String Matching in Hardware using the FM-Index

Edward Fernandez, Walid Najjar and Stefano Lonardi
Department of Computer Science & Engineering
University of California Riverside
Riverside, CA

Abstract—**String matching is a ubiquitous problem that arises in a wide range of applications in computing, e.g., packet routing, intrusion detection, web querying, and genome analysis. Due to its importance, dozens of algorithms and several data structures have been developed over the years. A recent breakthrough in this field is the FM-index, a data structure that synergistically combines the Burrows-Wheeler transform and the suffix array. In software, the FM-index allows searching (exact and approximate) in times comparable to the fastest known indices for large texts (suffix trees and suffix arrays), but has the additional advantage of being more space-efficient than those approaches. In this paper, we describe the first FPGA-based hardware implementation of the FM-index for exact pattern matching. We report experimental results on the problem of mapping short DNA sequences to a reference genome. We show that the throughput of the FM-index is significantly higher than the naïve (brute force) approach. Like the Bowtie software tool, the FM-index can abandon early the hardware matching. It outperforms Bowtie by two orders of magnitude*.**

I. INTRODUCTION

String matching is the problem of searching for patterns in a long text. It is a ubiquitous problem with a wide variety of applications: in network routing, prefixes of incoming packets are matched to a routing table to determine the next destination; in intrusion detection, signatures of known network attacks are matched to incoming packets to prevent malware from entering the system; in web querying, set of keywords entered by users have to be matched against million of webpages; in personal genomics, short patterns obtained from sequencing instruments have to be matched to the reference human genome.

Due to the importance of the matching problem, dozen of algorithms and several data structures have been developed since the seventies. Some of the algorithms are based on finite automata, for instance, the Aho-Corasick algorithm [16] or the Knuth-Morris-Pratt algorithm [17]. Other methods preprocess the text and build indices (e.g., hash tables or search trees) to allow faster searching. A recent breakthrough in this field is the FM-index, which is a compressed index that combines the properties of the suffix array [2] with the Burrows-Wheeler transform [1]. Software tools using this index are two orders of magnitude faster than tools relying on conventional indices like hash tables and variations thereof.

The Burrows-Wheeler transform (BWT) was initially introduced for data compression [1]. This invertible transform itself does not compress the text; instead it rearranges the order of the symbols to a form that is much easier to compress with classical encoders. The popular compression software bzip2 found on Linux/MacOS systems is based on the BWT. Years after the seminal paper [1] was published, evidence of the equivalence between the BWT and the suffix array [2] opened the road to the development of the FM-index [3,4].

The FM-index is an index that contains the BWT of the text to be searched in the form of a set of numerical arrays. The index allows searching for a pattern using binary search (in logarithmic time) instead of a linear time scan. Not only the FM-index enables string matching (exact and approximate) in times comparable to the best known indices for large texts (suffix trees and suffix arrays), but it has the additional advantage of being extremely space-efficient. To the best of our knowledge, the FM-index has been used only in the bioinformatics domain. Software tools such as Bowtie [12], BWA [13], and SOAP2 [14] have shown to be one to two orders of magnitude faster than older tools using classical indices like hash tables.

In this paper, we present the first FPGA-based hardware implementation of the FM-index. In [8] the authors describe the implementation of the BWT on an FPGA, i.e. given a text how to get its BWT in hardware. Our focus in this paper is the searching for a pattern on the BWT of a text. We obtain the BWT of a text in software and load it in memory on the FPGA. We then perform the search using our hardware implementation. The architecture is implemented on an FPGA using BRAMs as storage units. We compared the performance of our hardware implementation of the FM-index to a highly optimized hardware implementation of the naïve (brute force) approach. We synthesized both methods on a Xilinx Virtex 6 LX760, and exhausted all of the available resources on the FPGA. We defined the throughput as the number of character comparisons per second. Experimental results show that the FM-index has significantly higher throughput than the brute force. This measure of throughput does not, however, entirely reflect the real speed of the FM-index. Since the FM-index can abandon matching a pattern as soon as a mismatch is detected (our pipelined implementation of the naïve method cannot do this), the FM-index is expected to be a lot faster in an end-to-end application. We also compared our FPGA implementation of the FM-index to the software too Bowtie [12]. We show a two orders of magnitude speed-up when early abandon is accounted for.

The paper is organized as follows. In Section 2 we introduce data structures of the BWT. In Section 3 we describe

* This work has been supported in part by NSF Awards 0905509 and 0811416.

978-0-7695-4301-7 2011
U.S. Government Work Not Protected by U.S. Copyright
DOI 10.1109/FCCM.2011.55

carrying out string matching using the FM-index, and we discuss our hardware architecture and its implementation on the FPGA. In Section 4 we report on the throughput of the FM-index compared to the brute force approach and the Bowtie software tool [12]. In Section 5 we discuss works related to the BWT and the FM-index. In Section 6 we draw some concluding remarks and highlight possible direction for future research.

