Implementation of an FPGA Accelerator for Text Search Using a Wavelet-Tree-Based Succinct-Data-Structure

Taisuke Ono, Hasitha Muthumala Waidyasooriya
and Masanori Hariyama

Graduate School of Information Sciences, Tohoku University
6-3-09, Aramaki-Aza-Aoba, Aoba, Sendai, Miyagi 980-8579, Japan
{ono52@dc., hasitha@, hariyama@}tohoku.ac.jp

Abstract. Succinct data structures are introduced to efficiently solve a given problem while representing the data using as little space as possible. The full potential of the succinct data structures have not been utilized in the software-based implementations. This paper discusses an FPGA-based hardware architecture for text search that uses succinct data structures. We implemented an FPGA accelerator for text search using hardware-oriented succinct data structure. The proposed architecture is designed using OpenCL-based high-level design environment, and we obtained 40% reduction of the processing time compared to a multicore CPU based implementation.

1 Introduction

Succinct data structures [1] are introduced to efficiently solve a given problem while representing the data using as little space as possible. If the original data contain n bits, "a little space" means that the storage space is in the order of n (O(n)). To efficiently solve the problem, the processing time should not increase exponentially with the database size. Usually, $O(1)$ processing time is preferred. Due to such benefits, we can use this method for text search of extremely large databases without concerning its size. Succinct data structures are extensively used in genome sequence alignment [2]. Although a genome contains billions of symbols, it is possible to search a massive amount of DNA patterns in a few hours. In addition, huge speed-ups have been achieved by employing parallel computations by designing custom accelerators [3].

Due to the huge success of pattern search in genomes, we can consider using succinct data structures for general text search. General text contains many different symbols compared to just four in genomes. For example, English text has over hundred symbols (including numbers, upper, lower cases etc.), Japanese and Chinese text has thousands of symbols. Such large amount of symbols increase the storage size significantly. As a result, a large amount of storage space and storage access are required and that increases the processing time.

To reduce the storage size, a succinct data structure called the wavelet tree [4] has been introduced. However, it is difficult to achieve the full potential of such data structures using software, due to the lack of parallelism and the inefficient memory access. In our earlier work [5], we have introduced a basic FPGA accelerator architecture to process general text using a wavelet-tree based succinct data structure. The FPGA accelerator contains a large number of compact processing elements (PE). The data paths between the PEs and the memory can be designed to efficiently use the full memory bandwidth. The decompression/decoding can be done in parallel in minimum number of clock cycles.

In this paper, we improved the basic architecture proposed in [5], and implemented it on an FPGA. An FPGA is a reconfigurable LSI that contains millions of programmable logic gates. Recently, speed and power consumption of the FPGAs have been greatly improved, and it would be very practical to use the FPGA-based platform for real applications. We also used OpenCL-based high-level design environment [6], so that accelerator can be designed and improved much efficiently according the requirements of the algorithm. We also evaluated the FPGA accelerator and obtained 40% reduction of the processing time compared to multicore CPU implementation. We also observed that the processing time is independent of the database size.

2 Text search using succinct-data-structures

This section briefly explains how the text search is done. A more detailed explanation about the search method is available in our previous work [5].

2.1 Search using *rank* operation

In order to search a query in a text database X, we first apply Burrows-Wheeler transform (or BW transform) [7] and obtain text B. The Text B has the same size of the original text X. If a query q is a substring of the text X and $k(aq) \leq l(aq)$, the query aq is also a substring of X where aq equals the query $\{a, q\}$ [8]. 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.

$$k(aq) = C(a) + rank\,(B, a, k(q) - 1) \tag{1}$$

$$l(aq) = C(a) + rank\,(B, a, l(q)) - 1 \tag{2}$$

The operation $rank\,(B, a, x)$ returns the number of "a"s from a text B up to the position x. The element a could be any symbol such as a number, a letter, a byte, etc. The number of symbols that are lexicographically smaller than a is given by $C(a)$ where $a \in B$. The suffix array interval is given by $[k, l]$ and it corresponds to the position of query aq in text database X. We can find the position of a query Q by repeatedly applying Eqs.(1) and (2) to every symbol in the query as shown in algorithm 1.

