View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Ulster University's Research Portal

Hardware acceleration of *de novo* genome assembly

B. Sharat Chandra Varma*

Department of Electrical and Electronic Engineering,
The University of Hong Kong, Hong Kong
Email: varma@hku.hk
*Corresponding author

Kolin Paul and M. Balakrishnan

Department of Computer Science and Engineering,
Indian Institute of Technology Delhi,
110016, India
Email: kolin@cse.iitd.ac.in
Email: mbala@cse.iitd.ac.in

Dominique Lavenier

IRISA/INRIA,
35042 Rennes, France
Email: lavenier@irisa.fr

Abstract: The cost of genome assembly has gone down drastically with the advent of next generation sequencing technologies. These new sequencing technologies produce large amounts of DNA fragments. Software programs are used to construct the genome from these DNA fragments. The assembly programs take significant amount of time to execute. To reduce the execution time, these programs are being parallelised to take advantage of many cores available in present day processor chips. Further, hardware accelerators have been developed which when used along with processors speed up the execution. Velvet is a commonly used software for *de novo* assembly. We propose a novel method to reduce the overall time of assembly by using FPGAs. In this method, we perform pre-processing of these short reads on FPGAs and process the output using Velvet to reduce the overall time for assembly. We show that using our technique we can get significant speed-ups.

Keywords: FPGA; acceleration; next generation sequencing assembly; bioinformatics.

Reference to this paper should be made as follows: Varma, B.S.C., Paul, K., Balakrishnan, M. and Lavenier, D. (xxxx) 'Hardware acceleration of *de novo* genome assembly', *Int. J. Embedded Systems*, Vol. X, No. Y, pp.xxx–xxx.

Biographical notes: B. Sharat Chandra Varma is a Postdoctoral Research Fellow at the Department of Electronic and Electric Engineering, The University of Hong Kong. He completed his PhD from Indian Institute of Technology Delhi, India. He holds a Masters in VLSICAD from Manipal University and Bachelors in Electronics and Communications Engineering from Visvesvaraya Technological University. He previously worked at QuickLogic India Pvt. Ltd. where he developed EDA tools for QuickLogic FPGAs. His research interests include hardware-software co-design, FPGA architecture, FPGA-based acceleration and computer architecture.

Kolin Paul is an Associate Professor in the Department of Computer Science and Engineering at IIT Delhi India. He received his BE in Electronics and Telecommunication Engineering from NIT Silchar in 1992 and PhD in Computer Science in 2002 from BE College (DU), Shibpore. During 2002 to 2003, he did his postdoctoral studies at Colorado State University, Fort Collins, USA. He has previously worked at IBM Software Labs. His last appointment was as a Lecturer in the Department of Computer Science at the University of Bristol, UK. His research interests are in understanding high performance architectures and compilation systems. In particular, he works in the area of adaptive/reconfigurable computing trying to understand its use and implications in embedded systems.

M. Balakrishnan is a Professor in the Department of Computer Science and Engineering at I.I.T. Delhi. He received his Undergraduate degree from BITS Pilani in 1977 and PhD degree from IIT Delhi in 1985. For the last 28 years, he is involved in teaching and research in the areas of digital systems design, EDA and embedded systems. He has published nearly 75 papers in

Copyright © 20XX Inderscience Enterprises Ltd.

leading journals and conferences. Further, he has held visiting positions in universities in Germany, USA and Canada. He has been the HOD of CSE, Dean of Post Graduate Studies and Research and Deputy Director (Faculty) at IIT Delhi. His research interests are in system level synthesis and design exploration tools, hardware accelerators and embedded systems. His other major interests are related to development of affordable mobility and education aids for the visually impaired as well as higher education Institutions initiatives for supporting research.

Dominique Lavenier is a senior CNRS (French National Center for Scientific Research) researcher. He is currently leading the GenScale bioinformatics team at IRISA/INRIA, Rennes. He was the recipient of the 'Mdaille de Bronze' of the French Council for Research CNRS in 1992, and received the French Cray Prize in 1996 in algorithm, architecture, and micro-electronic. From August 1999 to August 2000, he has been working in the Nonproliferation and International Security Division at the Los Alamos National Laboratory, NM, USA. His research interests include HPC, parallel architecture, GPU computing bioinformatics, structural biology and the management of genomic data coming from next generation sequencing technologies.

This paper is a revised and expanded version of a paper entitled 'FAssem: FPGA based acceleration of de novo genome assembly' presented at the IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Seattle, USA, 28–30 April 2013.

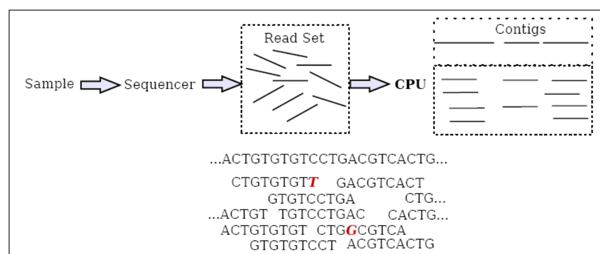
1 Introduction

The basic building block of all organisms is the *cell*. All the cells (irrespective of the size of organism) have a nucleus which carries a genetic material known as *deoxyribonucleic acid* (DNA). DNA holds the hereditary information and is responsible for the controlling and functioning of the organism. DNA is made up of four bases: adenine (A), guanine (G), cytosine (C), and thymine (T). Adenosine (A) pairs with thymine (T), and cytosine (C) pairs with guanine (G) forming *base pairs* (bp). This pairing is due to weak hydrogen bonding and is the basis for DNA replication. A segment of a DNA molecule can be written using first letter of the bases it contains (e.g., …TACGTAG…). *Genes*, which are made up of DNA store information needed for making proteins useful for the life of the organism. The complete set of all genes along with non-coding DNA in an organism is called a *genome*. The study of genomes of various organisms is known as genomics. It has a lot of applications in medicine, biotechnology, anthropology, forensics and synthetic biology. Also, comparative study of different genomes is helpful for evolutionary studies. Increasingly, genomics is also being used to study the contribution of genes in many diseases and is aiding in the development of personalised drugs. Hence, genome construction is very important, which helps considerably in the study of various biological processes in an organism.

DNA sequencing technology helps in generating the data needed for construction of genomes. Recently, next generation sequencing (NGS) platforms are being used for DNA sequencing. These platforms generate short fragments called 'reads' of length ranging from 35 to few hundreds of base-pairs. These reads are part of a large genome containing millions of base-pairs (the size of the human genome = 3×10^9 bp). The NGS platforms generate large amounts of accurate data at very low cost and at a greater speed when compared to older platforms (Nagarajan and Pop, 2013).

The large amount of data has posed many challenges for computer scientists who develop softwares to analyse these data. New algorithms and data structures have been proposed to speed-up the analysis. Databases have been created and statistical analysis programs have been developed for retrieving specific information from this data. *Sequence assembly* is a computational biology problem where the reads generated from the NGS machine are used to build the whole genome. An example of assembly is shown in Figure 1. A biological sample is preprocessed and given to a sequencing machine, which generates a set of short reads. These short reads are assembled to construct the genome. The 'T' in the read CTGTGTGTT, is an error as the exact match to the genome at that position was supposed to be 'C'. The error could be identified as the frequency of occurrence of 'C' at that position is more than that of 'T'. The error can occur during the sequencing process by the sequencing machine. Error can also occur while assembling the genome from the short reads, where the read is falsely mapped to a particular location of the genome. Due to these errors, genome assembly problem is more difficult to solve than the well-studied shortest superstring problem (Nagarajan and Pop, 2013).

Figure 1 NGS assembly: the DNA sample is given to sequencer, which generates read-set (see online version for colours)



Note: The read-set is processed in CPUS to generate contigs.

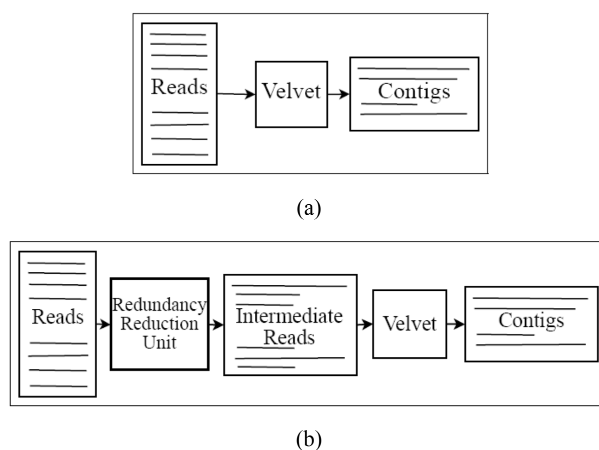
Clearly like most of the bioinformatics applications, NGS assembly is also data dominated. Even though compute infrastructure ranging from server racks to cloud farms exist for solving these problems, the time taken is enormous. For example, assembly of human genome using PASHA software took around 21 hours on a 8-core workstation with 72 GB memory (Liu et al., 2011).

Hardware accelerators like GPU and FPGAs are used along with processors to reduce this execution time by running the program on multiple computation units in parallel (Lin and Lin, 2014; Okuyama et al., 2012; Halstead et al., 2014; Marino and Li, 2014). Some bioinformatics applications along with assembly programs have been accelerated by FPGA-based accelerators. They have shown significant speedups (Che et al., 2008).

