Hybrid Architecture for Accelerating DNA Codeword Library Searching

Qinru Qiu   Daniel Burns∗   Qing Wu   Prakash Mukre

Department of Electrical and Computer Engineering, Binghamton University, Binghamton, NY 13902

∗Air Force Research Laboratory, Rome Site, 26 Electronic Parkway, Rome, NY 13441

qqiu@binghamton.edu, Daniel.Burns@rl.af.mil, qwu@binghamton.edu, pmukre1@binghamton.edu

Abstract — **A large and reliable DNA codeword library is the key to the success of DNA based computing. Searching for the set of reliable DNA codewords is an NP-hard problem, which can take days on the state-of-art high performance cluster computers. This work presents a hybrid architecture that consists of a general purpose microprocessor and a hardware accelerator for accelerating the discovery of DNA reverse complement, edit distance codes. Two applications of this architecture were implemented and evaluated, including a code generator that uses a genetic algorithm (GA) to produce nearly locally optimal codes in a few minutes, and a code extender that uses exhaustive search to produce locally optimum codes in about 1.5 hours for the case of length 16 codes. The experimental results demonstrate that the GA can find ~99% of the words in locally optimum libraries, and that the hybrid architecture provides more than 1000X speed-up compared to a software only implementation.**

I.   INTRODUCTION

The DNA molecule is now used in many areas far beyond its traditional function. The first DNA-based computation was proposed by Adleman [1]. It demonstrates the effectiveness of using DNA to solve hard combinatorial problems. DNA molecules have also been used as information storage media and three dimensional structural materials for nanotechnology.

One of the major concerns of DNA computing is reliability. In DNA computing, the information is encoded as DNA strands. Each DNA strand is composed of short codewords. DNA computing is based on the *hybridization* process, which allows short single-stranded DNA sequences (i.e. *oligonucleotides*) to self-assemble to form long DNA molecules. The reliability of the computing is determined by whether the oligonuleotides can hybridize in a predetermined way. The key to success in DNA computing is the availability of a large collection of DNA codeword pairs that do not crosshybridize.

Various quality metrics have been proposed to guide the construction process [1]-[5]. The computation of these metrics dominates the run time of the code building process. While metrics based on the Gibbs energy and nearest neighbor thermodynamics and consideration of secondary structure formation give accurate measurement of hybridization, they are computationally costly, motivating the use of simplified metrics. One such metric is the *Levenshtein distance*, or the so-called *deletion-correcting* or *edit distance*, which has been used to construct DNA codes [6].

Regardless of the quality metric used, composing DNA codes is NP-hard because the number of potential codewords that must be searched increases exponentially with the length of the DNA codewords. Exhaustive checking is generally impractical for words of length greater than about 12 base pairs. Various algorithms have been proposed for building DNA codes, including the GA [7], Markov processes [8], and Stochastic methods [9]. Recent work [10] has shown that a hybrid GA blended with Conway's lexicode algorithm [11][12] achieves better performance than either alone in terms of generating useful codes quickly.

Search methods for DNA codes are extremely time-consuming, and this has limited research on DNA codeword design, especially for codes of length greater than about 12-14 bases. Theory is lacking to provide tight upper bounds on the size of codeword sets, and the best known bounds are based on experiments. For example, the largest known reverse complement edit distance DNA codeword library (length 16, edit distance 10) consist of 132 pairs, composing such codes can take several days on a cluster of 10 G5 processors.

This paper focuses generally on speed-up techniques for the composition of reverse complement, edit distance, DNA codes of length 16, using a modified genetic algorithm that uses a locally exhaustive, mutation-only heuristic tuned for speed. Ongoing work to be reported elsewhere is addressing extensions to metrics involving nearest neighbor thermodynamics, a more general GA, codewords of length 32.

More specifically, we report a novel accelerator for DNA codeword composition that incorporates a hardware GA, hardware edit distance calculation, and hardware exhaustive search. Hardware exhaustive search extends an initial codeword library by doing a final scan across the entire universe of possible codewords, yielding a known locally optimum code. The proposed architecture consists of a host PC, a hardware accelerator implemented in reconfigurable logic on a *field programmable gate array* (FPGA) and a software program running in a host PC that controls and communicates with the hardware accelerator. The characteristics of the proposed architecture are as follows:

1. High performance. It utilizes programmable logic devices to enable pipelined and massively parallel processing of the data. Compared with software-only approaches, the new architecture can provide more than 1000X speed-up. For example, instead of 52 days, it only takes 1.5 hours to scan

the entire codeword space and to find all additional words that must be added to produce a locally optimum code.

2. High flexibility. The hardware accelerator can be configured by software program, and presently it can be run on a workstation PC equipped with an FPGA board, or on a notebook computer equipped with a PCMCIA FPGA card.

3. User friendly. The hardware accelerator is transparent to the user. Its control and access is accomplished by memory reads and writes based on a set of given protocols.

The remainder of this paper is organized as follows: Section II provides the necessary biological background and terminology. Section III introduces the problem definition and the genetic algorithm for DNA codeword search. Section IV gives the detailed information about how to accelerate the GA search fitness calculation. Sections V and VI provide details about the hybrid architecture and present a performance comparison between the software version of the GA and the best known (Markov) algorithm found in the literature, and early results on locally optimum codes. Finally, conclusions are given in Section VII.

