Received: 2 March 2020 | Revised: 29 July 2020 | Accepted: 10 August 2020

DOI: 10.1002/cpe.6007

WILEY

SPECIAL ISSUE PAPER

A CPU-FPGA heterogeneous approach for biological sequence comparison using high-level synthesis

Carlos A. C. Jorge[1] | Alexandre S. Nery[2] | Alba C. M. A. Melo[1] | Alfredo Goldman[3]

[1]Department of Computer Science, University of Brasilia, Brasilia, Brazil

[2]Department of Electrical Engineering, University of Brasilia, Brasilia, Brazil

[3]Institute of Mathematics and Statistics, University of Sao Paulo, Sao Paulo, Brazil

Correspondence
Carlos A. C. Jorge, Department of Computer Science, Predio CIC-EST, Campus Asa Norte, University of Brasilia, CEP-70900-100, Brasilia, Brazil.
Email: cacjorge@aluno.unb.br

Funding information
CNPq 426729/2018-8; FAPDF 00193-00002139/2018-79; Capes/PROCAD 183794

Summary

This article presents a high-level synthesis implementation of the longest common subsequence (LCS) algorithm combined with a weighted-based scheduler for comparing biological sequences prioritizing energy consumption or execution time. The LCS algorithm has been thoroughly tailored using Vivado High-Level Synthesis tool, which is able to synthesize register transfer level (RTL) from high-level language descriptions, such as C/C++. Performance and energy consumption results were obtained with a CPU Intel Core i7-3770 CPU and an Alpha-Data ADM-PCIE-KU3 board that has a Xilinx Kintex UltraScale XCKU060 FPGA chip. We executed a batch of 20 comparisons of sequences on 10k, 20k, and 50k sizes. Our experiments showed that the energy consumption on the combined approach was significantly lower when compared to the CPU, achieving 75% energy reduction on 50k comparisons. We also used the tool proposed in this article to do a case study on Covid-19, with real SARS-CoV-2 sequences, comparing their LCS scores.

KEYWORDS

biological sequence comparison, dynamic programming, field-programmable gate array, high-level synthesis, longest common subsequence

1 | INTRODUCTION

Sequence pattern recognition is one of the most challenging problems in molecular biology and computer science. Given a set of sequences, one must find the pattern that occurs most frequently. In exact pattern matching, the search for a pattern of *m* letters can be solved by a simple enumeration of all the patterns of *m* letters that appear in the strings. However, when working with biological sequences, the approximate pattern matching is usually performed because the patterns include nucleotide mutations, insertions or removals.[1] The focus of this article is on approximate pattern matching.

A sequence is considered as an ordered set of residues, that is, characters, and the sequence alignments clearly expose the patterns that occur most frequently, being useful to discover functional, structural, and evolutionary information in biological sequences. Therefore, it is important to find the optimal alignment, which maximizes the similarity between the sequences. Very similar sequences probably have the same function and, if the sequences are from different organisms, they may be defined as homologous, if there is a sequence which is ancestral to both.[2] Sequence similarity can be an indication of several possible ancestral relationships, including the absence of a common origin.[3]

When comparing two biological sequences, metrics are calculated using computational methods to help identify their degree of relationship. One of these metrics is the *score*, which is assigned to an alignment. An alignment is defined as a pairing, residue by residue, of the sequences. A pair of residues (characters) from the two sequences can be defined as *match*, when the characters are equal; *mismatch*, when the characters are distinct; or a *gap*, when the residue of a sequence is aligned with a gap.[2] Among the various biological sequence comparison algorithms in the literature, the *longest common subsequence (LCS)*[4] algorithm is one of the most used. Other algorithms that are also used for sequence comparison are Smith-Waterman,[5] Needleman-Wunsh,[6] and Hirschberg.[7]

Despite its long history, research on sequence alignment continues to flourish. Sequence alignment in modern computational biology is the basis of many bioinformatics studies, and advances in alignment methodology can provide comprehensive benefits in a wide variety of application domains.[2] While many of these approaches depend on the same basic principles, the details of the implementations can have major effects on performance, both in terms of accuracy and speed.

