FPGA Acceleration of Protein Back-Translation and Alignment

Sahand Salamat, Jaeyoung Kang, Yeseong Kim, Mohsen Imani*, Niema Moshiri, Tajana Rosing
Department of Computer Science and Engineering, UC San Diego, CA 92093, USA
*Department of Computer Science, University of California, Irvine, CA 92697, USA
{sasalama, j5kang, yek048, a1moshir, tajana}@ucsd.edu; *m.imani@uci.edu

***Abstract*—Identifying genome functionality changes our understanding of humans and helps us in disease diagnosis; as well as drug, bio-material, and genetic engineering of plants and animals. Comparing the structure of the protein sequences, when only sequence information is available, against a database with known functionality helps us to identify and recognize the functionality of the unknown sequence. The process of predicting the possible RNA sequence that a specific protein has originated from is called back-translation. Aligning the back-translated RNA sequence against the database locates the most similar sequences, which is used to predict the functionality of the unknown protein sequence. Providing massive parallelism, FPGAs can accelerate bioinformatics applications substantially. In this paper, we propose, `FabP`[1], an optimized FPGA-based accelerator for aligning a back-translated protein sequence against a database of DNA/RNA sequences. `FabP` is deeply optimized to fully utilize the FPGA resources and the DRAM memory bandwidth to maximize the performance. `FabP` on a mid-range FPGA provides 8.1% and 23.3× (24.8× and 266.8×) speedup and higher energy efficiency as compared to the GPU-based implementation on a high-end NVIDIA GPU (state-of-the-art CPU implementation), respectively.**

I. Introduction

The Central Dogma of Biology describes the process of information flow in cells: the DNA, a double-stranded molecule made up of two chains of A, C, G, T nucleotides, contains genes, which are *transcribed* to messenger RNA (mRNA), which are then *translated* to protein, which is generally considered the functional unit of the cell [1]. Proteins are made up of one or more chains of 20 common amino acids. An unknown protein can be characterized when its sequence shares significant similarity with a protein with known characteristics. Pairwise searches can find the most similar sequences in a database to the unknown query sequence [2], [3], and the task of searching for a known protein sequence against a database of DNA or RNA sequences is often desired by biologists. Such searches have enabled crucial discoveries in biomedical research, such as improving our understanding of the mechanism of human cancers [4], [5], discovering novel potential cancer therapeutics [6], and identifying and screening potent antimicrobial peptides [7].

Protein sequencing is the biochemical process of determining the amino acid sequence of a given protein. Figure 1 describes the overview of the computational task flow. From a computational standpoint, a protein sequence can be considered as a string over a set that includes different amino acid letters, e.g., $\mathbf{S} = \langle \texttt{Met}, \texttt{Phe}, \cdots \rangle$. The protein sequence is originally translated from an mRNA which can be considered a string over the four alphabets, A,C,G,U}. Each non-overlapping three-letter window of the mRNA, known as a Codon, encodes a

[1]FabP is also the name of a family of proteins, "Fatty-Acid-Binding Proteins".

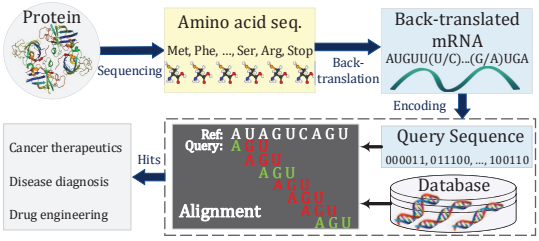


Fig. 1. Protein back-translation and alignment flow

specific letter in the amino acid alphabet according to the Codon table (Figure 2). The protein sequence is the result of replacing each codon of the mRNA sequence from start to end with its corresponding amino acid. The task of "back-translation" uses the Codon table to generate an mRNA sequence representing the most likely non-degenerate coding sequence. Since the back-translation in most cases does not yield a unique result, a consensus sequence derived from all possible Codons is needed. The mentioned protein sequence **S** can be back-translated to AUGUU$\underline{\text{U}}\cdots$, and AUGUU$\underline{\text{C}}\cdots$. The encoding step converts the consensus sequence to the binary representation while preserving all the isomorphic back-translated sequences of amino acids. Sequence alignment finds the regions with high similarity to the query sequence. The regions with high similarity, called hits, are used in many applications to predict the functionality of the unknown query sequence.

Several algorithms exist for aligning protein sequences. Dynamic Programming based (DP-based) algorithms consider all the possible sequence mutations [8], [9]. There are two general types of DNA mutations: *substitutions*, in which the letter at a single position of a sequence is replaced with another letter, or *indels* (short for "insertions and deletions"), in which one or more letters are either inserted into or deleted from the sequence, typically in contiguous blocks [10]. The other works rely on heuristic approaches to reduce the computation complexity of dp-based algorithms. The most widely-used heuristic solution is Basic Local Alignment Search Tool (BLAST), a sequence similarity search program developed by the National Center for Biotechnology Information (NCBI) to accelerate the alignment process [11]. It supports varieties of searches by type, and TBLASTN finds matches between protein sequences and a database of DNA/RNA sequences [12].

