Hardware-Acceleration of Short-Read Alignment Based on the Burrows-Wheeler Transform

Hasitha Muthumala Waidyasooriya, *Member, IEEE* and Masanori Hariyama, *Member, IEEE*

Abstract—The alignment of millions of short DNA fragments to a large genome is a very important aspect of the modern computational biology. However, software-based DNA sequence alignment takes many hours to complete. This paper proposes an FPGA-based hardware accelerator to reduce the alignment time. We apply a data encoding scheme that reduces the data size by 96 percent, and propose a pipelined hardware decoder to decode the data. We also design customized data paths to efficiently use the limited bandwidth of the DDR3 memories. The proposed accelerator can align a few hundred million short DNA fragments in an hour by using 80 processing elements in parallel. The proposed accelerator has the same mapping quality compared to the software-based methods.

Index Terms—Short-read alignment, genome mapping, Burrows-Wheeler alignment, FPGA accelerator

1 Introduction

Next Generation sequencers have a very high demand in bioinformatics due to the low-cost and high speed [1]. Computer-based genome sequence alignment has been widely used to construct a genome from the short-sequences (also known as short-reads) produced by the next generation sequencers. Fig. 1 shows the short-read alignment. Using the fact that the genomes of the same organism differ only slightly, short-reads are aligned to a reference genome in such a way that most bases are matched. Although this process is very simple, the alignment bottleneck is yet to be solved due to the huge quantities of the short-read data produced by the sequencers every day. The short-read data can be up to 600 Gbs (giga bases) which is very huge data amount for the present day computers to handle. Therefore, the existing software applications such as Maq [2], Bowtie [3], Burrows-Wheeler alignment (BWA) [4], etc require hours to days to align a whole genome.

Software applications used for the short-read alignment can be categorized into two groups. The first group of applications such as Maq [2], BFAST [5], and BLAST [6] use dynamic programming. In these applications, small parts of the short-reads called seeds are compared with the seeds of the reference genome. After the partial alignment is completed, they are combined to find the best alignment. These methods are usually good but take a large processing time. The second group of software applications such as Bowtie [3], BWA [4] and SOAP2 [7] use "Burrows-Wheeler Transform (BWT)" [8]. These methods are very fast and have a sufficient accuracy.

The main problem of BWT-based short-read alignment methods is the large data size. Usually the required data size is several times larger than the original reference genome data. BWT based string search is characterized by an unpredictable and highly irregular memory access pattern, which poses difficult challenges for the efficient implementation in CPUs or graphic processing units (GPUs). Due to the irregularity of the data access, it is difficult to store such huge data amount in the limited memory, let alone allowing efficient access for parallel processing while utilizing the available memory bandwidth.

In this paper, we propose an FPGA-based custom hardware accelerator for fast short-read alignment based on BWT. An FPGA is a reconfigurable LSI that contains millions of programmable logic gates [9]. The basic accelerator architecture is published in our previous works [10], [11]. This paper discusses the hardware architecture and memory access in detail. To efficiently utilize the limited bandwidth of the FPGA, we designed custom data paths that store all the address requests from multiple processing elements (PEs) in a pipeline, so that the memory access requests are issued continuously. We also exploit the special locality of the memory access to increase the cache hits. The huge data size is reduced by data encoding. A pipelined hardware decoder is proposed to decode the data efficiently. The proposed FPGA accelerator aligns over hundred million short-reads in an hour to construct a whole human genome. According to the experimental results, we achieved 21.8 times of speed-up compared to the BWA software that uses a four-core CPU. It is faster than many hardware-based implementations.

2 Short-Read Alignment Based on the Burrows-Wheeler Transform

BWA [4] is a widely used tool for the short-read alignment. It can be used for gapped-alignments also where a short-read is not exactly matched with the reference genome. Following are the reasons for the mismatches.

1) single nucleotide polymorphism (SNP): A nucleotide in the reference genome and short-read at the same position is different.
2) Deletion: A nucleotide in the reference genome is not present in the short-read.

• *The authors are with the Graduate School of Information Sciences, Tohoku University, Aoba 6-6-05, Aramaki, Aoba, Sendai, Miyagi 980-8579, Japan. E-mail: {hasitha, hariyama}@ecei.tohoku.ac.jp.*

Manuscript received 9 Dec. 2014; revised 2 June 2015; accepted 8 June 2015. Date of publication 10 June 2015; date of current version 13 Apr. 2016. Recommended for acceptance by R. Cumplido.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2444376

1045-9219 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

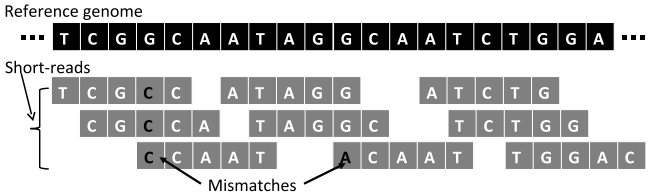


Fig. 1. Short-read alignment: short-reads (short sequences) of an unknown genome are aligned to a known (reference) genome in a such a way that most bases are matched. Most short-reads have all the bases matched with the reference, while some may have few mismatches.

3) Insertion: A short-read contains an additional nucleotide which has not been present in the reference genome.

Algorithm 1. Short-read Alignment Algorithm

```
CalculateD(W, D)
begin
  calculate D[i] for (0 ≤ i ≤ |W| − 1)
end
InexRecur(W, i, z, k, l, D)
begin
  if z < D(i) then
    return ψ
  end
  if i < 0 then
    return [k, l]
  end
  I = I ∪ InexRecur(W, i − 1, z − 1, k, l)
  for each a ∈ {A, C, G, T} do
    k_a = C(a) + O(a, k − 1) + 1
    l_a = C(a) + O(a, l)
    if k_a ≤ l_a then
      I = I ∪ InexRecur(W, i, z − 1, k_a, l_a)
      if a = W[i] then
        I = I ∪ InexRecur(W, i − 1, z, k_a, l_a)
      else
        I = I ∪ InexRecur(W, i − 1, z − 1, k_a, l_a)
      end
    end
  end
  return I
end
main()
begin
  CalculatedD(W, D)
  Suffix array interval I =
        InexRecur(W, |W| − 1, z, 1, |X| − 1, D)
  Aligned position = SA[I]
end
```

In this paper, we use the word "mismatches" to indicate the sum of all SNPs and indels (insertions and deletions). In this section, we briefly describe BWA using Algorithm 1. To explain the algorithm, let us consider the example shown in Fig. 2. We have a reference genome X and a short-read W as shown in Fig. 2a. The inputs are $C(.)$ shown in Fig. 2a, the occurrence array $O(.,.)$ shown in Fig. 2b and short-reads. The number of symbols that are lexicographically smaller than a is given by $C(a)$ where $a \in \{A, C, G, T\}$. The occurrence array is constructed by applying BW transform to the reference genome X. Please refer [8] for a detailed description on BW transform. BWA algorithm uses the "exact matching" method explained in [12]. According to [12], if a string W is a substring of the string X and $k(aW) \leq l(aW)$, string aW is also a substring of X where aW equals the string $\{a, W\}$. The terms k and l, given by Eqs. (1) and (2) respectively, are the lower and upper bounds of the suffix array interval of X

$$k(aW) = C(a) + O(a, k(W) - 1) + 1, \quad (1)$$

$$l(aW) = C(a) + O(a, l(W)). \quad (2)$$

Note that, the suffix array shown in Fig. 2b shows the corresponding positions to the reference genome, for the symbols of the BWT array. The BWT array is also shown in Fig. 2b

Fig. 3 shows the recursive search results of the short-read W in the "InexRecur" procedure. The position of a symbol in W, the number of mismatches allowed, the lower and upper bounds of the suffix array interval are given by i, z, k and l respectively. The number of symbols in W (or the size of W) is given by $|W|$. Fig. 3 shows all executions of the procedure "InexRecur" given in Algorithm 1. For example, the alignment result with one insertion has a suffix array interval of $[5, 5]$. According to Fig. 2b, Suffix array interval $[5, 5]$ refers to the position 3 (that is, $SA[5, 5] = 3$) in the reference genome X. Note that, the procedure "CalculateD" gives a lower bound $D[i]$ for the number of mismatches of $W[0, i]$. We recommend to refer [4], which is the original paper that proposes the BWA, for further details. Our intension of explaining the same BWA algorithm in this paper is to show what the inputs are, and what kind of calculations are required. The occurrence array data for the short-read alignment are created using the reference genome. In practical problems, the same reference genome data are used to align

Position	0	1	2	3	4	5	6
Reffrence genome (X)	C	C	T	G	A	G	$

Position	0	1	2
Short read (W)	C	G	A

a	A	C	G	T
$C(a)$	1	2	4	6

(a) Reference genome X, short-read W and $C(a)$ of X

Suffix array (SA)		BWT array	Occurrence array O(.,.)				
	Position in X		$	A	C	G	T
0	6	G	0	0	0	1	0
1	4	G	0	0	0	2	0
2	0	$	1	0	0	2	0
3	1	C	1	0	1	2	0
4	5	A	1	1	1	2	0
5	3	T	1	1	1	2	1
6	2	C	1	1	2	2	1

(b) Occurance array of X. It is constructed by counting the number of symbols up to each entry.

Fig. 2. Example of aligning the short-read W using X as the reference.

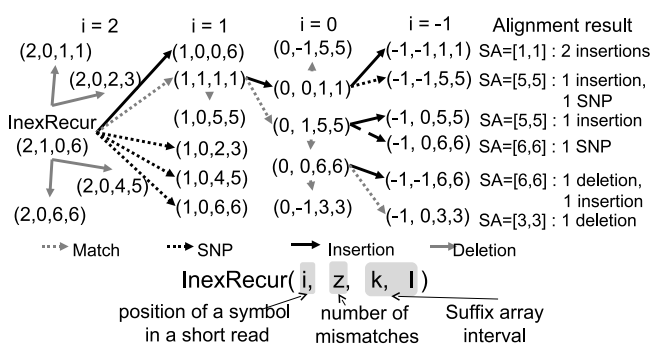


Fig. 3. Recursive search of the short-read W in the "InexRecur" procedure.

