**Computer Engineering**
Mekelweg 4,
2628 CD Delft
The Netherlands
**http://ce.et.tudelft.nl/**

2014

# MSc THESIS

# Hardware Acceleration of BWA-MEM Genome Mapping Application

**Phani Kiran**

**Abstract**

**CE-MS-2014**

Next Generation Sequencing technologies have had a tremendous impact on our understanding of DNA and its role in living organisms. The cost of DNA sequencing has decreased drastically over the past decade, leading the way for personalised genomics. Sequencers provide millions of fragments of DNA (termed short reads), which have to be aligned aginst a reference genome to reconstruct the full information of the DNA molecule under study. Processing short reads requires large computational resources, with many specialised computing platforms now being used to accelerate software aligners. We take up the challenge of accelerating a well know sequence alignment tool called Burrows Wheeler aligner on the Convey Hybrid Computing platform, which has FPGAs as co-processors. The focus of the research is to accelerate the BWA-MEM algorithm of the Burrows Wheeler aligner on the Convey HC-2 platform. The implementation was carried out using the Vivado HLS tool. The architectures proposed are targeted to overcome the memory bottleneck of the application. Two architectures are proposed, the Base architecture and the Batch architecture meant to address the memory bottleneck. Simulations were performed for the intended platform and it was found that, the Batch architecture is 18% faster than the Base architecture for reads with similar run time characteristics. The architectures provide possibilities of further pipelining and implementation of more cores, which is expected to provide better performance than the current implementation.

Faculty of Electrical Engineering, Mathematics and Computer Science

# Hardware Acceleration of BWA-MEM Genome Mapping Application

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Phani Kiran
born in Mysore, India

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

# Hardware Acceleration of BWA-MEM Genome Mapping Application

by Phani Kiran

**Abstract**

Next Generation Sequencing technologies have had a tremendous impact on our understanding of DNA and its role in living organisms. The cost of DNA sequencing has decreased drastically over the past decade, leading the way for personalised genomics. Sequencers provide millions of fragments of DNA (termed short reads), which have to be aligned aginst a reference genome to reconstruct the full information of the DNA molecule under study. Processing short reads requires large computational resources, with many specialised computing platforms now being used to accelerate software aligners. We take up the challenge of accelerating a well know sequence alignment tool called Burrows Wheeler aligner on the Convey Hybrid Computing platform, which has FPGAs as co-processors. The focus of the research is to accelerate the BWA-MEM algorithm of the Burrows Wheeler aligner on the Convey HC-2 platform. The implementation was carried out using the Vivado HLS tool. The architectures proposed are targeted to overcome the memory bottleneck of the application. Two architectures are proposed, the Base architecture and the Batch architecture meant to address the memory bottleneck. Simulations were performed for the intended platform and it was found that, the Batch architecture is 18% faster than the Base architecture for reads with similar run time characteristics. The architectures provide possibilities of further pipelining and implementation of more cores, which is expected to provide better performance than the current implementation.

| | | |
|---|---|---|
| **Laboratory** | : | Computer Engineering |
| **Codenumber** | : | CE-MS-2014 |

**Committee Members** :

| | |
|---|---|
| **Advisor:** | Zaid Al-Ars, CE, TU Delft |
| **Chairperson:** | Koen Bertels, CE, TU Delft |
| **Member:** | Rene Van Leuken, CAS, TU Delft |
| **Member:** | Vlad Mihai Sima, BlueBee, TU Delft |

*Dedicated to my grandmother*

# Contents

# List of Figures

# List of Tables

# Acknowledgements

Masters at TU Delft has been a humbling experience, where I have really had to reform my ways of going about learning. Now, when I am at the final step of my masters studies, I realise that this would not mean the end of my studies but rather, the beginning of the rest of my life as a student. Thanks to TU Delft and CE lab, for I have become a better student.

I would like to convey my gratitude to Dr Koen Bertels, for having provided me with the opportunity to work at BlueBee Multicore Technologies and making me feel part of the BlueBee family. I would also like to thank Dr Zaid Al-Ars, who has been my guide at every step through this experience. I would also like to thank him for providing valuable inputs in writing this document. Special thanks to Dr Vlad Mihai Sima and Giacomo Marchiori, for being my mentors and helping me throughout my thesis. Their ability and work ethic is certainly something I will continue to strive towards. I would also like to thank all the members of the BlueBee team for their support.

I would like to especially thank Adithya Pulli, who has been my friend and mentor, through my masters studies. I would also like to thank Sriram Adiga, for his continued support. Last but not the least, I would like to thank each and everyone of my friends here in Delft, who have been a necessary distraction away from studies. Finally, I would like to thank my family for having the faith in me to pursue studies away from home.

Phani Kiran
Delft, The Netherlands
October 8, 2014

# Introduction 1

Understanding the structure of DNA molecule has been hailed by many, as the most important discovery of the past century. This has paved way towards treating hazardous diseases, as well as identification of genes that trigger major diseases. Advancement in genomic studies has had a tremendous impact on the medical research. This advancement has been largely been accredited towards sequencing technologies. DNA sequencing is used to determine the order of the nucleic acid molecules in the DNA fragment. The initial advancements in sequencing were made in the 1970s with Sanger sequencing dominating the domain, till much of the early 2000s.

In the past decade or so, there has been a tremendous improvement in sequencing technologies, with the cost of sequencing a base halving every five months, while the throughput per machine has increased 500,000 fold and the number of reads per genome increased about 100 fold [6]. This has put tremendous computational pressure on the next stage of DNA processing, which is sequence assembly. The millions of reads which are produced after sequencing, need to be assembled together to make sense out of such quantities of data. The process of assembly has hence proved to be a computational challenge, where millions of reads needs to be processed. Generally, a reference genome is used to map the reads generated from sequencers, which is termed short read alignment.

Many software aligner tools have been developed to address short read alignment problem. The computational complexity and the sheer amount of data that needs to be processed, makes sequence alignment, an interesting challenge in the high performance computing domain. Different platforms such as GPUs and FPGAs have been investigated by many researchers, to speed up the process of short read alignment. In this thesis, we take up such a challenge of investigating the the performance of BWA aligner tool on the Convey hybrid super computing system.

## 1.1 Thesis Objective

The BWA aligner tool, is a well known tool in the bioinformatics research community. BWA recently introduced an algorithm called BWA-MEM, which the authors claim to be better than their previous algorithms, at least for long read alignment [7]. The thesis aims for hardware acceleration of the application, on the CPU-FPGA hybrid Convey computing platform. The thesis aims to perform detailed analysis of the algorithm, identify the performance bottlenecks and aims to come up with solutions, to overcome those bottlenecks. The thesis also adopts Xilinx Vivado High Level Synthesis tool for the hardware implementation of the design, with an objective of increased productivity.

## 1.2 Thesis Organisation

The thesis has been organised into several chapters explaining the intrinsics of the application, identifying opportunities for acceleration, the tools and methodology used during the development process, the solutions and the results obtained from the implementation of those solutions.

In Chapter 2, the background of sequence alignment problem and existing software aligners are mentioned. Also the BWA-MEM application is explained in detail, with the intrinsics of the algorithms used.

In Chapter 3, work related to hardware acceleration of BWA tool by other researchers are described along with the introduction of the Convey computing platform.

In Chapter 4, the application performance is analysed on the target platform, bottlenecks and possible acceleration opportunities are identified. Also the various tools which were developed and used during the course of implementation are described, along with the development methodology followed.

In Chapter 5, the architecture and implementation of the designs are discussed in detail.

In Chapter 6, the results obtained are presented and discussed, with a view of further improvements.

# Background 2

This chapter describes the impact of Next Generation Sequencing technologies in bioinformatics. Section 2.2 introduces the basic sequence alignment problem. We then describe the existing software aligners in section 2.3. Section 2.4 introduces the Burrows Wheeler aligner tool, also providing a detailed explanation of the BWA-MEM algorithm along with specialised data structures used such as FM index and FMD index.

## 2.1 Next Generation Sequencing

DNA sequencing, constitutes of generating data pertaining to the specific order of Adenine, Cytosine, Thymine and Guanine nucleic acid molecules in the strand of DNA. With the growth of sequencing technology, the information about DNA sequences is becoming indispensable to different aspects of biological research such as evolutionary biology, clinical biology, pathology and many more. Early efforts towards sequencing nucleic acids began around 1970s with Gilbert and Maxam [8], coming up with wandering spot analysis technique. In the mid 1970s, a more efficient and sophisticated technique was put forth by Sanger, considered ground breaking enough to be awarded a Nobel prize in Chemistry, 1980 [9].

There has been a significant shift in the past decade, where the bioinformatics community has moved on from the traditional Sanger sequencing to what is now known as Next Generation Sequencing (NGS) or massively parallel sequencing [10]. The impact of NGS technologies has been rightly compared with the impact of personal computers on computing industry. The success of sequencing technology in the recent years, mirrors or even exceeds, that of the well known Moore's law in the computing industry. Figure 2.1 shows the decline of sequencing costs in the past decade against the computing costs predicted by Moore's law [1].

The impetus given by NGS technologies, led to a sanction of large number of ambitious projects such as The 1000 genomes project and, the more recently, \$1000 genome project. The \$1000 Genome project has now alleged to have achieved its objective with Illumina recently announcing to provide that service with its latest release of HiSeq X Ten sequencers [11]. The impact of NGS is touted to be immense on genomics and the trend now leading towards personalised genomics. The emergence of NGS has also brought about a fresh set of computational challenges. To meet these challenges, a new caveat of computational tools are being developed, more specifically towards sequence assembly and reconstruction of genomes. NGS technologies produce shorter reads(fragments of DNA) than Sanger sequencing, but generate a copious amount of data. A single run can produce upto 6 billion reads with the read length varying from 50-150 base pairs, compared to 30 million reads of length 500 bp by Sanger sequencing [12]. The primary requirement for these tools, is being able to map these large number of reads onto a high

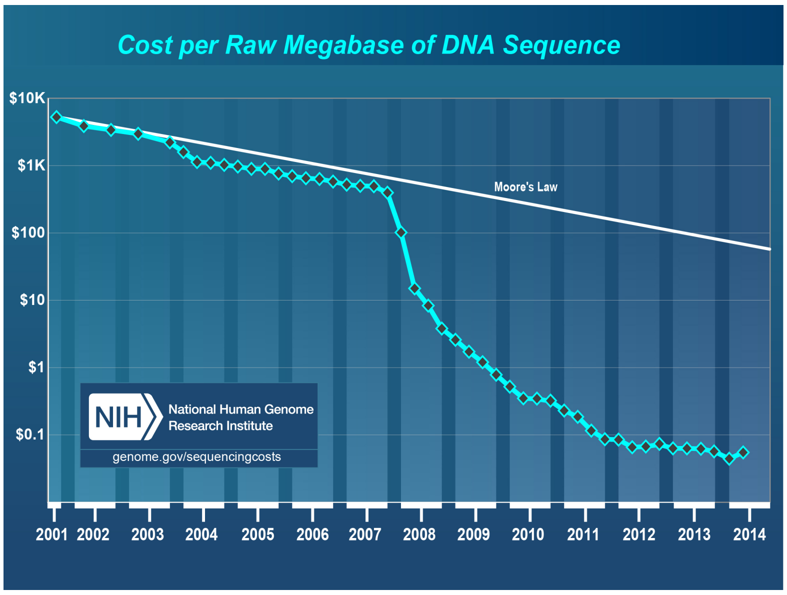


Figure 2.1: Comparison of computation costs against sequencing costs [1].

quality reference genome.

## 2.2 Sequence Alignment

This section gives a brief description of sequence alignment and its various aspects which are relevant towards this thesis. The process of *mapping* the reads obtained, to their correct position on the reference genome is termed as sequence alignment. Genome assemblers are generally classified into two categories.

- **De Novo Assembly**: Assembles reads to create full length sequences by comparing each read against every other read obtained. As such, the naive complexity of the process is $O(n^2)$ and requires a lot of memory and computation.
- **Reference based Assembly**: Assembles the reads obtained by comparing against a reference sequence and finding out the reads' approximate position on the reference. The complexity of the reference based assembly remains linear with O(n), since each read is compared against the constant reference.

To understand the process of mapping, few properties of the DNA sequence along with some features of the sequencing process itself need to be highlighted.

### 2.2.1 Exact Matching

The problem of sequence mapping is a string matching problem, where we have millions of reads being matched against a reference sequence. Consider a reference sequence termed R, and a query sequence Q, as illustrated in Figure 2.2. The mapping of the query sequence, Q, against R would be to find out the exact positions where the query would fit into the reference. The query Q is located at position 2 in the reference sequence. In this case, aligning the sequence Q against R is relatively simple, since we find a unique *exact match* of the sequence in the reference.

```
POSITION       : 0  1  2  3  4  5  6  7
REFERENCE R    : A  G  T  G  C  C  A  C
QUERY Q        :       T  G  C
```

Figure 2.2: Alignment of query with reference without differences.

### 2.2.2 Inexact Matching

However, there are a few problems introduced when we consider the practicalities of sequencing. Sequencing is error prone and sequence errors may be present in the process, which may cause some erroneous bases being introduced into the read. Also, mutations could be present in the DNA strand which is being sequenced and hence may be inherently different from the reference. These are the differences which we are interested in finding out, since they reveal the fundamental anomalies in the genetic make-up of the individual. These factors need to be considered when mapping the read to a reference strand.

Consider the original query sequence Q with the base pair 'G' being changed to 'A', either by process error or mutation, as shown in Figure 2.3. The mapping tool should still be able to point to the correct position on the reference strand. The term *mismatch* is used to denote such an occurrence. Hence, the mapping should allow mismatches to occur, to detect such conditions and still operate accurately. Such an alignment is termed *inexact matching* where mismatches are allowed for alignment.

```
POSITION       :  0  1  2  3  4  5  6  7
REFERENCE R    :  A  G  T  G  C  C  A  C
QUERY Q        :        T  A  C
MISMATCH       :           ↑
```

Figure 2.3: Alignment of query with reference with mismatch.

There is a possibility of one more such case, where inexact matching should be considered. The sequence Q, as shown in Figure 2.4 has an extra base being inserted during sequencing. If the mapping allows only one difference, the process would not be able to find the correct mapping. The extra base pair 'A' is actually considered an insertion to the read with respect to reference or a deletion from the reference in relation to the read. The two terms are inter exchangeable and are commonly referred to as indels or gaps.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| POSITION | : | 0 | 1 | 2 | 3 | | 4 | 5 | 6 | 7 |
| REFERENCE R | : | A | G | T | G | | C | C | A | C |
| QUERY Q | : | | | T | G | A | C | | | |
| INSERTION | : | | | | | ↑ | | | | |

Figure 2.4: Alignment of query with reference with insertion.

The quality of an alignment, therefore, is calculated based on a scoring system where mismatches, gaps and indels are penalised while matches are rewarded. For a given position a score is computed and the read with the best score is considered to be correct.

There are other constrictions on the problem as well, brought upon by the nature of the DNA sequence. These challenges posed by sequence mapping can be summarised as shown.

- The reference sequences are generally large and they themselves pose a huge burden on the computational resources available.
- Many short reads have to be aligned to the reference.
- The reads themselves have variations due to sequencing errors or mutations, which causes additional computational complexity.
- The DNA sequences themselves are very repetitive in nature. Nearly half of the human genome is found to be repetitive [12]. This creates ambiguities during the mapping process, and although it does not affect the mapping itself, it may lead to misinterpretation of results.

## 2.3 Existing Aligners

For simplicity, we consider in this section only the current software aligners which dont use any special execution platforms for their execution. Hatem et al provides a very comprehensive overview and comparison of the existing DNA short sequence aligners [13]. He broadly classifies the aligners based on the technique they use for index construction of the reference sequence. Index construction is the process of representing the reference sequence in a convenient way, for the benefit of the algorithm either in terms of speed or size. Before proceeding further, we describe here a set of features or terms, which are synonymous with alignment tools and these terms will be used henceforth in this thesis colloquially.

- Seeding : Seeds are the first few base pairs of the reads, which are expected to contain less errors.

