Hardware-Based Implementation of the Common Approximate Substring Algorithm

Kenneth B. Kent[†] ken@unb.ca
Sharon Van Schaick[†] j5mym@unb.ca
Jacqueline E. Rice[‡] j.rice@uleth.ca
Patricia A. Evans[†] pevans@unb.ca

Faculty of Computer Science[†]
University of New Brunswick
Fredericton, NB, Canada

Dept. of Mathematics & Computer Science[‡]
University of Lethbridge
Lethbridge, AB, Canada

Abstract

An implementation of an algorithm for string matching, commonly used in DNA string analysis, using configurable technology is proposed. The design of the circuit allows for pipelining to provide a performance increase. The proposal is unique in that we suggest a design that is specific to certain parameters of the problem, but may be reused for any particular instance of the problem that matches these parameters. The use of a Field Programmable Gate Array allows the implementation to be instance specific, thus ensuring maximal usage of the hardware. Analysis and preliminary results based on a prototype implementation are presented.

1 Introduction

A recent solution to such difficult problems as DNA matching, Boolean satisfaction, and image compression [1, 2, 3, 4] has been to make use of hardware in performing the most computationally-intensive portions of the algorithm. The problem is that design and fabrication of computer chips is generally very expensive. However, the use of Field Programmable Gate Array (FPGA) technology provides an alternative option that is far more feasible. By combining software and configurable hardware it is possible to retain the flexibility required to apply the algorithm to many different parameters, as well as gain the acceleration from implementing a portion of the algorithm on the FPGA. In this work we propose a parameter-specific implementation in which a finite-state-machine (FSM) is implemented on a FPGA, allowing for very fast comparisons between the pattern being sought and the sequences being searched. By parameter-specific we refer to the fact that the hardware design must be regenerated if either of the two fixed parameters required for the search are changed. These parameters are the length of the sequence being sought and the number of errors permitted in determining a match. This differs from instance-specific solutions in that our proposed solution may be applied to any number of problem instances, as long as the given parameters remain fixed [9]. This work also addresses the issue of maximizing the usage of the FPGA, and presents a number of options and related issues for this aspect of the design.

1.1 Common Approximate Substring Matching

The problem to be solved is that of finding a similar pattern of symbols within all of a given series of sequences, allowing a certain amount of error (see Figure 1) [10, 11]. The purpose behind this is that the discovery of sequence homology to a known protein or family of proteins often provides the first clues about the function of a newly sequenced gene [5]. Discovering homologous sequences and families frequently starts with searching for common motifs [5, 7]. Since the introduction of a fast method for comparing biological sequences, DNA and protein sequence comparison have become routine steps in biochemical characterization [6]. This computation is critical for sequence analyses and tends to be very time consuming.

Simplified Example

TGACTCGACC
TACTGCCTCG
CTGGCTAATA
ATTCCTGACT

Figure 1: An example of common approximate substrings of length 5 with an error of 1.

In the motif search the goal is to find similar sequences of symbols of a given length *m* within the database of DNA sequences. The number of errors that are allowed is fixed to a value of *d*. An error is encountered when a single symbol within the sequence being searched does not match the sequence being sought. In this work we limit the allowable

errors to simple replacements of a particular symbol within another; shifts or gaps within the sequence are not considered.

Finding these common approximate substrings is a two-phase process [10]. First, substrings from each sequence that are within distance 2*d* from an initial reference sequence are located. If a substring is not within 2*d* of this reference substring it could not possibly be part of a solution set including this reference. This produces a reduced search space where a solution may exist. Second, these similar substrings are compared as a block, aligning the positions of the substrings into columns, and searched for a string that is within distance *d* of each similar substring (see Figure 2). The common approximate substring cannot exist without the corresponding substrings in each sequence being within 2*d* errors of each other [7]; for many uses of common approximate substrings, such as finding motifs and regions of high similarity, locating these similar substrings is often sufficient as they are statistically significant [8].