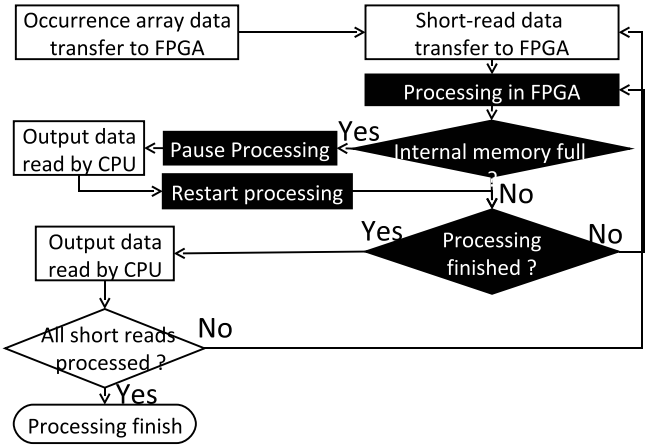


Fig. 4. Flow-chart of the short-read alignment process. Blocks shown in black background corresponds to the processing in FPGA. The other blocks corresponds to the processing in CPU. The processing in CPU are mainly control processing and data transfers while the alignment is done in FPGA.

different set of short-reads. Therefore, in this paper, we perform the Burrows-Wheeler transform and calculate the occurrence array beforehand and transfer the data to the FPGA for different alignments.

The most critical problem of the BWA algorithm is the enormous size of the occurrence array data of the reference genome. Since a human genome has over three billion symbols, there are three billion entries in the occurrence array. To represent this number, we need 32 bits. Therefore, each entry in the occurrence array needs 128 bits where, 32 bits each gives the number of A, C, G and T symbols. Total memory required to store the occurrence array is 128×3 billion bits, which is approximately 48 GBs. Since we use the reverse order of the reference genome data to determine the lower bound, we need another 48 GB memory. There is no FPGA board in the market that could hold 96 GB of data. One common way of dealing with such problems is to partially transfer the data to the memory. However, such a technique is impossible to use since the memory access pattern is data dependent and cannot predict which data are required next, before processing the current data. In this paper, we use the encoding scheme discussed in [4] to reduce the data amount. We propose a method to reduce the decoding overhead and propose a pipelined hardware decoder so that the data are decoded in every clock cycle after the pipeline is fully filled.

3 Acceleration of the Short-Read Alignment

3.1 Accelerator Architecture

Fig. 4 shows the flow-chart of the short-read alignment process. First, the occurrence array data are transfered to the two DDR3 memories in the FPGA board. Then two blocks of short-reads are transfered to the DDR3 memories. After that, the short-reads are processed in FPGA and the alignment results are written to the internal memory of the FPGA. When the internal memory is full, the processing is paused and the data are read-back from the FPGA. After processing the short-reads, new blocks of short-reads are transfered to the DDR3 memories. The accelerator can process unlimited number of blocks of short-reads sequentially.

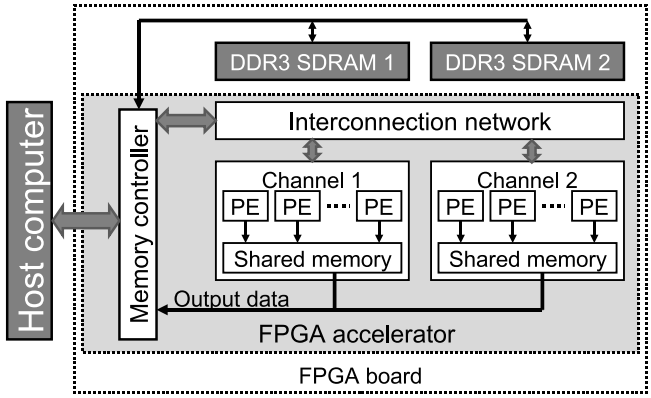


Fig. 5. Accelerator architecture. The inputs of the FPGA accelerator are stored in DDR3 memories. Output is write to the FPGA internal memory and read by the host computer.

The overall architecture of the accelerator is shown in Fig. 5. It consists of a memory controller and two groups of PEs belong to channel 1 and channel 2. A channel contains multiple PEs. The parallel data processing is achieved by executing different short-reads in parallel on those multiple PEs. The occurrence array data and the short-read data are transferred to the DDR3 memory. Each channel operates independent of each other. All PEs in a channel also works independently. There outputs are written to a shared memory. If there are multiple request at the same time to write output data, on request is granted by using a priority arbiter. The main reason of using the priority arbiter is that it is very simple and its implementation takes a very small logic area. The main disadvantage of the priority arbiter is that, the lower priority requests must wait for a long time to receive a grant. However, this problem does not have a big effect in the proposed accelerator due to the following reason. The search process in the "InexRecur" procedure is done for several thousands of cycles and just write one output to the memory. Therefore, between two memory access requests from the highest priority PE, there are several thousands of cycles. During that period, the other less priority PEs receive the grants to access the memory.

Structure of a PE is given in Fig. 6. It consists of a 32-bit adder, a comparator and pipeline registers to perform the calculations explained in Algorithm 1. After finishing one "InexRecur" procedure, a new one is loaded from the register file. In each "InexRecur" procedure, new calls to the same procedure are generated as explained in Algorithm 1. The

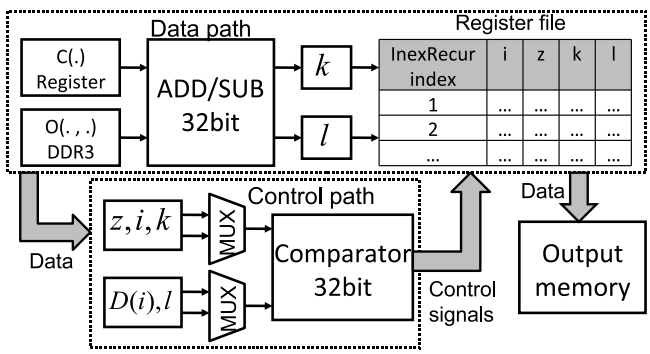
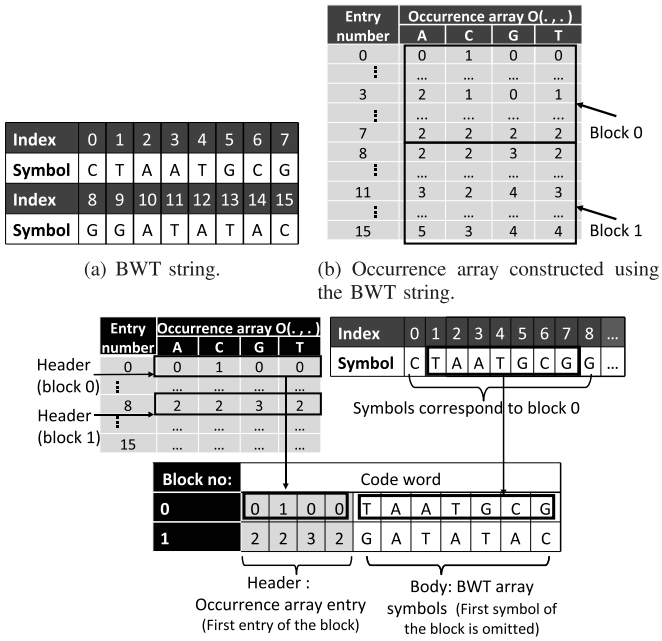


Fig. 6. Structure of a PE. The register file is used to store the parameters of the "InexRecur" procedure.



(a) BWT string.

(b) Occurrence array constructed using the BWT string.

(c) Code words of block 0 and 1.

Fig. 7. Example of the encoding scheme. A block of several entries is compressed to a one cord word.

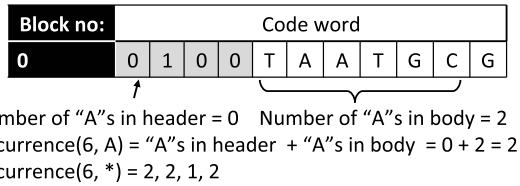


Fig. 8. Decoding a code word. The number of symbols in the body of the code word is added to the header to decode an entry.Although only the decoding process of symbol "A" is shown, it is similar for all the symbols.

parameters of such recursive calls are stored in the register file, so that we can keep a track of all the recursive calls. The "ADD/SUB" unit in PE is used to calculate the alignment positions given by Eqs. (1) and (2). The comparator and the control path do all the conditional branches in the "InexRecur" procedure. New short-reads are fed to the PEs after the old short-reads are aligned. The alignment result is read by the CPU. Unlike the CPU that has a complex floating-point ALU and very complicated control circuit, a PE is a very simple unit that specialized only to align a short-read. It is designed using minimum resources. Therefore, we can have a lot of PEs in the same FPGA to provide performance comparable to a computer cluster that has many CPUs.

To implement this architecture, we use an FPGA board that contains two 2 GB DDR3 SDRAMs, PCIe connector and an FPGA. The occurrence array data and the input short-read blocks are stored in DDR3 memories. One block contains 8.3 million short-reads which is approximately 500 MB large. There are 80 PEs in two channels where one channel contains 40 PEs. The outputs of a PE contains the details such as the short-read number, the suffix array interval, mapping quality, "CIGAR string" (this contains the positions of insertions and deletions), number of mismatches, etc. The output is read from the shared memory and stored in a binary file by the CPU and later converted to "sam" format. It is then converted to "bam" format using the software called "sam tools" [19]. This process is similar to the traditional usage of "BWA software" where it produces a binary "'sai" file which later converted to "sam" and then to "bam" formats.

