Bit-Level Processor Array Architecture for Flexible String Matching

Panagiotis D. Michailidis and Konstantinos G. Margaritis

Parallel and Distributed Processing Laboratory
Department of Applied Informatics, University of Macedonia
54006 Thessaloniki, Greece

Abstract. In this paper, we present a linear processor array architecture for flexible string matching. Initially, a bit-level sequential algorithm is discussed which consists of two phases, i.e. preprocessing and searching. Then, starting from the data-dependence graph of the searching phase a processor array architecture is derived. Further, the preprocessing phase is also accommodated onto the same processor array design.

1 Introduction

String matching is used in many applications including text processing, information retrieval, computational biology and pattern recognition. The basic string matching problem can be defined as follows: Let a given alphabet (a finite character set) Σ, a short pattern string $p = p_1...p_m$ of length m and a large text string $t = t_1...t_n$ of length n, in a alphabet Σ of size $|\Sigma|$, where $m, n > 0$ and $m \leq n$, find all occurrences of a pattern in a text. Recent surveys and experimental results of well known sequential algorithms for simple string matching can be found in [16].

The basic problem can be extended to include more flexible patterns, including patterns with don't care symbol, patterns with a complement symbol and patterns with class symbol. Some recent publications, which also provide surveys of previous work are [1–3, 6, 9, 11, 17, 22, 23]. Therein sequential and/or theoretical parallel algorithms are proposed for the solution of several aspects of the flexible string matching problem.

Recent advances in Very Large Scale Integration (VLSI) technology have made possible the development of application specific processor arrays for complex and computationally intensive problems. The characteristics of parallelism, concurrency, pipelining, modularity and regularity have become standard in VLSI designs. Similar processor arrays to string matching problems have been proposed by several researchers [5, 8, 13, 19, 15, 12]. Some surveys on processor arrays and architectures for string matching and related problems can be found in [4, 18, 7]. In [5] a VLSI architecture for simple string matching has been proposed. It allows for the concurrent pipelining of both text and search pattern through bidirectional data flow. This architecture has also been used for approximate string matching [8, 13] by means of a parallelizing a dynamic programming algorithm but it exhibits very limited flexibility due to the encoding scheme used. A similar approach is pursued in [19] using again a dynamic programming approach, but an improved encoding scheme and limited communication and control overheads.

However, the bidirectional data flow imposes low processor utilization (approximately 50%) and increased computation time. An improvement in the time complexity of the bidirectional data flow architectures has been proposed in [15], which reduces the computation time to approximately n steps by partitioning the problem into two subproblems which are solved concurrently in the same array. Recently, some researchers [20] and [21] presented Field Programmable Gate Arrays (FPGAs) implementations for simple string matching and regular expression matching respectively.

In this paper, the processor array implementation of a simple flexible string matching algorithm [1, 22] is discussed in the context of the systolic paradigm for VLSI computation [10].

2 A Bit-Level Algorithm for Flexible String Matching

A bit-level string matching algorithm is now presented which is derived from [1, 22]. In general, a string matching algorithm consists of two phases: the preprocessing and the searching phase. The preprocessing phase consists of gathering some information about the pattern, which can be used for fast implementation in the searching phase. On the other hand, the searching phase is based on the simulation of a non-deterministic finite automaton (NFA) built from the pattern and using the text as input in order to find all occurrences of the pattern in the text.

2.1 Preprocessing Phase

During the preprocessing phase the string p is encoded onto a bit-level memory map R of $(m \times |\Sigma|)$ bits, where $|\Sigma|$ is the size of the alphabet. The memory map can be seen as a two dimensional bit-level array where each row corresponds to a character of the pattern string and each column to a character of the alphabet. Therefore, column R_j^T, for $1 \leq j \leq |\Sigma|$, holds the information of the j-th character of the alphabet, which will be denoted as σ_j. The column R_j^T can be seen as a bit-level vector of m bits where the i-th bit, for $1 \leq i \leq m$, i.e. $R_{i,j}$ holds information concerning the j-th character σ_j and the i-th position of the search string. The basic information that can be recorded is whether the j-th character of the alphabet is the i-th character of the search string, that is whether $p_i = \sigma_j$. The preprocessing phase constructs the memory map R and the algorithm can be expressed in terms of the following piece of pseudocode.

```
for j=1 to |Σ| do
    for i=1 to m do
        R_{i,j} ← 1
        if p_i = σ_j then R_{i,j} ← 0
```