- Base quality scores : These scores give an indication towards the quality of base pair characters in the read. The quality score is -10*log e where 'e' is the probability that the base is wrong [13].
- Paired end reads : These are reads which are produced from sequencing from both ends of the DNA strands. They are generally used to provide some degree of confidence in the mapping process since we can estimate the distance between 2 bases across the ends of the reads.
- SNP : Single Nucleotide Polymorphisms are the variations of single nucleotide between members of the same species and are not considered as mismatches during the mapping process.

Mapping tools are usually based on constructing an index for the reference sequence and using the index to map the short reads onto the reference. Based on the methodology used to construct the index, the tools has been broadly classified into two categories.

### 2.3.1 Hash Table Index Aligners

These aligners construct hash tables either for the reads or the genome and use them to index the reference sequence in an efficient manner. The key of each entry of the table is a read or a subsequence of genome and the value is a list of positions where the reads/subsequences match. Tools based on hash table construction are described briefly below.

- **GSNAP** [14]: The hash tables are constructed for the genome and each read is divided further into subsequences. These are used to find the matching positions on the reference using the hashing technique. Finally all the positions for the subsequences are compiled together to find a final alignment.
- **Novoalign**: Novoalign is similar to GSNAP where the genome is hashed by overlapping oligomers and the reads are mapped to the reference using the hash tables.
- **FANGS** [15]: is also another genome hashing based mapping tool where it was specifically designed to handle long reads generated by 454 sequencer.
- **MAQ** [16]: is a read indexing tool where multiple hash tables are generated from the reads and the reference genome is matched against the multiple hash tables. It is different from the previously mentioned tools in that the reads are hashed as opposed to the reference genome.

### 2.3.2 BWT Index Aligners

The following tools described use the Burrows Wheeler transform for index construction.

- **BWA**: Burrows Wheeler aligner is a siftware sequencer developed by Heng Li, which utilises BWT based index construction for reference sequences. Specialised data structures such as FM index and FMD index are used for efficient indexing of reference. The latest algorithm employed by the tool called BWA-MEM is chosen for acceleration in this thesis.

- **Bowtie** [17]: The reference genome is transformed to FM index and the short reads are mapped against the FM index using the modified FM index searching algorithm proposed by Ferragina and Manzini [18]. There are two versions of the tool available Bowtie and Bowtie2, with the second version designed to handle longer reads.

- **SOAP2** [19]: is a unique tool in the sense it uses both BWT and hash tables to speed up the searches for mapping locations on the reference. It also utilises a split-read strategy to find the inexact searches on the genome.

## 2.4 Burrows Wheeler Aligner Tool

Burrows Wheeler Aligner Tool or BWA, is a software aligner designed to align low variant reads against reference genomes. Three algorithms are employed by BWA, namely, *BWA-backtrack*, *BWA-SW* and *BWA-MEM*. The first algorithm was exclusively designed for aligning short reads upto 100bp, while the latter two can handle read lengths upto mega bases. The author recommends the use of the BWA-MEM algorithm for the availability of latest features and performance enhancements compared to the former ones. The thesis, hence, concentrates on BWA-MEM algorithm and this section attempts to provide a detailed description of the fundamental concepts related to the BWA-MEM algorithm.

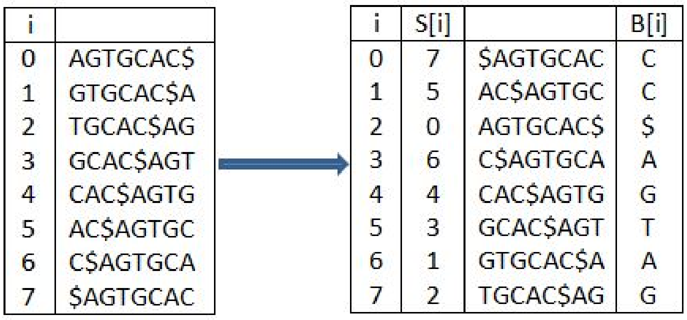
BWA, fundamentally, is a BWT index based aligner tool, which makes use of a special data structure called *FM index*, for improved accuracy and performance. BWA-MEM specifically uses an extension of the FM index termed FMD index. We proceed to describe briefly the BWT transform, FM index components and the newly introduced FMD index. We then explain the BWA-MEM algorithm and its components.

### 2.4.1 Burrows Wheeler Transform

Burrows wheeler transform was first proposed in 1994, by M. Burrows and D.J. Wheeler, as a block sorting lossless data compression algorithm [20]. BWT rearranges the string into runs of similar characters and produces a transformation which is *reversible*. The reversibility of the transformation is a key property, which ensures, the data can be retrieved without any additional data being stored. The transformation is performed by producing all possible rotations of the string and sorting them in a lexicographical order.

Consider the alphabet set $\Sigma = \{$'A', 'C', 'G', 'T'$\}$, which forms the base pairs of DNA. A special character termed '\$' is used to indicate the end of a string and also is lexicographically smaller than all the letters of the alphabet set. Consider a sample string G = "AGTGCAC". The special character '\$' is first inserted to produce the string "AGTGCAC\$". For the transformation, all rotations of the string are performed and placed in an array. Now the array has all possible suffixes of the string. Figure 2.5 shows the array with all possible suffixes on the right side table.

The array is then sorted in a lexicographical order to produce a sorted suffix array of the original string. The transformation now takes the last column of the sorted array, to



| i | |
|---|---|
| 0 | AGTGCAC$ |
| 1 | GTGCAC$A |
| 2 | TGCAC$AG |
| 3 | GCAC$AGT |
| 4 | CAC$AGTG |
| 5 | AC$AGTGC |
| 6 | C$AGTGCA |
| 7 | $AGTGCAC |

| i | S[i] | | B[i] |
|---|---|---|---|
| 0 | 7 | $AGTGCAC | C |
| 1 | 5 | AC$AGTGC | C |
| 2 | 0 | AGTGCAC$ | $ |
| 3 | 6 | C$AGTGCA | A |
| 4 | 4 | CAC$AGTG | G |
| 5 | 3 | GCAC$AGT | T |
| 6 | 1 | GTGCAC$A | A |
| 7 | 2 | TGCAC$AG | G |

Figure 2.5: BWT transformation process.

generate the BWT string termed B. B is the Burrows Wheeler transform of the original string G.

### 2.4.2 FM index

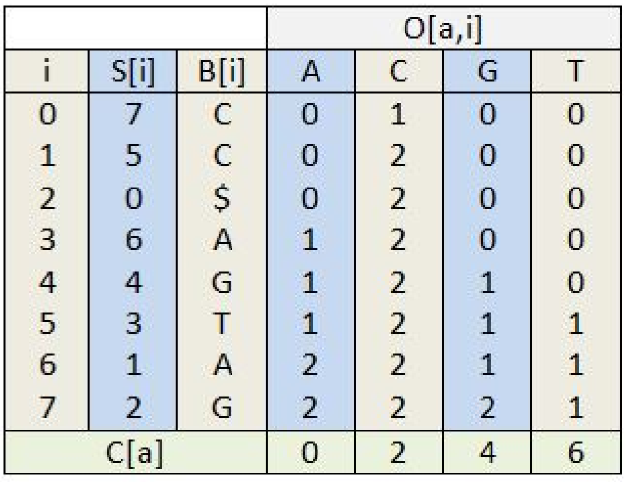
To search for a substring R = "GTG", we lookup the sorted suffix array for the character 'G' and by visual inspection, arrive at the interval [5,6] of the sorted suffix array table. We then proceed to the second character 'T' which narrows the interval to [6,6] which is where the substring lies. One more character 'G' is also considered, which results in the same interval. The suffix array value for the corresponding interval is found to be 1, which is the position of the substring being considered in the example.

Formally, the suffix array S, is a permutation of integers between 0 and |G |- 1, where S[i] gives the position of the i$^{th}$ smallest suffix of G. For a given string P, the suffix array interval or the *SA interval*, denoted as [$I^l$(P), $I^u$(P)], gives the lower and upper bounds in the suffix array, within which bounds, P is found to be the suffix of G. The BWT string is computed as B[i] = G[S[i] - 1] for S[i] >0 and B[i] = $ otherwise.

This procedure is nothing but traversing through a suffix tree, where the tree is being represented by the sorted suffix array. The disadvantage of using a suffix tree is the memory footprint of the tree. For genomes the tree uses 15 bytes per base pair which results in more than 50GB of data [21]. Based on the BWT transformation, Ferragina and Manzini [18] proposed an algorithm to search the compressed string without completely decompressing it [21] using some additional data.

FM index of a reference string G, consists of the following components.

- **S**: the sorted suffix array
- **B**: the BWT array of the string, obtained as shown in section 2.4.1.
- **O**: the occurrence array, O[a,i], which is obtained by calculating the occurrence of character $a$ in $\Sigma$ upto i$^{th}$ position of B, which is shown in Figure 2.6.
- **C**: the count array where C[a] denotes the number of characters in G which lexicographically smaller than $a$.



| | | | O[a,i] | | | |
|---|---|---|---|---|---|---|
| i | S[i] | B[i] | A | C | G | T |
| 0 | 7 | C | 0 | 1 | 0 | 0 |
| 1 | 5 | C | 0 | 2 | 0 | 0 |
| 2 | 0 | $ | 0 | 2 | 0 | 0 |
| 3 | 6 | A | 1 | 2 | 0 | 0 |
| 4 | 4 | G | 1 | 2 | 1 | 0 |
| 5 | 3 | T | 1 | 2 | 1 | 1 |
| 6 | 1 | A | 2 | 2 | 1 | 1 |
| 7 | 2 | G | 2 | 2 | 2 | 1 |
| C[a] | | | 0 | 2 | 4 | 6 |

Figure 2.6: FM index components.

#### 2.4.2.1 Exact Matching using FM index

Using the components of the FM index, an exact match of a substring can be obtained in minimal time, by utilising a key property of FM index.

$$I^{\mathrm{l}}[aP] = C[a] + O[a, I^{\mathrm{l}}[P] - 1] \tag{2.1}$$

$$I^{\mathrm{u}}[aP] = C[a] + O[a, I^{\mathrm{u}}[P]] - 1 \tag{2.2}$$

$$I^{\mathrm{s}}[P] = I^{\mathrm{u}}[P] - I^{\mathrm{l}}[P] + 1 \tag{2.3}$$

The equations 2.1 and 2.2 can be used iteratively, to provide for backward search of a substring on a reference string. The equations are applied until $\mathrm{I}^{\mathrm{l}}$[P] <$\mathrm{I}^{\mathrm{u}}$[P], which represents a valid interval in the suffix array.

**Algorithm 1**: Exact matching using FM index [21]

```
Function ExactMatch(R, C, O) begin
    $i \leftarrow |R| - 1$
    $k \leftarrow 0$
    $l \leftarrow |G| - 1;$
    while $k \leq l \cap i \geq 0$ do
        $\sigma \leftarrow R[i]$
        $k \leftarrow C[\sigma] + O[\sigma, k - 1] + 1;$
        $l \leftarrow C[\sigma] + O[\sigma, l];$
        $i \leftarrow i - 1;$
        end while
        if $k \leq l$ then
            return $k, l$
        else
            return $\phi$
        end if
    end function
```

The algorithm shown above, provides the details for the backward search procedure. The algorithm makes use of the equations above to back extend the string, with the last character of the string being considered first. The interval is first initialised to [0, |G|- 1], and above equations are applied iteratively, by performing one back extension at a time. The procedure is continued until there is a valid interval present for each extension or when the entire string is covered by back extension. One useful property of the algorithm is that, it has a linear time complexity dependent on the length of the read. It does not depend on the size of the reference sequence, which is extremely advantageous for sequence alignment.

#### 2.4.2.2 Memory Optimisation

The memory footprint of FM index is still a concern since, the occurrence array alone can occupy upto GBs of size when stored as 32-bit data format [21]. Li and Durbin [22] managed to devise methods to reduce the memory footprints of the data structures drastically, without the need for further compressing the structures. Li and Durbin proposed to store only part of the Suffix and Occurrence arrays in memory, and calculate the rest on the fly. The occurrence array, O[a,i], is stored in the memory only for i being a factor of 128. To calculate the actual occurrence value for particular character and index, the number of occurrences of the character in the BWT string B, from the previously stored 128 factor boundary, is counted upto the current index.

A similar technique, proposed by Grossi and Vitter [23], is used to reduce the footprint of the Suffix array in memory. An inverse compressed suffix array is used, along with Occurrence array and B, to calculate the suffix array. The actual relation is however out of scope for the thesis. S[i], for i being a factor of 32, will be stored in memory in this case. These optimisations result in a significant reduction in memory usage, with approximately n bytes used for a genome of size n, when genomes are less than 4Gb.

### 2.4.3 FMD index

When the DNA molecule is sequenced, the read can be taken from either the forward or the reverse strand. With FM index, the matching of reads has to be performed multiple times, to check whether the read actually belongs to the forward or the reverse strand and also to verify if the complemented read can map onto the reverse complemented strand. Initially, BWA performed the complemented read search on FM index and hence reverse reference also had to be included. Li et al [24] proposed to incorporate few modifications to the FM index structure to ease the matching of complemented reads and the use of reverse reference strands. The new data structure is termed FMD index and is used in *BWA-MEM*, the application under consideration for this thesis.

FMD index is calculated by concatenating the reference sequence by its reverse complement and calculating the FM index for the combined sequence. For example, considering our previous example of G = "AGTGCAC", the reverse complement of the sequence would be G' = "GTGCACT". G and G' are concatenated to obtain one single sequence, G" = "AGTGCACGTGCACT", and FM index is now calculated on the new sequence. This provides the possibility of matching the read against the forward and reverse strands in a more effective way. FMD index exhibits the useful property of symmetry and bi-directionality.

#### 2.4.3.1 Exact matching using FMD index

The exact search makes use of the symmetry of FMD index. *Bi-interval* of a substring P, is used instead of the suffix array interval of FM index. The Bi-interval of a string P is denoted as [$I^l$(P), $I^l(\bar{P})$, $I^s$(P)], the meaning of which will be explained shortly. $I^l$(P), holds the same meaning as it did with FM index, and provides the lower boundary of P to be the suffix of G in the suffix array. The relation for $I^s$(P) is shown in equation 2.3. $I^l$(P) and $I^s$(P), essentially provide the same functionality as the SA interval of FM index. $I^l(\bar{P})$,however, is simply the lower boundary of the complemented read in the suffix array. When dealing with $\bar{P}$, we note that $\bar{a}P = \bar{P} \o \bar{a}$, where ø is denoted string concatenation, and due to this symmetry, $I^s(P) = I^s(\bar{P})$. This allows for $I^l(\bar{P})$ and $I^s$(P), together, to provide the SA interval for the complemented read in the suffix array.

**Algorithm 2**: Backward extension [24]

```
Input: Bi-interval $[k, l, s]$ of string $P$ and a symbol $a$
Output: Bi-interval of string $aP$

Function BackwardExt([$k, l, s$],a) begin
    for $b \leftarrow 0$ to $5$ do
        $k_b \leftarrow C(b) + O(b, k-1)$
        $s_b \leftarrow O(b, k+s-1) - O(b, k-1)$
    $l_0 \leftarrow l$;
    $l_4 \leftarrow l_0 + s_0$;
    for $b \leftarrow 3$ to $1$ do
        $l_b \leftarrow l_{b+1} + s_{b+1}$
    $l_5 \leftarrow l_1 + s_1$;
    return $[k_a, l_a, s_a]$
```

The authors propose that backward extension of P would result in forward extension of $\bar{P}$ and forward extension of P, would mean backward extending $\bar{P}$. Just like the algorithm for exact matching with FM index which uses backward extension, the algorithms shown below provides for the backward and forward extension for calculating bi-intervals and exact matches.

**Algorithm 3**: Forward extension [24]

```
Input: Bi-interval $[k, l, s]$ of string $W$ and a symbol $a$
Output: Bi-interval of string $Wa$

Function ForwardExt($[k, l, s]$,a) begin
    $[l', k', s'] \leftarrow$ BackwardExt($[l, k, s]$,$\bar{a}$);
    return $[k', l', s']$
```

### 2.4.4 BWA-MEM

*BWA-MEM* [7] is the latest algorithm released by the authors of BWA alignment tool. As NGS technologies grow, the reads are not short any more, and especially for 100bp or more it becomes necessary for the algorithm to remain robust to sequencing errors. Many short read alignment tools are not very suited for these kind of reads. BWA-MEM is designed for efficient long read alignment, with the length of the reads ranging from 70bp to few megabases. The section describes BWA-MEM algorithm components in detail.

