HARDWARE ACCELERATION OF MULTI-DEME GENETIC ALGORITHM FOR DNA CODEWORD SEARCHING

Qinru Qiu[1], Daniel Burns[2], Prakash Mukre[1], Qing Wu[1]

[1]Department of Electrical and Computer Engineering,

Binghamton University, Binghamton, NY, USA

Email: {qqiu, pmukre, qwu}@binghamton.edu,

[2]Air Force Research Laboratory, Rome Site,

26 Electronic Parkway, Rome, NY, USA

Email: Daniel.Burns@rl.af.mil

Abstract

A large and reliable DNA codeword library is key to the success of DNA based computing. Searching for sets of reliable DNA codewords is an NP-hard problem, which can take days on 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 multi-deme genetic algorithm (GA) for the application of DNA codeword searching. The presented architecture provides more than 1000X speed-up compared to a software only implementation. A code

extender that uses exhaustive search to produce locally optimum codes in about 1.5 hours for the case of length 16 codes is also described. The experimental results demonstrate that the GA can find ~99% of the words in locally optimum libraries. Finally, we investigate the performance impact of migration, mating and mutation functions in the hardware accelerator. The analysis shows that a modified GA without mating is the most effective for DNA codeword searching.

Keywords

DNA, Genetic Algorithm, Hardware Acceleration

1. Introduction

The DNA molecule is now used in many areas far beyond its traditional function. The first DNA-based computation was proposed and implemented 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 stable double-stranded duplexes. 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, as a first step in this work we chose a simpler metric, the *Levenshtein distance*, or the so-called *deletion-correcting* or *edit distance*, which has also 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 base 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 multi-deme genetic algorithm. We propose a FPGA (Field Programmable Gate Array) based hardware accelerator design which performs multi-deme parallel GA on a single chip. The hardware accelerator and the host PC communicate via the system bus, and an appropriate software interface controls communication between them. The proposed architecture provides more than 1000X speed-up compared to a software only implementation. A hardware based code extender that uses exhaustive search to produce locally optimum codes is also described. The code extender does a final scan across the entire universe of possible codewords and completes the codeword library generated from GA by adding any additional words that satisfy the specified constraints.

The remainder of this paper is organized as follows: Section 2 provides the necessary biological background and terminology. Section 3 introduces the problem definition and the genetic algorithm for DNA codeword search. Section 4 gives the detailed information about how to accelerate the GA fitness calculation. Sections 5 and 6 provide details about the hybrid architecture and some performance analysis of the design. Performance comparison between the hardware and the software version of the GA, and early results on locally optimum codes are also presented in Section 6. Final conclusions are given in Section 7.

2. 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 it is denoted as 3' and the other as 5'.

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 and the structure they form is called *Watson-Crick* (*WC*) *duplex*. Figure 1 (a) shows an example of a WC duplex. We refer to the non-WC duplex as *crosshybridized* (*CH*) *duplex*. Figure 1 (b) shows an example of a CH duplex. Only WC duplexes are needed during DNA computing. Therefore, it is important to design the DNA codes such that a fixed temperature can be found that is well above the melting point of all CH duplexes and well below the melting point of all WC duplexes that can form from strands in the code.

Predicting crosshybridization involves many considerations. In this paper, we only consider the first order effect, and use the maximum number of possible Watson-Crick pairs between two 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 (b) shows two DNA strands that bind with 5 Watson-Crick pairs. The length of the longest complementary sequence between two flexible DNA strands, A and B, is the same as the *length of the longest common sequence* (*LLCS*) between A and $\overline{B}$ [6], where $\overline{B}$ is the Watson-Crick complement of B.

(a) WC duplex (b) CH duplex

Figure 1 Binding between DNA strands.

3. 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 length of the longest common sequence between DNA strands A and B is denoted as $LLCS(A, B)$. 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. $\{(s_1, \overline{s_1}), (s_2, \overline{s_2}), \ldots\}$, 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:

$$\max |S| \quad \text{such that} \qquad (1)$$

$$LLCS(s_1, \overline{s_1}) \le \sigma, \quad \forall s_1 \in S, \qquad (2)$$

$$LLCS(s_1, s_2) \le \sigma, \forall s_1, s_2 \in S \quad (3)$$

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

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. In this work, the single point cross-over mating operator is used.

