Efficient FPGA-based Hardware Algorithms for Approximate String Matching

Sadatoshi MIKAMI Yosuke KAWANAKA
Shin'ichi WAKABAYASHI[1] Shinobu NAGAYAMA[2]
Graduate School of Information Sciences, Hiroshima City University
3-4-1, Ozuka-higashi, Asaminami-ku, Hiroshima 731-3194 Japan
TEL: +81-82-830-1760 FAX: +81-82-830-1792
Email: [1]wakaba@hiroshima-cu.ac.jp, [2]s_naga@hiroshima-cu.ac.jp

Abstract: In this paper, an efficient FPGA-based hardware algorithm and its extensions are proposed for calculating the edit distance as a degree of similarity between two strings. The proposed algorithms are implemented on FPGA and compared to software programs. Experimental results show the effectiveness of the proposed algorithms.

1. Introduction

Approximate string matching is a problem to search for strings similar to a given pattern from the input string [3]. As main applications of approximate string matching, it can be used for text retrieval in database, analysis of DNA, protein sequences in bioinformatics, *etc*. Algorithms for approximate string matching have been extensively studied to shorten its computation time.

Since the advent of VLSI era, it has become possible to realize algorithms on VLSI circuits as hardware algorithms to drastically reduce the computation time to solve problems [6]. For string matching, several hardware algorithms have been proposed for various kinds of string matching [1]. For example, Mukhopadhyay [7], Foster and Kung [2] proposed hardware algorithms for the string pattern matching problem [5]. Kikuno, *et al.* proposed a hardware algorithm for the longest common subsequence problem [9].

In this paper, an efficient FPGA-based hardware algorithm and its extensions are proposed for the string-to-string correction problem. The string-to-string correction problem is a problem to calculate the edit distance as a degree of similarity between two strings [8]. The proposed algorithm and its extended versions are implemented on FPGA and compared to software programs. Experimental results show the effectiveness of the proposed algorithms.

As previous results related to our study, for the problem of calculating the edit distance, Yu, *et al.* have also proposed a hardware algorithm to be implemented on an FPGA chip [10]. However, in this algorithm, character symbols in a pattern were restricted to A, C, G, and T, since their algorithm was originally proposed for the analysis of DNA sequences. It would be very difficult to extend this algorithm for the general approximate string matching problem. As far as we investigated, no previous results have been known for hardware implementation of a string matching engine for an arbitrary set of character symbols.

This paper is organized as follows. Section 2 gives the definition of edit distance, and formulates the string matching problem discussed in this paper. Section 3 presents a hardware algorithm to solve the problem. Section 4 shows some extensions of the proposed hardware algorithm. Section 5 gives experimental results to show the effectiveness of the proposed algorithms. Finally, some concluding remarks are given in Section 6.

2. Edit Distance

Let A be a finite string (or sequence) or characters (or symbols). $A < i >$ is the ith character of string A; $A < i : j >$ is the ith through jth characters (inclusive) of A if $i \leq j$. $|A|$ denotes the length of string A.

An *edit operation* is a pair $(a, b) \neq (\Lambda, \Lambda)$ of strings of length less than or equal to 1, and usually written $a \rightarrow b$, where Λ denotes the null string. String B results from the application of the operation $a \rightarrow b$ to string A, written $A \Rightarrow B$ *via* $a \rightarrow b$, if $A = \sigma a \tau$ and $B = \sigma b \tau$ for some strings σ and τ. We call $a \rightarrow b$ a *change* operation if $a \neq \Lambda$ and $b \neq \Lambda$; a *delete* operation if $b = \Lambda$; and an *insert* operation if $a = \Lambda$.

Let γ be an arbitrary cost function which assigns to each edit operation $a \rightarrow b$ a nonnegative real number $\gamma(a \rightarrow b)$. Extend γ to a sequence of edit operations $S = s_1, s_2, \ldots, s_m$ by letting $\gamma(S) = \sum_{i=1}^{m} \gamma(s_i)$. We now let the *edit distance* $\delta(A, B)$ from string A to string B be the minimum cost of all sequences of edit operations which transform A into B [8].

