Parallel architecture for DNA sequence inexact matching with Burrows-Wheeler Transform

Yao Xin, Benben Liu, Biao Min, Will X.Y. Li, Ray C.C. Cheung, Anthony S. Fong, Ting Fung Chan

Department of Electronic Engineering, City University of Hong Kong, Hong Kong SAR, China

School of Life Sciences and Hong Kong Bioinformatics Centre, The Chinese University of Hong Kong, Hong Kong SAR, China

ABSTRACT

The Burrows-Wheeler Transform (BWT) based methodology seems ideally suited for DNA sequence alignment due to its high speed and low space complexity. Despite being efficient in exact matching, the application of BWT in inexact matching still has problems due to the excessive backtracking process. This paper presents a hardware architecture for the BWT-based inexact sequence mapping algorithm using the Field Programmable Gate Array (FPGA). The proposed design can handle up to two errors, including mismatches and gaps. The original recursive algorithm implementation is dealt with using hierarchical tables, and is then parallelized to a large extent through a dual-base extension method. Extensive performance evaluations for the proposed architecture have been conducted using both Virtex 6 and Virtex 7 FPGAs. This design is considerably faster than a direct implementation. When compared with the popular software evaluation tool BWA, our architecture can achieve the same match quality tolerating up to two errors. In an execution speed comparison with the BWA algorithm, our design outperforms a range of CPU platforms with multiple threads under the same configuration conditions.

ARTICLE INFO

Article history:
Received 28 September 2012
Received in revised form
20 May 2013
Accepted 22 May 2013

Keywords:
DNA sequence alignment
Inexact matching
Burrows-Wheeler Transform
FPGA

1. Introduction

DNA sequence alignment is a fundamental task in modern bioinformatics [1]. It has been extensively utilized in both basic and applied biological research. Complete sequencing of an organism makes it possible to explore and understand the valuable information encoded in genomes. In recent years, there has been a shift from the traditional Sanger sequencing approach to Next Generation Sequencing (NGS) technology. NGS, introduced in 2004, has revolutionized the landscape of genomic research [2].

NGS is also called high-throughput or massively parallel sequencing. It is capable of generating hundreds of millions of short sequences in each run. These short sequences vary in length between 20 and 200 DNA bases, which are also called short reads. This recent NGS technology presents a significant challenge for computing the huge volume of data produced by sequencing machines. Traditional alignment tools like the Smith–Waterman [3] and BLAST [4] are not suitable to process huge databases. The extensive adoption of NGS techniques transforms the sequence alignment process into a short-read mapping problem: short reads are mapped onto a reference genome before all subsequent steps in the analysis pipeline take place.

These types of alignment may be exact—tolerating no errors—or inexact—tolerating mismatches or gaps. Given the large number of reads to handle, the last three years have witnessed the rapid development of numerous sophisticated algorithms to tackle the read-mapping problem [5]. These algorithms can be categorized into two major types according to the indexing methods [6]: algorithms based on the hash table (hasing the reads or the reference genome) such as MAQ [7], SSAHA [8], BFAST [9] and algorithms based on Burrows-Wheeler Transform (BWT) such as Bowtie [10], BWA [11] and SOAP2 [12].

In some applications, the BWT-based algorithms require a significantly smaller memory footprint than the hash-indexing methods. Meanwhile, it is efficient and fast for exact matching because of being independent of reference length. Thus, BWT-based methods have been increasingly used in areas of pattern matching and DNA sequence alignment. One representative application is the BWA tool [11], which extends the exact matching to the inexact one.

Due to the fact that the ever-growing amount of sequencing data results in a great computational demand, the general-purpose CPU is not a suitable platform. The attention now has been shifted to parallel platforms, such as multi-core multi-thread processors, graphics processing units (GPUs), and Field Programmable Gate Arrays (FPGAs). Among these parallel platforms, FPGAs can bring more benefits. They provide optimal reconfigurability and

Please cite this article as: Y. Xin, et al., Parallel architecture for DNA sequence inexact matching with Burrows-Wheeler Transform, Microelectron. J (2013), http://dx.doi.org/10.1016/j.mejo.2013.05.004
maximum design space to satisfy different levels of parallel granularity. FPGAs are also competitive for future embedded and portable systems because of their low power consumption. The product of MinION— a USB device DNA sequencer—from Oxford Nanopore Inc. [13] indicates the future demand of portable bio-sequence analysis tools.

Due to the challenges imposed by the irregular data access mode, there is only limited literature on the parallel architecture of FPGA [14] or GPU [15–19] based on BWT. The designs in [14,15] are targeting exact matching, while the work in [15–19] supports inexact matching in the GPU. However, there is no FPGA-based hardware architecture for BWT-based inexact matching tolerating both mismatches and gaps. In practice, inexact matching supporting mismatches or gaps has more application value compared with mere exact matching. There may, however, be mismatches or gaps for possible alignments, due to sequencing errors or inherent genetic variations. Variations of human DNA sequences can affect the ways human beings develop diseases and respond to pathogens, drugs and vaccines. Furthermore, the most common genetic variation, the single-nucleotide polymorphism (SNP), will likely be a key enabler in personalized medicine [20].

