Hardware Acceleration of Approximate Palindromes Searching

Tomáš Martínek
Faculty of Information Technology
Brno University of Technology
Božetěchova 2, Brno, 612 66, Czech Republic
Email: martinto@fit.vutbr.cz

Matej Lexa
Faculty of Informatics
Masaryk University
Botanická 68a, Brno, 602 00, Czech Republic
Email: lexa@fi.muni.cz

Abstract

Understanding the structure and function of DNA sequences represents an important area of research in modern biology. Unfortunately, analysis of such data is often complicated by the presence of mutations introduced by evolutionary processes. They increase the time-complexity of algorithms for sequence analysis by introducing an element of uncertainty, complicating their practical usage. One class of such algorithms has been designed to search for palindromes with possible errors—approximate palindromes. The best state-of-the-art methods implemented in software show time-complexity between linear and quadratic, depending on required input parameters. This paper investigates the possibilities for hardware acceleration of approximate palindrome searching and describes a parametrized architecture suitable for chips with FPGA technology. A prototype of the proposed architecture was implemented in VHDL language and synthesized for Virtex technology. Application on test sequences shows that the circuit is able to speed up palindrome searching by up to 8000× in comparison with the best-known software method relying on suffix arrays.

1. Introduction

A palindrome is a sequence of symbols ordered in such a way, that the order is identical when read forward and backward (e.g. $abba$, $ababa$). Generally, palindromes can be written in the form $p = w.w^R$ or $p = w.c.w^R$, where w is a string, w^R is its reversed version and c is a central unpaired symbol. Depending on the presence of the symbol c, the palindrome is called *even* or *odd*. An occurrence of palindrome p in string S is called *maximal*, if its extension by one character to the left or right does not form a longer palindrome. Many algorithms for the detection of all maximal palindromes inside an arbitrary long string exist. Methods based on suffix arrays and the longest common extension [5] belong to the most significant works in this area with their ability to find all maximal palindromes in linear time $O(n)$.

In molecular biology, analysis of DNA molecules is one of the key subjects of research because it helps to understand functioning of living organisms at the molecular level. DNA is often represented as a long sequence of four letters A, C, G and T (corresponding to basic chemical units - nucleotides). Since the nucleotides are evenly spaced and able to combine into complementary DNA strands, palindromes in the DNA sequence can create structures differing from the well-known general helical structure of non-palindromic sequences. It is believed, that structures formed by palindromic subsequences play a role in regulation of gene activity or other processes in cells. For example, hairpin and triplex palindrome-based structures are known to be present in close vicinity of genes (e.g. in promoters, introns and 3'-untranslated regions) contributing to their normal functioning, or to diseases, such as cancer. A DNA sequence can also be copied into a chemically related RNA molecule, which also adopts interesting structures when containing palindromes. In short, the knowledge of the exact positions of palindromes in DNA is an important bit of information for molecular biologists trying to understand how entire genomes are organized and what the functions of its individual components are.

The process of searching for palindromes in biological data is often complicated by the presence of mutations introduced by evolutionary processes. These mutations occur in sequences as character insertions, deletions or substitutions. The algorithms, which search for palindromes in DNA sequences have to tolerate these changes, in order to find not only exact palindromes but also *approximate palindromes*. Unfortunately, the time-complexity of such algorithms is higher, which complicates their practical usage.

While software tools capable of identifying DNA palindromes exist [6], they differ in their ability to deal with large datasets. Newest programs based on suffix arrays, such as Reputer, try to reduce that problem, but are still slow for

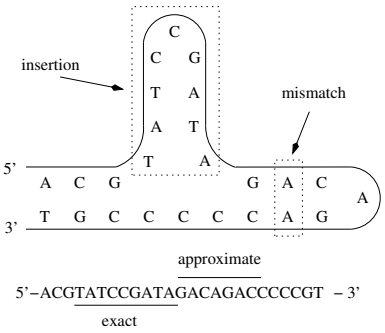

Figure 1. Schematic representation of a possible DNA structure with palindromes present in the nucleotide sequence.

