IEICE TRANS. INF. & SYST., VOL.E100–D, NO.5 MAY 2017

PAPER

A Fast and Accurate FPGA System for Short Read Mapping Based on Parallel Comparison on Hash Table

Yoko SOGABE[†a)], ***Nonmember and*** **Tsutomu MARUYAMA**[†b)], ***Member***

SUMMARY The purpose of DNA sequencing is to determine the order of nucleotides within a DNA molecule of target. The target DNA molecules are fragmented into short reads, which are short fixed-length subsequences composed of 'A', 'C', 'G' 'T', by next generation sequencing (NGS) machine. To reconstruct the target DNA from the short reads using a reference genome, which is a representative example of a species that was constructed in advance, it is necessary to determine their locations in the target DNA from where they have been extracted by aligning them onto the reference genome. This process is called short read mapping, and it is important to improve the performance of the short read mapping to realize fast DNA sequencing. We propose three types of FPGA acceleration methods based on hash table; (1) sorting and parallel comparison, (2) matching that allows one mutation to reduce the number of the candidates, (3) optimized hash function using variable masks. The first one reduces the number of accesses to off-chip memory to avoid the bottleneck by access latency. The second one enables to reduce the number of the candidates without degrading mapping sensitivity by allowing one mutation in the comparison. The last one reduces hash collisions using a table that was calculated from the reference genome in advance. We implemented the three methods on Xilinx Virtex-7 and evaluated them to show their effectiveness of them. In our experiments, our system achieves 20 fold of processing speed compared with BWA, which is one of the most popular mapping tools. Furthermore, we shows that the our system outperforms one of the fastest FPGA short read mapping systems.

key words: *field programmable gate array, genome sequence alignment, accelerated implementation, bioinformatics*

1. Introduction

DNA sequencing is a process to determine the order of nucleotides (A, C, G, T) in a target DNA, and this technology can be applied widely, for example, to ecology, evolutionary studies, agriculture, drug discovery and personalized medicine. In DNA sequencing, the target DNA are broken up into short DNA fragments. The fragments are read by a DNA sequencer, and the obtained fixed-length sequences of nucleotides are called short read. Not full length but the fixed length portion of the DNA fragment is read. In the paired-end sequencing, to achieve higher quality sequencing, the DNA fragment is read from both ends of the fixed length. Then the short reads are mapped onto a given reference genome, which is a representative example of a species that was constructed in advance. Using their locations obtained by the mapping, the target DNA can be reconstructed.

Next generation sequencer (NGS) have achieved high throughput more than 100G base pairs per day with one machine, and DNA sequencing of the human genome is becoming popular A base pair (bp) is a pair of complementary nucleotides ("A-T", "C-G") linked by hydrogen bonds. However, this mapping process requires long computing time; in general, it takes a few days. Furthermore, there are genetic variations between the target genome and the reference genome, and it makes the mapping more difficult. Hence, the short read mapping is becoming the bottle-neck of DNA sequencing using NGS.

Several software tools for the mapping such as BOWTIE [1], BOWTIE2 [2], BWA [3], and BFAST [4] have been developed, but their throughput are slower than the NGS. To accelerate the performance, several FPGA systems have also been proposed. In [5], an FPGA aligner based on the BFAST was implemented. The final system with 8 Virtex-6 FPGAs achieved two and one orders of magnitude speedup against BFAST and BOWTIE respectively. In [6], an FPGA system for a variant of the FM-index algorithm was implemented on Virtex-6 FPGA, and it showed similar performance with [5]. In [7], an FPGA system based on BWA was implemented. To reduce the index size, an encoded occurrence array was proposed. In [8], an FPGA-based acceleration of a drop-in replacement for BOWTIE was proposed. This system achieved 12 times of speedup compared to BOWTIE running 8 threads, however, it dose not support gapped alignment.

In this paper, we propose an FPGA acceleration method to improve the hash-index method used in BFAST. This approach have achieved higher processing speed while maintaining the high matching rate. Our approach consists of three parts:

(1) sorting seeds, which are a part of short reads, by using a hash function to collect P (P is the maximum number of seeds which are compared in parallel) seeds with the same hash value and comparing them in parallel,

(2) allowing one nucleotide substitution, insertion and deletion in the comparison to improve the mapping sensitivity without increasing the number of compared candidates, and

(3) optimizing hash function using variable masks to reduce fruitless comparison.

These three approaches have been published in [9]–[11]. In [9], [10], the above approach (1) and (2) have been proposed.

Manuscript received June 13, 2016.
Manuscript revised December 13, 2016.
Manuscript publicized January 30, 2017.

[†]The authors are with Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba-shi, 305–8573 Japan.
a) E-mail: sogabe@darwin.esys.tsukuba.ac.jp
b) E-mail: maruyama@darwin.esys.tsukuba.ac.jp

DOI: 10.1587/transinf.2016EDP7262

In this paper, we discuss the current implementation and conduct new experiments to evaluate more parameters (its result is shown in Fig. 8). These experiments show that our system improves the mapping rate compared with BWA if the short reads include many mutations.

Furthermore, to get a more detailed evaluation, we evaluate its performance using 45M paired-end short reads with 100 base-pairs, which were generated from whole genome sequence of a human by Illumina HiSeq 2000.

2. Short Read Mapping

Figure 1 shows the outline of the DNA sequencing using the reference genome. The goal of the DNA sequencing is to identify the correct full sequence of nucleotides of the target genome. First, the target genomes are randomly fragmented into short reads by NGS. Then, by finding the most similar parts in the reference genome, the short reads are mapped onto the reference genome. In this mapping process, a part of short reads do not exactly match with the reference genome because of the genetic variations and the read errors by NGS. To distinguish them, several same sequences are randomly fragmented in order to map different short reads onto the same location of the reference genome. By this redundant mapping, the genetic variation can be distinguished from the read error, because read error occurs randomly. This mapping process requires long computation time, and is becoming the bottle-neck of DNA sequencing by NGS. For improving the mapping performance, many algorithms have been proposed. They can be categorized into two groups: FM-index [12] and hash-index. FM-index is a compressed full-text index using the the Burrows-Wheeler transform (BWT) [13]. BOWTIE [1], BOWTIE2 [2] and BWA [3] belong to this category. With this approach, the total data size required for human genomes is kept relatively small (about 4GB), and this algorithm is one of the fastest on desktop computers. However, the running time of this class of algorithms is exponential with respect to the allowed number of genetic variations (in general, no variation is allowed to achieve higher processing speed), and it is not easy to accelerate this algorithm using hardware because the tables are repeatedly referred to find the final locations.