To explore the possibility of maximal parallelization of inexact matching algorithms, we have designed and implemented a hardware architecture for DNA sequence inexact alignment with FPGA, based on the basic algorithm from [11]. Our architecture is capable of finding all possible suffix array (SA) intervals (introduced in Section 3.2) for each short read, tolerating two errors including mismatches and gaps. The search of SA intervals is the most time-consuming part for BWT-based methods, and we think this process has more potential for parallelization. Thus, our architecture focuses only on this essential task, instead of the complete alignment process. As for the conversion of SA interval to original positions for each occurrence in the reference genome, this task can be completed by software and is not addressed here.

In this paper, we utilize hierarchical tables to enumerate possible conditions when performing an inexact search. The hierarchical table makes the recursive algorithm more suitable for hardware. Instead of a single-base extension (i.e., processing one DNA base in each step), a dual-base consecutive extension can parallelize the search process and reduce the computation burden. Therefore, the hardware architecture in this paper is based on the dual-base extension method and is implemented on Xilinx Virtex 6 (XC6VLX365T) and Virtex 7 (XC7VX980T) FPGA, respectively. Finally, we compare our design with BWA [11]—a state-of-the-art gap-tolerant software tool—executed on different CPU platforms. With the same search options, experimental results show that our architecture is superior in execution speed and can achieve the same search quality compared with the essential ah process in BWA. The major contributions of this paper are as follows:

1. The hardware parallel architecture is presented for BWT-based inexact matching which can handle both mismatches and gaps. This design can be extended to implement algorithms with irregular data accessing mode.
2. The algorithm has been modified to be suitable for hardware, while maximal parallelism has been exploited to enhance performance. To achieve this, we have utilized a dual-base proceeding method based on BWT. To our knowledge, this procedure has not been used in previous research.
3. Instead of a search trie, we employ a hierarchical table to enumerate all possible inexact combinations. A search of the literature indicated that this procedure is used for the first time in the present research.

The rest of the paper is organized as follows: Section 2 describes the work relevant to hardware design for DNA sequence alignment. Section 3 introduces the inexact search algorithm with BWT. Section 4 suggests different approaches to implement the recursive algorithm. Section 5 describes our hardware architecture and its components in detail. Section 6 analyzes the resource utilization with different implementation options. Section 7 presents performance evaluation and comparison results. Section 8 summarizes this paper.

2. Related work

The most frequently used techniques for traditional DNA sequence alignment are the Smith-Waterman [3] and BLAST [4]. The FPGA-based hardware architectures for Smith-Waterman usually employ systolic arrays [21,22]. In the case of emerging short-read mapping algorithms, only several hardware architectures have been published. This might be due to the complexity of algorithms and irregular data access mode.

The hardware designs in [23,24] are based on the hash-table method. Relying on a powerful hybrid reconfigurable platform, a complex architecture is constructed in [23]. This architecture is based on the indexing solution of hashing reference genomes in order to obtain more sensitive mapping results. The design finds all the seeds for a single read, locates in the reference where the seeds occur, then performs the Smith–Waterman alignment at these locations. This design is able to considerably accelerate mapping reads to the human genome, as compared with the BFAST [9] or the Bowtie [10] software. Although the procedure can perform inexact matching at very high speed, the hardware cost is relatively high, and the design implementation is quite complex.

BWT-based algorithms are memory footprint efficient compared with hash table based ones without affecting the speed. Even so, the research literature on hardware architecture is limited. In [14], the first FPGA-based hardware architecture with BWT for exact pattern matching is presented. It is an efficient architecture and has a relative good performance for exact matching. However, mere exact matching in DNA sequence alignment plays a limited role. Moreover—because of the larger search space—inexact matching requires exponentially increasing computation time compared with exact matching.

Apart from algorithms using indexing, there is also a kind of architecture based on the brute-force method [25,26]. By using direct comparison, it allows a freely adjustable character-mismatch threshold for alignment between reads and the reference database. This kind of architecture is an efficient example of coarse parallelism, which can scale up easily with FPGA clusters. It is also simple to implement without the need of storing reference genomes in the hardware. However, the speed of the brute-force method depends on the length of the reference, and it does not support gap alignment. Table 1 summarizes the differences among the aforementioned hardware architectures. Our proposed architecture is also included in this table.

<table>
<thead>
<tr>
<th>Table 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comparison of current hardware architectures.</strong></td>
</tr>
<tr>
<td><strong>Existing work</strong></td>
</tr>
<tr>
<td>Index method</td>
</tr>
<tr>
<td>Inexact matching</td>
</tr>
<tr>
<td>Gap alignment</td>
</tr>
<tr>
<td>Complexity</td>
</tr>
<tr>
<td>Memory efficiency</td>
</tr>
</tbody>
</table>
3. The Burrows-Wheeler Transform and inexact matching

The Burrows-Wheeler Transform (BWT) is a reversible loss-less transformation algorithm which was initially used in data compression [27]. Some of the short-read mapping tools employ BWT to index reference genomes. This section introduces BWT and the inexact matching algorithm based on it.

3.1. Burrows-Wheeler Transform

Let us consider a reference string \( R = a_1a_2...a_m \) of length \( m \) as the input to the BWT algorithm wherein all characters belong to the alphabet \( \Sigma \). For the DNA sequence, the \( \Sigma \) is \{A, C, G, T\}. To perform BWT, the string is first appended with a special symbol \( \$ \) which does not belong to \( \Sigma \) but is lexicographically smaller than all the characters in \( \Sigma \). Then the new string is rotated cyclically for one symbol in each step, and a matrix of size \((m+1) \times (m+1)\) is thus constructed with each row being the rotated string in each step. Each row represents each possible suffix of \( R \). Afterwards, all these rotations are sorted lexicographically. The BWT of \( R \) is the combination of the last characters of sorted rows, which is named BWT(\( R \)). The suffix array is an array of integers that indicate the original start positions of these suffixes permuted in lexicographical order.