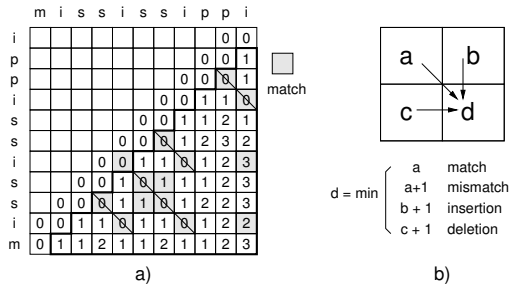

Figure 2. Dynamic programming algorithm demonstrated on the string $mississippi$: a) the DP matrix with calculated score values b) the scheme of DP rule calculating new score d based on neighbouring values a, b and c.

interactive large-scale searches. The size of genomic data, the progress in sequencing technology and the difficulty in storing sets of palindromes interesting to biologists [8] point towards the need for interactive searching of genomic data. One of the ways to obtain interactivity is to implement the searching procedures in hardware.

The objective of this work is to study the best algorithms for approximate palindrome searching and investigate possibilities for their hardware acceleration. This paper is organized as follows: Section 2 defines approximate palindromes and describes state-of-the-art software methods for approximate palindrome searching. Related work in this area is summarized in section 3. Section 4 contains detailed description of hardware architecture for acceleration of approximate palindrome searching. Evaluation of proposed architecture and its comparison with software implementation is given in section 5. Conclusions are summarized in section 6.

2. Approximate Palindromes

An approximate palindrome is formally defined by Porto and Barbosa [9]. In comparison to exact palindrome, errors in the form of character insertion, deletion and substitution are allowed. Generally, these errors represent edit operations and their number k is the number of necessary changes for conversion of an approximate palindrome into an exact palindrome. Unlike exact palindromes, the size of an even approximate palindrome does not have to be even, nor does the size of an odd approximate palindrome have to be odd. Similarly, the definition of palindrome maximality differs from exact palindromes. As described in [9], an approximate palindrome is maximal if no other approximate palindrome for the same center c and number of edit operations k exists, while having strictly greater size or the same size but strictly fewer errors. This definition clearly does not guarantee the uniqueness of a palindrome maximality.

The definition of exact or approximate palindrome has to be slightly modified to be useful in molecular biology. When a palindromic DNA sequence is read from the other end it should not give the same sequence, but rather a sequence which is complementary to the original. As a rule, in molecular biology only $C-G$ and $A-T$ pairs are complementary (we call any such pair a match and a sequence of complementary nucleotides written in opposite direction is called reversed). A pair made by non-complementary nucleotides is called a mismatch. For example, $ACGT$ is a palindrome, because when read backwards ($TGCA$) and reversed according to the complementarity rule, we obtain the original sequence ($reverse(TGCA) = ACGT$). An example of a palindrome is given in Fig. 1.

Algorithms for palindrome searching have been studied intensively in the past. One of the first algorithms for finding all palindromes [7] use dynamic programming (DP) techniques to calculate a two dimensional matrix of all possible palindrome alignments. Unfortunately, the algorithm time complexity converges to $O(n^2)$. The another (more practical) approach does not calculate the whole DP matrix, but only searches for palindromes with at most k errors. The best algorithms of this kind are based on suffix trees or suffix arrays [3, 1], which allow them to reduce time complexity to $O(kn)$. In the following subsections these important approaches are described in more detail.

2.1. Palindrome detection by a dynamic programming algorithm

A convenient way to search for approximate palindromes is by the way of a dynamic programming algorithm. Original algorithm can be traced back to [7]. A DP matrix is constructed so that one side represents the original sequence, while the other contains the same sequence reversed (according to the nucleotide-pairing rules for DNA sequences).

With such setup, the main antidiagonal of the DP matrix represents all the n possible starting positions for odd palindromes. The neighboring antidiagonal contains the other $n-1$ possible starting sites of all the even palindromes that can exist in the sequence. Consequently, diagonals starting at any of these positions represent potential palindromes. If we fill the cells representing the starting positions with zeros, we can start filling the DP matrix along the diagonals. The numbers entered will represent the number of errors found so far in the evaluated palindromes. At each position $[i, j]$ of the DP matrix, we compare the symbols at positions i and j in the original and reversed sequences. If they are the same, no penalty is introduced. If the symbols differ, the number of errors identified so far in the particular palindrome score is incremented by one.