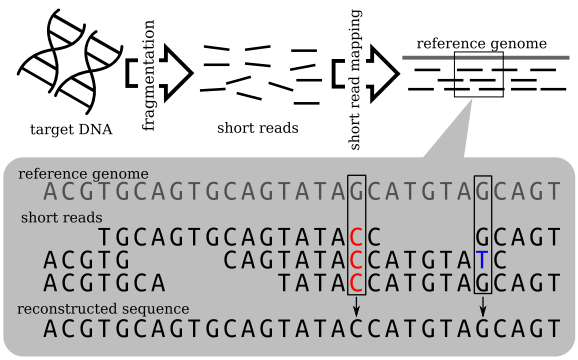

Fig. 1 DNA sequencing using the reference genome

2.1 Hash-Index Method

The hash-index method is an approach using subsequence of short reads, which are called seeds. BFAST [4] belongs to this category. In the first stage, it finds candidate alignment locations (CALs) by exactly matching seeds with a part of the reference genome. The CALs are scored by Smith-Waterman algorithm, and finally, the best alignment location is determined. The hash table is constructed in advance from the reference genome. In this mapping process, to find their locations in the reference genome, the hash table is looked up using seeds extracted from short reads. The hash table consists of the index table and the CAL table.

Figure 2 shows an example of hash table. In the following, we explain the details of the search using this example. In Fig. 2, the reference genome is "ACGTAACGTAGC", and the length of seeds is 4 letters. Each nucleotide (A, C, G, T) is encoded to 2 bits (`00,01,10,11`). In this case, the hash values of seeds are simply the first 2 letters (4 bits). For example, the hash value of "CGTA" is CG(`0110` ($= 6_{10}$)). There exist 9 seeds ($12 - 4 + 1$) in the reference genome. All seeds in the reference genome are extracted, and they are registered in the CAL table with their locations on the reference genome. The index table is accessed by the hash values of seeds and gives the regions (starting position and size) in the CAL table. Each region in the CAL table is called hash bucket. A hash bucket stores seeds with the same hash value in the reference genome. Because the first two letters are obvious, the CAL table stores only the remaining two letters of seeds, which is called key bits. For example, {CG,4} in the first bucket in the CAL table means "AACG" is located at 4 in the reference genome. Considering the range of the hash values, the length of the index table (which is called index size) is 16.

Suppose that a short read "CGTAATG" is given. The difference between short reads and the reference genome is very small in general, and we can expect that some seeds in a short read do not include the variations, and exactly match with the reference genome. There exist 4 seeds in the short read. For each seed, its location on the reference genome is looked up using the tables. For example, to find the location of the seed "GTAA", first, the index table is accessed using its index "GT". Then, "5 / 2" is obtained. Using 5 and 2, {"AA",2} and {"AG",7} are read from the CAL table. "AA" and "AG" are compared with "AA" (the key of "GTAA"), and 2 is obtained as the candidate location. In the

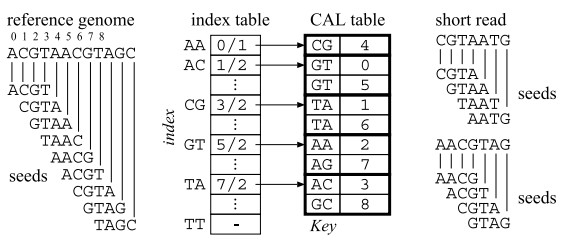

Fig. 2 The index and CAL tables

same way, the candidate locations 1 and 6 are obtained from "CGTA". From "TAAT" and "AATG", no candidate location is obtained because of the failure of their key comparisons. Then, the whole short read is compared with the reference genome starting from 1 and 6, and 1 is chosen as the final location. According to [4], in case of 22 letter seeds, more than 80% seeds have only one candidate location, while the number of the seeds that have 10 to more than 100000 candidate locations is more than 15%. The seeds which appear more than 8 times on the reference genome are called frequent seeds and removed from the CAL table because they unnecessarily increases the number of candidate locations. In BFAST, generally, the length of seed is 22, and the index size becomes 2^{28}.

When we consider to implement this method on a hardware system, the parallelism in key comparisons and the number of random accesses to off-chip DRAM banks are two important factors that dominate the system performance.

3. Our Approach

The target of our system is the human whole genome, and the length of short reads and seeds are 100 and 22 base pairs. The short reads and the reference sequence include ambiguous characters (non-A/C/G/T characters, typically 'N' which means unknown or any). In [3], the ambiguous characters are randomly converted to 'A', 'C', 'G', 'T'. We follow this method to simplify the computation by encoding the nucleotides in 2 bits. This method may cause false match, however, the chance that this may happen is very small as reported in [3].

Figure 3 shows the overview of our system and its data flow. 'SR' stands for short read. The hash table and the reference sequence are stored in off chip memory of FPGA in advance. The hash table is generated on the host-PC in two steps; (1) calculating the size of each hash bucket, and (2) creating the hash table the size of which is the sum of each hash bucket size. It takes several hours to create the hash table, however, this time is not important because it

