to a different address. In this experiment, a reference genome having 50 millions bases and 55 million short reads (coverage of 40) were used as input. The results

were analyzed using the report generated by the Convey performance monitor program.

3) *CAM size.* This experiment was conducted to observe the effect of varying CAM size on the time taken by the read mapping function in hardware for a fixed input dataset. CAM sizes of 50, 100, 150 and infinite size (i.e. no CAM) were used. To simulate a design having no CAM, the design was modified to send fetch requests for data associated with read IDs without checking if that read ID was in the CAM (refer Section 7.4). A real dataset of 50 million base genome and 20 million short reads were used for the experiment. For this dataset, 20% read-keys had more than 100 reads associated with them. This was useful to quickly fill up the CAM.

Metric for evaluation. The metric used for the performance evaluation experiments was wall clock time. It was measured using *get_time_of_day*(), a C timing function. The timer was placed across the read mapping function in the hardware-software code and also in the original RMAP software code to measure the execution time for both approaches. For the experiment evaluating binary search, a performance report generated by the Convey performance monitor tool was used as the metric.

CHAPTER 9. Results and Analysis

9.1 Analysis of Hardware Design

This section presents a theoretical analysis of the hardware design, to identify possible bottlenecks.

Chroms key search unit analysis. This unit performs a binary search of the chromosome key in the read-keys table. The maximum number of search requests for a chromosome key is “log n”, which is the worst case complexity for binary search. Here ‘n’ is the size of the read-keys table, which is dependent on the number of reads input to the design and also on how many of these reads form similar read-keys. The number of such chromosome key searches is dependent on the genome size. More search requests translate to more fetches from memory, increasing computation-time. Thus, the computation-time of this unit depends on the number of chromosome bases and reads.

The tables in Figure 9.1 and Figure 9.2 show the effect of the number of reads and chromosome bases on the computation-time of *Chrom key search unit*. Following gives a description of the columns in the table.

- new search request - number of clock cycles required to send new search requests for the input chromosome bases.
- old search request - number of clock cycles used for sending and processing the old search requests; depends on the number of search requests required to complete a search.
- total search memory fetches - number of clock cycles required to complete the search of all chromosome keys for a given dataset.

number of reads (n)	search complexity (log n)	new search request (clock cycles)	old search request (clock cycles)	total memory fetches (clock cycles)	end search process (clock cycles)	total clock cycles	total time (us)
1	1	1001	0	200	3	1204	8
10	4	1001	9	800	3	1813	12
100	7	1001	18	1400	3	2422	16
1000	10	1001	27	2000	3	3031	20
10000	14	1001	39	2800	3	3843	26
100000	17	1001	48	3400	3	4452	30
1000000	20	1001	57	4000	3	5061	34

Figure 9.1: Behavior of *Chroms key search unit* for 1000 base genome with varying number of reads.

number of chromosome bases	new search request (clock cycles)	old search request (clock cycles)	total memory fetches (clock cycles)	end search process (clock cycles)	total clock cycles	total time (us)
1	2	57	4000	3	4062	27
1000	1001	57	4000	3	5061	34
10000	10001	57	4000	3	14061	94
100000	100001	57	4000	3	104061	697
1000000	1000001	57	4000	3	1004061	6727
10000000	10000001	57	4000	3	10004061	67027
100000000	100000001	57	4000	3	100004061	670027

Figure 9.2: Behavior of *Chroms key search unit* for 1 million reads with varying genome size.

- end search process - number of clock cycles required to process the results once a search ends.

For the calculations performed in Figure 9.1 and Figure 9.2, worst case search complexity i.e. "log n" is considered. Memory load latency is considered to be 200 clock cycles, and clock frequency to be 150 MHz. Also, no idle cycles for the MCs and no stalls from them is considered.

In Figure 9.1, the number of chromosome bases is kept constant at 1000 and the number of reads is varied. Since the number of search requests for a chromosome key has a logarithmic relation to the number of reads, the run-time of *Chrom key search unit* increases logarithmically with the number of reads. In Figure 9.2, the number of reads is kept constant at 1 million and the number of chromosome bases is varied. The run-time *Chrom key search unit* increases linearly with the number of chromosome bases. This is due to the fact that addition of chromosome bases adds more elements to be searched in the read-keys table, resulting in more data fetches from memory.

Reads process unit analysis. In this unit, before sending any fetch request for reads' "upper", "lower", "bads" and two best scores associated with a read ID, the read ID is checked in *Content addressable memory (CAM).* If the read ID is not in CAM, then it is added to CAM. However if there are no free slots in the CAM to place the read ID, this unit is stalled until a free slot is available. A large CAM can therefore be useful in reducing the amount of stalls due to unavailability of a free slot. However, this unit can also be stalled due to 1) presence of a read ID in CAM for which data fetch request is to be issued, and 2) stalls from memory controllers 4 to 8. Hence, size of CAM might not be a possible bottleneck. Experiment 3, part of *Bottleneck analysis*, in Section 8.2 is performed to see the effect of CAM sizes on the overall design run-time.

9.2 Results and Analysis

Throughput evaluation. This section shows the results of the experiments run for throughput evaluation.

1) *Number of reads.* Figure 9.3, Figure 9.4 and Figure 9.5 show the impact on performance of varying number of reads with a fixed reference genome. Figure 9.3 illustrates that with increasing number of reads, the execution time of the read mapping function on hardware is considerably less than that on software. Figure 9.4 shows speedup of the hardware design increasing with number of reads, reaching a speedup of ∼5x. Figure 9.5 shows variation in time taken by the read mapping function in hardware with respect to number of reads. This substantiates the analysis in Section 9.1 (refer Figure 9.1) that the hardware run-time shows negligible change as the number of reads increase.
For this experiment, an estimate of the improvement in the original RMAP software performance by replacing its read mapping function with the hardware-implemented read mapping function was determined, using Equation 9.1. The results are shown in Figure 9.6.

$$T_{improved\_sw} = T_{total\_sw} - T_{sw\_read\_mapping} + T_{hw\_read\_mapping} \tag{9.1}$$

Where, $T_{improved\_sw}$ is the run-time of the software obtained by replacing the *map_reads* function time in software with that in hardware, T_{total_sw} is the run-time of the RMAP software, $T_{sw\_read\_mapping}$ is the run-time of the *map_reads* function in software, and $T_{hw\_read\_mapping}$ is the run-time of the *map_reads* function in hardware (i.e. FPGA of the coprocessor).