De novo assembly is a type of assembly process where genome is constructed without using a reference genome. It is the only way to construct a genome if the reference genome does not exist. As the shorter reads have less overlap information, the reads are generated with much more coverage in order to construct the genome. The overlap information from the reads are used to construct the contiguous consensus sequences known as *contigs*. The genome is constructed using these constructed contigs.

Two graph-based methods used for *de novo* assembly are the overlap layout consensus (OLC) and de Bruijn graph. In this paper, we propose a hybrid approach where we generate the overlap and layout using the OLC technique and build the consensus using a de Bruijn method. The key innovation is to use a (parallel) hardware implementation to remove the 'redundancy' in the input data and use state-of-the-art highly computationally intensive de Bruijn graph-based Velvet software (Zerbino and Birney, 2008) to build the consensus sequence.

Figure 2 Velvet flow and FPGA-based approach, (a) Velvet flow (b) FPGA-based approach



Note: In our approach, we generate intermediate contigs which are given to velvet for further processing.

Velvet is a widely used de Bruijn graph-based assembler (Zerbino and Birney, 2008). This software usually takes significant amount of time to execute. We attempt to accelerate it using FPGA-based accelerators. A high level design of our approach is shown in Figure 2. The reads are passed through redundancy remover unit (RRU) implemented in FPGA, which acts as preprocessor. The RRU is constructed using processing elements (PEs) connected in series. All the PEs form a pipelined structure and hence execute parallely. Each of the PEs stores a sequence. Reads are passed through each of the PE. The PE checks for overlap region at its ends and if the overlap region is greater than a threshold value, it is extended. These extended sequences form the intermediate contigs which are given to Velvet for constructing the final contigs. We have validated our design for 60 PEs. We estimate speed-ups of 13× using our approach using 3,000 PEs.

The key contributions of this paper are the following:

1. innovative way to speed-up *de novo* assembly of NGS data using FPGAs
2. hardware-software co-design to achieve this
3. efficient hardware implementation on FPGA.

Section 2 shows the different softwares available for genome assembly. Section 3 describes the NGS *de novo* genome assembly problem and the work done by other research groups that have been reported in the literature. In Section 4, we describe the overall approach and high level implementation used for initial analysis and to do feasibility study. This study leads us to prepare an overall methodology to achieve speedups using FPGAs which is described in Section 5. Section 6 shows some of the results and this is followed by conclusion in Section 7.

2 Genome assembly softwares

Many softwares have been developed to do assembly (Miller et al., 2010). Algorithms are modified in order to alleviate some of the complexities involved in the assembly and to execute efficiently on processors. Assembly softwares can be divided into two categories; mapping-based comparative assembly and *de novo* assembly. In the former method, assembly is done by mapping the reads to an already pre-existing reference genome. Even though the genomes of a particular organism contain lots of similarities, there are certain dissimilar regions which make each organism unique. These dissimilar regions are of interest to biologists as they show particular behaviour unique to that individual organism. Mapping the reads to a pre-existing reference genome might cause this uniqueness to be destroyed and hence, the software assemblers allow certain amount of mismatches and gaps. Some of the mapping-based assembly programs are SOAP (Li et al., 2008b), MAQ (Li et al., 2008a), Bowtie (Langmead et al., 2009) and RMAP (Smith et al., 2008).

The later method is called *de novo* assembly where the information is extracted from the reads and their overlaps. Some of the *de novo* assembly software programs are Velvet (Zerbino and Birney, 2008), Edena (Hernandez et al., 2008), PerM (Chen et al., 2009), BFAST (Homer et al., 2009) and Minia (Chikhi and Rizk, 2012).

De novo assembly takes more computational time than mapping-based assembly. Since the mapping-based assembly includes a pre-existing reference genome during mapping, the assembly process is biased and hence, in certain situations bioinformaticians prefer to use *de novo*-based assembly.

3 De novo assembly

De novo assembly is a method in which the genome is constructed using the reads without using reference sequence. It is the only way to construct new genomes. This method is also used when reference genome is available because the construction is unbiased.

3.1 Principle

The de novo genome construction from NGS data is complex due to the following reasons:

1 A very large amount of data has to be processed. Most of the algorithms need computers with large amount of RAM for processing the data. If the RAM usage has to be reduced, data partitioning has to be done to keep the 'data of interest' closer to the processor in the memory hierarchy. This process would involve swapping of data across the memory hierarchy and thus lead to increase the execution time.

2 There are some common sequences in the genome called repeats. Identifying the reads which form these repeats is non-trivial. Some plant genomes include more than 80% of repeat sequences.

3 The sequencing machine has a constraint on length of reads. If the read length is less than repeat it is almost impossible to detect which portion of the genome the read came from.

4 The data generated by the sequencing machines are not fully accurate and contain errors. The genome constructed may be erroneous if the programs do not take error correction into consideration.

De novo assembly can be divided into three categories; Greedy, OLC, and de Bruijn graph-based assemblers. The softwares like PCAP (Huang et al., 2003) and TIGER (Wu et al., 2012) that use greedy approaches make use of the overlap information for doing the assembly. In the greedy method, the pairwise alignment of all reads is done and the reads with the largest overlap is merged. This process is repeated till a single lengthy sequence is obtained.

The OLC method is a graph-based method where an overlap graph is constructed from the reads. Some of the software assemblers based on OLC are Edena (Hernandez et al., 2008) and CABOG (Miller et al., 2008). The reads become the node and edges show the overlap information. The nodes are placed in the form of a graph. Multiple sequence alignment (MSA) is done with the reads having more than two edges. Based on this, consensus sequence is constructed and sequencing errors are removed. A 'Hamiltonian path' in the graph is used to construct the contiguous sequences (contigs). Later, the whole genome is constructed using these contigs.

The de Bruijn graph assembly also uses a graph where the nodes are k-mers. A 'k-mer' is a sequence of 'k' base-pairs. Some of the software assemblers based on de Bruijn graph are PASHA (Liu et al., 2011), Velvet (Zerbino and Birney, 2008), Euler (Pevzner et al., 2001), etc. All the reads are broken into respective k-mers, i.e., all substrings of length 'k'. A graph is constructed with (k–1)-mers as nodes and the k-mers as edges. This graph contains all the overlap information contained in the reads for a particular k-mer length. Due to errors in the reads, there can be a chain of nodes that are disconnected, i.e., they do not converge into the graph. These are called 'tips'. The errors can also cause the graph to have redundant paths that have same starting and ending point, i.e., the paths converge back into the graph. These are called 'bubbles'. These tips and the bubbles are removed using heuristics and sometimes with sequence comparisons. 'Euler path' is used in this de Bruijn graph to construct the contigs, which in turn is used for constructing the whole genome.

The choice of assembly software is mainly dominated by the quality of assembly, speed of assembly and the RAM needed for the execution. Many techniques are used for error correction and improving the quality of assembly (Koren et al., 2012; Salmela and Schröder, 2011). Considering speed as the criteria, graph-based assemblers are preferred over greedy assemblers. In the graph-based assemblers, de Bruijn graph-based assemblers have become more popular as they are faster than OLC-based assemblers. This is because finding Hamiltonian path in a directed graph in OLC-based assemblers is a NP hard problem, while finding the Euler path is easier (Compeau et al., 2011). Also, MSA of the reads used in OLC method for removing errors is both compute intensive and memory intensive when compared to the techniques used in de Bruijn graph-based assemblers.

3.2 Related work with FPGA-based acceleration

Several groups have attempted to accelerate NGS short read mapping using FPGAs, where the genome is constructed by mapping the short reads to an already existing genome. Tang et al. (2012) have accelerated short read mapping and achieved 42× speed-up over software PerM (Chen et al., 2009). Aluru and Jammula (2014) review the different acceleration techniques used for genome assembly. Olson et al. (2012) have also shown acceleration of short read mapping on FPGA. The authors compare their results with BFAST software (Homer et al., 2009) and show 250× improvement and 31× when compared to Langmead et al. (2009). Fernandez et al. (2010) and Knodel et al. (2011) have also accelerated NGS short read mapping. The Convey Computer (2015) firm have developed the convey graph constructor (CGC), which use FPGAs to accelerate *de novo* assembly. They show speedups of 2.2× to 8.4×. Meng et al. (2014) have accelerated *de novo* genome assembly

using FPGAs, but they use optical methods for sequencing. As *de novo* assembly is non-biased, it is important. We attempt to accelerate it using FPGAs. As it is not fair to compare mapping-based assembly with *de novo* assembly, as both are different and have their own advantages and disadvantages, we compare our hardware implementation with existing *de novo* software-based implementation. In our previous work, we presented initial results (Varma et al., 2013). In this paper, we discuss the hardware implementation which was very briefly discussed. We also discuss the results in more detail and the algorithm to architecture design process. We have shown how the FPGA architecture can be modified by introduction of hard embedded blocks (HEBs) in order to get further speedups (Varma et al., 2014), where we discuss more on the introduction of HEBs.

