An FPGA Architecture for Text Search Using a Wavelet-Tree-Based Succinct-Data-Structure

Hasitha Muthumala Waidyasooriya, Daisuke Ono, Masanori Hariyama and Michitaka Kameyama
Graduate School of Information Sciences, Tohoku University
Aoba 6-6-05, Aramaki, Aoba, Sendai, Miyagi, 980-8579, Japan
Email: {hasitha, ono1831, hariyama, kameyama}@ecei.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 proposes a hardware-oriented data structure and its decoding method. The proposed architecture can be used in text searches using up to 4.3GB large text files.*

Keywords: Succinct data structures, text-search, FPGA.

1. Introduction

Succinct data structures [1] are introduced to efficiently solve a given problem while representing the data using as little space as possible. Such data structures are used in many fields such as bio-informatics, text processing, etc. To solve the problem efficiently, the original data are usually pre-processed. If the original data contain n bits, "a little space" means that the storage space of the pre-processed data must be in the order of n (O(n)). To efficiently solve the problem, the processing time must be in the order of 1 (O(1)). That is, the processing time does not depend on the input data size.

Although the data storage size is in the order of n, the actual storage size is $k \times n$, where k usually takes a value from tens to thousands. As a result, the storage size of the succinct data structure is many times larger than the original data size. However, recent computers have a very large memory capacity and extremely large hard disk space. Therefore, implementing such data structures is possible and some of those implementations have given reasonably good results. However, they have many limitations so that the full potential of the succinct data structures have not been utilized. The main problem is the memory access bottleneck. Although the processing time is independent of the data size, the memory access is unpredictable and requires many clock cycles. Moreover, the data are usually in a compressed or encoded state, so that a decompression or decoding overhead is required. Therefore, it is often a serious challenge to efficiently utilize the succinct data structures for massively parallel implementations.

Designing a custom hardware is a good solution to such problems. A custom hardware contains a large number of compact processing elements (PE) that are specialized to solve only the given problem. 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. we consider an FPGA-based accelerator for text search applications. An FPGA is a reconfigurable LSI that contains millions of programmable logic gates. Recently, speed and power consumption of the FPGAs are greatly improved, and it would be very practical to use the FPGA-based platform for real applications. However, the lack of huge DDR3 memories is a major problem in FPGA boards. Many high-end FPGA boards contain just 4 GB of memory capacity. Therefore, we have to compress the data as much as possible while still allowing the efficient access to the data.

To implement succinct data structures on hardware, we can not rely on the order of the computations. Even the order is small, the processing time or the storage space could be so large that the data structure may not be implemented on the hardware. In this paper, we consider the factors such as the memory bandwidth, word width of the memory, storage size etc to find a hardware-compatible succinct data structure, which could actually be implemented on the hardware. We also consider hardware-oriented data compression method to reduce the storage space further without increasing the processing time.

2. Succinct-Data-Structure

2.1 Text search using rank

In this paper, we limit the given problem to the text search. The operation $rank_q(T, x)$ returns the number of "element q"s from a text T up to the position x. The element q could be any symbol such as a number, a letter, a byte, etc, and T is an array that contain many elements. The implementation of the *rank* operation in a constant time is presented in [1], [2]. Using the *rank*, a quarry can be searched in a text with a processing time proportional to the size of the quarry and not proportional to the size of the text. That is, the search time does not increase with the size of the text.

```
Search(Q, i, k, l)
begin
    I = φ
    k = 0
    l = |X|
    for i = |Q| − 1 to i = −1 do
        if i == −1 then
            return [k, l]
        end
        k = C(Q[i]) + rank_{Q[i]}(B, k − 1)
        l = C(Q[i]) + rank_{Q[i]}(B, l) − 1
                    //B is the BWT string of X
        if k ≤ l then
            i = i − 1
        else
            return φ //return empty
        end
    end
end
```

