

CAAD Lab Technical Report 2004-01

Processing Repetitive Sequence Structures at Streaming Rate^{*}

Albert A. Conti¹, Tom Van Court¹, and Martin C. Herbordt¹

Department of Electrical and Computer Engineering
Boston University, Boston, MA 02215
herbordt|alconti|tvancour@bu.edu

Abstract. With the accelerating growth of biological databases and the beginning of genome-scale processing, cost-effective high-performance sequence analysis remains an essential problem in bioinformatics. We examine the use of FPGAs for finding repetitive structures such as tandem repeats and palindromes under various mismatch models. For all problems addressed here, we process strings in “streaming mode” and obtain processing times of 5ns per character for arbitrary length strings. Using a Xilinx XC2VP100, we can find: (i) all repeats up to size 1024, each with any number of mismatches; (ii) all precise tandem arrays with repeats up to size 1024; (iii) all palindromes up to size 256, each with any number of mismatches, or (iv) a somewhat smaller size of (i) and (iii) with a single insertion or deletion. We sketch methods for filtering the output also in streaming mode. The speed-up factors range from 250 to 6000 over an efficient serial implementation which is itself many times faster than a direct implementation of a theoretically optimal serial algorithm.

1 Introduction

One of the most significant technological achievements of recent times has been the development of fast methods for sequencing genes and proteins. This has enabled the creation of large databases which can be processed by abstracting sequences of nucleic acids (DNA, RNA) and amino acids (proteins) into strings of characters. This type of processing is one of the fundamental bases of modern bioinformatics and has transformed biology from a laboratory science to a computational one as well.

^{*} This work was supported in part by the National Science Foundation through CAREER award #9702483 and the National Institutes of Health through award RR020209-01.

For the last 15 years or so, the dominant production sequence analysis techniques have been based on partial string matching, either with the goal of obtaining a precise solution (e.g. with dynamic programming) or, much more commonly, a fast solution (e.g. with heuristic techniques such as BLAST). However, there are sequences of DNA for which these heuristic techniques do not work very well. These are often repetitive sequences; processing these has been found to be critical as a complement to—or as a preprocessing step for—not only partial string matching but also sequence assembly [1]. Repetitive sequences are also critical in their own right as this small sample shows: (i) they comprise perhaps 10% of the human genome, (ii) several diseases are a result of anomalies in repeat patterns, and (iii) repeats are fundamental to cell processes such as transcription [2, 3]).

Unlike sequence alignment—where there has been some interest in FPGA-based acceleration (e.g. [4–6])—analysis of repetitive structures has as yet received little such attention. Perhaps our primary contribution here is to show that this critical application is also well-suited to FPGA-based acceleration.

In this initial study, we have investigated the capabilities of a contemporary high-end FPGA with respect to a set of basic problems in the analysis of repetitive sequence structures. To constrain the study, we use a single algorithmic model: the data are passed through the FPGA in streaming mode and processed systolically. We have found that simple hardware structures are sufficient to provide the basic functions of finding all repeats and palindromes of arbitrary size with an arbitrary number of mismatch errors constrained only by what can fit on the chip. The simplicity of the structures ensures the fast cycle time: all processing described here can be done at a rate of one character per 5ns.

With the cycle time fixed, the issue becomes how complex a problem can we solve within the given hardware constraint. We can find: (i) all repeats up to size 1024, each with any number of mismatches; (ii) all precise tandem arrays with repeats up to size 1024; (iii) all palindromes up to size 256, each with any number of mismatches, or (iv) a somewhat smaller size of (i) and (iii) with a single insertion or deletion. We sketch methods for filtering the output also in streaming mode. The speed-up factors range from 250 to 6000 over an efficient serial implementation which is itself many times faster than a direct implementation of a theoretically optimal serial algorithm.

The primary significance of this work is that we have found a critical area so far little explored in terms of FPGA acceleration. We have found tremendous potential. In practical terms, processing sequences at a rate of 100s of millions of characters per second means that the few billion characters of the human genome can be processed in seconds. Further, this opens up the promise of large-scale genome processing at the desktop.

In the next section we present some formal definitions and asymptotic complexity of some important tasks. There follows the central part of the paper, a

description of the problems we addressed and the algorithms we used to do so. After that come some results and analysis followed by a brief description of work in progress and some ideas for future work.