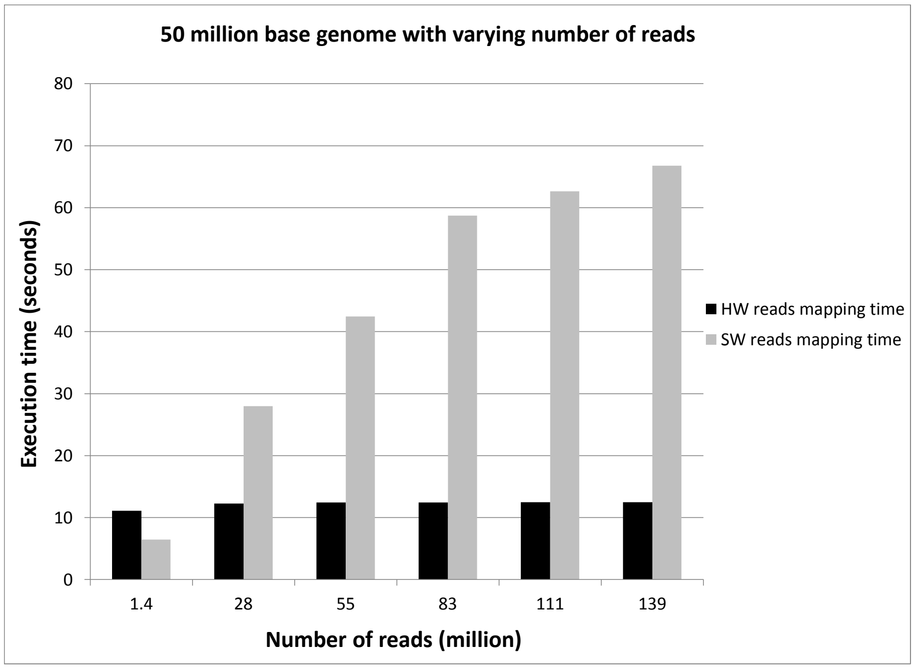


Figure 9.3: Hardware and software read-mapping function run-time comparison for 50 million base genome with varying number of reads.

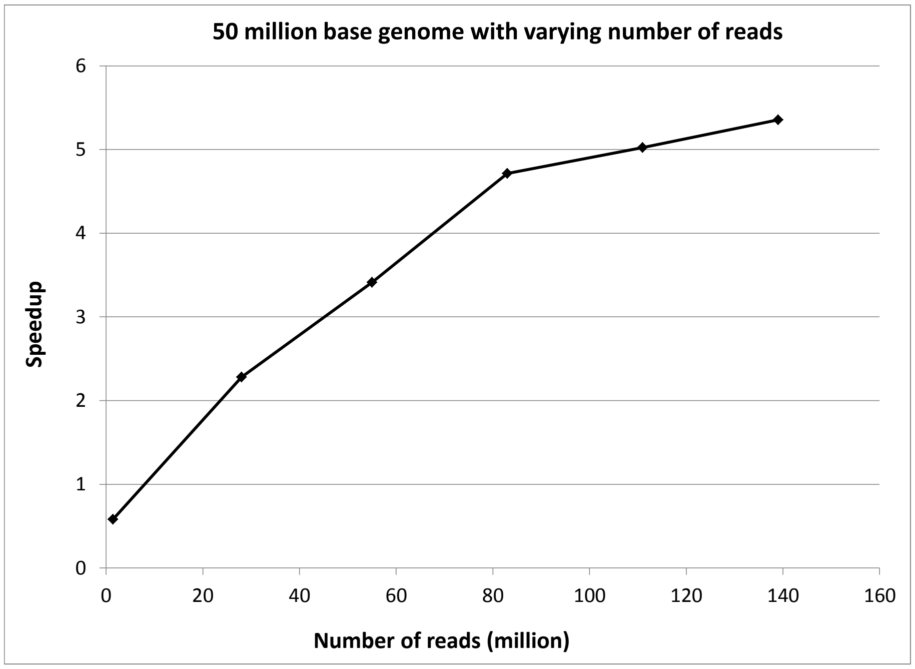


Figure 9.4: Speedup of the hardware-implemented read-mapping function for 50 million base genome with varying number of reads.

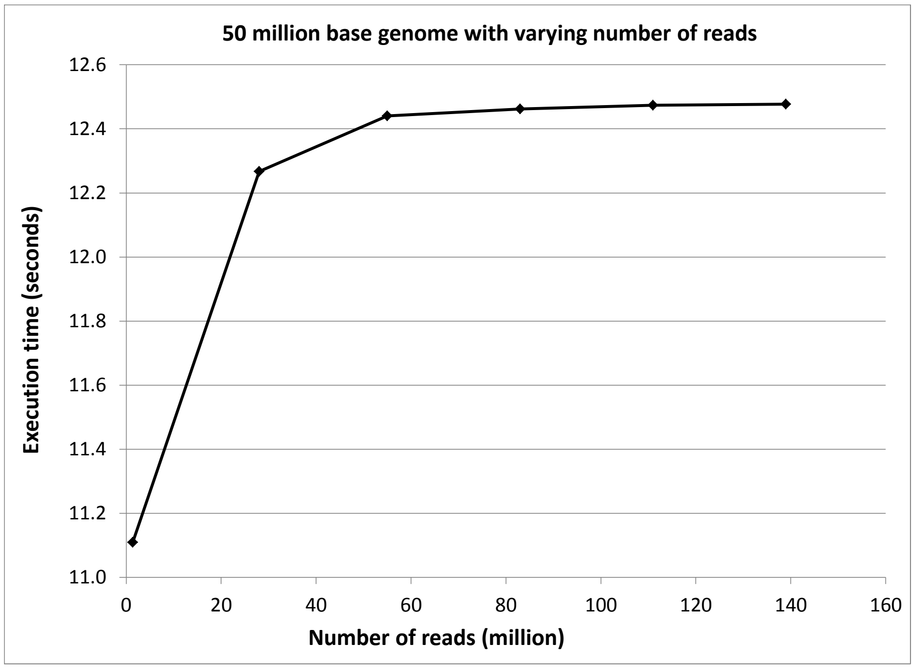


Figure 9.5: Hardware-implemented read-mapping function run-time for 50 million base genome with varying number of reads.

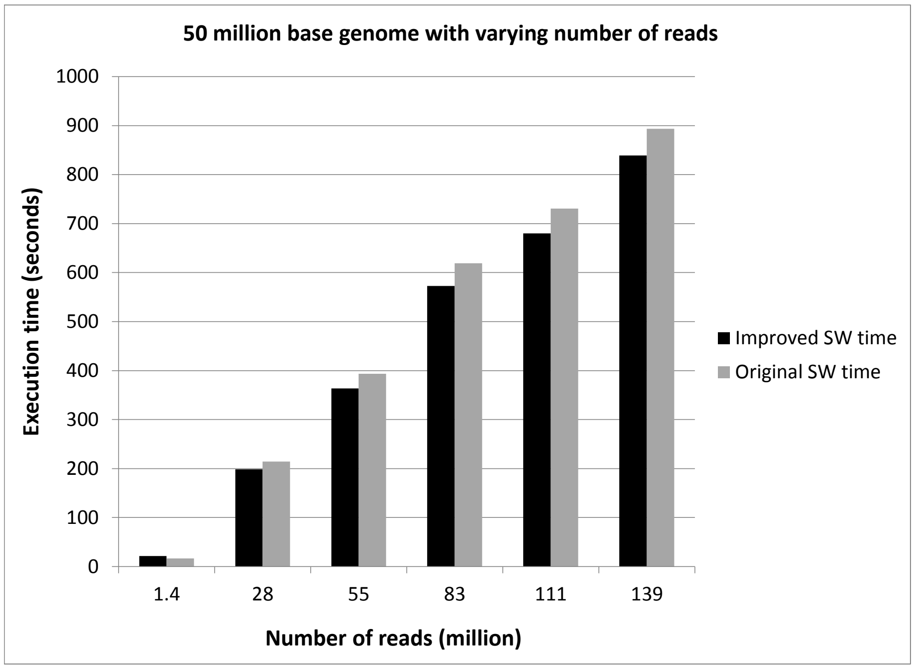


Figure 9.6: Improvement in the original RMAP software performance with the hardware read mapping function for 50 million base genome with varying number of reads.