The algorithms that produce optimal results, such as LCS, are executed in quadratic time ($O(n^2)$) where n is the length of the sequences. For this reason, its execution time is high if the sequences compared are long. Therefore, parallel hardware-dedicated architectures such as *field programmable gate arrays (FPGAs)* have been employed to accelerate the execution of sequence alignment algorithms. Such hardware systems are usually designed using hardware description languages (HDL), such as Verilog, System Verilog, and VHDL, which are low-level languages. As the name suggests, one can describe the hardware (digital systems) using HDLs. Nevertheless, designing hardware on FPGAs using HDLs requires in-depth knowledge of digital electronics, in addition to being more costly in terms of time to implement the solution, which has, consequently, a high impact on the overall project cost.

High-level synthesis (HLS) is the process that translates a system functionality described in a high-level language (usually C, SystemC, C ++ or Matlab), and produces an *register transfer level* (RTL) architecture corresponding to the implementation of the functionality on a target device (e.g., FPGA).

HLS was introduced to facilitate the specification and implementation of RTL architectures from high-level code,[8] significantly reducing the design time for complex systems. It also facilitates the design of these systems when it comes to achieving a particular model required by the hardware without worrying too much about the components of the circuit, which is especially important for software developers who, in general, do not have adequate training to specify digital systems. Finally, HLS also contributes to code portability, since the system described in high-level language can be compiled and executed on Von-Neumann architectures (e.g., traditional CPU architectures) and can also be recompiled by HLS for RTL implementation on other target devices.

With that in mind, in the present work, we propose an approach for biological sequence comparison with the LCS algorithm in HLS to compare two medium-sized DNA sequences (up to 50,000 residues), analyzing the resources used, execution time, and energy consumption of the proposed design. Our purpose is to provide an efficient implementation targeted to CPUs and FPGAs, where energy consumption is reduced. For this, we use the FPGA *Block RAMs* to store data (instead of the DDR memory present in the FPGA) and fill the diagonal matrix by diagonal. The experimental results show that the energy consumption of the FPGA solution is significantly lower than that of the CPU only solution and that the CPU-FPGA approach presents a very good tradeoff between execution time and energy consumption.

A preliminary version of the article was published in Reference 9 and it dealt with each implementation separately. In this article, we consider the application as a set of comparisons and propose a task allocation module for heterogeneous architectures (CPU+FPGA). With this, the user can either make one pairwise comparison or multiple pairwise comparisons in a single application. We show that using both the CPU and the FPGA for the LCS application composed of several comparison tasks has benefits in terms of execution time and energy consumption, when compared to the standalone CPU and standalone FPGA versions.

The increasing demands for computing performance have been a reality regardless of the requirements for smaller and more energy efficient devices, with the FPGA achieving much greater energy savings than the CPU in most situations. In our particular application, the CPU has better performance whereas the FPGA consumes less energy. In this scenario, our heterogeneous approach (CPU+FPGA) takes advantage of the best characteristics in both platforms, achieving a very good trade-off between execution time and energy consumption, when compared to the standalone solutions.

Our approach was used to do a case study on the Covid-19 disease. In this case study, we selected 20 real SARS-CoV-2 sequences from different countries, obtained in a three-month period of time. The sequences were pairwise compared to the SARS-CoV-2 reference sequence and they were ranked by the LCS score. We concluded that the least similar sequences were obtained more recently and indicate two sequences (MT750353.1 and MT762396.1, from USA and Bangladesh, respectively) that are of special interest and need further investigation from the biologists.

The remainder of this article is organized as follows. Section 2 presents the LCS problem and the algorithm that solves it, providing the optimal result. Related work in the area of parallel biological sequence comparison with FPGAs is discussed in Section 3. The design of the proposed architecture is presented in Section 4 and experimental results are shown in Section 5. Finally, Section 6 concludes this article and suggests future work.

2 | LONGEST COMMON SUBSEQUENCE

2.1 | Overview

The essence of the LCS problem[4] between the elements of a set of symbol strings of any kind is to determine the degree of similarity between the strings that are part of the set.

A sequence is an ordered set of objects, commonly alpha-numeric characters. As an example, the letters A, C, G, and T represent the four nucleotides of a DNA strand (i.e., the bases adenine, cytosine, guanine, and thymine), and sequences such as ACCCGGTTT represent a DNA sequence. Any subsequence can be obtained by extracting zero or more characters from the original sequence, maintaining its order. For example, the sequences ACCCGGTTT, ACC, AGTT, and ACGT are all subsequences of ACCCGGTTT. By extension, the empty sequence is present in all sequences in nature, and every sequence is a subsequence of itself.