2 Definitions and Previous Work

The following definitions are based on Gusfield's excellent introduction to repetitive structures in biological sequences (found in [3]). A *tandem array* A is a periodic string that can be written a^r for some $r \geq 2$. If $r = 2$, then the tandem array may be called a *tandem repeat*. A *palindrome* is a string that reads the same forwards and backwards. A *complemented palindrome* (in the context of a biological sequence) is a DNA or RNA sequence that becomes a palindrome if each character in one half of the string is changed to its complement character (C to G, etc.). We continue to use k , and r to mean number of errors, and number of repeats, respectively.

It is usually the case that the length of the string to be processed is unbounded though no practical algorithm can find (or would want to find) all repeats with size into the billions. It is also usually the case, however, that algorithms that work on strings of size n can be easily extended to work on strings of arbitrary size N as long as they do it n characters at a time. n is also then assumed to limit the size of the repeats $|a|$ that can be found. Sometimes in practical applications, the sizes of repeats of interest are known to be bounded—specifying minimum and maximum repeat sizes can improve program performance. The maximum repeat size of interest is almost always less than n , the size of the substring currently being processed.

Repetitive structures often occur in biological sequences that, although they are inexact, are still of interest. The two most important measures of error in a tandem repeat or a palindrome are the Hamming distance (number of mismatch errors) and the edit distance (number of insertions and deletions as well as mismatches). For the first type of error, we say that a substring is a k mismatch tandem repeat (or palindrome) if it becomes a tandem repeat (or palindrome) when at most k characters are changed. For the second type of error, we say that the edit distance is k if the substring becomes a tandem repeat (or palindrome) after at most k insertions, deletions, or changes.

Some results regarding asymptotic complexity follow. For a string of length n , all exact tandem repeats can be found in $O(n \log n)$ time [7] while all exact palindromes can be found in $O(n)$ time. All tandem repeats with k or fewer mismatches can be found in $O(nk \log(n/k))$ time [8] while all palindromes with k or fewer mismatches can be found in $O(nk)$ time [3]. All tandem repeats with k edit errors can be found in $O(nk \log k \log(n/k))$ time [8]. Other problems that have been studied involve finding tandem arrays with mismatch errors [8] and finding tandem repeats separated by some distance [9].

Due to the critical nature of this application, an important consideration is programmability. One aspect of this is whether k must be fixed or whether it can be expressed as a fraction of n ; another is whether k or the allowable size of the repeats (minimum or maximum) must be specified in advance. Yet another aspect is running time rather than simple asymptotic bound. Most of the theoretically optimal algorithms depend on properties of suffix trees, such as the fact that useful queries can be executed in $O(1)$. However, although asymptotically insignificant, these queries cause a substantial constant multiplier within the Big- O . Benson especially has done much work on these and other programmability issues (see e.g. [2]).

3 Our Model, Problems We Address, How we solve them

3.1 Model and Problems

Since there are many problems related to analysis of repetitive structures with many parameters and many ways of dealing with them with FPGAs, we felt the need in this first study to examine what could be done with the simplest algorithmic model, starting with the most basic problems, and moving forward from there. Our program is to investigate techniques for analyzing repetitive sequence structure by feeding sequences through the FPGA at streaming rate. By “streaming rate” we mean simply that characters are processed systolically with an emphasis on simple logic.

We quickly found that a two tier structure works well. In the first tier we feed the string through an array of comparitors/counters and get a stream of results for every point. These include the size of the repeat or palindrome about that point and the number of mismatches. In the second tier, which we call post-processing, we decide what information to send off chip, and determine higher order structure such as arrays of repeats.

The following tasks were examined, implemented on an FPGA, and analyzed. Each of these tasks enumerate quantities for strings of arbitrary length but with n determined by available hardware:

1. tandem repeats of length 1 to n with k or fewer mismatches
2. palindromes of length 1 to n with k or fewer mismatches
3. tandem-repeats of length 1 to n with k or fewer mismatches and 1 edit error
4. palindromes of length 1 to n with k or fewer mismatches and 1 edit error
5. tandem arrays of arbitrary length with period from 1 to n

Tasks 1 and 2 we do with basic structures alone, while the other 3 depend on output generated by those structures. Two tasks we do not address here are tandem repeats (or palindromes) with multiple edit errors and tandem arrays

with errors. Dealing with multiple edit errors is most likely to be done best using dynamic programming, while tandem arrays with errors may be best analyzed statistically [2].