2) *Varying genome size.* Figure 9.7, Figure 9.8 and Figure 9.9 show the impact on performance of varying genome sizes for a fixed number of reads. Figure 9.7 illustrates that with increasing genome size, the execution time of the hardware-implemented read mapping function is less than a software implementation. Figure 9.9 shows variation in time taken by the read mapping function in hardware with respect to genome size. This substantiates the analysis in Section 9.1 (refer Figure 9.2) that the genome size has significant impact on hardware run-time.

Figure 9.8 shows speedup of ∼2x with hardware design over the software read mapping function for increasing genome size. The speedup observed here is less than that observed in the experiment with varying number of reads. This is because the hardware run-time increases with genome size. Also, for the real datasets used in this experiment, many of the read-keys have more than 100 reads associated with them. This leads to filling up the CAM quickly, causing the design to wait for a free slot. The wait for a read ID to be cleared from CAM, in order that the processing of the same read ID starts again, is higher for these datasets. This further contributes towards reduced speedup.

For this experiment, an estimate of the improvement in the original RMAP software performance by replacing its read mapping function with the hardware-implemented read mapping function was determined using Equation 9.1. The results are shown in Figure 9.10.

The inference from this set of experiments is that more computationally intensive the mapping process, greater the gains obtained by implementing it on hardware.

Bottleneck analysis. This section shows the results of the experiments run for bottleneck analysis.

1) *Varying number of mapping sites on the reference genome.* Table 9.1 shows the effect on run-time for 1) no reads mapping and 2) reads mapping, on the reference genome. When there are no reads mapping on the genome, it indicates that the *Keysearcher* unit could not find a match for chromosome keys. For such a case, the number of search requests sent for each chromosome key is "log n", where 'n' is the size of the read-keys table. This leads to increased data fetches from memory, resulting in increased hardware time, as compared to

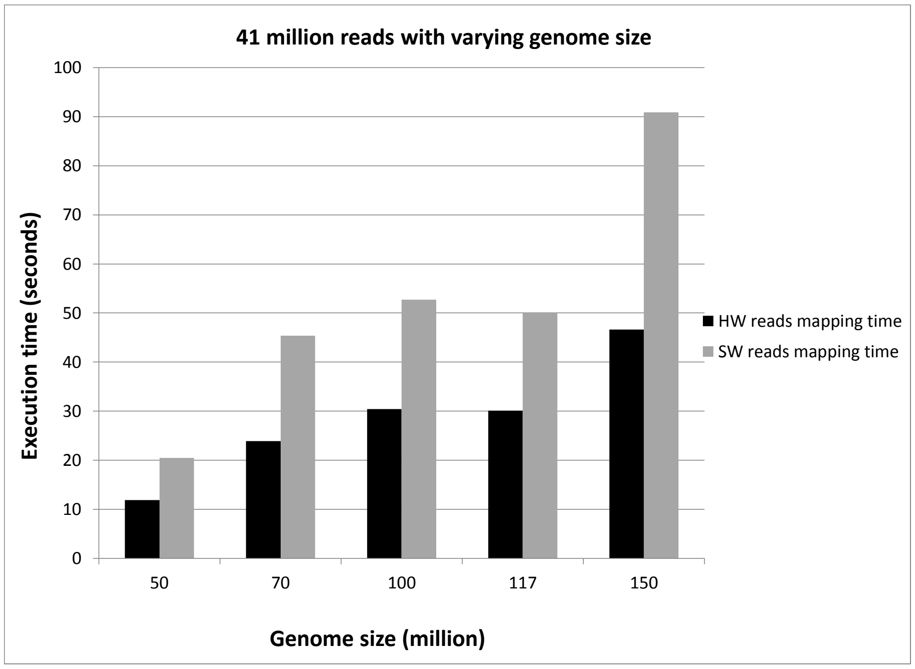


Figure 9.7: Hardware and software read-mapping function run-time comparison for 41 million reads with varying genome size.

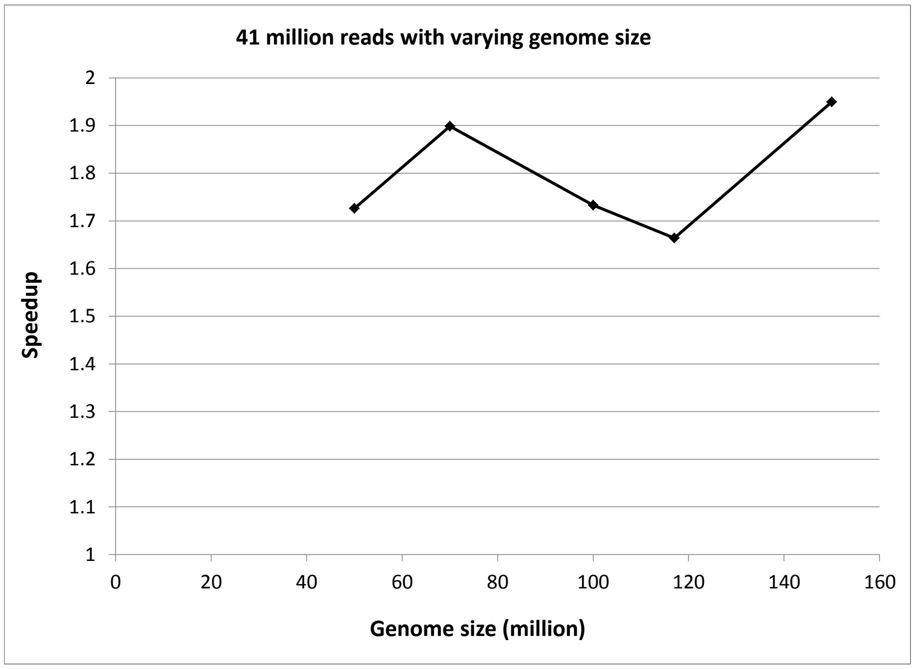


Figure 9.8: Speedup of the hardware-implemented read-mapping function for 41 million reads with varying genome size.