II. Data Structures of the BW Transform

In this section we build up the required data structures for the hardware implementation of the FM-Index searching. Given a text *Q* we denote by *BWT(Q)* its transform. The BWT of a string is generated in five steps:

1. Terminate the text Q with a unique character: "$".
2. Generate all rotations of the text.
3. Sort all the rotations.
4. Extract the last characters of all the entries of the sorted list.
5. Join the characters in the same order they appeared in the sorted list. The newly generated text is the BWT(Q).

Table 1 illustrates an example of deriving BWT(Q)[1]. Notice that characters to the left of the "$", in Table 1, form a suffix. A *suffix array* indicates the position of each possible suffix in the original string. Table 2 shows the suffix array representation of the text Q in Table 1. For example at index 5 the suffix value is "c$" and its position in the original string is 13. The suffix at index 8 is "gctaattaggtacc$" and its position is 0 since it is the whole string.

Table 1: Example of deriving the Burrows-Wheeler Transform of a text. The text is terminated by a "$" symbol.

Original String: GCTAATTAGGTACC$	
Rotations:	**Sorted Rotations:**
gctaattaggtacc$	$gctaattaggtac – C
ctaattaggtacc$g	aattaggtacc$gc – T
taattaggtacc$gc	acc$gctaattagg – T
aattaggtacc$gct	aggtacc$gctaat – T
attaggtacc$gcta	attaggtacc$gct – A
ttaggtacc$gctaa	c$gctaattaggta – C
taggtacc$gctaat	cc$gctaattaggt – A
aggtacc$gctaatt	ctaattaggtacc$ – G
ggtacc$gctaatta	gctaattaggtacc – $
gtacc$gctaattag	ggtacc$gctaatt – A
tacc$gctaattagg	gtacc$gctaatta – G
acc$gctaattaggt	taattaggtacc$g – C
cc$gctaattaggta	tacc$gctaattag – G
c$gctaattaggtac	taggtacc$gctaa – T
$gctaattaggtacc	ttaggtacc$gcta – A
Burrows-Wheeler Transform: CTTTACAG$AGCGTA	

[1] Throughout this paper we will use examples derived from DNA string matching because of the small alphabet size: A, G, C, and T.

Table 2: Example of deriving the suffix array of a text. The rightmost column is the suffix array of the text.

Original String: GCTAATTAGGTACC$		
Index	**Sorted Suffixes:**	**Suffix Array**
0	$	14
1	aattaggtacc$	3
2	acc$	11
3	aggtacc$	7
4	attaggtacc$	4
5	c$	13
6	cc$	12
7	ctaattaggtacc$	1
8	gctaattaggtacc$	0
9	ggtacc$	8
10	gtacc$	9
11	taattaggtacc$	2
12	tacc$	10
13	taggtacc$	6
14	ttaggtacc$	5

The equivalence of the BWT and the suffix array representation has been established in [3]. After generating the suffix array (Table 2) we sort the BWT(Q) (Figure 1) and generate the I and C tables. The sorted BWT(Q) is denoted as *SBWT(Q)*.

- **Table-I** – For every element x of the alphabet of Q, Table-I[x] = index of its first occurrence in SBWT(Q). For example, in Figure 1, A, C, G, and T appear at index 1, 5, 8, and 11 in *SBWT(Q)*.
- **Table-C** – For each index n in BWT(Q) and for each character x in the alphabet Table-C[n,x] = number of occurrences of x in BWT(Q) in the range [0, n-1]. As an example, consider index *n = 10* in the C-Table in Figure 1, column A has a value of 3 because there are three occurrences of A in the range *n = 0* to *n = 9.*

III. Pattern searching using the FM-index in Hardware

The FM-index is a pattern searching technique that operates on the BWT. The FM-index consists of two pointers: *top* and *bottom*. We discuss the *top* and *bottom* pointers in the context of suffix arrays because of the inherent equivalence of BWT and suffix arrays. The indices between the *top* and *bottom* pointers are all the suffix locations where a pattern occurs on the text. *Top* points to an index of the suffix array element where a specific pattern is first located. The *bottom* pointer limits where the pattern can be last found. If *bottom* points to an index that is less than or equal to an index pointed by *top,* then the pattern does not occur on the text.