The size of the genomics data is doubling every 7 months [13], which is considerably higher than Moore's Law, which expects to double the computation power every 2 years. As the amount of the generated data increases, general-purpose processors have become incapable of processing the genomics data. Thereby, FPGAs have been used to accelerate sequence alignment

978-3-9819263-5-4/DATE21/©2021 EDAA

algorithms including both pairwise and heuristic algorithms [3]. Recent popularity of FPGAs as accelerators has led to widely deployment of FPGAs in data centers to provide easier access for users to implement their accelerators on cloud FPGAs [14], [15], [16]. The existing pairwise alignment algorithms require significantly high computational power and resources since the algorithm complexity grows quadratically with the length of sequences, i.e., $O\left(k^2\right)$. Besides, heuristic-based solutions could not guarantee optimal results.

In this paper, we propose, FabP, a novel deeply optimized architecture for aligning back-translated protein sequences against a database of DNA/RNA sequences. Considering the biological fact that the likelihood of observing indels is relatively low [17], [18], FabP reduces the computation complexity of alignment by only support substitutions in the alignment process. FabP proposes an FPGA-friendly encoding approach that preserves the non-uniqueness features of the back-translated protein sequence while simplifying the alignment hardware. We, in turn, perform lookup table-level optimizations to minimize the resource utilization of each required module, maximizing the performance of the entire protein alignment. Our evaluation shows that FabP outperforms state-of-the-art solutions. As compared to the multi-threaded CPU-based TBLASTN, FabP provides 24.8× speedup and 266.8× higher energy efficiency. the state-of-the-art solutions.

II. Background and Related Work

The sequence alignment problem for a query sequence $Q = \{N_1, N_2 \ldots, N_q\}$ and a reference sequence $R = \{N_1, N_2 \ldots, N_r\}$ $r > q$, can be formulated as finding the regions of the reference sequence with highest similarity to the query sequence [3]. Query and/or reference sequence elements N can be either amino acids or nucleotides. Dynamic programming algorithms have been widely used to solve the alignment problem by finding the optimum result. The Smith-Waterman (SW) algorithm is a dynamic programming technique widely used for local alignment [8], [9]. It calculates a scoring matrix for all possible alignments supporting both substitution and indel mutations. The scoring matrix size is equal to $L_r \times L_q$ where L_r and L_q are the length of the reference and the query sequences respectively. Due to the high computational cost of SW algorithm, prior work tried to accelerate it on GPUs, FPGAs, and ASICs [3], [19]. The computation complexity of the SW algorithm increases quadratically with the length of the sequences. Heuristic algorithms have been proposed to reduce the computation complexity of the dynamic programming based algorithms while delivering near-optimal results [20], [21].

Among the heuristic alignment algorithms, Basic Local Alignment Search Tool (BLAST) is the most widely used algorithm [20], [22]. It is being developed by the National Center for Biotechnology Information (NCBI) for three decades. BLAST looks for similar k-mers, subsequences of length k, in both reference and query sequences. All the k-mers of the query sequence in a hash-table and use k-mers of the reference sequence to find the similar subsequences (hits) in the reference and query sequences. Finally, it calculates the alignment score for each hit using SW algorithm.

Adapting high-performance hardware platforms such as GPUs [23] and FPGAs [24], [25] for running the BLAST

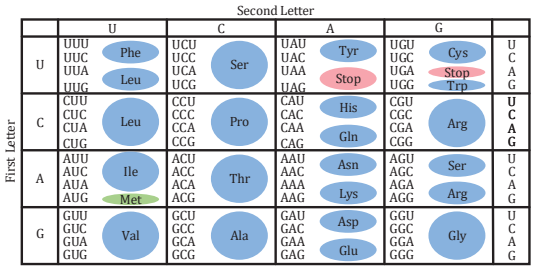


Fig. 2. Three-base RNA translation to amino acids (Codon table)

algorithm has emerged as a trend. Different variations of BLAST support alignment of protein/nucleotide sequences [26]. TBLASTN aligns protein queries against references of nucleotide sequences. It translates the reference sequences to proteins and then aligns the query with the translated reference sequence. The performance of the hash-table lookup step in all variations of BLAST algorithm including TBLASTN is limited by the numerous random memory accesses [24]. Biological observations have shown that the likelihood of observing indels in protein sequences is low [17]. Therefore, FabP calculates the alignment score based on the number of substitutions, thereby significantly reduces the computation complexity of the alignment without affecting the accuracy of the alignment. Unlike hash-table based alignment algorithms, FabP reads the reference sequences sequentially which significantly increases the memory bandwidth as compared to the random memory accesses. Thanks to the proposed highly efficient comparison module and sequential memory accesses, FabP is able to utilize the FPGA parallelism and accelerates the protein alignment.

