ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **12**, Number 1, 2009, 51–67

A Hardware Algorithm for The Exact Subsequence Matching Problem in DNA Strings

Octavian CREŢ[1], Zsolt MATHE[2], Paul CIOBANU[1], Sonia MĂRGINEAN[1], Adrian DĂRĂBANT[3]

[1]Computer Science Department,
Technical University of Cluj-Napoca
26-28 George Bariţiu street, Cluj-Napoca, Romania
E-mail: {Octavian.Cret, Paul.Ciobanu, Sonia.Marginean}@cs.utcluj.ro

[2]INRIA Futurs, team Alchemy
Parc Club Orsay Université, rue Jacques Monod 91893 Orsay
E-mail: Zsolt.Mathe@ens - lyon.org

[3]Computer Science Department,
Babeş-Bolyai University of Cluj-Napoca
1 M. Kogălniceanu street, Cluj-Napoca, Romania
E-mail: dadi@cs.ubbcluj.ro

Abstract. This paper introduces an algorithm for DNA string detection and proposes an efficient hardware implementation of it on FPGA devices. Its main application field is intended to be the detection of intron and exon strings in DNA chains, but its applicability is not limited to Genetics. The *GenDiv* algorithm is based on the dynamic programming method. For the software implementation, the algorithm's complexity is $O(m \cdot n)$, where m and n are the sizes of the two DNA strings being processed; in hardware, after a few adaptations that are presented, the algorithm can be implemented in a systolic array and its running time becomes linear $O(m + n)$. Simultaneously, the necessary circuitry resources in the FPGA chip are also showing a linear trade-off. The simulations and tests that have been run show a gain of speed of several orders of magnitude of the hardware implementation over the software one.

Key words: intron, exon, systolic array, FPGA, VHDL.

1. Introduction

In any practical analysis of a DNA fragment, the analysis system is based on a DNA analyzer (sequencer): a device that reads the nucleotide sequence (A, C, G or T – to represent them, a 4-symbol alphabet is thus necessary) from the analyzed DNA fragment. A sequence of 3 nucleotides is called *codon*, and it encodes an *amino acid*; a sequence of codons (varying from a few hundreds to a few thousands) forms a *gene*. If each amino acid is encoded by a symbol, a 20-symbol alphabet is necessary to represent them: {K, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y}. In practice, both alphabets can be used, according to the analysis level that is targeted.

A gene encodes a *protein*. DNA and protein sequences are stored in genomic databases. SWISS-PROT and TREMBL [1], for example, are two well-known protein sequence databases, while GenBank [2] and EMBL [3] are two well-known DNA databases.

One of the most frequent operations consists of highlighting the presence or the absence of a specific DNA fragment (DNA encoding strain) inside a much longer DNA sequence (typically obtained from a database). In many cases, the analyzed fragment is not found contiguously inside the DNA chain, but a sequence of introns and exons is identified. An intron is a region of DNA that does not code for the synthesis of a protein (to be more precise, an intron does not contain codons, since introns typically carry important signals, which influence the splicing machinery and impact construction of the final mRNA [4]), while an exon is a region of a gene containing DNA that codes for a protein. Figure 1 shows a small example of the above presented concepts.

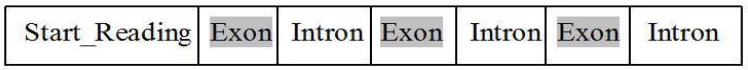


Fig. 1. Structure of a DNA sequence composed of introns and exons.

The discontiguity of specific encoding strains inside the DNA sequence is due either to the presence of non-encoding strains (introns) or to the presence of foreign encoding strains (mutations, translocations, etc.). The length of a DNA encoding strain varies according to the aim followed by the researcher, typically between 3 nucleotides and a few genes. The length of a DNA sequence from a database varies with the length of the analyzed zone, and can reach thousands of genes.

'Exon trapping' or 'gene trapping' is a molecular biology technique that exploits the existence of the intron-exon splicing to find new genes. After identifying the introns and exons regions in the DNA chain, one can move on to the next stage, which consists of establishing the resulting positive diagnosis in a large variety of diseases (at the human being), or of other operations in case other eukaryotic genomes are analyzed.

With today's devices, this operation takes, in average, a few days, depending mainly on the efficiency of the matching algorithm that is used, and secondarily on the length of the analyzed chain (DNA encoding strain) and of the region from the DNA sequence inside which the search is performed. The problem's acuity is increased

by the fact that genomic data grow exponentially ($\times$ 2 every 12–15 months) and at a faster rate than computing power (according to Moore's law computing power doubles every 18 months) [5].

Applications are extremely diverse: prenatal diagnosis (marital advice etc.), genetic screenings, calculation of a so called "genetic risk", finding genetic alterations produced by existing modifications, etc.

As one can notice, this problem can be formulated as a string matching (string identification) one, and its solution is portable in many other scientific fields. The original algorithm proposed here solves the problem in $O(m{\cdot}n)$ time; when optimized for a hardware implementation in FPGA devices, it yields linear performance, both for the runtime and the space (physical circuitry) involved.

The paper is structured as follows: in section II the algorithm is presented, together with the results of its software implementation. Section III treats the adaptations applied to the algorithm for its hardware implementation and presents some performance considerations. In section IV we present the experimental results, and in the Conclusions section we indicate a series of applications of this algorithm.

2. The GenDiv algorithm

The GenDiv algorithm is a string comparison algorithm based on the dynamic programming (DP) method, which was initially proposed by R. Bellman in 1940. DP methods for problems involving Genetics were first used by Needleman and Wunsch [6] and Smith and Waterman [7] in 1970 and 1981 respectively. It implies building an array, whose columns are the elements of the S string, and whose rows are the elements of the T string.

DP methods probably provide the best way to find similarly ties between two strings. However, their major drawback comes from their quadratic computation time, preventing them from being used to compare very large strings such as complete genomes. Efficient heuristics for similarity searches were proposed by Pearson (FASTA, [8]) and Altschul (BLAST, [9]). Although today there are numerous other heuristic algorithms that outperform it, the BLAST software remains the reference algorithm. [5]

Our problem is not a similarity search, it is a subpart search. The Smith-Waterman algorithm is not appropriate for solving this problem, as it only generates the *edit distance*, which plays no role in the detection of the slices (introns and exons), which is the problem of interest in this case. Our problem can be generalized as follows: given two strings $S_{1\ldots n}$ and $T_{1\ldots m}$, the former having n and the latter m characters respectively, find the optimal splicing of S as part of T. In other words, it is required to find the string S inside the string T, knowing that the elements of S are possibly not contiguously spread in the string T.