#### 2.4.4.1 Super Maximal Exact Matches

*Maximal Exact Match*(MEM) is an exact match that cannot be extended in either forward or backward direction. *Super Maximal Exact Matches*(SMEM) are the MEMs which are not found in other MEMs. BWA-MEM requires that, at any query position, the longest exact match covering the position must be a SMEM. SMEMs are found very quickly using the FMD index and the related backward-forward extension algorithms. The algorithm for finding out the SMEMs present in the query is shown below.

**Algorithm 5**: Finding SMEMs [24]

```
Input: String $P$ and start position $i_0; P[-1] = 0$
Output: Set of bi-intervals of SMEMs overlapping $i_0$

Function SuperMEM1($P, i_0$) begin
    Initialize Curr, Prev and Match as empty arrays;
    $[k, l, s] \leftarrow [C(P[i_0]), C(\overline{P[i_0]}), C(P[i_0] + 1) - C(P[i_0])]$;
    for $i \leftarrow i_0 + 1$ to $|P|$ do
        if $i = |P|$ then
            Append $[k, l, s]$ to Curr
        else
            $[k', l', s'] \leftarrow$ForwardExt($[k, l, s], P[i]$);
            if $s' \neq s$ then
                Append $[k, l, s]$ to Curr
            if $s' = 0$ then
                break;
        $[k, l, s] \leftarrow [k', l', s']$
    Swap array Curr and Prev;
    $i' \leftarrow |P|$;
    **for** $i \leftarrow i_0 - 1$ **to** $-1$ **do**
        Reset Curr to empty;
        $s'' \leftarrow -1$;
        **for** $[k, l, s]$ **in** Prev **do**
            $[k', l', s'] \leftarrow$BackwardExt$([k, l, s], P[i])$;
            **if** $s' = 0$ **or** $i = -1$ **then**
                **if** Curr is empty **and** $i + 1 < i' + 1$ **then**
                    $i' \leftarrow i$;
                    Append $[k, l, s]$ to Match
            **if** $s' \neq 0$ **and** $s' \neq s''$ **then**
                $s'' \leftarrow s'$;
                Append $[k, l, s]$ to Curr
        **if** Curr is empty **then**
            **break**
        Swap Curr and Prev;
    **return** Match
```

BWA-MEM uses the traditional *seed-and-extend* paradigm and SMEMs, found from the above algorithm, are used to seed an alignment. Seeds are parts of the reads used initially, to evaluate an alignment. Sometimes, SMEMs are not present in true alignments. To reduce the mismappings caused by this, *re-seeding* is performed, where, suppose we have an SMEM of length $l$ with $k$ occurrences in the reference. If $l$ is too large, re-seeding is performed with longest exact matches covering the middle base of the SMEM, with at least *k+1* occurrences. Such seeds can be found with minor modification to the SMEM finding algorithm.

#### 2.4.4.2 Chaining and chain filtering

The seeds obtained from the above procedure are than sent through a procedure of *chaining*. Groups of seeds which are close to each other and co-linear are termed *chains*. The chains are then filtered out based on the chain lengths, the shorter chains being removed, and whether they are contained in any of the other longer chains. Chain filtering proves to reduce unsuccessful seed extensions later on.

#### 2.4.4.3 Seed extension

Extension with banded dynamic programming (DP) is performed against a ranked list of seeds. Prior to extension, the seeds are first ranked based on the chain length and then by the seed length. Each seed, from best ranked to the worst, is checked if it is present in another alignment and if not, is extended with banded dynamic programming. Banded dynamic programming is implemented as a SSE2 based Smith-Waterman algorithm proposed by Farrar[25]. A couple of heuristics are applied during the extension phase of the algorithm.

- Banded DP may align through bad intermediate regions if the flanking alignment scores are high. Z-dropoff heuristic is employed to overcome this, where the alignment is stopped if the score drops below certain threshold from the best score. The Z-dropoff is similar to X-dropoff, except when it comes to long gaps, where the Z-dropoff does not penalise long gaps present in reference or query.

- BWA-MEM keeps record of the best alignment score reaching the end of query. If the difference between the end alignment score and local alignment score is below a certain threshold, the local alignment is dropped even if it has the higher score, there by giving a bonus to the extension reaching the end.

The BWA-MEM algorithm can be summarised as shown in Figure 2.7.

This section was mainly concerned with the algorithmic aspects of the application. BWA-MEM presents an interesting opportunity for acceleration, since the sheer amount of data involved is large and the application does not fit into any straight forward acceleration paradigm. We perform further analysis on the practical implementation of BWA-MEM, profile the application and deduce relevant information required, for any kind of acceleration opportunities which can be exploited.

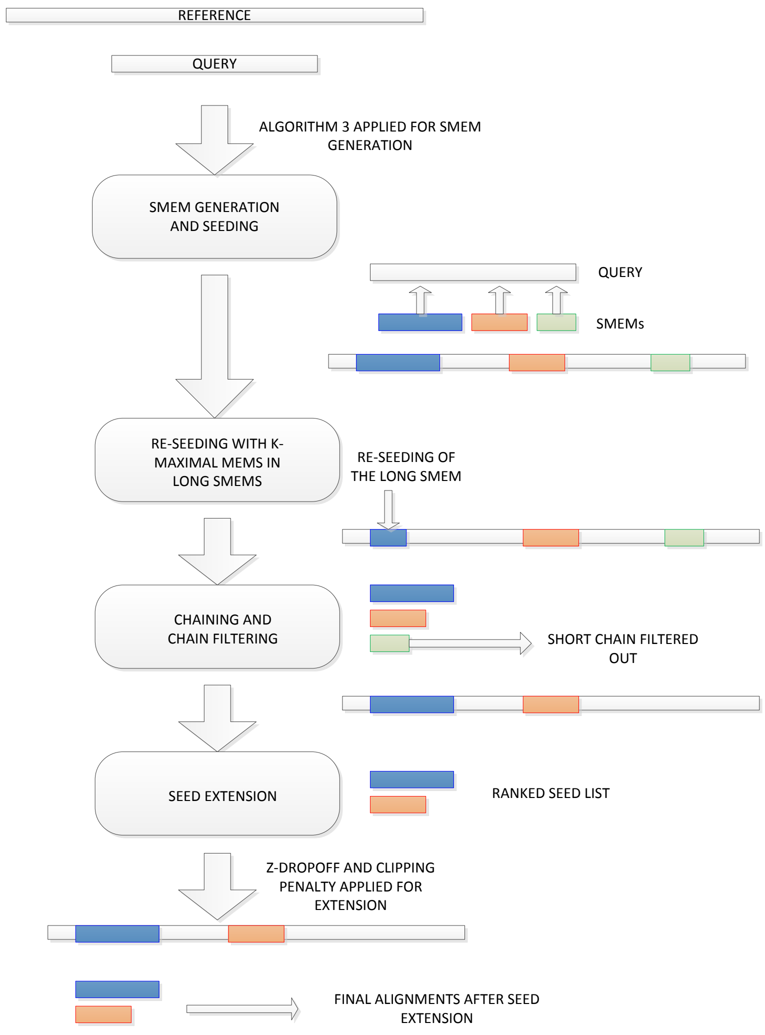

REFERENCE
QUERY
ALGORITHM 3 APPLIED FOR SMEM GENERATION
SMEM GENERATION AND SEEDING
QUERY
SMEMs
RE-SEEDING WITH K-MAXIMAL MEMS IN LONG SMEMS
RE-SEEDING OF THE LONG SMEM
CHAINING AND CHAIN FILTERING
SHORT CHAIN FILTERED OUT
SEED EXTENSION
RANKED SEED LIST
Z-DROPOFF AND CLIPPING PENALTY APPLIED FOR EXTENSION
FINAL ALIGNMENTS AFTER SEED EXTENSION

Figure 2.7: BWA-MEM algorithm [2].

# Related Work and Platform 3

Previous chapter introduced the sequence alignment problem and BWA-MEM algorithm. We proceed to study the research works from several authors related to acceleration of sequence alignment algorithm. To satisfy the ever growing demands from the application, wide variety of platforms have been used to obtain performance gains from multi cores,GPUs to FPGAs. Of all these parallel platforms, FPGAs provide the maximum flexibility in design space. Hence we restrict ourselves specifically to implementations of alignment tools on FPGA platforms.

## 3.1 Hardware Acceleration of BWA

The works, described in this section, address different algorithms of the BWA tool as well as different aspects of those algorithms. The first implementation of BWT based string matching on FPGA was proposed in [26]. The paper discusses, BWT generation as well as exact matching of string patterns on FPGA. It also proposes an architecture for exact matching using FM-index, with the FM-index components stored on FPGA itself. This increases the resource utilisation, with majority of resources allocated for FM-index storage. For a pattern length of range 36 to 108, speedup of upto 196 is reported, when compared with Bowtie software tool.

Xin et al [27], proposes a complete solution including exact and inexact matching for FM-index on hardware. Although, Fernandez et al [26], developed a simple and effective architecture, it does not cover the inexact matching paradigm. Computational complexity and resource utilisation increases exponentially for inexact matching when compared to exact matching. [27] was able to develop architectures for the exact matching algorithm, shown in section 2.4.2.1, as well as the inexact matching algorithm of [22]. For the case of inexact matching, it proposes the use of hierarchical tables instead of prefix tries. The architecture proposed is quite complex with a fair amount of resource usage. Another key factor of their design is that, they only concentrate on SA interval calculation rather than the entire alignment process. With read lengths of around 70 bp, the performance was comparable with CPU platforms with speedups in the range of 1-2.

Arram et al [28] proposes a backtracking version of FM-index algorithm, supporting both exact and inexact matches. They propose the use of a bi-directional FM-index, to support forward and backward search. The architecture also provides scope for memory optimisations, bandwidth utilisation and latency hiding. Latency hiding technique proposed by them, is particularly interesting, which tries to process $l$ reads in a pipelined fashion, one base at a time. For better bandwidth utilisation, they also propose interleaving BWT string and occurrence arrays in memory. Experiments were perormed on MAX3 board, with a Xilinx Virtex-6 FPGA, with a reported performance gain of 3.5 to

| Papers | [26] | [27] | [28] | [29] |
|---|---|---|---|---|
| Complexity | Medium | High | High | High |
| Supported | Exact search | Exact and Inexact search | Exact and Inexact search | Exact and Inexact search |
| Architectural Features | None | Hierarchical Table | Memory interleaving and Batch processing | Multi threaded implementation |
| Performance gains | 2x | 1-2x | 3.5-7x | 70x |

Table 3.1: Comparison of different architectures proposed for BWA hardware acceleration.

7 compared to the software versions on Intel Xeon CPU.

The work in [29] is not directly related to acceleration of BWA, but since it is related to exact and inexact searching using FM-index, it is worth describing. Najjar et al [29], proposes a multithreaded architecture to hide the memory latency which are inherent in the FM-index search algorithm. The sequencing application used here is another BWT based mapping tool, bowtie [17]. The architecture has several matching engines operating in parallel, in a multithreaded manner. The solution was implemented on a Convey HC-1 platform, and experiments reported speedups upto a factor of 70 with a fair amount of resource utilisation.

The key features of these works and performance have been summarised below in Table 3.1.

## 3.2 Platform

Heterogeneous computing has come to the forefront in the recent years, with GPGPUs and FPGAs used along with general purpose CPUs to provide some degree of application specific performance enhancement. Convey provides a such hybrid computing platform, where FPGAs act as reconfigurable coprocessors to the Intel Xeon CPUs. Such heterogeneous platforms not only provide the opportunity to generate application specific logic, but also delivers higher performance per watt, since FPGAs have relatively lesser power consumption characteristics. The thesis is aimed for Convey Hybrid Core-2 platform which combines Xilinx Virtex-5 and Virtex-6 FPGAs with Intel Xeon CPU modules. The Intel processors for the *host* system, which run on industry standard Linux, perform just as any other commodity servers. The co processor FPGAs run application specific operations which are termed *personalities*, aimed to reduce the application run time bottlenecks. These personalities can be invoked from software running on host, via instruction set extensions. Convey provides the development environment to develop application specific instruction set extensions. The Convey system architecture which is shown in Figure 3.1, is explained further in the following subsections.

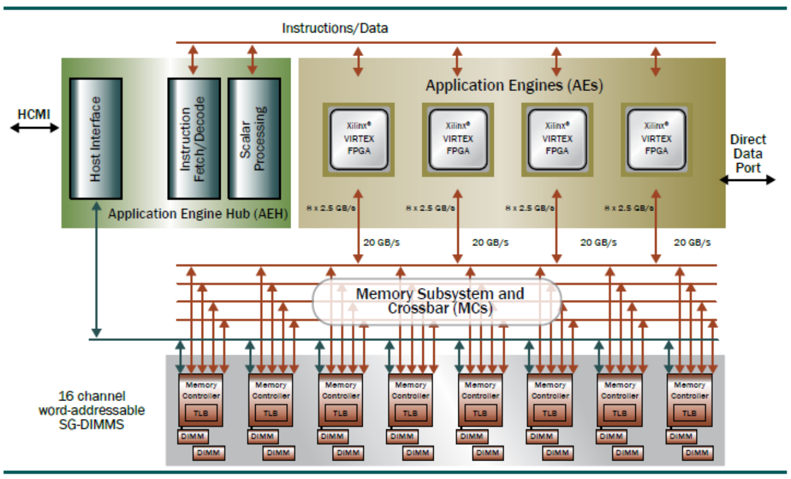


Figure 3.1: Convey system architecture [3].

### 3.2.1 Coprocessor

Coprocessor system consists of an Application Engine Hub (AEH), Memory subsystem (MCs), and the Application Engines (AEs). The AEH remains as the control unit, which implements the interconnect to the host system, also fetches and decodes the instructions. It also acts as the controller of memory requests going to the Application Engines from hosts or the other way around. The instructions are grouped into two categories, as canonical instructions and extended instructions. Canonical instructions are decoded in the AEH while the extended instructions are passed onto the AEs.

### 3.2.2 Memory Subsystem and Crossbar

The memory subsystem has total of eight memory controller supporting 16 DDR2 memory channels and a bandwidth of upto 80GB/sec between each AE and the coprocessor memory. The MCs supports scatter-gather DIMMs , which supports fine granularity memory accesses, with accesses to the size of individual words, maintaining a near peak bandwidth for non sequential accesses. Hence not only does the MCs provide higher bandwidth than what is available to the commodity processors but also deliver a high bandwidth for non sequential accesses.

### 3.2.3 Application Engines

The Application Engines implement the extended instructions and form the core of a personality. Convey supports four such AEs connected to the AEH. A command bus forms the interconnection between them delivering commands and opcodes to the AE. The AE provides flexibility in design and a high degree of parallel functional units which overcome the deficit of lower clock rates.

### 3.2.4 Hybrid-core Globally Shared Memory

Even though there are are two separate physical memories, one on the host side and another on the coprocessor, Convey supports a global shared memory paradigm, which makes it possible for the host to access the coprocessor memory and vice versa, using the virtual addresses themselves. The system hardware is responsible for memory protection, coherence and address translation. Code generated for the host processor uses a load/store approach, which makes it easier for programming the system. Hybrid-core Globally Shared Memory is implemented upon the PCIe physical connection. During code generation, the Convey compiler inserts the necessary procedures and interfaces necessary to start appropriate personality and control its execution.

The platform used in the thesis contains two Intel Xeon E5-2643 processors with a 128 GB DDR3 memory, acting as host processors. The coprocessor system contains four Xilinx Virtex-6 LX760 FPGAs, which has a coprocessor memory of 64 GB. Full utilisation of memory channels provides a maximum memory bandwidth of 80 GB/s. Having introduced the application, in the previous chapter, and the targeted platform, we proceed to the analysis of the application and its profile. We aim to identify the bottlenecks and how the target platform features can best be utilised in order to overcome them.