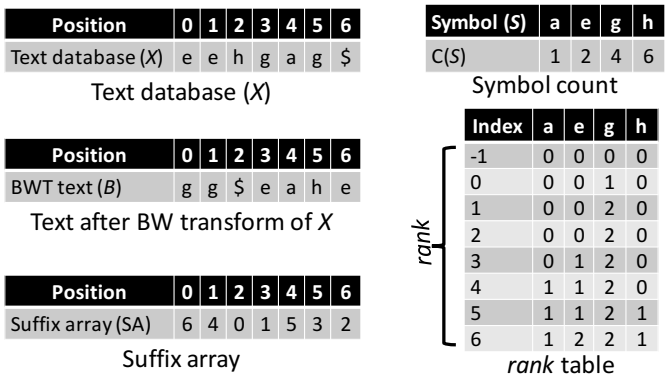
```
1  Search(Q, i, k, l)
2  begin
3  |  I = φ
4  |  k = 0
5  |  l = |X|
6  |  for  i = |Q| − 1 to i = 0 do
7  |  |  k = C (Q[i]) + rank (B, Q[i], k − 1)
8  |  |  l = C (Q[i]) + rank (B, Q[i], l) − 1
9  |  |            //B is the BWT string of X
10 |  |  if k ≤ l then
11 |  |  |  i = i − 1
12 |  |  else
13 |  |  |  return  φ //return empty
14 |  |  end
15 |  end
16 |  if  i == 0 then
17 |  |  return  [k, l]
18 |  end
19 end
```

Algorithm 1: Text search algorithm

To explain the algorithm, let us consider the example in Fig.1. As shown in Fig.1(a), we pre-process the text database X by doing BW transform on it. Then we create the suffix array, symbol count and *rank* table. The search query Q and the search procedure is shown in Fig.1(b). After the search, the suffix array interval (SA) is [6,6] so that we can find the actual position using the suffix array in Fig.1(a). In this case, $SA[6,6] = 2$. The search is done in 3 steps proportional to the number of symbols in the query Q.

2.2 Data storage and processing time

In the initial work of succinct data structures [1,9,10], a method to store the *rank* table to compute the *rank* in a constant time has been proposed. This method is based on a block structure. First, we store the *rank*s of the first symbol of each block. Then we use a look-up table to store the *rank* to every possible query of the symbol pattern in each block. All arrays and tables can be implemented using O(n) bits, and it supports *rank* operations in a constant time. Fig.2 shows an example of how the *rank* table is stored. The BW transformed text is divided into block of four symbols. The first entry of a block is the *rank* of the first symbol. Next, we store the rest of the 3 symbols. For example, $rank(B, 1, 6)$ is done as follows. Since the 6^{th} symbol belongs to the block 2, we access it and get the *rank* upto the 4^{th} symbol, which is 3. Then we search the bit pattern [010] of block 2 in the look-up table. It shows there are one 1's at the second symbol. Now we can add 3 and 1 to calculate $rank(B, 1, 6) = 4$. More details of this method can be found in [1, 9].



(a) Pre-processing of the text database.

Position	0	1	2
Search text (Q)	h	g	a

$k( ) = 0, \quad l( ) = 6$
$k(a) = C(a) + rank(B, a, 0 - 1) = 1 + 0 = 1$
$l(a) = C(a) + rank(B, a, 6) - 1 = 1 + 1 - 1 = 1$

$k(ga) = C(g) + rank(B, g, 1 - 1) = 4 + 1 = 5$
$l(ga) = C(g) + rank(B, g, 1) - 1 = 4 + 2 - 1 = 5$

$k(hga) = C(h) + rank(B, h, 5 - 1) = 6 + 0 = 6$
$l(hga) = C(h) + rank(B, h, 5) - 1 = 6 + 1 - 1 = 6$

Suffix array interval = $[6, 6]$

(b) Searching the query Q.