$	A	A	A	A	C	C	C	G	G	G	T	T	T	T
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

SBWT(Q) for Q = GCTAATTAGGTACC$

I-table

A	C	G	T
1	5	8	11

C-table

Index	BWT(Q)	A	C	G	T
0	C	0	0	0	0
1	T	0	1	0	0
2	T	0	1	0	1
3	T	0	1	0	2
4	A	0	1	0	3
5	C	1	1	0	3
6	A	1	2	0	3
7	G	2	2	0	3
8	$	2	2	1	3
9	A	2	2	1	3
10	G	3	2	1	3
11	C	3	2	2	3
12	G	3	3	2	3
13	T	3	3	3	3
14	A	3	3	3	4
15	Total	4	3	3	4

Figure 1: I-table stores the first occurrence of each character on the sorted BWT(Q). The C-table stores the count of each character on a previous location. The leftmost column of the C-table is the indices of the suffix array.

A. *Searching and Locating a Pattern*

Pattern searching using the FM-index starts with initializing the *top* and *bottom* pointers to the first and last indices of the C-table respectively. To search for a pattern, we process one character at a time, beginning with the last character of the pattern. The *top* and *bottom* pointers move to different suffix array indices according to the current character processed and the current index where the *top* and *bottom* pointers are indexing. To compute the new location of the pointers, we follow Equation 1 for the *top* and *bottom* pointer respectively.

$$\text{Top}_{new} = \text{C-table}[n, \text{Top}_{current}] + \text{I-table}[n]$$
$$\text{Bottom}_{new} = \text{C-table}[n, \text{Bottom}_{current}] + \text{I-table}[n]$$

Equation 1 – Updating of *Top* and *Bottom* pointers using the I-table and C-table..

Text: GCTAATTAGGTACC
Pattern: TAGG

1st iteration: n = G

$$\text{Top}_{new} = C_G(\text{Top}_{current}) + I(G) = 0 + 8 = 8$$
$$\text{Bot}_{new} = C_G(\text{Bot}_{current}) + I(G) = 3 + 8 = 11$$

2nd iteration: n = G

$$\text{Top}_{new} = C_G(\text{Top}_{current}) + I(G) = 1 + 8 = 9$$
$$\text{Bot}_{new} = C_G(\text{Bot}_{current}) + I(G) = 2 + 8 = 10$$

3rd iteration: n = A

$$\text{Top}_{new} = C_A(\text{Top}_{current}) + I(A) = 2 + 1 = 3$$
$$\text{Bot}_{new} = C_A(\text{Bot}_{current}) + I(A) = 3 + 1 = 4$$

4th iteration: n = T

$$\text{Top}_{new} = C_T(\text{Top}_{current}) + I(T) = 2 + 11 = 13$$
$$\text{Bot}_{new} = C_T(\text{Bot}_{current}) + I(T) = 3 + 11 = 14$$

Figure 2: Example of searching the pattern "TAGG" on the string "GCTAATTAGGTACC" using the FM-index. After the 4th iteration the pattern is found because the index of the top pointer is less than the bottom pointer.

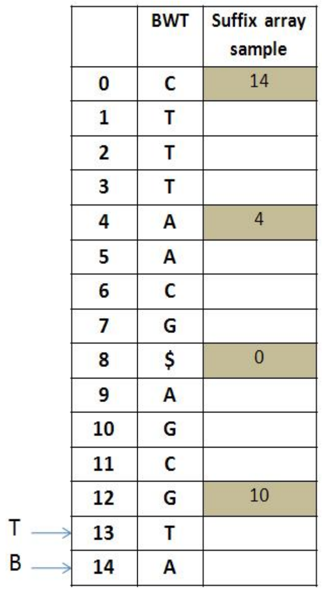
Figure 2 shows an example of searching the pattern "TAGG" on the example string in Table 1. We initialize the *top* and *bottom* pointers to 0 and 15 respectively. We begin with the last character, *G,* of the pattern. We then apply Equation 1 four times corresponding to four characters of the pattern. After the fourth iteration, the *top* and *bottom* pointers are at index 13 and 14 respectively. Since the index of the *top* pointer is less than the index of the *bottom* pointer, the pattern TAGG occurs on the string.

Three methods are described in [3] to identify the location of the search pattern once its existence has been determined:

- Use characters from BWT(Q) of the text instead of the pattern to trace back the end of the text using Equation 1. The number of steps Equation 1 is applied is the location where the pattern occurs on the text. Main advantage of this method is that no additional storage is required. However, its drawback is the potentially large number of steps that could lengthen the search time.
- Store the suffix array elements that indicate the positions where the suffixes are located. Upon locating the existence the pattern, the suffix array element that corresponds to that suffix array index is the location of the pattern. The main advantage of this approach is the immediate availability of pattern locations. However, its disadvantage is the storage area of all suffix array elements could be huge for very large texts.
- Store only samples of the suffix array and trace back until we reached a sampled suffix array element. We add the number of steps Equation 1 is used during the trace back and the location indicated by the sampled suffix

array element. The sum is the location of the pattern on the text.

The last approach is clearly the best approach which combines the advantages of the first two methods and limits their disadvantages. We implemented the last method in our hardware to locate the patterns in the text.