Algorithm 1: Text search algorithm

The text search method is shown in algorithm 1. In this algorithm, the search quarry Q is searched in the text X. The number of elements in X and Q are given by $|X|$ and $|Q|$ respectively. The text X is pre-processed to construct the *rank* table and the array $C(.)$. We explain the pre-processing using an example in Fig.1. The text X is shown in Fig.1(a) where the end of the text is identified by "$". As shown in Fig.1(b), the text is shifted to the left until all the symbols are moved. The shifted (rotated) text is sorted in lexicographical order as shown in Fig.1(c). The suffix array (SA) in Fig.1(c) shows the sorted array of all the suffixes. This rotation and sorting is also called the Burrows-Wheeler transform (or BW transform) [3] and the string in the last column of Fig.1(c) is called the "BWT string" and denoted by B. Then we count the number symbols from the beginning to each index and put those values on a table. This "*rank*" table is shown in Fig.1(d). For example, $rank_E(B, 3) = 1$, since there are only one "E" appears from the index 0 to 3 in the *rank* table. The number of symbols that are lexicographically smaller than a is given by $C(a)$ where $a \in B$.

Fig.2 shows the searching of the quarry (Q) in text (X). The quarry Q and $C(.)$ array are shown in Figs.2(a) and 2(b) respectively. According to [4], if a quarry q is a substring of the text X and $k(aq) \leq l(aq)$, the quarry aq is also a substring of X where aq equals the quarry $\{a, q\}$. 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.

$$k(aq) = C(a) + rank_a(B, k(q) - 1) \quad (1)$$

$$l(aq) = C(a) + rank_a(B, l(q)) - 1 \quad (2)$$

We can find the position of Q in X by repeatedly applying Eqs.(1) and (2) to every symbol in Q as shown in Fig.2(c).

Position	0	1	2	3	4	5	6
Text (X)	E	E	H	G	A	G	$

(a) Text data (X)

Position	0	1	2	3	4	5	6
0	E	E	H	G	A	G	$
1	E	H	G	A	G	$	E
2	H	G	A	G	$	E	E
3	G	A	G	$	E	E	H
4	A	G	$	E	E	H	G
5	G	$	E	E	H	G	A
6	$	E	E	H	G	A	G

(b) Moving (rotation) of the text

Suffix array		0	1	2	3	4	5	6
Index	Position in X							(BWT string B)
0	6	$	E	E	H	G	A	G
1	4	A	G	$	E	E	H	G
2	0	E	E	H	G	A	G	$
3	1	E	H	G	A	G	$	E
4	5	G	$	E	E	H	G	A
5	3	G	A	G	$	E	E	H
6	2	H	G	A	G	$	E	E

(c) Sorting of text data

a	A	E	G	H
C(a)	1	2	4	6

Index	A	E	G	H
0	0	0	1	0
1	0	0	2	0
2	0	0	2	0
3	0	1	2	0
4	1	1	2	0
5	1	1	2	1
6	1	2	2	1

(d) $C(.)$ and *rank* data

Fig. 1: Pre-processing the text

The suffix array interval (SA) is [6,6] so that we can find the actual position using the suffix array in Fig.1(c). In this case, $SA[6, 6] = 2$. The search is done in 3 steps proportional to the number of symbols in the quarry Q.

2.2 Data storage and processing time

In the initial work of succinct data structures [1], a method to store the *rank* data and compute the *rank* in a constant time is proposed. Given a binary sequence $B[0, n-1]$ of size n, a two-level directory structure is built. The first level contains large blocks of size $log_2 n \times log_2 n$. For each large block, the *rank* of the first entry is stored in a separate array. This requires $n/log_2 n$ storage. Each large block is divided in to small blocks of size $log_2 n/2$. Therefore, each large block contains $2log_2 n$ number of small blocks. Within a large block, another array is used to store the rank of the first entry of all small blocks. For all large blocks, this array requires $4nlog_2(log_2 n)/log_2 n$ bits. A look-up table is used to store the answer to every possible *rank* on a bit string of size $log_2 n/2$. It requires $2^{log_2 n/2} \times log_2/2 \times log_2(log_2 n/2)$ bits. All arrays and tables can be implemented using O(n) bits, and it supports *rank* queries in a constant time. Please refer [1] for more details. Since we use many arrays and tables, this method needs multiple (although a constant number of) memory reads to compute the *rank*. Moreover, this method is proposed for bit vectors. It is not efficient to use this