III. Proposed FabP

FabP aligns protein sequences against a reference of nucleotide sequences that can be either DNA or RNA sequences. FabP back-translates the amino acid sequences to all the possible RNA sequences (section III-A). To support the non-unique elements of the back-translated sequence, FabP proposes an FPGA-friendly encoding method to store the back-translated amino acid sequence, proposed in subsection III-B. FabP proposes a highly optimized and pipelined accelerator which utilizes the FPGA resources characteristics to accelerate the alignment excessively, as described in section III-C.

A. Back-Translation

Figure 2 shows how proteins are back-translated to their building RNA bases according to the Codon table. To find the functionality of an unknown protein sequence, we search in a database for a known sequence that has a high similarity and characteristics. First, each amino acid in the unknown query sequence is back-translated to it's building RNA three-base sequence (codon). As the codon table shows, multiple codons can be translated to the same amino acid (e.g., both UUC and UUU represent Phe amino acid); therefore, protein back-translation will not create a unique RNA sequence. Elements in the back-translated RNA sequence can be divided into three types: **Type I:** elements that are uniquely back-translated, **Type II:** elements with non-unique back-translation without dependency to the previous elements, and **Type III:** elements which are dependent on the previous elements of the RNA.

For instance, Phenylalanine (Phe) amino acid can be originated from $\{UUU, UUC\}$ RNA sequences. To simplify the representation, we represent the back-translated codon of Phe with UUU/C, where U/C represents both of the nucleotides. In this back-translation, both of the first and second U elements refer to Type I, since they both remain the same in both of the codons; however, the last element is a Type II element since it can be either U or C. Similarly, Isoleucine (Ile) is back-translated to $AT\overline{G}$, where $\overline{G}$ means that the element can be any nucleotide except G (Type II). Serine (Ser) amino acid can be translated from four RNA sequences $\{UCU, UCC, UCA, UCG\}$, all the four codons can be aggregated and represented as UCD (D represents all the four nucleotides–Type II). In addition, Leucine (Leu) back-translation can be represented as CUD or UUA/G, which means if the first element is C the last element can be any element, while if the first element is U, the third element can be either A or G (Type III).

Aligning back-translated sequences with the RNA sequence requires a comparison function handling all the three types. For Type I elements, FabP requires to perform an exact element-wise comparison. Type II elements require conditional comparison. For Type III elements, FabP needs to perform a dependent comparison. To implement the comparator, FabP keeps the information of each element, including the type, and the matching condition and uses a flexible comparison logic to handle all the three types of elements. Thus, we proposed FabP encoding preserving all the above-mentioned information such that it simplifies the comparison logic in the FPGA.

B. FabP Encoding

The reference RNA sequences contain four different nucleotides, $\{A, C, G, U\}$, which can be encoded into 2-bit numbers as $\{00, 01, 10, 11\}$, respectively. The query sequence is an RNA sequence back-translated from a protein sequence. Due to back-translation non-uniqueness, FabP requires more than two bits to encode the back-translated elements to preserve all the aforementioned information, including the type of the element and the condition by which the query should be compared against the reference. The query elements are encoded and stored as 6-bits instruction. This instruction has three distinct fields: variable-length opcode, matching condition, and the configuration bits. Opcode field stores the type of the elements and since we have only three types we use a variable-length opcode to fully utilize the 6-bit instructions. The matching condition field keeps the condition of matching and the configuration bit field controls the flow of data.

The first two bits of the instruction are dedicated to the opcode field in the elements of Types I and II (opcodes 00, and 01, respectively). Elements of Type III are represented with a single-bit opcode 1. In elements of Type I, two bits of the opcode are followed by the element that needs to be perfectly matched. In elements of this type, since the elements must be exactly matched in both sequences, the 2-bit representation of nucleotides is used as the matching condition. In elements of Type II, opcode bits are set to 01 and FabP uses the next two bits to represent the matching conditions. Five conditions observed in the Codon table ($U/C, A/G, \overline{G}$, A/C, and D). Since two bits are dedicated to represents the conditions, the first four conditions are represented. Although D is a Type II element, for the sake of hardware simplicity, FabP encodes it with opcode 1 along with the Type III elements.

Type III elements are represented with opcode 1, followed by two bits to identify four unique functions. Each function is dedicated to a particular amino acid that has an element of the Type III. FabP uses Function $F:00$ to describe the third element of Stop, and uses functions $F:01$ and $F:10$ to describe Leu and Arg amino acids. Since there are only three amino acids that require a special function, $F:11$ remains useless. Thus, FabP uses it to represent the element D. FabP sets the fourth bit to zero for the encoded elements of Type III.