Each individual in the population is a DNA codeword encoded as a binary string with length 2*n*, 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 the fitness, *reject_num* and *max_match*. The *reject_num* is the number of codeword*s* in the library which satisfies the condition that $LLCS(s, d) > \sigma$ or $LLCS(\bar{s}, d) > \sigma$. The *max_match* can be calculated as

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

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

Therefore, only codewords with *reject_num* = 0 (which 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, the elapsed time at which they are each found in memory during a run, and 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.

4. Hardware Acceleration of LLCS 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 *LLCS* between strands.

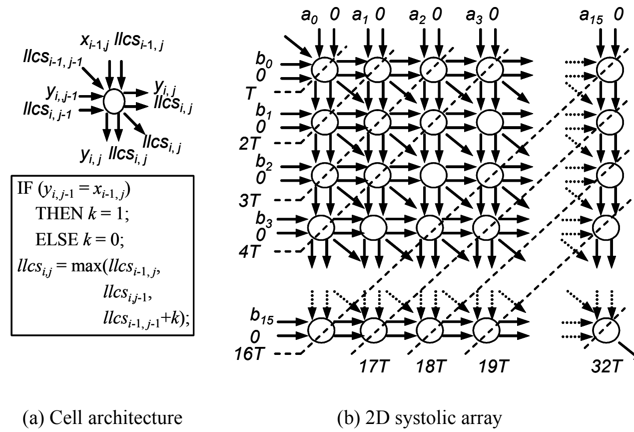
The *LLCS* is calculated using dynamic programming. Figure 3 gives the pseudo code of the algorithm. The intermediate results are stored in an $n \times n$ matrix, where n is the length of the DNA codeword in bases. The calculation starts at the top left corner of the matrix and the final result is the 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 *LLCS* calculation is less than $1/n^2$.

```
LLCS(a, b)
  Initialize llcs[0][i] and llcs[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;
      llcs[j][i] = max(llcs[j-1][i], llcs[j][i-1], llcs[j-1][i-1]+k);
    END
  END
END
```

Figure 3 LLCS distance calculation.

Many systolic algorithms for the LCS problem have been proposed [16][16]. The LLCS calculation is a much simpler problem. In this work, we implemented a 2D systolic array for the acceleration of LLCS calculation. The systolic array is an $n \times n$ array of identical cells. Figure 4 (a) gives the structure of each cell, including its input/output and the computation implemented. The computation is performed every other clock period. The overall architecture of the 2D systolic array as well as the data dependency and timing information are shown in Figure 4 (b). 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 perform the computation in the same clock period. The rest of the cells are also synchronous to each other but perform the computation in the next clock period. In this way, the results propagate through the array diagonally. 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.

It is interesting to note that as n increases, the hardware resource cost increases, but the throughput remains the same, as long as the reconfigurable hardware chip has sufficient resources to implement a full $n \times n$ array of cells. A version of this chip for words of length 32 is feasible. 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 word a and at the bottom of the word b. 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) 2D systolic array

Figure 4 2D systolic array for LLCS calculation.

5. Hybrid Architecture

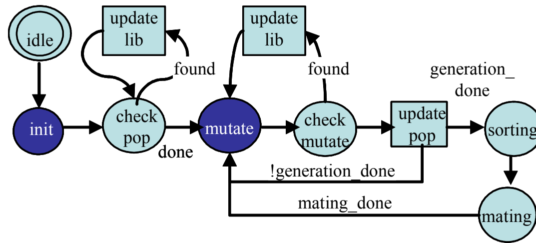
The proposed hybrid architecture consists of a host CPU, a hardware accelerator and a software program running on the host CPU. The host CPU and the hardware accelerator are connected via the system bus. In order to increase the portability of the design, we divide it into two modules: the bus interface and the hardware accelerator core. The hardware accelerator will also be called as processing element (PE) in the rest of the paper. 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 PE. For different bus architecture, a new bus interface must be developed.