In search algorithms based on BWT, two auxiliary data structures \( C \) and \( Occ \) generated from \( BWT(R) \) are required. \( C(b) \) denotes the total number of characters in \( R \) which are lexicographically smaller than the character \( b \). \( Occ(b, i) \) is the number of occurrences of character \( b \) in \( BWT(R) \) in the range of \([1, i]\).

Separate \( Occ \) tables for each base \( \{A, C, G, T\} \) are required for DNA sequence alignment. It would be a considerable burden on memory to store such a large amount of index data. To reduce the memory footprint, the \( Occ \) tables are always compacted by only storing a fraction of the complete ones. When the search is performed, the value needed is recovered through on-the-fly calculation with the BWT string BWT(\( R \)). As in \([11]\), only the values of \( Occ(., N) \) are stored in memory where \( N \) is multiple of 128. In our architecture, the BWT index datasets—including BWT string and the \( Occ \) are pre-calculated in the computer by software and transferred into Block RAMs of FPGA. Thus, we do not consider the complete hardware implementation of BWT in the following. Moreover, since the suffix array is needed only in converting SA intervals (introduced in Section 3.2) to original positions in reference genomes, the architecture does not need to contain the suffix array.

3.2. Backward search with BWT

In order to search a string \( P \) according to a reference \( R \), it is necessary to introduce the concept of SA interval. A SA interval for a substring of \( R \) represents a range of possible locations the substring occurs in the sorted suffix matrix \([11]\). The interval is denoted as \([first, last]\) in this paper. \( first \) is the first location of the substring appearing in the suffix matrix while \( last \) is the last location \([28]\).

To determine if a short sequence is a substring of the reference, the backward search algorithm is employed: the search starts from the end of short sequence and proceeds each time one character backward (i.e., one base for DNA reads). The interval \([first, last]\) then updates into different locations in the matrix according to newly increased suffix with a new character added \([14]\). This backward-search procedure does not end until the first character has been processed. The new position of the SA interval is calculated by following equations:

\[
last_{new} = C(b) + Occ(b, last_{old}) - 1 + 1
\]

The interval is valid only if \( first \leq last \), or—in other words—the sequence is a substring of the reference. Otherwise, the interval does not correspond to actual positions in the reference. This search algorithm is employed in exact matching.

3.3. Inexact matching with BWT and problems

The inexact matching refers to searching for reads that match with indels (gaps)—or mismatches—according to a given reference. In such a case, aligned sequences might not be completely identical to any substrings of the reference. The BWT-based method utilizes the backtracking algorithm to extend the exact search to the inexact search. This is performed by enumerating possible mismatches or inserting gaps and then backtracking and performing an exact search to find the possible matches. Algorithm 1 shows the algorithm of the inexact search for SA intervals of substrings in the reference that match the short read \( P \) \([11]\). \( n \) is the error threshold, and \(|P|\) denotes the length of \( P \).

Algorithm 1. Inexact match searching algorithm \([11]\).

\[
\begin{align*}
\text{Inexmatch} & : \text{function performs a depth-first search (DFS) to recursively calculates the SA intervals of substrings that match \( P \)} \\
\text{Input} & : \text{P, i, n, first, last} \\
\text{Output} & : \text{Inexmatch \((P, i, n, first, last)\)} \\
\text{Initialization:} & \text{first} - 1, last \rightarrow |P|, z \rightarrow 0, i \rightarrow |P| \\
\text{Inexmatch} & (P, i, n, first, last) \\
1: & \text{if } z > n \text{ then} \\
2: & \quad \text{return } \phi \\
3: & \text{end if} \\
4: & \text{if } i < 0 \text{ then} \\
5: & \quad \text{return } ([first, last]) \\
6: & \text{end if} \\
7: & \text{result} \rightarrow \phi \\
8: & \text{//For deletion from } P \\
9: & \text{result} \rightarrow \text{result} \cup \text{Inexmatch}(P, i-1, z+1, \text{first, last}) \\
10: & \text{for each } b \in \{A, C, G, T\} \text{ do} \\
11: & \quad \text{first} \rightarrow C(b) + Occ(b, first_1) + 1 \\
12: & \quad last \rightarrow C(b) + Occ(b, last) \\
13: & \text{if } first \leq last \text{ then} \\
14: & \quad \text{//For gap insertion to } P \\
15: & \quad \text{result} \rightarrow \text{result} \cup \text{Inexmatch}(P, i, z+1, \text{first, last}) \\
16: & \text{if } b = P[i] \text{ then} \\
17: & \quad \text{//Exact match} \\
18: & \quad \text{result} \rightarrow \text{result} \cup \text{Inexmatch}(P, i-1, z, \text{first, last}) \\
19: & \quad \text{else} \\
20: & \quad \text{//For mismatch} \\
21: & \quad \text{result} \rightarrow \text{result} \cup \text{Inexmatch}(P, i-1, z+1, \text{first, last}) \\
22: & \text{end if} \\
23: & \text{end if} \\
24: & \text{end for} \\
25: & \text{return } \text{result}
\end{align*}
\]
process can be depicted in prefix trie [11]. In each node of the search trie, before processing a base, we hypothesize that there are nine possible routes to pursue: one exact match, three mismatch (or substitution) conditions, four corresponding to insertions (by inserting a character in read to align with the reference—which could be A, C, G, T) and one corresponding to deletion (by deleting a character from read to align with the reference). Therefore, the search space for inexact matching is considerably larger than for exact matching. Furthermore, tolerating gaps instead of merely allowing mismatches further enlarges possible search space.