3.2 Reducing the Occurrence Array Size by Data Encoding

The total memory capacity of the FPGA board is only 4 GB, where each memory can hold 2 GB of data. However, the occurrence array data size is 96 GB. To reduce this huge data amount, we employ the data encoding method proposed in [12]. To explain the encoding method, we consider the example in Fig. 7. The BWT string is shown in Fig. 7a. Fig. 7b shows the occurrence array constructed using the BWT string. It is constructed by counting the number of symbols up to each entry. We divide the occurrence array into two blocks where each block has eight entries. Then we assign a code word for each block. Fig. 7c shows the code word. It has two parts; the header and the body. The first occurrence array entry of a block is stored as the header. The BWT symbols of the block except the first symbol is stored in the body. For example, lets consider the "block 0". The first entry of the block (entry "0,1,0,0") is stored as the header and the symbols from the BWT string from index 1 to 7 are stored in the body.

We explain the decoding method using Fig. 8. Let us consider an example of obtaining the occurrence array entry 6 denoted by "Occurrence(6, *)". Since the entry 6 belongs to the block 0, we use the code word 0. From the first six symbols of the body of the code word, we count the number of "A,C, G,T" symbols. In this example, there are two "A"s, one "C", one "G" and two "T"s in the body. Then we add the symbol counts to the header. Therefore, Occurrence(6, *) is "2,2,1,2". Note that, to get an occurrence array entry, we need to decode only one code word and the decoding of the entire occurrence array is not required. In the human genome, there are three billion entries in the occurrence array. Each entry is 128 bit long. We divide the occurrence array into blocks of 64 entries. The first entry of each block is stored as the header and the rest of the 63 BWT array symbols are stored in the body. Therefore, we require 128 bits for the header and $63 \times 2\,bits$ for the body. As a result, one codeword is 254 bit large and contains the data of 64 occurrence array entries. The compression ratio is $254\,bits/(64 \times 128\,bits) \approx 0.03$. However, in the worst case, we have to count the occurrence of 63 symbols which takes a large hardware overhead.

To reduce the decoding overhead, we propose a slightly different encoding scheme in Fig. 9. Note that, we use the same example given by Figs. 7a and 7b. In the proposed encoding scheme, the occurrence array entry at the middle of a block is stored as the header. For example, we use the entry number 3 and 7 from the block 0 and 1 respectively as the headers. The body is the same as in the conventional scheme shown in Fig. 7c. The decoding is done differently to the conventional method by considering the following two scenarios.

Scenario 1: Obtaining an occurrence array entry comes before the header.
Scenario 2: Obtaining an occurrence array entry comes after the header.

Let us consider an example of obtaining "Occurrence (1, *)". The entry 1 comes before the entry 3 (the header). Therefore, this example belongs to the scenario 1. The

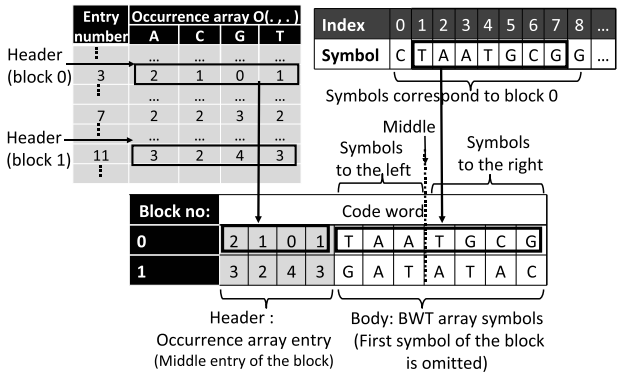
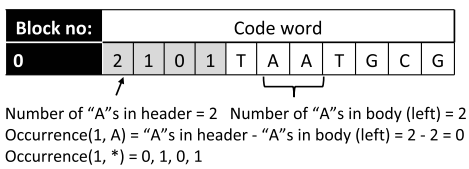


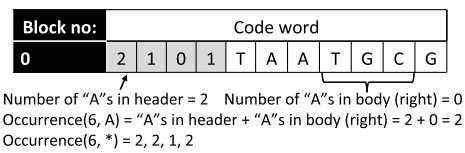
Fig. 9. Proposed encoding scheme. The occurrence array entry at the middle of a block is used as the header.

decoding is shown in Fig. 10a. We consider "three minus one" symbols from the middle to the left. Then we count the number "A,C,G,T" symbols and deduct the symbol count from the header to determine Occurrence(1, *). In this case, it is "0,1,0,1". Let us consider another example of obtaining "Occurrence(6, *)". Since the entry 6 comes after the entry 3 (the header), it belongs to the scenario 2. The decoding is shown in Fig. 10b. We consider "six minus three" symbols from the middle to the right and count the number of "A,C, G,T" symbols. By adding the symbol count to the header, 'Occurrence(6, *) is 6. Note that, the maximum symbol count in the proposed method is approximately half of that of the conventional method. Therefore, we can reduce the decoding overhead of the symbol count.

We explain the architecture of the hardware decoder using Fig. 11. The hardware module that decodes the occurrence array data of symbol "*A*" is shown in Fig. 11a. The decoder has a very simple architecture that consists of bit-shifters, adders, a population count (popcount) module, etc. The body of the code word that contains a part of the BWT string is decoded in to a 32bit word of 0's and 1's. If the symbol (in this case symbol "A") exists, we use 1, otherwise we use 0. Then we extract only the required bits. For example,

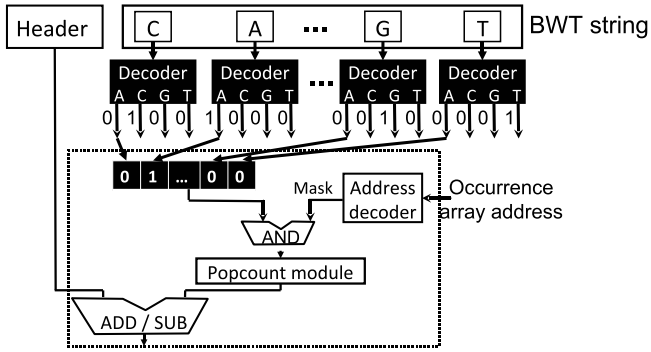


(a) Decoding an entry to the left of the header. The number of occurrences of each symbol in the body to the left is deduct from the header.



(b) Decoding an entry to the right of the header. The number of occurrences of each symbol in the body to the right is added to the header.

Fig. 10. Decoding a code word in the proposed encoding scheme. Since we count only the half of the occurrences, the counting overhead is reduced. Although only the decoding process of symbol "A" is shown, it is similar for all the symbols.



(a) Hardware decoder for the occurrence array data of symbol "*A*"

```
reg [31:0] x
x <= x – ((x >> 1) & 32'h55555555)
x <= (x & 32'h33333333) + ((x >> 2) & 32'h33333333)
x <= (x + (x >> 4)) & 32'h0f0f0f0f
x <= (x * 32'h01010101) >> 24
```

(b) Computations in the popcount module. Few additions, a subtraction, bitwise operations and a multiplication are required.

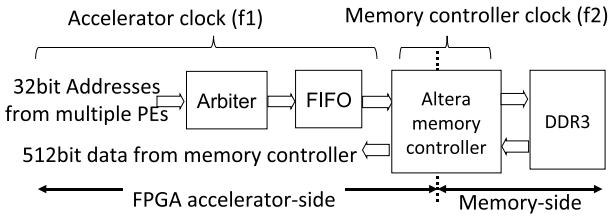
Fig. 11. Hardware decoder. Similar decoders are used to decode the symbols "C, G" and "T".

if we need only 16 bits starting from the least significant bit (LSB), we do the "bitwise AND" operation with a mask. In this case, the mask is 0x0000FFFF. Since the memory address corresponds to the occurrence array entry number, the mask is obtained by decoding the memory address. After the required bits are determined, we count the number of 1's using a popcount module. Finally, the number of occurrences are added to or subtracted from the header. There are many popcount methods available [13]. We use the popcount method shown in Fig. 11b, since it requires a small hardware overhead. Although it has the multiplication operation, we can perform fast multiplications in the FPGA using hardware-based multipliers (DSP units). The decoder has a latency of four clock cycles. However, it is fully pipelined, and an output is produced in every clock cycle after the pipeline is fully filled.

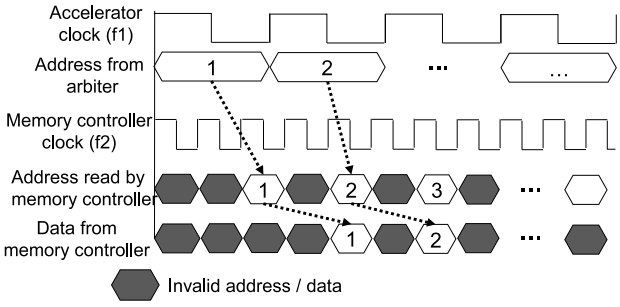
3.3 Efficient Utilization of the Limited Memory Bandwidth

3.3.1 *Efficient Bandwidth Utilization by Multiple Channels*

In the proposed architecture, we use 80 PEs to process the short-reads. Therefore, multiple PEs request access to the same DDR3 memory simultaneously. To grant access to a single request, we use an arbiter. The granted requests are stored in a FIFO, so that new requests can be issued seamlessly. Fig. 12 shows the data paths and memory access from multiple PEs. The data and the address paths between the accelerator and DDR3 memory is shown in Fig. 12a. The memory controller connected to the DDR3 memory handles all memory-side transactions such as data read/write, refresh, etc. It provides a 32 bit address bus and a 512 bit data bus to the FPGA-side. The memory controller reads the granted requests from the FIFO and sends the corresponding addresses to the DDR3 memory. Usually, the accelerator clock frequency f_1 is smaller than the memory controller



(a) The data paths between the PEs and the memory for the one channel implementation.