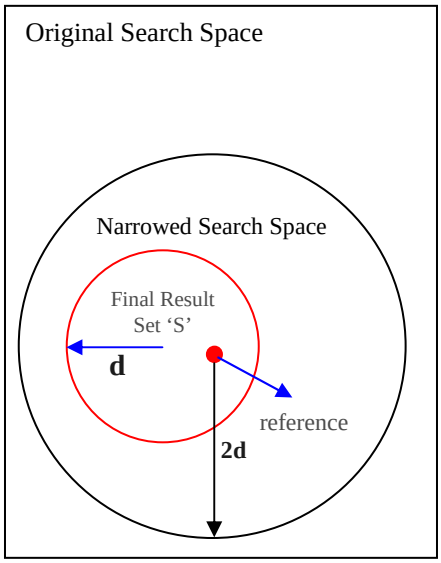

Figure 2: Narrowing down the original search space.

In this paper we outline the FPGA implementation for both phases of this algorithm. Substrings of the first sequence are compared to substrings from each of the other sequences, with desired distance $k = 2d$ (where d is the number of errors allowed for the common approximate substring). When a match to each sequence is found, the entire matching set of substrings will be recorded as a set of similar regions, and can be analyzed in parallel to search the space they span for a common substring within distance *d*.

2 Hardware Design

The completed hardware design uses the Common Approximate Substring algorithm that was described in Section 1. The hardware retrieves sequences of length l from an external input memory, performs the Common Approximate Substring algorithm and stores resulting substrings in the external memory. The following subsections describe the system in detail.

2.1 A Black-Box View

The typical data-path of the sequence and its substrings is shown in Figure 3 as the solid arrows. In Stage 1, a substring of length *m* is taken from the sequence. This substring is initially compared with a reference substring to check that it is within 2*d* errors of that reference. The substring is then passed to Stage 2 and a new sequence is fetched for Stage 1. In Stage 2, the *m*-length substring is compared against all substrings in a comparison memory to determine all substrings that are within *d* errors of this substring. The comparison memory was previously seeded with all *m*-length substrings that are within *d* errors of the reference substring.

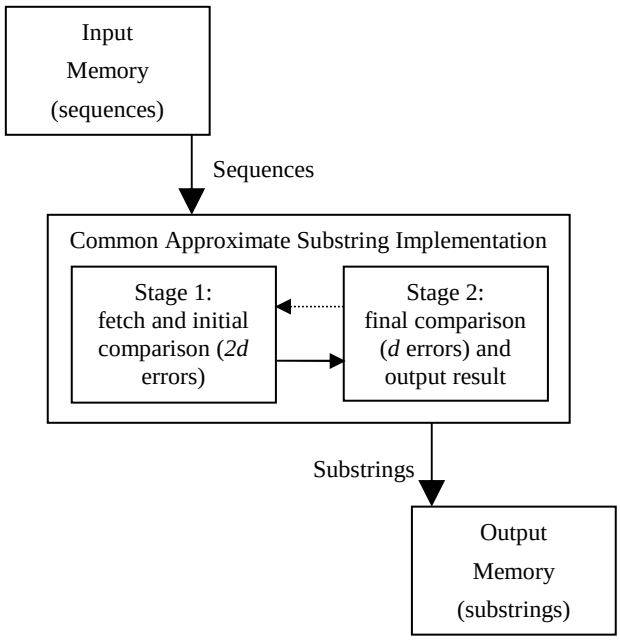

Figure 3: Black box system view.

The design is pipelined such that both the comparison in Stage 1 and the comparison in Stage 2 can occur at the same time on different substrings. While Stage 2 is working on a previously passed substring, reducing the potential solution space, Stage 1 is working on a new substring from a new sequence.

If the substring, in Stage 2, is found to not be within *d* errors of any existing substrings, then a back-track request is placed to Stage 1 (represented by the broken arrow in Figure 3). Otherwise, it is passed to an external output memory which will contain the solution substrings. Contained with each substring is the location of the parent sequence it was fetched from in the input memory and the substring's location within its parent sequence.