Therefore, the existing problems associated with inexact matching are the inefficient recursive function and the search process.

4. Non-recursive solution

4.1. Software approach

The Recursive method cannot be easily and efficiently realized in the hardware. For a software approach, the BWA tool utilizes a heap-like data structure storing all partial hits. This procedure enables it to implement a breadth-first search (BFS) rather than DFS [11]. However, the BFS requires huge memory usage, which is impossible for FPGAs. Furthermore, another essential contribution of BWA tool is the backtracking-boundary method. The workload process can be depicted in prefix trie [11]. In each node of the search trie, before processing a base, we hypothesize that there are nine possible routes to pursue: one exact match, three mismatch (or substitution) conditions, four corresponding to insertions (by inserting a character in read to align with the reference—which could be A, C, G, T) and one corresponding to deletion (by deleting a character from read to align with the reference). Therefore, the search space for inexact matching is considerably larger than for exact matching. Furthermore, tolerating gaps instead of merely allowing mismatches further enlarges possible search space.

Therefore, the existing problems associated with inexact matching are the inefficient recursive function and the search process.

4. Non-recursive solution

4.1. Software approach

The Recursive method cannot be easily and efficiently realized in the hardware. For a software approach, the BWA tool utilizes a heap-like data structure storing all partial hits. This procedure enables it to implement a breadth-first search (BFS) rather than DFS [11]. However, the BFS requires huge memory usage, which is impossible for FPGAs. Furthermore, another essential contribution of BWA tool is the backtracking-boundary method. The workload process can be depicted in prefix trie [11]. In each node of the search trie, before processing a base, we hypothesize that there are nine possible routes to pursue: one exact match, three mismatch (or substitution) conditions, four corresponding to insertions (by inserting a character in read to align with the reference—which could be A, C, G, T) and one corresponding to deletion (by deleting a character from read to align with the reference). Therefore, the search space for inexact matching is considerably larger than for exact matching. Furthermore, tolerating gaps instead of merely allowing mismatches further enlarges possible search space.

Therefore, the existing problems associated with inexact matching are the inefficient recursive function and the search process.
one table. This array is called a stack array in our architecture. Each stack array is divided into two parts: the first part contains entries with zero or one error, while the second part has two errors. Different parts employ different extension modes, which are described below.

The architecture of each processing element (PE) containing a storage unit and computing unit is shown in Fig. 3. The storage unit is a chain of register stack arrays. Any subsequent array in this chain is generated by extending the entries of the previous one. For entries in each stack array, there is a status array to record the search status and the next base index to proceed. The computing unit employed to calculate SA intervals by looking up the BWT index data, which is loaded in advance. Each BWT index set includes two kinds of extension module (discussed below). The Block RAM memory is arranged in each PE to store the BWT index data, which is loaded in advance. Each BWT index set consists of the compressed Occ tables of A, C, G, T respectively and the BWT string required to construct the remaining elements in Occ tables.

5.2. Search process

Given a short-read P with length \( |P| = l \), the inexact search process starts with the last two characters \( P[l], P[l−1] \) to produce the first stack array. Each entry in this array is updated with a calculated SA interval value, which corresponds to a certain case in Fig. 2. The next step is to extend from the first valid entry in this array to produce subsequent stack arrays, and to continue. If this process has reached the end of the search with a newly generated stack array, it terminates the extension and returns to the previous stack array, starting the search from the next non-empty entry. Otherwise, a similar procedure restarts from the first non-empty entry in this array according to the status array. After all entries in all stack arrays have been processed, the search is done. When the base index in latest array becomes zero and SA interval is valid, the result is output.

In practice, the dual-base extension is only performed for the entries of first part in stack arrays (see the middle section in Fig. 3), as each entry in this part may have multiple extension outputs. On the other hand, the entries of the second part have reached the maximal error number and can only proceed along the base index to produce one element at most. All entries in this part are put in the calculation pipeline and perform a single-base extension in the Single-base Pipelined Extension module (see the bottom section in Fig. 3).

Moreover, the stack array chain only stores the first part of the stack array in order to conserve resources. As illustrated in Fig. 4, the extension of two parts proceeds in two parallel processes simultaneously by using two sets of Block RAMs. In process 1, when the extension reaches the end of the short sequence, it backtracks to the next entry of the previous stack arrays and extends again in the aforementioned dual-base manner. By contrast, the second part searches until the end of the read by employing only one reusable stack array and continuously using a single-base extension.

The current architecture is designed for reads having an even number of bases. However, it can be extended to include odd-length reads by covering more conditions. Specifically, when approaching the end of the search in both scenarios, there are possible conditions requiring the single-base extension. As the architecture has both a dual-base extension module and a single-base extension module, these conditions are handled with minimal difficulties.

5.3. Computing units: extension modules

The computing unit consists of two modules: the Dual-base Extension module and the Single-base Pipelined Extension module. The former one is used to extend the first part of stack array entries in a dual-base manner. This is shown in Fig. 5.