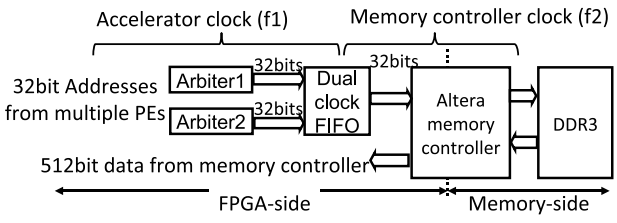


(b) Time chart of the memory access ($(f_1 < f_2)$).

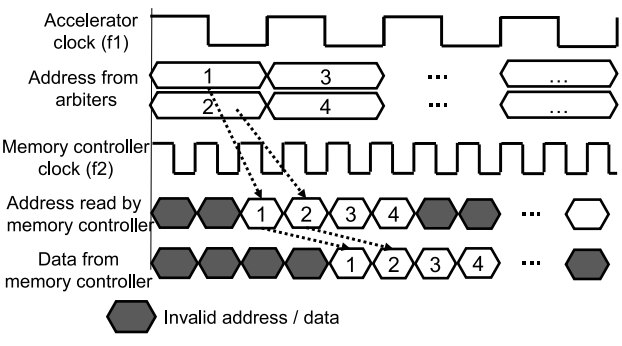
Fig. 12. Memory access using one channel. The memory access speed is decided by the slowest frequency f_1.

clock frequency f_2. Fig. 12b shows the time chart of the memory access when $f_1 < f_2$. The rate that the addresses are sent (by the PEs) is smaller than the rate that the memory controller is capable of receiving. Therefore, even a valid address is issued in every clock cycle from the arbiter, the data are received from the DDR3 at the same rate that the addresses are sent in. The maximum rate that the data are received is decided by the slower accelerator clock frequency (f_1).

We can solve this problem by using multiple channels where each channel contains a group of PEs and an arbiter. Fig. 13 shows an example where two channels are used in



(a) The data paths between the PEs and the memory for the two channel implementation.



(b) Time char of the memory access ($f_1 < f_2$)

Fig. 13. Memory access using two channels. The memory access speed is decided by $2 \times f_1$.

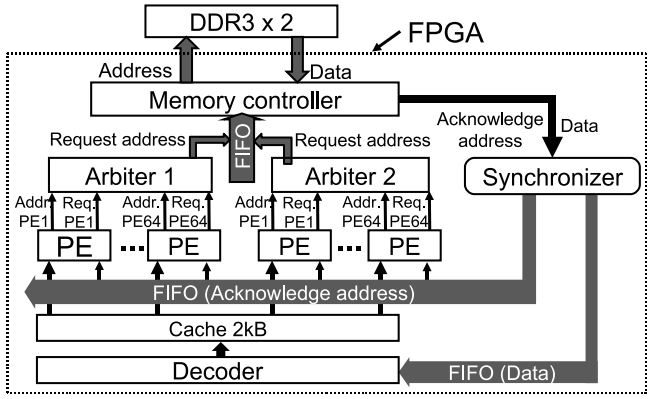


Fig. 14. The data paths of memory access using two channels.

parallel. The 32 bit addresses from the two arbiters are stored in a "dual clock FIFO". The write port of the FIFO is 64 bit large and operates at the clock frequency f_1. The read port of the FIFO is 32 bit large and operates at the clock frequency f_2. Therefore, two addresses of 32 bit each are written to the FIFO in parallel while one address is read by the memory controller. Fig. 12b shows the time chart of the data access when $f_1 < f_2$. Two address are written to the FIFO at f_1 and one address is read at f_2. The accelerator clock frequency is 85 MHz ($f_1 = 85\,\mathrm{MHz}$) and the memory controller clock frequency is 200 MHz ($f_2 = 200\,\mathrm{MHz}$). Therefore, $2 \times f_1 \leq f_2$, and the memory controller is capable of reading twice the amount of addresses compared to the situation where only one channel is used. Although it is theoretically possible to increase the memory access speed further by increasing the number of channels to three, it is practically difficult to achieve the optimal performance of the memory controller, that is one read from every clock cycle at f_2. Moreover, increasing the number of channels will also increase the hardware overhead. Therefore, in the proposed accelerator, we used only two channels.

The designed data path for two channel memory access is shown in Fig. 14. Multiple PEs send address requests in parallel to the arbiter. The arbiter allows one request to proceed in each clock cycle so that the FIFO is filled with addresses. Those addresses are sent one-by-one to the memory controller. We cannot determine how many cycles it takes to get the data after sending an address. The timing of the address sent and the data arrive varies for different requests. However, the order of the data coming out from the memory controller, and the order of their corresponding addresses are sent, is the same. Therefore, if we know when the first address is sent and the first data is arrived, we can synchronize the data with their corresponding addresses. We use a synchronizer circuit to align the data with their corresponding addresses. Note that, the requested address is sent back to all PEs along with the data so that each PE can determine whether the data are correspond to their request or not.

3.3.2 Memory Interleave

As explained in Section 2, the memory access in BWA is extremely irregular and unpredictable. However, extracting some features even in such an irregular pattern is the key to improve the performance. As shown in Algorithm 1, the short-read alignment process has two major procedures called "calculateD" and "InexRecur" executed one after the

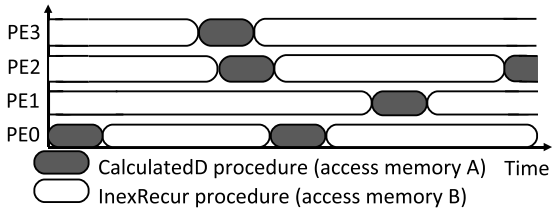


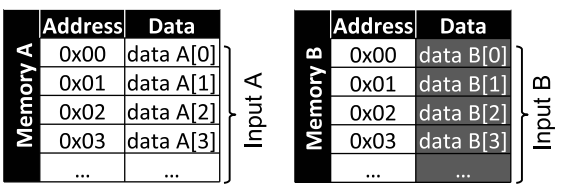
Fig. 15. The execution of Algorithm 1 by multiple PEs. Since the same procedure is executed by more than one PE simultaneously, multiple requests to access the same memory could be issued in parallel.

other. These procedures require two different sets of data, and we use two DDR3 memories on the FPGA board to store those. The memories are used as follows.

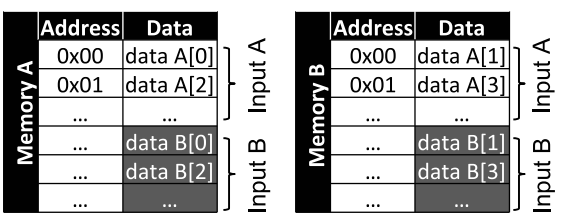
Memory A: Data required for "calculateD" ("data A").
Memory B: Data required for "InexRecur" ("data B").

In the proposed accelerator, each PE is capable of executing the Algorithm 1 independently using different short-reads as the inputs. The execution of Algorithm 1 by multiple PEs is shown in Fig. 15. Usually, the processing time of the "CalculatedD" procedure is smaller than that of the "InexRecur" procedure. Note that, the processing time of the "InexRecur" procedure depends on the short-read data, and the processing time is different for the different short-reads. Since, the "InexRecur" procedure takes a large processing time, different PEs may execute this procedure for different short-reads simultaneously. During the "InexRecur" procedure, multiple requests to access "memory B" could be issued in parallel by multiple PEs. However, the memory controller accept only one requests and the rest of PEs have to wait in a queue to access the memory. To solve this problem, we use memory interleave.

Fig. 16 shows the data allocation for interleave and non-interleave memory accesses. In the non-interleave memory access, the "data A" is allocated to the memory A and the "data B" is allocated to the memory B as shown in Fig. 16a. Therefore, if one PE wants data B[0] and another PE wants B[1] simultaneously, one PE has to wait while the other PE accesses the memory B. Fig. 16b shows the data allocation for interleave memory access. Part of the data A is allocated to the memory A while the other part is allocated to the memory B. Similarly, the data B is also allocated to both memories. As a result, if two PEs want the data B[0] and B[1] respectively, one PE can access memory A to get the data B[0] while the other PE can access memory B to get the data B[1]. Therefore, both PEs proceed with their computations without waiting in a queue to access the memory.



(a) Data allocation for non-interleave memory access. A data set is allocated to one memory.



(b) Data allocation for interleave memory access. A data set is allocated to multiple memories.

Fig. 16. Data allocation for interleave and non-interleave memory access. In the memory allocation for interleave memory access, both data A and B are distributed in multiple memories.

TABLE 1
Processing Time Reduction Due to Memory Interleave

Number of mismatches	Processing time (s)		Reduction %
	Non-interleave	Interleave	
0	82.6	63.4	23.2
1	104.6	83.3	20.3
2	151.6	140.4	7.4
3	1,635.3	922.3	43.6
4	24,117.0	13,027.7	45.9

Table 1 shows the comparison of interleave and non-interleave memory access. We conduct experiments using the short-read sample "DRR002191-1" obtained from [14]. The experiment is done for over 8.3 million bases. According to the results, we obtained a speed-up of over 20 percent by interleaving the memory. This speed-up increases with the maximum number of mismatches. Increasing the maximum number of mismatches will also increase the recursive calls to the procedure "InexRecur". As a result, data B is accessed more frequently than data A. If the memory interleave is not used, the access is concentrated on memory B, so that the processing time is determined by the access speed of a single memory. Interleaving the memory allows to access both memories evenly so that the processing time is reduced almost by half as shown in Table 1.

3.3.3 *Efficient Data Cache by Exploiting the Spacial Locality of the Memory Access*

Data caches are often used in CPUs to increase the memory access speed by serving the requests to the cached data many times faster than accessing the main memory. This method works well if the memory access pattern has a locality of reference in terms of temporal or spacial. As explained in Section 2, it is very hard to find a temporal locality for BWA since the memory access pattern is data dependent. However, we exploit the spacial locality of the memory access, and propose an efficient cache policy and its hardware architecture.