A subsequence common to two sequences is defined as a sequence that occurs in both sequences. A maximum common subsequence is one that, of all common subsequences, has the longest length. For the purpose of studying LCS, we may completely abstract the meaning of each symbol in any given sequence and keep the focus only on finding the maximum subsequence between strings. For this reason, it is common to define LCS as the problem of determining the largest substring of characters common to two strings. Figure 1 illustrates an LCS alignment between sequences $A = AATCGC$ and $B = ATGAC$.

The LCS problem is a problem with quadratic time complexity ($O(mn)$), where m and n are the lengths of the sequences. For this reason, this problem has an increasing difficulty of practical realization with the growth of the size of the sequences compared.

Given two sequences A and B with lengths m and n, respectively, where $A = A_1, A_2, \ldots, A_m$ and $B = B_1, B_2, \ldots, B_n$, we compute the LCS dynamic programming (DP) matrix $D(i, j) = \delta(A(i), B(j)), 0 \leq i \leq |A|, 0 \leq j \leq |B|$, where $|A|$ is the length of sequence A. We define $D(i, j)$ as the maximum common subsequence between $A[1..i]$ and $B[1..j]$. The LCS score between the whole sequences is found at $D(m, n)$. The recurrence equation for the calculation can be described as shown in Equation (1).[4]

$$D(i,j) = max\begin{pmatrix} D(i-1,j-1) + \gamma(A(i) \rightarrow B(j)) \\ D(i-1,j) + \gamma(A(i) \rightarrow \Lambda) \\ D(i,j-1) + \gamma(\Lambda \rightarrow B(j)) \end{pmatrix}$$

In Equation (1), the first element considers the *match* or *mismatch* scenario between two characters ($A[i]$ and $B[j]$) in the sequences, the second element considers the *gap* scenario in the first sequence and the third considers *gap* in the second sequence. Since the goal is to maximize the similarity, the value 1 is assigned to *matches* and the value 0 is assigned to *mismatches* and *gaps*. Figure 2 illustrates the DP matrix D for the sequences shown in Figure 1.

The recurrence relation shown in Equation (1) has data dependencies so, $D(i, j)$ is calculated after the computation of $D(i-1, j-1)$, $D(i-1, j)$, and $D(i, j-1)$. This leads to the observation that the elements of a given anti-diagonal of matrix D can be computed in parallel.

Since we intend to compare medium-sized sequences (up to 50,000 characters) in this work, the size of the DP matrix may be substantial.

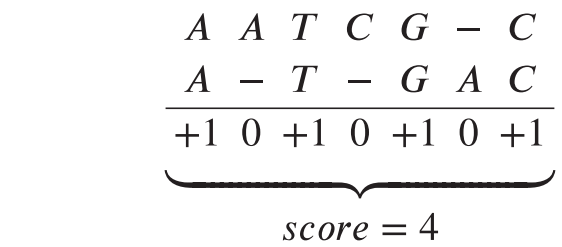

FIGURE 1 Example of LCS alignment and score

	*	A	A	T	C	G	C
*	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1
T	0	1	1	2	2	2	2
G	0	1	1	2	2	3	3
A	0	1	2	2	2	3	3
C	0	1	2	2	2	3	4

FIGURE 2 DP matrix for LCS alignment between sequences A and B

2.2 | Data dependencies and parallelism

The LCS recurrence relation shown in Equation (1) has data dependencies, so $D(i, j)$ is calculated after the computation of $D(i-1, j-1)$, $D(i-1, j)$, and $D(i, j-1)$. Therefore, the elements of a given anti-diagonal of matrix D can be computed in parallel, in a wavefront manner. In Figure 3A, we show the dependencies for a 4×4 DP matrix and the seven anti-diagonals ($d1$ to $d7$) which can be computed in parallel.