The necessity for a dynamic programming algorithm comes from the possibility to insert gaps into the palindromes, where symbols in some positions have no symbols to pair up with in the palindrome. In terms of the described algorithms, this means moving from one diagonal to a neighboring one when calculating the number of errors. At any position, three possibilities are evaluated:

1. Extending the existing palindrome along the diagonal - *match* or *mismatch*,
2. Inserting a gap at position *i* of the original sequence - *insertion*,
3. Inserting a gap at position *j* of the reversed sequence - *deletion*.

The solution that leads to the lowest number of errors is kept, the score is recorded in the DP matrix, while the other possibilities are discarded.

2.2. Palindrome detection using suffix arrays to obtain longest common prefix

When sequences contain long exact palindromes, the above algorithm suffers from the need to evaluate every cell of the DP matrix, while the overall score remains unchanged throughout the palindrome. Moreover, when there is a palindrome on diagonal d, no palindromes are usually present on diagonals $d+1$ and $d-1$ (except special cases). In the dynamic programming algorithm these neighboring diagonals adopt the better score from diagonal d with the addition of the appropriate penalty for character insertion or deletion. If we had a method to calculate the length of the next perfectly matching stretch of characters, we could simply jump to the next error-containing position.

It turns out, that this is a problem of finding the longest common prefix (LCP) at arbitrary positions in the two strings (original and reversed). Fortunately, there is a linear-time solution using suffix trees or suffix arrays for this problem [3]. The overall algorithm of searching for all approximate palindromes with k errors operates in the following steps:

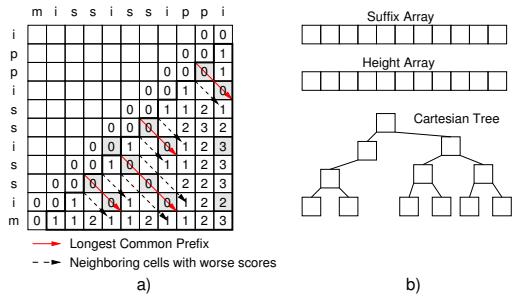

Figure 3. Suffix arrays algorithm demonstrated on the string $mississippi$: a) DP matrix calculation using the LCP, b) the scheme of suffix array, height array and cartesian-based tree

1. Create a suffix array of a string $S.S^R$, where S is the original string in which the palindromes are to be searched for. This suffix array (SA) is sorted alphabetically by prefixes.
2. A *rank* array is created to link original positions of the sorted suffixes with their positions in the sorted SA.
3. Create the corresponding LCP array identifying the longest common prefix of neighboring items in the sorted SA. The content of this array is also called *height*.
4. Then, LCP for any given pair of substrings can be calculated as the minimum over the *height* values corresponding to positions of compared substrings in sorted SA. A linear time solution for this problem exists as well. One of the solutions is to use a *cartesian-tree-based* algorithm [4] to issue a range minimum query on the heights.
5. Starting from each possible palindrome position (on the main and neighboring antidiagonal) perform: (a) Find the longest common prefix of substrings at offsets i and j and for the appropriate diagonal d store the reached position. This step jumps over the DP cells matching the input strings and preserving the same score (without errors); (b) The new reached position on diagonal d is calculated as the maximum of the reached positions at diagonals $d-1$, d and $d+1$ incremented by one This step simulates single error in DP matrix.
6. Repeat step 4 k-times starting from the last reached position.

The first three steps of the algorithm represent an initialization phase which can be performed in linear time. Then the main algorithm loop is composed of an LCP lookups and a single DP-like operation, both steps applied

to all matrix diagonals. As the LCP of substrings at arbitrary positions can be identified in constant time using *cartesian-based-tree*, the overall algorithm time complexity is $O(k.n)$. As depicted in Fig. 3, the algorithm effectively jumps over the matches along the diagonals and also updates the reached positions of neighboring diagonals.