	BWT	Suffix array sample
0	C	14
1	T	
2	T	
3	T	
4	A	4
5	A	
6	C	
7	G	
8	$	0
9	A	
10	G	
11	C	
12	G	10
T → 13	T	
B → 14	A	

Is Top = 13 sampled? NO
Count = 1
1st iteration: n = T
$Top_{new} = C_T(Top_{current}) + I(T)$
$= 11 + 3 = 14$

Is Top = 14 sampled? NO
Count = 2
2nd iteration: n = A
$Top_{new} = C_A(Top_{current}) + I(A)$
$= 3 + 1 = 4$

Is Top = 4 sampled? YES
Location = Suffix_Array(T) + Count
= 4 + 2 = 6

Figure 3: Example of locating the pattern "TAGG" on the string "GCTAATTAGGTACC" using the third approach. After the 2nd iteration the pattern is located on position 6 of the text.

To continue our example, we locate where the pattern occurs in the text shown in Figure 3. The figure shows the sampled suffix array elements and how the pattern "TAGG" is located on the sample string. After identifying that the pattern appears in the text, we check if the *top* points to a sampled suffix array element. If not, we utilize Equation 1 for the *top* pointer using the BWT character as the symbol for the C and I tables. We also increment a counter that tracks the number of steps performed. We continuously use Equation 1 and increment the counter until the *top* pointer index a sampled suffix array element. Figure 3 shows that the pattern "TAGG" is located after the end of the second iteration where the *top* pointer indexes a sampled suffix array element.

Suppose we search for the pattern "CCGA" on the example string shown in Table 1. Beginning from the last character, the *top* and *bottom* pointers move until the second iteration where the pointers index the same suffix array element. This pattern search is shown in Figure 4. In this example, we show that the FM-index does not have to look at all the characters of the pattern when it identifies the pattern does not appear on the text from its preliminary search.

Text: GCTAATTAGGTACC
Pattern: CCGA

1st iteration: n = A
$Top_{new} = C_G(Top_{current}) + I(G)$
$= 0 + 1 = 1$
$Bot_{new} = C_G(Bot_{current}) + I(G)$
$= 4 + 1 = 5$

2rd iteration: n = G
$Top_{new} = C_A(Top_{current}) + I(A)$
$= 0 + 8 = 8$
$Bot_{new} = C_A(Bot_{current}) + I(A)$
$= 0 + 8 = 8$

Figure 4: Example of searching the pattern "CCGA" on the string "GCTAATTAGGTACC" using the FM-index. After the 2nd iteration the pattern is not found because the top and bottom pointers index the same suffix array location.

B. Architecture

The pattern is placed on a shift register, where the last character is fed as input to the architecture shown in Figure 5. A text is placed on the FPGA using its memory resources. Two memory banks are used to store the C and I tables. The *top* and *bottom* pointers address the C-table. The outputs of the memory bank of the C-table are four values referring to the columns of A, C, G, and T. These values are inputs to a multiplexor and the current character symbol, *n,* is used as the selector. The same symbol *n* also addresses the I-table.

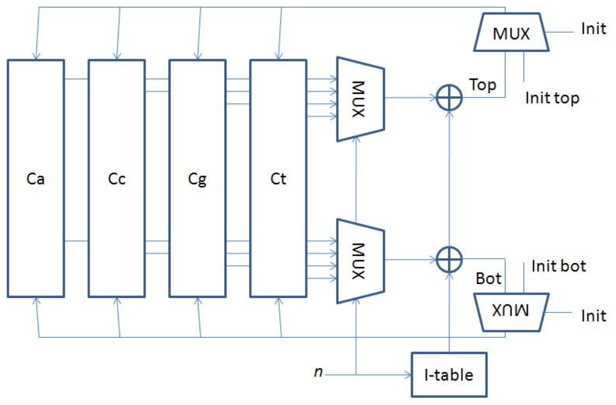


Figure 5: Block diagram for searching a pattern on a text using the FM-index.

The output of the I-table and the multiplexors of the C-table are added to compute the new values of *top* and *bottom*. The new pointers are then used as addresses in the next iteration. An additional multiplexor is used for initializing the *top* and *bottom* pointers as addresses of the C-table. Figure 5 shows the block diagram showing the architecture for searching using the FM-index.

The number of bits required to implement the C-table is the product of the text size (in number of characters), the size of the alphabet and the logarithm of the text size. The text size corresponds to the number of rows of the C-table, the size of the alphabet corresponds to the number of columns, and the logarithm of the text section length corresponds to the number of bits per element of each column entry of the C-table. The number of bits required to implement the I-table is product of the logarithm of the text size and the size of the alphabet.