FabP dedicates the first four bits of the instructions to store the information of the back-translated codons. It uses the last two bits of the 6-bit instructions as the configuration bits to control the flow of the data. These two bits are set to 00 for elements of Type I and II. In elements of Type III, FabP sets the configuration bits based on the amino acid. In Type III, comparison of the query element i^{th} depends on the previous nucleotides (either element of $(i-1)^{th}$ or $(i-2)^{th}$) of the corresponding codon in the back-translated RNA sequence. Therefore, the configuration bits are connected to a multiplexer to select from one of the earlier elements (illustrated in Figure 5(a)).

The back-translated sequence of a query sequence Q with 5 amino acids according to the codon table is as follows:

$$Q = \{\texttt{Met} - \texttt{Phe} - \texttt{Ser} - \texttt{Arg} - \texttt{Stop}\} \xrightarrow{back-translation}$$
$$\{AUG - UU(U/C) - UUD - (A/C)G(F:10) - U(G/A)(F:00)\}$$

FabP first creates the back-translated sequence. Then, it encodes that sequence and stores it in the FPGA main memory (DRAM). To encode the query sequence, Met has a unique back-translated sequence (AUG). The next amino acid is Phe, the next two elements of the back-translated sequence are uniquely back-translated as UU; while, the next nucleotide can be either U or C. The first 5 elements (AUGUU) are Type I and will be encoded as $\{00A00, 00U00, 00G00, 00U00, 00U00\}$. The next element can be either U or C (Type II) and it will be encoded as $\{010000\}$. FabP back-translates and encodes the next amino acid in the sequence, Ser, as $\{00U00, 00U00, 001100\}$. In Arg back-translation, the first element can be either C or A, followed by a G. Then, if the first element is C the third element can be any nucleotide, D. However, if the first element is A, the last element can be only A or G. FabP implements this dependent comparison as *Function:10* in the comparison module. Thus, FabP encodes Arg as $\{010100, 00U00, 110001\}$. Similarly it encodes Stop element as $\{00U00, 010100, 100010\}$.

C. FabP Alignment

The protein alignment algorithm finds the locations where the back-translated query has a high similarity to the reference sequence. It calculates the alignment score by accumulating the results of pairwise comparisons between the query elements and sub-sequences of the reference. Calculating the alignment score for all the positions in the reference sequence is a compute-intensive task that can be parallelized in FPGA. FabP utilizes the FPGA resources to parallelize the alignment computation. At first, the database of the reference sequences is transferred to the FPGA DRAM. Then the query sequences are back-translated,

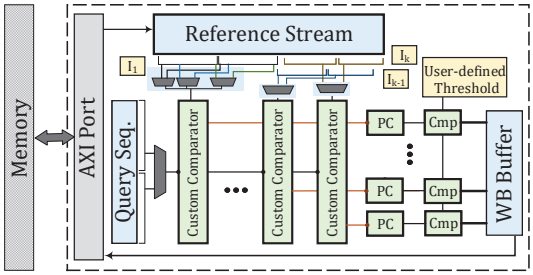


Fig. 3. The proposed architecture of FPGA-based FabP

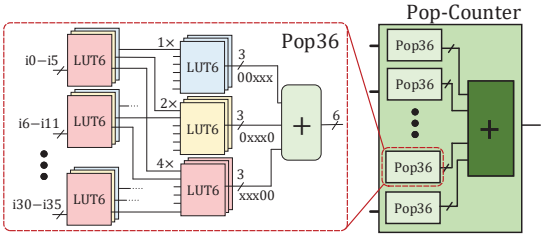


Fig. 4. Pop-Counter architecture

encoded, and transferred to the FPGA distributed memory, Flip-Flops (FFs). Then, FabP streams the reference sequence into the FPGA and aligns the query against the incoming reference subsequence.

FPGA communicates with the DRAM using AXI ports through the existing number of memory channels (CH). If each channel provides BW bandwidth, the total memory bandwidth is equal to $BW \times CH$. In practice, the AXI interface data width is 512 bits. Assuming the AXI interface can read from DRAM in every clock cycle, the nominal memory bandwidth equals to $BW = 512 \times Freq$, where $Freq$ is the clock frequency. In practice, if the memory access pattern is sequential, the achieved memory bandwidth will be close to the nominal value. In clock cycles that the AXI port does not have valid data from the DRAM, all the stages of the FabP will be stalled, and FabP will wait until the next valid data from the AXI port. In every cycle that the AXI port has valid data, FabP reads 512 bits of the reference sequence (per memory channel) from the FPGA DRAM. In the rest of the paper, we assume only one memory channel is available in the FPGA. If more memory channels are available, FabP is able to utilize multiple channels as long as the FPGA has enough resources. Since each element of the reference sequence is 2 bits, considering a single AXI interface, FabP reads 256 elements of the reference in each memory access. Finally, FabP calculates the alignment scores of a query and the incoming reference sequence in a multi-stage pipelined architecture.