# 4 Tools, Application analysis and Methodology

The previous chapters provide a detailed description of the application as well as the target platform. This chapter aims to provide profiling results and perform an in-depth analysis of the application performance on the target platform. Section 4.1 describes the various tools used for development, including the standard tools available as well as a few helper tools developed. Section 4.2 describes the application and its profile result. The methodology followed during the development is presented in section 4.3.

## 4.1 Tools

We now describe the tools which were used for the implementation of our designs. The high level synthesis tool Xilinx Vivado 2013.4 was used for the implementation of our designs and Modelsim 10.2a was used for our simulations. In this section, we briefly describe these tools as well as helper tools which were created for the ease of development and analysis such as code instrumentation, memory delay wrappers and hardware profiling tools.

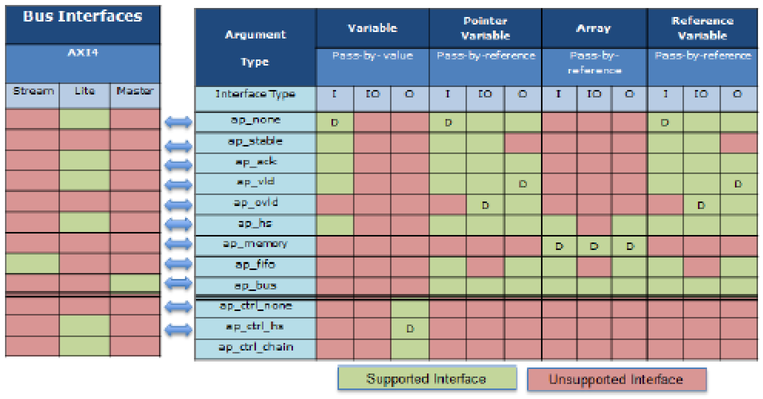
### 4.1.1 Vivado HLS

Xilinx Vivado HLS tool is part of the Xilinx Vivado Design Suite, and is aimed at creating RTL implementations directly from the software specifications. Vivado HLS takes input C, C++ or SystemC specifications and converts them to synthesisable RTL, which is compatible with Xilinx FPGAs. Vivado HLS accelerates development time from algorithm to IP, reduces IP verification time and supports pragma driven optimisations. Significant advantage of high level synthesis is the automated testbench generation, IP simulation and verification. Although the IP developed may not be as optimal as a hand written RTL, the availability of optimisation pragmas and the opportunity to perform accelerated high level design space explorations, makes it possible to come up with relatively good implementations.

We introduce a few significant features of Vivado HLS, which are relevant to our implementations.

- Interface synthesis: In RTL, the I/O operations should be performed by specific interface ports adhering to respective I/O protocols. Vivado HLS supports interface synthesis, which generates standard port interfaces based on the software arguments. The interfaces supported by Vivado HLS are shown in Figure 4.1.

- Function optimisations: Vivado HLS produces a hardware instantiation for each function by default. If the same function is called multiple times, the same hardware instantiation will be used multiple times. But the HLS also provides many



| Bus Interfaces | | | Argument Type | Variable | | | Pointer Variable | | | Array | | | Reference Variable | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AXI4 | | | | Pass-by- value | | | Pass-by-reference | | | Pass-by-reference | | | Pass-by-reference | | |
| Stream | Lite | Master | Interface Type | I | IO | O | I | IO | O | I | IO | O | I | IO | O |
| 🟥 | 🟩 | 🟥 | ap_none | 🟩 D | 🟥 | 🟥 | 🟩 D | 🟩 | 🟩 | 🟥 | 🟥 | 🟥 | 🟩 D | 🟩 | 🟩 |
| 🟥 | 🟥 | 🟥 | ap_stable | 🟩 | 🟥 | 🟥 | 🟩 | 🟩 | 🟥 | 🟥 | 🟥 | 🟥 | 🟩 | 🟩 | 🟥 |
| 🟥 | 🟩 | 🟥 | ap_ack | 🟩 | 🟥 | 🟥 | 🟩 | 🟩 | 🟩 | 🟥 | 🟥 | 🟥 | 🟩 | 🟩 | 🟩 |
| 🟥 | 🟩 | 🟥 | ap_vld | 🟩 | 🟥 | 🟥 | 🟩 | 🟩 | 🟩 D | 🟥 | 🟥 | 🟥 | 🟩 | 🟩 | 🟩 D |
| 🟥 | 🟥 | 🟥 | ap_ovld | 🟥 | 🟥 | 🟥 | 🟥 | 🟩 D | 🟩 | 🟥 | 🟥 | 🟥 | 🟥 | 🟩 D | 🟩 |
| 🟥 | 🟩 | 🟥 | ap_hs | 🟩 | 🟥 | 🟥 | 🟩 | 🟩 | 🟩 | 🟩 | 🟥 | 🟩 | 🟩 | 🟩 | 🟩 |
| 🟥 | 🟥 | 🟥 | ap_memory | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟩 D | 🟩 D | 🟩 D | 🟥 | 🟥 | 🟥 |
| 🟩 | 🟥 | 🟥 | ap_fifo | 🟥 | 🟥 | 🟥 | 🟩 | 🟥 | 🟩 | 🟩 | 🟥 | 🟩 | 🟩 | 🟥 | 🟩 |
| 🟥 | 🟥 | 🟩 | ap_bus | 🟥 | 🟥 | 🟥 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |
| 🟥 | 🟥 | 🟥 | ap_ctrl_none | 🟥 | 🟥 | 🟩 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 |
| 🟥 | 🟩 | 🟥 | ap_ctrl_hs | 🟥 | 🟥 | 🟩 D | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 |
| 🟥 | 🟩 | 🟥 | ap_ctrl_chain | 🟥 | 🟥 | 🟩 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 |

🟩 Supported Interface 🟥 Unsupported Interface

Figure 4.1: Supported interfaces by Vivado HLS [4].

optimisation pragmas which allows us to perform optimisations such as function inlining, function dataflow pipelining and function instantiation. Latency constraints can also be placed upon the functions through pragmas, which directs the function operations to be completed within a range of cycles specified. Often these optimisations are tradeoffs between performance and area utilised.

- Loop optimisations: Loops are often the most frequently occurring design pattern and Vivado HLS offers several loop optimisation options. Some of them include loop unrolling, loop pipelining, merging loops and flattening nested loops.

- Array optimisations: Performance benefits or area reductions can also be obtained by reorganising array structure.Vivado HLS provides options for array partitioning, where larger arrays are partitioned into multiple smaller arrays, to reduce RAM access bottleneck. Arrays can also be reshaped by altering their word widths and some can also be implemented as streaming interfaces using fifo interfaces.

- Arbitrary precision data types: Vivado HLS also supports the use of arbitrary precision fixed point data types, where the data types can be specified to arbitrary bit widths. A simple example relevant to our thesis, would be to encode the base pairs. Since there are only 4 base pairs present, they can be encoded using only 2 bit data types. Use of arbitrary precision data types also ensures efficient use of FPGA resources. For example, a BRAM can be perceived as two independent 18kb blocks. If an array of size 4096 of data type with a bit width of 4 is declared, an entire 18kb block is consumed. If instead of this, standard 8 bit data type is used for the array, two 18kb blocks will be used, with half of them filled with zeros.

### 4.1.2 Modelsim

Modelsim is a well known simulation and debug environment used in both ASIC and FPGA designs. The Modelsim debug environment supports a broad set of capabilities for languages such as VHDL, Verilog and SystemC. It also supports TCL scripting which makes it easier automate the design and verification procedures. For our implementations, Modelsim was the preferred simulator. Also TCL scripts were written for Modelsim to profile the run times in hardware, which explained further in section 4.1.4.3.

### 4.1.3 Gprof

Gprof or GNU Profiler [30], is a code profiling tool, which calculates the time spent in each function of the code. Code profiling is an important step in code run time analysis, which gives insights into the *hotspots* of the application under profile. Gprof provides two kinds of information, termed as *flat profile* and *callgraph*. Flat profile gives the amount and percentage of time spent in each function and callgraph provides information on how the functions are called. Details for each function are reported including the caller function and the child functions.

### 4.1.4 Helper Tools

Along with the standard tools that were used during the development, a set of helper tools were developed, which are discussed in the following subsections. The purpose of these tools were mainly to assist our application analysis, ease of development process and for accurate simulation of our target platform characteristics.

#### 4.1.4.1 Code Instrumentation

In order to fit an application on a FPGA along with profiling in the time domain, we also need to profile the application in the spatial domain. This was done by modifying the original code to collect statistics related various data structures involved in the application. A set of function templates are defined in a separate header file, to perform various calculations on the obtained values. These function templates take in the variable name and the variable type as inputs, this way, each of the variables has its own measurement functions. The statistics calculated involve mean, variance, maximum value as well the distribution of the values across different ranges. To perform measurements on any variable would first involve, the declaration of function templates with the variable name. The declared functions are then called at the place where the measurement needs to be performed. An example is shown below, which shows the instrumentation results and how it is used in our hardware designs.

```
INFO: DEF(BB_HW_MEM_INSERT_SEED_TREE_SIZE) tree_size variance: 119.728 mean: 11.600 nb: 1000000 max 9845.000000 has negative 0
INFO: DEF(BB_HW_MEM_INSERT_SEED_TREE_SIZE) tree_size bucket    0 (  0.000 -    2.461) = number 591609 59.16 % cumulative 59.31 %
INFO: DEF(BB_HW_MEM_INSERT_SEED_TREE_SIZE) tree_size bucket    1 (  2.461 -    4.923) = number 151163 15.12 % cumulative 74.42 %
INFO: DEF(BB_HW_MEM_INSERT_SEED_TREE_SIZE) tree_size bucket    2 (  4.923 -    7.384) = number  78775  7.88 % cumulative 82.30 %
INFO: DEF(BB_HW_MEM_INSERT_SEED_TREE_SIZE) tree_size bucket    3 (  7.384 -    9.845) = number  42949  4.29 % cumulative 86.60 %
INFO: DEF(BB_HW_MEM_INSERT_SEED_TREE_SIZE) tree_size bucket    4 (  9.845 -   12.306) = number  25780  2.58 % cumulative 89.17 %
INFO: DEF(BB_HW_MEM_INSERT_SEED_TREE_SIZE) tree_size bucket    5 ( 12.306 -   14.767) = number  15892  1.59 % cumulative 90.76 %
INFO: DEF(BB_HW_MEM_INSERT_SEED_TREE_SIZE) tree_size bucket    6 ( 14.767 -   17.229) = number  14563  1.46 % cumulative 92.22 %
INFO: DEF(BB_HW_MEM_INSERT_SEED_TREE_SIZE) tree_size bucket    8 ( 17.229 -   22.151) = number  16253  0.70 % cumulative 93.84 %
INFO: DEF(BB_HW_MEM_INSERT_SEED_TREE_SIZE) tree_size bucket   10 ( 22.151 -   27.074) = number  11029  0.57 % cumulative 94.95 %
INFO: DEF(BB_HW_MEM_INSERT_SEED_TREE_SIZE) tree_size bucket   14 ( 27.074 -   36.919) = number  12753  0.28 % cumulative 96.22 %
```

```
INFO: DEF(BB_HW_MEM_INSERT_SEED_TREE_SIZE) tree_size bucket   20 ( 36.919 -   51.686) = number  10696  0.12 % cumulative 97.29 %
INFO: DEF(BB_HW_MEM_INSERT_SEED_TREE_SIZE) tree_size bucket   32 ( 51.686 -   81.221) = number  10776  0.08 % cumulative 98.37 %
INFO: DEF(BB_HW_MEM_INSERT_SEED_TREE_SIZE) tree_size bucket   70 ( 81.221 -  174.749) = number  10058  0.01 % cumulative 99.38 %
INFO: DEF(BB_HW_MEM_INSERT_SEED_TREE_SIZE) tree_size bucket 3999 (174.749 - 9845.000) = number   6240  0.00 % cumulative 100.00 %
```

The example shows the measurement results of the tree data structure used in mem_insert_seed function, which is used to store all the seeds generated from the function. The output is annotated with the configuration parameter which will be used in our hardware designs, which in this case is *BB_HW_MEM_INSERT_SEED_TREE_SIZE*. The output reports the mean, variance and also the cumulative distribution of the values across different ranges. The selection of hardware configuration parameters for different input coverages is automated, by simply using the information as a look up table and selecting the appropriate number. In the example above, if we wish to select the tree size which would cover 90% of the input reads, we look up the cumulative distribution table, and we get a number of 14 from the table. Different configuration parameters are selected in a similar way.

#### 4.1.4.2 Memory Delay Wrappers

In order to connect the interfaces generated by Vivado with the Convey memory system, some enhancements had to be implemented to the Vivado generated port interfaces. The *AP_BUS* interface supported by Vivado, requires each read request to wait for its respective response before issuing another request. In case of burst request, all responses corresponding to burst request must arrive before the next request can be issued. The *AP_BUS* interface was extended to support non-blocking read requests, which means consecutive requests can be issued without waiting for the responses. The limitation of this design, is due to the lack of a control signal to stall the core from issuing too many requests and overflowing the FIFO placed between the ap_bus port and the Convey memory controller. The current design has a FIFO of size 512, which means a core can at most issue 512 concurrent read requests. Also care must be taken to ensure that each port must access different memory regions, since memory requests ordering between ports is not supported.

In order to have an accurate simulation of the Convey memory system, memory delay insertion wrapper was implemented, which would insert N number of cycles for memory access operation and N was configurable. Experiments were carried out with the enhanced *AP_BUS* implementation, to find an appropriate value of N. A pessimistic approximation of the memory latency of Convey memory system is around 300 cycles. Although, in reality, it maybe much less than that depending on the application run time behaviour.

#### 4.1.4.3 Hardware Profiling

Various profiling tools are available for software applications, which eases the analysis process. Once a hardware design is implemented, we would require a tool to perform such an analysis, through out the development process. Such a hardware profiling tool would make it much easier to identify the hotspots in the hardware implementation and also gives us the direction in which the optimisations need to be carried out in the design.

Vivado reports the latency of the top level module implemented, but we would also be interested in how the sub-modules perform.

In order to profile the generated hardware, a TCL script was written, which measured the total number of cycles spent in each instantiation of a module. Vivado, instantiates every function as a hardware module with a set of interfaces, by which, we could recognise if the module was active or inactive at any given time. The script would scan for all such modules instantiated, recursively, and collect the cycles spent when the module was active. This way, we were able to collect the run time of all the modules in the hardware design. Along with the memory delay wrappers and the profiling scripts, we could model the behaviour of our design in the convey system, in a realistic manner.

## 4.2 Application Analysis

A callgraph of the BWA-MEM application is shown in Figure 4.2, which shows the functions of the application. The application consists of two main worker threads, *worker1* and *worker2*. Worker1 is concerned with generation of chains, chain filtering and local alignments of the chains on the reference. Worker2 generates the global alignments and produces the required SAM [31] output format. The SMEM generation block of functions shown in Figure 4.2 are more memory intensive, while function blocks implementing dynamic programming are relatively more compute intensive. Also the software uses data structures such as trees to store the list of chains, which leaves considerable memory footprint.

### 4.2.1 Application Profile

We choose bwa software release version 0.7.7 for our development and profile experiments. The application was profiled on the target platform of Convey, which has two Intel Xeon processors with 128 GB RAM and operating at 3.33 GHz clock rate. Default options of the application were chosen for the experiments. *wgsim* [32] tool was used for generating the synthetic data sets. The data sets generated comprised of reads of different lengths, and had a total of 1M reads. The profile using synthetic reads was also performed against two references. One of them was taken from the GrCh38 human genome build with the total length being 3,209,286,105 bp [33] and the other was a fragment of the GrCh38 human genome with its total length being 106,500 bp. The two profiles gives us a comprehensive overview of the application dynamics, concerning both the read lengths and the size of reference index. The profile was performed using the *gprof* tool. The read lengths are taken from 100 bp to 250 bp, since BWA-MEM is designed for longer reads.

Table 4.1 shows the results of the synthetic reads against the fragmented genome, which is of lesser length than the full genome reference. The actual function names are replaced by names describing their functionality for the sake of clarity.

