Implementation of a Programmable Array Processor Architecture for Approximate String Matching Algorithms on FPGAs

Panagiotis D. Michailidis, and Konstantinos G. Margaritis

Parallel and Distributed Processing Laboratory
University of Macedonia, Department of Applied Informatics
156 Egnatia str., P.O. Box 1591, 54006 Thessaloniki, Greece
{panosm, kmarg}@uom.gr

Abstract

Approximate string matching problem is a common and often repeated task in information retrieval and bioinformatics. This paper proposes a generic design of a programmable array processor architecture for a wide variety of approximate string matching algorithms to gain high performance at low cost. Further, we describe the architecture of the array and the architecture of the cell in detail in order to efficiently implement for both the preprocessing and searching phases of most string matching algorithms. Further, the architecture performs approximate string matching for complex patterns that contain don't care, complement and classes symbols. We also implement and evaluate the proposed architecture on a field programmable gate array (FPGA) device using the JHDL tool for synthesis and the Xilinx Foundation tools for mapping, placement, and routing. Finally, our programmable implementation achieves about 9-340 times faster than a desktop computer with a Pentium 4 3.5 GHz for all algorithms when the length of the pattern is 1024.

1. Introduction

The problem of approximate string matching is stated as follows: Let a given alphabet (a finite character set) Σ, a short pattern string $p = p_1p_2...p_m$ of length m, a large text string $t = t_1t_2...t_n$ of length n, with $m << n$ and a maximal number of errors allowed, $k \geq 0$, we are interested in finding all approximate occurrences of a substring of t whose distance to p is at most k. Errors can be substituting, deleting or inserting a character. The approximate string matching problem can be extended to include more flexible patterns, including patterns that contain with don't care, complement and classes symbols. In general, an approximate string matching algorithm consists of two phases: the preprocessing phase in p and the searching phase of p in t. The preprocessing phase involves the construction of the two two-dimensional bit-level arrays R and M where each row corresponds to a character of the pattern which can be used in the searching phase. The searching phase consists of constructing of an array in order to find all approximate occurrences of p in t.

Several arrays processor architectures have been proposed by several researchers for flexible approximate string matching [10, 4, 11, 12, 6, 8]. The abovementioned application specific arrays processors can provide the fastest means of running a particular algorithm with very high processing element (PE) density. However, they are limited to a single algorithm, and thus cannot supply the flexibility necessary to run the wide variety of algorithms for approximate string matching. Therefore, this paper discusses a generic array processor architecture and the architecture of the cell, including descriptions of the major components that help the architecture achieve its goals.

There are two basic goals for the generic architecture. The first is to propose a unified programmable array processor architecture suitable for efficient execution of a class of approximate string matching algorithms [1, 13, 9, 3, 7]. There are a great number of approximate string matching algorithms in text retrieval, and programmability is required to accommodate these different algorithms within a single system. As new algorithms are developed, this architecture will be able to execute many of them without the need for redesigning the architecture.

The second goal is the proposed programmable array

1-4244-0054-6/06/$20.00

processor architecture to achieve flexibility while providing high performance on a level with the application specific architectures. Further, the proposed architecture to support approximate string matching for simple and complex patterns, like don't care, complement and classes symbols.

2. A Unified Array Processor Architecture

In this section, we describe the architecture of the programmable array processor and cell.

2.1. Architecture of the Array

The architecture of the linear programmable array is shown in Figure 1. This structure was obtained as a result of applying the data dependence graph method [5] to the partitioned realization of a variety of approximate string matching algorithms [1, 13, 9, 3, 7]. More details for the mapping of approximate string matching algorithms to specific array processor architectures using the dependence graph method is presented in [7, 6]. In all those cases the method produced arrays with the same architecture, which led us to conclude that a class-specific array for the efficient execution of a class of algorithms was possible.

The array is one linear structure of m processing elements (PEs), as shown in Figure 1 for $m = 4$; five additional I/O interfaces, located at the intermediate of the cells, are used to simplify data communications that transfer the binary representation of text characters and the bit or byte-level results but do not perform any computations. The communication with the host is performed through the boundary PEs. This array executes all flexible approximate string matching algorithms, including the dynamic programming and NFA algorithms. Communications are unidirectional, among neighbor cells and neighbor I/O modules. The operation of the linear array is controlled by a array controller. The array controller will be responsible for issuing signals to the array and for controlling communication with the host system.