Fig. 1. Example of a text search using *rank*.

This method is successfully used for bit arrays where the symbol are either "0" or "1". In order to use this method for general text, we have to store the *rank*s of every symbol. If there are m number of different symbols, the storage size is increased by m. For example, we need over a hundred times large storage to search English texts, and we need over a thousand times large storage to search Japanese or Chinese texts. In such cases, we have to store the data not in the memory but in an SSD of hard drive. That increases the data access time dramatically and reduces the processing speed.

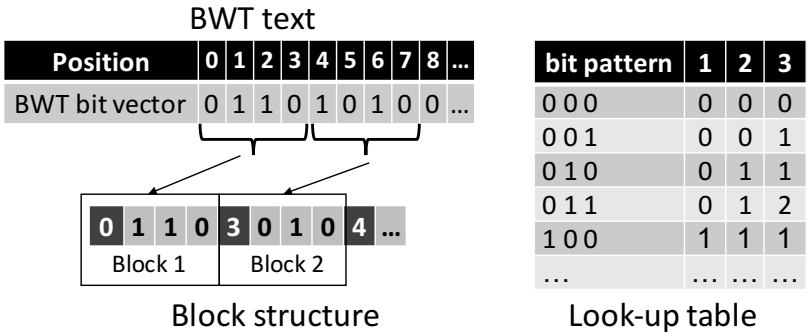
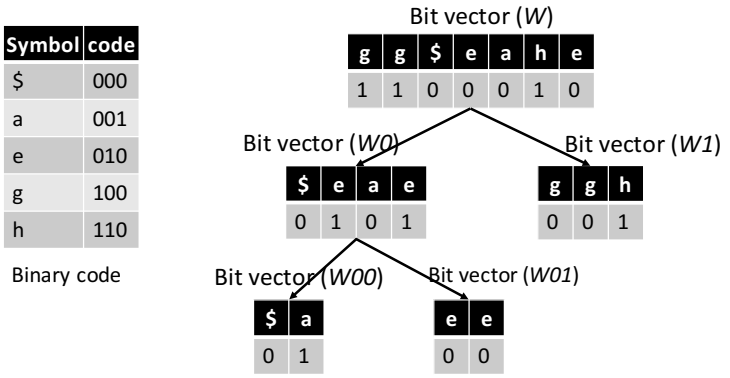


Fig. 2. Storage of *rank* data

2.3 Wavelet tree based data structure

The wavelet tree proposed in [4] permits a way to compute the *rank* of an arbitrary alphabet of size m efficiently. Let us explain the construction of the wavelet tree using the example in Fig.3. In this example, we use the text database X shown in Fig.1. A code is assigned to every symbol in the database as shown in Fig.3(a). The construction of the wavelet tree starts from the most significant bit (MSB) of the code. A bit vector W is created by using the MSB of each symbol in the text. That is, "0" is assigned to the symbols "$, a, e" and "1" is assigned to the symbols "g, h". Then we divide the bit vector in to two groups. One group contains the symbols that their corresponding bits in W are 0. The other group contains the symbols that their corresponding bits in W are 1. Then we assign bit vectors $W0$ and $W1$ for each group using the second most significant bit. This process continues until a unique bit (0 or 1) is assigned for every symbol in a group. Fig.3(b) shows how to compute *rank* using wavelet tree. In this example, $rank(B, e, 4)$ is considered. Note that, B is the BW transformed text of X shown in Fig.1. The computation of *rank* is done from the top to the bottom of the wavelet tree. Since the MSB of the symbol "e" is zero, we compute $rank(W, 0, 4)$. Then we come down to the second level of the wavelet tree and use the input vector $W0$, since "e" is included in the group that the MSB of "e" is zero. Then $rank(W0, 1, 2)$ is computed. Similarly, the computation is done for all the levels in the wavelet tree as shown in Fig.3(b).