In Algorithm 1, there are two kind of memory requests; one is to access the lower bound k and the other is to access the upper bound l. Fig. 17 shows the suffix array interval against time. Note that, since the size of the suffix array interval changes from zero to billions, we draw the y-axis in log scale to show the graph clearly. The upper and lower bounds correspond to memory addresses. Therefore, when the suffix array interval is small, two addresses close to each other are accessed. That means, we have a spacial locality in memory access. According to Fig. 17, the suffix array interval is large at the beginning but gets smaller when the computation proceeds. When a unique match is found, both the

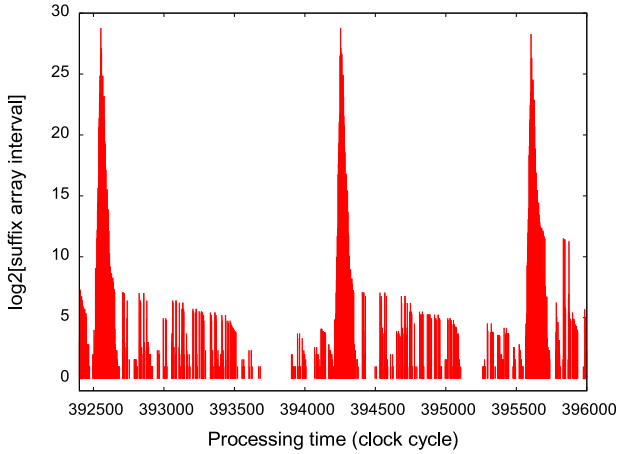


Fig. 17. The suffix array interval against the processing time. The suffix array interval is wide when the search begins, but gets narrower when the search proceeds. If a unique match is found, the suffix array interval is zero.

upper and the lower bounds must give the same value, so that the suffix array interval becomes zero. Now let see how we could exploit this property.

As explained in Section 3.1 we have encoded 64 occurrence array entries into a single 254 bit code-word. A word in a memory is 512 bits wide so that it contains two code-words. Therefore, accessing a single word in the memory gives us the access to 128 occurrence array entries. If the suffix array interval is smaller than 128, there is a greater possibility that we could find both the upper and the lower bounds of the suffix array interval in the same word. Therefore, we issue a request for the lower bound first and see whether we could find the upper bound also in the decoded data. If the upper bound is found in the decoded data, we do not have to access the memory again. This is like data caching in CPUs. We have to cache 512 bits to store the code-words, 128 bits to store the upper bound and another 128 bits to store the lower bound. Therefore, the required cache size is only 768 bits. This is significantly smaller compared to the cache size of the CPUs.

Table 2 shows the processing time comparison with and without the data cache. The processing time reduced by more than 25 percent. The maximum theoretical processing time reduction is 50 percent where both the upper and the lower bounds are obtained in a single memory access. According to Table 2, the processing time reduction increases with the number of mismatches. It reaches 40 percent when the number of mismatches is 4. This shows that the proposed method is very effective especially when the number of mismatches is large.

3.4 Exact Matching Using Reversed Reference Genome Data

As explained in Section 2 using Algorithm 1, the "CalculatedD" procedure is used to find a lower bound $D[i]$ for the number of mismatches of $W[0, i]$. An abstract version of the "CalculatedD" procedure is given in Algorithm 2. The strings X' and W' are the reverse of the reference genome X and the reverse of the short-read W respectively. The number of symbols in W and X (or the size of W and X) are given by $|W|$ and $|X|$ respectively. We have not shown the detailed computation of $D[i]$, and please refer [4] for the complete version of the "CalculatedD" procedure. In this paper, we just want to show that, if W' is a substring of X', $D[i] = 0$ for all i. If W' is a substring of X', then W is also a substring of X. That is, if $D[i] = 0$, there is an exact match. Moreover, the aligned position equals $|X| - SA'[I'] - |W|$, where I' is the suffix array interval in the reversed genome. The modified "main" procedure is shown in Algorithm 2. If $D[i] = 0$ for all i, we compute the aligned position directly from the suffix array interval in the reversed genome. Therefore, we do not call "InexRecur" procedure. If $D[i] \neq 0$, the aligned position is calculated similar to Algorithm 1 by calling "InexRecur" procedure.

TABLE 2
Processing Time Reduction Due to Data Caching

Number of mismatches	Processing time (s)		Reduction %
	No cache	With cache	
0	63.4	47.3	25.4
1	83.3	62.3	25.3
2	140.4	100.9	28.1
3	922.3	598.8	35.1
4	13,027.7	7,814.1	40.0

Algorithm 2. Short-Read Alignment Algorithm. Exact Matching Using Reversed Reference Genome data.

```
CalculateD(W, D)
begin
  i = 0 to |W| − 1: W'[i] = W[|W| − 1 − i]
  i = 0 to |X| − 1: X'[i] = X[|X| − 1 − i]
  for i = |W| − 1 to 0 do
    Compute [k', l'] using X'
    if {W'[i]...W'[|W| − 1]} is a substring of X' then
      D[i] = 0
    else
      Compute D[i] (note that, D[i] ≠ 0)
    end
  end
  return [k', l']
end
main()
begin
  Suffix array interval in the reversed genome I' =
        CalculatedD(W, D)
  if ∀i : D[i] == 0 then
    Aligned position = |X| − SA'[I'] − |W|
  else
    Suffix array interval I =
        InexRecur(W, |W| − 1, z, 1, |X| − 1, D)
    Aligned position = SA[I]
  end
end
```

Since the genomes of the same organism differ only slightly, most of the short-reads match exactly with the reference genome. Therefore, we can reduced the processing time by not calling the "InexRecur" procedure for the exact matches. Table 3 shows the processing time reduction using Algorithm 2. We achieved up to 47 percent of processing time reduction.

TABLE 3
Processing Time Reduction Due to the Exact Matching Using Reversed Reference Genome Data

Number of mismatches	Processing time (s)		Reduction %
	FPGA	Hybrid	
0	47.3	42.0	11.3
1	62.3	45.9	26.3
2	100.9	65.9	34.7
3	598.8	447.7	25.2
4	7,814.1	4,134.9	47.1

TABLE 4
Processing Time of the Hybrid Method

Number of mismatches	Processing time (s)		Reduction %
	FPGA	Hybrid	
0	42.0	42.0	0.0
1	45.9	60.7	−32.3
2	65.9	92.5	−40.4
3	447.7	160.8	64.1
4	4,134.9	219.1	94.7

3.5 Hybrid Processing Method Using CPU and FPGA

In real world problems, allowing less than two mismatches is enough to align some of the short-reads data samples. However, there are many data samples that have a considerable percentage of short-reads that could only be aligned considering a large number of mismatches. As we can see in Table 2, even with so many efforts, the processing time increases exponentially with the number of mismatches. To solve this problem, we analyze the processing time of different short-read alignments. Fig. 18 shows the processing time distribution. The x-axis is the number of cycles (or processing time) and the y-axis is the number of short-reads. According to Fig. 18, the most of the total processing time is spent to align less than 4 percent of the short-reads. In BWA software, complex but efficient algorithms are used to process these short-reads faster. However, a lot of hardware is required to implement those algorithms. FPGA has a limited hardware resources and most of those are already used in the proposed accelerator. To implement the algorithms that process only 4 percent of the resources, we have to sacrifice the resources that are been used to process the rest of the 96 percent short-reads. Such a method is not effective since it increases the total processing time.

To solve this problem, we consider hybrid processing method where both CPU and FPGA are used. We set a time (clock cycle) limit to process a short-read in the FPGA. If a hit is not found during the time limit, further processing of that short-read stops. Those unaligned short-reads are extracted and re-aligned in the CPU using BWA software. Table 4 shows the total time reduction due to the hybrid processing using FPGA and CPU. According to the results, a large processing time reduction is achieved when the number of mismatches is large. However, when the number of mismatches is small, the processing time is increased. Note that, the minus reduction percentages in Table 4 shows a processing time increase.

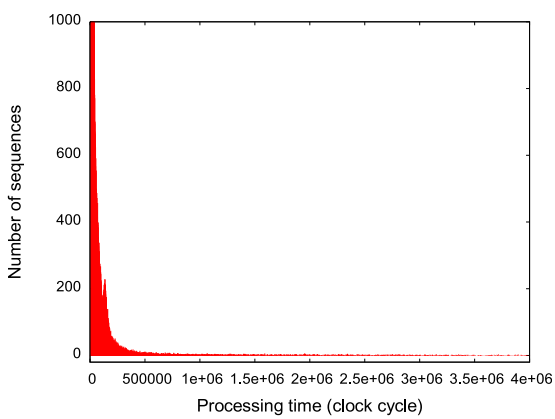


Fig. 18. Processing time distribution. Most short-reads require very small amount of clock cycles to find an alignment.

To find the cause for the processing time increase in smaller number of mismatches, we analyze the different component of the processing time. As shown in Table 5, the total processing time is composed of the processing times of the short-read alignment in FPGA, extraction of the unaligned short-reads and the re-alignment of the unaligned short-reads in CPU. To reduce the processing time of the hybrid method, we consider parallel processing using FPGA and CPU. The processing in FPGA accelerator is done in batches. One batch contains approximately 8.3 million short-reads. As shown in Fig. 19, after the short-read alignment of batch 1 is finished, the accelerator aligns the batch 2. Meanwhile, we extract the unaligned short-reads of the batch 1 from the FPGA output and realign those using the BWA software in CPU. Therefore, both FPGA and CPU are used in parallel. According to the experimental results, more than 96 percent of the short-reads in a batch are processed in the accelerator. The experiments are done for different number of mismatches using short-read samples with different lengths. Although the percentage of the short-reads processed by software is different for each sample, it does not exceed more than 4 percent. Therefore, it is safe to assume that, less than 4 percent of the short-reads are processed in CPU in every batch. Usually, the short-read alignment in FPGA is the most time consuming process. Therefore, the total processing time almost equals to the short-read alignment time in FPGA.