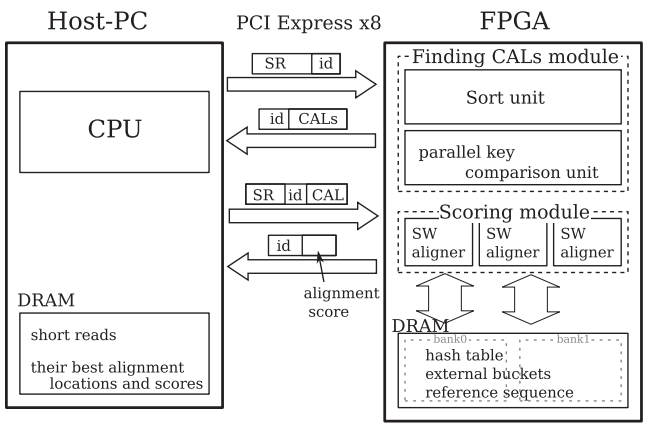

Fig. 3 The overview of our system and its data flow

can be used for sequencing all other human genomes. First, the host-PC sends short reads and their IDs (serial numbers) to FPGA. The "Finding CALs module" calculates CALs of the short reads and consists of the sort unit and the parallel comparison unit. The given short reads are sent to the sort unit. In the this unit, all seeds in the short reads are extracted, and then sorted by their hash values. When P seeds with the same hash value are obtained, the sort unit is stopped and the P seeds are compared in the parallel key comparison unit. The parallel key comparison unit sends back the IDs and their CALs of matched seeds to the host-PC. In the host-PC, the best alignment location and its score for each short read are managed and the host-PC sends the short read and its CAL to FPGA to calculate the score of the CAL, if the CAL is not the best location of the short read (unless the score is already known). Scoring module consists of several Smith-Waterman (SW) aligners that operate concurrently. The paired end reads is a pair of reads with almost fixed distance (this distance is called insert size), and used to improve the sensitivity of mapping. In our system, both paired-end reads is aligned on the reference on the condition that the max insert size is 800. This operation is performed by Smith-Waterman algorithm between paired-end reads (200 bps) and the candidate locations on the reference sequence (1000 bps).

Our research focuses on how to accelerate the finding CAL stage, because it requires most of the computation time and Smith-Waterman's implementation on FPGA have been already proposed in [14] and others. In this section, we describe our methods, which consists of three parts as follows.

1. Parallel comparison:
   All seeds in the given short reads are extracted, and they are sorted by their hash values using bucket sort. When P seeds with the same hash value are obtained, their candidate locations in the corresponding hash bucket are read from the CAL table in DRAM bank, and compared with the P seeds in parallel.
2. Flexible matching that makes it possible to reduce the number of candidate locations registered in the hash table:
   In the key comparison, one substitution, insertion and deletion are allowed to improve the robustness against variation between short reads and the reference genome. We call this matching 'flexible match'. By using flexible match, we can reduce the number of the key comparison by removing a number of seeds from the CAL table without degrading mapping accuracy.
3. Optimized hash function using variable masks:
   The ununiformity of the reference genome causes the great amount of hash collisions in the original hash function. To reduce the hash collisions, we propose a new hash function that uses variable masks.

3.1 Parallel Comparison

In FPGA implementation, the hash table must be stored on

off-chip DRAM memory because the size of the hash table of human genome is more than 4GB. When the hash table is accessed one by one, DRAM access delay becomes the bottleneck of the system. In order to use computing resources in FPGA efficiently, it is very important to get rid of DRAM access delay. In our approach, to reduce the number of DRAM accesses, seeds with the same hash values are sorted by bucket sort, and the hash table is accessed when P seeds with the same hash value become ready. The hash table is accessed once for the P seeds, and the P seeds are compared with the corresponding keys in parallel. This parallel comparison aims to reduce the computation time of the key comparison and the idle time caused by DRAM access delay by reducing the number of DRAM accesses to $1/P$.

3.1.1 Sorting

Two level buckets, internal and external, are used in our system. Figure 4 shows the bucket sort unit. First, the received short read is set to the shit register. The short read is shifted to left by 2b controlled by the counter, and the left most 44b is extracted as a seed. From one short read of length l, $l-22+1$ seeds are generated. The value of counter means the seed's position on the short read. Each entry of a bucket is a key field of the seed (the last 32b of the seed), its short read ID, and its position on the short read. The position on the short read and the ID are expressed by 7 and 25 bits. Therefore, the size of each entry is 8B. Each external bucket in off-chip DRAM bank stores P seeds of the same hash value. The total size of external buckets is $P \times w \times 8$B (w is the range of hash values). Each internal bucket on on-chip memory in FPGA stores 7 seeds of the same hash value and works as cache memory. The total size of internal buckets is $7 \times w \times 8$B. When the 8th seed is given to an internal bucket, the 8 seeds (7 seeds in the internal bucket and the 8th seed) are moved to the external bucket by burst-write. Internal buckets are used to reduce the accesses to the external buckets in the off-chip DRAM bank, and to achieve higher data transfer rate by burst-write. The sort module processes one seed per one clock cycle. When an external bucket becomes full, the sort module is stopped, and the key comparison unit reads P seeds from the bucket.

Although larger w is required to reduce the hash collisions, the realizable size of internal buckets is limited. In BFAST, w is 2^{28}, but it is reduced to 2^{14} to 2^{18} in our system because of the limitation of on-chip memory resources on FPGA. Although it raises the amount of the hash collisions, the bottleneck of DRAM access is avoided, and furthermore, more number of key comparisons can be executed in parallel.

3.1.2 Parallel Key Comparison

Figure 5 shows a block diagram of the parallel key comparison unit. When an external bucket becomes full, the P seeds in it are read back from the bucket, and given to this unit. Each seed in the bucket consists of the key, the short read ID and the position in the short read. At the same time, the keys and CALs in the corresponding hash bucket are read from the hash table, and also given to this unit. In this unit, the keys in P seeds are compared with all keys in the hash bucket in parallel. The comparison on this unit consists of