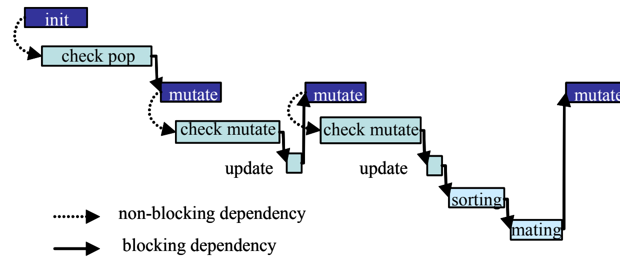
5.1 Hardware Acceleration for Multi-deme Parallel GA Based Codeword Search

A two-level method is adopted to control the PE. At the top level, the operations of the PE are categorized into 9 states: {*idle*, *init*, *check_pop*, *mutation*, *check_mutate*, *update_pop*, *update_lib*, *sorting*, *mating*}. In the *init* state, the PE 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 PE produces a population of 47 mutated individuals based on a chosen individual. The PE 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 PE writes the newly discovered acceptable codewords into the library. In the "*update_pop*" state, the PE writes the best (or a randomly chosen) mutated individual back to the working population. In the "*sorting*" state, the PE scans the entire population to pick the best *k* individuals. Two parents are randomly picked from these individuals when the PE is in the "*mating*" state and single-point cross-over is performed. A control flag is introduced which can be used to disable the sorting and mating functions in the PE.

Each state corresponds to an operation in the GA algorithm. Figure 5 (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 sub-states. When the top level state machine enters the state of a multi-cycle operation, the second level state machine is triggered.



(a) Control and data flow graph



(b) Scheduling of operations

Figure 5 Top level state machine controller.

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 PE 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*", "*sorting*", and "*mating*" operations are blocking operations. Their following operations cannot start until they have been finished. Figure 5 (b) shows the scheduling of the operations.

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. The output buffers are implemented using the FPGA built-in block memories. The block memories are dual port memories which can be read and written simultaneously. 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 buffers 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_{mute}
check_mutate	M_{mute}	M_{lib}
update_lib	M_{pop}	M_{lib}
update_pop	M_{mute}	M_{pop}
sorting	M_{pop}	M_{pop}
mating	M_{pop}	M_{pop}

The PE and the host CPU program run asynchronously. Four-way handshaking protocol is used to synchronize the communication between hardware and software.

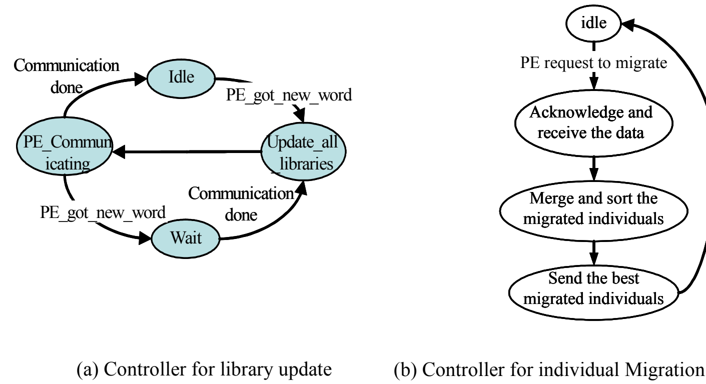
5.2 Parallel Multi-deme GA

The PE 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 PEs on a single FPGA. The architecture supports the exchange of best

individuals among PEs. Therefore, the overall system performs parallel multi-deme GA.

The system consists of *n* PE modules, which are denoted as GA1~GA*n*, an arbiter 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.

Communication and synchronization are two challenges that need to be addressed when designing a system that performs parallel GA. All the GA modules share the same bus interface. Codewords found by any one GA module must be harvested and passed to the other GA modules. In this design, all the GA modules are connected to 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 6 (a) shows the state machine controller of the arbiter for library update.



(a) Controller for library update (b) Controller for individual Migration

Figure 6 State machine controller of the arbitrator.

In the multi-deme island GA, the best few individuals of each sub-population migrate periodically according to an interconnect configuration, e.g. around a ring in one direction. This procedure is also controlled by the arbiter. A separate state machine controller in the arbiter is developed for the migration procedure. Figure 6 (b) shows the state diagram of the migration controller. Periodically, the PE sends a migration request to the arbiter. The arbiter will acknowledge this request if its migration controller is in the idle state. After receiving the acknowledgement from the arbiter, the PE sends its best few individuals and their fitness values to the arbiter. These data are placed in a memory together with similar data received from other PEs. The arbiter sorts and picks the best *m* individuals, where *m* is the number of individuals to be migrated, and sends them back to the PE which started the request for migration. For the case of 2 PEs on a chip served by one arbiter, this is equivalent to a directed ring configuration. However, for the case of more than 2 PEs on a chip, this approach implements a local pooling, or all-to-all configuration. Above the chip level, the host is still free to implement any communication configuration among host nodes in a cluster with standard MPI.