4 Evaluation

The proposed accelerator system has a host computer and an FPGA board. The proposed accelerator is designed using the FPGA board called DE5 [15] that contains an "Altera

TABLE 5
Processing Time Distribution in Hybrid Approach

Number of mismatches	Processing time in hybrid approach (s)		
	Short-read alignment in FPGA	Unaligned short-read	
		extraction in CPU	alignment in CPU
0	41.9	0	0
1	45.8	13.0	1.9
2	53.4	13.1	26.0
3	71.5	13.5	75.8
4	98.0	14.0	107.1

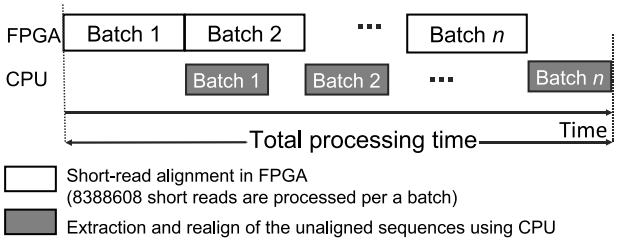


Fig. 19. Time chart of the hybrid processing. The processing in FPGA and CPU are done in parallel for different batches of short-reads.

TABLE 6
FPGA Resource Utilization

Module	LUT (%)	Registers (%)	DSP (%)	Memory Mbits (%)
PE array	135,834 (57.8)	111,871	144 (56.2)	32.0 (64.1)
DDR3	12,800 (5.5)	18,377	0 (0)	0.3 (0.6)
Other	41,526 (17.7)	48,185	0 (0)	1.1 (2.2)
Total	190,160 (81.0)	178,433	144 (56.2)	33.4 (66.9)

TABLE 7
Specifications of the Proposed Accelerator

Number of PEs	80
Memory	2 GB × 2 channels
Gonome size	less than four billion bases
Maximum number of mismaches	252 bases
Maximum short-read length	252 bases
Maximum amount of short-reads	unlimited

Stratix V 5SGXEA7N2F45C2 FPGA" and two 2 GB DDR3-SDRAMs. The DE5 board is connected to the host computer by the PCI express port. The CPU of the host processor is Intel core i7-3960x. The operating frequency of the accelerator is 85 MHz. Table 6 shows the resource usage. The FPGA accelerator uses 81 percent of the look-up-tables (LUTs) in the FPGA. Most of the resources are used by the PE array while 23 percent of the LUTs are used to design the DDR3 controller, PCIe controller and their data paths.

Table 7 shows the specifications of the designed accelerator. Note that, the short-read length is 252 bases. It is limited by word-length of the memory controller which is 512 bits. However, if we use more than one word to represent a short-read, we can increase the short-read length. For example, using two words to represent a short-read, we can increase the short-read length up to 507 bases. However, this will increase the processing time slightly since we have to access two words to read one short-read.

Fig. 20 shows the processing time against the number of PEs. The processing time reduces when the number of PEs increases. However, after around 40 PEs, processing time reduces only slightly. Therefore, when designing the architecture, we have given the priority to increase the efficiency of the memory access by allocating resources to the data decoding, pipelined data paths and data caches. After this goal is fulfilled, we implement many PEs as possible. Due to the limited hardware resources, we were able to implement only 80 PEs. It is impossible to allocate more PEs since such designed cannot be fit on to the FPGA, and the compilation process failed to give a valid mapping result.

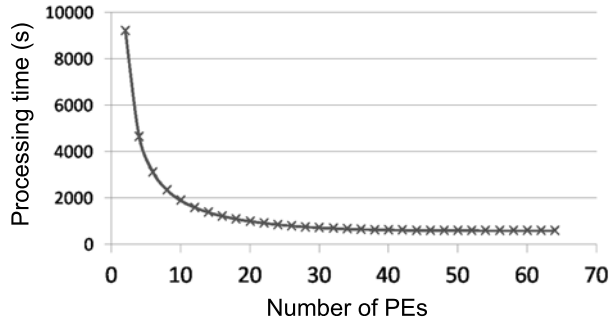


Fig. 20. Number of PEs versus processing time. The processing reduces slightly after using over 40 PEs.

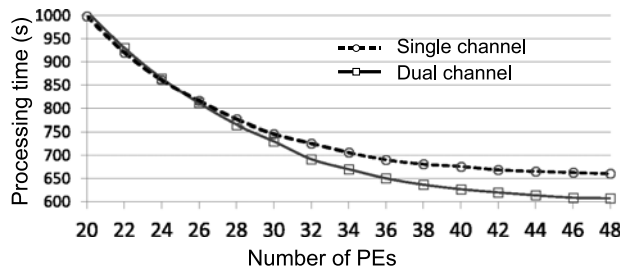


Fig. 21. Number of PEs in a channel versus processing time. The processing time difference between the single and dual channel implementations gets wider when the number of PEs increases.

Fig. 21 shows the processing time of single and dual channel implementations against the number of PEs. We use the same number of PEs in both implementations. For example, if the single channel uses 30 PEs, dual channel uses 15 PEs per a channel, so that the total is 30 PEs. According to the results, when the number of PEs is small, both implementations give similar processing time values. However, when the number of PEs is large, the processing time of the dual channel is smaller than that of the single channel. The processing time difference between the single and dual channel implementations gets wider when the number of PEs increases. For 48 PEs, the dual channel implementation reduces the processing time by 8 percent. As explained in Section 3.3.1, multiple PEs issue memory access requests simultaneously and the arbiter grants only one request per clock. Since the dual channel implementation uses two parallel arbiters, maximum of two memory access requests can be granted per a clock cycle. For the single channel implementation that uses only one arbiter, maximum of one memory access requests can be granted per a clock cycle. If the number of PEs are small, the number of memory access requests are also small. Therefore, very few requests are granted. As a result, the processing time is almost the same for both implementations. When the number of PEs are large, more requests are sent to the arbiters. In this case, two parallel arbiters grant more requests compared to the single arbiter. When more requests are granted, more data can be accessed. Therefore, the dual channel implementation is faster than the single channel implementation. Since we use a large number of PEs (80 PEs), the memory access speed is definitely faster in the dual channel implementation.

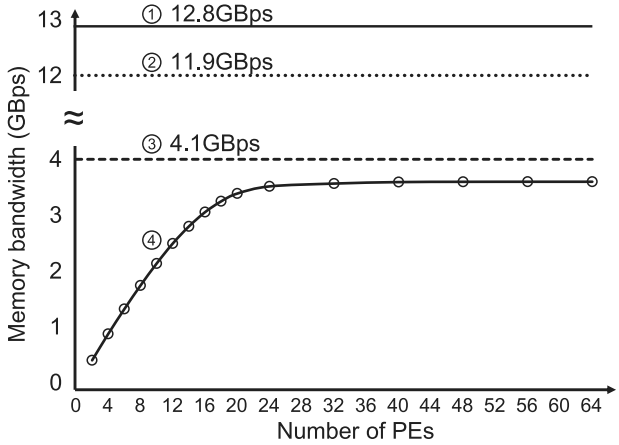


Fig. 22. Number of PEs versus memory read bandwidth. ① Theoretical bandwidth. It is calculated using the specification of the memory controller and DDR3 memory. ② Measured bandwidth using consecutive addresses. ③ Measured bandwidth using non-consecutive addresses. ④ Measured bandwidth of the proposed accelerator against different number of PEs.

Fig. 22 shows the memory bandwidth against the number of PEs. Initially the memory bandwidth increases linearly with the number of PEs. However, after using 16 PEs, the bandwidth increases slowly and comes to almost a constant value. When using 64 PEs, the memory bandwidth of the accelerator is measured as 3.7 GBps. In Section 2, we explained that the memory access of the short-read alignment is data dependent and unpredictable, so that the addresses are non-consecutive. The memory bandwidth using non-consecutive addresses is 4.1 GBps, and the proposed accelerator achieved more than 90 percent of this memory bandwidth. Therefore, the accelerator efficiently utilizes the limited memory bandwidth available for non-consecutive addressing.

However, as shown in Fig. 22, the bandwidth of non-consecutive addressing is only 32 percent of the theoretical bandwidth of the DDR3 memory. Therefore, the performance limiting factor could be either the memory controller or the DDR3 memory. To find this out, we measured the bandwidth using consecutive addresses and achieved 93 percent of the theoretical bandwidth. In this case, the memory addresses are sent at 200 MHz to the memory controller and valid data are received at almost the same rate. Therefore, we can say that the memory controller is capable of drawing most of the performance of the DDR3 memory. However, when using non-consecutive addresses, the valid data are not received at the same rate, although the address are sent to the memory controller in every clock cycle at 200 MHz. Also note that, one read from the memory controller gives 512 bits of data. That is, eight 64 bit words from the DDR3, where the word length of the DDR3 is 64bits. Therefore, the burst read access can be achieved for reading 512 bits, and it may not be the reason for the performance limitation. Based on these observations, we could say that the low bandwidth of the DDR3 memory for non-consecutive addressing is the performance limiting factor. To solve this problem, we have to use memories that have a larger bandwidth for non-consecutive addressing.

4.1 Comparison with the BWA Software

We compared the performance of the proposed accelerator against the conventional BWA software tool. The software version is bwa-0.7.5a [16]. The software is executed on an Intel Xeon ES-2643 (3.3GHz) processor that have four physical CPU cores. The software uses all four cores in parallel with 100 percent CPU usage. We used the hybrid method explained in Section 3.5, where most of the short-reads are aligned on the FPGA accelerator while the rest are re-aligned using the BWA software. The processing time of the proposed method includes the processing time of the data transfers also. It also includes the extraction and re-alignment time of the short-reads. Note that, as explained in Section 3.5, the extraction and re-alignment are done in parallel to the alignment on FPGA. The short-reads data are taken from [14] and the reference human genome data are taken from [17].