4 Approach

We chose to accelerate de Bruijn graph-based assembly, as they take less amount of time to execute when compared to OLC-based assemblers. We use the fact that there is a lot of redundancy in short read sequencing. Lander and Waterman (1988) describe the use of redundancy for getting good quality assembly. This redundancy helps in providing coverage as well as eliminating errors encountered during sequencing of these short reads.

De Bruijn graph-based software assemblers take several GBs of RAM space while executing (Zhang et al., 2011). Efficient implementation of de Bruijn graph-based assemblers on an FPGA is difficult due to the memory resource constraints in current FPGAs. We model a hybrid approach where we implement a part of OLC-based assemblers on FPGA to remove the redundancy present in the reads. We run the de Bruijn graph-based assembler in software on the reduced set of reads from the FPGA.

This approach allows us to effectively use the FPGA resources for removing redundancy in the reads.

4.1 Algorithm

The main thrust in our approach is to find the overlap region between reads and store the overlap region only once. This can be done using a greedy approach. A read from the readset is picked which is called ‘starter’ sequence. The starter sequence is checked for extension with the rest of the reads in read-set. This process has to be repeated until the starter does not extend, meanwhile removing the reads from readset which extend the starter. After many such iterations, we are left with a reduced read-set and a extended starter. This extended starter which can not further be extended is stored as an ‘intermediate contig’. This process has to be repeated by picking a read from the remaining read-set, so that we store the overlap information only once. After checking with all the reads, if it does not extend, it can be made an intermediate contig. If it gets extended the read causing the extension is removed. An example of a single contig construction is shown in Figure 3. In this example, the read 5 is made a starter. The extension in each round is shown. During the first round read number 4 and 6 extend the starter. Similarly, in round 2 reads 3 and 7 extend the starter. In round 3, all the reads are used up for extensions and the particular contig is constructed. For this ordering of reads and choice of starter 3 rounds were required for the construction of contig. We use this idea for constructing intermediate contigs.

To implement this in parallel, we can start by picking a small subset of reads (multiple starters) and start comparing and checking if they can be extended by the reads left in the remaining read-set. After single iteration, the starters which did not get extended can be removed and put in intermediate contigs set, as there is no chance for them to get extended in further iterations. The removed starters are replaced with new reads from the remaining read-set for next iteration.

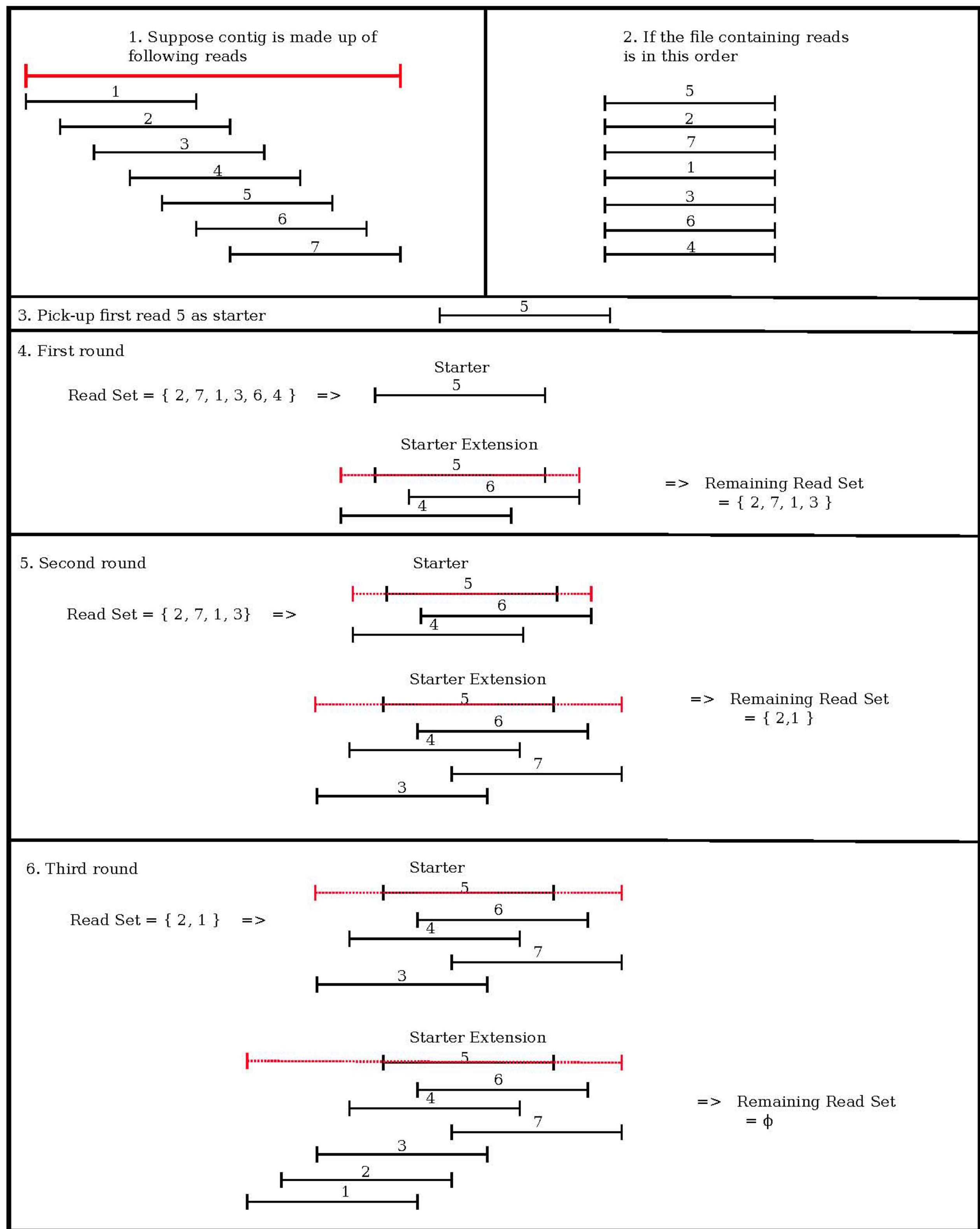
As we do not consider error checking while extending, we call the contigs generated from our approach as ‘intermediate contigs’, which can be further processed by other tools like velvet. The overall flow diagram is as shown in Figure 4. This approach reduces the size of the input to the velvet software, as shown in Figure 5, and thereby giving speedups compared to software. There may also be reduction in the RAM usage due to smaller input file.

To study the benefits of our approach, we used an open source software known as Mapsembler (Peterlongo and Chikhi, 2012), which does targeted assembly. It takes NGS raw reads and a set of input sequences (starters). The software determines if the starter is read coherent, i.e., starter is a part of the original sequence. The neighbourhood of the starter is given as output if the starter is read coherent.

The algorithm is described in Algorithm 1. All the k-mers in all of the starters are indexed and stored in a hash table ‘I’. The hash table consists of starter number and the corresponding position of the k-mer in that particular starter. A read is taken from the NGS read set and the respective k-mers are searched in the hash table. If the k-mer is already hashed, the corresponding starters are tried for extension with the reads.

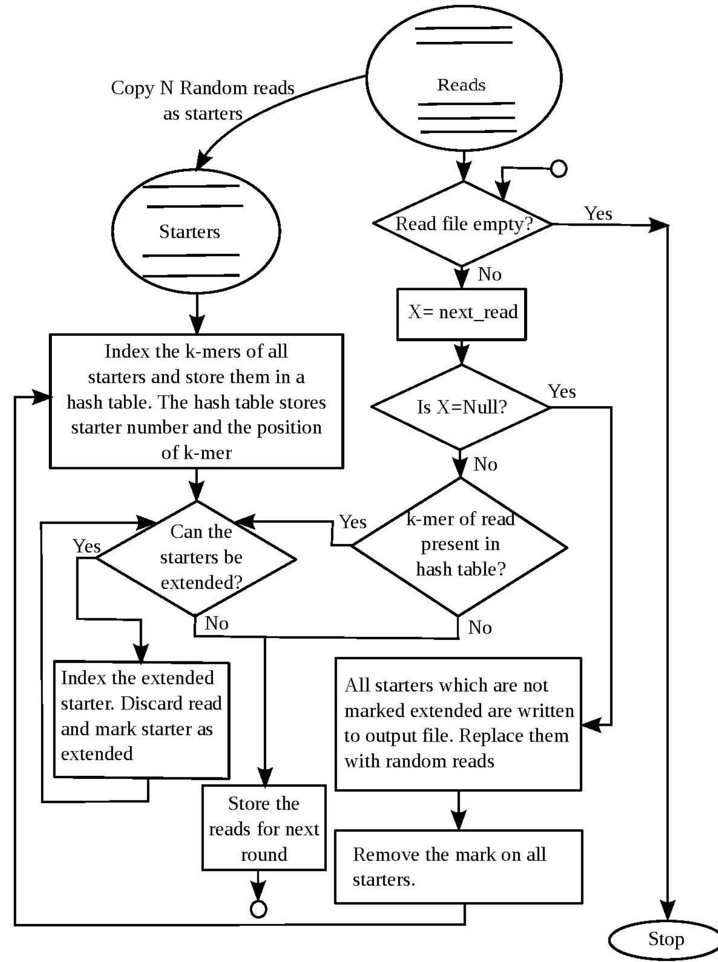
This high level model based on Mapsembler was used to study its effectiveness in removing redundancy. This model was also used for conducting experiments with Velvet software on various data sets. From these experiments, we verified that the time taken by Velvet software was dependent on the input file size. Removing the redundancy by our approach did not cause significant loss in the quality of output. We also studied the quality by varying the mismatches allowed during extension. Even though the software model was essentially done to study the initial benefits of our approach, we would also like to mention here that this approach takes very long time to execute, as the reads are compared serially with the starters. In fact, the time taken on a dual core desktop computer is more than the Velvet software time in most of the cases.