2.2 Design Modules

The chip design contains four modules, each with its own purpose and functionality. The four modules are: the *Interface*, the *Seeker*, the *Seeker Result Queue* and the *Verifier*. We fully designed and implemented the *Interface*, *Seeker* and *Verifier* ourselves; the *Seeker Result Queue* is implemented using an IP core.

2.2.1 Interface

This module interfaces with the external input memory and the *Seeker*. The *Interface* retrieves sequences from the external memory and passes them along to the *Seeker*. If word alignment is required (the external memory is not as wide as a sequence) the *Interface* continues fetching from memory until it has fetched the entire sequence. If the last line fetched from memory contains the end of one sequence and the beginning of the next sequence, then the *Interface* only keeps the characters it needs for the sequence and stores the starting location of the next sequence in a placement buffer. It will then pass the completed sequence on to the *Seeker*.

The *Interface* also handles back-track requests. A back-track request occurs when a substring is passed through the system to the *Verifier* that has more than *d* errors (see section 2.2.5 for more detail). The *Interface* responds to this request by retrieving the incorrect substring's parent sequence and passing it along to the *Seeker* again.

2.2.2 Seeker

The very first sequence that the *Seeker* receives from the *Interface* is used as a reference sequence. From this sequence, the *Seeker* takes an *m*-length substring of characters and holds this as the reference substring. This reference substring is also passed along to the other modules of the system as the first member of the solution set. When the *Seeker* receives future sequences from the *Interface* they are manipulated as follows:

1. an *m*-length substring is taken from the sequence
2. this substring is compared to the reference substring
3. if there are less than 2*d* errors between them, the substring is passed to the *Seeker Result Queue*
4. if this is not the case, a new *m*-length substring is taken from the sequence by shifting left by one character

This process repeats from step 2 until all possible *m*-length substrings have been taken from the sequence. If the *Seeker* proceeds to perform the comparison on all possible substrings from the sequence and none are within 2*d* errors of the reference, a new reference is fetched and the whole process begins again. A new reference is obtained by taking a new substring from the very first sequence and holding it as the reference substring.

2.2.3 Seeker Result Queue

The *Seeker Result Queue* exists only for the purpose of being a temporary resting place for substrings between modules. The Seeker module passes substrings that pass the first comparison test to this module. The Seeker tags each substring with the address of its parent in the input memory and the substring's location within its parent. These substrings wait here to be retrieved by the *Verifier* module.

2.2.4 Verifier

The second comparison test that the design performs is completed by the *Verifier*. The *Verifier* module interfaces with the *Seeker Result Queue*, the external output memory and the comparison memory to perform the test. The comparison begins when the *Verifier* retrieves a new input substring from the *Seeker Result Queue*.

If this is the first substring the *Verifier* has fetched then it begins the process of seeding the comparison memory. It takes this substring and performs all possible permutations that create m-length substrings within d errors of the reference substring. These substrings are all stored in the comparison memory along with a tag.

The tag is added for the process of eliminating substrings that would not be part of the solution set. Subsequent input substrings passed to the *Verifier* are compared to all substrings in the comparison memory. When an input substring is being compared to substrings in the comparison memory and is found to not be within *d* errors of a comparison substring, that comparison substring is tagged. If the input substring proceeds to tag all substrings that were not previously tagged then a back-track request is placed. The *Verifier* sends the appropriate signal to the *Interface* and resets all substrings that were recently tagged. Otherwise, the input substring is placed in the external output memory as a member of the solution set.

2.2.5 *Back-Track Requests*

As the substring is passed through the system it carries with it (in a locator sequence) the location of its parent sequence in the input memory and the substring's location within its parent sequence. When the *Verifier* encounters a situation where it needs to send a back-track request, it only needs to send this locator sequence to the *Interface*.