II.   BACKGROUND

The DNA molecule is a nucleic acid. It consists of two oligonucleotide sequences. Each sequence consists of a sugar-phosphate backbone and a set of *nucleotides* (also called *bases*) connecting with the backbone. The oligonucleotide sequence is oriented. One end of the sequence is denoted as 3' and the other as 5'. Only strands of opposite orientation can form stable duplex.

There are four types of bases: Adenine, Thymine, Cytosine, and Guanine. They are denoted briefly as A, T, C, and G respectively. Each base can pair up with only one particular base through hydrogen bonds: A+T, T+A, C+G and G+C. Sometimes we say that A and T are complementary to each other while C and G are complementary to each other. A Watson-Crick complement of a DNA sequence is another DNA sequence which replaces all the A with T or vise versa and replaces all the T with A or vise versa, and also switches the 5' and 3' ends. A DNA sequence binds most stably with its Watson-Crick complement. The stability of the binding is determined by the *free energy* of the hydrogen bonds.

The calculation of the free energy involves many considerations. In this paper, we only consider the first order effect, and use the number of Watson-Crick pairs between two DNA sequences to represent their bonding strength. Such approximation is widely adopted by the research works in DNA codeword design [6][12]. Furthermore, the DNA sequences of length 10 or greater are usually considered to be flexible [6]. Therefore, the binding strength of two DNA strands is measured by the length of the longest complementary subsequence (not necessarily contiguous) of one strand and the reverse of the other. For example, Figure 1 shows two DNA strands that bind with 5 Watson-Crick pairs. The longest complementary sequence between two flexible DNA strands, A and B, is the same as the *longest common sequence* (*LCS*) between A and $\overline{B}$ [6].

$$5' \ A \ A \ C \ G - T \ \overline{G}^{3'}$$
$$| \ | \quad | \quad | \ |$$
$$3' \ T \ T - C \ G \ A \ C_{5'}$$

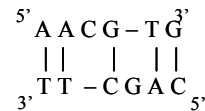**Figure 1 Binding between DNA strands.**

III.   PROBLEM FORMULATION AND OPTIMIZATION ALGORITHM

We consider each DNA codeword as a sequence of length n in which each symbol is an element of an alphabet of 4 elements. The longest common sequence between DNA strands A and B is denoted as $LCS(A, B)$. In this work, we focus on searching for a set of DNA codeword pairs S, where S consists of a set of DNA strands of length n and their reverse complement strands e.g. $\{(s_1, \overline{s_1}), (s_2, \overline{s_2}), …\}$, where $(s_1, \overline{s_1})$ denotes a strand and its Watson-Crick complement. The problem can be formulated as the following constrained optimization problem:

$$\max |S| \tag{1}$$

$$\text{s.t. } LCS(s_1, \overline{s_1}) \le \sigma, \ \forall s_1 \in S, \tag{2}$$

$$LCS(s_1, s_2) \le \sigma, \forall s_1, s_2 \in S \tag{3}$$

$$LCS(s_1, \overline{s_2}) \le \sigma, \forall s_1, s_2 \in S, \tag{4}$$

where σ is a predefined threshold. Equation (1) indicates that our objective is to maximize the size of the DNA codeword library. The first constraint specifies that a DNA codeword in the library cannot bind with itself. The second and the third constraints specify that a DNA codeword in the library cannot bind with another library word or its Watson-Crick complement. Both of these two constraints must be satisfied because a DNA strand always occurs with its Watson-Crick complement.

A genetic algorithm (GA) is a stochastic search technique based on the mechanism of natural selection and recombination. Solutions, which are also called *individuals*, are evolved from generation to generation, with *selection*, *mating*, and *mutation* operators that provide an effective combination of exploration of the global search space. The *Island multi-deme* GA is a widely used parallel GA model in which the population is divided into several sub-populations and distributed on different processors. Each sub-population evolves independently for a few generations, before one or more of the best individuals of the sub-populations migrate across processors.

Although it is effective for many other optimization problems, we observed that selection and mating slowed the evolution of beneficial fitnesses in the population. Therefore, in this work, we propose a modified GA without mating. The approach is similar to Tulpan's [9], except that we start with an empty library, and a separate GA population of next word candidate individuals with random base content. Each individual in the population is a DNA codeword encoded as a binary string with length $2n$, where n is the length of the codeword in bases. The four bases (A, T, C, G) are encoded as

(00, 01, 11, 10). Each DNA strand of length 16 can be represented as a 32 bit integer.

Given a codeword library S, the fitness of each individual d reflects how well the corresponding codeword fits into the current codeword library. Two values define fitness, *reject_num* and *max_match*. The *reject_num* is the number of codewords in the library which satisfies the condition that $LCS(s,d) > \sigma$ or $LCS(\bar{s},d) > \sigma$. The *max_match* can be calculated as

$$\max(|LCS(d,\bar{d})-\sigma|,|LCS(s,d)-\sigma|,|LCS(\bar{s},d)-\sigma|), \forall s \in S .$$

The codeword with lower fitness fits better in the library.