5.3 Hardware Acceleration for Exhaustive Search

The effectiveness of the stochastic search decreases when the size of the search space increases, or when the solution space decreases due to additional constraints.

As codewords are added to the library, more library words must be checked against candidates, and the new words act as new constraints. As a result, 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)-(4). For DNA codewords of length 16, and for an initial library with 100 codewords, exhaustive search in software would take 52 days on a 2.0GHz Intel Xeon processor if checking a pair takes 10 microseconds.

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 $LLCS(x,s)$ and $LLCS(x,\bar{s})$, where $s \in S$, are performed simultaneously by the two fitness checkers.

The hardware accelerator for exhaustive search of DNA codewords of length 16 uses 21,733 LUTs, which is about 75% of Virtex II 3000 FPGA. 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.

6. 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.

Table 2 Comparison of different platforms

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

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 different computing platforms.

The first set of experiments evaluates the performance impact of various parameters of the hardware multi-deme GA, including the size of the sub-population, the percentage of mutation during each generation, the length of epoch between

migrations and the number of best individuals that migrate. For this first set of experiments, the hardware implementations consisted of 2 parallel PEs that perform GA based codeword searching, each with one LLCS checker, and without exhaustive search.

We first ran the DNA codeword searching varying sub-population from 16 to 256. The number of keepers, the length of the epoch, number of migrated individuals, and the percentage mutation were fixed to be 8, 5, 7 and 10. Figure 7 shows a comparison of the average performance of those runs, 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 all the plots given by Figure 7-13, the x axis is the number of codewords found, where each codeword is either a strand or its reverse complement (a pair counts for 2). The GA is a stochastic algorithm, so each point in the curves is the average over 10 runs of the times taken to find the # of codewords on the x axis. For these experiments we set the length of the codewords n to be 16, and the permissible match (n- edit distance) σ to be 10. The experimental results show that with mating and migration enabled, a small population is superior to a large population in terms of search speed. This is because the most time consuming operation in mating and migration is pick up the best k individuals, which we call the number keepers. This does not require a full sort of the population, but even so, it is a sequential procedure that cannot be accelerated by a parallel architecture for typical population sizes. It takes more time to index through a larger population multiple times to find its best k individuals.

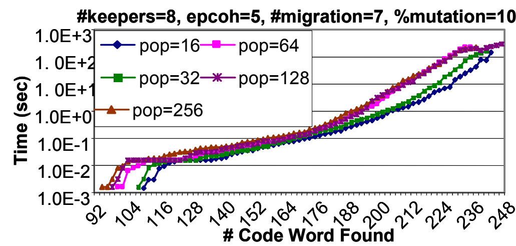


Figure 7 Effect of size of sub-population

In the second experiment, we vary the percentage mutation from 1 to 25 to evaluate its impact on performance. The size of sub-population, the number of migrated individuals, the length of epoch and the number of keepers were fixed to be 64, 7, 5 and 8. Figure 8 shows a comparison of the average performance of different configurations. As we can see, the percentage mutation has a significant impact on the system performance.

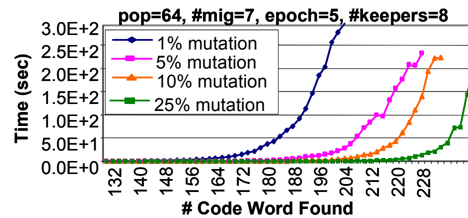


Figure 8 Effect of mutation

Higher percentage mutation leads to better performance. For example, to find 206 codewords, the hardware GA configured with 25% mutation is about 400X faster than the hardware GA configured with 1% mutation. This can partly be explained by the overhead of mating. When the size of population is fixed, the value of percentage mutation determines how many mutation operations will be performed between two mating operations. Because each mutation operation takes fixed amount of time, it also determines the frequency of mating operations. A higher percentage mutation implies less frequent mating, and thus, lower overhead from the sorting operation.

In the third and the fourth experiments, we vary the number of migrated individuals and the number of generations in the epoch between migrations, respectively, to evaluate the performance impact of these two parameters. However, the results show that there is little performance impact from the number of migrated individuals and the epoch length. Due to the space limit, we do not report this data in the paper.