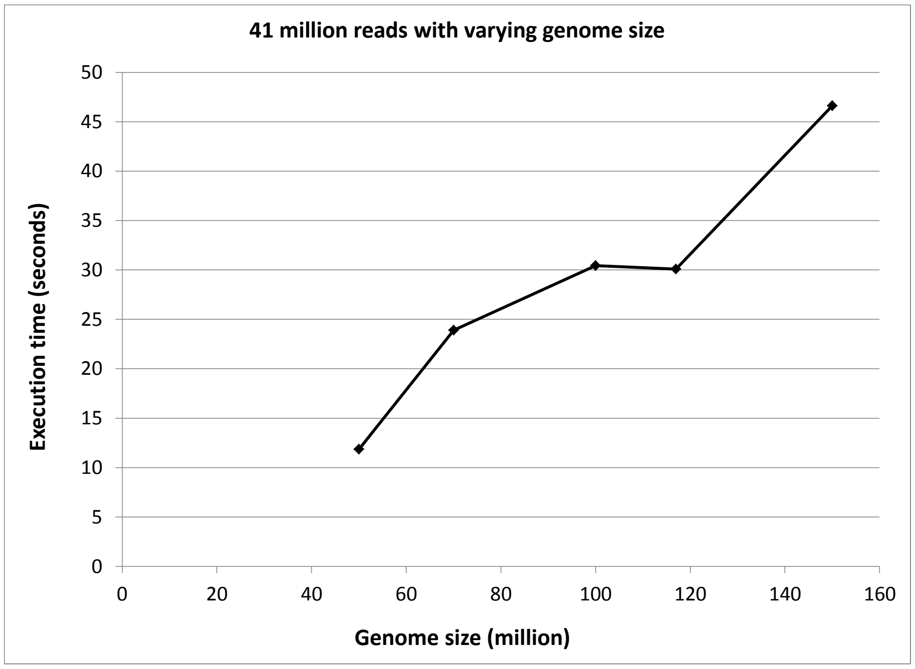


Figure 9.9: Hardware-implemented read-mapping function run-time for 41 million reads with varying genome size.

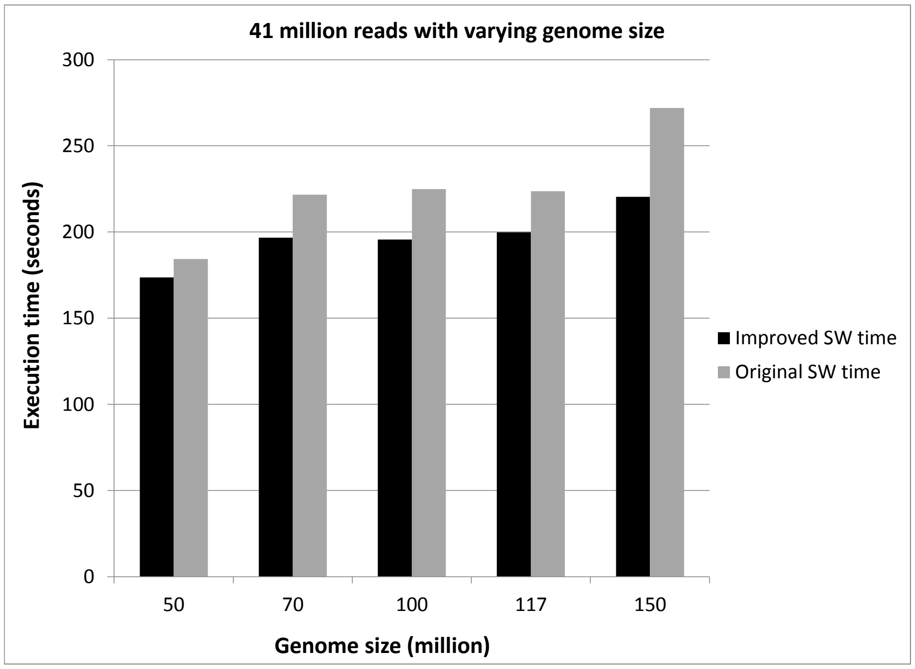


Figure 9.10: Improvement in the original RMAP software performance with the hardware read mapping function for 41 million reads with varying genome size.

Table 9.1: Effect of varying number of mapping sites on hardware design run-time for a 50 million base genome with 55 million reads.

No. of mapping sites	Time sending 'X' requests (secs)	Time sending '2X' requests (secs)
0	25.563	16.94
33221272	12.444	11.424

the case with reads mapping on the reference genome.

The *Keysearcher* sends a certain number of search requests. It is stalled until the requested search data is received back, following which more search requests are issued. Table 9.1 shows the impact of sending more search requests on run-time. Following is a description of the columns:

- Time sending 'X' requests - indicates run-time of the design when the *Keysearcher* unit issues 250 requests, and is stalled until some of the requested data is received.
- Time sending '2X' requests - indicates run-time of the design when the *Keysearcher* unit issues 500 requests, and is stalled until some of the requested data is received.

The results of this experiment substantiate the analysis of *Chrom key search unit* in Section 9.1. In case 2, where there are reads mapping on the genome, the complete hardware design runs. For case 1, *Reads process unit* and *Best maps process unit* do not function due to no match being found in the *Chrom key search unit*. However, the hardware run-time of case 1 is more than case 2. Thus, the *Chrom key search unit* could become a bottleneck depending on the search complexity and the number of search requests it can issue.

2) *Binary search.* Table 9.2 shows the convey performance report when executing the binary search without modifications to the search requests. Columns, "Stall LD" and "Stall ST" represent load and store stall cycles respectively. For this implementation, maximum stalls occur on Memory Controller 7, as seen in "Stall LD" column of Table 9.2. The reason for this is that the root node of binary search falls on this MC. Table 9.3 shows the convey performance report for the design sending search requests to the different addresses in order to align them on different memory controllers. For this implementation, no stall is observed on any MC as the search requests are distributed.

Table 9.2: Performance report for design with no modifications to search requests.

Memory controller	Loads	Stores	Stall LD	Stall ST
mc0_e	25,414,973	6,820,733	2059	0
mc0_o	156,701,006	13,641,584	2058	0
mc1_e	25,414,192	6,821,437	0	0
mc1_o	171,505,741	13,641,399	0	0
mc2_e	25,415,920	6,820,703	0	0
mc2_o	157,031,459	13,641,500	0	0
mc3_e	25,414,655	6,820,555	457	0
mc3_o	194,177,891	13,642,129	456	0
mc4_e	25,415,221	6,820,841	0	0
mc4_o	158,988,995	13,641,338	0	0
mc5_e	25,415,188	6,820,901	0	0
mc5_o	170,757,663	13,642,722	0	0
mc6_e	25,415,083	6,820,642	0	0
mc6_o	158,277,608	13,641,422	0	0
mc7_e	25,414,532	6,821,249	114,087,990	0
mc7_o	193,668,167	13,641,992	114,087,612	0

Table 9.3: Performance report for design with modifications to search requests.