Given a pair of strings A and B, the *approximate string matching problem* is to find the edit distance between two strings A and B. For this problem, the following theorem holds [8].

[Theorem 1] Let $A(i) = A < 1 : i >$ and $B(j) = B < 1 : j >$, and $D(i, j) = \delta(A(i), B(j))$, $0 \leq i \leq |A|$, $0 \leq j \leq |B|$. Then,

$$\begin{aligned} D(i,j) = \min\{ & \\ & D(i-1, j-1) + \gamma(A < i > \rightarrow B < j >), \\ & D(i-1, j) + \gamma(A < i > \rightarrow \Lambda), \\ & D(i, j-1) + \gamma(\Lambda \rightarrow B < j >)\} \end{aligned} \quad (1)$$

for all i, j, $1 \leq i \leq |A|$, $1 \leq j \leq |B|$. □

From Theorem 1, for given two strings A and B, the edit distance $\delta(A, B)$ from string A to string B is given as $D(|A|, |B|)$. In this paper, the matrix D is called the *edit distance matrix*.

In this paper, we slightly extend the approximate string matching problem, and formulate this extended problem as the *multiple string matching problem*. Given a set of finite strings $R = \{S_1, S_2, \ldots, S_m\}$ and a *pattern* string P, the

multiple string matching problem is to calculate the edit distance $\delta(P, S_i)$ from string P to string S_i for all i, $1 \leq i \leq m$. We call each string S_i to be matched a *text string*. When $m = 1$, this problem is equivalent to the original approximate string matching problem. Applications of this problem include the text retrieval in large text data base, and DNA sequence alignment in bioinformatics.

3. Basic Algorithm

In this paper, for the multiple string matching problem, we propose a hardware algorithm, which is implemented on FPGAs, to realize a high-speed calculation of edit distance. The basic idea of the proposed algorithm is as follows. From Theorem 1, for a given pair of two strings, the edit distance is obtained by computing the edit distance matrix. In this matrix computation, one can easily understand that all entries on any positive slope diagonal lines can be computed in parallel, since there is no data dependencies among them. Figure 1 shows how to compute the edit distance matrix in parallel. The basic idea of the proposed algorithm is to assign a processing element to each row of the edit distance matrix, and all processing units calculate the values of matrix elements on each positive slope diagonal line in parallel.

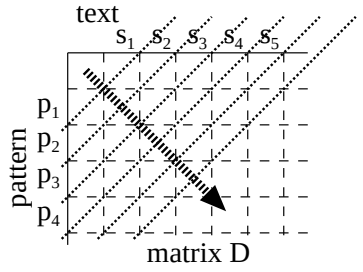

Figure 1. Parallel calculation of the edit distance matrix.

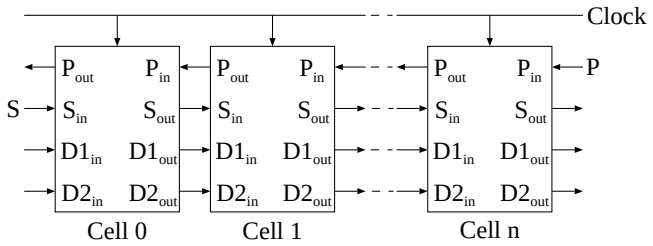

Figure 2. Systolic architecture.

Figure 2 shows the overview of the proposed architecture. The architecture consists of $(n + 1)$ simple processing units, called *cells*, where $n = |P|$. As mentioned, the entire algorithm calculates diagonal elements in the edit distance matrix in the parallel and pipeline fashion. In the following, the details of the algorithm are explained.

3.1 Inputs of the Algorithm

For a given pattern string $P = p_1 p_2 \ldots p_n$, let $P' = \theta p_1 p_2 \ldots p_n$, and each character in P' is stored in each cell in advance, where θ is a special start symbol. The pattern string is input from the rightmost cell. On the other hand, a given set of text strings to be matched is concatenated and sequentially input from the leftmost cell. When concatenating multiple text strings, special symbol "," is inserted as a delimiter, and θ and λ are added as the first and last symbols, respectively. For example, if $S_1 = abb$, $S_2 = cba$, and $S_3 = acb$ then the text string to be input is $S = \theta abb, cba, acb\lambda$.