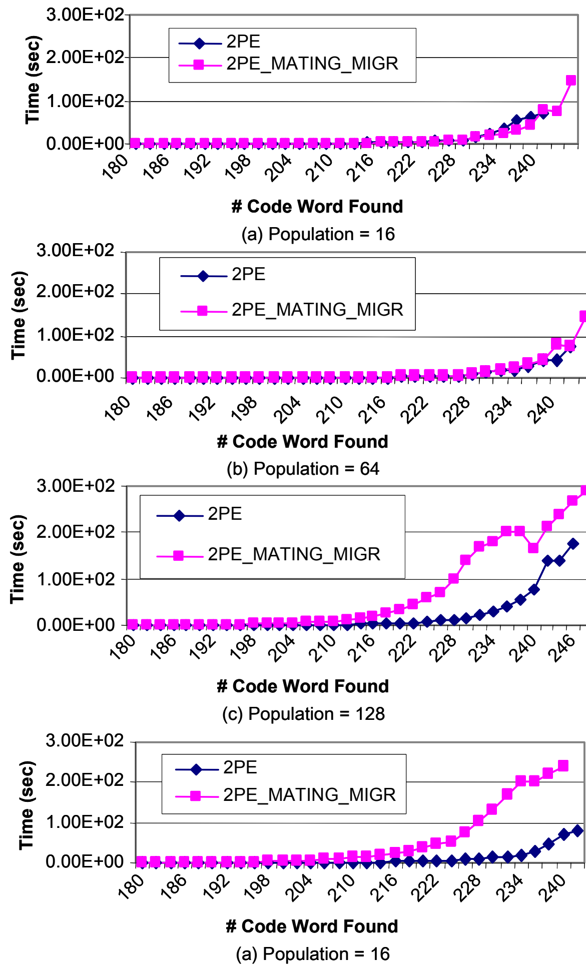


Figure 9 Effect of mating and migration.

The second set of experiments compares the performance of multi-deme hardware GA with and without mating and migration. Figure 9 shows a comparison of the average performance of GA with mating and migration versus GA without mating and migration when the size of sub-population varies from 16 to 256. The number of keepers, the length of the epoch, the size of migrated individuals and the percentage mutation are fixed to be 8, 5, 7 and 10. As we can see, overall, the parallel GA without mating and migration is more efficient than the parallel GA

with mating and migration. The difference becomes more significant as the size of population increases. Again, this shows that the overhead of mating increases as the population size increases.

Figure 10 analyzes the data from this experiment in terms of the speed improvement of parallel GA without mating and migration, for different population sizes, normalized to the performance with population size 16. As we can see, at the beginning of the search, smaller populations find words faster, but as the number of codewords increases, the larger populations find word slightly faster. This effect may be due to the beneficial effect of processing more mutations in between pick-up operations at the end of generations (doing wider search) outweighs the negative effect of the overhead of the pickup operation that also increases with population size.

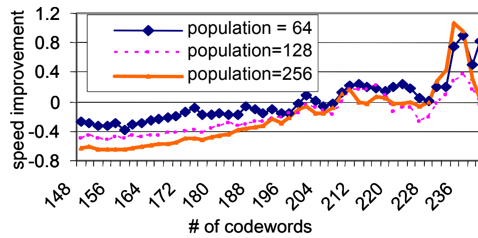


Figure 10 Effect of size of population in GA w.o. mating and migration.Figure 11 shows the performance comparison between a single PE system and a 2 PE system. Both systems are configured with population size equal to 16 and both are running without mating and migration. As expected, the 2-PE system is about twice as fast as the one PE system

The next set of experiments compares the hardware GA with a software version of the GA, again without mating and migration, and with one PE is instantiated in the hardware.

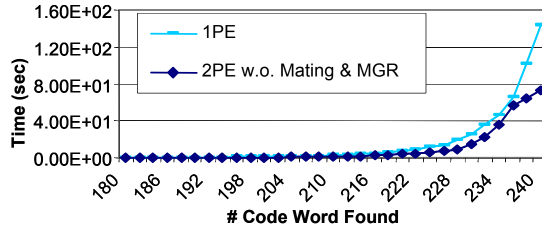


Figure 11 Performance comparison of single PE vs. 2-PE

Figure 12 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 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.