Memory controller	Loads	Stores	Stall LD	Stall ST
mc0_e	781,271	0	0	0
mc0_o	159,657,183	0	0	0
mc1_e	781,269	0	0	0
mc1_o	159,638,953	0	0	0
mc2_e	781,267	8	0	0
mc2_o	159,723,163	0	0	0
mc3_e	781,271	8	0	0
mc3_o	159,703,473	0	0	0
mc4_e	781,271	0	0	0
mc4_o	159,738,353	0	0	0
mc5_e	781,274	0	0	0
mc5_o	159,673,182	0	0	0
mc6_e	781,289	0	0	0
mc6_o	159,726,429	0	0	0
mc7_e	781,288	0	0	0
mc7_o	159,673,060	0	0	0

Table 9.4: Effect of varying CAM size on hardware-implemented read mapping function run-time for a 50 million base genome with 20 million reads.

Cam Size	Time(seconds)
50	10.035
100	9.883
150	9.878
No CAM	9.801

Table 9.5: Virtex-5 LX330 resource usage for the hardware-implemented read mapping function design having CAM size of 100.

FPGA Resource	Available	Used(%)
Lookup Tables	207,360	39
Flip Flops	207,360	45
Block RAM (36 Kbit)	288	34

3) *CAM size.* In Table 9.4, it can be observed that the CAM size has little effect on hardware run-time. This substantiates the reasoning stated in analysis of *Reads process unit* in Section 9.1.

9.3 Resource usage

This section lists the usage statistics of resources such as FPGA logic, memory controllers, and global memory for the implemented design.

FPGA logic. Table 9.5 lists the percentage of the total LUTs, Flip-Flops, and BRAMs available on the Virtex 5 LX330, used for the hardware design. A significant portion of the resources is occupied by Convey's memory controller interface and hardware-software interface.

Memory controller. The Convey HC-1 system has 8 memory controllers, with two ports each. The read mapping function on the coprocessor uses 6 memory controllers with both ports, effectively using 12 memory controllers. Of these, 3 are used for storing best maps data for each read while the remaining are used for fetching data from memory. The memory controllers used for stores use the "write complete interface".

Global memory. The host processor and coprocessor are associated with 256 GB and 64 GB of the global memory respectively. The memory layout is such that, due to physical proximity, it provides the host processor and coprocessor fastest access to their respective memory spaces.

Table 9.6: Memory usage of the RMAP hardware-software co-design for 50 million base genome with varying number of reads.

Reads (million)	Host Memory(GB)	Coprocessor Memory(GB)
1.4	0.3	0.18
28	5.8	2.7
55	11.6	5.3
83	17.4	7.9
111	23.2	10.6
139	29	13.2

Table 9.7: Memory usage of the RMAP hardware-software co-design for 41 million reads with varying genome size.

Genome (million)	Host Memory(GB)	Coprocessor Memory(GB)
50	11.4	5
70	11.4	5.1
100	11.4	5.4
117	11.4	5.6
150	11.4	5.9

For the hardware-software RMAP design on the HC-1 system, data allocated on the host processor memory is solely dependent on the number of reads. However, coprocessor memory usage is dependent on both number of reads and genome size. The read mapping function in hardware uses only coprocessor memory. Tables 9.6 and 9.7 show the amount of coprocessor memory used while the mapping function is running in hardware, indicated in the "Coprocessor Memory" column.

Table 9.6 lists the memory usage for the experiment where the number of reads are varied with the genome size constant, described in Section 8.2. The increase in coprocessor memory usage with increasing number of reads can be attributed to several reasons. Firstly, each read is associated with 8 pieces of data: 1) upper, 2) lower, 3) bads, 4) score1, 5) score2, 6) chrom ID1, 7) chrom ID2, and 8) strand of the chromosome. Each of these occupy 8-bytes. Next, the memory allocated to read-keys table is also dependent on the number of reads. Table 9.7 lists the memory usage for the experiment where the genome size is varied with the number of reads constant, described in Section 8.2. The small increase in coprocessor memory usage is due to increase in genome size as the genome array is allocated on the coprocessor memory.

9.4 Pending Issues and Concerns

This section puts forth some open issues pertaining to the hardware design.

Performance runs for large datasets. The datasets used here for the performance experiments are relatively small compared to real world datasets that RMAP is typically used to run. The design implemented on the coprocessor has an unresolved bug which prevents it from running on datasets larger than the ones shown. Based on the results of the throughput evaluation experiments, a speedup of at least 2x is expected for large datasets.

Binary search. The experiments performed for identifying bottlenecks in the binary search process, described in Section 9.2, determines the effect of distributing search requests evenly across all 8 memory controllers. The design modified for this purpose results in very few matches in the *Chroms key search unit* for the dataset used. This dataset, otherwise, results in matches for 60% of the chromosome keys. This prevents the *Reads process unit* and the *Best maps process unit* from being called into action, making them almost redundant for this case. Hence, this experiment should be conducted with a design which, along with not causing traffic on a particular MC, also results in significant matches during the chromosome key search process. This will enable the *Reads process unit* and the *Best maps process unit* to function, allowing the analysis to incorporate their effect.

Bandwidth utilization. The coprocessor design does not operate at the full bandwidth of 20 GB/s, available to an AE (FPGA) of the Convey HC-1 system. In order to fully utilize this, the design needs to make one request per clock cycle to each MC port (odd and even). There are mainly three reasons which prevent the design from achieving full bandwidth utilization:

1) MC Stalls - These arise because of the MCs not being able to service requests at the rate at which they are sent from the design.

2) Idle cycles - These correspond to the cycles during which the MCs are idle (i.e. when they are not servicing requests). One of the possible reasons for this is that the design does not allow data requests larger than the size of the fifos, which store the requested data, until some of the data is read from the fifos, resulting in idle cyles. The other possible reason is the *Reads process unit* taking 3-4 cycles to send each fetch request for the data associated

with the read IDs, because of its finite state machine (FSM) based design. This contributes towards idle cycles, during which fetch requests are not being sent for the read's data.

3) Unused crossbar MCs - Full bandwidth utilization requires the use of all (16) crossbar MCs available to the AE. However, the design uses only a subset of the MCs, thus impacting the percentage of the peak bandwidth utilized.

CHAPTER 10. Conclusion and Future Work

This chapter concludes the thesis and discusses some areas of future work in order to further accelerate RMAP.

10.1 Conclusion

This work has described a technique to accelerate the short-read mapping function of the RMAP short read mapping tool. As a first step, RMAP was profiled to determine the execution time of each of its functions. Based on the results of this profiling, short-read mapping was found to be the most computatonally intensive function, and hence was chosen for acceleration.

To achieve the acceleration, the read mapping function was implemented on the reconfigurable hardware (FPGA) of a Convey HC-1 system. The hybrid feature of the HC-1 platform provided a highly abstract way of designing a hardware-software co-design solution for RMAP, with the read mapping function running on the reconfigurable fabric of the coprocessor, and the remaining RMAP functions running on the host processor. RMAP was modified to separate the read mapping function for porting, and to align it with the HC-1's hybrid architecture. The hardware was designed in the form of a pipelined architecture implementing the mapping function.