Table 8 shows the processing time comparison for single-end short reads. We have used different lengths of short-reads with different number of mismatches to make the evaluation as fair as possible. The speed-up varies from 3.8 to 15.34 times depending on the sample data. The proposed accelerator is faster than the conventional software-based method for all short-read samples.

Table 9 shows the processing time comparison for paired-end short reads. There are two data files with the same number of short-reads. We processed them in parallel on two channels independent of each other. The total processing time is measured after all the short-reads in both files are processed. According to the results in Table 9, the speed-up varies from 6.91 to 21.84. The proposed accelerator is faster than the conventional software-based method for all short-read samples.

Table 10 shows the number of hits and the matching ratio of the proposed accelerator, compared to the BWA software. The matching ratio is defined by Eq.(3)

$$\text{Matching ratio} = \frac{\text{number of hits}}{\text{number of short} - \text{reads}} \times 100\%. \quad (3)$$

According to the results, the matching ratio is very similar to the BWA software. Note that, the FPGA accelerator executes the exact method given in Algorithm 1. Although the

TABLE 8
Processing Speed for Single-end Reads

Example	Number of short-reads	Short-read length	Processing time (minutes)		Speed-up (times)	Maximum number of mismatches
			BWA software	Our approach		
DRR013803	60,200,625	100	236.2	24.60	9.60	5
DRR000366	44,948,037	36	47.1	12.40	3.80	2
SRR064944	13,946,507	40	17.0	1.11	15.34	3
SRR042411	8,136,133	36	6.6	0.66	10.04	2

TABLE 9
Processing Speed for Paired-End Reads

Example	Number of short-reads	Short-read length (bases)	Processing time (minutes)		Speed-up (times)	Maximum number of mismatches
			BWA software	Our approach		
DRR002191	126,605,856	90	371.5	30.83	12.04	4
SRR609083	59,087,904	100	482.0	22.09	21.84	37
SRR538828	20,125,197	88	68.8	7.51	9.14	13
SRR068810	1,008,910	36	4.0	0.57	6.91	8
SRR065033	277,328,866	101	2,091.8	182.25	11.47	42

TABLE 10
Matching Ratio of the Proposed Approach

Example	Number of Short-read	Short-read length (bases)	BWA software		Our approach	
			Number of hits	Matching ratio (%)	Number of hits	Matching ratio (%)
SRR042411	8,136,133	36	7,924,046	97.39	7,675,806	94.86
SRR064944	13,946,507	40	13,763,219	98.69	13,693,678	98.19
DRR002191-1	126,605,856	90	111,497,861	88.06	111,699,118	89.80

TABLE 11
Workload Distribution in the Proposed Hybrid Method

Data sample	FPGA			CPU			
	Short-read		Matching ratio (%)	Short-read		Matching ratio (%)	Contribution to total hits (%)
	amount	%		amount	%		
SRR042411	7,906,404	97.18	96.48	229,729	2.82	38.97	1.16
SRR064944	13,657,684	97.93	99.44	288,823	2.07	38.96	0.82
DRR002191-1	122,783,538	96.98	92.01	3,822,318	3.02	87.17	2.93

BWA software is based on Algorithm 1, it employs some changes to increase the efficiency. This has caused the slight difference in the matching ratio.

Table 11 shows the workload distribution among FPGA and CPU in the proposed hybrid method. According to the results, FPGA processes more than 96 percent of the short-reads while CPU processes less than 4 percent. The contribution to the total hits by the re-alignment in CPU is less than 3 percent. That means, over 97 percent of the hits are generated by the FPGA accelerator. If the number of input short-reads are as large as several billions, the output of the FPGA alone may enough to construct the genome. In such cases, we can skip using software. If we use FPGA only, we can reduce the power consumption also, since FPGAs consume less power compared to high-end CPUs.

The mapping quality of a short-read refers to the phred-scaled posterior probability that the aligned position of the short-read is incorrect [18], [19]. Therefore, higher the mapping quality is, lower the probability of the aligned position been incorrect. Mapping quality is determined by the alignment algorithm, and different algorithms use different functions to calculate it. In BWA, the mapping quality is determine by considering several factors such as the base quality of a short-read (this is determined by sequencer at the process of extracting a short-read), sensitivity of the algorithm, types and the number of mismatches, multiple aligned positions, (or multi-hits) etc. We have used the same method using in the BWA software to calculate the mapping quality. Therefore, both the BWA software and the proposed accelerator have the same mapping quality.

4.2 Comparison with Other FPGA-Based Accelerators

Recently, many FPGA-based accelerators have been proposed for short-read alignment. Some of the most recent works are [20], [21], [22], [23], [24] and [25]. The method proposed in [20] can be used only for very small genomes such as "E. Coli" [26]. The memory requirement for a human genome is so large that it does not fit to any FPGA board. The work in [21] reports a processing time of 34 seconds using eight FPGAs to align 50 million short-reads of 76 bases long. In this method, 22 GB of the reference human genome data is divided to eight partitions and each partition is stored in a separate FPGA. All eight FPGAs are required to align the short-reads, so that the processing speed per an FPGA is small. Table 12 shows the comparison with [22] that uses Virtex 7 FPGA. The experimental results are obtained using one million short-reads of 36 bases long. Although we achieved over three times of speed-up compared to [22], this comparison may not be very reliable since the input short-read sample of [22] is not given. As shown in Tables 8 and 9, the processing speed varies heavily with the experimental conditions such as the input short-read data, number of mismatches, etc.

The works in [23], [24] have mentioned the experimental conditions and sample data files, so that we compare our

TABLE 12
Comparison with the Work in [21]

Experimental conditions		Processing time (s)		Speed-up
Short-read length	Number of mismatches	Work in [22]	Our approach	
36 bases	0	10	2.7	3.67
	1	12	3.8	3.19
	2	15	4.8	3.10

work with those. The work in [23] uses the first one million short-reads from the data sample "SRR385773-1.fastq". According to the results in Table 13, our proposed method is 6.2 times faster. Although our matching ratio is smaller than that of [23], it is almost the same or even better compared to other software-based approaches such as BFAST, BOWTIE and BWA.

Table 14 shows the comparison with [24]. It uses the first 50 million short-reads from the data sample "SRR385773-1.fastq". The maximum mismatches allowed in [24] is just one. Therefore, we conduct two comparisons, the first one is for zero mismatches (exact matching), and the second one is for one mismatch. According to the results, our approach is 2.29 times faster for exact matching and 2.85 times faster for "one mismatch alignment" compared to [24] that uses the "Xilinx Virtex-6 XC6VLX240T" as the FPGA. However, when a high-end Xilinx Virtex-7 "XC7VX690T" is used as the FPGA, the speed-up in exact matching is larger than that of our approach. However, for "one mismatch alignment" our approach is still better which gives 1.07 speed-up compared to [24]. Unfortunately, the work in [24] used only a single short-read sample in its experiments. An extensive evaluation under different conditions using different data samples is required to have a fair comparison. Moreover, the proposed accelerator can handle multi-mismatches and the processing time does not increase exponentially. Main reason for this is the hybrid processing method discussed in Section 3.5, where we process some of the unaligned short-reads that take unusually large processing time, using the BWA software in CPU.

The accelerator proposed in [25] is a major contribution of short-read alignment using FPGAs. In Table 15, we compare our work with [25] considering the processing speed, hardware resources and the scope of the implementation. In [25], the processing speed is measured in-terms of "bases aligned per second $(baps)$". Equation (4) shows how to calculate $baps$

$$baps = \frac{\text{short} - \text{read size} \times \text{number of short} - \text{reads}}{\text{Processing time (seconds)}}. \quad (4)$$

According to the comparison, the accelerator proposed in [25] has a larger $baps$ value compared to ours when the maximum mismatches is two. We achieved a larger $baps$ value for the exact matching. However, we cannot compare it with [25] since it has not done such an experiment. The $baps$ value of the proposed accelerator drops slightly when the number of mismatches are more than 2. However, we cannot compare this performance with [25] since it is not compatible of such large number of mismatches. Using the $baps$ value to measure the performance may not be fair due to the following reasons. One reason is that, Eq.(4) assumes that a linear relationship exists between the number of bases

TABLE 13
Comparison of the Processing Time and the Matching Ratio

Method	Processing time (s)	Speed-up (times)	Matching ratio (%)
BFAST (1,2)	16,428.0	0.004	95.35
BOWTIE (1,2)	154.0	0.40	92.55
BWA (2)	133.3	0.46	95.36
Work in [23] (1)	61.9	1	99.48
Our approach	10.0	6.2	95.39

(1) *Data are taken from [23].*
(2) *Software based approach.*

(short-read size) and the processing time. More bases could increase the search space and time exponentially, so that a linear processing time increase may not be possible. Another reason is that, the number of mismatches are not considered in Eq.(4). Therefore, it is difficult to compare two methods using $baps$ value alone unless the short-read sample and the number of mismatches are the same.

The major performance bottleneck in the proposed accelerator is the memory bandwidth. If we increase the memory bandwidth, we could be able to achieve a larger processing speed. To use such larger bandwidths, we need wider data paths also. Therefore, we have to increase both the memory bandwidth and the logic area to increase the processing speed. The number of logic elements in the high-end FPGA (XC7VX690T) used in [24] is three times larger than the FPGA used by us. The number of logic elements in the FPGA used in [25] is also two times larger than the one used by us, and the memory of the FPGA board in [25] is six times larger than ours. However, neither [24] nor [25] does mention the memory bandwidth. Therefore, even [25] gives a slightly better speed-up for the exact matching and [25] gives a better $baps$ value, it is very difficult to say those accelerators are better than ours without evaluating under the same conditions.