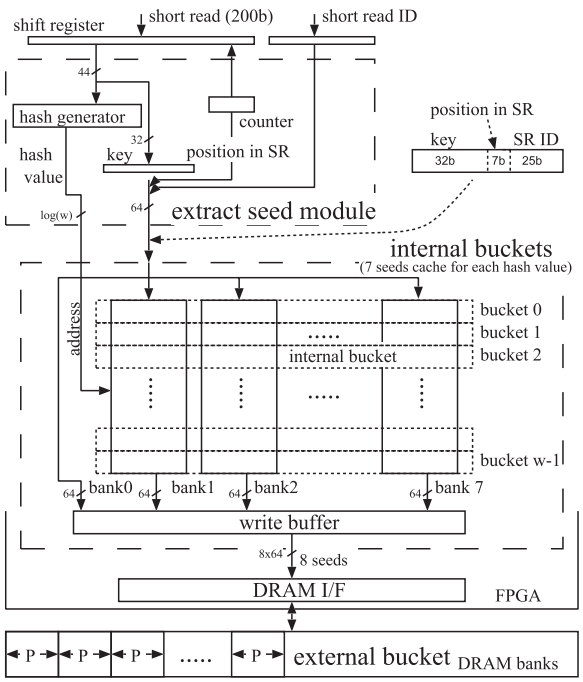

Fig. 4 Sort unit by bucket sort

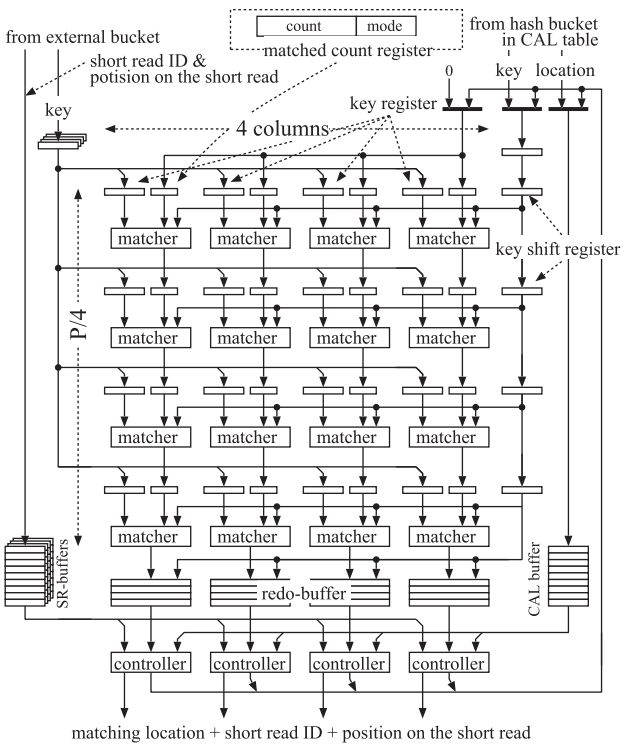

Fig. 5 A diagram of the parallel key comparison unit

Table 1 The modes and operations

mode	operations executed when keys match
first stage	increase count's value
second stage (not reach)	if count's value is zero, change count's value and mode to the position of the key and 'lock' respectively, otherwise decrease count's value
lock (second stage)	none

two phases, and the mode fields in 'matched count register' manage these two phases.

With burst-read, 4 seeds in the external bucket can be given to this unit every clock cycle. The keys from the CAL table are given to the top of the array of the key shift registers one by one. They are shifted down every clock cycle, and are compared with the four seeds on each line in parallel. With each key, four 'matched count', which were initialized to zero, are shifted down. These counts are incremented independently when the key matches the key shift register on the same column. When each key reaches the bottom of the array (it takes $P/4$ clock cycles), it was compared with all P seeds, and the four 'matched counts' are stored in the four redo-buffers respectively (if the count is zero, it is discarded). The short read ID and the position on the short read from the external bucket and the location from the CAL table are directly sent to the SR-buffer and the CAL buffer because they are not necessary for the comparisons. The SR-buffer stores P pairs of the ID and the position on the short read and is accessed using the position on the array at the end of the second phase. CAL buffer is a FIFO with $P/4$ CALs, and the input CAL appears at the output after $P/4$ clock cycles.

After the last entry of the hash bucket is given to the array, the second stage is started. The purpose of this stage is to find the pairs of a matched short read ID and a CAL using the result of the first stage. Each entry in the redo-buffers is copied m times if its matched count is m, and the m copies are re-entered into the array one by one. For this reprocessing, j $(0 \le j < m)$ is attached to each copy as its initial count. When the key matching is successful, one of the following action is taken. If the count is zero, the line number of array is copied into the count field, and the mode is changed so that the count is not changed any more. Otherwise the count is decremented by one. When the copy reaches the bottom, the count field shows with which seed the copy matches. By using the count field and the SR-buffer, a pair of the short read ID and its CAL are obtained and sent back to the host-PC.

In our implementation, to realize the procedure above, the mode field of the matched count register is used like a state machine. Table 1 show a list of mode and what to do when the keys are matched. Figure 6 shows the data flow on one column of the parallel key comparison unit (actually four columns). In the first phase, four columns work in parallel. In the second phase, only one column work exclusively because the key shift registers is only one column.

In this unit, it is necessary to read 4 seeds per clock cycle from the external bucket and one seed per clock cycle from the hash table. The memory bandwidth of the system

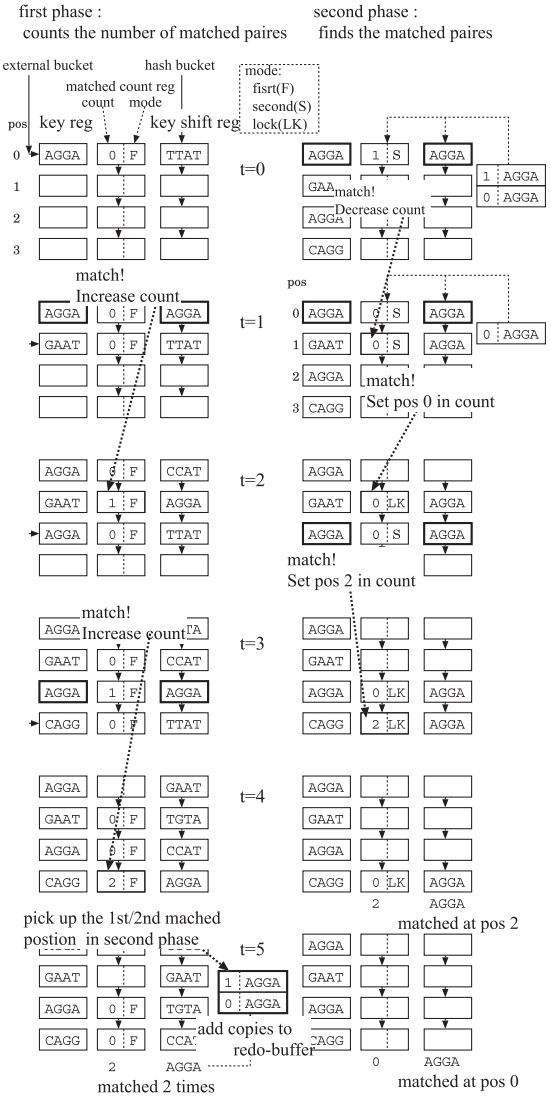

Fig. 6 The data flow on the parallel key comparison unit (one column)

is fast enough for this requirements.