Figure 3 Example showing construction of contig using our approach (see online version for colours)



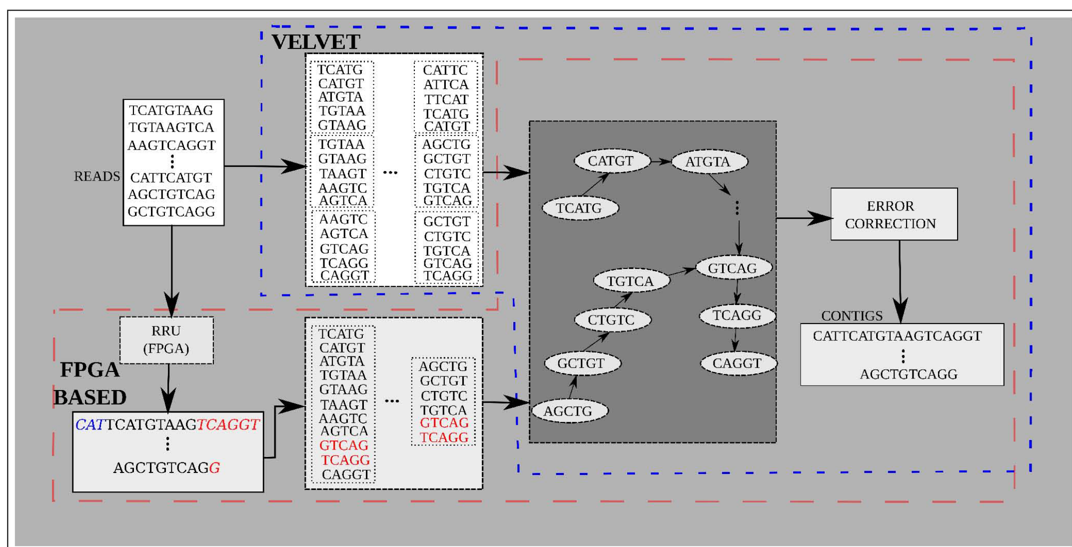
Notes: In each round starter is extended. Contig is constructed from the reads in three rounds.

Figure 4 Software flow for estimation



Note: Mapsembler was used for feasibility test and estimation.

Figure 5 FPGA-based *de novo* assembly (see online version for colours)



Note: The input size to velvet is reduced considerably by using our approach.

Algorithm 1 Redundancy removal using Mapsembler

```
1:  procedure RRUMAPSEMBLER(Read-set R)
2:      pick N random reads and store as starter s ∈ S
3:      delete these reads from R
4:      for each starter s ∈ S do
5:          index all k-mers of s in index-table I
6:          starter → extendedFlag = 0
7:      end for
8:      if | R |≠ ∅ then
9:          for each read r ∈ R do
10:             for each k-mer k in r do
11:                 if k indexed in I then
12:                     if r extends corresponding s then
13:                         s = extended(s)
14:                         set starter → extendedFlag = 1
15:                         delete r from R
16:                     end if
17:                 end if
18:             end for
19:         end for
20:         for each starter s ∈ S do
21:             if starter → extendedFlag = 0 then
22:                 store s as intermediate contig in IC
23:                 replace s with random read r ∈ R
24:             else
25:                 starter → extendedFlag = 0
26:             end if
27:         end for
28:     end if
29:     store all starters to IC
30:     Return IC
31: end procedure
```

4.2 From algorithm to architecture

To take advantage of the FPGA architecture, we do a streaming design where PEs are connected in series. Each of the PEs stores a sequence called starter. The PEs are connected in a series. 'N' starters in the corresponding 'N' PEs are populated with 'N' random reads. A read from remaining read set is streamed through the PEs. In each PE, the read is checked if it can extend the starter. If extended, the starter is updated with the extended starter. This process is continued till all the reads are exhausted. We use the term 'round' frequently in the rest of the paper which means that all the reads from the read set are compared once with current set of starters and tried for extensions. After each *round*, the starters which are not extended are replaced with new random reads from the remaining read set. These non-extended starters of the current *round* are stored in an output file.

This process removes redundancy in the reads. The redundancy reduction process is repeated several times till there are no more reads. The remaining reads along with all the non-extending starters from previous rounds constitute the *intermediate reads*. These intermediate reads are stored in an output file. The intermediate reads are less in number and longer in length. This output file is given as input to Velvet software for removing errors and generating contigs.

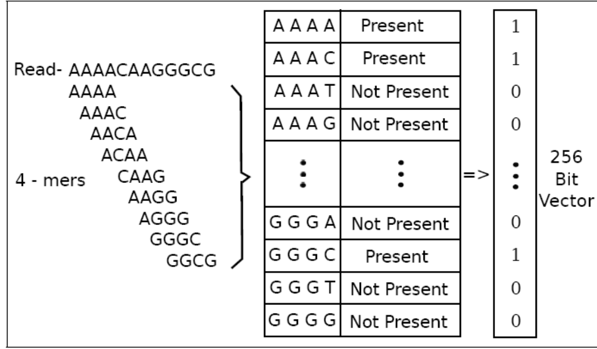
In order to get better performance, we do the hardware implementation of the redundancy removal unit using FPGAs. The proposed hardware model differs significantly form the software model. The hash-based searching of k-mers in the software model is not implemented in this hardware model due to memory resource constraints. The reads are compared with the starter ends and tried for extension. In each cycle, the read is shifted and checked if it can extend the starter. For example:

$$\left[\begin{array}{ll} Cycle1 & \\ Starter- & ACTGTCGTGTCTGC \\ Read- & TGTCGTGTCTGCGC \\ Cycle4- & \\ Starter- & ACTGTCGTGTCTGC \\ Shifted\ Read- & TCGTGTCTGCGCTG \\ Extd:Starter- & ACTGTCGTGTCTGCGCTG \end{array}\right.$$

The extension phase is expensive as it is a long process. To avoid this long delay in the extender, we add a filter which eliminates reads with no probable extensions. The number of cycles needed for extension is equivalent to the difference of read length and k-mer length. From the software implementation, it is observed that for a single round, the number of reads used for extension are very small when compared to reads that extend the starters. To take advantage of this feature, we propose a *pre-filter* block. The pre-filter is added before the extension phase. Pre-filter compares the signature of the reads with the signature of the starter. This signature is called the *ReadVector* and is constructed by encoding the 4-mers in binary format. 4-mer was chosen for signature because the vector width would be 256 corresponding to 4^4. If we choose a signature with more than 4-mer, then the signature will become much more lengthy and hence would require large memory for its storage and larger amount of resources for doing the pre-filter.

An example of construction of the readvector is shown in Figure 6. In the example, the readvector for read AAAAAAAGGGGG is '100100…001'. Each bit represents a particular 4-mer. It is set if the 4-mer exists else it is reset. Only one bit is stored if there is repetition of the 4-mer. The construction of read vectors has to be done only once, as it does not change during the whole process of assembly. We first implemented this in software and found that it was taking significant amount of time and so we implemented this in hardware. The details of this implementation are explained in Section 5.

Figure 6 Example showing construction of Read Vector



The PE consists of the pre-filter and the extender part. Each PE has to store the starter sequence. As the starter keeps increasing in length, it becomes difficult to store the whole starter inside the PE, due to limited resources available in the FPGA. In order to alleviate this problem, we decided to store the left end and right end of the starter in the PE, and reconstruct the starters in the host. The length of the starter ends stored in PE is equivalent to the read-length, thus, allowing extensions at the ends. With this approach, the reads which are completely covered by the starter and do not extend the starter are not eliminated. We found that this does not cause significant difference in the speed-up. By using this approach, we gain two advantages:

1. The memory resource usage is reduced, as we store only the left-end and the right-end of the starter.
2. We need not re-construct the vectors for the starter-ends used in the pre-filter. This is because the extension is caused by the read for which read-vector is already available. If starter is extended, the starter end and its corresponding vector is replaced by the read and its vector. This saves considerable amount of time, as construction of vectors for each extension would be very expensive, both in terms of resource usage and execution time.