Figure 3 shows the architecture of the FabP accelerator. As Figure shows, *Query Seq.* memory stores a query sequence with length of L_q. In each AXI data transfer, 256 elements of the reference are loaded into the *Reference Stream* buffer. The query sequence slides through the reference to calculates the alignment score for all the possible sequence positions. Since the probability of observing indels in parts of RNA sequences that generate proteins is relatively low, FabP only counts the differences in the query and the reference sequence. In addition, since the length of the query sequences is less than the length of the reference sequences ($L_q < L_r$), sliding the query sequence over the *Reference Stream* buffer results in $L_r - L_q + 1$ independent alignment instances.

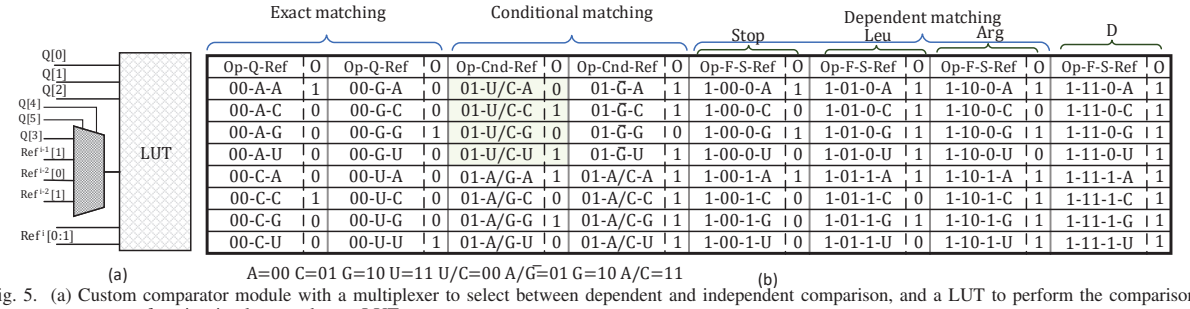
Each alignment instance calculates the independent alignment score for the query and a sub-sequence of the incoming reference sequence, as a potential solution. Having a highly parallelizable architecture, FPGAs can execute all the alignment instances concurrently. The first instance (I_1) calculates the alignment score for the query and *Reference Stream* elements between $[0 : L_q - 1]$. The k^{th} instance (I_k) compares the first element of the query with the k^{th} element of the *Reference Stream* and the L_q^{th} element of the query with the $L_q + k^{th}$ of the *Reference Stream*, in order to calculates the alignment score. To cover the alignment positions that overlap between two consecutive incoming reference sequences, FabP keeps the last L_q elements of the current *Reference Stream* buffer and concatenates it with the next incoming reference sequence. Therefore in each iteration, FabP stores a sub-sequence with length $L_{Ref.\ Stream} = L_q + 256$ in the *Reference Stream* buffer.

The number of element-wise comparisons in each alignment instance is equal to the number of the query elements. Thus, as the number of query elements increases, FabP requires more resources for each alignment instance. Due to FPGA resource limitation, for long query sizes, there are not enough resources to perform all the operations in one cycle. FabP uses a set of multiplexers to divide *Query Seq.* and *Reference Stream* into multiple segments and process each segment in a cycle. Therefore, for longer queries, FabP needs multiple iterations to calculate all the alignment instances. In each iteration, $L_r - L_q + 1$ instances of *custom comparators* compare the query and reference sub-sequences. The output of a Custom comparator is L_q bits, where each bit shows if the corresponding elements of the reference sequence and the query sequence are matched. The outputs of comparisons are aggregated in the Population Counter (pop-Counter) module, shown in figure 3 as PC, representing the alignment score. At the end, FabP uses threshold comparators to compute and write the position of every alignment instance with a higher score than a user-defined threshold. The *WB buffer* writes back all aligned positions to the FPGA DRAM using an AXI bus.

D. FPGA Optimization

To implement the custom comparator module, performing the element-wise comparisons, FabP uses only *two Lookup Tables* (LUTs). Each LUT has 6 inputs, and every function with 6 inputs can be implemented in a LUT. As Figure 5(a) shows, the last two bits of the query is used to select between performing dependent comparison (Type III) or exact/conditional comparison (type I and II). FabP uses a single 6-input LUT to implement the multiplexer. To map the multiplexers to a single LUT, we directly instantiate LUT primitives.