3.2 Flexible Match to Reduce the Hash Table Size

3.2.1 Flexible Match

Figure 7 shows a block diagram of the matcher. In Fig. 7, key0 and key1 are compared. 'Exact matching' module compares its two inputs nucleotide (2b) by nucleotide (2b). '#zero==1' module counts the number of zeros in its input, and outputs 1 if the number of zeros is 1. 'leading gen' module extracts the leading '1's from the most significant bit (e.g., `1110XXXX` ⇒ `11100000`, `0XXXXXXX` ⇒ `00000000`). By taking OR of the output of 'leading gen' and 'exact matching' with two bit left/right shift of its input, we can detect one nucleotide insertion and deletion on arbitrary position on key0. 'valid bit' is attached to each key in the CAL table to suppress the excessive match by the flexible matching. The 'valid bit' of the keys in the CAL table

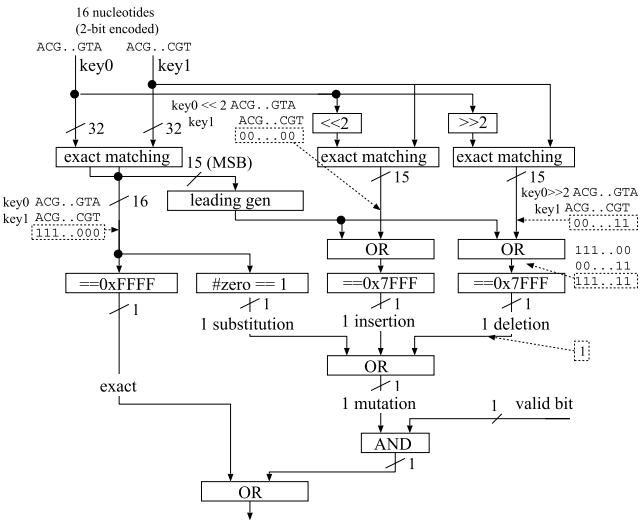

Fig. 7 A block diagram of matcher (the example that key1 has one deletion)

that match frequently are turned off so that the number of successful matches does not become larger than 8 for any given key. About 80% of valid bits in the CAL table is on.

3.2.2 Reducing the Size of the Hash Table

The sort and parallel comparison enable to further improve the performance by reducing the size of hash bucket. In BFAST, all seeds in the reference are not registered in the hash table. The frequent seeds are removed from the hash table, and only 80% of them are used. If one seed gives the true location, it is enough to know the true location of the short read. 80% is enough for the mapping, because each short read has 79 seeds $(100 - 22 + 1)$ in it.

In our approach, to reduce the number of key comparisons to $1/s$ (s is an arbitrary integer), only the seeds at $s \times i$th position (i is also an arbitrary integer) on the reference genome are registered in the hash table. By using the seeds that start from sth, $2s$th, $3s$th and so on the reference genome, we can reduce the number of key comparisons to $1/s$. If the seed at $s \times i$ is a frequent seed, the nearest non-frequent one is registered instead of it. With this method, the number of CALs can be reduced to almost $1/s$ and it enables to increase the performance by about s fold.

This methods may degrade the mapping rate, but flexible match enables to achieve the high mapping rate as with BFAST. To evaluate the rate of short reads which are mapped correctly, we have created a single-end simulated data set with their true locations from the reference genome (GRCh37) because the real data sets do not have the true locations. The simulated data consists of 100k short reads with 100 base pair and mutations are added to them artificially to clarify the relation between the amount of mutations and the mapping rate. Figure 8 shows a graph of the mapping rate of BFAST, BWA, which is one of the most accurate software tools, and our method when we change s.

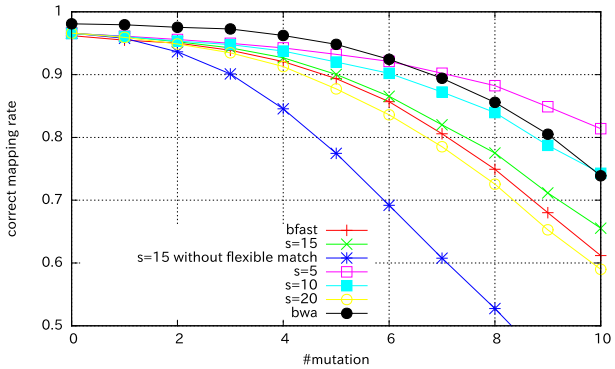

Fig. 8 Mapping rate by exact and flexible matching when s is changed

The horizontal axis shows the mutation rate (%) per base-pair, and the vertical line shows the mapping rate. The distribution of the substitution, insertion and deletion on mutations is 0.75, 0.125, 0.125. As shown in this graph, the mapping rate decreases by reducing the number of seeds to $1/s$, but by using flexible match, we can achieve higher mapping rate than BFAST on each mutation rate when $s <= 15$. Furthermore, when $s = 5$, our mapping rate outperforms BWA when the mutation rate is high. The larger s enables more acceleration. In our implementation, we choose $s = 15$ because this is the maximum of s that shows better mapping rate than BFAST.