Taking as reference alphabet the UNICODE character set it is straightforward to calculate the memory requirements of the bit map R. Thus $|\Sigma| = 64K$ and therefore the memory map of an m character search pattern requires $16m$ Kbytes. The overall space requirements are $\lceil m|\Sigma|/16 \rceil$ bytes, assuming that a character is encoded in two bytes. Further, the preprocessing phase can be performed in approximately $O(m|\Sigma|)$ steps.

The algorithm performance can be improved if a specialised addressing is introduced such that a character is mapped directly to the appropriate column of the memory map. Such an addressing can take the form of a mapping function of a character ch of an alphabet Σ, $map(ch, \Sigma)$, returning the column number of the memory map.

An example for the construction of the memory map R is now given. The example alphabet Σ is {a,b,c,d,e} with $|\Sigma|$=5. The example pattern string p is the string "abba" with m=4. The bit-level memory map R is given in Table 1.

R	a	b	c	d	e
a	0	1	1	1	1
b	1	0	1	1	1
b	1	0	1	1	1
a	0	1	1	1	1

Table 1. Bit-level map R

2.2 Searching Phase

Consider the non-deterministic finite automata (NFA) for pattern of length m=4 without errors, shown in Figure 1. Every column represents matching the pattern up to a given position. The self-loop at the initial state allows us to consider any character as a potential starting point of a match. This NFA has $m+1$ states. We assign number i to the state at column i, $0 \leq i \leq m$. Initially, the active state is the column 0.

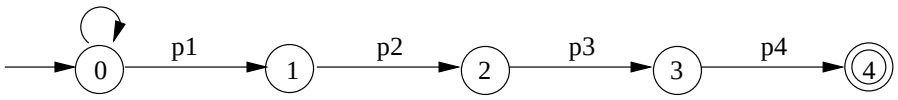

Fig. 1. NFA for the exact string matching

We have a vector F^0 corresponding to the NFA automaton. F_i^0 is 0 if state i is active and 1 otherwise. The vector changes as each character of the text is read. The new values of vector $F_i'^0$, $0 \leq i \leq m$ after we read a new text character t_j, $1 \leq j \leq n$, are computed as follows [1]:

for i=0 to m do
    $F_i^0 \leftarrow 1$
for i=0 to 0 do
    $F_i^0 \leftarrow 0$
for j=1 to n do
    $c \leftarrow map(t_j, \Sigma)$
    for i=1 to m do

$F'^0_i \leftarrow F^0_{i-1}$ OR $R_{i,c}$
for i=0 to m do
$F^0_i \leftarrow F'^0_i$

The first term of the formula F'^0 represents matching. If F'^0_m=0 then there is an exact occurrence of pattern in the text. Finally, this NFA simulation requires $O(mn)$ steps. Table 2 shows the vector F'^0.

F'^0		e	a	b	c	d	b	b	a	b	b	a	c	d
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	1	0	1	1	1	1	1	0	1	1	0	1	1
b	1	1	1	0	1	1	1	1	1	0	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	0	1	1	1
a	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1

Table 2. Vector F' to search the pattern p="abba" in text t="eabcdbbabbacd". Bold entries indicate matching text positions.

2.3 Extensions

In this subsection, we discuss some extensions that can be support the previous search algorithm.

Limited Expressions A limited expression in [23] is a pattern that matches not only a single character but an arbitrary set of characters and it is a subset of the wealth of alternatives for extended patterns presented in [1, 22]. For convenience of notation, we use the symbols '*', '^', '[', and ']' to denote common types of symbols as follows (this is consistent with *grep* and *agrep*):

- A '*' don't care symbol can be represent the matching with any single character.
- A '^' complement symbol can represent the matching with all characters except the one that is complemented.
- A pair of '[' and ']' defines a class symbol that allows the matching with a subrange of characters.