The data dependencies expressed by Equation (1) lead to a diamond-shaped task graph (Figure 3B). Note that, in this graph, the dependencies on $(i-1, j-1)$ are redundant, so they can be removed from it. The properties of the diamond-shaped dependency graph have been studied for several decades in the literature. The seminal work of Papadimitriou and Ullman[10] discusses the diagonal-striped row-based pattern to process diamond-shaped graphs and defines a lower communication bound for it. In 1998, lower bounds on the execution time for diamond-shaped graphs (called *dim*-D grid in this article) are proposed by Bampis et al.[11] The authors show that, if the number of processing units is unbounded, $(2(dim-1))(n-1)+1$ is a lower bound on the execution time, where *dim* is the dimension of the grid and n is the size of one sequence. In the paper, it is shown that the antidiagonal processing is able to attain this lower bound on execution time. More recently, Daoudi et al[12] considered the scheduling problem in diamond-shaped graphs with large communication delays and showed upper and lower bounds by dividing the *nxn* matrix into squares

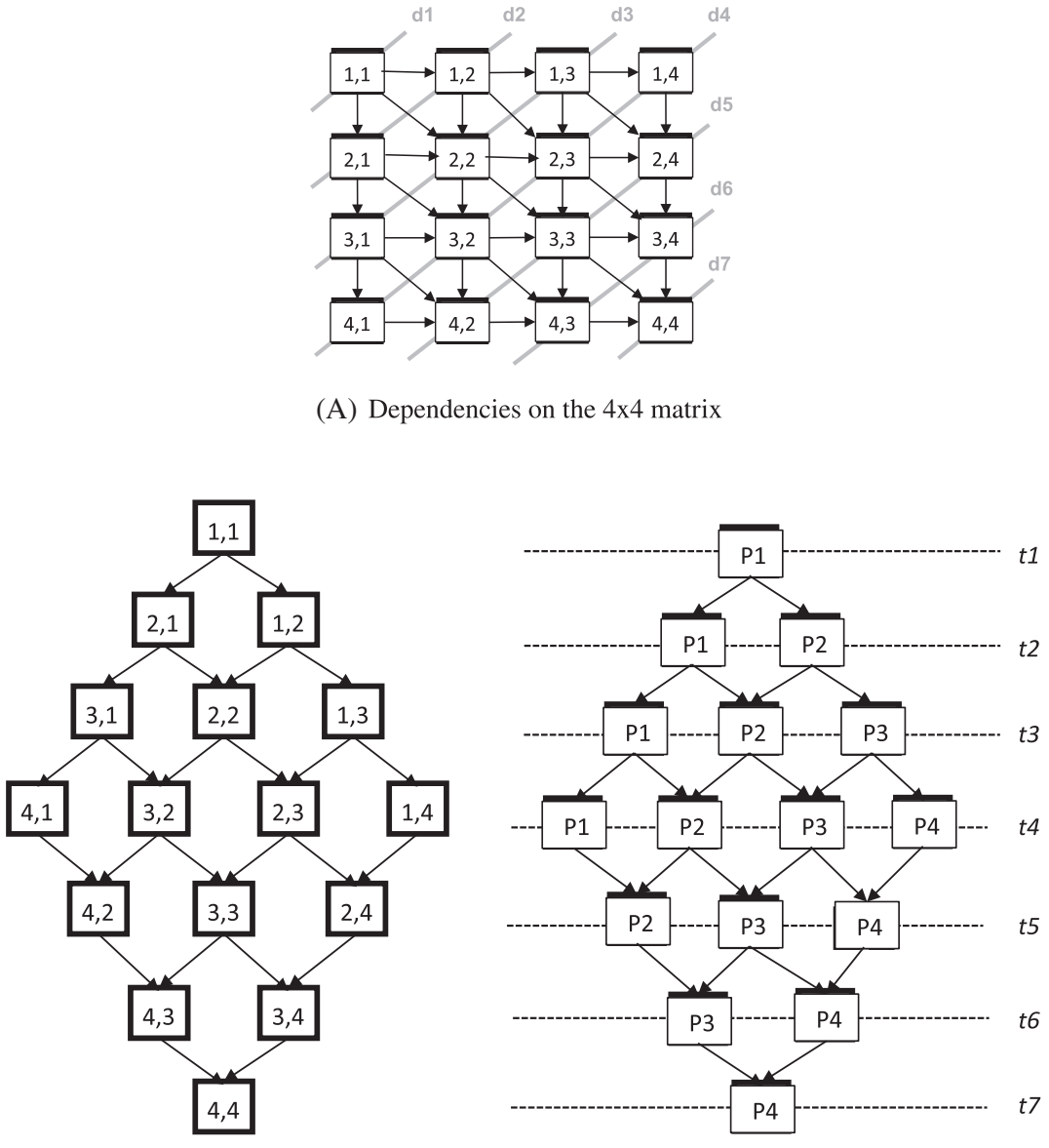

FIGURE 3 Dependencies in the LCS DP matrix and its associated graph

and rectangles, with anti-diagonal processing inside them. In Figure 3C, assuming negligible communication times, it takes 7 time units to compute the 4×4 2-DP matrix, which is the lower bound ($2 * 3 + 1 = 7$) of Reference 11. Therefore, a good strategy to compute the LCS is to process its matrix anti-diagonal by anti-diagonal, in a wavefront approach.

In the literature, several papers have discussed mapping arbitrary dependency graphs into task graphs and proposed performance prediction models for these generic cases. D'Amore et al[13] propose a generic model for multi-level performance analysis of parallel algorithms, considering dependency relations, the problem, the algorithm, and the architecture. First, partially ordered relations are used to define dependency relations for groups and then dependency matrices are defined inside each group. Second, the problem is decomposed and the subproblems are expressed as decomposition matrices. Third, the algorithm to solve the problem is characterized as a set of operators and, fourthly, the architecture, with a fixed number of computational units, is defined. Finally, an execution matrix is built that maps the algorithm into the architecture. The authors define the upper limit as the maximum number of independent sub-algorithms that can be executed simultaneously in the architecture and the lower limit is provided by the dependency degree. The performance prediction model proposed in Reference 13 was employed for the MGRIT (MUlti-Grid In Time) algorithm[14] and a simplified version of the model was applied for a matrix-multiply algorithm.[15]