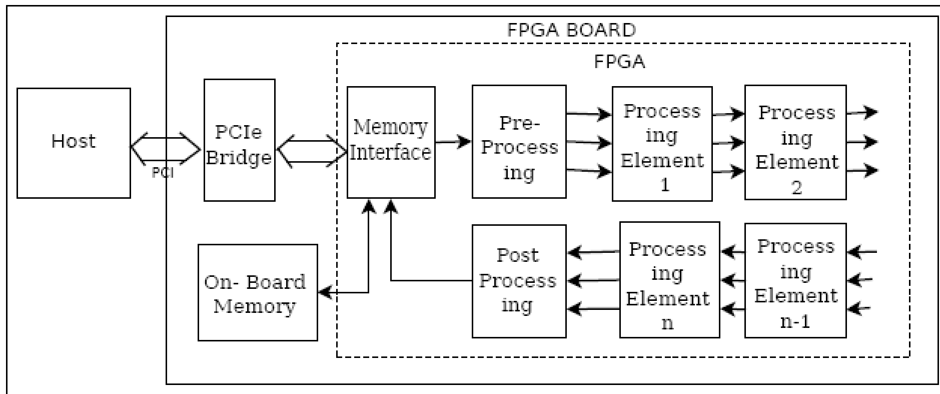
The clock cycles required by each PE to process a read varies widely. The number of cycles is highly dependent on the read which is being processed. Due to rejection by the pre-filter there could be lot of data generated in a very few cycles for the next PE, or the next PE could be waiting for the extension phase of the PE. Due to this, there is irregularity in the time which PE can start processing the reads. To keep the PEs busy for most of the time, we have introduced FIFOs in between the PEs. As the implementation is done on FPGAs, the BRAMs were used for the FIFO implementation for effective resource usage.

5 Hardware implementation

The overall block diagram is shown in Figure 7. It shows the FPGA board interface with the host. We use Alpha-Data (Alpha-Data, 2015) board for hardware implementation. The board has a PCIe bridge which is used for data transmission between host and vice versa. The memory interface unit connects the onboard memory and the PCIe bridge. In our design, we use the 'memory interface' unit to send data to a 'pre-processor'. From the pre-processor, a series of 'PEs' are connected through a set of FIFOs. The FIFOs are not shown in Figure 7 for clarity. The last PE is connected to a 'post processor' connected back to memory interface unit. The expanded diagram showing different stages is shown in Figure 8.

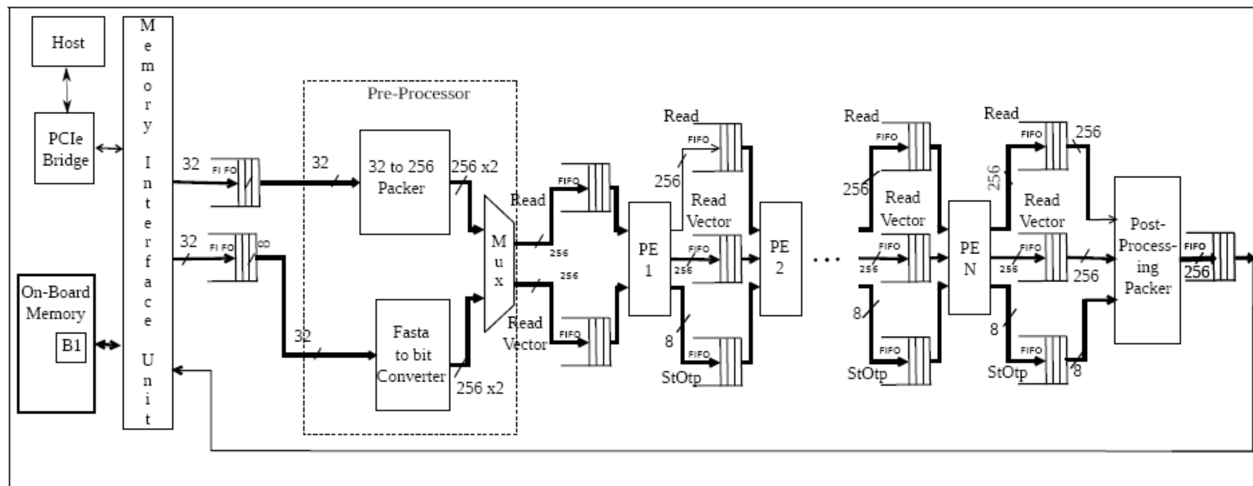
The read set in fasta input file format is sent from the host to the FPGA board through PCIe bus. For initialising the starters, we encode the most significant three bits of the read. The fourth bit is used for marking the read that it has extended as a starter. In order to reconstruct the starters, the starter and the position of the extension is sent as output through the fifoSet. Reconstruction of starters is done in software.

Figure 7 FPGA board



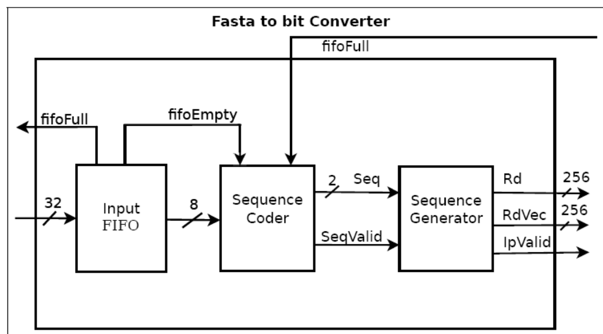
Note: The RRU unit is implemented on an FPGA board connected to host through PCIe interface.

Figure 8 Block diagram of hardware implementation



Note: The interconnectivity between modules is shown.

Figure 9 Fasta to bit converter



5.1 Fasta file to bit file converter

The *fasta to bit converter block* reads data from the input buffer and encodes the base-pairs in binary format; 'A' as '00', 'C' as '01', 'T' as '10' and 'G' as '11'. It also generates the readvector. The block diagram is as shown in Figure 9. The Fasta to bit converter has an input BRAM which stores a part of the Fasta file. This block has two parts – the sequence coder and bit sequence generator which generates the read vector.

The sequence coder reads data from the input FIFO and encodes the base pairs in binary format and removes the comments. Sequence coder is implemented as a state machine. The state diagram is shown in Figure 10. The bit sequence generator is made up of a 256 bit shift register, a 256 bit register and a control unit as shown in Figure 11. The two bit sequence code from the sequence coder unit is pushed into the shift register by shifting two bits to the left. The first 8 bits are taken as an address to set the bit on the second 256 bit register. This register is read after a single read is encoded completely, which is known by the 'seqValid' signal coming from the sequence coder unit. Control unit controls the shift operation and generates 'FifoWr' signal for writing the readVector and the read when it is ready.

Figure 10 State diagram for sequence coder

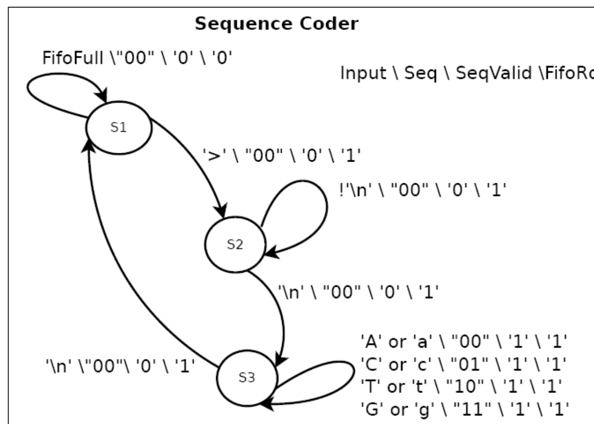
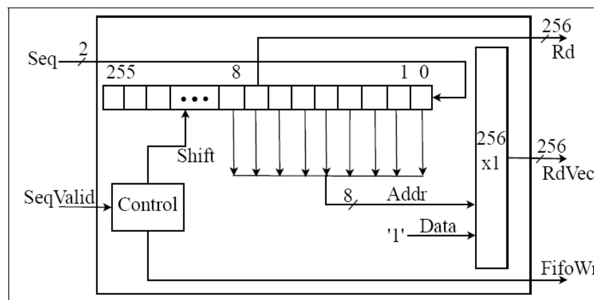
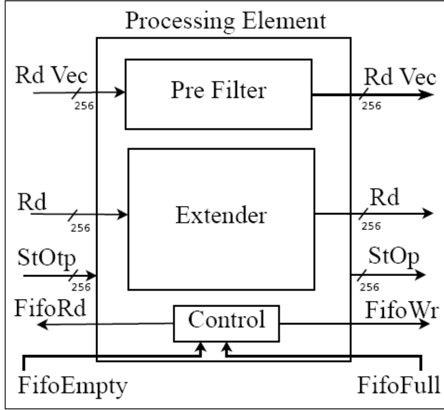


Figure 11 Bit sequence generator



This binary conversion from ASCII is done only once and the rest of the units use the binary format for further processing. During the next rounds, this state machine remains idle and so 'mux' and 'control' is used for selecting the required FIFO.

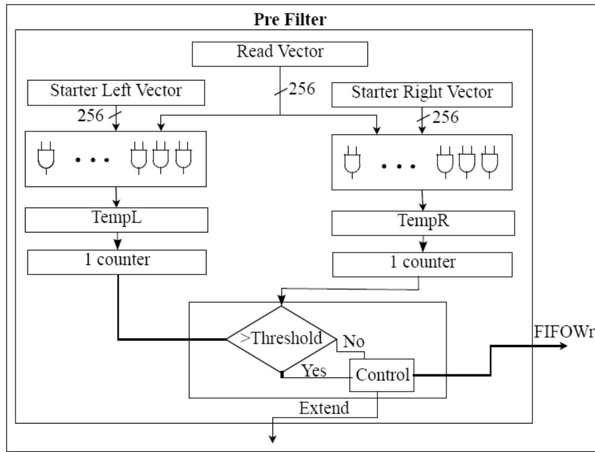
Figure 12 Processing element



5.2 PE design

The pre-processing block is followed by a series of *PEs*. The PE contains two parts; the *pre-filter* and the *extender* as shown in Figure 12. A read and corresponding readVector is taken from the remaining read set and compared with the starters for extension. Each PE stores the information of a single starter. The starter will be saved as an intermediate contig if it does not extend in the current round. Due to memory constraints in the FPGA, we store only the right end and the left end of the string for extension. Each of the starters have two parts-each of length equivalent to the read length. The left end is called *starterLeft* and the right end is called the *starterRight*. First '*n*' reads, considering that the reads are arranged randomly are copied to starterLeft and starterRight. Similarly, we store *starterVectorRight* and *starterVectorLeft* used for pre-filtering.