2.3. Alternative definition of palindrome quality

The methods for detection of palindromes that have at most k errors are well-balanced, resulting in time complexity $O(kn)$. However, palindromes in real data may have variable number of errors and variable length as well. We tend to tolerate more errors in longer palindromes and less errors in shorter ones. Therefore, it is more natural to define an acceptable average frequency of errors, rather than a fixed value k. This frequency can be expressed as $e = l/k$, where l is the length of the palindrome in question. The matter becomes even more complex if we consider the possibility of different scoring schemes. In this case, e may be replaced by e_S (a measure of the average score S per unit palindrome length) yielding the equation $e_S = l/S$. However, this is beyond the scope of the presented paper. We modified the suffix array-based method accordingly, so that k is not fixed, but rather changes as calculation progresses towards longer and longer palindromes to satisfy a fixed value of e. Whenever the average frequency decreases below the calculated threshold, we store the number of the diagonal to which the palindrome belongs and the palindrome is *exported* (its position is recorded). In the worst case, the palindrome never drops below e over its entire length and covers the whole string of length n. In this case the method has to go through $k = n/e$ cycles of diagonal extensions. The time complexity in such case will be in $O(n^2/e)$. Our analysis of real-world data files of practical interest shows that these cases are quite rare. Most palindromes are much shorter (see detailed discussion in Section 5).

3. Related Works

Palindrome search acceleration in hardware has been studied by Conti et al. [2]. Their architecture is made of an array of processing elements connected into a loop The input string progresses through the array from left to right, changes direction at the end of the array and continues in the opposite direction. The processing elements contain only comparators which signal palindrome-forming matches on individual positions. This architecture is able to detect approximate palindromes of all relevant lengths at every step. Only mismatches can be evaluated. Time-complexity of this approach is $O(n)$ as compared to $O(kn)$ for the best algorithms implemented in software. Using a longer array of processing elements leads to detection of longer palindromes without changes in time-complexity.

The authors also address the problem of detecting palindromes containing insertions and deletions. To do that they rely on a triplet of the above mentioned arrays. The first array operates as described for detecting palindromes with mismatches. The second array compares one symbol ahead and the third array compares one symbol behind the normal position. Analyzing the results from these three arrays the authors can detect possible insertions. Unfortunately, generalization of this approach to k-errors leads to high number of comparator arrays and an unnecessarily high use of resources on the chip.

The objective of this work is to design a more scalable hardware architecture that will be able to utilize the parallelism inherent to palindrome searching. Computation array of more processing elements should not only be capable of analyzing longer palindromes, but also accelerate the computation as such. Similarly, amount of accepted insertions and deletions should not cause a huge increase of consumed resources.

4. Hardware Architecture

We investigated the possibilities of hardware acceleration of both basic approaches for palindrome searching described in subsections 2.2 and 2.1. Our observations are as follows:

1. The method based on suffix arrays is very effective and calculates only the necessary number of DP matrix items. Unfortunately, its hardware realization is complicated by several factors: i) entire suffix array (or cartesian-based-tree) would apparently have to reside inside the chip and thus consume significant amount of resources, and ii) during the computation the suffix array would have to be concurrently accessed from a number of chip locations corresponding to the level of parallelization. This would result in a complex interconnection system.

2. On the other side, the method based solely on the dynamic programing (DP) technique offers easier parallelization and possible hardware acceleration. The remaining problem is the algorithm time-complexity, that requires up to $n^2/2$ steps for DP matrix computation. However, if we consider properties of real data, then the probability of palindrome occurrence decreases exponentially with palindrome length. Based on this fact, only a limited number of steps (the first k antidiagonals) is sufficient for finding almost all palindromes.

In our first approach, we design a hardware circuit that implements the basic DP technique, but calculating only a

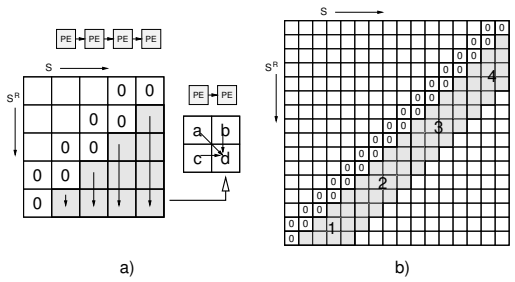

Figure 4. (a) Allocation of PEs for DP matrix computation (b) Computation of the first k-antidiagonals split into stripes