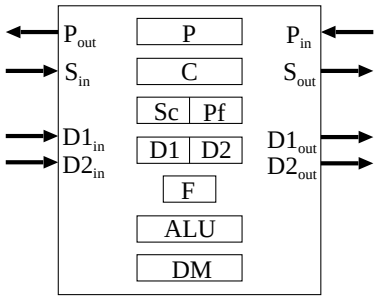

Figure 3. The cell.

```
t0: begin
  C ← Sin;
  if Sin = θ then Sc ← true;
  if Sin = λ then Sc ← false;
  F ← (Sin = θ) ∨ (Sin = ",") ∨ (Sin = λ);
  sub ← DM(P, C);
end;
t1: begin
  if Sc then
    case (Pf, F) begin
      00: D1 ← min{D1in + del,
                   D1 + ins, D2in + sub};
      01: D1 ← D1in + del;
      10: D1 ← D1 + ins;
      11: D1 ← 0;
    end;
  D2 ← D1;
end;
```

Figure 4. The algorithm of a cell.

3.2 The Cell

The structure of a cell is shown in Fig. 3. Each cell consists of two latches P and C, which are used to store characters in P and S_i. The cost function γ is stored in the memory DM in each cell. DM is realized as a two-dimensional array of words, and is called the *edit cost matrix*. For any pair of two characters a and b, $DM(a, b)$ returns the value of $\gamma(a \to b)$. The values of all elements of the edit cost matrix are set in advance before starting the string matching. $D1$ and $D2$ are also latches, which store values of the edit distance matrix D. Sc, Pf and F are flags to be used in the cell algorithm.

3.3 Behavior of the Cell

As noted, the behavior of each cell is classified into two phases, the pattern input phase and the text matching phase. In the former phase, the pattern string is input from the rightmost cell one character by one character, and shifted left until all characters are stored in corresponding cells. Any cell which stores the start symbol θ in latch P sets the flag Pf to 1, and any other cell sets Pf to 0. In the text matching phase, the actual text matching is performed. In the following, we only describe the cell behavior of the text matching phase.

Figure 4 shows the behavior of a cell during the text matching phase. We assume that each clock cycle consists of two clock phases (t_0, t_1). We also assume that the delete and insert costs of any character are the same values, and denoted as del and ins, respectively. For each clock cycle, each cell repeatedly executes this algorithm.

Cell i stores pattern character p_i, and it calculates all elements $D(i, *)$ of the edit distance matrix. Assume that cell i receives text string character s_i from its left neighbor cell, stores it in latch C, and starts calculating $D(i, j)$ in clock cycle T_k. Figure 5 shows this situation. From Theorem 1, to calculate $D(i, j)$, values of $D(i, j-1)$, $D(i-1, j)$, and $D(i-1, j-1)$ are required. Cell i holds $D(i, j-1)$ in latch $D1$, which was calculated in clock cycle T_{k-1}. $D(i-1, j)$ was calculated also in clock cycle T_{k-1} in cell $i-1$, and stored in $D1$. $D(i-1, j-1)$ was calculated in clock cycle T_{k-2} in cell $i-1$, and stored in $D1$. This value was shifted to $D2$ in clock cycle T_{k-1}. Thus, all values required to calculate $D(i, j)$ are stored in cell i or cell $i-1$.

When cell i receives θ or "," or "λ", then the latch $D1$ is initialized, and set to 0. It means that matching for a next text string is started.

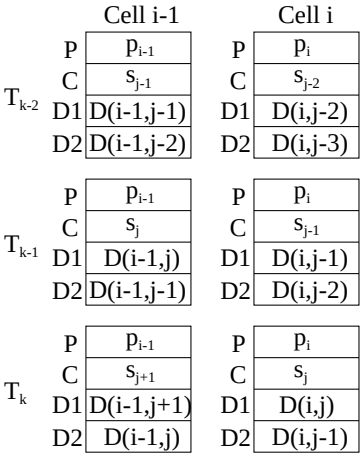

Figure 5. Behavior of the algorithm.