3.3 Optimization of the Hash Function

In this section, we improve the hash function from the seed of 22 letters (44b) to the hash value ($\log_2(w)$b). First, we formulate the execution time of the key comparisons;

R: a set of the seeds in the reference genome

D: a set of the seeds in the CAL table

SR: a set of the seeds extracted from the given short reads

Here, let X_i be a subset of X with the same hash value i ($X_i : \{a \in X_i | \ hash(a) = i\}$), and let $|X|$ be the number of elements in X. The execution time of key comparisons (T_c) is given by

$$T_c = C_c \times |SR|$$

where C_c is the average number of the key comparisons for each seed. C_c is given by

$$C_c = \sum_{i \in w} p_i |D_i|$$

where p_i is the probability that a seed is mapped into the hash bucket i. This equation means that when a seed is mapped to i, it is necessary to compare it with $|D_i|$ keys in D_i. Here, p_i can be approximated as

$$p_i = \frac{|SR_i|}{|SR|} \simeq \frac{|Ri|}{|R|}$$

because the distribution of hash values of the short reads is very similar to that of the reference genome. This is caused by the fact that the difference between the short reads and

Table 2 C_{min} and C_r on the original hash function

w	2^{14}	2^{16}	2^{18}
C_r	701728.7	228917.5	88532.7
C_{min}	349282.6	87320.6	21830.2
C_r/C_{min}	2.0	2.6	4.1

the reference is very small. Therefore, C_c can be rewritten as

$$C_c \simeq \frac{\sum_{i \in w} |R_i||D_i|}{|R|}.$$

$|D_i| \propto |R_i|$ because the distribution of D and R is very similar. Thus, the average number of the key comparisons in the hash table with R (which is called C_r) is given by

$$C_r = \frac{\sum_{i \in w} |R_i|^2}{|R|} \simeq K \times C_c,$$

where K is a proportionality coefficient. Hence, to minimize C_c, it is necessary to minimize C_r by equalizing $|R_i|$. We can minimize C_r by improving the hash function because it defines R_i. In addition, when $|R_i| = |R|/w$ (all $|R_i|$ are perfectly equalized), the value of C_r can be minimized (C_{min}). Table 2 shows a comparison of C_r and C_{min} when we change w from 2^{14} to 2^{18}. The bottom line (C_r/C_{min}) indicates that the maximum acceleration rates by optimizing the hash function are more than 2.

3.3.1 Hash Function Using Mask Table

We propose a new hash function using a mask table, to equalize $|R_i|$. The basic idea of our method is to divide larger R_i and merge smaller R_js using the mask table.

First, we define three fields of each seed: key, prefix and extended bit. The key bit is the last 32b of a seed, The prefix bit is the first l_p bit and the extended bit is the next $l_e$bit to the prefix. We define a mask operation here. The mask operation extracts bits from a given bit sequence using its mask bit. We express that x is masked by m as "mask(x,m)". For example, when a data $x = x_4x_3x_2x_1x_0$ and its mask $m = 10110$ are given, the mask function $mask(x, m)$ generates $x_4x_2x_1$. The mask table consists of a pair of an offset and mask bit, and it is accessed by prefix bits. The mask bit and the offset work to divide and to merge hash buckets. The hash value is calculated from prefix and extended fields of a seed using the mask table. The hash function is defined as

$$\begin{aligned} &hash(prefix\_bit, extended\_bit) = \\ &\quad mask\_table[prefix\_bit].offset + \\ &\quad mask(extended\_bit, mask\_table[prefix\_bit].mask) \end{aligned}$$

As shown this equation, to calculate a hash value, first, the offset and mask bit are read from the mask table using the prefix of the seed. The extended bit of the seed is masked by the mask bit, and the numerical value of the result is added to the offset to obtain the hash value.

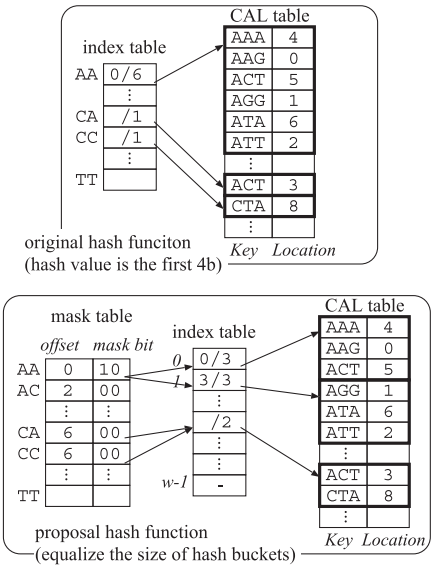

Fig. 9 Comparison between previous and proposal hash function

Figure 9 shows an example of the proposal hash function and original one, which is used in BFAST. In this figure, the length of seed, key, prefix, extended bit are 8b (4-letters), 6b (3-letters), 4b (1-letters) and 2b (1-letters). For example, the first entry in the CAL table means that "AAAA" appears at the location 4 on the reference. In the original method, the hash value is just the prefix bits. In this example, the seeds in the reference genome that start from "AA", "CA" and "CC" are stored in three buckets, and their sizes are 6, 1 and 1.

When a seed "AAGG" in a short read is given, it is mapped to the bucket 0, and compared 6 keys in R_0. Then '1' is obtained as CAL of this seed. In the proposal method, the mask table is used to calculate its hash value. The {0,10} are read as the offset and mask bit from the mask table using the prefix "AA". The extended bits "G"='(10)' is masked by 10, and 1 is extracted. It is added to the offset, and finally, "AAGG" is mapped onto 1. In the original method, all seeds with prefix "AA" are mapped into 0. However, they are mapped into 0 and 1 in our method. In this way, the seeds with the frequently appeared prefixes are divided into several hash buckets to equalize $|R_i|$. Actually, the mask bit is an arbitrary bit sequence with l_e bits, and the prefix is divided into $2^{pop(m)}$ hash buckets, where m is the mask and $pop(m)$ is the number of 1's in m. For example, when m = 1011, seeds with the same prefix are mapped onto 2^3 hash buckets. On the other hand, the seeds with the less frequently appeared prefixes are merged into the same hash bucket like 'CA' and 'CC' in Fig. 9 by giving them the same offset.

By using the mask table, the seeds that start from "AA", "CA" and "CC" are mapped onto three buckets as well as the original method, but the size of the three buckets are equalized; 3, 3, 2 from 6, 1, 1, by the original method.