From equations (2)-(4) we know that a valid library word must have *reject_num* equal to 0. It is observed that adding a codeword with *reject_num* = 0 and *max_match* > 0 into the library will restrict the future growth of the library. Such codewords bind very weakly with other library words, but they are too far apart in the search space and interfere with closest packing. To maximize the library size, we want to select only those codewords that are "just good enough". To ensure this, we add another constraint to the optimization problem:

$$\max(LCS(s_1,s_2), LCS(s_1,\overline{s_2})) = \sigma, \forall s_1, s_2 \in S \quad (5)$$

Therefore, only codewords with *reject_num* = 0 (which also implies *max_match* = 0) will be added into the library.

A traditional GA mutation function might randomly pick an individual in the population, randomly pick a pair of bits in the individual representing one of its 16 bases, and randomly change the base to one of the 3 other bases in the set of 4 possible bases. In the proposed algorithm, however, we randomly select an individual, but then to exhaustively check all of the 48 possible base changes. This is an attempt to speed beneficial evolution of the population by minimizing the overhead that would be associated with randomly picking this individual again and again in order to test those mutations. We also specify that if none of the 48 mutations were beneficial, one of them is selected at random. This enables the individual to remain in the population and possibly experience subsequent (multiple) mutations. Figure 2 gives the pseudo code for the modified mutation function.

When an individual in the population achieves a fitness of 0, it is added to the set of good codewords, and the selected individual in the population is replaced by a new random individual. The GA is allowed to run until one of three termination criteria is satisfied: the number of codewords in the set is as large as desired; the algorithm has run for a specified maximum number of generations; or the algorithm has run for a specified maximum amount of time. We store the codeword values and the elapsed time at which they are each found, in memory during a run, and we store that data to a disk file at the end of a run. We also calculate and store the average time at which the *i*th words are found across multiple runs to assess average performance.

```
Mutation( )
    //M is the set of mutated individuals;
    //L is the set of library codewords;
    Randomly select an individual s from initial population;
    M = Φ;
    FOR i = 1 TO n
        B = {A, T, C, G} − {s[i]}; //B is the set of three nucleotides that
    is different from the ith nucleotide of s
        Generate three mutated individuals {s1, s2, s3} by replacing the
    ith nucleotide with one of the elements of B;
        M = M ∪ {s1, s2, s3};
    END
    Evaluate the fitness for each m ∈ M;
    IF  (∃m, fitness(m) = 0)  THEN L = L ∪ {m};
    ELSE    //evolve the population by replacing the original
    individual with a new individual with better fitness
        Select the individual x which has the lowest (best) fitness and
    x∈M;
        IF  fitness(x) < fitness(s)  THEN  replace s with x;
        ELSE replace s with a random individual from M;
    END
RETURN
```

Figure 2 Modified mutation algorithm.

IV. HARDWARE ACCELERATION OF LCS CALCULATION

The most time consuming part of the proposed GA algorithm is to calculate the fitness value for each individual. Performance profiling of our software GA version showed that >98% of the computing time was spent calculating the *LCS* distance between DNA strands. The LCS distance is calculated using dynamic programming. Figure 3 gives the pseudo code of the algorithm. The intermediate results are stored in an $n \times n$ matrix, where n is the length of the DNA codeword in bases. The calculation starts at the top left corner of the matrix and the final result is the value calculated in the cell located at the bottom right corner. For DNA codewords with length 16, at least 256 operations are needed before we can obtain the final result. Therefore, the throughput of the software based LCS calculation is less than $1/n^2$.

```
LCS(a, b)
    Initialize lcs[0][i] and lcs[i][0], 0≤i≤n-1
    FOR i = 0 TO n-1 BEGIN
        FOR j = 0 TO n-1 BEGIN
            IF (a[i] = b[j]) THEN k = 1;
            ELSE k = 0;
            lcs[j][i] = max(lcs[j-1][i], lcs[j][i-1], lcs[j-1][i-
    1]+k);
        END
    END
```

Figure 3 LCS distance calculation.

The algorithm can be implemented using a 2D systolic array. The systolic array is an $n \times n$ matrix. Figure 4 (a) gives

the structure of each cell in the matrix. Each cell consists of three registers: *A*, *B* and *ans*. For the cell at location (i, j), the registers *A* and *B* are used to store the *i*th nucleotide of one DNA codeword (north word) and the *j*th nucleotide of the other DNA codeword (west word) respectively. The register *ans* is used to store the intermediate result of the dynamic programming calculation. Each cell has five inputs. Two of the inputs connect to the register A and register B of the upper and left neighbor cells. The other three inputs connect to the *ans* registers of the upper, left and diagonal neighbor cells. In the present hardware version it takes two clock cycles for a cell to update its answer. In the first clock period, input registers *A* and *B* are updated, and in the second clock period, the cell output answer is calculated and the register *ans* is updated. In order to prevent ripple through operation, the cells in the even columns and even rows or odd columns and odd rows are synchronous to each other and operate as described above, but in the rest of the cells (which are also synchronous) the two operations are reversed, i.e. the *ans* output is calculated in the first clock period and the A and B inputs are updated in the second clock period.