Figure 13 Pre-filter design



5.2.1 Pre-filter design

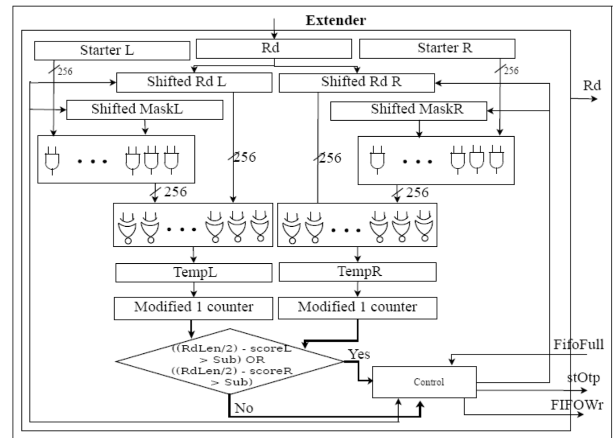
The pre-filter design is shown in Figure 13. In the pre-filter a logical 'AND' is done between readVector and starterVector-Left and stored in TempL register. Similarly, logical 'AND' is done between readVector and starVectorRight and stored in TempR register. The 1-counter block counts the number of '1's in the TempL and TempR register. If the number of '1's is greater than the threshold, the read is passed to the extender else it is passed to the FIFO for next processing element to evaluate it. The 1-counter block lies in the critical path and hence defines the clock period of operation. We implemented two versions of the one counter-one using Wallace tree and the second using the carry chain available in the Xilinx FPGA slices. The Wallace tree is built out of 6:3 compressors as the Xilinx Virtex 6 FPGA has 6 input LUTs. For the 256 bit implementation, we need a Wallace tree of 258 bits using the 6:3 compressors. The tree is built in five stages and requires 100 (43 + 24 + 14 + 10 + 9) 6:3 compressors with a carry adder at the final stage. The second implementation is done by adding each bit using the carry-chain available in the FPGA. This implementation took less time, but slightly more area when compared to the Wallace tree implementation.

5.2.2 Extender design

The extender design is shown in Figure 14. We use a 'maskL' and 'maskR' register for masking the corresponding bits in the starter after shifting the read. In the beginning, these registers are initialised with twice read length '1's on the right and rest of the bits are set to '0'. The following operations are done in the extender:

$$\begin{cases} tempL = (starterLeft \ AND \ maskL) \ XNOR \ shiftedRdL \\ tempR = (starterRight \ AND \ maskR) \ XNOR \ shiftedRdR. \end{cases}$$

Figure 14 Extender design



The corresponding scores, scoreL and scoreR are calculated from tempL and tempR, respectively using modified 1-counter. A modified 1-counter is needed as we are encoding the basepair in two bits. An example is shown in Figure 15. Here, we see that the total score should be calculated by checking 2 consecutive 1s. For calculating this, we modify the Wallace tree implemented in pre-filter block. We store the appropriate values in the LUTs of the first stage of the Wallace tree implementation. For example, we store output as 11 for 11 11 11, 01 for 01 10 11 and 10 for 11 11 01. So the score gives the exact matches of the

two sequences. If ((readLength/2) – scoreL) is less than allowed substitutions, the starter is extended on the left side or if ((readLength/2) – scoreR) is less than allowed substitutions, the starter is extended on the right side. The starterLeft and starterVectorLeft are replaced by the read and readVector, respectively, if the starter is extended on the left side. Similarly, starterRight and starterVectorRight are replaced by the read and readVector, respectively, if the starter gets extended on the right side. This step of counting the number of '1's can be eliminated if the allowed substitution is set to zero. For this, only an 'XOR' operation of between (shifted read 'AND' with mask) and starter has to be done and checked if it is equivalent to zero for extension. This saves lots of resources and also from the experiments conducted we found that the quality of results is better with threshold set to '0'. If a starter is not extended on either of the two sides, then shiftedReadL and shiftedReadR are shifted to right and left, respectively by two bits as base-pair is encoded with two bits. The maskL and maskR are changed as follows:

$$\begin{cases} maskL = maskL \ AND \ (maskL << 2) \\ maskR = maskR >> 2. \end{cases}$$

This process is repeated till the read is shifted (readLength – kmer length), as we do not extend starter if there are less than 'k' matches. The reads that do not extend any of the starters are put in the next fifoSet for further processing by the next PEs.

Figure 15 Example of score calculation in extender; e.g., score = 11 (see online version for colours)

```
                   TCGTGTCGTCTTG              -- Starter
                         CGTCGTTGTACTA        -- Shifted Read
 00 00 00 00 00 00 11 11 11 11 11 11 11       -- MASK
                  AND
 10 01 11 10 11 10 01 11 10 01 10 10 11       --Starter
--------------------------------------------------------------
 00 00 00 00 00 00 01 11 10 01 10 10 11       --temp
                 X-NOR
 00 00 00 00 00 00 01 11 10 01 11 10 10 11 10 00 01 10 00
                                              --shifted read
--------------------------------------------------------------
 11 11 11 11 11 11 11 11 11 11 10 11 10       --tempScore
```

6 Results and discussion

Zhang et al. (2011) have done a comparison of *de novo* assembly softwares. The authors have provided scripts for generating the read-set from the genome. We used these scripts to generate the read files for *E. coli*, swinepox and human influenza. For evaluating our approach, we generated the single ended readset with 100× coverage for read-length 36 and 75 and 1% error rate similar to what was reported by Zhang et al. (2011). For the software only flow time, Velvet software was run using the readset directly on a desktop computer with Intel (R) Core (TM) 2 Duo CPU E4700 running at 2.60 GHz with 4 GB RAM.

6.1 *Resource utilisation and operating frequency*

We have implemented the RRU on FPGA and obtained the clock period and utilised FPGA resources after running place and route tools provided by Xilinx (2015) ISE 14.1. We use these parameters to estimate the speed-ups for running the Velvet on the output of RRU after each round. From place and route tools, the maximum clock frequency for the whole of the design was found to be 200 MHz. We get better performance by using multiple clocks. The sequence coder and generate units were able to run at a maximum frequency of 350 MHz on Virtex-6 FPGA. The maximum frequency of operation for the rest of the units was 200 MHz. The hardware implementation was done on Alpha-Data board having Xilinx Virtex-6 (XC6VSX475T) FPGA.

Table 1 Resource utilisation

Component	*Slices*	*BRAM*
pre-processor	559	-
in-fifo	17	32
fifo-set	244	8
post-processor	200	-
out-fifo	93	15
PCI-interface	2047	15
Others	35	-
PE-RdLen-36 Threshold-0	380	-
PE-RdLen-36 Theshold-11	939	-
PE-RdLen-75 Theshold-0	643	-
PE-RdLen-75 Theshold-11	1,175	-

The resource utilisation obtained from ISE module level utilisation, for the different units are shown in Table 1. Here, we considered design which does not allow any substitution. The resources occupied by PEPE vary depending on the read length. When the threshold value of the pre-filter is set to 'zero' the 1-counter is removed and thus number of slices occupied is significantly reduced.

Table 2 Number of PEs on Xilinx devices

Xilinx device	*XC6VSX475T*	*XC7V2000T*
PE-RdLen-36 Threshold-0	110	467
PE-RdLen-36 Threshold-11	58	247
PE-RdLen-75 Threshold-0	78	330
PE-RdLen-75 Threshold-11	48	206

Table 2 shows the number of PEs that could be implemented on Xilinx Virtex-6 (XC6VSX475T) FPGA. This table also shows the estimates of number of PEs on a larger Xilinx Virtex-7 (XC7V2000T) device considering 97% resource utilisation. This is actually an under-estimate as many slices from various units get combined during the synthesis flow and more logic can be realised on the device. The threshold in this table also refers to the pre-filter

threshold. If this pre-filter threshold is zero, the resource usage is less and hence the number of PEs which can be implemented on the device increases.

Note that for design with larger number of PEs where multiple FPGAs would be required, we have not considered inter FPGA transfer time in our estimates. We assume FPGAs are connected in series and the data is streamed from the host, through the FPGAs and finally, back to the host.