Here, we must mention one constraint on the mask table. The CAL table stores not whole field of seeds but only

Table 3 The improvement rate of C_r by equalizing $|R_i|$ and the mask table size (Mb) when $w = 2^{16}$

		l_p				
		14	15	16	17	18
l_e	8	2.02 (0.38)	2.11 (0.75)	2.15 (1.50)	2.21 (3.00)	2.26 (6.00)
	10	2.11 (0.41)	2.16 (0.81)	2.20 (1.63)	2.28 (3.25)	2.29 (6.50)
	12	2.17 (0.44)	2.21 (0.88)	2.21 (1.75)	2.31 (3.50)	2.32 (7.00)
	14	2.22 (0.47)	2.25 (0.94)	2.26 (1.88)	-	-
	16	2.25 (0.50)	-	-	-	-

† The theoretical maximum performance improvement is 2.62.

Table 4 Internal bucket size and C_c

w	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}
internal bucket size (MB)	0.875	1.75	3.5	7	14
C_c	24786	12609	6281	3182	1788

keys. We cannot distinguish two seeds of different first 12b if they are stored int the same hash bucket. Thus, the seeds in the same hash bucket must have the same first 12b. We must merge buckets based on this constraint.

3.3.2 Finding Better Mask Table

The next problem is how to calculate better mask table efficiently to equalize $|R_i|$. The problem to find an optimal mask table is a combinatorial problem, and it is not easy to find the optimal solution. Here, we propose a simple heuristic method based on two steps: dividing and merging.

1. Initialize a mask table:
   The mask table is initialized so as to output the same result as the original hash function.
2. Devide the largest hash bucket by randomly raising one bit of the mask bit.
3. Merge the smallest hash bucket with the same first 12b, by giving the same offset.
4. Repeat (2) and (3), until the search converges.
5. Finalize:
   All mask bits are refined to find a local optimum by evaluating all possible combinations within the number of 1's in the mask bit.

It takes several hours to calculate the mask table by our heuristic method, however, once it is calculated, it can be used for all individuals of the same species.

Table 3 shows the improvement rate of C_r by equalizing $|R_i|$ using our heuristic method, when $w = 2^{16}$. To calculate hash values faster, the mask table must be placed in on-chip memory in FPGA. This requirement limits the width of l_p and l_e as show in Table 3. The values in brackets are the size of each mask table in MB. The acceleration rates are close enough to the theoretical maximum. As show in this table, there is a tradeoff between speedup and the usage of FPGA resources. We have chosen $l_p = 15$ and $l_e = 14$ considering the balance of them. We have created the mask tables for each w, and Table 4 shows the size of the internal bucket and C_c. As shown in this table, C_c can be improved using the mask table.

Table 5 FPGA resource utilization

#LUTs (K)	#Registers (K)	#BRAMs	freq (MHz)
336 (78%)	120 (14%)	898 (61%)	200

Table 6 The mainly memory consumption

in FPGA	mask table	0.12 MB
	internal buckets	3.50 MB
in off-chip DRAM banks	index table	1.25 MB
	reference genome	0.78 GB
	external buckets	1.0 GB
	CAL table	2.84 GB

Table 7 Mapping accuracy by simulated data

mutation rate	2	4	6	8	10
BWA [3]	98.816	98.594	98.247	96.586	92.228
BOWTIE2 [2]	97.776	90.448	72.093	46.414	23.276
BFAST [4]	94.177	90.983	85.419	76.985	65.989
Our Approach	97.968	97.464	96.028	92.675	85.826

4. Implementation and Performance Evaluation

To implement our approach on an FPGA, we must decide P, w and the number of Smith-Waterman aligner according to the amount of available hardware resources. P is decided by the amount of configurable logic blocks (CLBs) in the FPGA, and w is decided by the size of available on-chip memory resources. Larger w is required to reduce the number of key comparisons, however, it requires more hardware resources. As shown in Table 4, with larger w, the more the performance is improved, however, the size of internal bucket also becomes larger. Our target device is Xilinx Virtex-7 XC7VX690T, and it has 52.92Mb (6.615MB) of on-chip memory (Block RAMs). We have implemented a circuit for $w = 16$, because it is the largest w for this FPGA.

In the scoring module, 16 Smith-Waterman aligners are implemented. The finding CAL module and the scoring module run concurrently, and when a sufficient number of SW aligners are prepared, the total execution time is decided by the execution time of the finding CAL module. 16 SW aligners are enough through all experiments.

We have implemented the circuit using Verilog HDL on a Virtex-7 XC7VX690T, and Vivado 2014.2 was used to compile it. The operating frequently is 200MHz to synchronize the DRAM interface (DDR-800). Table 5 shows the summary of the implementation result reported by Vivado. Table 6 shows the main memory consumption of on-chip memory (BlockRAMs) in FPGA and off-chip DRAM banks.

To compare the rate that are mapped correctly, we have created 100K simulated paired-end short reads with 100 base-pairs from the reference genome of human (GRCh37). Table 7 show the rate of the short reads that are mapped onto the true locations by software tools and our approach. The mutation rate (%) means the rate of the added mutation (the ratio of substitution, insertion and deletion is 0.75, 0.125, 0.125) per a base-pair. BWA [3], BOWTIE2 [2], and

Table 8 Execution time and the mappingg rate

	t (sec)	rate (%)	speedup
BWA [3]	4156	99.60	1.00
BOWTIE2 [2]	4117	96.19	1.01
BFAST [4]	32878	98.99	0.13
Olson et al. [5] (Virtex-6 XC6VLX240T)†	644	98.99	6.45
our approach (Virtex-7 XC7VX690T)	176	99.05	23.61

† performance with one XC6VLX240T is estimated from the eight FPGA system for 76 base-pairs short read.

BFAST [4] are software tools. As shown in this table, the mapping accuracy of our approach outperforms BOWTIE2 and BFAST, and very close to BWA.