3 | RELATED WORKS

Because of the quadratic execution time complexity, many parallel solutions have been proposed in the literature to accelerate biological sequence comparisons. In this section, we discuss high performance computing based proposals that implement LCS (Section 2) and associated sequence comparison algorithms such as Smith-Waterman,[5] Needleman-Wunsch,[6] and Hirschberg.[7]

Al Junid et al[16] present the design and development of a high performance acceleration and memory optimization technique to align DNA sequences with the Smith-Waterman (SW)[5] algorithm. The article focuses on optimizing memory and speed, by converting the sequences to 2-bit values and compressing these values before the alignment. The goal of this optimization technique is to use data compression to reduce the data transmitted from the desktop to the accelerator, which in this case is an FPGA. The proposed technique was designed and implemented on an Altera Cyclone II 2C70 FPGA, with the clock set to 50 MHz, and the code was written in Verilog HDL. The authors do a theoretical analysis of the reduction in memory space, which can be up to 4x lower than the traditional character based method.

Chen et al[17] propose a reconfigurable systolic architecture for the sequence alignment problem that retrieves the optimal global alignment[6] in linear space using Hirshberg's divide and conquer technique.[7] The authors use an Altera Cyclone II EP2C35 FPGA with VHDL programming, the clock was set to 50 MHz, and the interface with the desktop host is done through the serial interface (RS232). Only simulation results are provided.

Mousavi et al[18] propose an algorithm based on the constructive beam search method for the LCS problem applied to more than two sequences. This method is based on breadth-first search but, instead of keeping all leaves, the number of leaves kept is restricted to β. Thus, this is a heuristic method and there is no guarantee that the optimal result will be found. The authors created a heuristic inspired by the theory of probability and use special data structures and DP to reduce the time complexity of the algorithm. The proposed algorithm is compared to the original beam algorithm[19] using real and random biological sequences. The platform used was an old machine (Intel Pentium IV, 3.40 GHz clock, and 1 GB of RAM) and β was set to 200. Up to 100 sequences with 5000 characters were compared and the authors show that their solution is able to find alignments with better quality in less time, when compared to Reference 19, in most cases tested.

Ozsoy et al[20] propose a technique to execute the LCS algorithm with multiple GPUs, using *bit-wise* operations and performing a post-processing step. They used the NVIDIA M2090 Fermi platform with CUDA programming and obtained an 8.3x speedup in relation to the CPU with DNA sequences up to 4000 in size.