The overall architecture of the 2D systolic array is shown in Figure 4 (b). The marked cells calculate their answers in the same clock cycle while the unmarked cells calculate their answers in the next clock cycle. In this way, the results propagate through the array diagonally. The final result is given by the *ans* register of the cell at the right bottom corner of the 2D array. It is easy to see that after a latency period that is required to fill the pipeline, the throughput of the systolic array is ½, i.e. 1 output result per 2 clock periods. When *n* increases, the throughput remains the same while the hardware cost increases, as long as the reconfigurable hardware chip has sufficient resources to implement a full $n{\times}n$ array of cells. Another detail is that the systolic array must be fed by an array of registers that delay the entry of the bases on the right of the North word and at the bottom of the West word. In effect, this synchronizes the presentation of those parts of the operand words with the diagonal waves of intermediate calculations in the cells that proceed from the upper left corner down and to the right through the array.
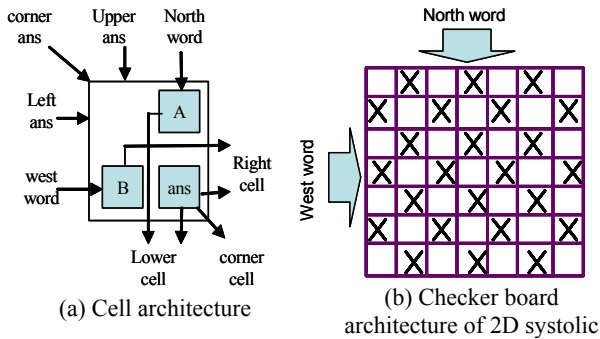

(a) Cell architecture (b) Checker board architecture of 2D systolic

Figure 4 2D systolic array for LCS calculation.

We note that version of this array for words of length 32 vs. 16 would use 4X the resources, but clock at the same rate.
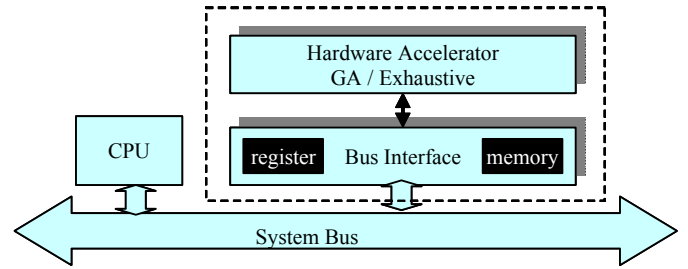
Figure 5 System architecture.

The proposed hybrid architecture consists of a host CPU, a hardware accelerator and a software program running on the host CPU. The host CPU and the hardware accelerator are connected via the system bus. Figure 5 shows the architecture of the system. In order to increase the portability of the design, we divide it into two modules: the bus interface and the hardware accelerator core. The bus interface module connects to the bus as a slave. It has a set of command registers and an information exchange memory, which can be accessed by both CPU and the hardware accelerator. For different bus architecture, a new bus interface must be developed.

A. Hardware acceleration for GA based codeword search

A two-level method is adopted to control the hardware accelerator. At the top level, the operations of the hardware accelerator are categorized into 7 states: {*idle*, *init*, *check_pop*, *mutation*, *check_mutate*, *update_pop*, *update_lib*}. In the *init* state, the hardware accelerator generates a random initial population, and sets up either an empty initial library, or reads an initial partial library from a disk file. In the *mutate* state, the hardware accelerator produces a population of 47 mutated individuals based on a chosen individual. The hardware accelerator calculates the fitness for all the individuals in the initial population, and in the mutated population, in the "*check_pop*" and "*check_mutate*" states, respectively. In the "*update_lib*" state, the hardware accelerator writes the newly discovered acceptable codewords into the library. In the "*update_pop*" state, the hardware accelerator writes the best (or a randomly chosen) mutated individual back to the working population.

Each state corresponds to an operation in the GA algorithm. Figure 6 (a) shows the control and data flow graph (CDFG) of the algorithm based on this state division. The "*update_lib*" and "*update_pop*" operations are one cycle operations because they only perform a memory write. All the other operations are multi-cycle operations, which again can be divided into several sub-states. When the top level state machine enters the corresponding state of a multi-cycle operation, the second level state machine is triggered.

We call an operation a *blocking operation* if its successors in the CDFG cannot start until this operation is done. Similarly, an operation is called *non-blocking operation* if its successors can start right after this operation started. The "*init*" and "*mutation*" operations are both non-blocking operations. While the hardware accelerator is generating the initial population and

the mutated population, it is at the same time checking the fitness of the generated individual. The "*check_pop*" and "*check_mutate*" operations are blocking operations. Their successors, i.e. "*mutate*" and "*update_pop*", cannot start until they have been finished. Figure 6 (b) shows the scheduling of the operations.
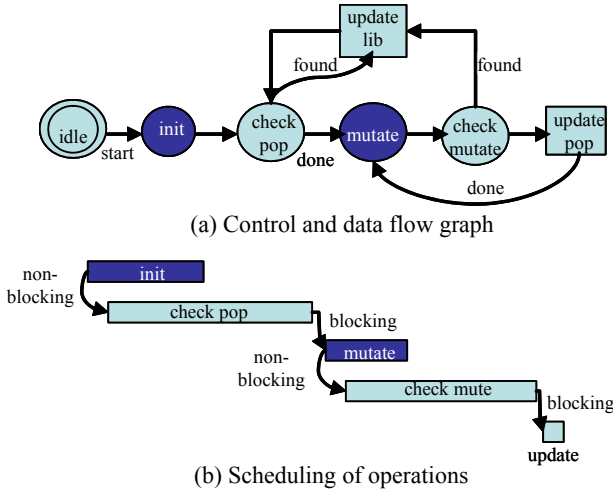

(a) Control and data flow graph

(b) Scheduling of operations

Figure 6 Top level state machine controller.