To solve this problem, some software algorithms are appropriate, namely those based on suffix trees, or hash tables. These algorithms have a linear runtime. For instance, in the suffix tree-based method, any string of length m can be degenerated into m suffixes, and these suffixes can be stored in a suffix tree. Creating this struc-

ture requires time $O(m)$ and searching for a pattern in it requires time $O(n)$, where n is the length of the pattern. These two properties make the suffix tree an appealing structure for a diverse range of bioinformatics applications including: multiple genome alignment [10]; selection of signature oligonucleotides for DNA arrays [11]; and identification of sequence repeats [12]. The problem with these algorithms is the fact that they are inappropriate for a hardware implementation. It is indeed difficult to generate in hardware a new suffix tree for each new T string, in case of multiple runs of the algorithm.

The main advantage of our algorithm is that it is adequate for a hardware implementation, while maintaining a linear execution time $O(m)$. To take a simple example of our problem, consider the following two strings: T = "one of the problems of being a pioneer is you always make mistakes", S = "the problem is you wait". For simplicity we remove spaces, without loss of the generality of the problem.

The solution for finding S inside T consists of 5 parts: E_1 = "theproblem", E_2 = "isyou", E_3 = "wa", E_4 = "i", and E_5 = "t". If we highlight the five parts in the T string: T=oneofTHEPROBLEMsofbeingapioneerISYOUalWAysmakemIsTakes.

This example was given in natural language, to facilitate by analogy the understanding of the problem's nature. In Genetics, S corresponds to the DNA encoding strain, while T corresponds to the DNA sequence that is taken from a genes database. E1-E5 correspond to the exons, while the six substrings between the exons correspond to the introns (I_1 = "oneof", I_2 = "sofbeingapioneer", I_3 = "al", I_4 = "ysmakem", I_5 = "s", and I_6 = "akes"). Of course, in Genetics, S and T contain only the four characters representing the nucleotides {A, C, T, G} or the 20 elements of the amino acids alphabet {K, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y}.

The GenDiv algorithm uses two iterators, j and i, over S and T respectively, and uses two variables:

– The first variable is simply called a and it indicates in how many parts (or slices) the string $S_{1...j}$ is found inside the string $T_{1...i}$.

– The second variable is called *dist* and it shows, in case S_j is not equal to T_i, at which distance from the current value of i the previous solution is located. For instance, if $a = 1$ and $dist = 3$, this means that $S_{1...j}$ is found inside $T_{1...i-3}$ in one slice.

The following example illustrates the meaning of a and *dist*. Suppose T = "tabbbxxcy" and S = "abc". When $i = 6$ and $j = 2$, the string $S_{1...2}$ = "ab" is found inside $T_{1...6}$ = "tabbbx" in one piece: $a = 1$, and $dist = 3$, because this solution appeared at $i = 3$; at $T_{1...7}$ = "tabbbxx" we'll have $a = 1$, and $dist = 4$, and so on. When $i = 8$ and $j = 3$, the string $S_{1...3}$ = "abc" is found inside the string $T_{1...8}$ = "tabbbxxc" in two pieces: $a = 2$, and $dist = 0$; at $T_{1...9}$ = "tabbbxxcy" we'll have $a = 2$, and $dist = 1$.

GenDiv is DP-based, so each new element of the $m \times n$ array is computed based on the previously obtained ones, when the i (row) and the j (column) iterators start from the upper left corner. In this algorithm, the S and T strings contain each an additional dummy element, S_0 and T_0 respectively. As a consequence, the array contains one more row and one more column, as follows:

– In the additional row, all the a parameters are initialized with MAX_INTEGER

$(+\infty)$ and all the *dist* parameters with 0 (notation: $\infty(0)$).

– In the additional column, all the a parameters are initialized with 1 and all the *dist* parameters with 0 (notation: 1(0)).

– In the upper left corner, the element $a_{0,0}$ has the a parameter initialized with 1 and the *dist* parameter with 0 (notation: 1(0)).

The algorithm computes the new a and *dist* values by the following rules (Fig. 2):

When a match is found between S_j and T_i ($S_j = T_i$) two possibilities can arise:

– This new match can prolong an existing slice, by appending the new element S_j to the chain (if $dist_{i-1,j-1} = 0$), OR a new chain will start, with S_j at its beginning (if $dist_{i-1,j-1} > 0$, case 1.1 in Fig. 2);

– This new match can neither prolong an existing slice nor start a new slice – in this case, *dist* will be incremented and a will be copied from the above row (case 1.2 in Fig. 2).

If S_j and T_i differ ($S_j <> T_i$), *dist* will be incremented and a will be copied from the above row (case 2 in Fig. 2).

For $i = 1$ **to** m

For $j = 1$ **to** n

Compare T_i **with** S_j

1. **If** $T_i = S_j$

1.1 **if** $a_{i-1,j-1} + (dist_{i-1,j-1} > 0) <= a_{i-1,j}$, **then**

$a_{i,j} = a_{i-1,j-1} + (dist_{i-1,j-1} > 0)$

$dist_{i,j} = 0$

1.2 **if** $a_{i-1,j} < a_{i-1,j-1}$, **then**

$a_{i,j} = a_{i-1,j}$

$dist_{i,j} = dist_{i-1,j} + 1$

2. If $T_i <> S_j$, **then**

$a_{i,j} = a_{i-1,j}$

$dist_{i,j} = dist_{i-1,j} + 1$

End for;

End for;

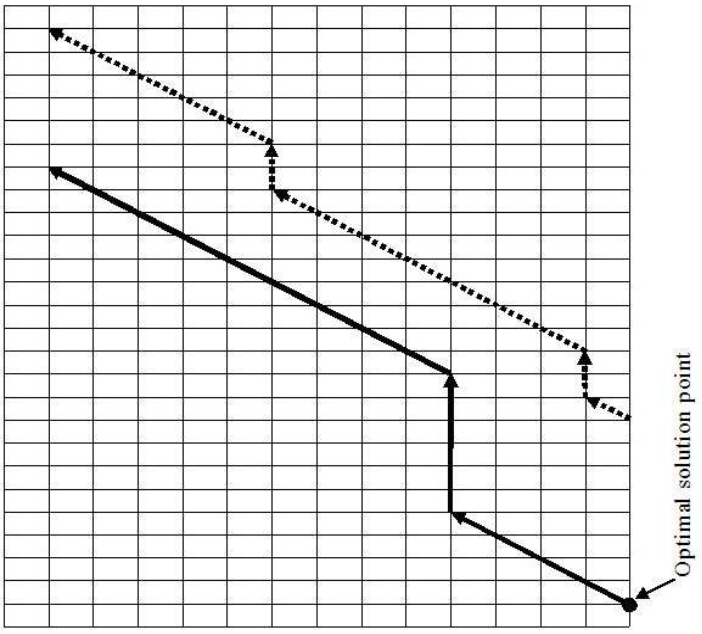
Fig. 2. The GenDiv algorithm.