3.2 Tandem Repeats With Mismatches

We begin with tandem repeats and tandem repeats with mismatches. Let α_1 and α_2 be two strings with equal length which would be identical but for k mismatches. Let α_1 and α_2 both be shifted to the right by one character with i_1, i_2, o_1 , and o_2 being the characters that get shifted into α_1 and α_2 and out of α_1 and α_2 , respectively. We keep a running count of k as the string is shifted through α_1 and α_2 . There are four cases: two where k is unchanged, $i_1 = i_2$ & $o_1 = o_2$ and $i_1 \neq i_2$ & $o_1 \neq o_2$; one where k decreases, $i_1 = i_2$ & $o_1 \neq o_2$; and one where k increases, $i_1 \neq i_2$ & $o_1 = o_2$. We observe that when α_1 and α_2 are concatenated, then i_2 and o_1 are the same character. It follows that if we stream characters through $\alpha_1\alpha_2$, we can keep a running count of the number of mismatches between α_1 and α_2 simply by, on every iteration, comparing the single character i_2/o_1 with i_1 and with o_2 and adjusting an up-down counter accordingly. We can perform this computation for each length up to $n/2$ by replicating the logic (see Figure 1a) and broadcasting the i_2/o_1 character to each copy. Folding the structure over on itself results in the simple structure shown in Figure 1b.

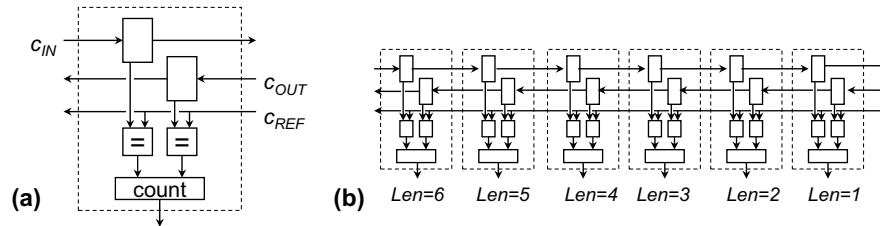


Fig. 1. Structure for tracking tandem repeats of all lengths and all mismatches. Character i_2/o_1 is broadcast to the array of comparators.

3.3 Palindromes With Mismatches

We now describe the structure for palindromes and palindromes with mismatches. A substring about the center $\alpha_1\alpha_2$ is a palindrome of length l when the first l characters of α_2 match those of $\alpha_1^{reverse}$. We now describe a structure to find the number of errors in all palindromes up to length $n/2$. As shown in Figure 2, we again fold the string upon itself. This time, however, instead of comparing

a single mid-point character with all other characters in the string, we perform pair-wise comparisons for all characters from 1 to $n/2$ of $\alpha_1^{reverse}$ and α_2 . The results form a bit vector of length $n/2$. For each length l from 1 to $n/2$, the number of errors in the palindrome is equal to the number of zeros (mismatches) between 1 to l . A simple combinational circuit would perform these sums in a single cycle, but result in unreasonable delay (see Figure 3a). We therefore use the pipelined structure shown in Figure 3b. Note that this structure can easily be extended to detect palindromes separated by a constant distance with the queue shown in Figure 2.

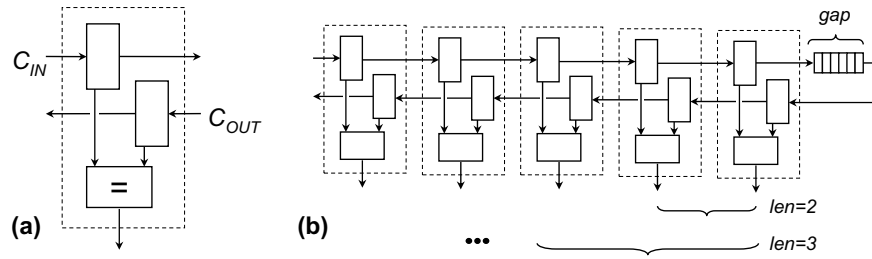


Fig. 2. Part of structure for tracking palindromes repeats of all lengths and all mismatches. Here we generate the initial bit vector.