The *Interface* takes this information and retrieves the appropriate starting sequence from the input memory. The *Interface* then passes this sequence along to the *Seeker* as if it were any other sequence it fetches. The difference in this case is the *Interface* also sends part of the locator with the sequence. It sends the location of the substring within the parent sequence. With this information the *Seeker* skips forward in the sequence to the point indicated by the locator and begins processing the sequence from this point.

3 Hardware Implementation

The final design is implemented targeting a Stratix EP1S40F780 device on the Altera Nios Development board containing 1 Mbyte SRAM (16-bit wide), 16 Mbytes SDRAM (32-bit wide) and 8 Mbytes Flash on-board memories [12]. The implementation is completed and simulated using the Altera Quartus II Development tools. Figure 4 is a detailed schematic of the system as implemented using the aforementioned development environment. For simplicity, the external memory used to store the sequences to process and store results is omitted. This memory is connected to the *Interface* component in the schematic using standard memory interface read/write, address, and data signals. Currently, the operating frequency of the design is 65.44 MHz and utilizes 14% of the available logical element (LE) resources within the FPGA.

4 Circuit Operation

The implementation processes sequences at a rate of one character comparison per clock tick. This one comparison being completed consists of a comparison performed by the *Seeker* and one by the *Verifier* at the exact same time. The parallelism of the system allows this to occur. At the same time a substring is being checked against the reference substring in the *Seeker*, another earlier substring is being checked against the output memory in the *Verifier*.

Simple test cases were used to retrieve results and verify the design. These test cases were composed of 8 sequences with 8 characters in each sequence. The test parameters involved finding a length-5 substring in each sequence that had at most 2 errors between it and any other length-5 substring in the solution. These test cases are defined in Table