Figure 3a shows the array obtained for a bigger example: T = "*abcdaxabcdabcdaxefxxxbcdef*" and S = "*abcdabcdabcde*". The format of the numbers in the array is: a(*dist*). In this case, two solutions can be extracted: *ABCDAxaBCDABCDaxEfxxxbcdef* (in 3 slices – this solution is highlighted in italics in the array) and *abcdaxABCDABCDAxefxxxBCDEf* (in 2 slices – this solution is optimal and is highlighted in bold characters). The optimal solution appears like a normal poly line, while the other one is highlighted as a dashed one in Fig. 3b.

***S* string**

***T* string**	0	1 A	2 B	3 C	4 D	5 A	6 B	7 C	8 D	9 A	10 B	11 C	12 D	13 E
0	1(0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞(0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
1A	1(0)	*1(0)*	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
2B	1(0)	1(1)	*1(0)*	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
3C	1(0)	1(2)	1(1)	*1(0)*	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
4D	1(0)	1(3)	1(2)	1(1)	*1(0)*	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
5A	1(0)	1(0)	1(3)	1(2)	1(1)	*1(0)*	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
6X	1(0)	1(1)	1(4)	1(3)	1(2)	1(1)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
7A	1(0)	**1(0)**	1(5)	1(4)	1(3)	1(2)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
8B	1(0)	1(1)	**1(0)**	1(5)	1(4)	1(3)	*2(0)*	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
9C	1(0)	1(2)	1(1)	**1(0)**	1(5)	1(4)	2(1)	*2(0)*	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
10D	1(0)	1(3)	1(2)	1(1)	**1(0)**	1(5)	2(2)	2(1)	*2(0)*	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
11A	1(0)	1(0)	1(3)	1(2)	1(1)	**1(0)**	2(3)	2(2)	2(1)	*2(0)*	∞ (0)	∞ (0)	∞ (0)	∞ (0)
12B	1(0)	1(1)	1(0)	1(3)	1(2)	1(1)	**1(0)**	2(3)	2(2)	2(1)	*2(0)*	∞ (0)	∞ (0)	∞ (0)
13C	1(0)	1(2)	1(1)	1(0)	1(3)	1(2)	1(1)	**1(0)**	2(3)	2(2)	2(1)	*2(0)*	∞ (0)	∞ (0)
14D	1(0)	1(3)	1(2)	1(1)	1(0)	1(3)	1(2)	1(1)	**1(0)**	2(3)	2(2)	2(1)	*2(0)*	∞ (0)
15A	1(0)	1(0)	1(3)	1(2)	1(1)	1(0)	1(3)	1(2)	1(1)	**1(0)**	2(3)	2(2)	2(1)	∞ (0)
16X	1(0)	1(1)	1(4)	1(3)	1(2)	1(1)	1(4)	1(3)	1(2)	1(1)	2(4)	2(3)	2(2)	∞ (0)
17E	1(0)	1(2)	1(5)	1(4)	1(3)	1(2)	1(5)	1(4)	1(3)	1(2)	2(5)	2(4)	2(3)	*3(0)*
18F	1(0)	1(3)	1(6)	1(5)	1(4)	1(3)	1(6)	1(5)	1(4)	1(3)	2(6)	2(5)	2(4)	3(1)
19X	1(0)	1(4)	1(7)	1(6)	1(5)	1(4)	1(7)	1(6)	1(5)	1(4)	2(7)	2(6)	2(5)	3(2)
20X	1(0)	1(5)	1(8)	1(7)	1(6)	1(5)	1(8)	1(7)	1(6)	1(5)	2(8)	2(7)	2(6)	3(3)
21X	1(0)	1(6)	1(9)	1(8)	1(7)	1(6)	1(9)	1(8)	1(7)	1(6)	2(9)	2(8)	2(7)	3(4)
22B	1(0)	1(7)	1(10)	1(9)	1(8)	1(7)	1(10)	1(9)	1(8)	1(7)	**2(0)**	2(9)	2(8)	3(5)
23C	1(0)	1(8)	1(11)	1(10)	1(9)	1(8)	1(11)	1(10)	1(9)	1(8)	2(1)	**2(0)**	2(9)	3(6)
24D	1(0)	1(9)	1(12)	1(11)	1(10)	1(9)	1(12)	1(11)	1(10)	1(9)	2(2)	2(1)	**2(0)**	3(7)
25E	1(0)	1(10)	1(13)	1(12)	1(11)	1(10)	1(13)	1(12)	1(11)	1(10)	2(3)	2(2)	2(1)	**2(0)**
26F	1(0)	1(11)	1(14)	1(13)	1(12)	1(11)	1(14)	1(13)	1(12)	1(11)	2(4)	2(3)	2(2)	2(1)

a)



b)

Fig. 3. a) The Gendiv array for T = "abcdaxabcdabcdaxefxxxbcdef" and S = "abcdabcdabcde"; b) The method for extracting solutions from it.

As the example shows, the algorithm generates all the solutions, but only the optimal one will be considered interesting. To extract the solution, the array must be processed starting with the last column: we must find the smallest a value in the last column for which *dist* is 0. This array element, if it exists, indicates the end of the last chain; if it doesn't exist, this means no solution exists (the string S can not be found inside the string T). From this element, the array must be inspected towards the left, in diagonal (if the element $dist_{i-1,j-1} = 0$) or towards the top of the array (if the element $dist_{i-1,j-1} \neq 0$).

As one can see from its description, the GenDiv algorithm's complexity is $O(m{\cdot}n)$. As mentioned in the Introduction, the values of m and n can vary significantly from a problem to another, but we considered here an average, real life case.

Table 1 shows the execution times obtained by the algorithm compiled with gcc with the –fast compiler option, for an Intel CoreDuo2 architecture. The results are slightly better than the ones obtained with the –O3 option. The table includes n, m and *variance*, a parameter that shows how many different characters (amino acids or nucleotides) are used in the searched sequence. 100% means a use of all the 20 symbols, while 20% means use of only the 4 basic nucleotides. The variance issue is more important from the point of view of the biological experiments.

The test results shown in Table 1 come from automated measurements which ran one hundred random samples with different seeds for each test case. Out of these results the "90% mean" interval [13] was built: the fastest five and the slowest five runs were discarded and the arithmetic mean of the remaining 90 percent yielded the shown time.