To search for these limited expressions patterns, we need only to modify the preprocessing phase without additional search complexity. We have:

$$R_{i,j} = \begin{cases} 0, & \text{if } p_i = *, \text{ for } 1 \leq j \leq |\Sigma| \\ 0, & \text{if } p_i = \hat{}\sigma_1, \text{ for } 1 \leq j \leq |\Sigma| \text{ and } j \neq map(\sigma_1, \Sigma) \text{ for } 1 \leq i \leq m \\ 0, & \text{if } p_i = [\sigma_1\sigma_2], \text{ for map}(\sigma_1, \Sigma) \leq j \leq map(\sigma_2, \Sigma) \\ 1, & \text{otherwise} \end{cases}$$

where σ_j is the j-th character of the alphabet.

An example for the construction of the R for these simple cases of limited expressions is now presented. The alphabet Σ is {a,b,c,d,e} with $|\Sigma|$=5. The pattern string p is the "abba", with m=4. If the first character of the pattern string is replaced by a don't care character then the bit-level memory map R is modified as shown in Table 3. Similarly, if the first search character is the complement of 'a' then the bit-level memory map R is modified as shown in Table 3. Finally, if the first pattern character is the subrange [ad] then the bit-level memory map R is modified as shown in Table 3.

R	a	b	c	d	e
*	0	0	0	0	0
b	1	0	1	1	1
b	1	0	1	1	1
a	0	1	1	1	1

R	a	b	c	d	e
^a	1	0	0	0	0
b	1	0	1	1	1
b	1	0	1	1	1
a	0	1	1	1	1

R	a	b	c	d	e
[ad]	0	1	1	0	1
b	1	0	1	1	1
b	1	0	1	1	1
a	0	1	1	1	1

Table 3. Bit-level maps for limited expressions

3 Mapping Searching Phase onto Processor Array

Initially, the implementation of the searching phase of the flexible string matching algorithm is discussed on the processor array, which imposes the main computation load since $m << n$. Figure 2a shows the data-dependence graph and the parallel timing diagram for flexible string matching algorithm. Node (i, j) of the graph 2, $(1 \leq i \leq m, 1 \leq j \leq n)$ is stored the character p_i of the pattern string p and the character t_j of the input text string t. Moreover, each node (i, j) of the graph is assigned an row i of the R for the pattern character p_i. In Figure 2a, to calculate an element of the vector $F_i'^0$ $(1 \leq i \leq m)$ after we read an new text character t_j $(1 \leq j \leq n)$, we need the previous value F_{i-1}^0 as indicated in Figure 2b. We suppose each processor is responsible to compute all nodes of each row of Figure 2a. We conclude from the dependence graph that all nodes which lie on the same diagonal (from left-bottom to right-top) can be performed at the same time as is shown in Figure 2a by dotted diagonal lines.

We transform the original dependence graph of Figure 2a to local dependence graph as shown in Figure 3a so that the characters of the text may flow via local edges while the vertical and horizontal axis of the graph represent the time steps and the characters of the pattern respectively. The area complexity of the algorithm is defined as the number of elementary cells (or Processing Elements, PEs) being active at any time step. This information is given by the number of columns of the dependence graph whereas the time steps required are given by the number of rows of graph.

It should be clear that the processor array is derived from projecting the time axis of the dependence graph of Figure 3a and is shown in Figures 3b for a generic problem with m=4 and for arbitrary n and $|\Sigma|$. It is a linear array consisting of m cells connected to each other via two communication channels, one transferring the binary representation of text characters and an another transferring the bit-level results. Each