A buffer is needed to pass the results of one operation to its successor. In particular, a first-in-first-out (FIFO) storage should be used as the output buffer of a non-blocking operation. However, the implementation of the FIFO is relatively easy in this design because the non-blocking operations are always faster than their successors. Therefore, it is not necessary to check the FIFO underflow condition. We use dual port memory as the output buffer for the design. Three memory blocks are used: Initial Population Memory (M_{pop}), Mutated Population Memory (M_{mutate}) and CodeWord Library Memory (M_{lib}). The input and output buffer of different operations are given in Table 1.

Table 1. The input/output buffer of operations.

operations	Input	Output
init	-	M_{pop}
check_pop	M_{pop}	M_{lib}
mutate	M_{pop}	M_{mutate}
check_mutate	M_{mutate}	M_{lib}
update_lib	M_{pop}	M_{lib}
update_pop	M_{mutate}	M_{pop}

B. Hardware software interface

The hardware accelerator and the host CPU program run asynchronously. Four-way handshaking protocol is used to synchronize the communication between hardware and software, as shown in Figure 7. For example, when the hardware accelerator finds a new codeword, it raises the "PE_got_new_word" flag to the host program. After detecting this flag, the host program reads the new codeword then raises

the "host_got_new_word" flag. After detecting this flag, the hardware accelerator then clears the "PE_got_new_word" flag and acknowledges the host program by raising the "PE_got_message" flag.
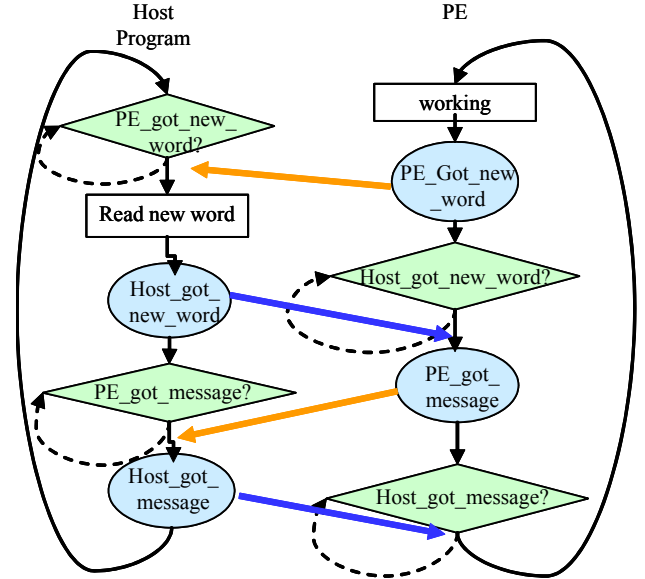

Figure 7 Hand-shaking between host and PE.

After detecting this flag, the host program then clears the "*host_got_new_word*" flag and acknowledges the hardware accelerator by raising the "*host_got_message*" flag, and continues. After detecting this flag, the hardware accelerator then clears the "*PE_got_message*" flag and continues. After the handshaking, the host program and the hardware accelerator work asynchronously until the host or hardware accelerator raises another message flag.
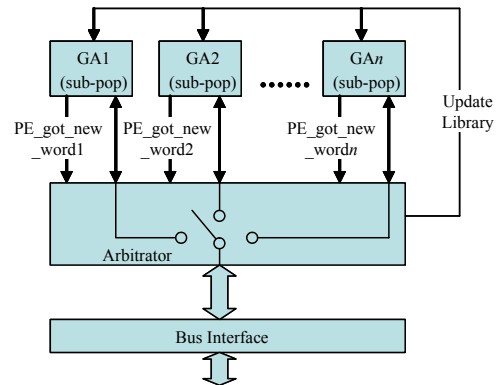
C. Parallel GA

Figure 8 Hardware architecture for parallel GA.

The hardware accelerator discussed above uses about 12,263 LUTs (look-up-tables), which is only about 42% of the programmable resources in a Xilinx Virtex II 3000 FPGA and about 16% of the programmable resources in a Xilinx XC2VP70 FPGA. Therefore, we evaluated a further speed-up

enhancement that involved implementing multiple parallel hardware accelerators on a single FPGA, as shown in Figure 8.

The system consists of *n* hardware accelerator modules, which are denoted as GA1~GA*n*, an arbitrator and a bus interface. The value of *n* is determined by the size of the FPGA. For example, *n* is 2 for the Virtex II 3000 FPGA and 5 for the XC2VP70. Each module implements the above mentioned genetic algorithm to search for the DNA codeword. They are independent to each other. The populations in different GA modules are initialized using different random seeds.