Figure 6 shows how string matching was processed by the proposed algorithm when $P = abc$ and $R = \{S_1, S_2, S_3\}$, where $S_1 = abb$, $S_2 = cba$, and $S_3 = acb$. From this figure, it is easy to understand that the proposed algorithm solves the multiple string matching problem in $O(L)$ clock cycles, where L is the total length of input string R.

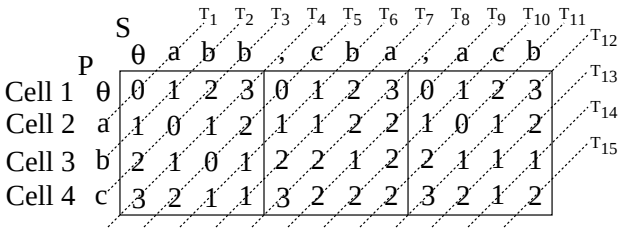

Figure 6. Example of string matching.

3.4 Memory Structure

As mentioned, each cell has its own copy of the edit cost matrix DM, in which matrix element $DM(a, b)$ represents the change cost from character a to character b, i.e., $\gamma(a \rightarrow b)$. When implementing the proposed algorithm on FPGAs, the edit cost matrices are implemented by using the block RAMs. When the edit cost matrix is symmetric, it is easy to reduce the total memory size to its half size with a simple address transformation.

In the algorithm shown above, the insert and delete costs are assumed to be fixed. However, it is easy to extend the algorithm so that arbitrary insert and delete costs are allowed by extending the edit cost matrix. We newly introduce a special symbol, denoted Λ to show the null character. Then, $DM(a, \Lambda)$ represents the delete cost of character a, and $DM(\Lambda, b)$ represents the insert cost of character b. Before starting string matching, each cell reads the delete cost from the edit cost matrix for a pattern character which the cell has in latch P, and stores it in a register. The leftmost cell reads the insert cost from the edit cost matrix for a text character which the cell receives from the input terminal, and stores it in another register. This value will be shifted right as the text string character is moved right. Note that the performance would be degraded if each cell reads the insert and delete costs from the edit cost matrix when $D(i, j)$ is calculated, since it would require three times of memory accesses.

4. Extensions of the Algorithm

In this paper, we propose two extensions of the basic hardware pattern matching algorithm presented in the previous section. The aim of the first extension is to reduce the size of memory DM. There are some applications, in which once a pattern string is set, the same pattern will be used for a fairly long period of time, that is, the number of text strings, m, is large. For such applications, the total memory size can be further reduced.

When a pattern P is given, each character in P is assigned to each cell. If the ith cell has character x in P, then this cell requires $DM(x, *)$ to calculate the edit distance. Thus, in this extension, called Ext1, each cell maintains the data in one row in DM, and those data are set before starting string matching.

The aim of the second extension, called Ext2, is to shorten the computation time. In the original algorithm, each cell compares one character in the input text S_i with one charac-

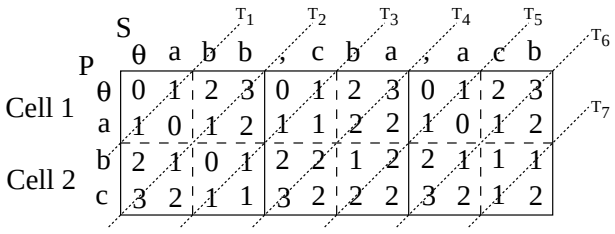

Figure 7. Extension of the algorithm.

ter in the pattern string P in each clock cycle. We modify the algorithm so that two characters in the text with two characters in the pattern simultaneously. Figure 7 shows how string matching was performed by this modified algorithm, when the input strings were the same as shown in Fig. 6. Compared to Fig. 6, it is easy to understand the total clock cycle of this extension was the half of the one for the original algorithm, although the behavior of each cell of this extension was much complicated than the original one. From Theorem 1, we introduced the recurrence relations to represent four matrix elements to be calculated in parallel. The cell structure was modified from the original one so that it calculates four matrix elements simultaneously. Due to lack of space, we omit its details.