Position	0	1	2
Quarry (Q)	H	G	A

(a) Search quarry (Q)

a	A	E	G	H
$C(a)$	1	2	4	6

(b) C array

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

$k(GA) = C(G) + rank_G(B, 1-1) = 4 + 1 = 5$
$l(GA) = C(G) + rank_G(B, 1) - 1 = 4 + 2 - 1 = 5$

$k(HGA) = C(H) + rank_H(B, 5-1) = 6 + 0 = 6$
$l(HGA) = C(H) + rank_H(B, 5) - 1 = 6 + 1 - 1 = 6$

$SA[6,6] = 2$

(c) Searching of the quarry (Q) in text (X)

Fig. 2: Searching the quarry Q in text X

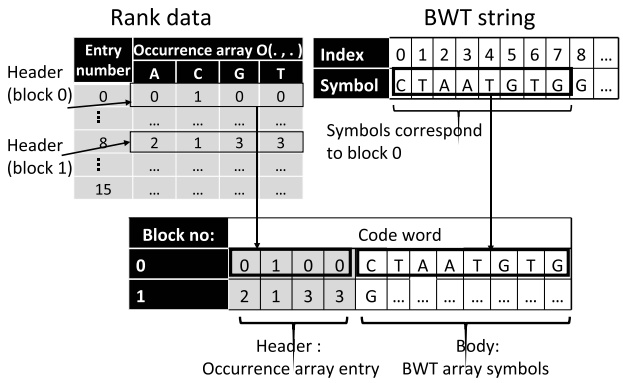


Fig. 3: *rank* data encoding of a human genome

method when the input is not a bit vector but contain multiple characters, such as general text.

A different data structure for the multi-character text search is proposed in bio-informatics applications such as short-read alignment [5]. In short-read alignment, a short DNA fragment is searched in a large genome, which is basically a text search. Genome data contains 4 symbols, "A,C,G,T" that are represented by 2 bits. Fig.3 shows a *rank* table of 16 entries. We divide the rank data table into two blocks where each block contains 8 entries. Then the first entry is chosen as the header. The rest of the entries are replaced by the BWT symbols. Since one BWT symbol has significantly smaller size compared to a *rank* entry, this method reduces the storage size. However, in the decoding, we have to count the number of symbols of each character in the body. To do this in constant time, we need a population count (popcount) hardware.

A human genome contains approximately 3 billion symbols. Therefore the rank table contain 3 billion entries where each entry has $log_2(3\text{ billion}) \times 4$ bits. That is 128 bits. Therefore, the storage space for the rank table requires 128×3 billion bits which is approximately 48GB. The above encoding method is used in [6] to successfully implement the short-read alignment using just 1.5 GB of data. In [6], the *rank* table is divided in to blocks where each block contain 64 entries. The first entry of each block is used for the header, which requires 128 bits. The rest of the entries in each block are replaced by the BWT symbols. Since 2 bits are required to represent the "A,C,G,T" symbols, the body contains only 128 bits (2×64). Therefore, one code word is 256 bits and we need 3 billion/64 of such code words. That is 1.5 GB. Moreover, 256 bits can be read in one memory read in FPGA. Note that, one memory read provides the access to a block of consecutive data. The popcount of 64 symbols can be done in a few steps in hardware and many such popcount architectures are already proposed [7].