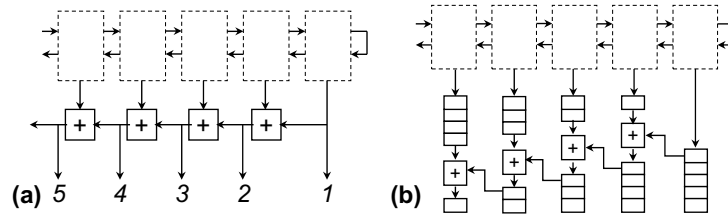


Fig. 3. Part of structure for tracking palindromes repeats of all lengths and all mismatches. Here we sum up the bit vector for each length up to $n/2$.

3.4 Palindromes With k Mismatches and One Edit Error

Extending the palindrome structure to detect a single edit error was done as follows. The circuit described above for k mismatches was replicated two times so that there are three systems running at once. The first is exactly the same as just described. The second is identical except that the array of character-width comparators is offset so that each comparator looks one spot ahead. This, in

combination with the original array of comparators, enables detection of palindromes with a single insertion. Analogously, the third array of comparators looks one spot behind and enables detection of palindromes with a single deletion. For each length palindrome, an insertion or deletion can occur in a number of positions equal to the length. For this reason, another structure had to be added. A two-dimensional array keeps track of whether or not a palindrome for each possible edit error position in each length has a number of matches greater than or equal to the length minus the number of allowable mismatches.

3.5 Tandem Repeats With k Mismatches and One Edit Error

Our method for finding tandem repeats with one edit error is based on the observation that, for every segment $\alpha_1\alpha_2$ observed in streaming mode and of size $2l$, a matching prefix of length p in α_1 and α_2 will be a matching suffix of the same length in $l - p$ cycles. This observation (together with some other simple computations) allow us to generate, on every cycle and for every length l from 1 to $n/2$, the maximal prefix and suffix common to α_1 and α_2 .

A table that is identical to the one used for palindrome detection with one edit error is also implemented here. But instead of looking at the results of comparator arrays, we examine prefixes and suffixes that when aligned leave a single gap (or a single overlap in the deletion case). If the sum of these two counts for each element of the table is greater than the threshold value, the cell is a 1, otherwise it is a zero. To identify a match with an edit error above threshold we OR two stripes of the table (1 for the insertion, 1 for the deletion).

3.6 Precise Tandem Arrays

Finding precise tandem arrays follows immediately from the structure used to find tandem repeats with k mismatches. When the counter for any length l (the l -counter) reaches l , this indicates a tandem repeat at that position with no errors. As the string is streamed character by character, the precise tandem array of that period continues for as long as the contents of the l -counter remains equal to l . The correctness follows immediately from the basic algorithm. When the l -counter is equal to l , each pair of characters “falling off the end” of $\alpha_1\alpha_2$ is a match. Therefore, the only way for the value l to be maintained in the l -counter is for the incoming pair also to be a match. This indicates that the precise tandem array has been preserved for another character.

Precise tandem arrays of period l ($|\alpha|$) are enumerated with an additional counter and logic to detect when the l -counter for tandem repeats is equal to l . When this happens initially, the *tandem array* l -counter is set to one. Every cycle that l is maintained, the *tandem array* l -counter is incremented. When the tandem *repeat* l -counter is no longer equal to l , the tandem array count stops

and its value can be output. The number of repeats in the tandem array is the number of cycles for which the tandem repeat counter was l divided by l .

3.7 Filtering Results

The issue of selecting data for output arises from the very high rate at which results are generated: there is no way to get them all off chip, even if they were all interesting, which they almost certainly are not. In practice, the choice will depend on the bioinformatic application. Two data sets that are easy to generate from the data streams generated by the preceding structures are as follows. For every character position, output:

- a bit vector indicating, for a fixed range, which repeat (or palindrome) lengths have k or fewer match errors.
- a fixed number of the longest repeats (or palindromes) with fewer than k errors.

Alternatively, a list of data guaranteed to be bounded can be output. One example is a listing of the positions and sizes of precise tandem arrays of minimum length and bounded period.

4 Analysis

We look first at asymptotic complexity, then performance numbers.