Better results can be obtained if the source code is written so as to run optimally on a given processor, or if written so as to be executed on a supercomputer, but our goal is to compare our hardware implementation with the results obtained by a program running on a performing x86 architecture. Some additional results are shown in Table 2, where the software implementation running times are compared to those of the hardware implementation.

Table 1. Performance of the software implementation

Gene sequence length (n)	**Gene database length (m)**	**Variance (%)**	**Execution Time (seconds)**
10	100,000	20%	0.532
10	100,000	100%	0.547
30	100,000	20%	1.578
30	100,000	100%	1.620
50	10,000	20%	0.093
50	10,000	100%	0.110
256	1,000,000	20%	27.958
256	1,000,000	100%	28.312
512	1,000,000	20%	42.176
512	1,000,000	100%	43.374

3. Hardware implementation

Due to its intrinsic parallel nature, this problem can be solved in hardware using some basic concepts as systolic architecture and dynamic programming.

In a parallel implementation, the positive slope diagonal entries of Fig. 4a can be computed simultaneously. The data dependencies deduced from the GenDiv algorithm can be graphically represented like in Fig. 5, meaning that entry $a_{i,j}$ in the array can only be calculated if the $a_{i-1,j}$ and $a_{i-1,j-1}$ values are already known, and so the computation of the array elements spreads out.

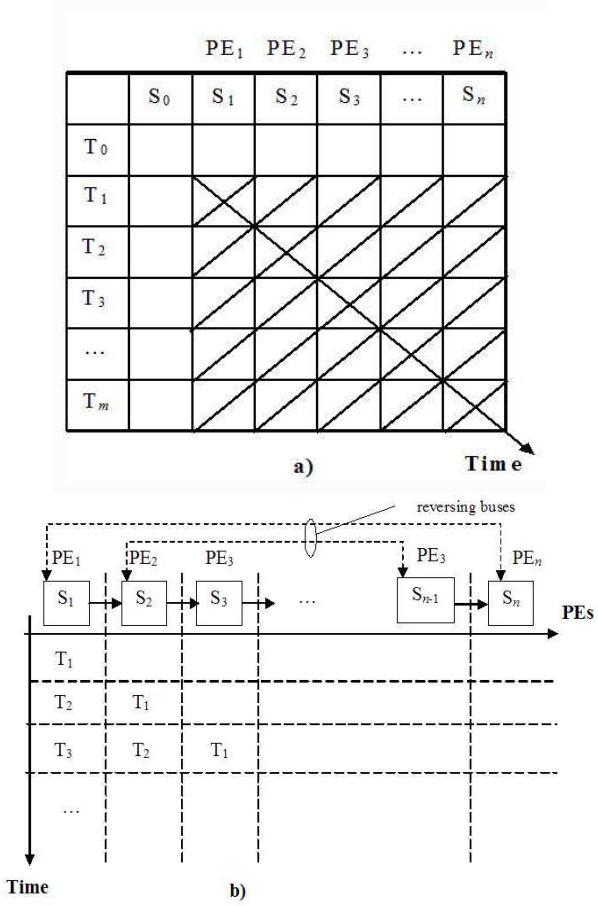


Fig. 4 a) Data dependencies in the systolic array (entries computed in parallel, on the Time axis); b) The systolic operating mode.

The systolic array (SA) is composed of identical PEs. In this implementation, it can contain up to 512 PEs, the only limitation being the capacity of the FPGA device which constitutes its physical support. As we had a VIRTEX2PRO30 device available, the number of PEs was limited to 512, but in a more recent FPGA device,

like Virtex5, we can reach up to 2048 PEs. Also, the operational frequency will be much greater (400 MHz in a Virtex5 device). Moreover, if a bigger S string is needed, the architecture can be implemented on several FPGA chips. The size of the implementation can easily be adjusted due to the parameterizable VHDL code that was used for describing its components.

3.1. Functioning of the Systolic Architecture

In the systolic architecture, the characters of the$S$string are stored in the PEs, while the characters of the other string, T, are fed serially into the array (Fig. 4b). Thus, a comparison between pairs of characters (one belonging to the S, the other belonging to the$T$string) is done on each machine cycle.

In each machine cycle, a PE computes new $a_{i,j}$ and $dist_{i,j}$ values, based on the values that it stores locally ($a_{i-1,j}$ and $dist_{i-1,j}$) and on the values $a_{i-1,j-1}$ and $dist_{i-1,j-1}$ that it receives from its left neighbor. The arrows between the PEs in Fig. 4b are buses carrying the following information:

– In *Phase* 0, the characters from the S string.

– In the next *Phases*, the values of $a_{i-1,j-1}$ and $dist_{i-1,j-1}$.

When a PE finishes computing the $a_{i,j}$ and $dist_{i,j}$ values, it will "move on" to the following diagonal. It will have to store the values of its current $a_{i,j}$ and $dist_{i,j}$ parameters, that will become the $a_{i-1,j}$ and $dist_{i-1,j}$ parameters in the next machine cycle; the $a_{i-1,j-1}$ and $dist_{i-1,j-1}$values will be received from its left neighbor, or from outside the array (if this is the very first PE in the array).

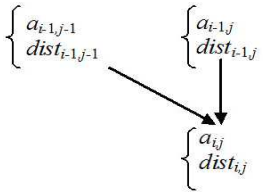


Fig. 5. Data dependencies in the GenDiv array.

3.2. Differences and Necessary Adaptations

When moving from the software version of the algorithm to its hardware implementation, a series of improvements and adaptations must be done.

– *Initial row values*: for the first row of the GenDiv array, all the $a_{i-1,j}$ and $dist_{i-1,j}$ values are initialized with $\infty(0)$.

– *Initial column values*: for the first column of the GenDiv array, an additional PE (PE_0) will be added at the left of the SA. PE_0 will be initialized with the *constant* values of 1(0), which play the role of $a_{i-1,j-1}$ and $dist_{i-1,j-1}$ for all the elements on the Column 1 of the GenDiv array.

– Dist *values*: in the GenDiv algorithm, one can notice that the *dist* parameter plays a role only in two cases: if it is 0 or greater than 0. The algorithm can be modified to simply set or reset *dist*, because the concept of "distance" is helpful only

for the algorithm's understanding (for finding a solution it is enough to know whether *dist* is 0 or greater). In the hardware implementation, *dist* will become a simple flag (a Flip-Flop) that can be either 0 or 1.

– *Storing the GenDiv array*: The major problem that arises in the hardware implementation is the fact that it is impossible to store the GenDiv array inside the FPGA chip, due to space limitations. But without storing the GenDiv array, how can the solution be extracted? That was one of the major challenges of this research and its solution is presented below.