The block diagram in Figure 5 is modified to support locating a pattern that appears in the text (Figure 6). Additional storage is required for the sampled suffix array elements, flag bits and the BWT characters of the text section length. The flags indicate if a character is sampled. The BWT(Q) characters (*BWT_char* in Figure 6) are used to address the I-table and to select the data from the C-table when the occurrence of the pattern in the text is established. Lastly, a finite state machine is required to facilitate the transfer of data from the different storage structures (Figure 7).

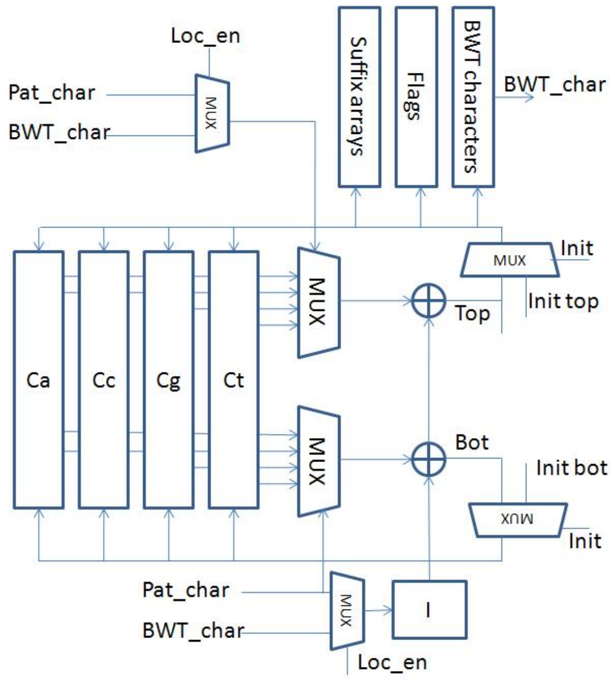


Figure 6: Block diagram for searching and locating a pattern on a text using the FM-index.

Figure 6 shows the modified block diagram to support locating the pattern on the text. The *top* pointer addresses the additional memory of the sampled suffix array elements, flags, and BWT(Q) characters. Additional multiplexors are needed to select whether a search pattern character or a BWT(Q) character is used for the C and I tables. The selector for these multiplexors is the *Loc_en* control signal from the finite state machine.

Figure 7 shows the diagram of the finite state machine controller. It has four states: The *Init* state initializes the *top* and *bottom* pointers. This state asserts the *init* signal used as multiplexor select for initialization. The *Search* phase is where the *top* and *bottom* pointers move in finding the pattern on the text. A counter is set on the *Search* phase to execute a number of times equal to the length of the pattern. If the *bottom* pointer addresses a location that is equal or less that the *top* pointer at any time instant, then the pattern does not occur on the text and the state machine goes back to the *Init* phase. If the *bottom* pointer addresses a location that is greater than the *top* pointer at the end of the count, then the next state is the *Check Flag* indicating that the pattern appears on the text. This state asserts the *loc_en* signal so that the C and I tables use the BWT(Q) characters as the multiplexor instead of pattern characters. The *Check flag* state checks if a character position is a sampled suffix array element. If the position is sampled, then it returns to the *Init* phase otherwise it moves to the *Increment* state. The *Increment* state adds one to a counter that indicates the number of steps the trace back occurred. After the *Increment* state, it returns again to the *Check Flag* state.

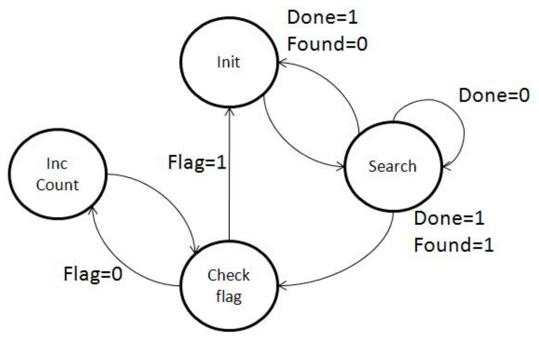


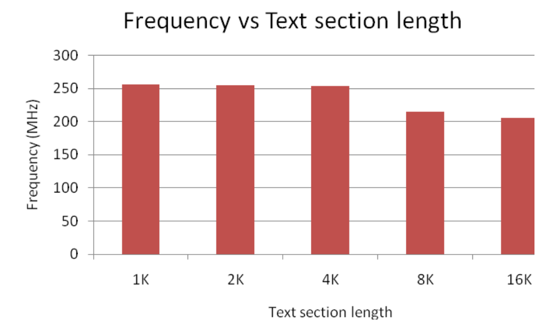
Figure 7: Controller used for architecture.