As in Fig. 5, the Dual-base Extension module takes full advantage of a SA Interval Calculation pipeline. The structure of the pipeline is described in next section, and the process of reusing the pipeline is demonstrated in Fig. 6. In the first stage of dual-base extension, one calculates the set of first level intervals \( [first_1, last_1] \) extended with A, C, G, T from the initial input interval \( [first_0, last_0] \). After filtration, the remaining valid ones are continuously input to the pipeline to obtain the dual-base extended interval \( [first_2, last_2] \) of 16 combinations. Because there exists data dependency between \( [first_1, last_1] \) and \( [first_2, last_2] \), the first stage is a major bottleneck in the dual-base extension while the second stage is more efficient.

An indicating process, happening at the same time as the calculation, is carried out to build an indicating array for dual-base combinations, depending upon the input base index and error level. This array indicates the subsequent combination for each possible entry in the new stack array. When the calculation is done, the valid interval results are then updated to the entries of the newly produced stack array, in accordance with the indicating array. The detailed time study for dual-base extension is shown in Fig. 7. It takes 37 clock cycles to complete this procedure, while the indicating, updating and filtration process for all entries are performed parallelly within only one or two clock cycles, given the parallel property of FPGAs. The Single-base Pipelined Extension module utilizes another set of calculation pipeline. The control and use of the pipeline to perform a single-base extension is relatively simple and is not elaborated in this paper.

5.4. SA Interval Calculation pipeline

The essential part of the computing unit is the SA Interval Calculation module. This module is a fully pipelined system and is employed by two extension modules. The entire pipeline is
presented in Fig. 8. In this pipeline, the first step is to generate memory access addresses for Occ and BWT string memory with an input SA interval, while the subsequent major task is to calculate separate count of A, C, G, T in the segment of BWT string read from Block RAMs. This segment is needed to recover the elements between adjacent sampled Occ values. We use three level adder trees to do the calculation.

To make the recovery of the Occ original value efficient, we have designed a custom data structure for the BWT string in RAM memory. As Fig. 9 shows, according to the Occ sample rate, N (128)...
characters of the BWT string are stored in each address. Using this structure, all bases to rebuild the Occ elements in each step can be accessed in one clock cycle. Furthermore, we only need to perform a summation with up to \(N/2\) characters rather than \(N\), thus saving computing cost. The compressed Occ elements are selected according to the access location in the BWT string memory.

5.5. Excessive search prevention

We also incorporate the \(D(i)\) array from [11] into our architecture to further limit the excessive search process. Before the inexact search, \(D\) calculation module is able to determine the

Fig. 5. Dual-base extension module.

Fig. 6. Usage of pipeline in dual-base extension.

Fig. 7. Time study for dual-base extension. The unit is the number of clock cycles.

Fig. 9. each sampled Occ element and its corresponding location range are labeled in same shadow level.

5.5. Excessive search prevention

We also incorporate the \(D(i)\) array from [11] into our architecture to further limit the excessive search process. Before the inexact search, \(D\) calculation module is able to determine the

Please cite this article as: Y. Xin, et al., Parallel architecture for DNA sequence inexact matching with Burrows-Wheeler Transform, Microelectron. J (2013), http://dx.doi.org/10.1016/j.mejo.2013.05.004
reference sequence for the PEs. The public RAMs contain the BWT index of the reverse reference sequence. Each set consists of separate RAMs for the BWT string and the dual-base extension while the other set is for single-base extension.

5.6. Memory structure

In this module, only the first level SA interval is calculated and then checked for validity; z counter calculates the lower limit of error number in \( \{1, \text{index}\} \) according to comparator output, and records the \( z \) into the \( D_i(\cdot) \) array. The entire process is controlled by a finite state machine (FSM). Overall, the resource consumption of the \( D_i(\cdot) \) calculation module is trivial, except for the arrangement of another set of Block RAMs to maintain the BWT index of the reverse reference sequence.

5.6. Memory structure

Each PE contains two sets of internal Block RAMs; one set is for dual-base extension while the other set is for single-base extension. Each set consists of separate RAMs for the BWT string and the compressed \( \text{Occ} \) tables of A, C, G, T respectively. There is also a public set of dual-port Block RAMs which is shared by adjacent PEs. The public RAMs contain the BWT index of the reverse reference sequence for the \( D_i(\cdot) \) array calculation and the reverse search for complemented short reads. Each PE also contains a \( C \) table, which is stored in registers rather than in Block RAMs. The detailed structure is depicted in Fig. 11.

5.7. Parallel architecture with multiple PEs

The inexact search architecture achieves parallelization in scalable numbers of PEs. As each PE is capable of processing short reads independently, and each has its own index table to lookup, the scaling up of multiple PEs is not a difficult task. This is shown in Fig. 12. Before the search, the RAM write-control unit uniformly loads the BWT index data into the Block RAMs of each PE. The search-control unit loads the short reads into each PE, monitors the status, receives finish signals from PEs and loads new reads into them.

6. Resource utilization

In our architecture, one of the factors that affect the hardware resource consumption lies in the number of register stack arrays implemented, as a longer read requires more stack arrays to buffer the temporary SA intervals calculated on the fly. The read of length \( l \) requires at most \( l/2 \) stack arrays in our architecture.