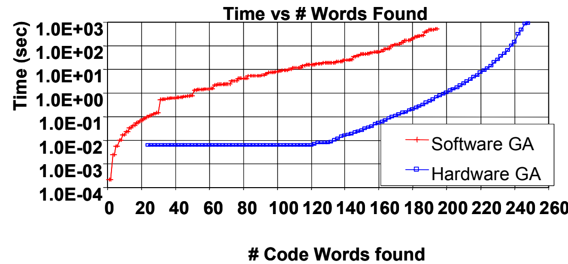


Figure 12 Comparison of average performance.

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. Based on previous work [15] we would expect almost linear speed-up (a/0.98) vs. the number of fitness calculators, and linear speed-up as the number of distributed GA populations p is increased.

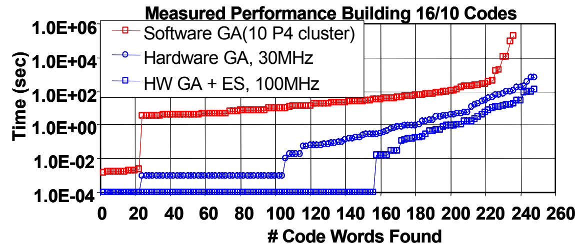


Figure 13 Comparison of best performance.

Figure 13 shows a comparison of the best performance to date of the software GA and the hardware GA. In this case, the top red curve for the distributed software multi-deme GA was run on a cluster using 10 P4 processors without mating, but with migration. 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 notebook PC with a 30MHz FPGA clock frequency, without mating and migration. 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, also without mating and migration (exhaustive search found 8 more words).

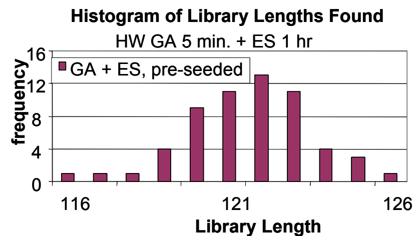


Figure 14 Size of local optimal DNA codeword libraries built with 300sec. GA plus exhaustive search.

In a final set of experiments, we used the exhaustive search version of the hardware accelerator to determine the average size of locally optimum codeword libraries that can be built, and the efficacy of the GA for building them. Figure 14 shows a histogram of the sizes of libraries generated by running hardware GA (without

mating and migration) for 300 seconds followed by hardware exhaustive search. The results show that the size of the local optimal DNA codeword library follows approximately 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 15 shows data from a second experiment involving 32 runs of the same hardware GA for 600 sec. followed by exhaustive search. The number of words found during the GA phase (red) and the exhaustive search phase (green) is highlighted.

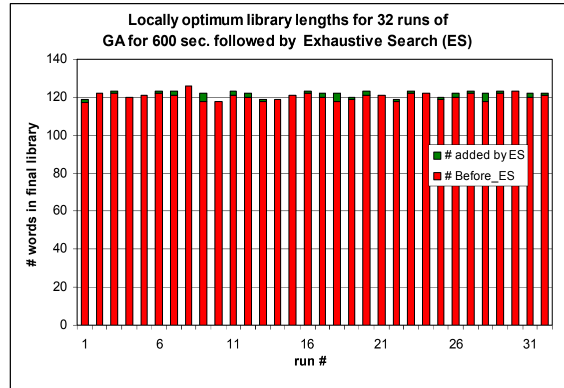


Figure 15 Sizes of Libraries built with 600 sec. GA followed by exhaustive search.

The GA phase alone finds an average of 120.4 words, and exhaustive search raises the number found to vs. 121.7. So, GA alone found about 98.9% of the words that can be found.

7. Conclusions and Future Work

In this work, we propose a novel architecture for accelerating a multi-deme parallel GA based DNA codeword searching algorithm. Our preliminary research results show that, using a new hardware and software hybrid 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 incorporate thermodynamics based metrics for estimating 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 Tech. Rer.t # 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.

[16] P.D. Michailidis and K.G. Margaritis, "New Processor Array Architectures for the Longest Common Subsequence Problem," *The Journal of Supercomputing*, vol. 32, pp. 51-69, 2005.

[17] Y.C. Lin and J.W. Yeh, "A Scalabl and Efficient Systolic Algorithm for the Longest Common subsequence Problem," *Journal of Information Science and Engineering*, vol. 18, pp. 519-532, 2002.