IV. Implementation and Evaluation

A. Hardware Implementation

We performed experiments to determine whether there are any advantages to splitting one large text, with a single set of C and I tables, into multiple text sections with one set of C and I tables for each section. In this experiment, we start with a text of 262,144 characters corresponding to 16 modules of 16K characters each, or 32 modules of 8K characters, or 64 modules of 4K characters etc. We sample the suffix array every 32 elements and the search pattern length is 36 characters. A text section corresponds to one hardware module. The length of the text in one module ranges from 1K to 16K characters.

We synthesized with place and route our implementations on a Xilinx Virtex 6 (XC6VLX760) FPGA. Figure 8 shows the operating frequency and slices utilized (as % of total slices) for increasing text section length per module. Increasing the section size, from 1K to 16K characters, decrease the operating frequency by 20%. This is due to larger adder sizes and larger module circuits. The percentage of slices required for these decreases from 25% to 2.5%. This is because more logic is required for processing multiple FM-indices compared to a single unit for one long text section length. Looking at both graphs, it is better to implement the FM-index on an FPGA using longer text lengths.



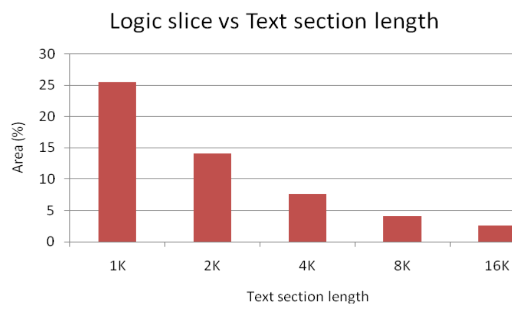


Figure 8: Operating frequency and area versus text section length. Note that the frequency decreases by only 20% as the text section size increases from 1K to 16K. Logic area decreases exponentially as we increase the text subsection length.

B. Maximizing Resource Utilization

The objective of our implementation is to maximize the throughput of the design. In the FM-index algorithm, as in BWT, the search for the pattern is applied to the whole text. Therefore the larger the text that can be fitted on the FPGA the higher the effective throughput. We have also attempted to maximize the clock frequency by pipelining the implementations.

Table 3: Resource utilization and Frequency of FM-index

Modules		BRAM-18 Utilization (%)	Slices (%)	Freq (MHz)	Throughput (10^{12} CC/s)
Size	No. of				
FM-1K	352	1,430 (99.0%)	35	251	84.31
FM-2K	168	1,366 (94.8%)	19	244	81.10
FM-4K	126	1,422 (98.7%)	13	241	112.40
FM-8K	58	1,422 (98.7%)	8	213	100.13
FM-16K	27	1,432 (99.4%)	4	201	88.55

We experimentally evaluate the tradeoffs between smaller text sections having larger clock frequencies and larger text sections with smaller clock frequencies. Table 3 shows the resource utilization, operating frequencies and throughput of the five FM-index implementations with a pattern length of 36 on the Virtex 6 LX760, which has a total of 1,440 BRAM-18. Note that we have used a mix of BRAM-18 and BRAM-36 but are reporting the total number in BRAM-18.

Table 4: Overhead due to text overlap for search string length 36

Module Size	Total Characters	Overlap Characters	Overlap Overhead
FM-1K	360448	24570	6.8%
FM-2K	344064	11690	3.4%
FM-4K	475136	8750	1.8%
FM-8K	475136	3990	0.84%
FM-16K	442368	1820	0.41%

Having the text partitioned across multiple modules requires that segments of the text be replicated in contiguous modules. This introduces some area overhead as shown in Table 4 for a search string length of 36. More overlaps occur for longer pattern lengths because we need to repeat longer end sections of the text. These overlaps are deducted from the total text characters placed on the FPGA, which reduces the effective length of the text. As can be see in Table 3, the FM-4K achieves the highest throughput and will be used in the comparison with the brute force approach and software implementations.

V. PERFORMANCE EVALUATION

In this section we evaluate the performance of the FM-index, first by comparing it to a "naïve" or brute force implementation [11], and second by evaluating the expected number of character matches that would be performed for each search pattern.

A. Comparison to the Brute Force Approach

We compared the FM-index implementations to a brute force approach [11] that uses direct character comparisons between a pattern and a text. It takes a set of search strings (patterns) storing them in registers on the FPGA. The text is streamed into shift registers and compared to the pattern registers. A signal is asserted when a match occurs at a specific location. To maximize parallelism, the text is divided into a multiple concurrent streams. It is also implemented on a Xilinx Virtex 6 LX760 FPGA with search string lengths of 36. We also utilized 92% of logic slices so that we could place the most number of patterns.

Table 5: Character comparisons (CC) per second for FM-index and Brute Force implementations.

	Freq (MHz)	CC/cycle	Throughput (10^{12} CC/s)
FM-4K	241	466,386	112.40
Brute force	342	200,880	68.75

Table 5 shows the total number of character comparisons performed of the FM-index and brute force respectively. For the FM-index we have used the FM-4K implementation as it has the highest effective throughput. For the FM-index we compute the throughput as the product of the text size and the operating clock frequency.