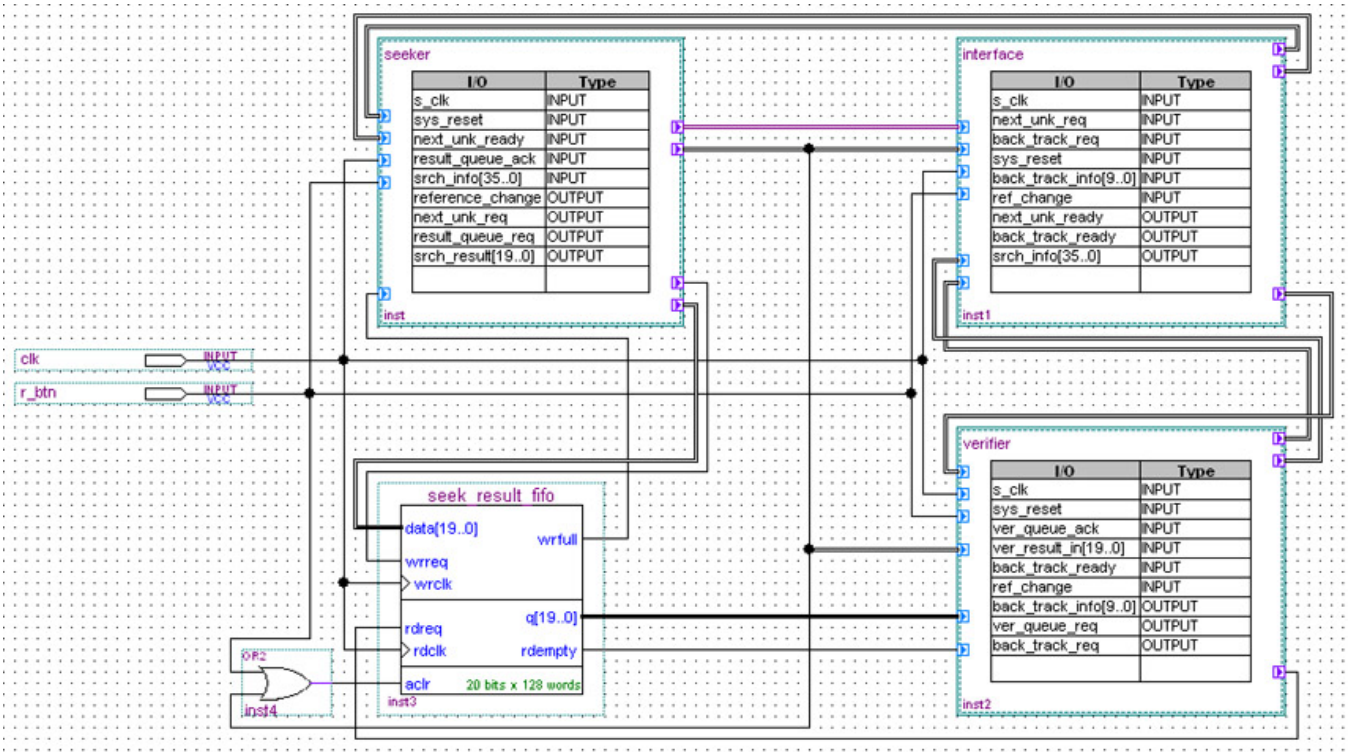

Figure 4. Block Diagram/Schematic of the hardware implementation in the Altera Quartus II tools.

1. Test Case A consists of 8 sequences which have a solution (the solution is the highlighted substrings). Test Case B has no solution and is an example of a worst-case scenario for testing the system.

Test Case A	Test Case B
ACTGTTTT	ACTGTTTT
AGTTTAGC	TATGTTTA
GCATTTTT	GCATTTTT
AAAGGTTT	AAAGGTTT
GGCGTATT	GGCGTATT
TGGATTTT	TGGATTTT
GTGCTCTT	GTGCTCTT
CCTGTGAT	CCTGATAG

Table 1: Simple test cases for the implementation.

The worst-case scenario is defined by a test case where every *m*-length substring from each sequence is passed through the *Seeker* and the *Verifier* and only the last substring results in the system determining there is no solution. This occurs when the only possible solution substring in each sequence is at the end of the sequence (top 7 highlighted substrings in Test Case B, Table 1) and that substring in the last sequence is not within *d* errors of the rest of the solution (last highlighted substring in Test Case B, Table 1).

Examining Test Case A in detail, the *Interface* begins the processing of the sequences by loading the first sequence (GGGGTTTT) into a buffer. The *Interface* then passes this sequence, along with its address in memory, to the *Seeker* and retrieves the next sequence to be passed along.

The *Seeker* recognizes this sequence as the first sequence by determining that its memory address is 0 and places it in a reference buffer. The *Seeker* then retrieves a 5 character substring from the beginning of the sequence (GTTTT) to be the first reference substring. Sequences are processed from the right to the left so the beginning of the sequence is the rightmost character. This substring is sent to the *Seeker Result Queue* along with its address and its location in the sequence (currently both 0). The location of the substring within the sequence is called its position.

The *Verifier* retrieves this reference substring from the *Seeker Result Queue*. As this is the first substring to be passed to the *Verifier* it begins the process of seeding the comparison memory. The *Verifier* proceeds to find all possible m-length strings of characters (in the character set) with a difference of d errors from the reference substring. These strings are stored in the comparison memory. An example of the beginning of the comparison memory is show in Table 2. The *Verifier* then passes the reference substring along to the external output memory.

When the *Seeker* receives the signal from the *Interface* that the second sequence (AGTTTAGC) is ready it retrieves it and stores it in a buffer. The *Seeker* takes a 5 character substring from the front of the sequence (TTAGC) and compares it, one character at a time, to the reference substring (GTTTT). This comparison determines there are 4 errors between these substrings which is equal to the 2*d* (or 4) errors allowed. This substring is passed on to the *Seeker Result Queue* with the address 1 and position 0.

Comparison Memory for A
GTTTT GATTA
GTTTA GCTTA
GTTAA GGTTA
GTTCA TTTTA
GTTGA ATTTA
GTATA CTTTA
GTCTA GTTAT
GTGTA ...

Table 2: Sample comparison memory for test case A.