4.1 Comments on Asymptotic Complexity

Studies of this kind expose the inherent limitations of asymptotic algorithmic complexity but a discussion is still illuminating. There are two issues, the size of the constants within the Big-O and the realistic bounds of n for various practical implementations, both FPGA-based and PC-based. The complexity of all the algorithms described here is $O(N)$, the length of the input string (on which there really is no practical limit). Exactly what we are solving depends on how strictly we hold to the definition of Big-O. For example, are we finding all repeats of length n , or all repeats of constant length (i.e. for which we have room to process on the chip)? If we take the broad view, then we take n to be unbounded in the asymptotic sense. Then comparing our results with the bounds presented in Section 2 (divided by n to account for the application of n processors in the FPGA algorithm), we find that perhaps our only optimal result is that of finding all tandem repeats with an unbounded number of mismatches. On the other hand, the constant within our $O(N)$ is so small that it compares with the

time it would take to distribute data in a parallel system, an $O(N)$ task that is usually ignored when parallel algorithms are analyzed. In that interpretation all of our results are optimal, even if the problem sizes are bounded.

Program implementations of the optimal serial algorithms run into similar problems. Although they do not “break” at some fixed n (the capacity of the chip), they can become very slow long before they approach a hard bound such as the limit of virtual memory capacity. As mentioned earlier most of the theoretically optimal algorithms depend on properties of suffix trees, such as the fact that useful queries can be executed in $O(1)$. However, although asymptotically insignificant, these queries cause a substantial constant multiplier within the Big- O . In particular, we have found that a very simple program implementation was many times faster than the code for the theoretically optimal algorithms with the performance difference *increasing* with n , not decreasing as one would expect if the constants in the big- O were becoming less significant.

4.2 Timing and Problem Size

All designs described in Section 3 have been implemented in VHDL, simulated, synthesized, and undergone place-and-route. We used the Xilinx tool set. When using the Xilinx XC2VP100 as the target, we found all post place-and-route cycle times to be almost exactly 5ns. Table 1 gives the maximum size of each problem that fits on the chip.

Task	n
1. All tandem repeats of size l from 1 to n with up to n mismatches	1024
2. All palindromes of size l from 1 to n with k mismatches up to n	256
3. Same as 1 with one edit error	64
4. Same as 2 with one edit error	40
5. Tandem arrays with period of size l from 1 to n	1024

Table 1. Maximum size of each problem that fits on the FPGA.

The following table compares the timing of the FPGA implementation with that of a C program running on a 3GHz Xeon-based workstation-class PC.

5 Discussion and Extensions

We have investigated the potential of FPGA acceleration of a critical set of problems in sequence analysis. We have found substantial speed-up and the promise of a significant new application for computational coprocessors based on FPGAs. The potential is for large-scale genome processing at the desktop.

maximum repeat	Serial version for Task 1	FPGA Version for Task 1	Serial version for Task 3	FPGA Version for Task 3
32	1.3us	5ns	10.5us	5ns
64	2.3us	5ns	36us	5ns
128	4.6us	5ns		
256	8.8us	5ns		
512	17.1us	5ns		
1024	33.1us	5ns		

Table 2. Time given is the time per character for arbitrary length strings.

We are in the process of installing these algorithms on an Avnet development board with a Xilinx XC2VP20. Unfortunately, we are still having trouble getting the high-speed links working – however we expect to have results from this system shortly.

There are many possible extensions to this work. Several significant articles a year are appearing in this area exploring new problem definitions and algorithms. The most fruitful next target, however, is likely to be integration of an FPGA-based accelerator into a production system for sequence analysis.

References

- Schuler, G.: Sequence alignment and database searching. In Baxevanis, A., Ouellette, B., eds.: *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins*. Wiley (2001)
- Benson, G.: Tandem repeats finder: A program to analyze DNA sequences. *Nucleic Acids Research* **27** (1999) 573–580
- Gusfield, D.: *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, U.K. (1997)
- Guccione, S., Keller, E.: Gene matching using JBits. In: *Proc. of Field Programmable Logic and Applications*. (2002) 1168–1171
- Time Logic Corp. www.timelogic.com: Web Site. (2003)
- Yu, C., Kwong, K., Lee, K., Leong, P.: A Smith-Waterman systolic cell. In: *Proc. of Field Programmable Logic and Applications*. (2003) 375–384
- Main, M., Lorentz, R.: An $O(n \log n)$ algorithm for finding all repeats in a string. *J. Algorithms* **5** (1984) 422–432
- Landau, G., Schmidt, J., Sokol, D.: An algorithm for approximate tandem repeats. *J. Computational Biology* **8** (2001) 1–18
- Brodal, G., Lyngso, R., Pedersen, C., Stoye, J.: Finding maximal pairs with bounded gap. *J. Discrete Algorithms* **1** (2000)