All the GA modules are connected to the bus interface through an arbiter. When a GA module finds a new codeword, it raises the "*PE_got_new_word*" flag and requests to be connected to the bus interface to communicate with the host. The arbiter broadcasts the new codeword to all other GA modules and raises the "*update_library*" flag. The GA module that receives the "*update_library*" request must terminate its current operation and go to "*update_lib*" state. If multiple GA modules raise the "*PE_got_new_word*" flag simultaneously, the arbiter must select one of them and invalidate the others. The decision is based on a fixed priority. The arbiter also connects the selected GA module that has found a new codeword with the bus interface to communicate with the host. If another GA module finds a new word, it must wait till the end of the current host-PE communication procedure to be connected to the bus interface. Figure 9 shows the state machine controller of the arbiter. The arbiter will be in the idle state after reset. When one of the GA modules raises the "*PE_got_new_word*" flag, the arbiter will go to the "*update_all_libraries*" state during which the arbiter raises the "*update_library*" flag. In the next clock period, it goes into the "*PE_communicating*" state during which the arbiter connects the GA module to the bus interface.

If the communication finishes before another GA module finds a new word, then the arbitrator goes back to the idle state. Otherwise, it first goes to the waiting state. After the communication is done, it goes to the "*update_all_libraries*" state and repeats the previous steps.
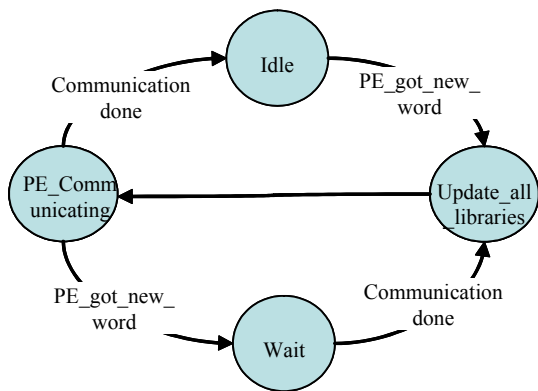

Figure 9 State machine controller of the arbitrator.

D. Hardware acceleration for exhaustive search

The effectiveness of the stochastic search starts decreasing when the search space increases and the solution space decreases. Therefore, as codewords are added to the library, the time required for the GA to find a new codeword increases exponentially. Furthermore, using stochastic search, we will never know whether still another new codeword can be added to the library. The only way to answer this question is by using exhaustive search, i.e. checking every possible codeword in the universe of all possible codewords. The complexity of exhaustive search increases linearly with the number of codewords already in the library. However, the complexity of exhaustive search also increases exponentially with the length of the codewords. As the name suggests, for a given initial library, the exhaustive search portion of the hybrid algorithm must scan the entire codeword space and find all remaining additional valid codewords that satisfy constraint equations (2)-(5). For DNA codewords of length 16, and for an initial library with 100 codewords, exhaustive search would take 52 days on a 2.0GHz Intel Xeon processor running a software fitness checker at 10 microseconds per check.

With small modification, we can implement the exhaustive DNA codeword search using hardware. The hardware accelerator for exhaustive codeword search consists of only one memory, which is used to store the codeword library, a 32 bit counter cycled from 0 to its maximum value to represent the potential new word, and two systolic array fitness checkers. For each codeword *x*, the calculation of $LCS(x, s)$ and $LCS(x, \bar{s})$, where $s \in S$, are performed simultaneously by the two fitness checkers. At 100Mhz clock frequency, the hardware accelerator takes about 1.5 hours to scan the entire ~4.3 billion codeword space for codewords of length 16, which is over 800 times faster than the workstation PC software only case. At the completion of exhaustive search we can say that a codeword set is *locally optimum*, in the sense that given the series of random numbers used to drive the stochastic GA in the early phase of building, no additional codewords can be added to increase the size of the library. To date, little data has been published in the literature on locally optimum edit distance codes of lengths greater than about 12 bases, and this hardware accelerator enables us to efficiently explore this aspect of the problem domain for the first time.

VI. EXPERIMENTAL RESULTS

A hardware accelerator that uses a stochastic GA to build DNA codeword libraries of codeword length 16 has been designed, implemented, and tested. The first version uses one fitness evaluator and is implemented on a single FPGA chip.

The design has actually been ported onto three different reconfigurable computing platforms, including a Xilinx XUP Virtex-II Pro evaluation board [13], a laptop computer with the Annapolis Wildcard FPGA board [14], and a desktop computer with the Annapolis Wildstar–II FPGA board. Different bus architectures are used to connect the hardware accelerator to the host CPU in each of the different platforms. The PLB bus is used in the Xilinx Virtex-II Pro evaluation board, while the PCMCIA card bus and PCI-X bus are used in the system with

WildStar and WildCard, respectively. The other difference among these platforms is the amount of resources available on the FPGA chips resident on the boards.

Table 2 shows the size of the reconfigurable logic and the on-chip memory for the three different computing platforms. The design is synthesized using Synplify from Synplicity. It uses 12,263 LUTs (look-up-tables), which is about 42% of the programmable resources in a Xilinx Virtex II 3000 FPGA. The hardware accelerator for exhaustive search of DNA codeword length 16 uses 21,733 LUTs, which is about 75% of Virtex II 3000 FPGA.

Table 2 Available reconfigurable logic and on-chip memory resources of different platforms.