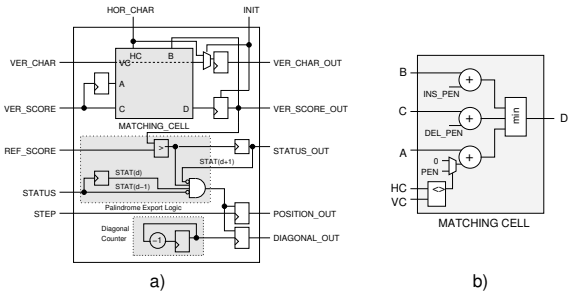

Figure 5. (a) Processing Element Architecture (b) Matching Cell Architecture

limited number of cells (the first k antidiagonals). The proposed architecture is similar to the one used in acceleration of approximate string matching (ASM) [10]. Nevertheless, there are some important differences between the two architectures. While the whole area of the DP matrix is computed in ASM, only the lower triangular part is significant in palindrome searching (see Fig. 4a).

Similarly to ASM, the architecture of the circuit is based on a systolic array of processing elements, where each element calculates a single column of the DP matrix. Note that the first ASM element begins computation in the upper left corner of the matrix and the next elements start their computation consequently in an ordered fashion because of data dependencies. On the other hand, palindrome detection begins on the central antidiagonal and all elements can proceed on the diagonals in parallel. Data dependency does not require the elements to wait for each other.

If the number of processing elements is lower than the length of the input string, computation is divided into bands, so that results generated by the last element are temporarily stored in a buffer. Upon transfer of computation to the next band, the first processing element accesses the data in the buffer (see Fig. 4b).

4.1. Processing Element

Detailed architecture of the processing element is shown in Fig. 5. The main part is the *Matching cell*, which carries out the basic DP rule (see Fig. 5b). The unit compares two characters in the relevant column and row of the matrix and calculates the score for a match/mismatch. At the same time it calculates values for an insertion/deletion by adding penalization constants to the values on the neighboring diagonals (inputs B, C). As resulting score (output D) determines the minimum of these three values.

While the horizontal character is the same for the whole column, the vertical character moves from element to element from right to left in a pipeline fashion. Initialization of computation is somewhat special. Please, note that the rows are reversed columns and therefore the vertical character can be taken from the horizontal character at the previous pipeline stage. Distribution of the vertical character is therefore quite simple and requires only little amount of resources. Similarly, the last score of each diagonal is passed to the neighboring element via an auxiliary register.

4.2. Palindrome Detection

Compared to an ordinary ASM element, in palindrome detection, we have to wait for the moment when the palindrome cannot be extended any further without loss of quality, as measured by the e ratio (see description in Section 2.3). A comparator is used for this, which compares the resulting ratio with a pre-defined threshold and stores the logical value of such comparison in a *status* register. If the score is higher than the threshold, the number of the diagonal and the achieved position are stored and the resulting palindrome is *exported*.

Usually, when there is a palindrome on diagonal d, no palindromes are present on diagonals $d+1$ and $d-1$ (except a few special cases). In the DP algorithm these diagonals adopt the score from diagonal d with the addition of the appropriate penalty for character insertion or deletion. Because of this, palindromes are also exported from these neighboring diagonals. This happens just before the d-diagonal palindrome gets exported. To prevent this behavior, additional logic circuits are included in the design to export a palindrome only if the neighboring diagonals have also exceed the threshold for palindrome exporting.

4.3. Systolic Array

The overall architecture of the systolic array is shown in Fig. 6. The input string or its fragment is stored in the *String Memory* block. All the characters have to be prepared before the first computation cycle. This is reflected in the appropriate output memory width. The main part of the

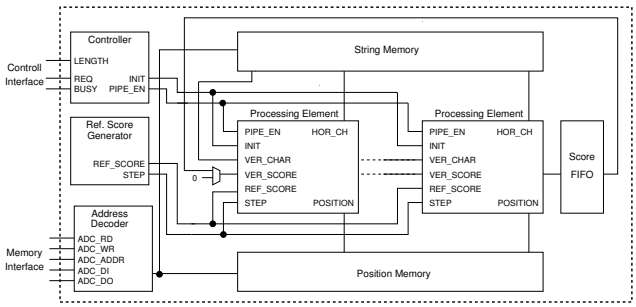

Figure 6. Systolic Array Architecture

architecture is an array of processing elements connected so as to pass the vertical bits, score values and diagonal numbers to their neighbors.