Another factor is the Block RAM which determines the size of the BWT index table. The memory structure in our architecture was shown in Section 5.6. In order to obtain a resource-usage balance, the size of RAM containing a BWT string is set to 2M bits. In our design, we encode each DNA base into 2 bits, so the maximal length of the reference genome that can be introduced in our architecture is 1M bases. We set the data-width of Occ table RAM, SA interval and entries in stack arrays to 20 bits according to the reference length.

6. Resource utilization

In this section, extensive evaluation experiments are conducted. Our architecture is first compared to the direct resource consumption lies in the number of register stack arrays implemented, as a longer read requires more stack arrays to buffer the temporary SA intervals calculated on the fly. The read of length \( l \) requires at most \( l/2 \) stack arrays in our architecture.

Another factor is the Block RAM which determines the size of the BWT index table. The memory structure in our architecture was shown in Section 5.6. In order to obtain a resource-usage balance, the size of RAM containing a BWT string is set to 2M bits. In our design, we encode each DNA base into 2 bits, so the maximal length of the reference genome that can be introduced in our architecture is 1M bases. We set the data-width of Occ table RAM, SA interval and entries in stack arrays to 20 bits according to the reference length.

6. Resource utilization

In our architecture, one of the factors that affect the hardware resource consumption lies in the number of register stack arrays implemented, as a longer read requires more stack arrays to buffer the temporary SA intervals calculated on the fly. The read of length \( l \) requires at most \( l/2 \) stack arrays in our architecture.

Another factor is the Block RAM which determines the size of the BWT index table. The memory structure in our architecture was shown in Section 5.6. In order to obtain a resource-usage balance, the size of RAM containing a BWT string is set to 2M bits. In our design, we encode each DNA base into 2 bits, so the maximal length of the reference genome that can be introduced in our architecture is 1M bases. We set the data-width of Occ table RAM, SA interval and entries in stack arrays to 20 bits according to the reference length.

6. Resource utilization

In our architecture, one of the factors that affect the hardware resource consumption lies in the number of register stack arrays implemented, as a longer read requires more stack arrays to buffer the temporary SA intervals calculated on the fly. The read of length \( l \) requires at most \( l/2 \) stack arrays in our architecture.

Another factor is the Block RAM which determines the size of the BWT index table. The memory structure in our architecture was shown in Section 5.6. In order to obtain a resource-usage balance, the size of RAM containing a BWT string is set to 2M bits. In our design, we encode each DNA base into 2 bits, so the maximal length of the reference genome that can be introduced in our architecture is 1M bases. We set the data-width of Occ table RAM, SA interval and entries in stack arrays to 20 bits according to the reference length.

6. Resource utilization

In our architecture, one of the factors that affect the hardware resource consumption lies in the number of register stack arrays implemented, as a longer read requires more stack arrays to buffer the temporary SA intervals calculated on the fly. The read of length \( l \) requires at most \( l/2 \) stack arrays in our architecture.

Another factor is the Block RAM which determines the size of the BWT index table. The memory structure in our architecture was shown in Section 5.6. In order to obtain a resource-usage balance, the size of RAM containing a BWT string is set to 2M bits. In our design, we encode each DNA base into 2 bits, so the maximal length of the reference genome that can be introduced in our architecture is 1M bases. We set the data-width of Occ table RAM, SA interval and entries in stack arrays to 20 bits according to the reference length.

6. Resource utilization

In our architecture, one of the factors that affect the hardware resource consumption lies in the number of register stack arrays implemented, as a longer read requires more stack arrays to buffer the temporary SA intervals calculated on the fly. The read of length \( l \) requires at most \( l/2 \) stack arrays in our architecture.

Another factor is the Block RAM which determines the size of the BWT index table. The memory structure in our architecture was shown in Section 5.6. In order to obtain a resource-usage balance, the size of RAM containing a BWT string is set to 2M bits. In our design, we encode each DNA base into 2 bits, so the maximal length of the reference genome that can be introduced in our architecture is 1M bases. We set the data-width of Occ table RAM, SA interval and entries in stack arrays to 20 bits according to the reference length.

6. Resource utilization

In our architecture, one of the factors that affect the hardware resource consumption lies in the number of register stack arrays implemented, as a longer read requires more stack arrays to buffer the temporary SA intervals calculated on the fly. The read of length \( l \) requires at most \( l/2 \) stack arrays in our architecture.

Another factor is the Block RAM which determines the size of the BWT index table. The memory structure in our architecture was shown in Section 5.6. In order to obtain a resource-usage balance, the size of RAM containing a BWT string is set to 2M bits. In our design, we encode each DNA base into 2 bits, so the maximal length of the reference genome that can be introduced in our architecture is 1M bases. We set the data-width of Occ table RAM, SA interval and entries in stack arrays to 20 bits according to the reference length.
implementation of inexact search. Afterwards, both simulation and real datasets are utilized to make a thorough comparison between our design and the software tool.

7.1. Comparison with single-base architecture

In order to verify the efficacy of our architecture, we have also developed a hardware architecture merely based on the single-base extension, which is similar to that of the dual-base search, except that the stack arrays are established based on information provided in the table in Fig. 1. Thus, it is simpler to implement. This architecture represents the direct implementation of inexact search without high-level parallelism and an efficient pipeline.