Computing platform	FPGA	Logic Cells	BRAMs (kb)	PPCs
XUP eval. board	XC2VP30	30,816	2,448	2
WildCard-II	Xilinx Virtex II 3000	28,672	1,728	0
WildStar Pro	XC2VP70	74,448	5,904	2

Figure 10 shows a comparison of the average performance of the GA based codeword search algorithm running in software on a single workstation processor (upper curve) and the hardware accelerated hybrid architecture (lower line). The performance is measured in terms of the time it takes to build a large library. Less time is better, so the lower curve is better than the upper curve. In this plot the x axis is codewords found, where each codeword consists of a strand and its reverse complement. The GA is a stochastic algorithm, so each point in the curves is the average over multiple runs of the times taken to find the # of codewords on the x axis. For these experiments we set n and σ to be 16 and 10 respectively. The upper curve for the software version was run on one workstation with 1 P4 processor. The lower curve for the hardware GA was run with a 100MHz FPGA clock frequency.
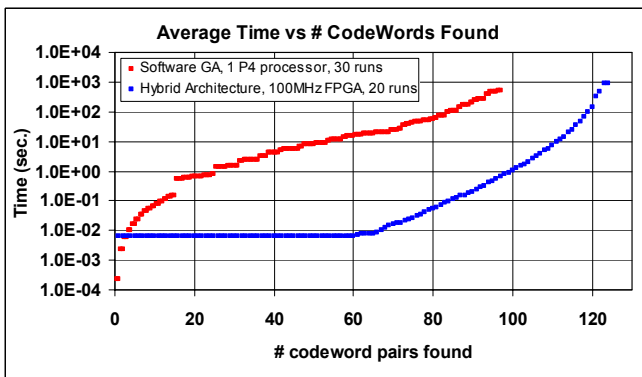

Figure 10 Comparison of average performance.

Compared to the software only implementation, the hardware accelerator running at 100MHz provides approximately a 1000X speed-up. The speed-up of the hardware versions is due to the parallel and pipelined architecture of the hardware. If we were able to increase either the number of fitness calculating arrays a we would expect almost linear speed-up ($a/0.98$). Also, based on previous work [15] that used a distributed Island Model GA run on a cluster of workstations, we would expect linear speed-up as the number of distributed GA populations p is increased.

Figure 11 shows a comparison of the best performance among software GA, and two versions of the hardware GA.
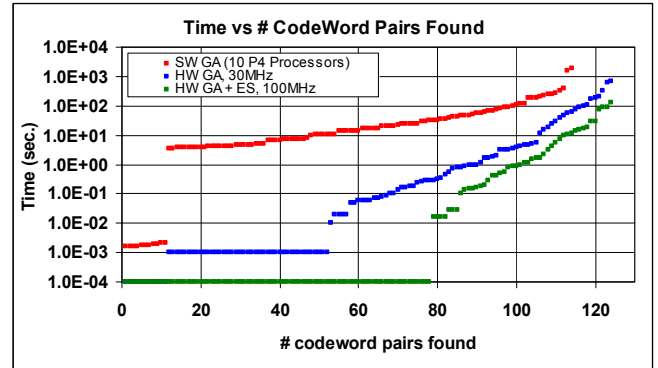

Figure 11 Comparison of best performance.

The top red curve for the distributed software multi-deme GA was run on a cluster using 10 P4 processors. The inter-processor communication is implemented using MPI (message passing interface). The middle blue curve for the hardware GA was run on the Annapolis Wildcard-II in a P3 notebook PC with a 30MHz FPGA clock frequency. The lower magenta curve for the hardware GA with exhaustive search was run on a Wildcard board in a P4 workstation with a 100MHz FPGA clock frequency. The later run was set up to run the GA until 240 words were found, and then switch to exhaustive search, after which 8 more words were found.

We also used the exhaustive search version of the hardware accelerator to investigate the average size of locally optimum codeword libraries that can be built, and the efficacy of the GA for building them. Figure 12 shows the distribution of the size of local optimal DNA codeword libraries that were generated by running hardware GA for 300 seconds followed by hardware exhaustive search. The results show that the size of the local optimal DNA codeword library follows a normal distribution with mean of about 122 codewords (word/word' pairs). The experiment consists of 60 tests, which took about 90 hours. The equivalent test on a 30 workstation cluster would have taken about 3000 hours (4 months).
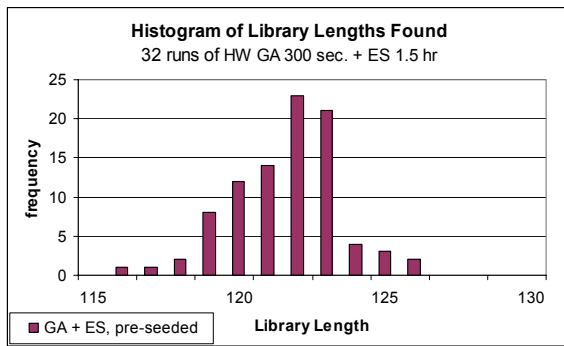
Figure 12

Figure 12 Size of local optimal DNA codeword libraries built with 300sec. GA plus exhaustive search.