In each step, the actual threshold score and position values are incremented for all elements by the *Reference Score Generator* block. Positions and diagonal numbers of the exported palindromes are stored in the output *Position Memory*. When a long string is analyzed, computation is split into bands and the output score of the last element is stored into an auxiliary FIFO memory. Subsequently, these values are used by the first element when analyzing the next band of the DP matrix. The overall activity of the array is controlled by a *Controller* unit.

Using a simple modification of the architecture, it is possible to achieve *incremental computation*. If the accelerated software operating at higher level decides that the originally given number of k steps is not sufficient for finding all needed palindromes, it can direct the hardware to calculate the next k steps from the last reached antidiagonal. At hardware level, it is only necessary to store the intermediate scores from the last two antidiagonals and take these values for the initialization of the next run (instead of zero values at the central antidiagonals). Using this approach, there is no limit on the maximal length of sequences or palindromes, apart from linearly growing computation time.

The proposed architecture represents a general template of a circuit for palindrome searching. When applying this template to the target application, it is necessary to specify several parameters such as character data width. While two bits are sufficient for expression of DNA characters, general ASCII text requires 7 bits per character. Further, it is possible to configure penalization constants for character ins/del/mismatch or even to change the function for character comparison. While a general text palindrome is composed of pairs of identical characters, for DNA sequences the characters have to be complementary (A-T, C-G).

The configurability of the chips with FPGA technology allows us to specify all these parameters before the synthesis process and thus create a circuit optimized for target application. Created circuit can achieve higher working frequency, consume less resources and contain more PEs.

5. Results

When analyzing real-world sequences, such as DNA, we can see that with increasing palindrome length the number of palindromes exponentially decreases almost to zero. This is very similar to random sequences, where this decrease corresponds to the probability of complementary character pair occurrence. The histogram in Fig. 7 compares the number of palindromes for random and DNA sequences. The depicted values were calculated as an average on ten samples of random and DNA sequences, each with 10k characters in length. Additionally, the histogram was measured for different parameter e, which impacts the quality of exported palindromes (as described in Section 2.3). The graphs in Fig. 7 show, the DNA sequences have approximately identical characteristics as the random sequences, when the four matches per one mismatch are allowed. However, with increasing e, the differences between random and DNA sequences become more significant and confirms the surmise, that the DNA have a reason for creating the palindrome structures and does not generated them in random only.

The main goal of this chapter is to compare the performance of the software-implemented method for palindrome detection with its hardware counterpart proposed in this paper. The comparison of method has to be established on the common criterion. As the histogram in Fig. 7 indicates, it seems reasonable to search palindromes only to the limited length. We will therefore concentrate on hardware and software approaches that can detect all palindromes of prescribed quality up to length l.

As software implementation, we selected suffix array-based method described in [3], which seems to be the most time-efficient method for palindrome detection to date. The method utilizes the open source libraries for LCP and RMQ computation with slight modification in order to accomplish the common criterion. The detection of all palindromes of prescribed length and quality is ensured only if the algorithm exceeds the boundary (corresponding to the specified length) at all diagonals. We implemented this algorithm in C language and tested it on a computer system with a Xeon 3GHz processor and 4GB of RAM. In repeated tests on a sequence of 10k characters we determined the performance of the processor to be $p_1 = 0.03B$ (billion) steps per second, where one step is the calculation of LCP (using an access to the suffix array) plus application of one DP matrix rule.

The hardware architecture proposed in Section 4 was implemented in VHDL and synthesized into FPGA chip xc2vp100 with a Xilinx Virtex II Pro technology, speed grade 7. The basic characteristics of the circuit after carrying out Place and Route are shown in the following table 1. Single chip can harbor a systolic array of up to 1015 processing elements working at 241 MHz. Multiplication of the frequency and the number of elements indicates that the