We utilize three experimental short reads of different lengths together with the BWT index of a reference sequence—which we develop—to evaluate the performance in extreme cases. The length of the short reads is 32, 36 and 40 respectively. Using this dataset, we test the performance of matching short sequences to the reference index within two errors, with no limitation for the result number. Two architectures are all driven by a 100 MHz clock, with only one PE implemented.

Fig. 14 shows the comparison of run time necessary to find all possible SA interval locations for each short read in the reference index. For the performance of the inexact search, the speed of our architecture is about 4 times faster than that of direct implementation for all three reads of different lengths. Besides execution speed, the single-base design consumes even more slice resources, as it requires twice the number of stack arrays in the dual-base one. Therefore, the architecture only using single-base extension is not very efficient to implement.

7.2. Comparison with software tool

We also compare the dual-base architecture implementation with the BWA [11] software tool to perform inexact matching for short reads according to a reference genome. BWA is a popular software tool based on BWT, and is capable of aligning reads and allowing both mismatches and gaps. It is continuously being

![Fig. 11. Memory structure in PE.](image1)

![Fig. 12. Scaling up of multiple PEs.](image2)

![Fig. 13. Resource usage with increasing number of PEs.](image3)

<table>
<thead>
<tr>
<th>Table 2 Resource utilization of single PE.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>Stack array number: 20</td>
</tr>
<tr>
<td>Slice LUTs</td>
</tr>
<tr>
<td>RAMB36E1</td>
</tr>
<tr>
<td>Max Freq</td>
</tr>
<tr>
<td>Stack array number: 40</td>
</tr>
<tr>
<td>Slice LUTs</td>
</tr>
<tr>
<td>RAMB36E1</td>
</tr>
<tr>
<td>Max Freq</td>
</tr>
</tbody>
</table>
updated. Furthermore, our entire architecture is based on the algorithm of BWA, which itself is optimized for hardware. Therefore, the comparison with BWA software is the best way to see the speed enhancement of parallel architecture.

Moreover, it is difficult to make a comparison under same conditions for our design with other popular software tools like SOAP2 and Bowtie2. For example, the SOAP2 software does not permit gapped alignment for unpaired reads [29]. Since our design permits gapped search for unpaired reads, comparison with SOAP2 is not possible.

As for the Bowtie2 software, it combines Bowtie and dynamic programming (like the Smith-Waterman) [30]. A BWT-based search is used to find short, ungapped seed alignments rather than full alignments, in which the seeds are short substrings and are extracted from the read. Dynamic programming is then used to extend seed alignments into complete and gapped alignments, rather than to be used to align to the entire reference genome. And it shows that for the seed alignments of short reads, Bowtie2 can only perform an ungapped search with 0 or 1 mismatch. The default setting of seed substring length is 20. However, our design allows two mismatches or gaps through each entire read. Furthermore, the Bowtie2 hires a scoring scheme instead of setting the error number to 2 to compare with our architecture. We have tried to set a much shorter seed length to allow more errors throughout the read. The speed is very slow with this setting because dynamic programming controls the main part of the search process. All these aspects make a direct comparison between our design and Bowtie2 very difficult. These are the reasons why only the BWA tool is chosen as the reference for comparison.

We choose the version of bwa-0.6.2 for comparison in our experiment. The software tool is executed on three platforms: Dual-core Intel Core2 Duo @ 2.2 GHz with 2 MB cache, Quad-core Intel Core i5-2400 @ 3.10 GHz with 6 MB cache and Six-core Intel Xeon X5675 @ 3.06 GHz with 12 MB cache. For the hardware architecture, we choose Virtex 6 (XC6VLX365T) implementing 2 PEs and Virtex 7 (XC7VX980T) implementing 6 PEs to perform the inexact search.

Six different datasets of short read are selected which all came from the *Escherichia coli* genome, each dataset containing 1,000,000 reads. The length of the reference sequence which is also from the *Escherichia coli* genome is 1M-base. Table 3 lists the datasets utilized in our experiment.

The datasets with a “Simul” prefix are simulation reads generated from the *Escherichia coli* genome by using the wgsim software program [31]. This tool is used for simulating sequence reads from a certain reference genome, which is also employed in the experiment in [16]. This can generate reads with substitution errors and insertion/deletion in a customized ratio. The datasets with “Real” prefixes are real short reads selected from the NCBI Sequence Read Archive (SRA) [32].

As our architecture is designed to search the SA intervals for short reads, and this procedure corresponds to the aln process in BWA, so we only make a comparison between the execution time of the aln process in the software tool and the execution time of hardware implementation for the inexact search. The options for BWA aln are set to tolerate two differences including both mismatches and gaps, and the penalty for mismatch and gap is set the same, since our architecture treats mismatches and gaps equally. Because BWA supports the multiple threads option, a single thread together with maximal multiple threads are executed on different CPU platforms. Our architecture is configured to continue to find all possible SA intervals until repetitive results appeared.

### 7.2.1. Performance for simulation datasets

Four simulation datasets are generated to evaluate the performance of our architecture compared with the BWA. Different error rate options are chosen to make the comparison thoroughgoing for short sequences of the same length. For example, Simul-36-002 represents the dataset with a 2% base error rate and a 2% rate of mutations. Among the mutations of these four simulation datasets, the proportion of indel (gap) mutation is 10%, a relatively high ratio. We execute both the BWA tool and our architecture to search these simulation datasets according to the reference genome. The performance evaluation concentrates on two aspects: search quality and execution speed.