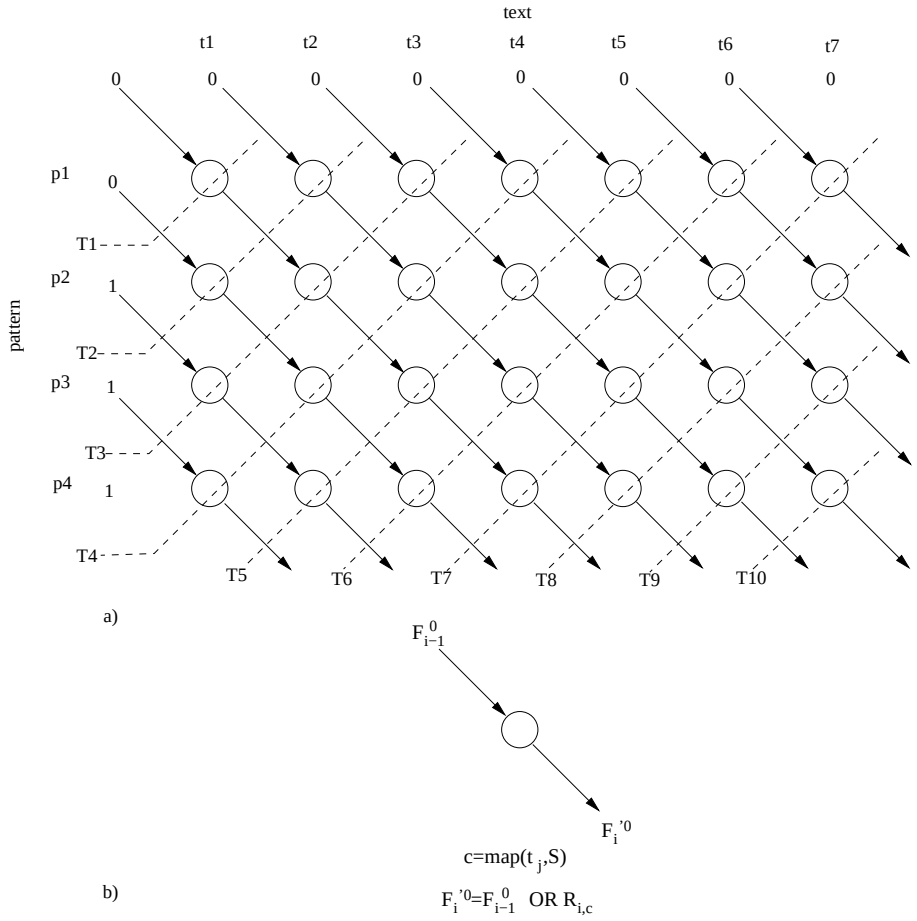

Fig. 2. (a) Dependence graph and parallel timing diagram for flexible string matching (b) Computation

row R_i, $1 \leq i \leq m$ of the bit-level memory map R is allocated to a PE, so that reading through the specialized addressing function $map(ch, \Sigma)$ produces a single bit per PE. For the implementation of the $map(ch, \Sigma)$ function we introduced a programmable hardware (decoder) in each cell. Further, each cell performs the logical operation as shown in Figure 3b right, i.e. both mapping and OR operations. Therefore, each cell updates the result and it is sent to next cell. It is noted that the bit-level results travels along the same direction as the text characters but at half speed, that is with an intermediate delay between cells. The overall computation time is $n + m - 1$ time steps. Given the fact that usually $m << n$ it can be argued that the computation time is approximately n steps. The area required is m PEs. Taking the example of the UNICODE character set the local memory requirements are 16Kbytes per cell.