Cinti et al[21] present a new algorithm for online approximate string matching (OASM) capable of filtering *shadow hits* in real time, according to general purpose priority rules that assign priorities to overlapping occurrences. An FPGA implementation of OASM is proposed and compared with a serial software version. Even when implemented in entry-level FPGAs, the proposed procedure can achieve a high degree of parallelism and superior performance over time compared to software implementation, while keeping the usage of logic elements low. The authors used an Intel i7 4700MQ CPU and an Altera Cyclone IV E FPGA with C ++ programming for the CPU and VHDL for the FPGA, and compared 3104 synthetic sequences.

Finally, we have Alser et al[22] featuring an algorithm called Shouji, a highly parallel realignment filter, which uses a sliding search window approach to quickly identify different sequences, without the need for computationally expensive alignment algorithms. Shouji relies on a filtering algorithm that reduces the need for optimal alignment by quickly excluding different sequences from the optimal alignment calculation and makes use of the parallelism from FPGAs to accelerate this new filtering algorithm. The authors used Intel i7-3820 CPU and FPGA Xilinx Virtex-7 VC709 with C programming for CPU and Verilog for FPGA. The size of the largest test sequences is 250 characters and the FPGA execution resulted in a speedup of 1.07x when compared to the CPU.

Table 1 shows that the papers studied in this section used small DNA sequences (up to 4000 characters). In addition, none of the papers considered energy consumption. It should still be noted that the FPGA-based solutions used HDL programming languages (VHDL or Verilog).

TABLE 1 Related works

Paper	Year	Technique	Platform	Programming	Size	Power	DNA/Prot
Junid et al.	2010	Data Compression+SW	Altera Cyclone II 2C70	Verilog (FPGA)	1024	NA	DNA
Chen et al.	2011	Divide and Conquer+Systolic Array+NW	Altera Cyclone II EP2C35	VHDL (FPGA)	NP	NA	DNA
Mousavi et al.	2012	Constructive Beam Search Method	Intel i7 2770	Java (CPU)	100	NA	DNA
Ozsoy et al.	2013	LCS	NVIDIA M2090 Fermi	CUDA (GPU)	4000	NA	DNA
Cinti et al.	2018	OASM SW-OASM and HW-OASM	Intel i7 4700MQ+Altera Cyclone IV E	C++ (CPU)+VHDL (FPGA)	3104	NA	NP
Alser et al.	2019	Shouji	Intel i7-3820+Xilinx Virtex-7 VC709	C (CPU)+Verilog (FPGA)	250	NA	DNA

Note: NA stands for *Not Addressed* and NP stands for *Not Provided*.

4 | PROPOSED ARCHITECTURE

As stated in Section 3, most of the FPGA approaches for sequence comparison in the literature use hardware description languages (VHDL or Verilog) in their design. Even though this strategy may lead to high performance, the development time is very long, it is highly error-prone and the portability of code is low. In order to tackle these issues, high-level languages such as Vivado HLS have been proposed and they are indeed a very good alternative to the traditional hardware design.

4.1 | Design of the FPGA component

In this work, the *Vivado HLS* tool developed by *Xilinx* is used in order to provide support for the development of circuits in Xilinx FPGAs. The tool allows the functional specification of a high-level system (C/C ++) to be used for the production of a circuit at *RTL* level, without the need to do it manually.[23] In addition, it provides some compilation directives for optimizing the RTL architecture produced, for example: *loop unrolling*, *pipeline*, *array partition*, and so on.

The usual way to compute LCS in FPGAs is designing a systolic array composed of x processing elements (PEs), which compute x matrix elements in the same anti-diagonal of D in parallel, at the same clock cycle,[17] in order to be close to the lower bound on execution time (Section 2.2). In most designs, there may be hundreds of PEs. Since we intend to compare medium-sized sequences (up to 50,000 characters) in this work, the size of the DP matrix may be substantial. Hence, it is expected that there will be less PEs than the size of the longest anti-diagonals. In this case, partitioning strategies are employed, where the DP matrix is divided into submatrices. Since, in this article, the wavefront-based algorithm is programmed in HLS, we do not have total control over the code that will compose the FPGA bitstream. Nevertheless, we can force anti-diagonal processing by making the loop iterate over the anti-diagonals and that is what we did in our solution.