To use this method in text search, let us consider a general case that has n symbols in the BWT string. We consider m different symbols in the alphabet. Therefore, one rank data entry requires $m \times log_2 n$ bits. Since there are n entries we need a total of $n \times m \times log_2 n$ bits. We divide the rank data into multiple blocks where each block contain p entries. In each block we store the first entry as the header. The rest of the entries in a block is replaced by the symbols in the BWT string. Therefore, the required total bits (T_{bit}) is given by Eq.3.

$$T_{bit} = (m \times log_2 n) \times \frac{n}{p} + n \times log_2 m \tag{3}$$

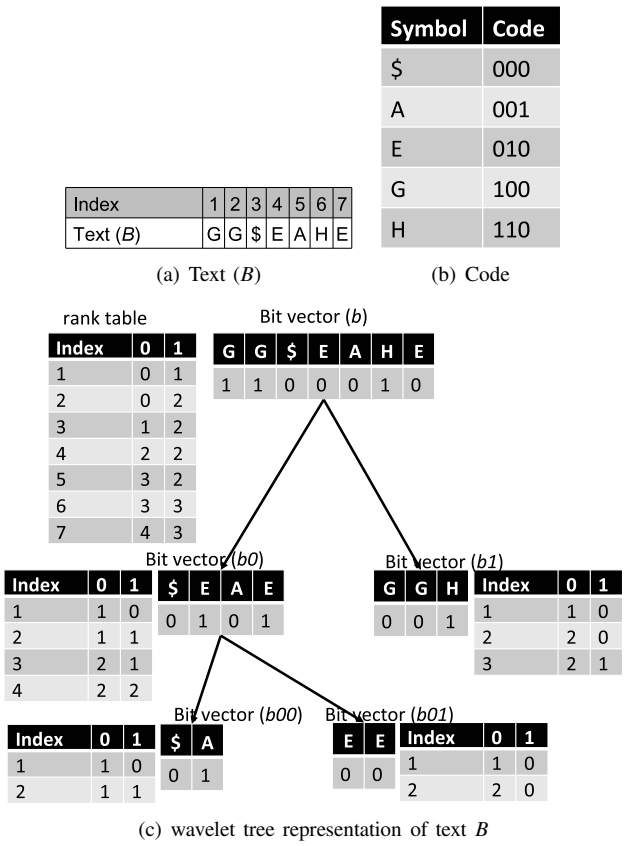
If this data structure is to be succinct, T_{bit} must be in the order of O(n). To satisfy this condition, the block size p must be greater than or equals to $log_2 n$. Moreover, the symbol count of a block must be done in a constant time irrespective of the size of n. That is, popcount(p) must be done in a constant time. Since there are popcount hardware that have constant computation time, constant processing time is achieved.

Since we use FPGA, we consider a memory size of 4GB. This condition is reasonable since many FPGA boards with high-end FPGAs contain this much of memory. Now let us calculate how much memory is required and how large is a block when we consider a 1GB input text file. We also consider each letter in the input file contain 8 bits (1 Byte) and there are 128 meaningful letters in the alphabet. Therefore, $n = 1GB/1B$ and $m = 128$. From Eq.3, when the total bits T_{bit} equals to 4GB, the block size $p = 1229$. As a result, one code word contains a header of 128×30 bits and a body of 1229×7 bits. That is 12443 bits. Therefore, if the word width of the memory is 512 bits (512 bits are accessed in one read), 25 memory reads are required to get one rank data value. After that, we have to perform the popcount function for 1229 symbols. As wee can see here, although we can store the data, accessing it and decoding it is very costly in terms of both time and area. Even a single memory access may take several cycles to complete,

25 memory reads per a *rank* data is not practical. Therefore, we need a better data structure.

2.3 Wavelet tree based data structure

As we saw above, the strategies relating to the binary sequences or small number of symbols cannot be applied directly to the data structures with many symbols. The wavelet tree proposed in [8] 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.4. For a given text B shown in Fig.4(a), a code is assigned to every symbol as shown in Fig.4(b). The construction of the wavelet tree start from the most significant bit (MSB) of the code. A bit vector b is created by using the MSB of each symbol in the text B as shown in Fig.4(c). 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 b are 0. The other group contains the symbols that their corresponding bits in b are 1. Then we assign bit vectors $b0$ and $b1$ 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. After the construction of the wavelet tree, we create *rank* tables for each bit vectors.