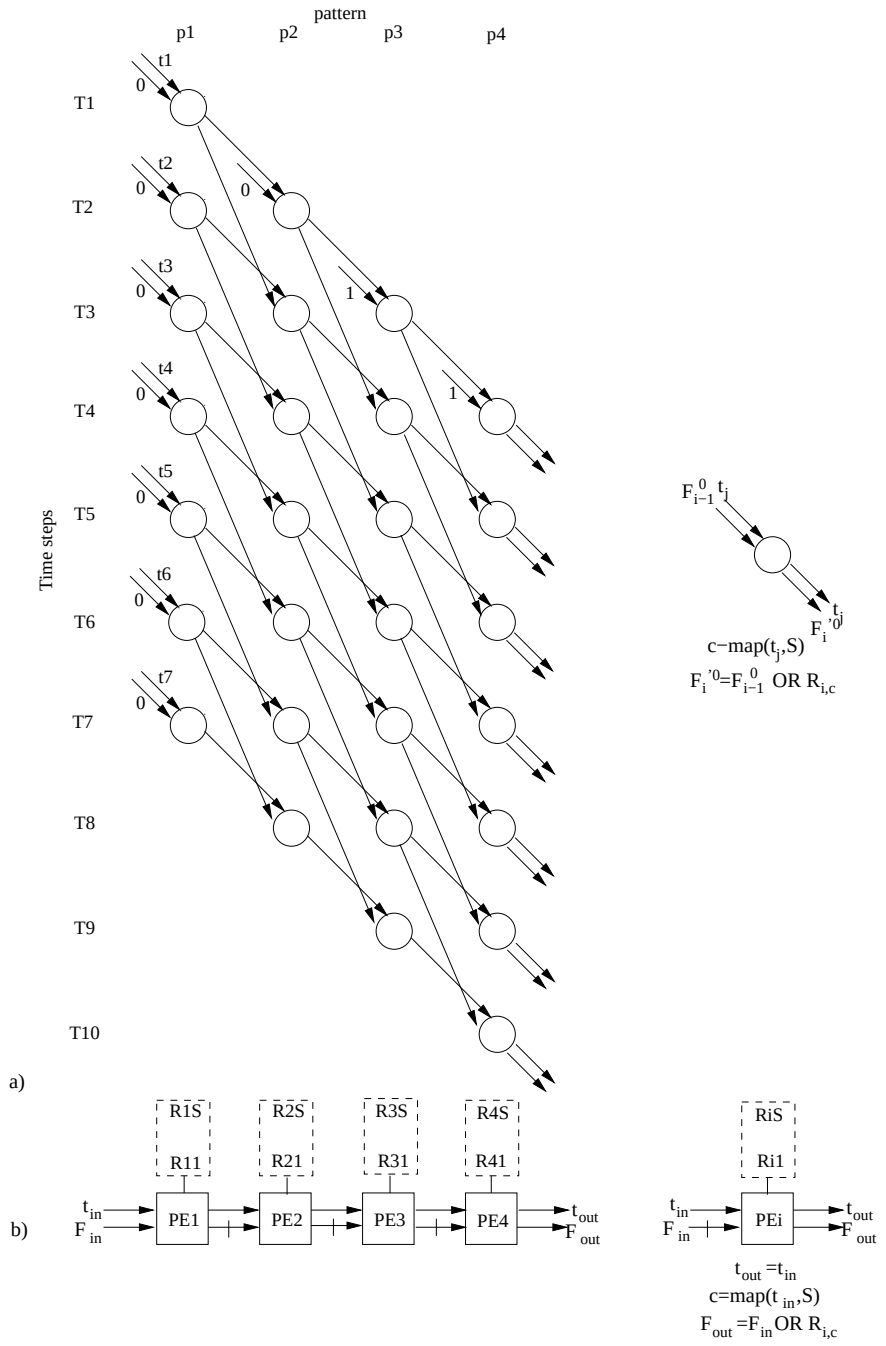

Fig. 3. (a) Graph transformation and (b) Processor array

The important differences of this proposed architecture in contrast to the other processor array designs are as follows: First, many architectures suitable for different applications of approximate string matching have already proposed in literature, most of which accommodated primarily for DNA sequence analysis, like [5, 8, 13, 19, 15, 12]. These architectures are optimized for the matching of very long strings of nearly the same length. In contrast, our proposed processor array requires a solution accommodated for string matching of mostly small patterns of arbitrary alphabets against a large free textbases. Therefore, our systolic data flow is unidirectional in contrast to the bidirectional data flow architectures. Second, the processor array design presented herein performs flexible string matching as opposed to the previous architectures [5, 8, 13, 19, 15, 12] that perform simple approximate string matching. Therefore, we introduced the implementation of the encoding scheme, i.e. the introduction of the encoder and the bit-level memory modules. This encoding scheme enables the efficient VLSI implementation of the flexible string matching algorithm in contrast to the limited flexibility of the encoding schemes used in the previous designs. Finally, an advantage of our architecture is that perform simple operations in cells and minimal hardware is required.