FabP encodes the back-translated query elements to 6-bit instruction including two configuration bits. The first four bits of the query element along with the 2 bits of the reference element are used as the inputs of a LUT, which is programmed to perform the required comparison function. The first four bits of the instructions specify the matching condition with the reference element and the content of the LUT is programmed with the result of the comparison. As illustrated in Figure 5, FabP uses one LUT to implement the multiplexer and one LUT



Op-Q-Ref	O	Op-Q-Ref	O	Op-Cnd-Ref	O	Op-Cnd-Ref	O	Op-F-S-Ref	O	Op-F-S-Ref	O	Op-F-S-Ref	O	Op-F-S-Ref	O
00-A-A	1	00-G-A	0	01-U/C-A	0	01-G-A	1	1-00-0-A	1	1-01-0-A	1	1-10-0-A	1	1-11-0-A	1
00-A-C	0	00-G-C	0	01-U/C-C	1	01-G-C	1	1-00-0-C	0	1-01-0-C	1	1-10-0-C	0	1-11-0-C	1
00-A-G	0	00-G-G	1	01-U/C-G	0	01-G-G	0	1-00-0-G	1	1-01-0-G	1	1-10-0-G	1	1-11-0-G	1
00-A-U	0	00-G-U	0	01-U/C-U	1	01-G-U	1	1-00-0-U	0	1-01-0-U	1	1-10-0-U	0	1-11-0-U	1
00-C-A	0	00-U-A	0	01-A/G-A	1	01-A/C-A	1	1-00-1-A	1	1-01-1-A	1	1-10-1-A	1	1-11-1-A	1
00-C-C	1	00-U-C	0	01-A/G-C	0	01-A/C-C	1	1-00-1-C	0	1-01-1-C	0	1-10-1-C	1	1-11-1-C	1
00-C-G	0	00-U-G	0	01-A/G-G	1	01-A/C-G	1	1-00-1-G	0	1-01-1-G	1	1-10-1-G	1	1-11-1-G	1
00-C-U	0	00-U-U	1	01-A/G-U	0	01-A/C-U	1	1-00-1-U	0	1-01-1-U	0	1-10-1-U	1	1-11-1-U	1

Fig. 5. (a) Custom comparator module with a multiplexer to select between dependent and independent comparison, and a LUT to perform the comparison. (b) Custom comparator function implemented on a LUT

to perform the comparison. For example, when the query element is $01-00-00$, the first two bits (01) identify the element as a Type II element, and the next two bits (00) shows that the query element (U/C) is matched with reference elements U or C. Therefore, the first four rows of the third column in the table (highlighted in Figure 5(b)) show the output of the comparison between the query element and any reference element.

The comparison LUT outputs whether an element of the query sequence can be originated from an element in the reference sequence. The alignment score for each alignment instance is the summation of the element-wise comparisons. We use a Pop-Counter module to aggregate the comparison results. We designed a hand-crafted hardware implementation of the Pop-Counter, shown in Figure 4, to increase the efficiency of the generated hardware. Pop-Counters contribute to a significant portion of the overall area footprint since there are $r-q+1$ PCs (one for each alignment instance). The main building block of the implemented Pop-Counter is *Pop36* that produces a 6-bit output of summing up a given 36-bit input. The first stage of Pop36 is made up of six groups of three-LUTs that share six inputs. This stage outputs the 3-bit resultants which are summed up together in the subsequent stage according to their bit order (position). For all the explained modules, We directly instantiated FPGA *LUT6* and *FF* primitive resources to build up the custom comparator and the pipelined Pop-Counter. `FabP` LUT-level optimized Pop-Counter shows 20% area reduction as compared to the simple HDL description of a tree-adder-style Pop-Counter.

IV. Experimental Results

`FabP` accelerates the protein back-translation and alignment on FPGA which has been implemented in Verilog HDL. The synthesized code has been implemented on the mid-range Kintex-7 FPGA. `FabP` host code is written in OpenCL to encode the queries and send them along with the reference sequences from the host DRAM to the FPGA DRAM. The host code invokes the RTL kernel which aligns the sequences and writes the results to the FPGA DRAM. The host code, at the end, reads the results from the FPGA DRAM. In all experiments, we measured the end-to-end execution time that includes reading both query and reference sequences from the FPGA DRAM, aligning the sequences, and writing the results to the FPGA DRAM. We compare the performance and energy efficiency of `FabP` with the state-of-the-art protein alignment tool (TBLASTN [11]) running on Intel i7-8700K CPU with 16GB memory. To evaluate the efficiency of the proposed `FabP` architecture, we compared `FabP` with our highly optimized GPU implementation on the high-end NVIDIA GTX 1080Ti GPU written in CUDA. We perform all evaluations using the query sequences randomly sampled from the NCBI protein database [27], and 1 GByte of reference sequences from the NCBI DNA Database [28].

A. FabP Evaluation

Figure 6(a) compares FPGA-based `FabP` execution time and energy efficiency with NVIDIA GPU and Intel CPU. The state-of-the-art TBLASTN is running on both single-thread and multi-thread (12 threads) CPU. All results in the figure are normalized to the single-thread execution time and energy consumption of the TBLASTN running on a single core.