We can see from the table above that the smith waterman blocks of the application prove to be the time intensive parts of the application. We also observe that as the read length increases, the time spent in smith waterman blocks increase, which shows a direct dependency of dynamic programming on the length of the reads.

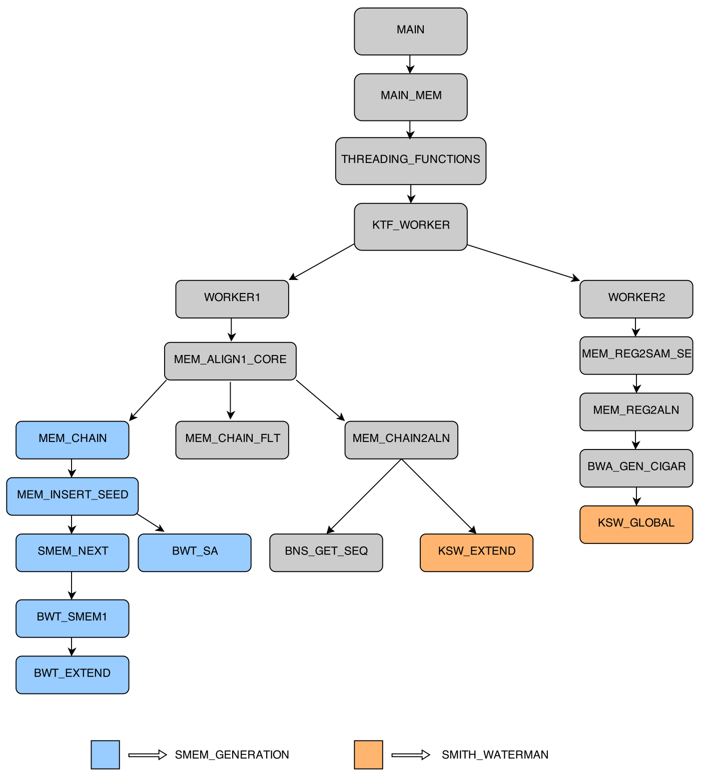


Figure 4.2: BWA-MEM callgraph.

| Read length | Smith waterman in worker1 | Smith waterman in worker2 | Occurrence array access functions | Suffix array access functions | SMEM identification |
|---|---|---|---|---|---|
| 100 | 19% | 42% | 25% | 3% | 5% |
| 150 | 26% | 44% | 18% | 2.5% | 4% |
| 200 | 30% | 46% | 14% | 2% | 3% |
| 250 | 33% | 47% | 12% | 1% | 2% |

Table 4.1: Profile results from aligning the synthetic reads against a fragment of human genome.

Table 4.2 gives a much more realistic view of the application runtime characteristics. We observe here that the functions with memory accesses take much more time than the dynamic programming blocks. This behaviour is a direct consequence of the bigger size of the index. Since there is a much larger index, the possibility of finding more matches increases. But we still observe the basic behaviour of increasing smith waterman run time with increasing read lengths.

As mentioned earlier, the occurrence array values are organised into bin intervals,

| Read length | Smith waterman in worker1 | Smith waterman in worker2 | Occurrence array access functions | Suffix array access functions | SMEM identification |
|---|---|---|---|---|---|
| 100 | 23% | 8.5% | 39% | 19% | 2.5% |
| 150 | 25% | 12% | 38% | 17.7% | 2.5% |
| 200 | 26% | 15% | 36% | 16% | 2.5% |
| 250 | 25.8% | 17% | 35% | 21% | 2% |

Table 4.2: Profile results from aligning the synthetic reads against the full human genome.

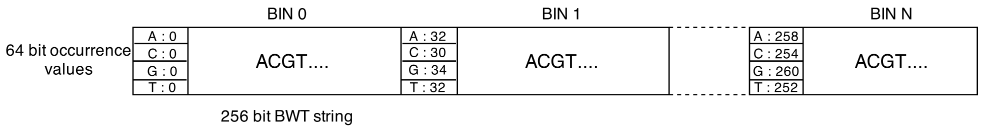


Figure 4.3: BWT memory layout [5].

where each bin has the occurrence values of 128 bp. The BWT array is interleaved in such a way that the first four 64 bit values are the occurrence array values of the four bases, and the next 256 bits carry the actual BWT string of 128 bp. The BWT array structure is as shown in Figure 4.3.

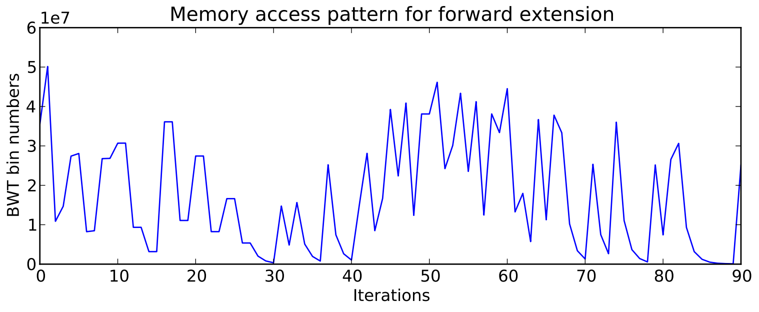


Figure 4.4: Memory access pattern for finding a MEM

The memory access pattern of the application was obtained with one read of length 250 bp aligned against the human genome. Figure 4.4 shows the memory access pattern of the application, when finding for a SMEM. We see from the Figure 4.4 that the pattern is quite random, with a large variation in the occurrence array bins accessed. Clearly such a behaviour, which is an integral part of the algorithm, precludes any benefit the application can have from providing a cache, as it exhibits very little temporal and spacial locality of memory accesses.

**Discussion**

There are a few noteworthy points to be considered from the above experiments. The application behaviour is highly dependent upon the nature of the reads and reference. It is very difficult to predict the behaviour of a certain set of reads prior to execution. But we manage to have some insights into the runtime characteristics of the application. The runtime of smith waterman blocks have a high correlation with the length of the reads, while the runtime of the memory access functions have a high correlation with the size of the reference index. We also see from Figure 4.4 the memory access pattern of the application is highly random. The access to occurrence array in the current iteration is dependent on the value fetched during the previous iteration. This becomes obvious when we observe the equations 2.1 and 2.2. The accesses to occurrence array are hence serialised. This inter dependency is an integral part of the algorithm. The profile results of Table 4.2, also makes it clear that the BWA-MEM is a memory intensive application. We see from the the results that, more than 50% of the time is spent in accessing memory.

We see two opportunities for improvement, them being in the memory access functions and the smith waterman functions. There is already a considerable amount of research work produced related to accelerating smith waterman algorithm on FPGAs [34].The thesis would hence concentrate on the memory bottleneck of the application, and designs to hide the memory latency are proposed.

### 4.2.2 Code Instrumentation Results

By performing code instrumentation on the original application, we present here statistics regarding few of the significant parameters specific to BWA-MEM. Table 4.3 gives the summary of the statistics involving the mean, variance, maximum and also the respective sizes for different read coverages.

Table 4.3 shows the variation of different of parameters of the application for varying read lengths and read coverage. The parameters chosen to be shown in the table have a significant impact on resource utilisation. We see in particular, the parameters of tree size and number of alignments found grow exponentially as the read coverage increases. This makes sense as we try to support more reads in the design, we get closer to the reads which consume large memory. If we choose to cover all reads, as we see from Table 4.3, few parameters such as tree size have impractical maximum values to be supported on the limited resources of the FPGA.

When implementing the final design, these results could play a significant role, providing possibility of trade off for read coverage to duplication of cores. We could support multiple cores, supporting lesser read coverage or provide lesser cores supporting larger read coverage.

## 4.3 Methodology

The use of Vivado HLS definitely increased the productivity and rate of development. Since approximately 90% of the time was spent in worker1 thread of the application, worker1 was chosen for hardware implementation. For each function in the call tree of worker1, the input and output data were collected, which was later to be used in verification process. The functions were implemented in Vivado separately, synthesised and verified using the data files collected earlier. The development flow using Vivado HLS is shown in Figure 4.5.

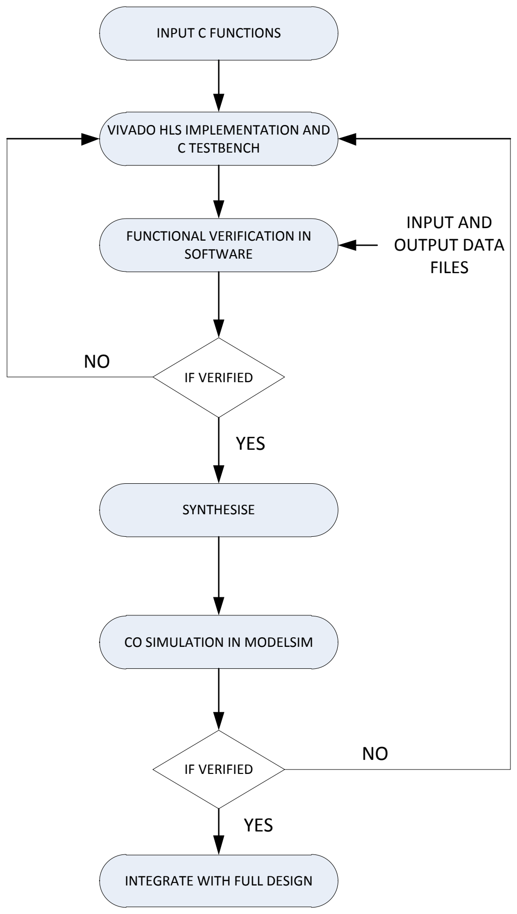

INPUT C FUNCTIONS
VIVADO HLS IMPLEMENTATION AND C TESTBENCH
FUNCTIONAL VERIFICATION IN SOFTWARE
INPUT AND OUTPUT DATA FILES
NO
IF VERIFIED
YES
SYNTHESISE
CO SIMULATION IN MODELSIM
IF VERIFIED
NO
YES
INTEGRATE WITH FULL DESIGN

Figure 4.5: Vivado HLS implementation methodology.

The Vivado HLS tool provides three stages for conversion of an input C function to its RTL equivalent. Corresponding data files need to be generated prior to be used during verfication as well as test vector generation. A separate testbench file has to be written which also will be used later to generate RTL testbenches. The first stage involves performing necessary changes to the C function, so that tool will be able to synthesise and simulate it. The steps and rules to be followed for such a change is clearly defined in it user manual [4]. Once the function is transformed, the first stage involves the software simulation to check its functional validity. If it is verified to be true, the function is synthesised, during which stage, the equivalent RTL files are generated in VHDL, Verilog and SystemC. Also the test vectors and testbenches are generated during this stage. At the final stage, a Co-simulation is performed with generated RTL testbenches and files, in any of the RTL simulators.

| Read Length | Variables | Variance | Mean | Maximum | Coverage | | |
|---|---|---|---|---|---|---|---|
| | | | | | 90% | 96% | 99% |
| 70 bp | tree size for chains | 119.7 | 11.6 | 9845 | 13 | 28 | 82 |
| | number of MEMs | 4.2 | 7.0 | 28 | 12 | 13 | 17 |
| | seeds per chain | 0.5 | 1.34 | 46 | 2 | 3 | 3 |
| | number of alignments | 117.8 | 10.2 | 9841 | 10 | 28 | 94 |
| 100 bp | tree size for chains | 77.9 | 11.9 | 9678 | 17 | 39 | 97 |
| | number of MEMs | 3.9 | 8.5 | 28 | 12 | 15 | 18 |
| | seeds per chain | 0.6 | 1.7 | 50 | 4 | 4 | 4 |
| | number of alignments | 61.9 | 6.8 | 9298 | 5 | 21 | 80 |
| 150 bp | tree size for chains | 70.2 | 15.4 | 9599 | 27 | 70 | 123 |
| | number of MEMs | 3.2 | 9.9 | 27 | 14 | 16 | 18 |
| | seeds per chain | 0.89 | 2.4 | 52 | 4 | 4 | 5 |
| | number of alignments | 37.3 | 5.3 | 6653 | 4 | 14 | 69 |
| 200 bp | tree size for chains | 83.8 | 19.5 | 9811 | 35 | 84 | 194 |
| | number of MEMs | 2.7 | 10.7 | 26 | 14 | 16 | 19 |
| | seeds per chain | 1.0 | 3.14 | 51 | 4 | 5 | 6 |
| | number of alignments | 33.7 | 4.8 | 9339 | 3 | 10 | 43 |
| 250 bp | tree size for chains | 90.7 | 23.1 | 9780 | 45 | 108 | 177 |
| | number of MEMs | 2.5 | 11.7 | 27 | 15 | 17 | 20 |
| | seeds per chain | 1.2 | 3.8 | 20 | 6 | 6 | 7 |
| | number of alignments | 23.9 | 4.4 | 4780 | 4 | 11 | 49 |

Table 4.3: Code instrumentation results for BWA-MEM.

# 5 Architecture and Implementation

The chapter introduces and discusses the architectures that has been developed to overcome the memory bottleneck of the application. The chapter first describes the design motivations and the challenges involved in porting the application to Vivado. The Base architecture is first described in section 5.3, with the high level description of the design. The Batch architecture is then proposed in section 5.4, which is aimed at hiding the memory latency of the target platform. The synthesis results are also discussed of both the architectures in view of future scalability of the designs.

## 5.1 Design Motivation

Before we proceed to describe the architecture details, we discuss here our design goals and the motivation behind our designs proposed. We propose two hardware architectures for the application, implemented using Vivado. The application is quite complex and dynamic in nature, but is poorly documented. There is little scientific literature available about the algorithm itself and sufficient time had to be spent on understanding the application and mapping of algorithm to application, before we could choose the implementation platform. According to our limited knowledge, this seems to be the first work carried out on hardware acceleration of BWA-MEM. It is difficult to predict the behaviour of the application against a data set since it is highly dependent on the nature of reads. To concur upon the best possible architecture, performing high level architecture explorations gains more importance. Vivado provides us the option of faster development and verification time, which enables more time with architecture exploration. The trade off of increased porting time to Vivado, against reduced architecture exploration time was judged to be worthwhile.

Our initial goal was to port the application to Vivado, without any major architectural optimisations done. The porting of the application was the first implementation step, since this would give us a stable base to carry out further architecture explorations. We term this version as the *Base* architecture. The architectures developed after the Base architecture, had the primary design goal of throughput. The designs should be able to align as much reads as possible for a given time. We would also want the hardware designs to produce similar outputs to the software application. This goal was incorporated in our development methodology itself. As explained in section 4.3, for each function ported to Vivado, input and output data files were collected, and were used to verify the correctness of the ported design. The designs should also support the various options of the application. With these goals in mind, we proceed to the porting process of the application to Vivado.

## 5.2 Porting to Vivado HLS

Before we introduce the *Base* architecture, we describe the steps and some challenges involved in porting the application to HLS. The porting procedure had the following modifications which needed to be carried out to the original application.

- Vivado does not support dynamic memory allocation functions. The software uses sophisticated C constructs, such as trees and lists, all of which require dynamic memory allocation. These structures need to be changed to arrays of pre determined size. This is where code instrumentation results play a vital role. The static values of these arrays, were chosen from the instrumentation results and are configurable at design time.

- Vivado also does not support structures as part of the function interfaces. Most of the functions in software had complex structures as part of the interface. These interfaces had to be split in to its elements, which increased the interface size of the Vivado functions.

- The above two changes induced a fair amount of code rewriting and restructuring. Often times, the code size increased because of this.

- Along with the above modifications, there were a few unorthodox challenges encountered along the way, which consumed a fair bit of time to resolve as well. Challenges such as use of basic data types such as *int32_t*, caused issues during compilation. One of the other such unexpected problems was regarding passing of local variables as pointers to functions, which Vivado was not able to recognise. Although these were not major problems, resolving them often took, not negligible, effort and time.

## 5.3 Base Architecture

We describe here the base architecture which was developed from the application source code using Vivado. Figure 5.1 shows the top level architecture of the design. There are a few enhancements to the very first ported version of the software. The hardware architectures developed, only incorporate the worker1 of the software application. The key feature of the architecture is the separation of SMEM generation part and the smith waterman of worker1. The separation is done in such a way so that the two parts can be pipelined in the future. The SMEM generation part here is indicated as *MEM_ALIGN1_CORE_PART1* and the smith waterman block is indicated as *MEM_ALIGN1_CORE_PART2*.