In Figure 1 one character of the pattern and one row of the bit-level memory maps R and M are preloaded into the PE of the array processor so that using the specialised 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. The other text string (or textbase) flows from left to right through the array. During each step, one elementary computation of any approximate string matching algorithm of the class is

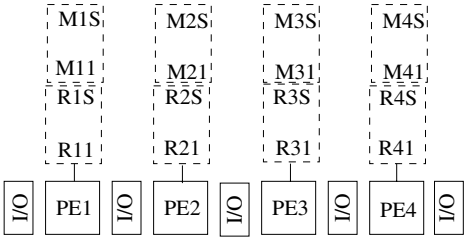

Figure 1. Linear programmable array processors for approximate string matching algorithms

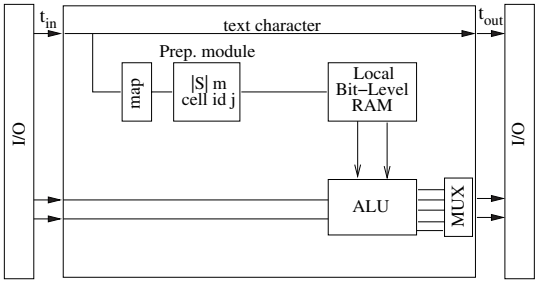

Figure 2. Cell specification for the programmable array processors

performed in each PE. The result is collected on the rightmost cell when the last character of the flowing string is output. If m is the length of the pattern and n is the length of the textbase, the computation of any algorithm in the class is performed in $m + n - 1$ steps on m PEs. A partitioning strategy to handle situations where the pattern length is usually larger than the array processor size is discussed in [7, 6].

2.2. Architecture of the Cells

For the design of the programmable architecture, we examine the requirements (input, output, memory, registers and operations) of the cells of a variety of flexible approximate string matching algorithms and incorporated hardware within the cell to provide for fast execution of these algorithms while maintaining flexibility. Therefore, the architecture of the processing element (PE) is shown in Figure 2. Each PE is connected to each other via three input and output communication channels when $k = 1$. One channel transferring the binary representation of text characters and another two transferring the bit-level (or byte-level) results. The main components of the PE are described as follows:

1. **Preprocessing Module**: This module is used to implement the preprocessing phase of several approximate string matching algorithms. More specifically, this phase constructs the bit-level memory maps R and M take into account the patterns that contain don't care, complement and classes symbols. The description of the implementation for the preprocessing module is presented in [7, 6].

2. **Character to Address Decoder**: From the text stream of Figure 2 it is observed that the binary code of the text character ch currently being transferred is used as input for the Character to Address Decoder. The decoder is essentially the programmable hardware implementation of the $map(ch, \Sigma)$ function. The output of the decoder is the address c of a bit-level memory location.

3. **Local Bit-Level Random Access Memory**: The local bit-level RAM keeps stored a row R_i and M_i, $1 \leq i \leq m$, of the bit-level memory maps R and M respectively. Therefore, the local memory size is $|\Sigma|$ bits and the address required to access such a memory is of $log|\Sigma|$ bits. Taking the example of the ASCII character set the local memory requirements are 32 bytes per cell. The result or the memory reading operation is two bits quantities $R_{i,c}$ and $M_{i,k}$ which in turn is one of the three operands of arithmetic unit.

4. **Arithmetic Unit**: The arithmetic unit consists of a arithmetic logic unit (ALU) and general purpose registers. The ALU is programmable and it consists of seven small units, each unit implements all common logic functions and the standard arithmetic functions (addition or subtraction) of a string matching algorithm. The unit is determined by three bits which are provided by array controller. The general purpose registers store intermediate results (such as D, Dv, DL, F, aux) and are usually directly connected to the data bus.

5. **Multiplexers**: Output multiplexers (MUX) of the ALU take care of selecting the proper communication channels that transfer the bit/byte-level results in the next cell when an algorithm is selected.

3 Implementation and Performance Evaluation

In this section, we present the performance evaluation of the proposed parallel architecture on a field programmable gate arrays (FPGA) device. Firstly, we have synthesized the PE design in JHDL (Java Hardware Description Language) [2]. The JHDL is a complete structural design environment, including debugging, netlisting and other design aids. Circuits are described by writing Java code that programmatically builds the circuit via the JHDL libraries. Once constructed, these circuits can be debugged and verified with the design browser, a circuit verification and debugging tool. After the successful verification of the proposed design, we generated an EDIF netlist format of the circuit by JHDL and this can be passed to Xilinx Foundation software tool for placement and routing so that targeted on a Xilinx Virtex II device.

Table 1. Search times (in seconds) of several algorithms for various length of the pattern on a Pentium 4 3.6 GHz

Pattern size	32 - 256	512	1024
Dyn. Prog. - mismatches	21,791	43,047	90,75
Dyn. Prog. - differences	72,469	144,609	288,157
Dyn. Prog. - Myers	75,12	146,34	295,157
NFA - exact	81,219	199,37	401,7
NFA - mismatches	265,141	515,24	1101,34
NFA - differences	309,73	602,13	1206,32

We have implemented a linear array of PEs and mapped onto a Virtex device of Xilinx using the Xilinx Integrated Software Environment (ISE) 5.2i. The test platform used was a Xilinx Virtex II XC2V4000 FPGA, which is able to accommodate 256 PEs. We included in the experiment a target clock rate of 100 MHz, which is the maximum frequency supported by the FPGA platform. The size of the internal memory of the FPGA is 2160 Kbits.