The results are reported for different protein query lengths ranging from 50 to 250 elements. After the back-translation, the length of the query sequence is multiplied by three (ranging 150 to 750). Note that the length refers to the maximum sequence length, and `FabP` can work with any sequence smaller than that. Our evaluation shows that, for all platforms, increasing the number of query elements increases the execution time and energy consumption. Comparing the efficiency of different platforms, we can see that `FabP` provides slightly better throughput than the high-end GPU implementation while consuming significantly less power. Over all lengths of the query, `FabP` outperforms GPU (multi-thread CPU) performance, on average, by 8.1% (24.8×). In terms of energy consumption (Figure 6(b)), `FabP` provides significantly higher efficiency than both of the platforms. As compare to GPU (multi-thread CPU), `FabP` provides 23.2× (266.8×) higher energy efficiency at the cost of negligible drop in the alignment accuracy. This efficiency comes from `FabP` LUT-level optimization that enables highly parallel computing in a deep pipeline architecture, with a negligible drop in the alignment accuracy. Not supporting indels has a minimal impact on the alignment accuracy since indels are infrequent in protein sequences. Statistically, there is a very small probability that indel affects the protein sequence alignment results. The work in [18] has shown that the distribution of empirical frequency of indels in protein-coding regions has a median of 0 and a mean of 0.09 indels per kilobase with a standard deviation of 0.36 index per kilobase. In our experiments, among 10,000 queries, only two of them involved indels ($\sim 0.02\%$).

B. FabP Resource Utilization

Table I shows the FPGA available resources as well as the resource utilization of `FabP` implementation aligning protein sequence with length of 50 and 250 elements. As the results indicate, the FPGA has very high LUTs and FFs utilization which are used to implement the custom comparator modules and Pop-Counters. The alignment score is a 10-bit number and

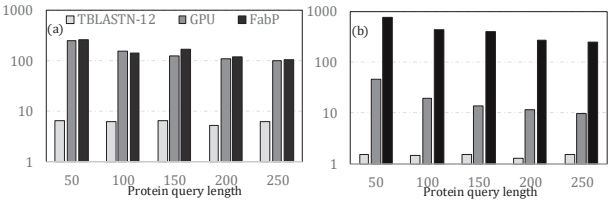


Fig. 6. Performance improvement (a) and energy efficiency (b) of FabP over GPU and 12-threaded state-of-the-art protein alignment (TBLASTN-12) running CPU

TABLE I
RESOURCE UTILIZATION OF FABP FOR THE MAXIMUM PROTEIN QUERY LENGTH OF 50 AND 250

Resources	LUT	FF	BRAM	DSP	DRAM BW
Available	326k	407k	16Mb	840	12.8 GB/s
FabP-50	58%	16%	19%	31%	12.2 GB/s
FabP-250	98%	40%	15%	68%	3.4 GB/s

to save the LUTs for the Custom comparators and Pop-Counters, FabP uses DSPs to compare the alignment score with the user-defined threshold. Regardless of the supported query length, FabP uses distributed memory resources (FFs) for the query sequence and the reference stream buffer rather than using the BRAMs to avoids the routing congestion that may happen due to high fanout of the memory blocks, and reduce the power consumption of the entire design.

As Table I also shows, FPGA has the maximum bandwidth utilization during processing both sequence lengths, meaning that the single memory channel is almost fully utilized to read the reference sequence from the FPGA DRAM. For the sequence length of 50, the memory bandwidth bounds the maximum performance/parallelism. Therefore, more memory channels will further accelerate alignment. Longer query sequences (e.g., 250) increase the number and the complexity of alignment instances. Therefore, even with fully utilized FPGA, FabP still requires multiple iterations to process the alignment. This results in a lower effective bandwidth for longer sequences. Therefore, an FPGA with more LUTs can outperform the GPU-based implementation. Our observation shows that for sequences longer than $\sim$70, the resource utilization is the bottleneck of computation; while for shorter sequences the bandwidth is the limiting factor.

V. CONCLUSION

In this paper, we proposed a novel FPGA-based accelerator for aligning back-translated protein sequences against a database of DNA/RNA sequences. Our proposed back-translation and encoding method leverages the FPGA characteristics to optimize the alignment process. FabP is deeply optimized to fully utilize the FPGA resources. Thanks to the sequential memory access, it is able to fully utilize the memory bandwidth to maximize the performance. Our evaluation shows that FabP outperforms the state-of-the-art CPU-based alignment implementation by 24.8$\times$ in performance and 266.8$\times$ in energy, respectively.

ACKNOWLEDGEMENTS

This work was supported in part by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, in part by SRC Global Research Collaboration (GRC) grant, DARPA HyDDENN grant, and also NSF grants #2028040, #1730158, #1911095, #2003279 and #1826967.