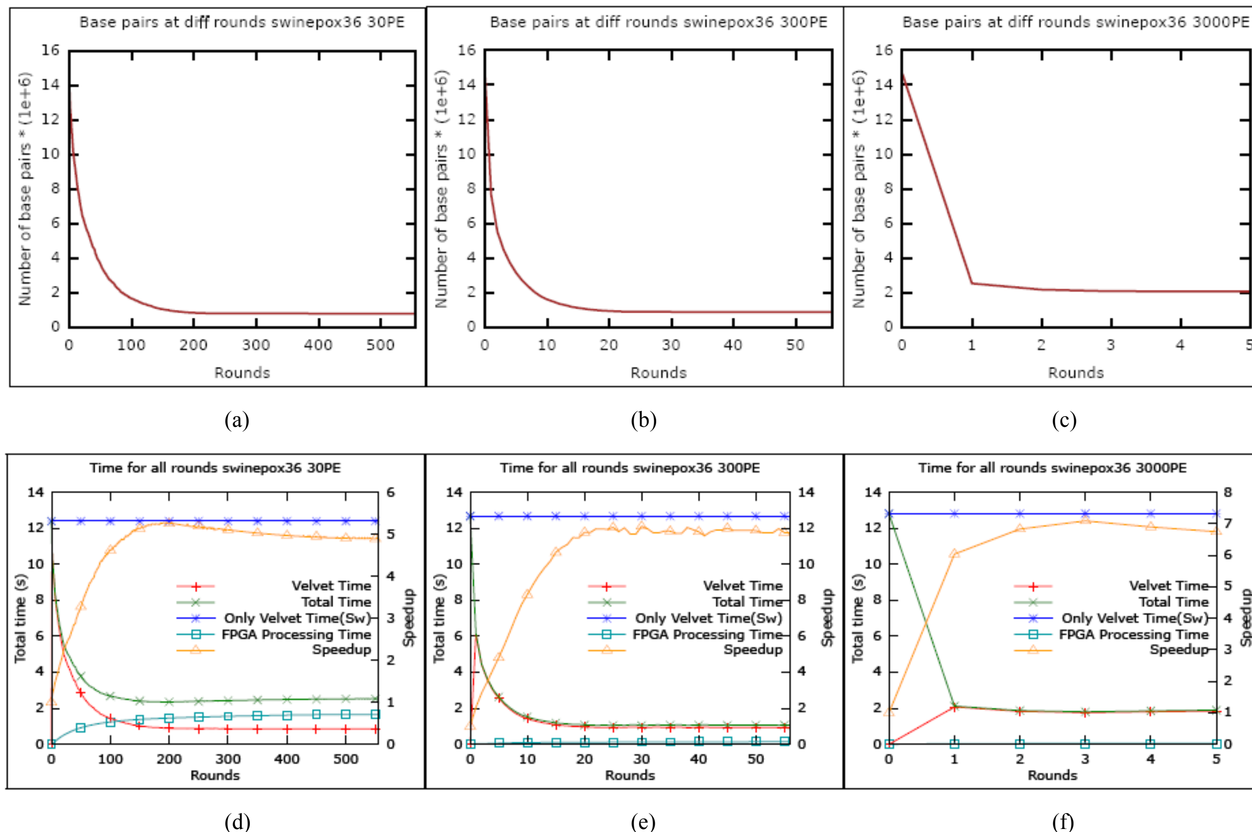
6.2 *Speedups over software*

Figure 16 shows the graphs of the speed-ups at different rounds for swinepox with read length 36 using 30 PE, 300 PE and 3,000 PE using Velvet software software. Figure 16 shows maximum speed-ups in the range of 5.2× to 11.9× for swinepox with read-length 36 over Velvet software. We also observe that the speed-ups reach a maximum and then start to decrease with increasing number of PEs. The reason for this is that the utilisation of the PEs goes down after a peak and hence the read and write cycles dominate.

We have considered the worst case time by setting the threshold value for the pre-filter to zero. Similarly, we have the results on *E. coli*, swinepox and human influenza, not shown here due to space constraints. We observe same trends. The reduction in size of the input file in terms of base-pairs to Velvet software is shown in Figure 17. For a larger genome like *E. coli*, we need to have more PEs to get significant speed-ups. The maximum speed-ups are tabulated in Table 3. The speed-ups in each case first increases, reaches a peak and then tapers down. The initial increase can be attributed to the high reduction of input file size during the initial rounds. After these initial rounds, the redundancy removal is more limited and so the time taken by Velvet is almost constant. The FPGA processing time is incremental in nature and hence goes on increasing after each round. Even though there is not much redundancy removal during the later rounds, the hardware unit takes at least as many cycles as the number of reads and writes in each PE.

From these results, we can determine a termination criterion for stopping FPGA processing to get maximum benefits from FPGA processing. For this, we keep track of reduction in the total size of data set in each round and if this is incrementally less than threshold we stop further rounds. In the future, this step would be integrated with the auto tuning of the pre-filter threshold.

Figure 16 Speedups and number of base-pairs after each round for swinepox read-length 36, (a) input size 30 PE (b) input size 300 PE (c) input size 3,000 PE (d) speedups 30 PE (e) speedups 300 PE (f) speedups 3,000 PE (see online version for colours)



Note: Figures 16(d), 16(e) and 16(f) show speedups over Velvet software.

Table 3 Maximum speed-ups over Velvet software

Sample	*Swinepox*	*Swinepox*	*H. influenza*	*H. influenza*	*E. coli*	*E. coli*
Read-length	75	36	75	36	75	36
PE\size	30.8 MB	48.4 MB	449 MB	729.7 MB	1.2 GB	2.1 GB
30 PE	3.5×	5.2×	1.1×	1.09×	1.2×	1.09×
300 PE	13×	11.9×	3.2×	3.6×	2.5×	2.1×
1,000 PE	6.5×	10.5×	6.8×	6.0×	4.4×	5.02×
3,000 PE	4.8×	7×	6.8×	10×	6.5×	9.2×

Figure 17 Input sizes (bp) before and after FPGA processing (see online version for colours)

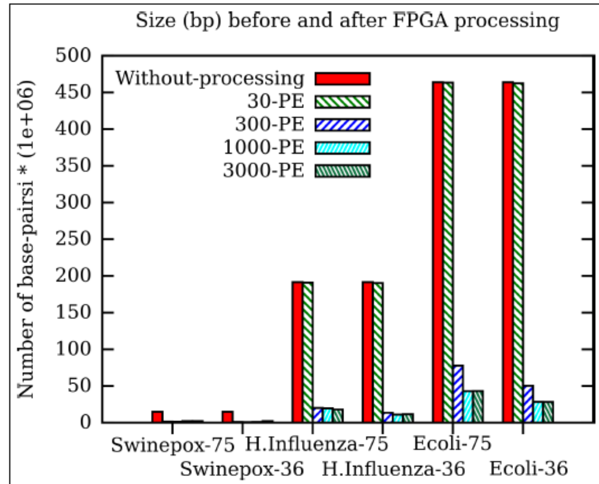


Table 4 Quality of assembly

Sample parameters	*Swinepox 75*		*E. coli 36*	
	N-50	*Max contig*	*N-50*	*Max contig*
FPGA-based\ Velvet	119,046	119,046	14,988	100,485
30 PE	119,046	119,046	14,988	100,485
300 PE	102,563	102,566	14,988	100,485
1,000 PE	119,046	119,046	15,344	100,485
3,000 PE	119,046	119,046	15,351	100,485

6.3 *Quality*

There are many factors which affect the quality of assembly. The quality is dependent on the sequencing machine. After the sequencing, the quality of assembly is dependent on many parameters that are used in the assembly algorithms. Mostly the input parameters to the assembler like k-mer length and number of mismatches allowed can significantly affect the quality of the output. The most popular metrics to measure quality are the maximum length of the contigs and the ‘N50’. N50 is the minimum length of the contig such that summing up the length of only those contigs whose length is more than N50 cover 50% of the genome. The quality of Velvet output using these metrics for different PEs is tabulated in Table 4. For example, the N-50 for Swinepox using Velvet software was 119046 which remained the same when the FPGA-based RRU was used. From the various experiments conducted we observed that by not allowing mismatches during extension, there was no (significant) loss in quality of output as shown in results.

7 Conclusions

Genome assembly is used in many fields like personalised medicine, meta-genomics, forensics, etc. NGS can be used to solve diverse biological problems. These platforms produce several gigabytes of data in a single run. There is a need for faster and memory efficient tools to analyse and make sense of this large data. *De novo* assembly has some advantages over the mapping-based assembly, but these software programs take more time to execute. We have used a hybrid approach which uses techniques from both OLC method and de Bruijn method for accelerating assembly. We implemented our design using FPGAs and used them as hardware accelerators.

From the results, we find that the speed-up is dependent on the nature and size of input data. For a fixed number of PEs, the speed-up first increases and then tapers down with larger number of rounds as FPGA processing time starts dominating. Maximum speed-up increases with number of PEs and reduces after reaching peak. We estimate speed-ups up-to 13× using our hybrid approach.

The use of phred values which provide the information on the quality of reads generated by the NGS platform are becoming increasingly popular. A filter is typically used to filter out reads with lower score. This can be easily be integrated in our flow by modifying the pre-processor block. This can be done by adding one more state in the state diagram shown in Figure 10. The hardware area overhead is minimal as it would require a comparator and the delay would not be a concern since it is implemented as a pipeline stage.