Experiments were performed to observe the speedup of the read mapping function implemented on the custom hardware architecture, and for finding potential bottlenecks in the hardware design. For throughput evaluation, data was collected using different datasets by 1) varying the number of reads for a fixed genome size, and 2) varying genome size for a fixed number of reads. The first experiment showed a speedup of ∼5x for 139 million reads (the largest reads dataset used in this experiment) with a 50 million base genome, as compared to a software implementation of the read mapping function. The second experiment showed a speedup of ∼2x for 150 million base genome (the largest genome size used in this experiment) with 41 million reads. For bottleneck analysis, experiments were run to determine the impact of the chromosome key search process and the size of the content addressable memory unit used, on coprocessor design run-time. It was determined that the search process could be a bottleneck affecting performance. However, variation in memory unit size proved to have no significant impact on performance.

10.2 Future Work

Three possible directions for future work to further increase the performance of the read mapping function implemented in hardware are suggested in this section.

1) **Splitting the genome across four FPGAs.** This work makes use of only one FPGA of the HC-1 system out of the four that are available. For further speedup, all the four FPGAs, also called as Application Engines (AEs), could be used. In order to achieve this, the reference genome, input to the design, could be divided across the four AEs. Each AE would perform the same function of 1) processing the genome, 2) searching the chromosome key in the read-keys table, 3) scoring the matches, 4) determining the best maps for each read based on the score, and 5) storing the best maps result. The read-keys table and the *Content addressable memory unit (CAM)* would be common to all the four AEs. Each AE would place the read IDs, that are being processed, in the CAM. If there is a contention in writing to the CAM, priority would be given to the AE which streams the first division of genome, and so on. Memory load/store contentions would be taken care of by the HC-1 “coprocessor memory ordering” feature.

2) **Binary search data buffering.** Based on the results of Section 9.2, the binary search process in the *Chroms key search unit* could be a bottleneck. Performing the search for each chromosome key in the hardware design involves accessing the same middle elements multiple times, with the root node of the binary search tree (i.e. the read-keys table) always being accessed. The workaround for this could be, 1) a data buffer or cache in the AE for storing the inital N middle elements, or 2) replicating the first N levels of the binary search

tree in the AE. This could eliminate the need for fetching the middle element from memory for each chromosome key, which could be in millions to billions in number depending on the reference genome size. For workaround 1), a BRAM could be used as a cache to save these middle elements. An address-mapping function could be used to map the binary search address on to the BRAM address. When a search for a chromosome key is under process, the addresses involved can be checked to determine if they map with the BRAM addresses, and then the the data in the BRAM can be used. For workaround 2), the initial stages of the search for a chromosome key would be on the AE. The later stages, requiring access to the read-keys table outside the first N levels, would be transferred to the coprocessor memory.

3) **Software optimization.** In this work, RMAP was modified for implementation on the HC-1 hybrid platform. The functions of RMAP that run on the host processor are not optimized. Also, the host processor operates on some of the arrays stored on the coprocessor side of memory. Hence, the total run-time of the RMAP implementation on the HC-1 system turns out to be more than the original optimized version of RMAP. The software functions running on the host processor with this hybrid design could be optimized to take equal or lesser amount of time than the original RMAP's software functions. Thus, speedup for the overall RMAP hardware-software co-design could be targeted.

BIBLIOGRAPHY

1000 Genomes (2010). 1000 genomes: A deep catalog of human genetic variation. http://www.1000genomes.org/data.

Abouelhoda, M. I., Kurtz, S., and Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *J. of Discrete Algorithms*, 2(1):53–86.

Amazon EC2 (2012). Amazon elastic compute cloud. http://aws.amazon.com/ec2/.

Ashwin, M. A., Zhang, L., and Feng, W. (2010). Gpu-rmap: Accelerating short-read mapping on graphics processors. In *Proceedings of the 2010 13th IEEE International Conference on Computational Science and Engineering*, pages 168–175.

Bioinformatics I (2008). Sequence analysis and phylogenetics. http://www.master-bioinformatik.at/curriculum/BioInf_I_Notes.pdf.

BLAST (2012). Blast. http://blast.ncbi.nlm.nih.gov/Blast.cgi.

Boukerche, A., Correa, J., de Melo, A., Jacobi, R., and Rocha, A. (2007). Reconfigurable architecture for biological sequence comparison in reduced memory space. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8.

Bourne, P. E. and Shindyalov, I. N. (1998). Protein structure alignment by incremental combinatorial extension (ce) of the optimal path. *Protein Engineering*, 9:739–747.

Bourne, P. E. and Shindyalov, I. N. (2003). Structure comparison and alignment. *Methods of Biochemical Analysis*, 44:321–337.

Bowtie (2012). Bowtie: An ultrafast memory-efficient short read aligner. http://bowtie-bio.sourceforge.net/index.shtml.

Bravo, H., Pihur, V., McCall, M., Irizarry, R. A., and Leek, J. T. (2012). Gene expression anti-profiles as a basis for accurate universal cancer signatures. *BMC Bioinformatics*, 13.

Burrows, M. and Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm. Technical report, DEC Systems Research Centre, Palo Alto, California, USA.

CeBiTec (2012). Bielefeld university center for biotechnology uses convey hybrid-core system to accelerate research. http://www.conveycomputer.com/files/1313/5075/6265/ Bielefeld-University-CeBiTec-Biotechnology-Genetic-Engineering-DNA-Analysis-CONV-12-028- pdf.

Chen, Y., Schmidt, B., and Maskell, D. L. (2009). A reconfigurable bloom filter architecture for blastn. In *Proceedings of the 22nd International Conference on Architecture of Computing Systems*, pages 40–49.

Chou, P. Y. and Fasman, G. D. (1974). Prediction of protein conformation. *Biochemistry*, 13:222–245.

Close, T. (2011). Cowpea: Snp-based diversity analyses and genetic improvement for marignal environments. http://a-c-s.confex.com/crops/2011am/webprogram/Paper65643.html.

Convey (2012). Convey reference manual version 1.1. http://www.conveysupport.com/alldocs/ConveyReferenceManual.pdf.

D, D. W., Emde, A. K., Rausch, T., Dring, A., and Reinert, K. (2009). Razers–fast read mapping with sensitivity control. *Genome Research*, 19(9):1646–1654.

Ding, Y. and Lawrence, C. E. (2003). A statistical sampling algorithm for RNA secondary structure prediction. *Nucleic Acids Res.*, 31(24):7280–7301.

Dohm, J. C., Lottaz, C., and Himmelbauer, T. B. H. (2007). Sharcgs, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. *Genome Research*, 17(11):1697–1706.

Driga, A., Lu, P., Schaeffer, J., Szafron, D., Charter, K., and Parsons, I. (2003). Fastlsa: a fast, linear-space, parallel and sequential algorithm for sequence alignment. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 48–57.