B. *Expected Number of Character Comparisons*

In the FM-index algorithm each character of the search pattern is matched against the whole text section at once. These matches are done sequentially. However, at the first mismatch the search is terminated ("early abandon") and a new search pattern is initiated. Therefore, given a pattern of length *p*, unless there is match, the number of character comparisons is $c < p$. In the previous section we derived the raw throughput in terms of character comparisons the hardware can deliver every cycle. However, this throughput does not reflect the ability of the algorithm to abandon the search and hence the expected run time. In [15] the authors prove that the *expected shared prefix* between any two substrings in a random text, *e*, is $e = log(n)/E$, where *E* is the entropy of the random source generating the text and *n* is the text size. Using the DNA alphabet and the text section lengths used in this paper, the expected length of the common prefix ranges from 5 (n = 1K) to 7 (n = 16K), as shown in Table 7. One can expect than very soon after this 5-7 matches to see a mismatch, and therefore abandon the search.

Table 5: Expected length of shared suffixes between two substrings for a specific text length.

Text section length	Expected length of shared suffix (*e*)
1K	5
2K	5.5
4K	6
8K	6.5
16K	7

C. *Comparison to Software*

We compare our FM-index hardware implementation to the Bowtie [12] software tool used for mapping DNA sequences to a reference genome. In this experiment we executed the software tool using only one core of a Quad-core Intel Xeon at 2.5GHz with 12 MB cache. We measured the execution time of searching one thousand DNA sequences with lengths of 36, 72 and 108 on a section of the E-coli genome with a length of 490,000 characters. We compare the total execution times of the software tool to the execution time of the hardware implementation based on throughput on Table 8.

Table 6: Execution times of Bowtie and FPGA Implementation with the number of matching DNA sequences on the E-coli genome in percent.

Pattern Length	Software run time (ms)	FM-index on FPGA		
		Run time (ms)	Sequences occurring on section (%)	Speed up
36	11.5	0.0586	6	196
72	7.5	0.0602	3	124
108	7.5	0.0565	1	133

The values of the expected length of the shared suffix (or prefix) reported in Table 5 assume a random text. To account for the non-random nature of most real text and take a pessimistic approach to the evaluation of the FM-index on FPGAs, we have used have doubled these values. We computed separately the execution times for non-existing (no match) and existing patterns (a match is found). The cost in cycles in searching for a non-existing pattern is twice the expected shared suffix length for a 4K module shown in Table 5. For existing patterns the full pattern length is the cost in cycles for the search. Table 6 shows that the FPGA achieved a speed up ranging from 124 to 196 compared to software execution times.

One of the limitations of the FM-index hardware implementation is its dependence on the size of the memory available on the FPGA. In fact, the throughput that is achieved with our implementation is directly proportional to the memory size and the clock frequency. It is to be expected therefore that with future FPGA technologies and designs that the throughput will rise very rapidly.

VI. Related Works

The Burrows Wheeler transform is an algorithm that converts an input data to another form that is easily compressible and reversible to its original form [1]. The transform is done by listing all the cyclic permutations of the string, sorting that list and extracting the last characters of each entry and putting them together on the same order it appeared on the sorted list. The transform can be easily compressed because the sorted list has the same elements contiguous to each other.

Suffix arrays are essentially related to the Burrows-Wheeler transform of a string. A suffix array is a data structure that lists all the locations of the suffixes of a string that is sorted lexicographically [2]. It allows fast string matching because of the ordered arrangement of the suffixes on the list. However, the disadvantage of suffix arrays is the time and space needed to construct the data structure.

The equivalence of the Burrows-Wheeler transform and suffix arrays rely on the sorting of rotations of the string terminated by a special unique character. Because of this, the Burrows-Wheeler transform essentially becomes a compressed version of the suffix array [3][4].

Since the introduction of the FM-index, it has been a subject for improvements and optimizations. One area that the algorithm could be improved is to make it independent of the number of character symbols. Huffman compression could be first applied on the text resulting to a new text where the Burrows-Wheeler transform could be performed. The result is a text independent of the number of character symbols. [5]

The FM-index could also be improved in terms of its implementation on a host system. One optimization is to split the text into fixed block sizes and indexing each block separately [6]. The section of the text in each block overlaps each other so that the entirety of the text could still be searched. Another optimization is executing pattern searching using the FM-index on multiple cores [7]. Multiple cores could operate on different memory blocks containing contiguous text sections to achieve parallelism and higher throughput.