The architecture is implemented in such a way, where several *MEM_ALIGN1_CORE_PART1* cores can be implemented to be working with a single *MEM_ALIGN1_CORE_PART2* core. Currently, these cores operate synchronously, but provisions are provided to make them operate asynchronously in the future. Since time consumed in memory access functions is much higher than the smith waterman function, multiple SMEM generation cores are implemented. The number of such cores

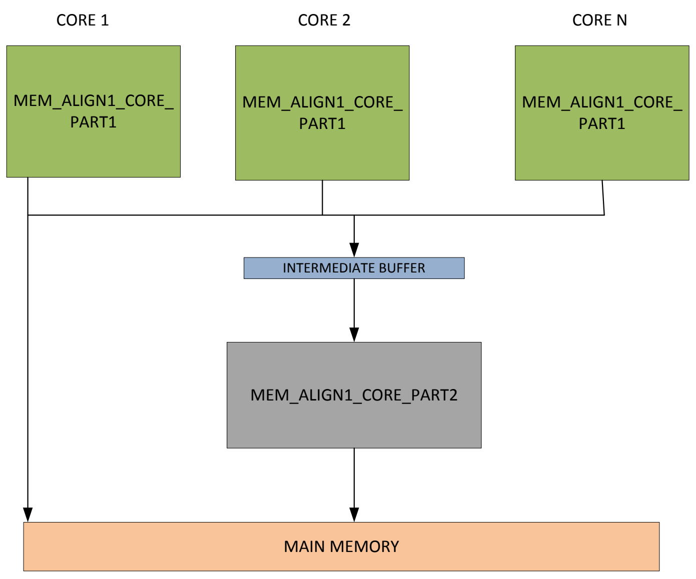


Figure 5.1: Top level Base architecture.

is reconfigurable. Once the chains are generated, they are copied to an intermediate buffer and *core done* signal is raised. *MEM_ALIGN1_CORE_PART2* core keeps polling the signal and when it is active, it starts execution on the data on the intermediate buffer. The *MEM_ALIGN1_CORE_PART1* cores are implemented in such a way where, to change the number of cores, the cores are provided as defines of code templates and these defines have to be declared at the required place. The number of such defines decides the number of cores in the design. Also each core has its own set of buffers, which are also provided as templates to be defined.

The top level core obtains the reads and the options, along with few other data in a packed format. These data are packed together into words of 64 bits, to optimise bandwidth utilisation. The reads are also obtained in a packed format, where each base is encoded using a 3 bit data type, to support the four bases and an ambiguous base. After receiving the initial data, they are unpacked, with the options updated and reads placed in appropriate buffers.

There are few implementation changes compared to the software version. The mem_chain function of software, has been removed, since it only forms a wrapper for mem_insert_seed function. Figure 5.2 shows the internal architecture of *MEM_ALIGN1_CORE_PART1*. The Vivado functions are annotated with *hw* string. The mem_insert_seed_hw block, is rewritten to separate the code paths of smem_next_hw and bwt_sa_hw. In the software application, for each SMEM found, bwt_sa would be applied to find out the mapping to actual reference positions. In mem_insert_seed_hw block, all the SMEMs for a read are collected and placed in a local buffer. A new hardware function called mem_insert_seed_1_hw is created as a wrapper to bwt_sa_hw, which contains helper functions for chaining. The dependency between bwt_sa_hw and smem_next_hw is removed in this way, which provides the possibility of pipelining in future.

smem_next_hw and bwt_smem1_hw remain similar to the software implementations, with no major restructuring. bwt_extend_hw module has internally the occurrence array access functions, which perform the actual access of BWT string. The occurrence array access function also has the counting of bases in the BWT string, using the hamming weight calculation algorithm.

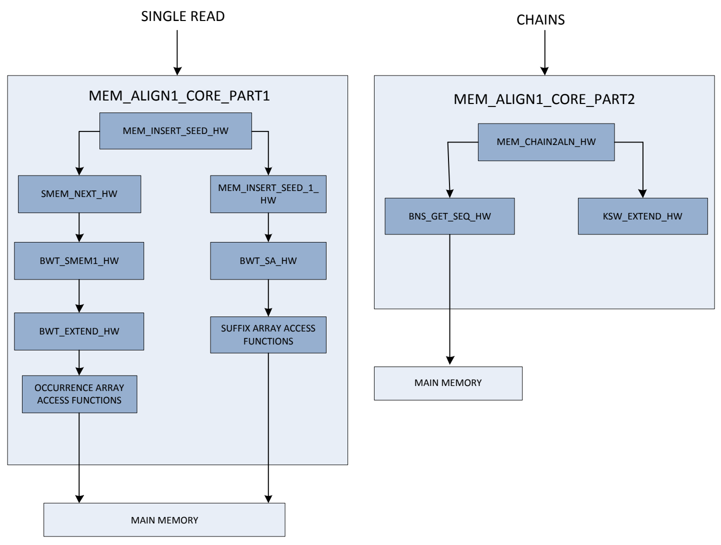


Figure 5.2: Sub modules of Base architecture.

The software also had a fair amount of sorting functions, particularly in the chain filtering process and chain tree parsing. The sorting functions implemented often utilised *introsort* algorithm from ksort library. In hardware, sorting functions take negligible amount of cycles, and hence a simple *bubblesort* algorithm was written for Vivado to replace those functions. Also additional debug support was provided for each Vivado function, by providing error and debug ports. When processing a read, if the memory requirements for the read, exceed those provided, errors are raised on these ports to signal the software that the particular read cannot be processed. To enable better debug support when performing cosimulations, debug ports are provided which prints the debug message from the particular hardware function block, whenever an error is encountered.

### 5.3.1 Synthesis Results

The architecture supports different read lengths and different number of cores, and hence results for these combinations are presented below. Although the cores do not support asynchronous execution for now, it will be useful to obtain some insights into the resource utilisation characteristics for these cores. The designs were synthesised in Vivado HLS for the Xilinx Virtex 6 LX760 FPGA, at time period of 3.5 ns. The available resources on the FPGA is as shown in Table 5.1.

| Resources | BRAM18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| Total | 1440 | 864 | 948480 | 474240 |
| Convey modules | 178 | 0 | 53660 | 55676 |
| Available | 88% | 100% | 94% | 88% |

Table 5.1: Resource utilisation on Virtex-6 LX760 FPGA for Convey modules.

Table 5.1 shows the total resources available on the FPGA. Convey modules such as Crossbar memory subsystem interface, Read order cache and Performance monitor were then instantiated on the FPGA. The resource utilisation for these convey modules are also shown. We need to remember that the available resources would be lesser than what is reported in the table since, it would also include the core to Convey interfaces.

Figure 5.3 shows the resource utilisation numbers for varying number of cores and for different read lengths. The results show consistent numbers across different lengths of reads for similar number of cores. There is slight increase in resource utilisation for longer reads of 200 bp and 250 bp compared to their lesser counterparts. The synthesis was done with instrumentation results obtained for a read coverage of 99.99%. Some tradeoffs with read coverage would have been required, if the utilisation was nearing the limits of available resources.

## 5.4 Batch Architecture

To address the basic bottleneck of the application, which is memory latency, a *Batch* processing architecture is discussed in this section. Although there is very little low level parallelism available in the application, since millions of reads are being processed, we could make use of the task level parallelism to hide the memory latency of the system. The batch architecture, is similar to the base architecture, but instead of each core processing one read at a time, a batch of reads are processed by every hardware function. The top level batch architecture is shown in Figure 5.4.

The implementation of batch architecture was well defined since we already had a stable base to work upon. Each Vivado function had to be modified to support multiple number reads to be processed, which could be configured to an arbitrary number at design time. The modifications for this involved increasing the memory allocated for arrays, increasing the function interface sizes as well as restructuring the control path for each of the functions. The control code restructuring was more complex, since the majority of functions had complex control paths which were data dependent.

The underlying architecture of *MEM_ALIGN1_CORE_PART1_BATCH* is shown in Figure 5.5. Each of the functions from the base architecture in the smem_next_hw branch is converted to a batch format. mem_insert_seed_1_hw branch however has not yet been

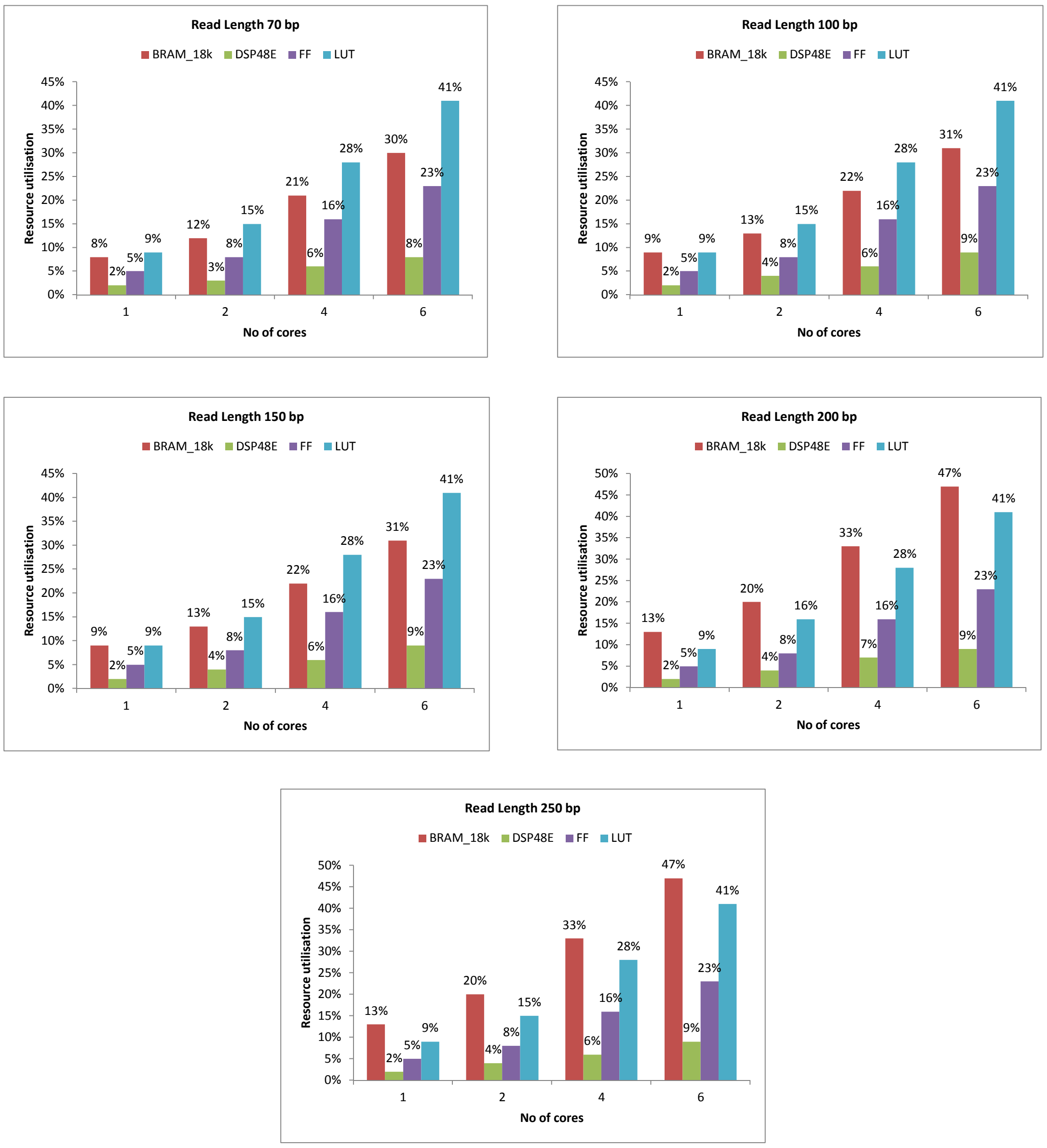


Figure 5.3: Synthesis results for Batch architecture.

converted to a batch processing version. Hence the suffix array access still occurs one read at a time. It will be interesting to see the performance of batch version of occurrence array access functions, which would give us some insight with regards to whether a batch version of suffix array access functions is necessary.

The bwt_extend function in software contains the occurrence array access functions termed bwt_2occ4 and bwt_occ4 which calculate the number of occurrences of the specified base given an index on the BWT string. The function also requires to copy the

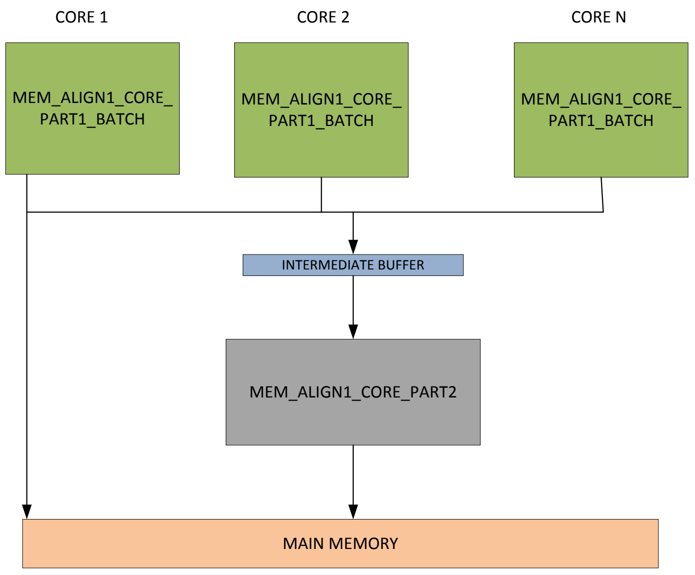

CORE 1
CORE 2
CORE N
MEM_ALIGN1_CORE_ PART1_BATCH
MEM_ALIGN1_CORE_ PART1_BATCH
MEM_ALIGN1_CORE_ PART1_BATCH
INTERMEDIATE BUFFER
MEM_ALIGN1_CORE_PART2
MAIN MEMORY

Figure 5.4: Top level Batch architecture.

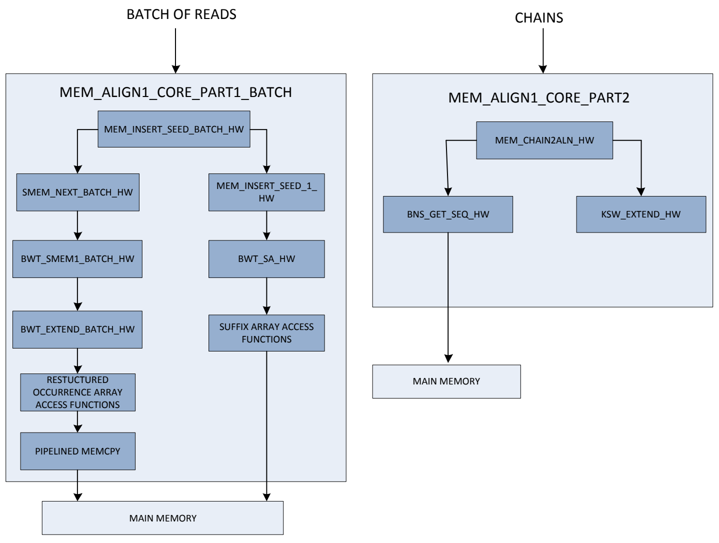


Figure 5.5: Sub modules of Batch architecture.

BWT string to a local buffer before the counting operation can be performed on the string. In the batch version of hardware, bwt_extend_hw is restructured to have the memcpy function aggregated together as a separate module. The aggregated memcpy module generated by Vivado interface can only send memcpy requests sequentially. A separate memcpy module was written which would pipeline the memory requests. This would enable each memcpy to start in non-blocking mode every two cycles apart. Now instead of suffering a 300 cycle memory latency upon every request, a batch of requests can be sent in a pipelined manner with only the first request suffering a delay due to memory latency. This way a batch of requests suffer the penalty of memory latency instead of every request encountering the delay. This forms the primary reason for the implementation of batch architecture.