3.3. Hardware Architecture and Operating Mode

The S string is stored in the SA's PEs, while the T string is stored in a large FIFO memory (built from the Virtex device's BlockRAMs). In each machine cycle, a new set of $a_{i,j}$ and $dist_{i,j}$ values are obtained in each PE.

The output of the last PE (PE_n) will be connected to a supplementary module PE_{n+1} that computes the minimal a value for which dist is 0 ($a_{min}(0)$) and also memorizes the position i (inside the T string) where this minimal value was encountered. This module is relatively easy to design.

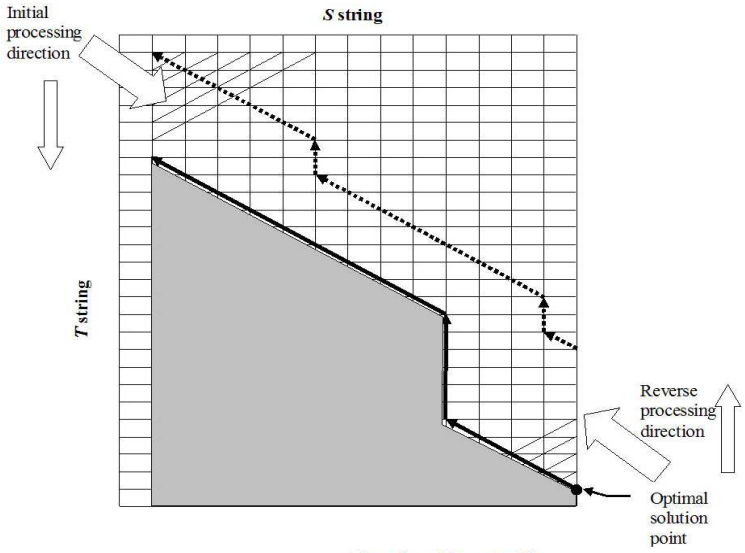
In the first phase, the T string is fed serially (element by element) in the SA. At the end of this phase, PE_{n+1} (the Minimum Generator module) will contain $a_{min}(0)$ and i_{min}, which indicates if there is a solution (a_{min} must be less than ∞) and, if the solution exists, this means that the string S can be found inside T in a_{min} slices. In the example from Fig. 3, $a_{min} = 2$ and $i_{min} = 25$, which is a better solution than $a = 3$ and $i = 17$.

To exploit this information, we must re-feed the T string in the SA, but in reverse order. Also, the SA will contain the S string's elements in reverse order (PE_1 will contain S_n, PE_2 will contain $S_{n-1}\ldots$, PE_n will contain S_1). The idea is: if $S_{1\ldots n}$ was found in $T_{1\ldots imin}$, then $S_{n\ldots 1}$ will also be found in $T_{imin\ldots 1}$ in the same number of slices. Since the first solution was optimal, the solution of the reverse search will also be optimal, but not necessarily identical to the first one! Because of the GenDiv algorithm's nature, the processing tends to yield a solution that is "agglomerated" towards the first element; in the reverse processing, the solution will be "agglomerated" towards the last element. For example, if we process T = "tabbbcy" and S = "abc", in the direct processing the solution is "tABbbCy", while in the reverse processing the solution is "yCBbbAt".

So, there will be a second phase, in which the SA is inverted (the elements of the S string are interchanged, from the head – first element – to the tail – last element) and re-initialized – this can easily be done in one clock cycle. Simultaneously, the T string will be prepared to be fed in the SA in reverse order – this can be done by simply loading i_{min} in a reverse counter that will also be the Address Register of the FIFO where T is stored.

The third phase is again a processing phase during which a new GenDiv array is obtained. The PEs will now monitor the first appearance of a 0 value of the dist parameter; on the first occurrence, each PE will memorize the i value where dist was 0 for the first time. This way, we know the coordinates of all the points that constitute

the solution poly line: i, j (the PE itself can store its position inside the S string) and also a (this indicates on which segment is located the current S element).



S string (inverted)

T string (inverted)	0	1 E	2 D	3 C	4 B	5 A	6 D	7 C	8 B	9 A	10 D	11 C	12 B	13 A
0	1(0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
1E	1(0)	**1(0)**	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
2D	1(0)	1(1)	**1(0)**	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
3C	1(0)	1(2)	1(1)	**1(0)**	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
4B	1(0)	1(3)	1(2)	1(1)	**1(0)**	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
5X	1(0)	1(4)	1(3)	1(2)	1(1)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
6X	1(0)	1(5)	1(4)	1(3)	1(2)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
7X	1(0)	1(6)	1(5)	1(4)	1(3)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
8F	1(0)	1(7)	1(6)	1(5)	1(4)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
9E	1(0)	1(0)	1(7)	1(6)	1(5)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
10X	1(0)	1(1)	1(8)	1(7)	1(6)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
11A	1(0)	1(2)	1(9)	1(8)	1(7)	**2(0)**	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
12D	1(0)	1(3)	1(10)	1(9)	1(8)	2(1)	**2(0)**	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
13C	1(0)	1(4)	1(11)	1(10)	1(9)	2(2)	2(1)	**2(0)**	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
14B	1(0)	1(5)	1(12)	1(11)	1(10)	2(3)	2(2)	2(1)	**2(0)**	∞ (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
15A	1(0)	1(6)	1(13)	1(12)	1(11)	2(0)	2(3)	2(2)	2(1)	**2(0)**	∞ (0)	∞ (0)	∞ (0)	∞ (0)
16D	1(0)	1(7)	1(14)	1(13)	1(12)	2(1)	2(0)	2(3)	2(2)	2(1)	**2(0)**	∞ (0)	∞ (0)	∞ (0)
17C	1(0)	1(8)	1(15)	1(14)	1(13)	2(2)	2(1)	2(0)	2(3)	2(2)	2(1)	**2(0)**	∞ (0)	∞ (0)
18B	1(0)	1(9)	1(16)	1(15)	1(14)	2(3)	2(2)	2(1)	2(0)	2(3)	2(2)	2(1)	**2(0)**	∞ (0)
19A	1(0)	1(10)	1(17)	1(16)	1(15)	2(0)	2(3)	2(2)	2(1)	2(0)	2(3)	2(2)	2(1)	**2(0)**
20X	1(0)	1(11)	1(18)	1(17)	1(16)	2(1)	2(4)	2(3)	2(2)	2(1)	2(4)	2(3)	2(2)	2(1)
21A	1(0)	1(12)	1(19)	1(18)	1(17)	2(0)	2(5)	2(4)	2(3)	2(2)	2(5)	2(4)	2(3)	2(2)
22D	1(0)	1(13)	1(20)	1(19)	1(18)	2(1)	2(0)	2(5)	2(4)	2(3)	2(6)	2(5)	2(4)	2(3)
23C	1(0)	1(14)	1(21)	1(20)	1(19)	2(2)	2(1)	2(0)	2(5)	2(4)	2(7)	2(6)	2(5)	2(4)
24B	1(0)	1(15)	1(22)	1(21)	1(20)	2(3)	2(2)	2(1)	2(0)	2(5)	2(8)	2(7)	2(6)	2(5)
25A	1(0)	1(16)	1(23)	1(22)	1(21)	2(0)	2(3)	2(2)	2(1)	2(0)	2(9)	2(8)	2(7)	2(6)

Fig. 6. Demonstration of the proposed solution extracting method: a) Principle; b) GenDiv array example.