The *Verifier* then fetches this second input substring (TTAGC) from the *Seeker Result Queue* and begins its comparison process. The *Verifier* fetches one substring stored in the comparison memory (GTTTT) and compares it, one character at a time, to the input substring (TTAGC). If the comparison substring fetched is tagged the *Verifier* ignores it and fetches the next comparison substring. In this case this is the first substring to be compared to the comparison substrings so none are tagged. This comparison determines there are 4 errors between these substrings. At this point, the *Verifier* tags the comparison substring with the input substring's address (1). The *Verifier* then proceeds to repeat these steps for all substrings in the comparison memory. If the input substring happens to tag all of the comparison substrings (meaning TTAGC is not within *d* errors of any of the comparison substrings), then a backtrack request is sent to the *Interface* and the *Verifier* untags all comparison substrings tagged by the input substring.

The *Verifier* sends a signal to the *Interface* that a backtrack must occur and it sends the address (1) and position (0) of the problem substring. The *Interface* stores the address of the sequence it is currently working on as the *Interface* has been fetching sequences for the *Seeker* the whole time the *Verifier* has been working. It then fetches the se-

quence that contains the problem substring from memory address 1. It passes this sequence along to the *Seeker* with the address 1 and position 1. The *Interface* then continues working as normal, by fetching the sequence from the memory address it stored.

The *Seeker* treats the sequence passed to it as it would any other sequence; except that it sees the position is not a 0. When the position is not a 0 it must select 5 characters for the new substring starting at the specified position (which is 1), not the front of the sequence. The substring that is fetched (TTTAG) is compared to the reference substring (GTTTT) as any other substring would.

The process of retrieving a substring from a sequence, comparing it to a reference, comparing it to all substrings in the comparison memory and back-tracking if necessary repeats. The process only stops when one of the following occurs: every sequence has one substring in the external output memory, or the system has exhaustively searched all possible substrings in one of the sequences and not found a common substring within the allowed d errors of each other comparison substring and the reference (GTTTT).

When the second case does occur the *Seeker* takes a new substring from the reference sequence in the reference buffer that is one character over from the previous substring (GGTTT). This becomes the new reference and the *Seeker* sends a signal to all other components that the reference has changed. At this point all of the other components reset to their initial states and the process begins again.

5 Analysis & Results

As character comparisons are the dominant cost to the system, we define our results on this basis. Let num_{subs} be the number of *m*-length substrings possible in a sequence, num_{seqs} be the number of sequences, and seq_{len} be the length of a sequence. The number of *m*-length substrings possible in a sequence, num_{subs} is:

$$num_{subs} = seq_{len} - m + 1.$$

This number is also the same as the number of possible reference substrings in the reference sequence, num_{refs}.

The maximum number of single character comparisons possible in both the *Seeker* and *Verifier* is:

$$num_{comp} = num_{subs} * num_{seqs} * num_{refs} * m$$

Let the implementation clock rate be clk_{rate} and clk_{tick} be the number of clock ticks per comparison. Currently, clk_{tick} is 1 in this implementation with a 4 character set. Assuming that the input sequences are created as such that no possible solution exists, the time the system takes to complete this scenario test case would be:

$$time = \frac{num_{comp} * clk_{tick}}{clk_{rate}}$$

As an example, the test case defined in Test Case B above would result in 560 character comparisons being completed in total.

The complexity of the execution increases as potential solutions exist in the input strings. Consider that each potential solution results in the seeding of the *Verifier* memory with all possible solutions that can exist. Once seeded, each potentially matching substring is checked against all of these strings to remove potential solutions. To consider the complexity of this, the size of the memory required to contain the potential solutions generated from the reference string is:

$$mem_{size} = 1 + (c-1)\binom{m}{1} + \sum_{i=2}^{d} (c-1)^i \binom{m-(i-2)}{2}$$

where *c* is the number of characters (i.e. A, C, T, G = 4). This can result in an increased runtime; however the input data typically does not generate a lot of potential solutions. We are currently in the process of executing several benchmark tests with the circuit design to provide specific test performance results.