(a) Text (B) (b) Code

(c) wavelet tree representation of text B

Fig. 4: Construction of a wavelet tree

Fig.5 shows how to compute *rank* using wavelet tree. In this example, $rank_E(B, 4)$ is considered. Note that, B is the text shown in Fig.4(a). 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 $rank_0(b, 4)$. Then we come down to the second level of the wavelet tree and use the input vector $b0$, since "E" is included in the group that the MSB of "E" is zero. Then $rank_1(b0, 2)$ is calculated. Similarly, the calculation is done for all the levels in the wavelet tree as shown in Fig.5.

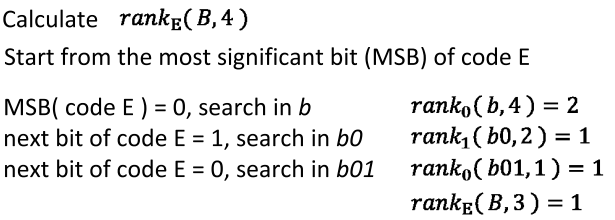


Fig. 5: Computation of $rank_E(B, 4)$ using wavelet tree

Although Fig.4(c) shows the *rank* table for the symbols "0" and "1", we just have to store the *rank* of only one symbol. The *rank* of the other symbol is derived by subtracting the *rank* of the known symbol from the index. For example, $rank_0(b, x) = x - rank_1(b, x)$ where b is the bit vector and x is the index. There are many *rank* tables in $log_2 m$ levels. However, in each level, the sum of all entries in all tables equals to the number of symbols in the reference text. Therefore, the total number of bits required ($T_{wavelet}$) is given by Eq.(4).

$$T_{wavelet} = \left\{ log_2 n \times \frac{n}{p} + n \right\} \times log_2 m \tag{4}$$

Using Eq.(4), we can obtain the block size for the example of 1GB input data. In this case, $p = 68$. Therefore, in one block we have a 30 bits large header and 68 bits large body. Therefore, a code word contains a total of 98 bits. This much of data can be accessed in one memory read. Note that, the wavelet tree has $log_2 m$ levels of bit vectors and we have to access a bit vector in each level. Therefore, in this example, a total of 7 memory reads are required to obtain one *rank*. This is a substantial reduction of the memory reads.

2.4 Proposed data structure for FPGAs

In this paper, we discuss a data structure that considers the hardware specification. We consider a memory model where each read access W bits from the memory. As shown in Fig.6, a code word consists of a header and a body. The size of the header is decided by the number of symbols n in the text. The size of the body is decided by the number of entries in a block of the *rank* table. Since the header requires $log_2 n$ bits, the body contains maximum of $W - log_2 n$ bits which should be equal to the number of entries in a block. Therefore, the block size $p = W - log_2 n$. The body size

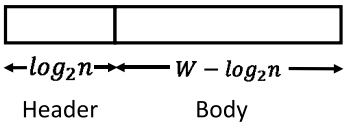


Fig. 6: A cord word of W bits long

could be further reduced by using byte-pair encoding (BPE). BPE [9] is a simple data compression method that the most common pair of consecutive bytes of data is replaced with a byte that is not been used already in the compressed data file. The same method could be applied for the bit array too and it is already used in text processing in [10]. In this case, we apply BPE for each cord word separately by using a common dictionary data. According to the experimental results using various text files. we found that 80% compression ratio could be achieved. The compression ratio is decided by the worst case. The required memory size T_{prop} is given by Eq.(5).

$$T_{prop} = \left\{ log_2 n \times \frac{0.8n}{W - log_2 n} + n \right\} \times log_2 m \qquad (5)$$

In practical cases, $W - log_2 n$ is much larger than $log_2 n$. Therefore, the storage is in the order of $O(n)$. The memory access is in the order of $O(log_2 m)$ which is independent of n. Therefore, we can say that this data structure is succinct.

3. FPGA architecture and evaluation

Fig.7 shows the overall architecture. It consists of a PE array and two DDR3 memories. The *rank* data of the text are stored in the DDR3 memory. Then the search queries are transfered to the DDR3. PEs process the search queries and find the search positions. Those data are written to a shared memory and later read by the host computer. The search queries can be sent in batches. After one batch is finished, another batch is transfered to the DDR3 memory. Therefore, we can process any number of search quarries while the quaries in a batch are processed in parallel by multiple PEs.