The multi-FPGA boards will be able to accommodate more number of PEs and hence can be used for reducing assembly time of larger genomes. We intend to implement the redundancy removal unit on multiple multi-FPGA boards and estimate the speedups for large genomes. We also want to modify Velvet to accelerate the assembly process using FPGAs. There could be reduction in the RAM usage using our approach. This study of reduction in RAM usage and velvet intervention to further speedup *de novo* genome assembly is a part of our future work. We also

intend to study benefit of our approach on softwares like Minia (Chikhi and Rizk, 2012), which use less memory for assembly.

Acknowledgements

We would like to thank Pierre Peterlongo, INRIA Rennes, for introducing us to Mapsembler and helping us understand the code. We would like to thank Xilinx for their generous donations. We would also like to thank IRISA and Amar Nath and Shashi Khosla School of Information Technology, IIT Delhi, for allowing us to collaborate.

References

Alpha-Data (2015) *Alpha-Data FPGA Boards* [online] http://www.alpha-data.com/ (accessed 2-1-12015).

Aluru, S. and Jammula, N. (2014) 'A review of hardware acceleration for computational genomics', *Design Test, IEEE*, February, Vol. 31, No. 1, pp.19–30.

Che, S., Li, J., Sheaffer, J., Skadron, K. and Lach, J. (2008) 'Accelerating compute-intensive applications with gpus and FPGAs', in *Symposium on Application Specific Processors, 2008, SASP 2008*, June.

Chen, Y., Souaiaia, T. and Chen, T. (2009) 'PerM: efficient mapping of short sequencing reads with periodic full sensitive spaced seeds', *Bioinformatics*, Vol. 25, No. 19, pp.2514–2521.

Chikhi, R. and Rizk, G. (2012) 'Space-efficient and exact de Bruijn graph representation based on a bloom filter', in Raphael, B. and Tang, J. (Eds.): *Algorithms in Bioinformatics, ser. Lecture Notes in Computer Science*, Vol. 7534, pp.236–248, Springer, Berlin, Heidelberg.

Compeau, P.E.C., Pevzner, P.A. and Tesler, G. (2011) 'How to apply de Bruijn graphs to genome assembly', *Nature Biotechnology*, Vol. 29, No. 11, pp.987–991.

Convey Computer (2015) *Convey GraphConstructor* [online] http://www.conveycomputer.com (accessed 2-1-12015).

Fernandez, E., Najjar, W., Harris, E. and Lonardi, S. (2010) 'Exploration of short reads genome mapping in hardware', in *International Conference on FPL*, September, pp.360–363.

Halstead, R.J., Villarreal, J. and Najjar, W.A. (2014) 'Compiling irregular applications for reconfigurable systems', *International Journal of High Performance Computer Networks*, June, Vol. 7, No. 4, pp.258–268.

Hernandez, D., François, P., Farinelli, L., Østerås, M. and Schrenzel, J. (2008) 'De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer', *Genome Research*, May, Vol. 18, No. 5, pp.802–809.

Homer, N., Merriman, B. and Nelson, S.F. (2009) 'BFAST: an alignment tool for large scale genome resequencing', *PLoS ONE*, November, Vol. 4, No. 11, p.e7767.

Huang, X., Wang, J., Aluru, S., Yang, S-P. and Hillier, L. (2003) 'PCAP: a whole-genome assembly program', *Genome Research*, Vol. 13, No. 9, pp.2164–2170.

Knodel, O., Preusser, T. and Spallek, R. (2011) 'Next-generation massively parallel short-read mapping on FPGAs', in *IEEE International Conference on ASAP*, September, pp.195–201.

Koren, S., Schatz, M.C., Walenz, B.P., Martin, J., Howard, J.T., Ganapathy, G., Wang, Z., Rasko, D.A., McCombie, W.R., Jarvis, E.D., Ed, J. and Phillippy, A.M. (2012) 'Hybrid error correction and de novo assembly of single-molecule sequencing reads', *Nature Biotechnology*, Vol. 30, No. 7, pp.693–700.

Lander, E.S. and Waterman, M.S. (1988) 'Genomic mapping by fingerprinting random clones: a mathematical analysis', *Genomics*, Vol. 2, No. 3, pp.231–239.

Langmead, B., Trapnell, C., Pop, M. and Salzberg, S. (2009) 'Ultrafast and memory-efficient alignment of short DNA sequences to the human genome', *Genome Biology*, Vol. 10, No. 3, p.R25.

Li, H., Ruan, J. and Durbin, R. (2008a) 'Mapping short DNA sequencing reads and calling variants using mapping quality scores', *Genome Research*, Vol. 18, No. 11, pp.1851–1858.

Li, R., Li, Y., Kristiansen, K. and Wang, J. (2008b) 'SOAP: short oligonucleotide alignment program', *Bioinformatics*, Vol. 24, No. 5, pp.713–714.

Lin, C.Y. and Lin, Y.S. (2014) 'Efficient parallel algorithm for multiple sequence alignments with regular expression constraints on graphics processing units', *International Journal Computer Science Engineering*, January, Vol. 9, Nos. 1/2, pp.11–20.

Liu, Y., Schmidt, B. and Maskell, D. (2011) 'Parallelized short read assembly of large genomes using de Bruijn graphs', *BMC Bioinformatics*, Vol. 12, No. 1, pp.354–363.

Marino, M.D. and Li, K. (2014) 'Insights on memory controller scaling in multi-core embedded systems', *International Journal of Embedded Systems*, Vol. 6, No. 4, pp.351–361.

Meng, P., Jacobsen, M., Kimura, M., Dergachev, V., Anantharaman, T., Requa, M. and Kastner, R. (2014) 'Hardware accelerated novel optical de novo assembly for large-scale genomes', in *24th International Conference on Field Programmable Logic and Applications (FPL)*, pp.1–8.

Miller, J.R., Delcher, A.L., Koren, S., Venter, E., Walenz, B.P., Brownley, A., Johnson, J., Li, K., Mobarry, C. and Sutton, G. (2008) 'Aggressive assembly of pyrosequencing reads with mates', *Bioinformatics*, Vol. 24, No. 24, pp.2818–2824.

Miller, J.R., Koren, S. and Sutton, G. (2010) 'Assembly algorithms for next generation sequencing data', *Genomics*, Vol. 95, No. 6, pp.315–327.

Nagarajan, N. and Pop, M. (2013) 'Sequence assembly demystified', in *Nature Reviews Genetics*, March, Vol. 14, No. 3, pp.157–167.

Okuyama, T., Ino, F. and Hagihara, K. (2012) 'A task parallel algorithm for finding all-pairs shortest paths using the GPU', *International Journal of High Performance Computer Networks*, April, Vol. 7, No. 2, pp.87–98.

Olson, C., Kim, M., Clauson, C., Kogon, B., Ebeling, C., Hauck, S. and Ruzzo, W. (2012) 'Hardware acceleration of short read mapping', in *IEEE Symposium on FCCM*, May, pp.161–168.

Peterlongo, P. and Chikhi, R. (2012) 'Mapsembler, targeted and micro assembly of large NGS datasets on a desktop computer', *BMC Bioinformatics*, Vol. 13, No. 1, pp.13–48.

Pevzner, P.A., Tang, H. and Waterman, M.S. (2001) 'An Eulerian path approach to DNA fragment assembly', *Proceedings of the National Academy of Sciences*, Vol. 98, No. 17, pp.9748–9753.

Salmela, L. and Schröder, J. (2011) 'Correcting errors in short reads by multiple alignments', *Bioinformatics*, Vol. 27, No. 11, pp.1455–1461.

Smith, A.D., Xuan, Z. and Zhang, M.Q. (2008) 'Using quality scores and longer reads improves accuracy of Solexa read mapping', *BMC Bioinformatics*, Vol. 9, No. 2, p.128.

Tang, W., Wang, W., Duan, B., Zhang, C., Tan, G., Zhang, P. and Sun, N. (2012) 'Accelerating millions of short reads mapping on a heterogeneous architecture with FPGA accelerator', *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp.184–187.

Varma, B.S.C., Paul, K. and Balakrishnan, M. (2014) 'Accelerating genome assembly using hard embedded blocks in FPGAs', in *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, IEEE, Mumbai, India, 5–9 January, pp.306–311.

Varma, B.S.C., Paul, K., Balakrishnan, M. and Lavenier, D. (2013) 'Fassem: FPGA based acceleration of de novo genome assembly', *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp.173–176.

Wu, X-L., Heo, Y., El Hajj, I., Hwu, W-M., Chen, D. and Ma, J. (2012) 'TIGER: tiled iterative genome assembler', *BMC Bioinformatics*, Vol. 13, No. 12, p.S18.

Xilinx (2015), *Xilinx FPGAs, ISE* [online] http://www.xilinx.com (accessed 2-1-12015).

Zerbino, D.R. and Birney, E. (2008) 'Velvet: algorithms for de novo short read assembly using de Bruijn graphs', *Genome Research*, Vol. 18, No. 5, pp.821–829.

Zhang, W., Chen, J., Yang, Y., Tang, Y., Shang, J. and Shen, B. (2011) 'A practical comparison of de novo genome assembly software tools for next-generation sequencing technologies', *PLoS ONE*, March, Vol. 6, No. 3, p.e17915.