To understand the correctness of the solution, we must examine Fig. 6. Since now the processing direction is inverted, no solution line can appear in the grayed region (all elements are $\infty(0)$). Suppose there is another solution starting above the Optimal Solution Point. Because of the algorithm's nature, the points that constitute the non-optimal poly line (the points where the dist parameter is 0), on each column, will be obtained in a further processing step than those obtained on the optimal poly line.

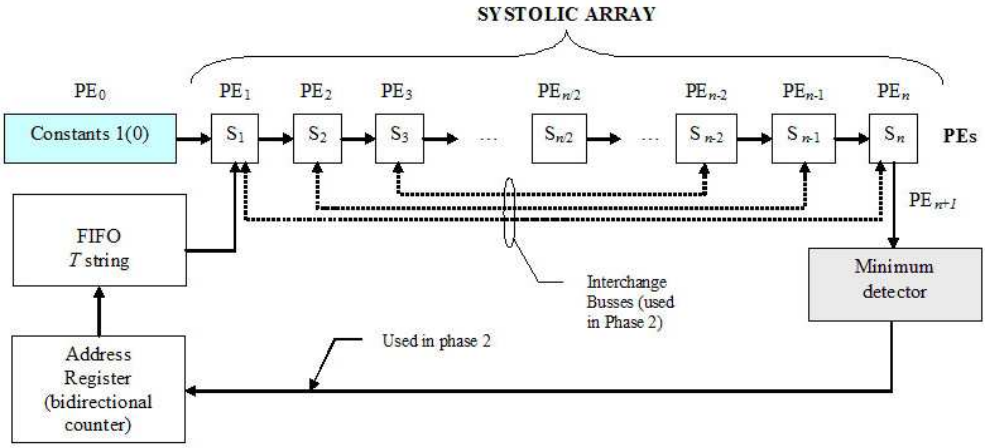


Fig. 7. Overall GenDiv architecture.

In the fourth phase (final), all data from the PEs are shifted out of the SA and returned to the user as the solution. Given the above considerations, the architecture of the proposed hardware implementation is the one shown in Fig. 7. The detailed interconnection between two adjacent PEs is shown in Fig. 8.

The structure of a PE is presented in Fig. 9. For a 20-character alphabet, the elements of the S and T strings are stored in registers sized on 5 bits, if the alphabet contains only the four nucleotides, then the thick lines are 2 bits wide. To simplify the schematic, CLOCK was not figured.

The S_REG stores not only the value of the S string's elements, but also their index inside S (called j in the GenDiv algorithm). So, buses S_IN, S_REV_IN, S_OUT and S_REV_OUT carry also this information (the j index).

The PEs start working progressively, controlled by the Flip-Flops on the Compute line. These are initialized with 0, then a logic '1' is shifted on that path, thus giving clock enables to the registers inside the PE.

During Phase 2, the S string is reversed; this process occurs on the S_REV_IN S_REV_OUT path.

During Phase 3, the Counter indicates the index of the T string which is being processed in the current PE. When $dist_{i,j}$ becomes 0 for the first time, the Counter will be stopped (through a Set-Reset Latch), thus storing the index value, which is part of the final solution. During Phases 0-2, the Counter's output is meaningless.

In the final Phase the solution is extracted using the Extract control signal, which allows data to be shifted outside the SA using the i_IN i_OUT path.

The minimum detector module is a synchronous component whose construction is straightforward. It uses just a comparator and a data register with Parallel Load (PL) and Clock Enable (CE). If the a values are read synchronously one in each clock cycle, on the MIN output the module will yield the minimal a value that was read, but only if the corresponding dist value was 0 (CE is active low). Its structure is shown in Fig. 10a (for the case when there are 20 characters in the alphabet).

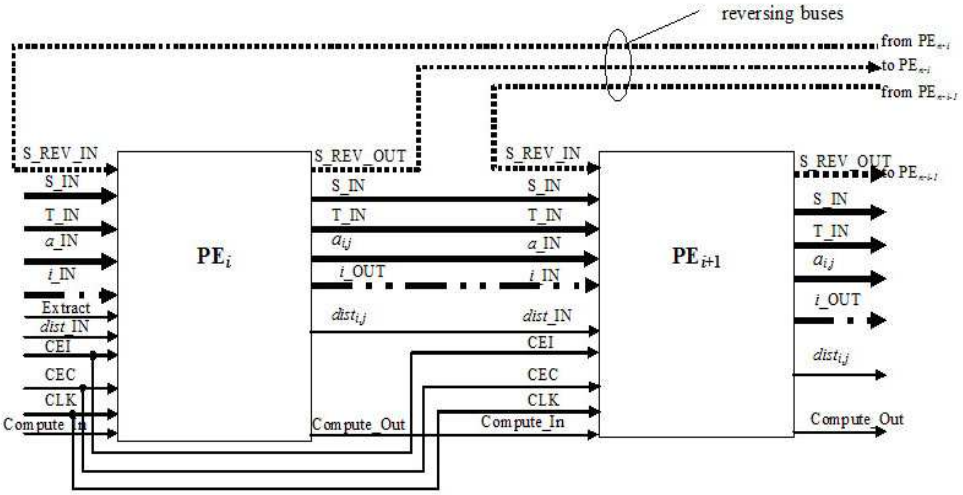


Fig. 8. Two consecutive PEs and their connections.