With the batch architecture, we expect a linear speed up with batch size, since with greater batch size we have more memory requests to be pipelined. One significant point to be noted here, would be that, the function latency could still not be reduced beyond a certain point, since now all the reads would be stalled until the read with the longest match completes execution of the bwt_extend function. The trade off between this delay and the gain from the memory pipelined requests will be explained in the results chapter.

### 5.4.1 Synthesis Results

The batch design was again synthesised in Vivado HLS for the Xilinx Virtex 6 LX760 FPGA, at time period of 3.5 ns. We observe in Figure 5.3, that the variation between different reads remains consistent. For this reason, we present here the results for read lengths of only 250 bp. The interesting part about this architecture would be that, it has many degrees of freedom to be explored. Since each module can support an arbitrary number of batch reads, the number of batch reads also plays a significant role in resource utilisation. Along with the number of cores, the resource utilisation now becomes a trade off between read coverage, the batch size and the number of cores of *MEM_ALIGN1_CORE_PART1_BATCH*. We present below the table concerning the variation of resource utilisation numbers with different parameters.

To give a clearer view of the impact of read coverage on resource utilisation, we provide two set of synthesis results in Figure 5.6 and Figure 5.7. Figure 5.6 provides the synthesis results for a varying number of cores and different batch sizes for a read coverage of 99.99%. Figure 5.7 provides the synthesis results of a similar kind for a read coverage of 98.99%. With a coverage of 99.99%, we observe that we can fit only one core of batch size 8, while with the coverage reduced just by 1%, we see it is possible to fit two cores of batch size 8.

### 5.4.2 Scalability

The cores, although not operating concurrently at present, would provide speed up when operating asynchronously. Hence it would be an interesting question, to explore the impact of batch sizes together with the number of cores in operation. Ideally we would like to fit more cores with more batch sizes, assuming both provide speed up. The case for selecting the configuration of cores and batch sizes would then depend on throughput. An interesting observation from the synthesis results, is that configurations

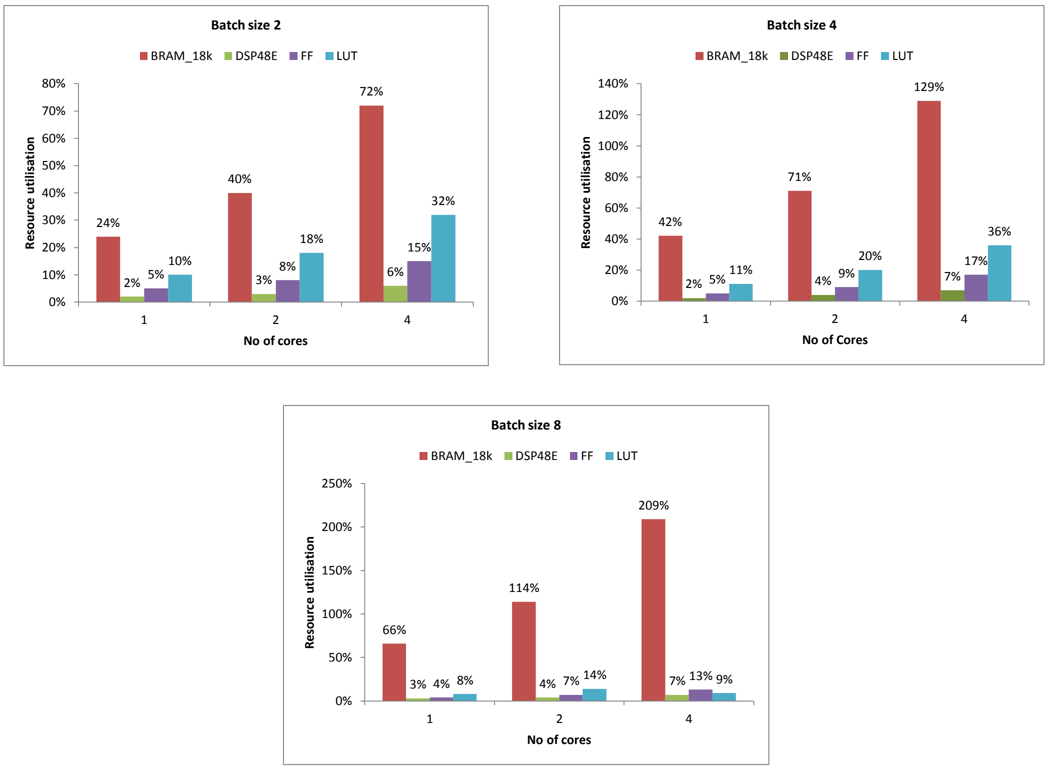


Figure 5.6: Synthesis results for Batch architecture at read coverage of 99.99%

with similar through put have similar utilisation of resources. We see that 4 cores with a batch size of 2, 2 cores with a batch size of 4 and 1 core with a batch size of 8, all have the same throughput of processing 8 reads. In this case, the decision can only be taken after the results for different batch sizes and number of cores have been analysed.

Also the opportunity to fit more cores arises when the read coverage is decreased. We see from the results that by decreasing the coverage by 1% provides the opportunity to fit an extra core of batch size 8. Another interesting question then would be the cut-off of read coverage, at which the scaling the number of cores would not provide benefit to the application. We have to remember that, even reducing coverage by 1% in a test case of 1M reads, would result in 10,000 reads, which would need to be processed in software and there is a high possibility that these would remain some of the most time consuming reads. The answer to this question can only be obtained after getting more insight into the behaviour of the architecture. The scalability opportunities discussed are based on assumptions of speed up from batch size and number of cores. We will address these points once we have the results from experiments with the architecture.

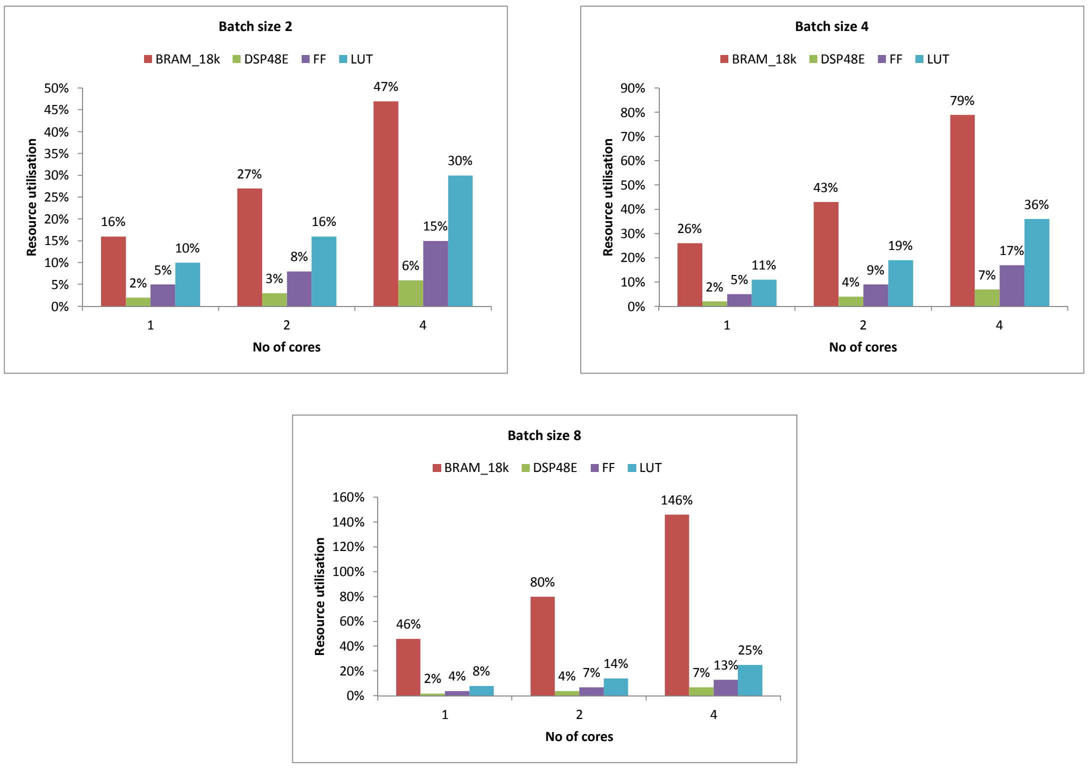


Figure 5.7: Synthesis results for Batch architecture at read coverage of 98.99%

# 6 Results

This chapter provides the experimental results of the architectures proposed and discusses the impact of these results on scaling. Section 6.2 discusses the performance results of the Base architecture, while section 6.3 presents the performance results of the Batch architecture. We also discuss the opportunities present in the proposed architectures where pipelining can be performed.

## 6.1 Methodology

Before proceeding to the actual numbers from the experiments, we would like to explain the methodology followed to obtain the results and a few limitations encountered. The measurements are performed for a test genome of length 106,500 bp length. The data set used was of 100 reads of length 150 bp, 200 bp and 250 bp, which are synthetically generated using wgsim tool. The measurements were done using Modelsim during the co-simulation of the design in Vivado. The reason for using a small data set was to accommodate the simulation time of the designs. The designs have not yet been fully synthesised for the Convey FPGAs, which would be done once they are verified to bring benefits. The simulations are performed with default option parameters of the application. Although the current implementation of the design supports multiple cores, since they are not operating concurrently, the simulations have been performed with a single core.

The software run times on the Convey platform is measured for a single thread, and the measurements obtained in the simulation are compared against them. The software was also run using the same data set which is being used for Modelsim simulations. The simulations are performed with the memory delay wrapper included, to have a realistic model of the Convey memory latency. The designs are synthesised at 3.5 ns time period and the performance numbers are measured at the same clock period. We first present the performance numbers of the Base architecture and also provide a hardware profile of the architecture.

## 6.2 Base Architecture Performance

We present the performance numbers and speed-up obtained from the base architecture against the single thread software performance. Figure 6.1 shows the comparison with varying read lengths. We see that for read length of 250 bp, the speed up is slightly above 0.05, which is about 19 times slower than the software. Similarly, the performance results for different read lengths are shown in Figure 6.1. On average, the base architecture is about 22 times slower than the software version.

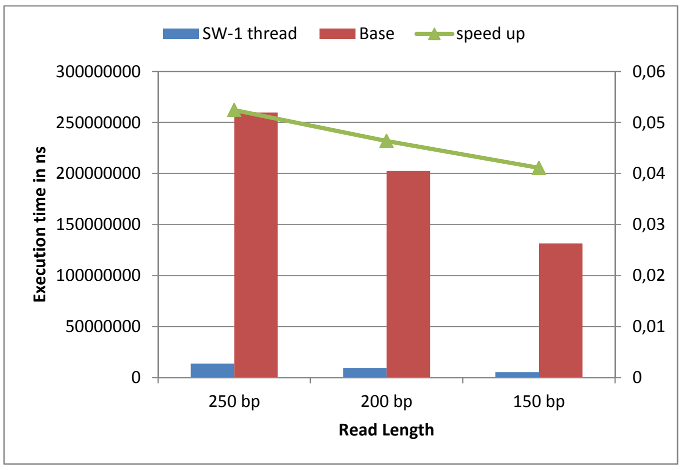


Figure 6.1: Performance results of Base architecture

| Modules | Execution profile |
|---|---|
| MEM_ALIGN1_CORE_PART1 | 57.99% |
| MEM_INSERT_SEED_HW | 57.95% |
| SMEM_NEXT_HW | 50.86% |
| BWT_SMEM1_HW | 50.82% |
| BWT_EXTEND_HW | 49.32% |
| MEM_INSERT_SEED_1_HW | 7.07% |
| BWT_SA_HW | 7.04% |
| MEM_ALIGN1_CORE_PART2 | 25.70% |
| MEM_CHAIN2ALN_HW | 25.70% |

Table 6.1: Hardware profile results for Base architecture

The Base architecture still suffers from the effect of latency of the memory system. Each memory access still incurs a penalty of 300 cycles from the memory delay wrapper. This remains the main cause for the slow down of the Base architecture compared to the software.

The hardware profile numbers for the base architecture is shown in Table 6.1, just to give an insight into the design performance bottlenecks and an indication to perform estimations for further improvements. The measurements for the hardware profile are done for a single read of length 150 bp is aligned against the fragment genome. We see from the results that, bwt_extend_hw and mem_chain2aln_hw are the most time intensive. These are the occurrence array access functions and the smith waterman cores respectively. We also observe that memory access is still the bottleneck of the design.

## 6.3 Batch Architecture Performance

The Batch architecture is designed precisely to overcome the effects of the memory latency of Convey system. The data set used for simulations is the same as that used for Base architecture simulations. We now present the performance numbers of the Batch architecture. Figure 6.2 shows the run times of the batch architecture compared with the software.

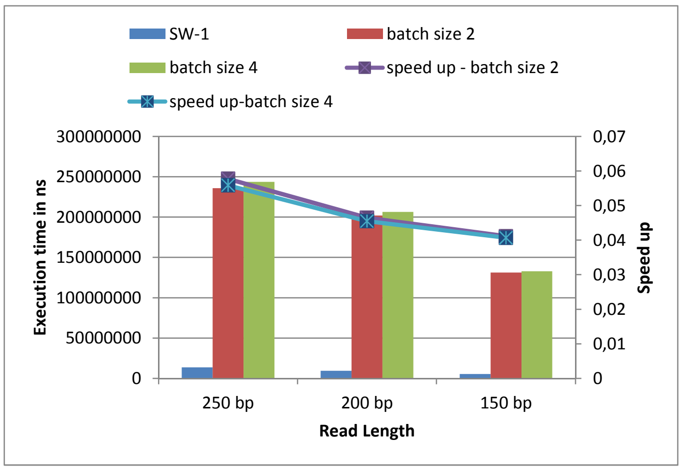


Figure 6.2: Performance results of Batch architecture.

The speed up obtained ranges from 0.058 to 0.040 for differing read lengths, on average this means that the batch architecture with batch size of 2 is about 17 times slower than the software. Although there is an improvement in the run times, the improvement is lesser than what was expected. The reason for this being the variance in the number of matches found for each read in the batch of reads. If in a batch, one of the reads finds longer matches or MEMs, the processing of other reads in the batch is stalled until the most time intensive read finishes the processing. The penalty incurred by this is alleviating the improvements provided by pipelining the memcpy requests. This is also proved by the fact that, as the batch size increases, the speed up is reduced. Since now more reads are incurring the penalty of stalled cycles, while the most time intensive read finishes the operation. Figure 6.3 shows the comparison of latency for processing of reads of length 250 bp for different batch sizes.

We see from Figure 6.3 that, the latency of the Batch architecture with a batch size of 2 remains lower than with a batch size of 4. We actually see for read lengths of 150 bp and 200 bp, batch size of 4 incurs more latency than the Base architecture itself. To further emphasise the impact of the variance of run time behaviour between the reads, we have a measurement with a data set which contains a single read repeated multiple times. The measurements were performed for 10 such reads, for the read length of 150 bp. The latency, with the same read in a batch, should now improve with batch size. Figure 6.4 shows that, this indeed is the case.

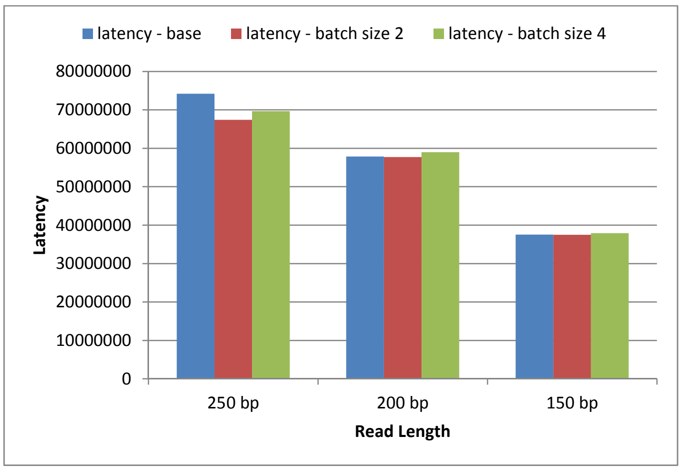


Figure 6.3: Latency comparison of different batch sizes for Batch architecture.