In each level of the wavelet tree, there are only 2 symbols, "0" and "1". There are also $log(m)$ levels of bit vectors for m types of symbols. Therefore, the storage size of the *rank* data is reduced by $m/log(m)$ times compared to the method without using wavelet tree.



(a) Wavelet tree representation.

Calculate $rank(B, e, 4)$

Start from the most significant bit (MSB) of code e

MSB(code e) = 0, search in *W* — $rank(W, 0, 4) = 2$

next bit of code e = 1, search in *W0* — $rank(W0, 1, 2) = 1$

next bit of code e = 0, search in *W01* — $rank(W01, 0, 1) = 1$

$rank(B, e, 4) = 1$

(b) Computation of $rank(B, e, 4)$.

Fig. 3. Example of *rank* computation using wavelet tree.

3 FPGA accelerator for text search using OpenCL

To accelerate the wavelet-tree based text search using hardware, we consider two important things, one is to reduce the memory access time as much as possible, and the other is to increase the amount of parallel operations. We propose an FPGA-oriented data structure for fast memory access and OpenCL-based accelerator architecture for parallel processing.

3.1 FPGA-oriented data structure for fast memory access

FPGA contains a hierarchical memory structure consists with off-chip global memory and on-chip local memory. The global memory is large in size but the access time is very long. On the other hand, local memory is very small in size but the access time is short. Therefore, we propose a data structure to store the bit vectors of the wavelet tree using a little space as possible and can access using minimum number of transactions, using both global and local memories efficiently.

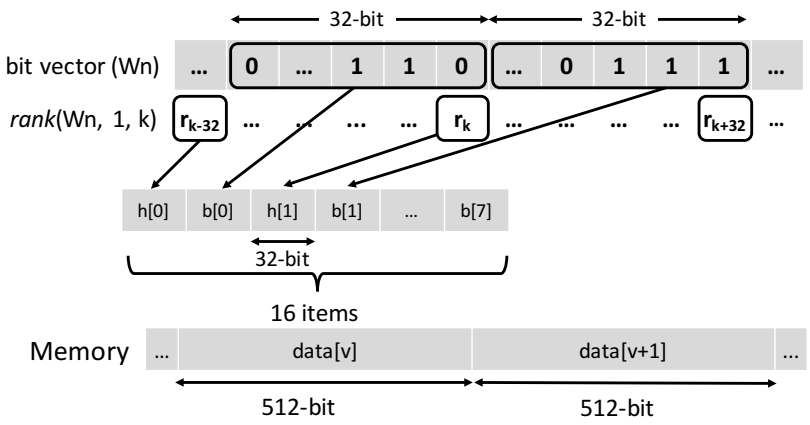


Fig. 4. The proposed data structure to store the *rank* data in the global memory of the FPGA board.

Fig.4 shows the proposed data structure. It shows how to store the data of one bit vector belonging to one level of the wavelet tree. The same method is used to store the bit vectors belonging to the other levels. We divide the bit vector into groups of 32bits. We calculate the *rank* of the first bit of the each group and arrange the *rank* data and the data of the next group as shown in Fig.4. In one memory access transactions, FPGA accesses 512bits of global memory data. Therefore, we can store 8 *rank* entries and 8 groups.

The computation of *rank* is done as follows. To compute $rank(W, 1, k)$, we first find the memory address of the group that contains the k^{th} bit, that is the group $\lfloor k/32 \rfloor$. Since one 512bits memory region contains 8 groups of 32 bits, that is 256bits, we can easily find the memory address of the relevant group. Next, we can get the *rank* until the previous group, that is the group $\lfloor k/32 \rfloor - 1$. Then we have to count the number of "1"s untill the relevant bit in the group $\lfloor k/32 \rfloor$. This is done by a table look-up. We remove the unnecessary bits, that is the bits after the k^{th} bit by applying a bit mask. Then we divide the 32bits group into 4 small groups where each contain 8 bits. For each bit pattern in the 8bit group, we store the bit counts in the local memory as shown in Fig.5. The look-up table contain 256 entries. Using this table, we can find the bit count and add it to the *rank* of the previous group to compute $rank(W, 1, k)$.