The search quality is commonly defined as how many reads are aligned to the reference genome. In our experiment, we use the percentage of short reads that are found with valid SA intervals within two errors to describe inexact search quality. The search quality of BWA tool and our architecture for the simulation datasets is presented in Table 4. Because the hardware architecture is constructed on the basis of the original algorithm of BWA, it is capable of achieving the same quality as BWA and finding all possible SA intervals within two errors, including mismatches and gaps.

#### Table 3

Datasets for performance evaluation.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Read length</th>
<th>Number</th>
<th>Error rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simul-36-002</td>
<td>36</td>
<td>1,000,000</td>
<td>2</td>
</tr>
<tr>
<td>Simul-36-001</td>
<td>36</td>
<td>1,000,000</td>
<td>1</td>
</tr>
<tr>
<td>Simul-72-001</td>
<td>72</td>
<td>1,000,000</td>
<td>1</td>
</tr>
<tr>
<td>Simul-72-0005</td>
<td>72</td>
<td>1,000,000</td>
<td>0.5</td>
</tr>
<tr>
<td>Real-36</td>
<td>36</td>
<td>1,000,000</td>
<td>–</td>
</tr>
<tr>
<td>Real-72</td>
<td>72</td>
<td>1,000,000</td>
<td>–</td>
</tr>
</tbody>
</table>

#### Table 4

Inexact search quality for simulation datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>BWA (%)</th>
<th>Hardware (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simul-36-002</td>
<td>88.20</td>
<td>88.20</td>
</tr>
<tr>
<td>Simul-36-001</td>
<td>97.63</td>
<td>97.63</td>
</tr>
<tr>
<td>Simul-72-001</td>
<td>87.95</td>
<td>87.95</td>
</tr>
<tr>
<td>Simul-72-0005</td>
<td>97.46</td>
<td>97.46</td>
</tr>
</tbody>
</table>
The charts in Fig. 15 show the execution time comparison for searching 1,000,000 simulation short reads of a length of 36 and 72 separately. Compared with the BWA tool running on various CPU platforms, our hardware architecture has a significant enhancement in speed. The Virtex 6 with 2 PEs well exceeds the i5 CPU with 4 threads, and is nearly two times more rapid when compared with 1 thread Xeon CPU. The execution time of the Virtex 7 implementing 6 PEs is significantly less than all other platforms that are multi-thread enabled. Compared with the most powerful CPU Xeon X5675 with 6 threads enabled in our experiment, the increase in speed reaches 1.5 times at most for a range of datasets with a difference in length and error rate.

### 7.2.2. Performance for real datasets

The search quality for real datasets attained by the BWA tool and hardware is presented in Table 5. As expected, the search quality is the same for both kinds of platforms.

<table>
<thead>
<tr>
<th></th>
<th>BWA (%)</th>
<th>Hardware (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-36</td>
<td>91.26</td>
<td>91.26</td>
</tr>
<tr>
<td>Real-72</td>
<td>95.37</td>
<td>95.37</td>
</tr>
</tbody>
</table>

We also evaluate the performance in execution speed using real datasets in the same fashion as the experiment with the aforementioned simulation datasets. The charts comparing the execution time are shown in Fig. 16. As in the case of the simulation datasets, we again obtain very good results for the speed: Virtex 6 performs well in comparison with all CPU platforms having a single thread, while Virtex 7 is superior to all CPU platforms with multi-thread operations. It is even more than twice as fast as the 6-thread Xeon platform. Furthermore, as our architecture scales with PE number, more PEs can be implemented in a more powerful chip or in FPGA clusters.

### 7.2.3. Time analysis for hardware architecture

Through a statistical analysis with the experimental datasets, we calculate the average time occupied by different steps of the algorithm in hardware (each PE) for each read. The pie chart is shown in Fig. 17. The algorithm consists of many steps, wherein the D calculation, dual-base extension and single-base extension make up the majority of the total time, while other ones including controlling and stack array updating are relatively trivial in time consumption. The dual-base and single-base extension are overlapped, since they are designed to run in parallel. Two rounds of D calculation are needed for each read: one D(·) array is for the short read according to the reference genome, and the other is for read’s complemented base sequence according to the reverse reference genome.
index from external memory to on-chip memory. Although such design would compromise the performance, the size limitation of on-chip Block RAM can be overcome. Minimizing resource utilization is another concern. An optimal solution that the length of short read being independent of the resource usage is the subject of our future research.

8. Conclusions

This paper proposes a novel hardware architecture to parallelize the inexact matching algorithm based on BWT, and implements it on FPGAs. Currently, the extendible architecture can handle two errors including both mismatches and gaps. Our major contributions include: (1) improving the original inefficient recursive algorithm using hierarchical tables; (2) parallelizing the inexact search process and constructing a parallel architecture by using the consecutive dual-base extension method; (3) evaluating inexact search process and constructing a parallel architecture by using the consecutive dual-base extension method; (4) building an index that can store the information of 6 PE per block and allow the architecture to handle up to 6 threads; (5) evaluating the hardware architecture with external memory. The current design is limited by the capacity of Block RAM. If external RAM is available, we can segment the BWT index, and dynamically load the segmented index from external memory to on-chip memory. Although such design would compromise the performance, the size limitation of on-chip Block RAM can be overcome. Minimizing resource utilization is another concern. An optimal solution that the length of short read being independent of the resource usage is the subject of our future research.

References