Eddy, S. R. (1995). Multiple alignment using hidden markov models. In *International Conference on Intelligent Systems for Molecular Biology*, volume 3, pages 114–120.

Eddy, S. R. (2004). What is dynamic programming? *Nature Biotechnology*, 22:909–910.

Ewing, B., Hillier, L., Wendl, M. C., and Green, P. (1998). Base-Calling of automated sequencer traces using phred. i. accuracy assessment. *Genome Research*, 8(3):175–185.

Gálvez, S., Díaz, D., Hernández, P., Esteban, F. J., Caballero, J. A., and Dorado, G. (2010). Next-generation bioinformatics: using many-core processor architecture to develop a web service for sequence alignment. *Bioinformatics*, 26(5):683–686.

Garnier, J., Gibrat, J. F., and Robson, B. (1996). Gor method for predicting protein secondary structure from amino acid sequence. *Methods in Enzymology*, 266:540–553.

Hadoop (2012). Apache hadoop. http://hadoop.apache.org/.

Hanash, S. (2003). Disease proteomics. *Nature*, 422:226–232.

Helaers, R. and Milinkovitch, M. (2010). Metapiga v2.0: maximum likelihood large phylogeny estimation using the metapopulation genetic algorithm and other stochastic heuristics. *BMC Bioinformatics*, 11(1):379.

Hoffmann, S., Otto, C., Kurtz, S., Sharma, C., Khaitovich, P., Vogel, J., Stadler, P., and Hackermuller, J. (2009). Fast mapping of short sequences with mismatches, insertions and deletions using index structures. *Plos Computational Biology*, 5(9).

Hoksza, D. and Svozil, D. (2012). Efficient rna pairwise structure comparison by setter method. *Bioinformatics*, 28(14):1858–1864.

Holm, L. and Sander, C. (1993). Protein structure comparison by alignment of distance matrices. *Journal of molecular biology*, 233:123–128.

Holub, J. and Melichar, B. (1999). Implementation of nondeterministic finite automata for approximate pattern matching. In *Revised Papers from the Third International Workshop on Automata Implementation*, pages 92–99.

Imming, P., Sinning, C., and Meyer, A. (2003). Drugs, their targets and the nature and number of drug targets. *Nat Rev Drug Discov*, 5.

Jones, N. C. and Pevzner, P. A. (2004). An introduction to bioinformatics algorithms.

Jung, S. and Main, D. (2011). Tools for comparative genomics in crop science. http://a-c-s.confex.com/crops/2011am/webprogram/Paper65641.html.

K, K. P., Stenzel, U., Dannemann, M., Green, R. E., Lachmann, M., and Kelso, J. (2008). Patman: rapid alignment of short sequences to large databases. *Bioinformatics*, 24(13):1530–1532.

Kasap, S., Benkrid, K., and Liu, Y. (2008a). Design and implementation of an fpga-based core for gapped blast sequence alignment with the two-hit method. *Engineering Letters*, 16(3):443–452.

Kasap, S., Benkrid, K., and Liu, Y. (2008b). High performance fpga-based core for blast sequence alignment with the two-hit method. In *BioInformatics and BioEngineering, 2008. BIBE 2008. 8th IEEE International Conference on*, pages 1–7.

Krawetz, S. and Womble, D. D. (2003). Introduction to bioinformatics: A theoretical and practical approach. http://www.springer.com/life+sciences/systems+biology+and+bioinformatics/book/978-1-58829-064-9.

Kung, H. T. and Leiserson, C. E. (1978). Systolic arrays (for vlsi). Technical report, Carnegie-Mellon University, Department of Computer Science.

Labeda, D., Goodfellow, M., Brown, R., Ward, A., Lanoot, B., Vanncanneyt, M., Swings, J., Kim, S.-B., Liu, Z., Chun, J., Tamura, T., Oguchi, A., Kikuchi, T., Kikuchi, H., Nishii, T., Tsuji, K., Yamaguchi, Y., Tase, A., Takahashi, M., Sakane, T., Suzuki, K., and Hatano, K.

(2012). Phylogenetic study of the species within the family streptomycetaceae. *Antonie van Leeuwenhoek*, 101:73–104.

Langmead, B., Schatz, M. C., Lin, J., Pop, M., and Salzberg, S. L. (2009a). Human snps from short reads in hours using cloud computing. http://www.cbcb.umd.edu/~mschatz/Posters/Crossbow_WABI_Sept2009.pdf.

Langmead, B., Trapnell, C., Pop, M., and Salzberg, S. L. (2009b). Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Ultrafast and memory-efficient alignment of short DNA sequences to the human genome*, 10.

Ley, T. J., Mardis, E. R., Ding, L., Fulton, B., McLellan, M. D., Chen, K., Dooling, D., Dunford-Shore, B. H., McGrath, S., Hickenbotham, M., Cook, L., Abbott, R., Larson, D. E., Koboldt, D. C., Pohl, C., Smith, S., Hawkins, A., Abbott, S., Locke, D., Hillier, L. W., Miner, T., Fulton, L., Magrini, V., Wylie, T., Glasscock, J., Conyers, J., Sander, N., Shi, X., Osborne, J. R., Minx, P., Gordon, D., Chinwalla, A., Zhao, Y., Ries, R. E., Payton, J. E., Westervelt, P., Tomasson, M. H., Watson, M., Baty, J., Ivanovich, J., Heath, S., Shannon, W. D., Nagarajan, R., Walter, M. J., Link, D. C., Graubert, T. A., DiPersio, J. F., and Wilson, R. K. (2008). Dna sequencing of a cytogenetically normal acute myeloid leukaemia genome. *Nature*, 456:66–72.

Li, H. and Durbin, R. (2009). Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 25(14):1754–1760.

Li, H., Ruan, J., and Durbin, R. (2008a). Mapping short dna sequencing reads and calling variants using mapping quality scores. *Genome Research*, 18(11):1851–1858.

Li, R., Li, Y., Fang, X., Yang, H., Wang, J., Kristiansen, K., and Wang, J. (2009). Snp detection for massively parallel whole-genome resequencing. *Genome Research*, 19(6):1124–1132.

Li, R., Li, Y., Kristiansen, K., and Wang, J. (2008b). Soap: short oligonucleotide alignment program. *Bioinformatics*, 1:713–714.

Lin, H., Zhang, Z., Zhang, M. Q., Ma, B., and Li, M. (2008). Zoom! zillions of oligos mapped. *Bioinformatics*, 24(21):2431–2437.

Liu, C.-M., Wong, T., Wu, E., Luo, R., Yiu, S.-M., Li, Y., Wang, B., Yu, C., Chu, X., Zhao, K., Li, R., and Lam, T.-W. (2012). Soap3: Ultra-fast gpu-based parallel alignment tool for short reads. *Bioinformatics*.

Mardis, E. R. (2008). Next-generation dna sequencing methods. *Annual Review of Genomics and Human Genetics*, 9:387–402.