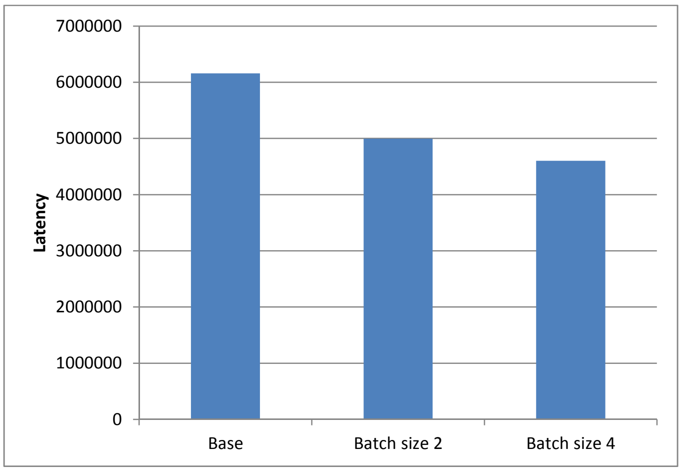


Figure 6.4: Latency comparison of different batch sizes for Batch architecture with the same read

We see from Figure 6.4, that the performance does improve with increasing batch sizes. We see that the Batch architecture with a batch size of 2 gives a performance improvement of about 18% over the Base architecture. With a batch size of 4, the Batch architecture performs about 25% better that the Base architecture for this particular data set. This would mean that we would obtain better performance if the batches contain reads with similar run time behaviour. This however cannot be determined beforehand to perform some pre-processing on the reads, to utilise the Batch architecture to its optimum.

Hardware profile for the Batch architecture is also shown in Table 6.2. These numbers

| Modules | Batch size 2 | Batch size 4 |
|---|---|---|
| MEM_ALIGN1_CORE_PART1_BATCH | 60.1% | 60.89% |
| MEM_INSERT_SEED_BATCH_HW | 60.0% | 60.88% |
| SMEM_NEXT_BATCH_HW | 49.16% | 48.89% |
| BWT_SMEM1_BATCH_HW | 49.06% | 48.79% |
| BWT_EXTEND_BATCH_HW | 44.79% | 43.39% |
| MEM_INSERT_SEED_1_HW | 9.77% | 10.79% |
| BWT_SA_HW | 9.72% | 10.70% |
| MEM_ALIGN1_CORE_PART2 | 29.86% | 33.47% |
| MEM_CHAIN2ALN_HW | 29.70% | 33.45% |

Table 6.2: Hardware profile results for Batch architecture.

are obtained by aligning a single read of length 150 bp against the fragment reference. Although they do not provide the complete profile over large number of reads, they give us an indication upon which we can make some estimations for further improvements. We see that, the bwt_extend_batch_hw and mem_chain2aln_hw still remain as the most time intensive modules.

## 6.4 Pipelining

We mention the pipelining opportunities available in the architectures in this section. Although the performance of the single core Base and Batch architectures do not meet the intended levels of acceleration, there are pipelining opportunities present in the architectures to provide some significant improvements in the future.The Base and Batch architectures provide pipelining opportunities at three stages.

- MEM_ALIGN1_CORE_PART1 and MEM_ALIGN1_CORE_PART2 cores can be pipelined to hide the execution times of smith waterman modules. From Table 6.1 and Table 6.2, we see that the average fraction of time spent in MEM_ALIGN1_CORE_PART2 core is about 25% to 30%. Pipelining at this stage would provide us with approximately 30% improvement.
- In the MEM_ALIGN1_CORE_PART1 core, the smem_next_hw and mem_insert_seed_1_hw modules can be pipelined. This would mean that, the occurrence array accesses and the suffix array accesses can be executed concurrently. mem_insert_seed_1_hw module takes, on an average, 10% of the execution time. Pipelining them would also result in about 10% improvement.
- Finally, multiple MEM_ALIGN1_CORE_PART1 cores can be executed concurrently. This should provide a linear speed up, depending on the number of cores implemented. For implementing more cores, the scalability of the design should be taken into account. The scalability is affected by the resources on the board as well as the memory ports available. Convey memory system supports a total of 16

memory ports for the 8 memory controllers available. Since the synchronisation between ports is not guaranteed, we would like to reserve a single memory port per core, which would mean a total of 16 cores can be supported at maximum. Since measurements are not available for multiple cores execution, providing an estimate at this stage is difficult.

**Discussion**

The architectures that have been proposed in the thesis were meant to overcome the memory bottleneck of the application. The measurements taken for single core hardware modules were compared against the single thread software execution times for different lengths of reads. The Base architecture on average remains about 22 times slower than the software, while the Batch architecture was about 20 times slower on average, for different read lengths. The Batch architecture does not provide the intended level of performance improvement mainly due to dynamic nature of the architecture for different reads. The Batch architecture introduces stalls to processing of all reads in a batch till the read with the highest matches finishes processing. Hence the batch architecture provides better results for batches of small sizes, since this alleviates the effect of stalls induced by the read with longest matches.

The dependency of scalability of the architectures on resource utilisation was discussed in the section 5.4.2. For different configurations of cores and batch size, from the current results, we conclude that better performance would be attained for configurations of smaller batch sizes and more cores. In the example taken in section 5.4.2, the configuration of 4 cores with batch size of 2 would perform better than 2 cores with batch size of 4. With the pipelining suggestions of the previous section, we would expect the execution time of single core Batch architecture to come closer to the software execution time. With the utilisation of all four FPGAs of the Convey system, we would expect to have a faster hardware implementation than the single thread software.

# 7 Conclusions and Recommendations

We present our final conclusions and recommendations for future work in this chapter.

## 7.1 Conclusions

The impact of NGS technologies in bioinformatics has been hailed as a revolutionary development, with the sequencing costs reducing at a faster rate than the more popular Moores law. This has paved way to the possibility of personalised genomics and more effective medical research. NGS also has brought with it increased computational demands. The thesis presents the basic problem of aligning millions of fragments of DNA, called reads, obtained from sequencing against a large reference genome.

Existing software aligners are described and the increasingly popular aligner, Burrows Wheeler aligner tool is described. The new algorithm of Burrows Wheeler aligner tool, called BWA-MEM is described in detail. The aim of the thesis is to produce an hardware accelerated version of the application, on Convey super computing platform. The Convey HC-2 platform consists of four Xilinx Virtex-6 LX760 FPGAs as co processors, and the hardware architectures proposed are for targeted for them.

BWA-MEM is a very dynamic and memory intensive application, which makes use of the Burrows Wheeler transform to index the large reference sequence so that it can fit on memories of conventional CPUs. The algorithm makes use of FMD index to the index the BWT, while also providing a faster way to align the reads against the reference. FMD index acts as a better alternative to the earlier FM index, since both the read and the reverse complement of the read can be aligned simultaneously.

Various research works of relating to hardware acceleration of BWT based aligners are also introduced, with a brief comparison between them. Vivado HLS tool was used to implement the hardware architecture for the FPGAs. Various helper tools were also developed to facilitate the analysis and development process. A detailed analysis of the application was performed and two possibilities of improvement were identified as the smith waterman functions and the memory bottleneck. The thesis aimed to overcome the memory bottleneck of the application, with designs developed to alleviate its effect.

Implementation procedure using Vivado HLS and methodology followed during the development are discussed. Two architectures are proposed, which are termed as Base architecture and Batch architecture. The Base architecture represents the initial Vivado HLS ported version of the application with a few enhancements. The Batch architecture is developed with a view to provide improvements by hiding the memory latency of the system. The architectures remain reconfigurable, supporting different read lengths, different number of cores, various read coverages and batch sizes. The architectures remain functionally similar to the software. The Batch architecture provides the possibility to hide the memory latency of the Convey memory system by restructuring occurrence array access functions to support pipelined memory requests.

The implementations were synthesised in Vivado and simulated in Modelsim, using synthetic reads generated using wgsim tool. Measurements were performed for different read lengths and single core performance was compared with single thread software run times. The base architecture, on average, remains slower than the software by about 22 times and the Batch architecture is slower by about 20 times. The batch architecture does not provide the intended speed up in performance. This is due to the fact that, the variance in the run time behaviour of different reads alleviates the performance gains of the architecture. In a batch of reads, all the reads are stalled for the next step of processing until the read with the longest matches has finished the current stage of processing. This proves detrimental to the performance gains of the Batch architecture. There are however possibilities in the current implementation to pipeline the design, to achieve faster execution times. By scaling the number of cores, we can fit upto 16 cores on the four FPGAs, limited only by the available memory ports to the Convey memory system. This is expected to provide a linear speed up over the current designs.

## 7.2 Recommendations for Future Work

The recommendations for performance improvement of current designs are discussed in this section, while also providing suggestions for looking into new architectures.

- **Smith Waterman core**: The current implementation of Smith waterman core is in Vivado HLS. But various works are already present, to have a faster manual hardware implementation of the core. This would present increased resource utilisation, but with manual implementation we could be able to provide a faster smith waterman core than the current version.

- **Pipelining**: We have already discussed the possibilities of pipelining at the current design. This is bound to provide better performance than the current implementation.

- **Batch version of suffix array access functions**: Current implementation has only batched the memory accesses to occurrence array. Performance improvements can be expected by batching the suffix array access functions as well.

- **Resource constrained model**: The Batch architecture supports arbitrary number of batch sizes for all the sub modules of the MEM_ALIGN1_CORE_PART1 core. The current measurements were performed by keeping the batch size constant across all the sub modules. It remains to be seen, whether different number of batch sizes at different levels of core, provides benefit. Also the resource utilisation would then need to be taken into account as well. Having a resource constrained model of the architecture, which would provide optimum configuration numbers based on resource utilisation would benefit the further enhancements to the architecture.

- **Alternative architectures**: The current implementation has mapped the entire worker1 code branch onto the FPGA, even though the bottleneck remains just at the memory access functions. Perhaps developing further software models, would help the analysis of, at which level in the software, the hardware must be cut out from.

- **Investigation of the effect of options**: Further investigation of the effect of various options of the application on the proposed architectures, also needs to be carried out to have a comprehensive understanding of the architectures.

# Bibliography

[1] W. KA, "Dna sequencing costs: Data from the nhgri genome sequencing program (gsp)," 2014. [Online]. Available: www.genome.gov/sequencingcosts

[2] H. Li, *BWA MEM poster*, February 6th, 2014. [Online]. Available: http://www.homolog.us/blogs/blog/2014/02/06/bwa-mem-poster-heng-li/

[3] C. C. Corporation, *The Convey HC-2 Computer - Architectural Overview*, 2012(accessed September 12, 2014). [Online]. Available: http://www.conveycomputer.com/index.php/download_file/view/143/142

[4] X. Inc, *Vivado Design Suite User Guide - High Level Synthesis*, December 19, 2013. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_4/ug902-vivado-high-level-synthesis.pdf

[5] J. Zhang, H. Lin, P. Balaji, and W.-C. Feng, "Optimizing burrows-wheeler transform-based sequence alignment on multicore architectures," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, May 2013, pp. 377–384.

[6] M. Baker, "Next-generation sequencing: adjusting to data overload," *nature methods*, vol. 7, no. 7, pp. 495–499, 2010.

[7] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with bwa-mem," *arXiv preprint arXiv:1303.3997*, 2013.

[8] A. M. Maxam and W. Gilbert, "A new method for sequencing dna," *Proceedings of the National Academy of Sciences*, vol. 74, no. 2, pp. 560–564, 1977.

[9] X. Zhou, L. Ren, Y. Li, M. Zhang, Y. Yu, and J. Yu, "The next-generation sequencing technology: a technology review and future perspective," *Science China Life Sciences*, vol. 53, no. 1, pp. 44–57, 2010.

[10] J. S. Reis-Filho *et al.*, "Next-generation sequencing," *Breast Cancer Res*, vol. 11, no. Suppl 3, p. S12, 2009.

[11] I. inc, *HiSeq X Ten for Population-Scale Sequencing*, 2014(). [Online]. Available: http://www.illumina.com/systems/hiseq-x-sequencing-system/system.ilmn

[12] T. J. Treangen and S. L. Salzberg, "Repetitive dna and next-generation sequencing: computational challenges and solutions," *Nature Reviews Genetics*, vol. 13, no. 1, pp. 36–46, 2011.

[13] A. Hatem, D. Bozdağ, A. E. Toland, and Ü. V. Çatalyürek, "Benchmarking short sequence mapping tools," *BMC bioinformatics*, vol. 14, no. 1, p. 184, 2013.

[14] T. D. Wu and S. Nacu, "Fast and snp-tolerant detection of complex variants and splicing in short reads," *Bioinformatics*, vol. 26, no. 7, pp. 873–881, 2010.

[15] S. Misra, R. Narayanan, S. Lin, and A. Choudhary, “Fangs: High speed sequence mapping for next generation sequencers,” in *Proceedings of the 2010 ACM Symposium on Applied Computing.* ACM, 2010, pp. 1539–1546.

[16] H. Li, J. Ruan, and R. Durbin, “Mapping short dna sequencing reads and calling variants using mapping quality scores,” *Genome research*, vol. 18, no. 11, pp. 1851–1858, 2008.

[17] B. Langmead, C. Trapnell, M. Pop, S. L. Salzberg *et al.*, “Ultrafast and memory-efficient alignment of short dna sequences to the human genome,” *Genome Biol*, vol. 10, no. 3, p. R25, 2009.

[18] P. Ferragina and G. Manzini, “Opportunistic data structures with applications,” in *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on.* IEEE, 2000, pp. 390–398.

[19] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang, “Soap2: an improved ultrafast tool for short read alignment,” *Bioinformatics*, vol. 25, no. 15, pp. 1966–1967, 2009.

[20] M. Burrows and D. J. Wheeler, “A block-sorting lossless data compression algorithm,” 1994.

[21] J.W.Peltenburg, “Hardware acceleration of short-read mapping with the burrows-wheeler aligner,” Master’s thesis, 2014.

[22] H. Li and R. Durbin, “Fast and accurate short read alignment with burrows–wheeler transform,” *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[23] R. Grossi and J. S. Vitter, “Compressed suffix arrays and suffix trees with applications to text indexing and string matching,” *SIAM Journal on Computing*, vol. 35, no. 2, pp. 378–407, 2005.

[24] H. Li, “Exploring single-sample snp and indel calling with whole-genome de novo assembly,” *Bioinformatics*, vol. 28, no. 14, pp. 1838–1844, 2012.

[25] M. Farrar, “Striped smith–waterman speeds database searches six times over other simd implementations,” *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.

[26] E. Fernandez, W. Najjar, and S. Lonardi, “String matching in hardware using the fm-index,” in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on.* IEEE, 2011, pp. 218–225.

[27] Y. Xin, B. Liu, B. Min, W. X. Li, R. C. Cheung, A. S. Fong, and T. F. Chan, “Parallel architecture for dna sequence inexact matching with burrows-wheeler transform,” *Microelectronics Journal*, vol. 44, no. 8, pp. 670–682, 2013.

[28] J. Arram, K. H. Tsoi, W. Luk, and P. Jiang, “Hardware acceleration of genetic sequence alignment,” in *Reconfigurable Computing: Architectures, Tools and Applications.* Springer, 2013, pp. 13–24.

[29] E. B. Fernandez, W. A. Najjar, S. Lonardi, and J. Villarreal, "Multithreaded fpga acceleration of dna sequence mapping," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on.* IEEE, 2012, pp. 1–6.

[30] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, Jun. 1982. [Online]. Available: http://doi.acm.org/10.1145/872726.806987

[31] H. Li, *Manual Reference Pages - samtools*, July 05, 2011. [Online]. Available: http://samtools.sourceforge.net/samtools.shtml

[32] ——, *wgsim*, 2011(accessed September 12, 2014). [Online]. Available: https://github.com/lh3/wgsim

[33] G. R. Consortium, *Human genome assembly data*, December 24, 2013. [Online]. Available: http://www.ncbi.nlm.nih.gov/projects/genome/assembly/grc/human/data/

[34] E. Vermij, "Genetic sequence alignment on a supercomputing platform," Ph.D. dissertation, M. Sc. Thesis, 2011.