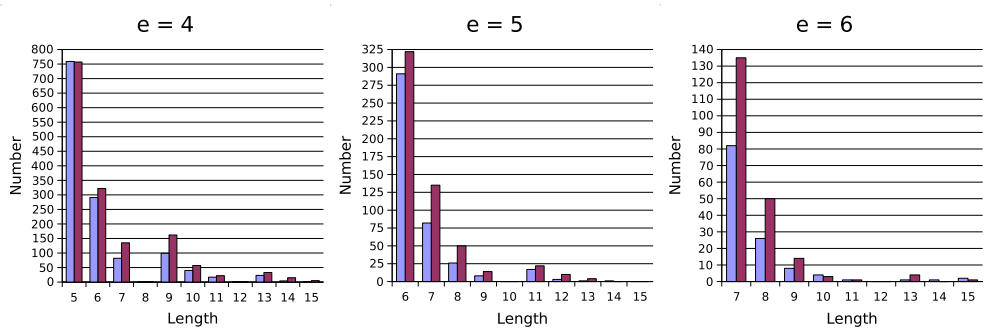

Figure 7. Histogram of exported palindromes for random (the first column) and DNA (the second column) sequences. The histogram is repeated for different values of parameter e

performance of the hardware is up to $p_2 = 244B$ (billions) steps per second.

Table 1. Comparison of Hardware and Software

	HW	SW
PE Resources FDD/LUTs	31/54	-
Number of PEs	1015	1
Working frequency	241 MHz	3 GHz
Performance	244 BSps	0.03 BSps

Because it is not possible to directly compare hardware and software performance, let us consider the number of steps each method has to go through. To find all palindromes up to length l, the hardware architecture needs $(n/n_{PE}).2l$ steps, where n is the length of the input sequence, n_{PE} is the number of processing elements and the calculation is done in (n/n_{PE}) bands.

On the other hand, it is not possible to determine the exact number of steps for the software method without considering the characteristics of the analyzed sequence. In general, the time-complexity of the method is $2kn$, where k is the number of allowed errors. For example, if the input sequence is a^n (palindromes at every position), the software method finds them in the first batch of $2n$ steps while $k = 1$ (the best case). On the other hand, if there is no palindrome in the sequence, the software method has to go through $k = l$ batches of computation, making all the $2kn$ steps necessary (this is equal to hardware). Depending on the input sequence, k will varying the range $1..l$.

Let us now derive an expression for the speed-up of hardware-implemented palindrome detection as compared to software implementation. The computation time in software resp. hardware is:

$$T_{SW} = \frac{2kn}{p_1}, \qquad T_{HW} = \frac{\frac{n}{n_{PE}}.2l}{p_2}. \tag{1}$$

Speed-up α of hardware against software is:

$$\alpha = \frac{T_{HW}}{T_{SW}} = \frac{k}{l}.\frac{p_2.n_{PE}}{p_1} \tag{2}$$

Please, note that the expression $\frac{p_2.n_{PE}}{p_1}$ is constant. After substituting appropriate values we arrive at: $\alpha = 8255.\frac{k}{l}$. It is clear, that the obtained speed-up will depend on the ratio of steps necessary to cover length l and length itself. We determined the value of k for different values of l experimentally using the following DNA sequences of length 5*10000 nucleotides:

1. human DNA sequence from promoter regions (the first 25 promoters from chromosome 1 in the hg18 release of UCSC Human Genome)
2. Randomly generated sequence of symbols A, C, G, T (each character with equal probability),
3. String which (under DNA complementarity rules) has no palindromes: c^n.
4. String with palindromes at each positions: $(cg)^n$.

The obtained relationships of k and l are shown in Fig. 8. As expected, $k = l$ for a sequence with no palindromes. Also, $k = 1$ when there is a large palindrome in the sequence. For random and DNA sequences, the k lies somewhere between 1 and l. Interestingly, DNA sequences require slightly more computation steps then random sequences. This means, that the random sequences contain better conditions for longest common prefix searching and application of DP rule. On the other hand, the histogram shows, that the DNA sequences contain more palindromes than the random sequences. This is caused by the fact, that DNA sequences creates the palindromes only in restricted areas while other parts remain palindrome free.