In the proposed data structure, we can reduce the storage size of the global memory further by increasing the group size. However, that also increase the number of bits patterns in a group. As a result, we need a large look-up table to store the bit counts.

bit pattern	bit count after masking
0000_0000	0
0000_0001	1
...	...
0001_0100	2
...	...

Fig. 5. Look-up table to find the bit counts.

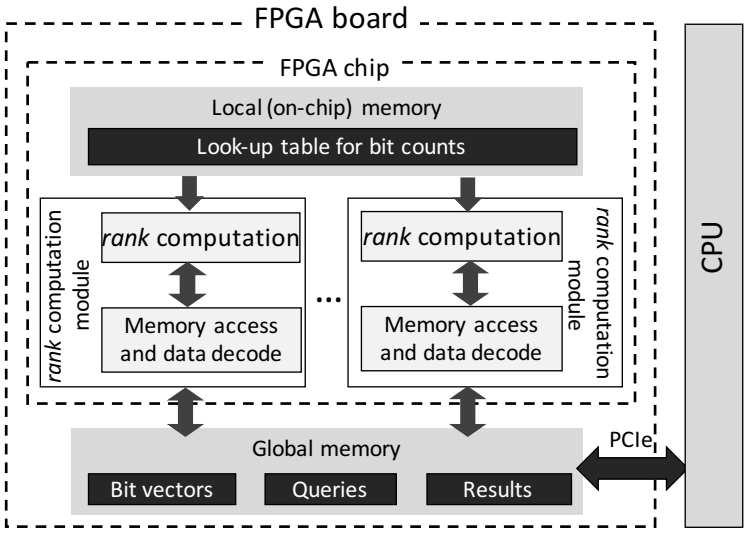


Fig. 6. FPGA architecture for text processing.

3.2 OpenCL-based FPGA accelerator for parallel processing

Fig.6 shows the proposed FPGA accelerator architecture. After the BW transform and wavelet bit vector generation is done in the CPU, those data are transferred to the global memory of the FPGA. The search queries are also transferred to the global memory. The FPGA contains multiple *rank* computation modules. Each module computes a *rank* of a different query independently. The look-up tables are shared between all modules. The computed results are written back to the global memory and then transferred to the CPU.

Listing 1.1 shows the OpenCL-based kernel code for the implementation. Since the *rank* computation modules operate independently, the data access is irregular and the processing times are different in each module. Therefore, we cannot use SIMD type parallelism. Instead, we use MIMD type parallelism by

using an "ND-range kernel" in OpenCL with 18 compute units. Each compute unit corresponds to a *rank* computation module and operates independently. The disadvantage of this kernel implementation is the generation of multiple datapaths to the global memory. Since each compute unit operates independently, they use dedicated paths to the global memory and that increases the resource utilization.

```
uint count( global uint2_8* restrict x,
                        ...,
                        uint end)
{
  //decoding
  uint x_index = end / BS_X;
  uint x_bit   = end % BS_X;
  uint2_8 y = x[offset + x_index];
  ...

  //refering look-up tables
  uint count = ((end / BS_Y) == 0) ? 0 : cell.s0;
  count += (T[bit.c[0]] + T[bit.c[1]] + T[bit.c[2]] + T[
    bit.c[3]]);
  return count;
}

__attribute__((num_compute_units(18)))
__attribute__((reqd_work_group_size(64, 1, 1)))
__kernel void rank(     global uint2_8* restrict x,
                        ...
                        global uint2* restrict query,
                        global uint* restrict out)
{
  size_t id = get_global_id(0);
  uint2 data = query[id];
  ...

  for(uint j = 0; j < HEIGHT; j++) {
    ...
    start = count(x, offset, is_one, start);
    end   = count(x, offset, is_one, end);
  }
  out[id] = end - start;
}
```

Listing 1.1. OpenCL-based kernel code.