REFERENCES

[1] C. C. Liu, M. C. Jewett, J. W. Chin, and C. A. Voigt, "Toward an orthogonal central dogma," *Nature chemical biology*, vol. 14, 2018.
[2] W. Haque, A. Aravind, and B. Reddy, "Pairwise sequence alignment algorithms: a survey," in *Proceedings of the 2009 conference on Information Science, Technology and Applications*, 2009.
[3] H.-C. Ng, S. Liu, and W. Luk, "Reconfigurable acceleration of genetic sequence alignment: A survey of two decades of efforts," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017.
[4] e. a. Neuwald, Andrew F, "Extracting protein alignment models from the sequence database," *Nucleic Acids Research*, vol. 25, 1997.
[5] C. E. Bronner, S. M. Baker, P. T. Morrison, G. Warren, L. G. Smith, M. K. Lescoe, M. Kane, C. Earabino, J. Lipford, A. Lindblom *et al.*, "Mutation in the dna mismatch repair gene homologue hmlh 1 is associated with hereditary non-polyposis colon cancer," *Nature*, vol. 368, 1994.
[6] M. J. e. a. Scanlan, "Glycoprotein a34, a novel target for antibody-based cancer immunotherapy." *Cancer immunity*, vol. 6, 2006.
[7] e. a. Duwadi, Deepesh, "Identification and screening of potent antimicrobial peptides in arthropod genomes," *Peptides*, vol. 103, 2018.
[8] T. F. Smith, M. S. Waterman *et al.*, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, 1981.
[9] W. R. Pearson, "Searching protein sequence libraries: comparison of the sensitivity and selectivity of the smith-waterman and fasta algorithms," *Genomics*, vol. 11, 1991.
[10] e. a. Lodish, Harvey, *Molecular cell biology*. Macmillan, 2008.
[11] S. McGinnis and T. L. Madden, "Blast: at the core of a powerful and diverse set of sequence analysis tools," *Nucleic acids research*, vol. 32, 2004.
[12] E. M. Gertz, Y.-K. Yu, R. Agarwala, A. A. Schäffer, and S. F. Altschul, "Composition-based statistics and translated nucleotide searches: improving the tblastn module of blast," *BMC biology*, vol. 4, 2006.
[13] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly," in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018.
[14] S. Salamat, B. Khaleghi, M. Imani, and T. Rosing, "Workload-aware opportunistic energy efficiency in multi-fpga platforms," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019.
[15] C. Kachris and D. Soudris, "A survey on reconfigurable accelerators for cloud computing," in *2016 26th International conference on field programmable logic and applications (FPL)*. IEEE, 2016.
[16] B. Khaleghi, S. Salamat, M. Imani, and T. Rosing, "Fpga energy efficiency by leveraging thermal margin," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*. IEEE, 2019.
[17] B. Qian and R. A. Goldstein, "Distribution of indel lengths," *Proteins: Structure, Function, and Bioinformatics*, vol. 45, 2001.
[18] K. Neininger, T. Marschall, and V. Helms, "Snp and indel frequencies at transcription start sites and at canonical and alternative translation initiation sites in the human genome," *PloS one*, vol. 14, 2019.
[19] A. Al Kawam, S. Khatri, and A. Datta, "A survey of software and hardware approaches to performing read alignment in next generation sequencing," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 14, 2016.
[20] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, 1990.
[21] N. Homer, B. Merriman, and S. F. Nelson, "Bfast: an alignment tool for large scale genome resequencing," *PloS one*, vol. 4, 2009.
[22] e. a. McVicar, Nathaniel, "Fpga acceleration of short read alignment," *arXiv preprint arXiv:1805.00106*, 2018.
[23] W. Ye, Y. Chen, Y. Zhang, and Y. Xu, "H-blast: a fast protein sequence alignment toolkit on heterogeneous computers with gpus," *Bioinformatics*, vol. 33, 2017.
[24] M. Yoshimi, C. Wu, and T. Yoshinaga, "Accelerating blast computation on an fpga-enhanced pc cluster," in *2016 Fourth International Symposium on Computing and Networking (CANDAR)*. IEEE, 2016.
[25] M. Bekbolat, S. Kairatova, A. Shymyrbay, and K. Vipin, "Hblast: An open-source fpga library for dna sequencing acceleration," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2019.
[26] T. Madden, "The blast sequence analysis tool," in *The NCBI Handbook [Internet]. 2nd edition*. National Center for Biotechnology Information (US), 2013.
[27] "Ncbi protein database," ftp://ftp.ncbi.nlm.nih.gov/blast/db/FASTA/nr.gz.
[28] "Ncbi nucleotide database," ftp://ftp.ncbi.nlm.nih.gov/blast/db/FASTA/nt.gz.