The structure of a PE is given in Fig.8. It consists of a 32-bit adder, a comparator and pipeline registers to perform the calculations explained in algorithm 1. The "ADD/SUB" unit in PE is used to calculate the suffix array interval given by Eqs.(1) and (2). The comparator and the control path do all the conditional branches in the "Search" procedure. New search queries are fed to the PEs after the old ones are searched. The output 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 search a query. 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.

The hardware module that decodes the *rank* data is shown in Fig.9. We extract only the required bits from the body of

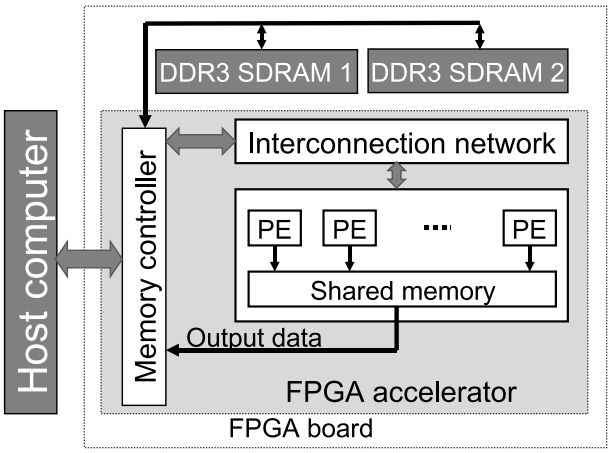


Fig. 7: Accelerator architecture

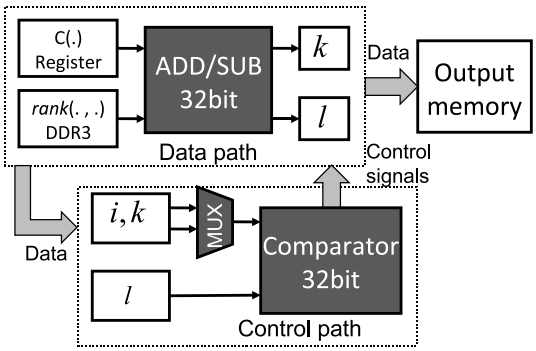


Fig. 8: Structure of a PE

a code word. For example, if we need only 16 bits starting from the LSB (least significant bit), we do the "bitwise AND" operation with a mask. In this case, the mask is 0xFFFF. Since the memory address corresponds to the *rank* 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 symbol count is added to the header. The decoder is pipelined, and an output is produced in every clock cycle after the pipeline is fully filled.