Margulies, M., Egholm, M., Altman, W. E., Attiya, S., Bader, J. S., Bemben, L. A., Berka, J., Braverman, M. S., ju Chen, Y., Chen, Z., Dewell, B., Du, L., Fierro, J. M., Gomes, X. V., Goodwin, B. C., He, W., Helgesen, S., Ho, C. H., Irzyk, G. P., J, S. C., I, M. L., Jarvie, T. P., Jirage, K. B., bum Kim, J., Knight, J. R., Lanza, R., Leamon, J. H., Lefkowitz, S. M., Lei, M., Li, J., L, K., Lu, H., Makhijani, V. B., Mcdade, K. E., Mckenna, M. P., Ronan, T., Roth, G. T., Sarkis, G. J., Simons, J. F., Simpson, J. W., Srinivasan, M., Tartaro, K. R., Tomasz, E., Vogt, K. A., A, G., Wang, S. H., Wang, Y., Weiner, M. P., Yu, P., F, R., and Rothberg, J. M. (2005). Genome sequencing in open microfabricated high-density picoliter reactors. *Nature*, 437:376–380.

Mochida, K. and Shinozaki, K. (2010). Genomics and bioinformatics resources for crop improvement. *Physiol*, 51:497–523.

Mount, D. (2004). Using the basic local alignment search tool (blast). http://cshprotocols.cshlp.org/content/2007/7/pdb.top17.full.

Myers, E. W., Sutton, G. G., Delcher, A. L., Dew, I. M., Fasulo, D. P., Flanigan, M. J., Kravitz, S. A., Mobarry, C. M., Reinert, K. H. J., Remington, K. A., Anson, E. L., Bolanos, A. A., Zhang, Q., Zheng, X., Rubin, G. M., Adams, M. D., and Venter, J. C. (2000). A whole-genome assembly of drosophila. *Science*, 287(5461):2196–2204.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453.

Nelson, C., Townsend, K., Rao, B. S., Jones, P. H., and Zambreno, J. (2012). Shepard: A fast exact match short read aligner. In *MEMOCODE*, pages 91–94.

NGS (2012). An introduction to next-generation sequencing technology. http://www.illumina.com/Documents/products/Illumina_Sequencing_Introduction.pdf.

Olson, C., Kim, M., Clauson, C., Kogon, B., Ebeling, C., Hauck, S., and Ruzzo, W. (2012). Hardware acceleration of short read mapping. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 161 –168.

Paterson, A. H., Bomanc, R. K., Brownb, S. M., Cheeb, P. W., Gannawayc, J. R., Gingled, A. R., Mayb, O. L., and Smithe, C. W. (2004). Reducing the genetic vulnerability of cotton. *Crop Sci*, 44:1900–1901.

Pevzner, P. and Waterman, M. (1995). Multiple filtration and approximate pattern matching. *Algorithmica*, 13:135–154.

Quijadaa, P. A., Udallb, J. A., Polewiczc, H., Vogelzanga, R. D., and Osborn, T. C. (2004a). Genomics and plant breeding. *Crop Sci*, 44:1893–1893.

Quijadaa, P. A., Udallb, J. A., Polewiczc, H., Vogelzanga, R. D., and Osborn, T. C. (2004b). Phenotypic effects of introgressing french winter germplasm into hybrid spring canola. *Crop Sci*, 44:1982–1989.

Rumble, S. M., Lacroute, P., Dalca, A. V., Fiume, M., Sidow, A., and Brudno, M. (2009). Shrimp: Accurate mapping of short color-space reads. *PLoS Comput Biol*, 5(5):e1000386.

Sahoo, B. and Padhy, S. (2009). A reconfigurable accelerator for parallel longest common protein subsequence algorithm. In *Advance Computing Conference, 2009. IACC 2009. IEEE International*, pages 260–265.

Sanger, F. and Coulson, A. (1975). A rapid method for determining sequences in dna by primed synthesis with dna polymerase. *Journal of Molecular Biology*, 94(3):441–448.

Schatz, M. C. (2009). Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics*, 25:1363–1369.

Schatz, M. C., Trapnell, C., Delcher, A. L., and Varshney, A. (2007). High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 10(8):474.

Schuster, S. C. (2008). Next-generation sequencing transforms today's biology. *Nature Methods*, 5:16–18.

Sequence and Genome Analysis (2004). Bioinformatics: Sequence and genome analysis, second edition. http://www.cshlpress.com/default.tpl?cart=135529681060076461&fromlink=T&linkaction=full&linksortby=oop_title&--eqSKUdatarq=466.

Shen, H., Yin, Y., Chen, F., Xu, Y., and Dixon, R. A. (2009). A bioinformatic analysis of nac genes for plant cell wall development in relation to lignocellulosic bioenergy production. *Bioenerg Res*, 2:217–232.

Simpson, J. T., Wong, K., Jackman, S. D., Schein, J. E., Jones, S. J., and Birol, I. (2009). Abyss: a parallel assembler for short read sequence data. *Genome Research*, 19(6):1117–1123.

Smith, A., Xuan, Z., and Zhang, M. (2008). Using quality scores and longer reads improves accuracy of solexa read mapping. *BMC Bioinformatics*, 9(1):128.

Smith, A. D., Chung, W.-Y., Hodges, E., Kendall, J., Hannon, G. J., Hicks, J., Xuan, Z., and Zhang, M. Q. (2009). Updates to the rmap short-read mapping software. *Bioinformatics*, 25(21):2841–2842.

Swofford, D. L. and Begle, D. P. (1993). Phylogenetic analysis using parsimony (paup) version 3.1. http://paup.csit.fsu.edu/Paup_Doc_31.pdf.

Trapnell, C. and Salzberg, S. L. (2009). How to map billions of short reads onto genomes. *Nature Biotechnology*, 27:455–457.

Vij, S., Gupta, V., Kumar, D., Vydianathan, R., Raghuvanshi, S., Khurana, P., Khurana, J. P., and Tyagi, A. K. (2006). Decoding the rice genome. *Bioessays*, 28:421–432.

W. E. Stein, J. (1987). Phylogenetic analysis and fossil plants. *Review of Palaeobotany and Palynology*, 50:31–61.

Warren, R. L., Sutton, G. G., Jones, S. J. M., and Holt, R. A. (2007). Assembling millions of short dna sequences using ssake. *Bioinformatics*, 23(4):500–501.

Xilinx (2012). Fpga. http://www.xilinx.com/fpga/index.htm.

XtremeData (2006). Xtremedata, inc., xd1000 development system. http://www.xtremedata.com/.

Yang, Z. (2007). Paml 4: Phylogenetic analysis by maximum likelihood. *Molecular Biology and Evolution*, 24(8):1586–1591.

Yang, Z. and Rannala, B. (1997). Bayesian phylogenetic inference using dna sequences: a markov chain monte carlo method. *Molecular Biology and Evolution*, 14(7):717–724.

Zerbino, D. R. and Birney, E. (2008). Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, 18(5):821–829.

Zhang, P., Tan, G., and Gao, G. R. (2007). Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform. In *Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications: held in conjunction with SC07*, pages 39–48.