4 An Implementation of the Preprocessing Phase

In this section the implementation of the preprocessing phase, i.e. the construction of the bit-level memory map R is discussed. The aim is to use the same processor array architecture as in the previous section in order to produce the bit-level memory map with its elements allocated to the appropriate PEs. Furthermore, all the alternatives of flexible searching stated should be accomodated. For this reason each character p_i, $1 \leq i \leq m$, of the pattern p should be accompanied by some control information, denoting the presence of the don't care symbol, the complement symbol or the subrange symbol. Especially in the case of subrange symbol two characters are required denoting the boundaries of the subrange. Therefore each augmented character p_i of the pattern consists of (i) at most two bit strings which correspond to the binary codes of two valid characters belonging to Σ and (ii) necessary control information.

The operations that should be performed by the pre-processing phase are essentially writing operations to the appropriate row R_i, $1 \leq i \leq m$, of the bit-level memory map R that is allocated to the i-th PE. Since each memory location is a single bit the writing operation consists of setting this bit either to 0 or to 1. Using the example of the extended UNICODE character set the following protocol is proposed in Table 4. Three control bits are specified: Bit 0 for setting a memory map bit either to 1 or to 0. Bits 1 and 2 denoting the presence of 0, 1 or 2 valid characters to be read.

In order to implement the preprocessing phase protocol onto the linear processor array of Figure 3b the following assumptions are made. First, it is assumed that the text channel is used for transferring the character codes at a rate of a single character per transfer step, while the result channel is wide enough to carry the necessary control bits. Second, it is assumed that the alphabet Σ, its encoding and size $|\Sigma|$ are preloaded to the PEs. Third, the maximum length of the pattern is equal to the processor array size m and this is known to the system. Finally, the i-th PE should be aware of its

Functions	Ctrl	2	1	0	Chars 0/7	8/15
Clear (set to 0)		0	0	0	-	-
Don't care character (set to 1)		0	0	1	-	-
Simple character encoding		0	1	0	σ_1	-
Complement of a character		0	1	1	σ_1	-
Character subrange		1	1	0	σ_1	σ_2

Table 4. A preprocessing phase protocol

position in the array, that is it should have a cell id equal to i, as numbered in Figure 3b. The preprocessing phase can start by the reset signal through the control input and then each cell counts the loading steps that is j steps for the j-th PE. Therefore, each cell executes the following piece of pseudocode for the construction of the row of the bit-level memory map R:

$bit \leftarrow ctrl_0$
if $ctrl_1$=0 and $ctrl_2$=0 then $l_0 \leftarrow map(0, \Sigma)$, $hi \leftarrow map(|\Sigma|, \Sigma)$
if $ctrl_1$=1 and $ctrl_2$=0 then $l_0 \leftarrow map(\sigma_1, \Sigma)$, $hi \leftarrow map(\sigma_1, \Sigma)$
if $ctrl_1$=1 and $ctrl_2$=1 then $l_0 \leftarrow map(\sigma_1, \Sigma)$, $hi \leftarrow map(\sigma_2, \Sigma)$
for i=$map(0, \Sigma)$ to $l_0 - 1$ do
    $R_i \leftarrow$ NOT bit
for i=l_0 to hi do
    $R_i \leftarrow bit$
for i=$hi + 1$ to $map(|\Sigma|, \Sigma)$ do
    $R_i \leftarrow$ NOT bit

There are $|\Sigma|$ writing operations to consecutive single bit locations in the row of the bit-level memory map M. The lowest address of the memory is denoted is denoted by $map(0, \Sigma)$ whereas the highest address is denoted by $map(|\Sigma|, \Sigma)$. All these memory locations are accessed once during the pre-processing phase, by means of three consecutive FOR loops. This allows to keep the computation time uniform for any type of preprocessing operation, that is $2m$ loading plus $|\Sigma|$ writing steps. The completion of the preprocessing phase enables the commencement of the searching phase. From the above description the preprocessing phase can be implemented in the some PE that performs the searching phase with the addition of limited programmable hardware. Therefore, the whole systolic algorithm can be performed onto a special purpose processor array.