4 Evaluation

We implement the proposed FPGA accelerator on Terasic DE5a-Net board [11] that has an Intel Arria 10 10AX115N2F45E1SG FPGA. The OpenCL-based

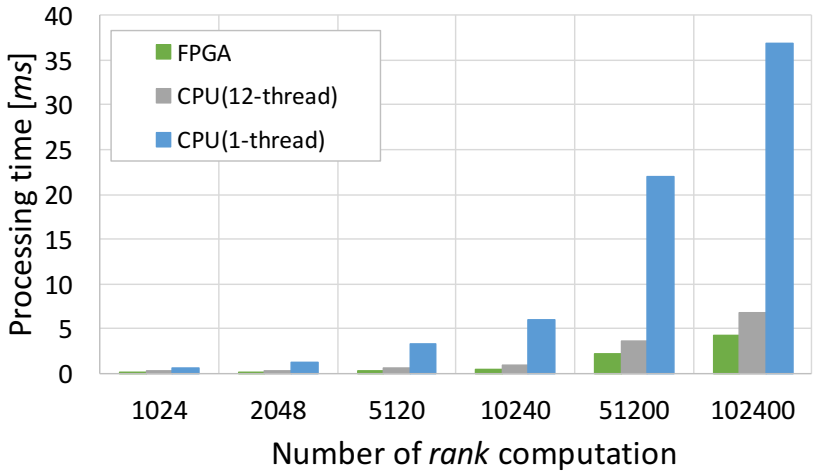


Fig. 7. Relationship between the processing time and the number of rank operations

kernel codes are compiled using Intel FPGA SDK for OpenCL version 16.1. For the comparison with the conventional CPU-based implementation, we use a workstation that has an Intel Xeon E5-1650 v3 CPU. The program of the *rank* computation for the CPU is written in C-language with OpenMP parallelization, and compiled using GNU C-compiler with relevant optimization options. The text database has 256 different characters, so that the wavelet matrix has 8 bit vectors ($log(256)$).

Fig.7 shows the comparison of the processing times of the proposed FPGA accelerator and the conventional CPU-based implementation. We measure the processing time by changing the number of *rank* computations. The text database size is 10MB. When the number of *rank* computations is 2048, the processing time of the proposed FPGA accelerator is approximately 11% of that of the single thread CPU implementation and 42% of that of the 12-thread CPU implementation.

Fig.8 shows the comparison of the processing times for different database sizes. In this evaluation, the number of *rank* operations are 2,048 and the text database size changes from 1MB to 1000MB. We can see that the processing time of the CPU implementation is slightly increasing until the text database size is 100MB. Then there is a significant increase when the database size is 1000MB. A major reason for this could be the CPU cache. For small databases, the involvement of the cache is very large. As a result, the global memory access (DDR3 access) is small and the processing time is small. When the database size increases, the data are not fit into the cache and the processing time increases. On the other hand, we can see a constant processing time for all database sizes for the FPGA accelerator. FPGA accelerator does not use cache, so that we can expect a constant processing time for even larger databases. According to