The graph shows that the relationship of k and l is almost linear for the first three sequences. Consequently, the k/l ratio allows us to express acceleration obtained in hardware as compared to software using equation (2). The ta-

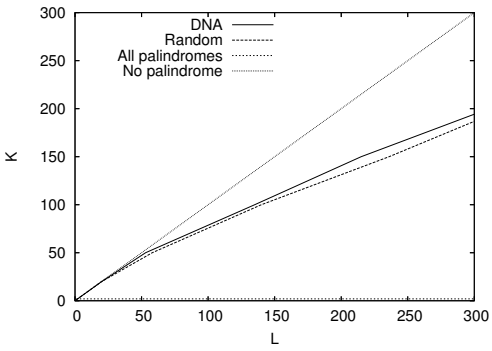

Figure 8. Number of software algorithm steps k for searching all palindromes of given length

ble 2 summarizes the values of the k/l ratio and the resulting speed-up.

This table shows that acceleration is maximal for sequences with no palindromes. The corresponding speed-up is 8255. DNA sequences and random sequences provide for intermediate acceleration of about 4400 compared to software. A sequence with palindrome at every position is a special case. While software needs only one set of calculation steps, hardware needs l steps. As a result, the software implementation will become faster above a certain value of l. This threshold length can be determined from equation (2), if we substitute $k = 1$ and $\alpha = 1$. This results in threshold length of: $l = \frac{p_2.n_{PE}}{p_2} = 8255$.

6. Conclusions

This paper describes a novel hardware architecture for approximate palindrome searching. The proposed circuit is based on dynamic programming and allows concurrent computation of a large number of antidiagonal DP matrix cells. The presented architecture ensures very good scalability. Increasing number of processing elements allows processing proportionally longer sequences. Very long sequences can be processed in bands.

The proposed hardware architecture utilizes the advantages of FPGA technology. Configurability of the chips allows us to specify parameters such as character width, penalization scores for character ins/del/sub and comparison function before the synthesis process and thus create circuit optimized for target application. The created circuit can achieve higher working frequency, consume less resources and contain more processing elements.

Comparing our architecture to an equivalent software implementation of a well-known algorithm using suffix arrays, the circuit shows $8255\times$ acceleration. In such comparisons, one has to take into account the characteristics of the sequences to be analyzed, especially their ability to take advantage of suffix arrays. Experimental treatment of this problem showed that we should expect acceleration $4400\times$ on DNA sequences.

Table 2. Speed-up of HW for different sequences

	Tangent	Speed Up
Sequence without palindromes	1	8.255
Random sequences	0,536	4.410
DNA sequences	0,535	4.404

Acknowledgment

This research has been partially supported by the Research Plan No. GACR, 204081560 – In vitro and in silico identification of non-canonical DNA structures in genomic sequences and Research Plan No. MSM, 0021630528 – Security-Oriented Research in Information Technology.

References

[1] L. Allison. Finding approximate palindromes in strings quickly and simply. *CoRR*, abs/cs/0412004, 2004. informal publication.

[2] A. A. Conti, T. V. Court, and M. C. Herbordt. Processing repetitive sequence structures with mismatches at streaming rate. In *Field Programmable Logic and Application (FPL 2004)*, Lecture Notes in Computer Science, pages 1080–1083. Springer, 2004.

[3] R. de Castro Miranda and M. Ayala-Rincón. A modification of the landau-vishkin algorithm computing longest common extensions via suffix arrays. In *BSB*, pages 210–213, 2005.

[4] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 135–143, New York, NY, USA, 1984. ACM.

[5] D. Gusfield. Algorithms on stings, trees, and sequences: Computer science and computational biology. *SIGACT News*, 28(4):197–198, 1997.

[6] S. Kurtz, J. V. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich. Reputer: the manifold applications of repeat analysis on genomic scale. *Nucleic Acids Research*, 29(22):4633–4642, 2007.

[7] G. M. Landau and U. Vishkin. Efficient parallel and serial approximate string matching. Technical Report Computer Science Department Technical Report #221, February 1986.

[8] L. Lu, H. Jia, P. Dr oge, and J. Li. The human genome-wide distribution of dna palindromes. *Functional and Integrative Genomics*, 7(3):221–227, 2007.

[9] A. H. L. Porto and V. C. Barbosa. Finding approximate palindromes in strings. *Pattern Recognition*, 35:2581, 2002.

[10] C. W. Yu, K. H. Kwong, K.-H. Lee, and P. H. W. Leong. A smith-waterman systolic cell. In *Field Programmable Logic and Application (FPL 2003)*, pages 375–384, Lisbon, Portugal, September 2003.