5 Conclusions

A linear processor array implementation for flexible string matching have been presented in this paper. This architecture is a bit-parallel realization of the non-deterministic finite automaton, which minimizes the amount of data flow between adjacent cells. Future research plans include the extension of the bit-level architecture to the block-based architecture which requires fewer cells and time steps.

References

1. R. Baeza-Yates, G.H. Gonnet, A New Approach to Text Searching, *Communications of the ACM*, 35(10):74-82, (1992)
2. R. Baeza-Yates, G. Navarro, Faster Approximate String Matching, *Algorithmica*, 23(2):127-158, (1999)
3. A.A. Bertossi, F. Logi, Parallel String Matching with Variable Length don't Cares, *Journal of Parallel and Distributed Computing*, 22(2):229-234, (1994)
4. H.D. Cheng, K.S. Fu, VLSI Architectures for String Matching and Pattern Matching, *Pattern Recognition*, 20(1):125-141, (1987)
5. M.J. Foster, H.T. Kung, The Design of Special Purpose VLSI chip, *IEEE Computer*, 13:26-40, (1980)
6. Z. Galil, A Constant Time Optimal Parallel String Matching Algorithm, *Journal of the ACM*, 42(4):908-918, (1995)
7. R. Hughey, Parallel Hardware for Sequence Comparison and Alignment, *CABIOS*, 12(6):473-479, (1996)
8. R. Hughey, D.P. Lopresti, Architecture of a Programmable Systolic Array, *Proceedings International Conference Systolic Arrays* (1998), 41-50
9. Y. Jiang, A.H. Wright, O(k) Parallel Algorithms for Approximate String Matching, *Neural, Parallel and Scientific Computations*, 1:443-452, (1993)
10. S.Y. Kung, *VLSI Array Processors*, Prentice-Hall, 1988.
11. C.M. Landau, U. Vishkin, Fast Parallel and Serial Approximate String Matching, *Journal of Algorithms*, Vol.10, No.2, pp.157-169, 1989.
12. D. Lavenier, Speeding up Genome Computations with a Systolic Accelerator, *SIAM News*, 31(8):6-7, (1998)
13. D.P. Lopresti, P-NAC: A Systolic Array for Comparing Nucleic Acid Sequences, *IEEE Computer*, 20(1):98-99, (1987)
14. D.P. Lopresti, Rapid Implementation of a Genetic Sequence Comparator Using Field-programmable Logic Arrays, *Advanced Research in VLSI*, 138-152, (1991)
15. G.M. Megson, Efficient Systolic String Matching, *Electronic Letters*, 26(24):2040-2042, (1990)
16. P.D. Michailidis, K.G. Margaritis, On-line String Matching Algorithms: Survey and Experimental Results, *International Journal of Computer Mathematics*, 76(4):411-434, (2001)
17. G. Navarro, R. Raffinot, Fast and Flexible String Matching by Combining Bit-parallelism and Suffix Automata, *ACM Journal of Experimental Algorithmics*, 5(4), (2000)
18. N. Ranganathan, R. Sastry, VLSI Architectures for Pattern Matching, *International Journal of Pattern Recognition and Artificial Intelligence*, 8(4):815-843, (1994)
19. R. Sastry, N. Ranganathan, K. Remedios, CASM: A VLSI Chip for Approximate String Matching, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(8):824-830, (1995)
20. R. Sidhu, A. Mei, V.K. Prasanna, String Matching on Multicontext FPGAs Using Self-reconfiguration, *Proceedings International Symposium on Field- Programmable Gate Arrays*, (1999)
21. R. Sidhu, V.K. Prasanna, Fast Regular Expression Matching Using FPGAs, *Proceedings IEEE Symposium on Field-Programmable Custom Computing Machines*, (2001)
22. S. Wu, U. Manber, Fast Text Searching Allowing Errors, *Communications of the ACM*, 35(10):83-91, (1992)
23. S. Wu, U. Manber and G. Myers, A Subquadratic Algorithm for Approximate Limited Expression Matching, *Algorithmica*, 15:50-67, (1996)