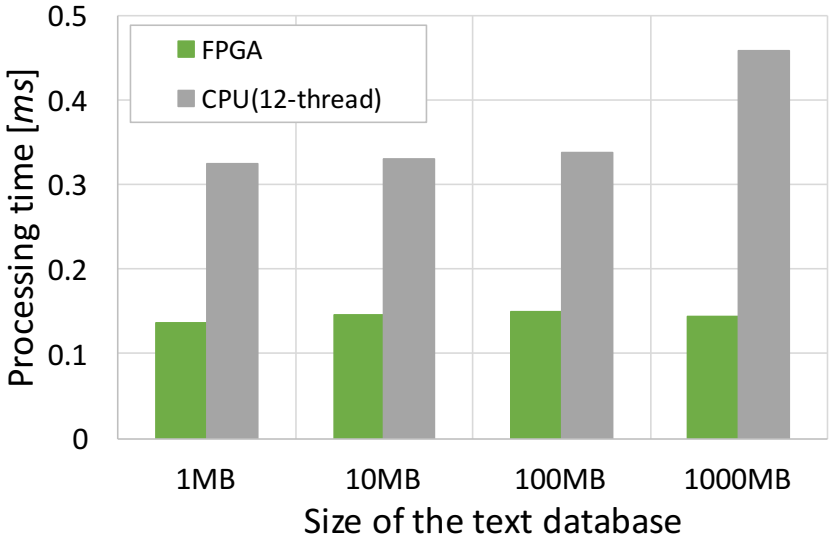


Fig. 8. Relationship between the processing time and the size of the text database

Table 1. Resource utilization

Resource type	**Utilization**
Logic utilization	202,396 (47%)
DSP blocks	36 (2%)
Memory	3.64 MB (55%)
RAM blocks	2,539 (94%)

the theoretical background of the succinct data structures [1], the processing time for *rank* operations must be a constant and should not increase with the database size. This evaluation shows that, we can achieve the full potential of the succinct data structures using the proposed accelerator. When the length of text is 1000MB, the processing time of the proposed FPGA accelerator is approximately 31% of that of the 12-thread CPU-based implementation.

The implemented accelerator operates at 232MHz. The resource utilization is shown in Table 1. Although less than half of the logic blocks are used, the parallelism is restricted due to large RAM block usage. RAM blocks are mainly used as look-up tables for bit counts and also to temporally store the data from the global memory. Increasing the sharing of RAM blocks should be considered in future to reduce the resource utilization and to increase the degree of parallelism.

5 Conclusion

We have proposed an FPGA accelerator for text search using a wavelet-tree-based succinct data structure. The FPGA accelerator produces a constant processing time even the database size increases. This is a major benefit of the succinct data structures, since we can use them with extremely large databases without concerning their size. We also find that the processing time of the text search using the proposed accelerator is less than 40% compared to multicore CPU implementation.

References

1. G. Jacobson, "Succinct static data structures. PhD thesis", Carnegie Mellon University, 1989.
2. Heng Li and Richard Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform", Bioinfomatics, Vol.25, No.14, pp.1754-1760, 2009.
3. Hasitha Muthumala Waidyasooriya and Masanori Hariyama, "Hardware-Acceleration of Short-read Alignment Based on the Burrows-Wheeler Transform", IEEE Transactions on Parallel and Distributed Systems, Vol.27, No.5, pp.1358-1372, 2016.
4. R. Grossi, A. Gupta, J.S. Vitter, "High-order entropy-compressed text indexes", Proc. of ACM-SIAM Symposium on Discrete Algorithms (SODA'03), pp.641-650, 2003.
5. Hasitha Muthumala Waidyasooriya, Daisuke Ono, Masanori Hariyama and Michitaka Kameyama, "An FPGA Architecture for Text Search Using a Wavelet-Tree-Based Succinct-Data-Structure", International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), pp.354-359, 2015.
6. `https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html`, 2017.
7. M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm", Digital Equipment Corporation, Palo Alto, CA, Technical report 124, 1994.
8. P. Ferragina and G. Manzini, "Opportunistic data structures with applications", Proc. of 41st Symp. on Foundations of Computer Science, pp.390-398, 2009.
9. G. Jacobson, "Space-efficient static trees and graphs", 30th Annual Symposium on Foundations of Computer Science, pp.549-554, 1989.
10. Rajeev Raman, Venkatesh Raman and S. Srinivasa Rao, Succinct indexable dictionaries with applications to encoding k-ary trees and multisets, Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms Pages 233-242, 2002.
11. `https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=228&No=970&PartNo=2`, 2017.