The maximum number of mismatches of the proposed method is 252 compared to 0 to 2 in most other methods such as [25]. Since, many short-read samples have few tens of mismatches, our method is more practical and have a wider coverage. Moreover, the processing time does not increase exponentially with the number of mismatches, which is a great advantage compared to other works. Selecting the tolerant number of mismatches is a difficult problem. Considering a large number of mismatches may increase the matching ratio (number of hits), but also may decrease the mapping quality. (Note that, the mapping quality depends on several factors. For some short-reads, increasing the number of

TABLE 14
Comparison with the Work in [24]

	FPGA specification			Processing time (s)			
	FPGA	Logic elements	Block RAMs (kb)	Number of mismatches			
				0	1	2	3
Work in [24]	Virtex-7	693,120	52,920	97	158	-	-
	Virtex-6	241,152	14,976	296	419	-	-
Ours	Stratix V	234,720	51,200	129	147	160	173

TABLE 15
Comparison with the Work in [25]

		Work in [25]	Our approach
Accelerator board		MAX3	DE5
FPGA		Xilinx Virtex-6 SX475T	Altera Stratix V 5SGXEA7N2F45C2
Processing speed	Number of mismatches used in experiments	2	2 to 42
	Size of the short-reads used in experiments	75 bases	36 $\sim$ 101 bases
	Speed (Maximum mismatches $>$ 2)	not possible	1.04 $\sim$ 8.37 million (*baps*)
	Speed (Maximum mismatches = 2)	13.5 million (*baps*)	1.12 $\sim$ 8.75 million (*baps*)
	Speed (Maximum mismatches = 0)	not given	5.21 $\sim$ 17.45 million (*baps*)
Hardware resources	Logic elements	476,160	234,720
	Block RAM	38,304 kb	51,200 kb
	Memory on board	24 GB	4 GB
	Theoretical memory bandwidth	not given	12.8 GBps $\times$ 2 channels
Scope of the implementation	Genome type used in experiments	human	human
	Maximum number of mismatches	up to 2	up to 252
	Maximum short-read length	100 bases	252 bases

mismatches may also increases the mapping quality as well). In order to construct the whole genome, we need more hits. However, if the mapping qualities of the hits are low, the constructed genome may not be accurate. Therefore, it is necessary to take a balance between the quality and the quantity. Usually, many aligners including BWA software have the option of changing the number of mismatches. In the default settings of the BWA software, it allows some percentage of base errors. Usually, setting the proper amount of mismatches is done by the user and we allowed that option in the proposed accelerator. If the users have not changed it, the default settings, which are the same as in the BWA software, will be applied.

5 Conclusion

We have proposed a hardware accelerator architecture for short-read alignment. The processing speed of the proposed accelerator is larger than that of the software implementation for all experimented short-read samples. The reason behind this is the efficient memory access and parallel processing using FPGA hardware. The proposed accelerator has a similar or better processing speed compared to many other FPGA-based acceleration methods. The biggest advantage of the proposed method compared to other FPGA-based methods is the degree of freedom for short-read alignment. Most other works are limited to exact matching or less than two mismatches with less than 100 bases. Since increasing the number of mismatches could increase the processing time exponentially, many other methods do not consider such large number of mismatches. Our method can process up to 252-base long short-reads with 252 mismatches. The short-read length could be increased further by allowing a small processing time increase. Moreover, the processing time does not increase exponentially with the number of mismatches.

It is possible to increase the processing speed further by choosing the latest FPGAs such as Altera Stratix 10 with more LUTs and memory bandwidth. Multiple FPGAs connected by fiber optic cables can be used to increase the processing speed massively. Since we use FPGA, we can update the accelerator to be compatible with the future versions of BWA by hardware reconfiguration. Therefore, the proposed FPGA accelerator has a great potential for a large processing speed increase.

Acknowledgments

This work is partially supported by MEXT KAKENHI Grant Numbers 15K15958 and 24300013.

References

[1] L. Liu, Y. Li, S. Li, N. Hu, Y. He, R. Pong, D. Lin, L. Lu, and M. Law, "Comparison of next-generation sequencing systems," *J. Biomed. Biotechnol.*, vol. 2012, pp. 1–11, 2012.
[2] H. Li. (2008). Maq: Mapping and assembly with qualities [Online]. Available: http://maq.sourceforge.net/
[3] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome Biol.*, vol. 10, no. 3, p. R25, 2009.
[4] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-wheeler transform," *Bioinfomatics*, vol. 25, no. 14, pp. 1754–1760, 2009.
[5] N. Homer, B. Merriman, and S. F. Nelson, "BFAST: An alignment tool for large scale genome resequencing," *PLoS ONE*, vol. 4, no. 11, p. e7767, 2009.
[6] S. Altschul, W. Gish, W. Miller, E Myers, and D Lipman, "Basic local alignment search tool," *J. Molecular Biol.*, vol. 215, no. 3, pp. 403–410, 1990.
[7] R. Li, C. Yu, Y. Li, T. Lam, S. Yiu, K. Kristiansen, and J. Wang, "SOAP2: An improved ultrafast tool for short read alignment," *Bioinformatics*, vol. 25, no. 15, p. 1966, 2009.
[8] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Digital Equipment Corporation, Palo Alto, CA, USA, Tech. Rep. 124, 1994.
[9] W. Carter, K. Duong, R. H. Freeman, H. Hsieh, J. Y. Ja, J. E. Mahoney, and L. T. Ngo, "A user programmable reconfigurable gate array," in *Proc. CICC*, 1986, pp. 233–235.
[10] H. M. Waidyasooriya, M. Hariyama, and M. Kameyama, "Implementation of a custom hardware-accelerator for short-read mapping using Burrows-Wheeler alignment," in *Proc. 35th Annu. Conf. Eng. Med. Biol. Soc.*, 2013, pp. 651–654.
[11] H. M. Waidyasooriya, M. Hariyama, and M. Kameyama, "FPGAAccelerator for DNA Sequence alignment based on an efficient data-dependent memory access scheme," in *Proc. 5th Int. Symp. Highly-Efficient Accelerators Reconfigurable Technol.*, 2014, pp. 127–130.
[12] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proc. 41st Symp. Found. Comput. Sci.*, 2009, pp. 390–398.
[13] H. S. Warren, *Hacker's Delight*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2012, ch. 5.
[14] [Online]. Available: https://trace.ddbj.nig.ac.jp/DRASearch/, 2015.

[15] [Online]. Available: https://www.altera.com/support/training/university/de5.html, 2012.
[16] [Online]. Available: http://bio-bwa.sourceforge.net/, 2013.
[17] [Online]. Available: http://hgdownload.cse.ucsc.edu/goldenpath/hg18/chromosomes/, 2006.
[18] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin and 1000 Genome Project Data Processing Subgroup, "The Sequence alignment/map (SAM) format and SAMtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.
[19] "Sequence Alignment/Map Format Specification," The SAM/BAM Format Specification Working Group, 2015.
[20] Y. Chen, B. Schmidt, and D. L. Maskell, "A hybrid short read mapping accelerator," *BMC Bioinformat.*, vol. 14, p. 67, 2013.
[21] C. B. Olson1, M. Kim, C. Clauson1, B. Kogon1, C. Ebeling, S. Hauck1, and W. L. Ruzzo, "Hardware acceleration of short read mapping," in *Proc. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2012, pp. 161–168.
[22] Y. Xin, B. Liu, B. Min, W. X. Y. Li, R .C. C. Cheung, A. S. Fong, and T. F. Chan, "Parallel architecture for DNA sequence inexact matching with Burrows-Wheeler Transform," *Microelectron. J.*, vol. 44, pp. 670–682, 2013.
[23] Y. Sogabe and T. Maruyama, "An Acceleration Method of Short Read mapping using FPGA," in *Proc. Int. Conf. Field-Programmable Technol.*, 2013, pp. 350–353.
[24] Y. Sogabe and T. Maruyama, "FPGA acceleration of short read mapping based on sort and parallel comparison," in *Proc. Int. Conf. Field Programmable Logic Appl.*, 2014, pp. 1–4.
[25] J. Arram, K.H. Tsoi, Wayne Luk, and P. Jiang, "Hardware acceleration of genetic sequence alignment," in *Proc. 9th Int. Conf. Reconfigurable Comput.: Architectures, Tools Appl.*, 2013, pp. 13–24.
[26] F. R. Blattner, G. Plunkett III, C. A. Bloch, N. T. Perna, V. Burland, M. Riley, J. Collado-Vides, J. D. Glasner, C. K. Rode, G. F. Mayhew, J. Gregor, N. W. Davis, H. A. Kirkpatrick, M. A. Goeden, D. J. Rose, B. Mau, and Y. Shao, "The complete genome sequence of Escherichia coli K-12," *Science*, vol. 277, no. 5331, pp. 1453–1462, 1997.

Hasitha Muthumala Waidyasooriya received the BE degree in information engineering, and the MS and PhD degrees in information sciences from Tohoku University, Japan, in 2006, 2008, and 2010, respectively. He is currently an assistant professor with the Graduate School of Information Sciences, Tohoku University. His research interests include reconfigurable computing, processor architectures for big-data processing, and high-level design methodology for VLSIs. He is a member of the IEEE.

Masanori Hariyama received the BE degree in electronic engineering, and the MS and PhD degrees in information sciences from Tohoku University, Sendai, Japan, in 1992, 1994, and 1997, respectively. He is currently an associate professor with the Graduate School of Information Sciences, Tohoku University. His research interests include real-world applications such as robotics and medical applications, big data applications such as bio informatics, high-performance computing, VLSI computing for real-world application, high-level design methodology for VLSIs, and reconfigurable computing. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.