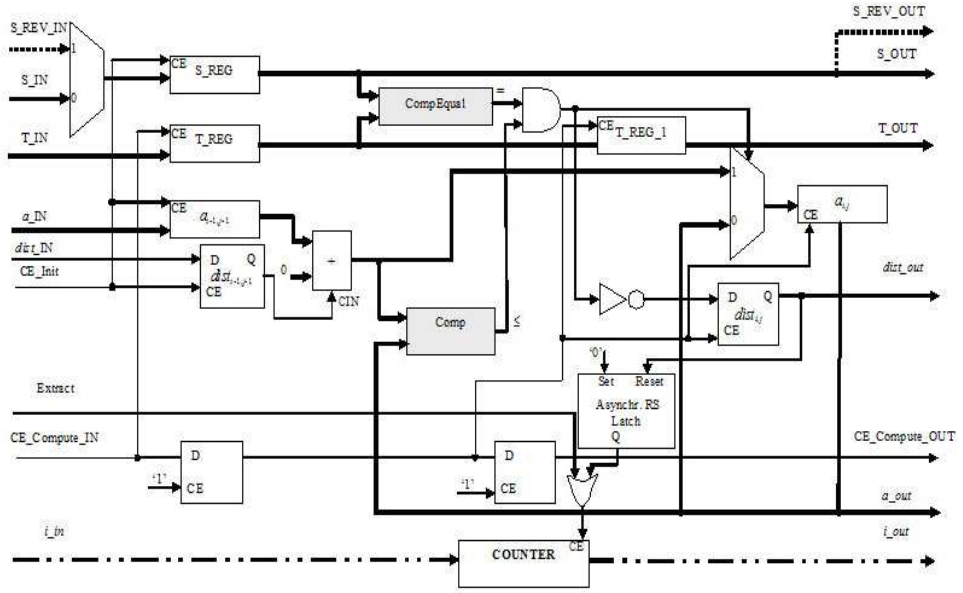


Fig. 9. The structure of a PE.

As shown in Fig. 9, the PE is pipelined in two stages: in the first clock cycle, a new T_j value is loaded; then, in the next clock cycle, the PE produces its result.

Figure 10b shows the structure of a module that detects the first two minimums. As one can notice, the complexity of the minimum detector modules grows exponentially from the space (active circuitry) point of view: the space used by a module that detects the first three minimums will use approximately twice the CLBs that are necessary for the first two minimums detector. That is why, we can say that the SA's size grows linearly, but if more solutions are desired (not only the optimal one, but also the first two, the first three etc. best solutions), the space occupied inside the Virtex device will grow as follows:

– as for the SA – the space grows linearly with the size of the S string

– as for the Minimum detector – the space grows exponentially with the number of the first k optimal solutions (if desired).

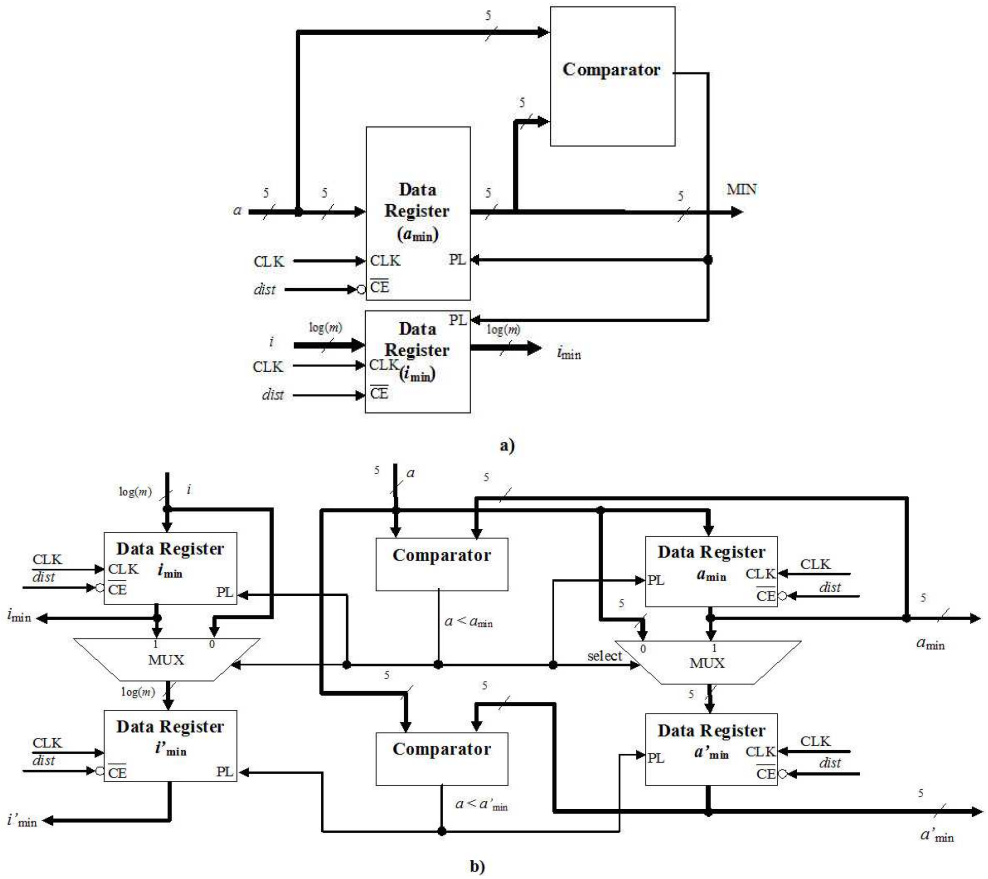


Fig. 10 a) The structure of the minimum (a_{min}, i_{min}) detector module, when $dist = 0$; b) The structure of the first two minimums (a_{min}, i_{min}), (a'_{min},i'_{min}) detector module, when $dist = 0$.

3.4. Performance Considerations

If the software algorithm's complexity is $O(m{\cdot}n)$, with the above presented improvements, the hardware implementation yields a running time of $O(m+n)$for feasible number of processing elements: even if two processing phases are necessary, both of them require $m + 2{\cdot}n$ machine cycles. Since $n \ll m$ the running time is actually $O(m)$. Here we must add one clock cycle for Phase 2 and the supplementary n cycles that are necessary for shifting the solution out of the systolic array.

As for the space occupied in the FPGA chip, the proposed hardware implementation is $O(n)$. This is especially due to the fact that the active logic circuitry is reused in the reverse processing phase (in other words, the same structure is used for computing the GenDiv array and for extracting the solution). This allows the implementation of large SAs in the FPGA chip, thus increasing the system's computing power.

The operating mode can be described as follows:

1. Phase 0 – *Initialization.* The SA is loaded with the S string elements. Duration: n clock cycles.

2. Phase 1 – *Direct processing.* The T string is fed serially in the SA and the GenDiv algorithm is applied. Duration: $m + 2{\cdot}n$ cycles.

3. Phase 2 – *Inversion.* All elements of the S string are interchanged, from the head to the tail. Simultaneously, the FIFO's Address Register is loaded with i_{min}, which is a preparation for reading the T string elements in reverse order starting from the Optimal Solution Point towards the beginning. Duration: 1 clock cycle.