We have evaluated the performances by using a real data set, 45M paired-end reads ($45M \times 2$ short reads) with 100 base-pairs (ERR251631), which were generated from whole genome sequence of a human by Illumina HiSeq 2000. The FPGA system by Olson et al. is one of the fastest FPGA systems for short read mapping. The software tools are executed on Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz with 32GB main memory by 8 threads. To create the index of BFAST, we used 'bfast index -f /ref/GRCh37/v37.fa -A 0 -m 1111111111111111111 -w 14 -i 1 -n 8' as arguments, which means that the length of seeds is 22 letters and their first 28b (14 letters) is used as the hash value. In other software programs, their default options are used. Table 8 shows the execution time and the mapping rate. Here, it should be noted that 'mapping rate' means not 'mapped correctly' but 'reported that at least one location is found'. Our system is about 24 times faster than BWA, which is one of the fastest software programs. The system by Olson et al. was implemented on eight Virtex-6 XC6VLX240T and the target short reads length is 76 base-pairs. In Table 8, we have estimated the performance on one XC6VLX240T for shot reads with 100 base-pairs. It is difficult to compare FPGA systems on different devices. In order to fairly compare both FPGA systems, we normalize the performances by the number of CLBs (the amount of hardware resources). Our system is about 3.7 times faster than the system by Olson et al. although our system requires 2.8 times of hardware resources. This mean that our system works more efficiently than the system by Olson et al. As shown in Fig. 8, when $s = 5$, the mapping rate of our system outperforms BWA when the mutation rate is high, and the performance is still 8 times faster than BWA.

5. Conclusion

In this paper, we proposed an FPGA acceleration method based on hash table to realize faster short read mapping. Although we have optimized our system for the whole human genome, our system works effectively on other target genomes which have similar order length (whole human genome is about 3G). First, to reduce the number of accesses to the hash table, the seeds in the short reads are sorted by their hash values, and P seeds with the same hash value are compared in parallel. Second, in the comparison between the seeds in the short reads and the reference genome, one mutation are allowed to improve the mapping accuracy. This extension enables to reduce the number of the seeds in the hash table to 1/15 while keeping the high mapping rate. Third, to equalize the number of the seeds that are hashed into each hash bucket, we proposed an optimized hash function using masks. We implemented our approach on Xilinx Virtex-7, and showed that its performance is more than 20 times faster than BWA, which is one of the fastest software program. Furthermore, the mapping rate of our approach is very close to BWA, which is one of the most accurate software tools.

Due to further improvement of NGS technology, the length of reads is becoming longer. In our approach, the hardware size of the finding CALs modules does not depend on the read length because these units process not the whole short reads but the seeds (fixed-length sub-sequences of the reads), and the performance of the modules is almost proportional to the total number of seeds which are extracted from the reads. The total number of seeds is almost the product of the read length and the total number of reads. In general, by enlarging the length of reads, the total number of them becomes smaller. Thus, we can consider that the longer length of the reads has only small effect on the finding CALs modules. On the other hand, the performance and hardware size of scoring modules are simply proportional to the read length. In our current implementation, those two kinds of modules run concurrently, and when the length of short reads is 100 base pairs, the total performance is dominated by the finding CALs modules. However, for longer reads, this balance will be gradually changed, and when they are longer than 500 base pairs, the scoring modules become the bottleneck. In this case, we need to re-balance the system. These are our future work.

References

[1] B. Langmead, C. Trapnell, M. Pop, and S.L. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," Genome Biology, vol.10, no.3, 2009.

[2] B. Langmead and S.L. Salzberg, "Fast gapped-read alignment with bowtie 2," Nature Methods, vol.9, no.4, pp.357–359, 2012.

[3] H. Li and R. Durbin, "Fast and accurate short read alingment with burrows-wheeler transform," Bioinformatics, vol.25, no.4, pp.1754–1760, 2009.

[4] 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.

[5] C.B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W.L. Ruzzo, "Hardware acceleration of short read mapping," FCCM, pp.161–168, 2012.

[6] J. Arram, K.H. Tsoi, W. Luk, and P. Jiang, "Hardware acceleration of genetic sequence alignment," ARC, vol.7806, pp.13–24, 2013.

[7] H.M. Waidyasooriya, M. Hariyama, and M. Kameyama, "Fpga-accelerator for dna sequence alignment based on an efficient data-dependent memory access scheme," Proc. 5th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART), pp.127–130, 2014.

[8] E.B. Fernandez, J. Villarreal, S. Lonardi, and W.A. Najjar, "Fhast: Fpga-based acceleration of bowtie in hardware," Computational Biology and Bioinformatics, IEEE/ACM Transactions on, vol.12, no.5, pp.973–981, 2015.

[9] Y. Sogabe and T. Maruyama, "An acceleration method of short read mapping using fpga," Field-Programmable Technology (FPT), 2013 International Conference on, pp.350–353, IEEE, 2013.
[10] Y. Sogabe and T. Maruyama, "Fpga acceleration of short read mapping based on sort and parallel comparison," Field Programmable Logic and Applications (FPL), 2014 24th International Conference on, pp.1–4, IEEE, 2014.
[11] Y. Sogabe and T. Maruyama, "A variable length hash method for faster short read mapping on FPGA," Field Programmable Logic and Applications (FPL), 2015 25th International Conference on, pp.1–6, IEEE, 2015.
[12] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," Proceedings of the Annual Symposium on Foundations of Computer Science, pp.390–398, 2000.
[13] M. Burrows and D.J. Wheeler, "A block-sorting lossless data compression algorithm," Digital Equipment Corporation, Technical report 124, 1994.
[14] Y. Yamaguchi, T. Maruyama, and A. Konagaya, "High speed homology search with FPGAs," PSB, pp.271–282, 2002.

Yoko Sogabe received the B.S. and M.S. (engineering) degrees from university of Tsukuba in 2013 and 2015. He is now phD student of university of Tsukuba. His research interest is an accelerated implementation on FPGA and GPU for applications of bioinformatics, etc.

Tsutomu Maruyama is a professor of the faculty of engineering, information and systems at University of Tsukuba. He received his Doctor of Engineering degree from Tokyo University in 1987. His research interest is in the application acceleration using FPGAs and GPUs.