Figure 13 shows data from a second experiment involving 32 runs of GA for 600 sec. followed by exhaustive search, in terms of the size of the library built during the GA phase (red) and the number of words added by exhaustive search (green).
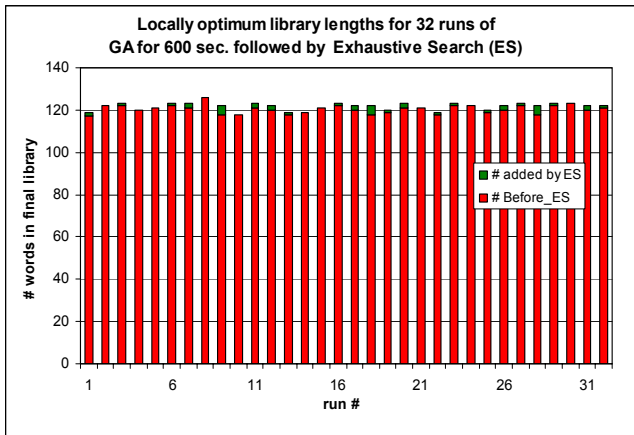

Figure 13 Sizes of Libraries built with 600 sec. GA followed by exhaustive search.

Figure 14 shows a histogram of the # of words added by exhaustive search for these runs. On average, the GA alone finds 120.4 words vs. 121.7 with GA + exhaustive search, or about 98.9% of the words that can be found.
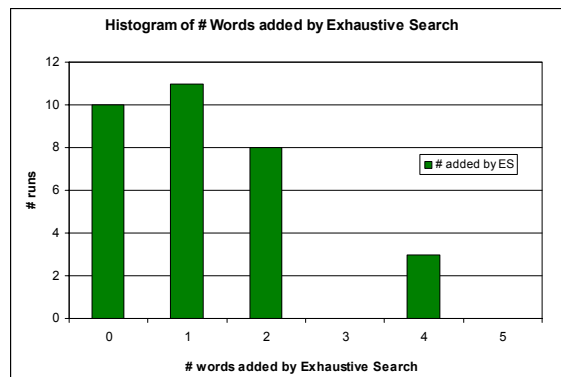

Figure 14 Histogram of # words added by Exhaustive Search for the runs of Figure 13.

VII. CONCLUSIONS AND FUTURE WORK

In this work, we propose a novel architecture for accelerating a GA based DNA codeword searching algorithm. Our preliminary results show that, using a new hybrid hardware/software implementation, we can speedup the DNA codeword search procedure by more than 1000X. We have also described a hardware exhaustive search extension that can produce known locally optimum codes. In the future, we plan to extend the current architecture to implement a multi-deme GA on a single FPGA, a more general GA, more accurate techniques to measure the binding strength of DNA pairs, and a checker for codes word of at least length 32.

REFERENCES

[1]   L. M. Adleman, "Molecular Computation of Solutions to Combinatorial Problems," *Science*, vol. 266, pp. 1021-1024, November 1994.

[2]   A. Brenneman and A. Condon, "Strand Design for Biomolecular Computation", *Theoretical Computer Science*, vol. 287, pp.39-58, 2002.

[3]   S.-Y. Shin, I.-H. Lee, D. Kim, and B.-T. Zhang, Multiobjective Evolutionary Optimization of DNA Sequences for Reliable DNA Computing", *IEEE Transactions on Evolutionary Computation*, vol. 9(20), pp.143-158, 2005.

[4]   F. Tanaka, A. Kameda, M. Yamamoto, and A. Ohuchi, Design of Nucleic Acid Sequences for DNA Computing based on a Thermodynamic Approach, Nucleic Acids Research, 33(3), pp.903-911, 2005.

[5]   J. Santalucia, " A Unified View of polymer, dumbbell, and oligonucleotide DNA nearest neighbor thermodynamics", *Proc. Natl. Acad. Sci., Biochemistry*, pp. 1460-1465, February 1998.

[6]   A. D'yachkov, P.L. Erdös, A. Macula, V. Rykov, D. Torney, C-S. Tung, P. Vilenkin and S. White, "Exordium for DNA Codes," *Journal of Combinatorial Optimization*, vol. 7, no. 4, pp. 369-379, 2003.

[7]   R. Deaton, M. Garzon, R.C. Murphy, J.A. Rose, D.R. Franceschetti, and S.E. Jr. Stevens, "Genetic search of reliable encodings for DNA-based computation," *Proceedings of the First Annual Conference on Genetic Programming*, pp. 9-15, July 1996.

[8]   Bishop, M. , Macula, A. , Pogozelski, W. , and Rykov, V. , "DNA Codeword Library Design", *Proc. Foundations of Nanoscience – Self Assembled Architectures and Devices, (FNANO)*, April 2005.

[9]   Tulpan, D.C. , Hoos, H. , Condon, A. ,"Stochastic Local Search Algorithms for DNA Word Design", *Eighth International Meeting on DNA Based Computers(DNA8)*, June 2002.

[10] S. Houghten, D. Ashlock and J. Lennarz, "Bounds on Optimal Edit Metric Codes", *Brock University Technical Report # CS-05-07*, July 2005.

[11] O. Milenkovic and N. Kashyap, "On the Design of Codes for DNA Computing," *Lecture Notes in Computer Science*, pp. 100-119, Springer Verlag, Berlin-Heidelberg, 2006.

[12] R. Brualdi, and V. Pless, "Greedy Codes," *Journal of Combinatorial Theory Series A*, vol. 64, pp. 10-30, 1993.

[13] http://www.xilinx.com/

[14] http://www.annapmicro.com/

[15] D. Burns, K. May, T. Renz, and V. Ross, "Spiraling in on Speed-Ups of Genetic Algorithm Solvers for Coupled Non-Linear ODE System Parameterization and DNA Code Word Library Synthesis," *MAPLD International Conference*, 2005.