The Burrows-Wheeler transform of a string has also been a focus of hardware implementations where the central concern is the sorting of suffixes. An architecture composed of comparators, registers, shifters and multiplexors was described to perform sorting of the suffixes [8]. It improves the weavesort approach where shift left, shift right, and swap are performed on registers based on comparisons of register contents [9]. Another hardware implementation is using a bitonic sorting algorithm [10] where the comparisons of the list entries are not data dependent making it suitable for hardware implementations.

VII. Conclusion and Future reasearch

This study discusses and shows in detail the first implementation of the FM-index in hardware. It is compared to the brute force approach and it is shown that the FM-index has a higher effective throughput than the brute force. This is due to the higher number of character comparisons per cycle performed by the FM-index even though it operates on a lower operating.

Furthermore, the FM-index does not need to perform all character comparisons compared to the brute force approach. As soon as one character from the search pattern is mismatched the search can be terminated (early abandon) and a new one initiated, hence reducing the total search time.

Comparison to the Bowtie [12] software tool shows a two orders of magnitude speed-up of the FM-index on FGPA when accounting for early abandon property of this algorithm.

Future directions of this research immediately point to improving the operating frequency that reduces the throughput of the FM-index. Furthermore, we could also improve how data is stored in the C-table, because of the existence of redundant data. We could store a longer text section length on the available resources of the FPGA by removing this redundant data that will result to a higher throughput.

The FM-index discussed in this paper is only limited to exact string matching. We could further extend the FM-index implementation to approximate string matching where mismatches between pattern and text is allowed.

References

[1] M. Burrows and D.J. Wheeler, “A Block-sorting Lossless Data Compression Algorithm,” SRC Research Report, May, 1994.

[2] U. Manber and G. Myers, “Suffix Arrays: A New method for On-line String Searches,” SIAM Journal of Computing, pp. 935-948. 1993.

[3] P. Ferrragina and G. Manzini, “Opportunistic Data Structures with Application” In Proceeding of 41st IEEE Symposium on Foundations of Computer Science, pp. 390-398, 2000.

[4] P. Ferrragina and G. Manzini, “An Experimental Study of an Opportunistic Index,” In Proceeding of 12th ACM-SIAM Symposium on Discrete Algorithms, pp. 269-278, 2001.

[5] S. Grabowski, G. Navarro, R. Przywarski, A. Salinger and V. Makinen, “A Simple Alphabet-Independent FM-Index,”International Journal of Foundations of Computer Science, Vol. 17, No. 6, pp. 1365-1384, 2006.

[6] D. Zhang, Y. Zhang, and J. Chen, “Efficient Construction of FM-index using overlapping block processing for large scale text,” In Proceedings of the 29th European Conference on IR Research, pp. 113-123, 2007.

[7] D. Zhang, Y. Zhang, S. Liu, and X. Huang, “Parallelization of the FM-Index,” In Proceedings of 10th IEEE International Conference on High Performance Computing and Communications, 2008.

[8] J. Martinez, R. Cumplido, and C. Feregrino, “An FPGA-based Parallel Sorting Architecture for the Burrows-Wheeler Transform,” In Proceedings of International Conference on Reconfigurable Computing and FPGAs”, 2005.

[9] A. Mukherjee, N.Motgi, J. Becker, A. Friebe, C. Haberman, and M. Glesner, “Prototyping of Efficient Hardware Algorithm for Data Compression in Future Communication Systems,” In Proceedings of 12th IEEE Workshop on Rapid System Prototyping, pp. 58, 2001.

[10] P. Szecowka and T. Mandrysz, “Towards Hardware Implementation of BZIP2 Data Compression Algorithm,” In Proceedings of Mixed Design of Integrated Circuits and Systems, pp. 337-340, 2009.

[11] E. Fernandez, W. Najjar, E. Harris and S. Lonardi, “Exploration of Short Reads Genome Mapping in Hardware, In Proceeding of 20th International Conference on Field Programmable Logic and Application, 2010.

[12] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg, “Ultrafast and Memory-Efficient Alignment of Short DNA sequences to the Human Genome,” Genome Biology, 2009.

[13] H. Li and R. Durbin, “Fast and Accurate Short Read Alignment with Burrows-Wheeler Transforms,” Bioinformatics, 2009.

[14] R. Li, C. Yu, Y. Li, T. Lam, S. Yiu, K. Kristiansen, and J. Wang, “SOAP2: An Improved Ultrafast Tool for Short Read Alignment,” Bioinformatics, 2009.

[15] A. Apostolico and W. Szpankowski, Self-alignments in words and their applications, Journal of Algorithms, Volume 13, Issue 3, September 1992.

[16] A. Aho and M. Corasick. Efficient String Matching: An Aid to Bibliographic Search. Communication of the ACM, June Vol. 18, No. 6, pp330-340, 1975.

[17] D. Knuth, J. Morris, and V. Pratt, “Fast Pattern Matching in Strings,” SIAM Journal of Computing, 1977.