Tables 1 and 2 report the performance of several algorithms for searching English database (which contains 30,000,000 characters) for pattern sizes of various lengths on a Pentium 4 3.6 GHz and on Virtex II XC2V4000 respectively. Note that the value of k of approximate string matching algorithms is used in the experiments was 2 and the programs of algorithms on Pentium 4 were optimised in C language. The pattern sizes have been chosen to illustrate the effect that length has on performance. Maximum performance is achieved when the pattern size is closely matched to an integer multiple of the array processor size for the dynamic programming algorithms. Note that the performance increases when the pattern whose length is much higher than the number of PE cells of the array processor because requires the computation to be par-

Table 2. Search times (in seconds) of several algorithms for various length of the pattern on Virtex II. The speed up compared to the Pentium 4 is also reported

Pattern size	32 - 256	512	1024
Dyn. Prog. - mis.	2,49 (8)	4,98 (8)	9,96 (9)
Dyn. Prog. - dif.	5,23 (13)	10,45 (13)	20,91 (13)
Dyn. Prog. - Myers	6,72 (11)	13,44 (11)	26,89 (11)
NFA - exact	2,36 (33)	2,34 (80)	2,36 (161)
NFA - mismatches	3,24 (82)	3,20 (159)	3,22 (340)
NFA - differences	4,23 (73)	4,17 (142)	4,19 (285)

titioned into more processing passes which re-use the data coming from the text database. Therefore, the number of passes of dynamic programming algorithms were 1, 2 and 4 when the pattern length was 32-256, 512 and 1024 respectively. On the other hand, the constant performance is achieved for the nondeterministic finite automata (NFA) algorithms when the pattern size is larger than the array processor size. This fact is due to the advantage of the intrinsic parallelism of the bit-operations inside a word of the cell. In this case, the number of passes of NFA algorithms was 1 for all pattern lengths. Further, we observe from the results that the NFA algorithms produce better performance than the dynamic programming algorithms. This due to the fact that the NFA algorithms perform simple and few operations. Finally, our programmable implementation is about 9-340 times faster than a desktop computer with a Pentium 4 for all algorithms when the length of the pattern is 1024.

4. Conclusions

In this paper we have demonstrated that the programmable architecture provide an effective solution to high performance string searching. We have presented the design and implementation of the proposed programmable architecture for efficient execution of a class of approximate string matching algorithms. Further, both of the preprocessing and the searching phases of most approximate string matching algorithms can be efficiently implemented onto the same programmable architecture. The proposed implementation showed supercomputer performance at low cost on an off-the-shelf FPGA. The programmable architecture derived in this work can speed up approximate string matching algorithms by software in several orders of magnitude. Previous works in the literature on string matching in hardware focused on sequence comparison for biological problems using different approaches but, if implemented with current technologies, should provide similar speed up. The main contribution of this work is the computation of the approximate string matching for simple and complex patterns instead of sequence comparison. Therefore, it should be noted that the design of PE is flexible and simple. Finally, the proposed architecture may be adopted as a basic structure for a universal flexible approximate string matching engine.

References

[1] R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

[2] P. Bellows and B. Hutchings. JHDL-an HDL for reconfigurable systems. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184, 1998. Software available at http://www.jhdl.org.

[3] M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and J. R. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Information Processing Letters*, 80:279–285, 2001.

[4] R. Hughey. Parallel hardware for sequence comparison and alignment. 12(6):473–479, 1996.

[5] S. Y. Kung. *VLSI Array Processors*. Prentice-Hall, 1988.

[6] P. Michailidis. *Parallel and Distributed Implementations for Approximate String Matching*. PhD thesis, Department of Applied Informatics, University of Macedonia, 2004. (in Greek).

[7] P. Michailidis and K. Margaritis. Processor array implementations for flexible approximate string matching. Technical report, Department of Applied Informatics, University of Macedonia, 2002.

[8] P. D. Michailidis and K. G. Margaritis. Bit-level processor array architecture for flexible string matching. In *Proceedings of the 1st Balkan Conference in Informatics*, pages 517–526, 2003.

[9] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.

[10] N. Ranganathan and R. Sastry. VLSI architectures for pattern matching. *International Journal of Pattern Recognition and Artificial Intelligence*, 8(4):815–843, 1994.

[11] R. Sidhu, A. Mei, and V. K. Prasanna. String matching on multicontext FPGAs using self-reconfiguration. In *Proceedings of International Symposium on Field-Programmable Gate Arrays*, 1999.

[12] R. Sidhu and V. K. Prasanna. Fast regular expression matching using FPGAs. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[13] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.