4. Phase 3 – *Reverse processing.* The T string elements are fed into the systolic array in reverse order, starting from the Optimal Solution Point towards the beginning and the GenDiv algorithm is applied. Duration: $m + 2{\cdot}n$ cycles (worst case).

5. Phase 4 – *Solution extraction.* The results are shifted out of the SA. Duration: n clock cycles.

4. Experimental results

The design was simulated and physical tests were performed on a Xilinx Virtex2 PRO©30 FG676 -7 device, on a Digilent© board. This device has a total capacity of 13 696 slices. The maximal operating frequency, as reported by Xilinx synthesis tools, was around 300 MHz. Different sizes of the SA were implemented, showing that relatively large size SAs can be implemented.

A simple calculus shows that, for an $m \times n$ array of 1 000 000 $\times$ 472 elements, the total time is (472 + 1 000 000 + 2 $\times$ 472 + 1 + 1 000 000 + 2 $\times$ 472 + 472) $\times$ 3.355 ns = 6 719 504.72 ns $\approx$ 6.71 ms, which is at least three orders of magnitude better than the software implementation for two strings S and T of the same size.

The table clearly shows that the running times for the software implementation grow considering n but it shows a very small variation in the case of the hardware implementation. The hardware space required grows with n, but for the values needed in the current research of the domain the hardware available on current devices is more then enough.

For algorithms in this class, the performance is usually estimated in Giga Cell Updates Per Second (GCUPS). In our case, since in each clock cycle a new $(a_{i,j}, c_{i,j})$ pair is obtained, the speed is directly dependent on the operating frequency and the number of cells in the systolic array. The maximal performance is obtained for the maximal size of the systolic array (472 PEs): 236 GCUPS.

Of course, as mentioned before, for an implementation in a greater FPGA device or in several FPGA devices, the number of PEs in the SA will be greater and the performance will be better.

Table 2. Performance of the GenDiv algorithm (synthesis results concerning the working frequency and the occupied space) in the amino acid search case (20-characters alphabet) for $m = 1\ 000\ 000$

Systolic array size (the n parameter)	**Number of slices (percentage of the total device capacity)**	**Maximal frequency (MHz)**	**Default period analysis for Clock (ns)**	**Total run time (ms) in FPGA**	**Total run time (ms) in PC**
GenDiv	29 (0.2 %)	306	3.268	6.536	-
GenDiv_4x	123 (0.9 %)	299	3.345	6.690	2204
GenDiv_16x	471 (3.4 %)	301	3.322	6.644	8391
GenDiv_64x	1863 (13.6 %)	299	3.345	6.691	14524
GenDiv_128x	3719 (27.1 %)	299	3.345	6.692	21562
GenDiv_256x	7431(54.25 %)	303	3.300	6.605	28312
GenDiv_472x	13695(99.9 %)	298	3.355	6.719	43374

5. Conclusions

This paper introduces Gendiv, a new algorithm for DNA string detection. It is mainly targeted for the detection of intron and exon strings in DNA chains, but its applicability is not limited to Genetics. The GenDiv algorithm is based on the dynamic programming method, but its adaptation for a hardware implementation was a particular challenge which is described in detail.

The usage of FPGAs gives a great portability and flexibility to this hardware solution (no device-specific components are instantiated) and the whole architecture is described in parameterizable VHDL code only.

Experimental results show a significant improvement, of over three orders of magnitude, of the hardware on the software implementation.

With very small modifications, the hardware implementation can yield the second optimal solution, the third optimal solution etc. To get these results, the MIN module must be designed to yield not only the minimal value of the a parameter, but also the second order minimum (a'), the third order minimum (a'') etc.

By using the same method (reverse processing), but starting from i'_{min}, i''_{min} etc., the result can be the second order solution, third order solution, etc.

This algorithm can be applied for pattern searches in Genetic databases with significantly increased speed over the software implementation, thus contributing to

solving one of the most stringent problems of nowadays biomedical engineering.

This hardware algorithm gives many possibilities for executing it on multi-core microprocessors, highly parallel SoCs or DSPs and parallel GPUs. Future work will consist on testing the algorithm on different systems like a Cell processor, a highly performing GPU and on an Itanium 2 processor.

References

[1] BOECKMAN B. et al., *The Swiss-prot protein knowledgebase and its supplement TrEMBL in 2003*, Nucleic Acids Research, **31**, no. 1, 2003, pp. 365–370.

[2] BENSON D. A., KARSCH-MIZRACHI I., LIPMAN D. J., OSTELL J., WHEELER D. L., *Genbank: update*, Nucleic Acids Research, Database issue **32**, 2004, D23-D26.

[3] STOESSER G. et al., *The EMBL nucleotide sequence database*, Nucleic Acids Research, **30**, 2002, pp. 21–26.

[4] ZHANG M.Q., *Computational prediction of eukaryotic protein-coding genes*, Nature Rev. Genet., **3**, 2002, pp. 698–709.

[5] GOKHALE M. B., GRAHAM P. S., *Reconfigurable computing*, Springer, 2005, p. 160.

[6] NEEDLEMAN S. B., WUNSCH C. D., *A general method applicable to the search for similarities in the amino acid sequences of two proteins*, Journal of Molecular Biology, **48**, 1970, pp. 443–453.

[7] SMITH T. F., WATERMAN M. S., *Identification of common molecular subsequences*, Journal of Molecular Biology, **147**, 1981, pp. 195–197.

[8] PEARSON W. R., LIPMAN D. J., *Improved tools for biological sequence comparison*, Proc. Natl. Acad. Sci., **85**, 1988, pp. 3244–3248.

[9] ALTSCHUL S. F., GISH W., MILLER W., MYERS W. E., LIPMAN D. J., *Basic local alignment search tool*, Journal of Molecular Biology, **215**, 1990, no. 3, pp. 403–410.

[10] HÖHL M.,. KURTZ S, OHLEBUSCH E., *Efficient multiple genome alignment*, Bioinformatics, 2002, **18**:312S–320S.

[11] KADERALI L., SCHLIEP A., *Selecting signature oligonucleotides to identify organisms using DNA arrays*, Bioinformatics, 2002, **18**:1340–1349.

[12] KURTZ S., SCHLEIERMACHER C., *REPuter: fast computation of maximal repeats in complete genomes*, Bioinformatics, 1999, **15**:426–427.

[13] TSAFRIR D., ETSION Y., FEITELSON D. G., *Modeling user runtime estimates*, in 11^{th} *Job Scheduling Strategies for Parallel Processing (JSSPP)*, pp. 1–35, Springer-Verlag, Jun 2005. LNCS 3834.