5. FPGA Implementation

We have designed three hardware algorithms presented in this paper with Verilog-HDL, and implemented them on an FPGA board, which consists of a Xilinx FPGA chip VC4VLX100-11F1513. We have also developed a software program for solving the multiple string matching problem, and compared it with the proposed hardware algorithms implemented on the FPGA board. The software program was executed on a PC with a Pentium 4 3.6GHz CPU.

Table 1 shows the experimental results. In this table, "Software" shows the result of software program, and "Basic" shows the result of the basic hardware algorithm described in Section 3."Ext1" and "Ext2" show the results of two extended algorithms described in Section 4."#LUT", "Clock", "Time" and "Ratio" show the number of LUTs used to implement the circuit, the clock frequency of the FPGA chip, the execution time, and the speedup ratio of the hardware algorithms compared to the software program, respectively. The length of a pattern was 120 for the cases of "Software", "Basic" and "Ext1". For the case of "Ext2", the length of a pattern was set to 60. The total length of input strings to be matched with a pattern was set to $120,000$.

Table 1. Experimental results.

Algorithm	#LUT	Clock [MHz]	Time [μS]	Ratio
Software	–	–	1,040,000	1
Basic	12248	165	729	1427
Ext1	16331	224	537	1937
Ext2	43965	169	355	2930

From the experimental results, we see that the proposed hardware string matching engines drastically outperformed the software program. Compared to the "Basic" algorithm, performance of "Ext1" was improved, since the circuit of the latter was simpler than the former one, and thus the critical path of the latter was shorter than the former. Performance of the "Ext2" was twice of the "Basic", since the number of clock cycles of the former was the half of the latter.

We would also like to point out that, since the proposed hardware algorithms have an simple one-dimensional systolic architecture, it is easy to implement the algorithms for longer patterns by connecting multiple FPGA chips.

6. Conclusion

We have proposed a hardware algorithm for the string matching problem, and implemented it on an FPGA. We have also shown several extensions of the proposed algorithm. Experimental results showed the effectiveness of the proposed algorithms. Future research includes the development of hardware algorithms for different kinds of string matching. In particular, besides the string-to-string correction problem, there are many other problems, for which dynamic programming algorithms have been known in bioinformatics [4]. It is interesting and important to develop hardware algorithms for those problems.

Acknowledgments

This research was supported in part by Grant-in-Aid for Scientific Research (C)(No.20500054) from Japan Society for the Promotion of Science.

References

[1] J.Aoe (eds.), *Computer Algorithms: String Pattern Matching Strategies*, IEEE Computer Society Press, 1994.

[2] M.J.Foster, H.T.Kung, The design of special-purpose VLSI chips, *IEEE Computer*, 13, 1, pp.26–40, 1980.

[3] P.A.V.Hall, G.R.Dowling, Approximate string matching, *ACM Computing Surveys*, 12, 4, pp.381–402, 1980.

[4] N.C.Jones, P.A.Pevzner, *An Introduction to Bioinformatics Algorithms*, The MIT Press, 2004.

[5] D.E.Knuth, J.H.Morris, V.R.Pratt, Fast pattern matching in strings, *SIAM Journal on Computing*, 6, 2, pp.323–350, 1977.

[6] C.Mead, L.Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[7] A.Mukhopadhyay, Hardware algorithms for nonnumeric computation, *IEEE Trans. Computers*, C-28, 6, pp.384-394, 1979.

[8] R.A.Wagner, M.J.Fischer, The string-to-string correction problem, *Journal of ACM*, 21, 1, pp.168–173, 1974.

[9] T.Kikuno, N.Yoshida, S.Wakabayashi, Hardware algorithms for computing longest common subsequence, *Trans. IECE*, J65-D, 8, pp.997–1004, 1982, in Japanese.

[10] C.W.Yu, K.H.Kwong, K.H.Lee, P.H.W.Leong, A Smith-Waterman systolic cell, *Proc. FPL2003*, LNCS 2778, pp.375–384, 2003.