6 Future Work

Future work to be completed on this project involves improving the parallelization of the hardware design. As was seen from the results we obtained not all of the available resources on the FPGA chip are being utilized. By creating multiple copies of the *Seeker*, *Seeker Result Queue* and *Verifier* modules we would be able to begin each *Seeker* with a different reference substring from the initial sequence or place a different sequence for comparison to the same reference in each *Seeker*. Doing either would use some of the remaining unused resources on the FPGA to further increase performance.

As well as parallelization the design could be made more flexible with the further use of parameterization. This would give the user the ability to define the parameters of the problem and the hardware could conform to the user specification. The user would be able to specify the length, l, of the input sequences, the length of the substring, m, to be taken and the number of errors, *d*, allowed between substrings. From these parameters a circuit design could be generated to solve the appropriate size problem.

7 Acknowledgments

We would like to acknowledge Thomas Hall, MCS Student, Faculty of Computer Science, University of New Brunswick for his contribution to the initial design of the system. We would also like to acknowledge funding received from various sources for this work. An NSERC USRA was granted to the second author along with NB Seed funding. NSERC funding as well as CMC hardware, software and funding were granted to the first author.

8 References

[1] Lee, H. and Ercal, F. *RMESH Algorithms for Parallel String Matching*. Proceedings of the 3rd International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN'97), pp. 223-226, 1997.

[2] Yamaguchi, Y., Miyajima, Y., Maruyama, T. and Konagaya, A. *High Speed Homology Using Run-Time Reconfiguration*. Proceedings of FPL 2002, LNCS 2438, Springer-Verlag, pp. 281-291, 2002.

[3] Zhong, P. M., Martonosi, M., Ashar, P. and Malik, S. *Using Configurable Computing to Accelerate Boolean Satisfiability*. IEEE Transactions on Computer-Aided Design, volume 18, number 6 (June) 1999.

[4] Simpson, A., Hunter, J., Wylie, M., Hu, Y., and Mann, D. *Demonstrating Real-Time JPEG Image Compression-Decompression Using Standard Component IP Cores on a Programmable Logic Based Platform for DSP and Image Processing*. Proceedings of FPL 2001, LNCS 2147, Springer-Verlag, pp. 441-450, 2001.

[5] Altschul, S. F., Gish, W., Miller, W., Meyers, E. W. and Lipman, D. J. *Basic Local Alignment Search Tool*. Journal of Molecular Biology, volume 215, number 3, pp. 403-410, October 1990.

[6] Pearson, W. R. *Flexible Sequence Similarity Searching With the FASTA3 Program Package*. Methods in Molecular Biology, volume 142, pp. 185-219, 2000.

[7] Pevzner, P. and Sze, S.-H. *Combinatorial Approaches to Finding Subtle Signals in DNA Sequences*. Proceedings of the Eighth International Conference on Intelligent Systems for Molecular Biology (ISMB 2000), pp. 269-278, 2000.

[8] Bailey, T. L. and Gribskov, M. *Methods and Statistics for Combining Motif Match Scores*. Journal of Computational Biology, volume 5, pp. 211-221, 1998.

[9] Kent, Kenneth B., and Serra, Micaela. *Using FPGAs to Solve the Hamiltonian Cycle Problem.* International Symposium on Circuits and Systems (ISCAS 2003), vol. III, pp. 228 – 231, May 2003.

[10] Smith, Andrew D. *Common Approximate Substrings.* PhD thesis, Faculty of Computer Science, University of New Brunswick, October 2003.

[11] Evans, Patricia A., Smith, Andrew D., and Wareham, Harold T. *On the Complexity of Finding Common Approximate Substrings*. Theoretical Computer Science, 306(I-2); pp. 407-430, 2003.

[12] *NIOS Development Board Reference Manual, Stratix Professional Edition*. http://www.altera.com.cn/literature/ manual/mnl_nios2_board_stratixII_2s60.pdf, September 2004, v1-1, accessed Feb 20, 2005.