This decoding method has an added advantage of reducing

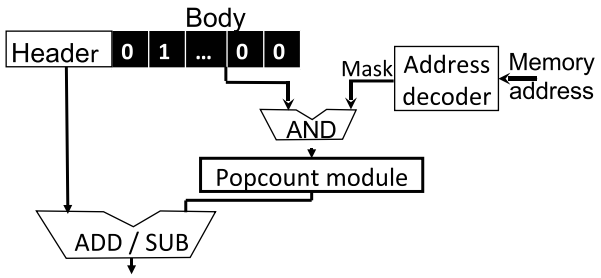


Fig. 9: Hardware decoder

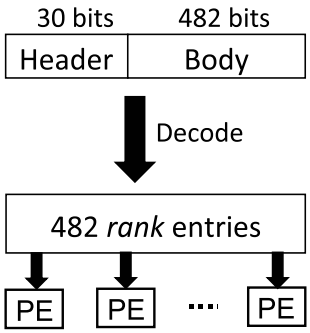


Fig. 10: Sharing of the decoded data

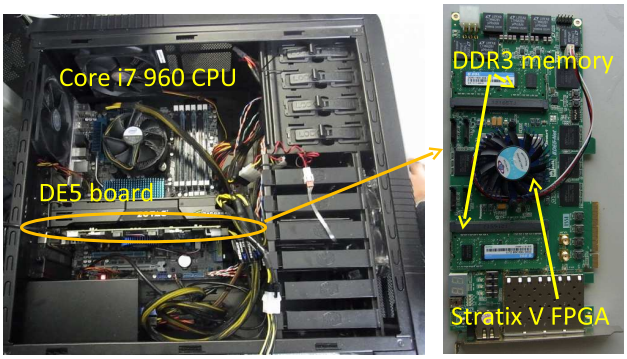


Fig. 11: Evaluation environment

the memory access. For example, let us consider the text file of 1GB large. The number of symbols in the file is given by n and $log_2 n = 30$, so that the header is 30 bits. If the word size of one memory read is 512 bits, the body contain 482 bits. Therefore, is it possible to obtain 482 *rank* data entries by decoding one code word as shown in Fig.10. Since we do parallel processing using multiple PEs, some of those *rank* data could be used in more than one PE. In such cases, the number of memory accesses are reduced.

For the evaluation, we used DE5 board [11] that contains "Altera 5SGXEA7N2F45C2 FPGA" and two 2GB DDR3-SDRAMs. The system shown in Fig.11 contains a core i7-960 CPU and a DE5 board connected through the PCI express port. The operating frequency of the accelerator is estimated to be 100MHz. We estimated that around 128 PEs can be implement on the FPGA.

Table 1 shows the size of the original text and encoded text. Usually, text data are in bytes so that the original text size is calculated by the actual file size. However, in the evaluation, we consider an alphabet of 128 characters. Therefore, one symbol requires only 7 bits. Compared to that, the FPGA implementation require a similar amount of bits with an increase of just 5.5%. In fact, the storage size is smaller than the original text file size. The reason for the small storage size is that we encode a large block of over 400 entries into a single code word. Therefore, the header size is very small. Since we use the wavelet tree representation, the header size is further reduced. However, using more bits in the body require a larger popcount function. That increases the hardware overhead. To reduce the hardware overhead, we have to reduce the number of bits in the body.

Table 1: Required data size

Original data size of the text (8bits per a symbol)	(7bits per a symbol)	Required storage size after encoding
1GB	0.88GB	0.92GB
2GB	1.75GB	1.84GB
4GB	3.50GB	3.69GB
4.3GB	3.76GB	3.97GB
5GB	4.38GB	4.61GB

4. Conclusion

This paper discusses an FPGA-based hardware architecture for text search that uses succinct data structures. We proposes a hardware-oriented data structure and its decoding method. The proposed architecture can be used in text searches up to 4.3GB large data. The storage space is just 5.5% larger than the original data size (7bits per symbol) and smaller than the input file size (1 byte per symbol).

Acknowledgment

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

References

[1] G. Jacobson, "Succinct static data structures. PhD thesis", Carnegie Mellon University, 1989.
[2] G. Jacobson, "Space-efficient static trees and graphs", 30th Annual Symposium on Foundations of Computer Science, pp.549-554, 1989.
[3] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm", Digital Equipment Corporation, Palo Alto, CA, Technical report 124, 1994.
[4] P. Ferragina and G. Manzini, "Opportunistic data structures with applications", Proc. of 41st Symp. on Foundations of Computer Science, pp.390-398, 2009.
[5] Heng Li and Richard Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform", Bioinfomatics, Vol.25, No.14, pp.1754-1760, 2009.
[6] H. M. Waidyasooriya, M. Hariyama and M. Kameyama, "Implementation of a custom hardware-accelerator for short-read mapping using Burrows-Wheeler alignment", Conf Proc IEEE Eng Med Biol Soc., pp.651-654, 2013.
[7] H. S. Warren, "Hacker's Delight (2^{nd} edition) - Chapter 5", 2012.
[8] 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.
[9] Philip Gage, "A New Algorithm for Data Compression", C/C++ Users Journal, 12(2), pp23-28, 1994.
[10] H. M. Waidyasooriya, D. Ono, M. Hariyama and M. Kameyama, "Efficient Data Transfer Scheme Using Word-Pair-Encoding-Based Compression for Large-Scale Text-Data Processing", Conf Proc IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), pp.639-642, 2014.
[11] http://www.altera.com/education/univ/materials/boards/de5/unv-de5-board.html.