Figure 4 illustrates our FPGA design. It can be seen (from right to left) that the HLS synthesis translates the functional specification of the LCS into a corresponding RTL architecture with four main components: I/O ports, state machine, data, and memory elements, which communicate via

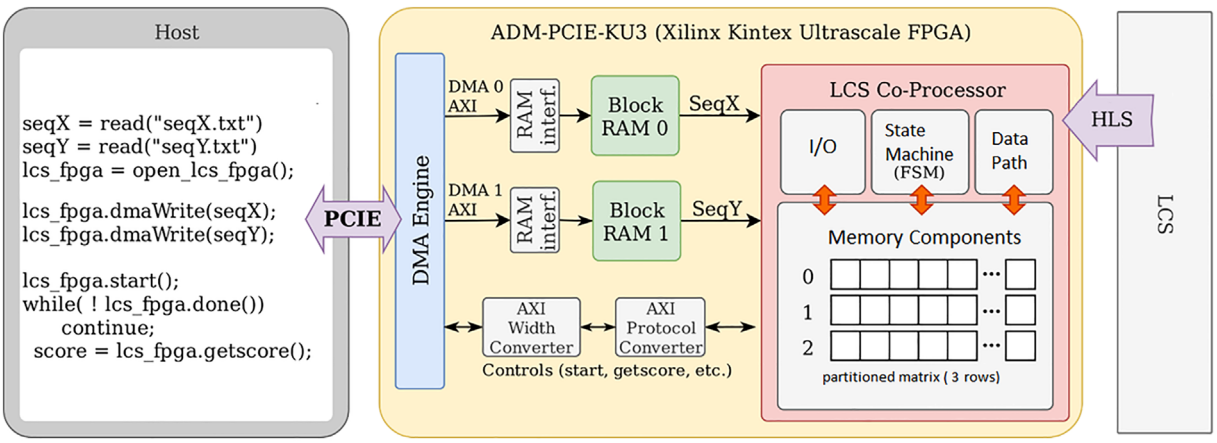

FIGURE 4 Architecture of the proposed FPGA implementation

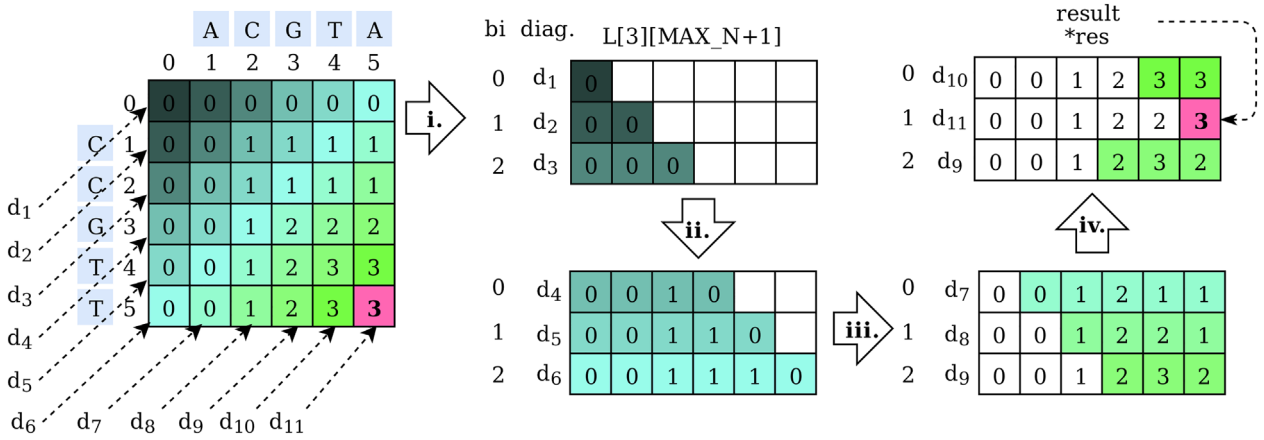

FIGURE 5 Dynamic programming matrix calculation flow using only three rows to optimize BRAM usage

specific data lines or buses. Such components are created from the analysis of the functional specification and the extraction of its control-flow and arithmetic/logic operations. This process is not exclusive to the LCS and, in general, occurs for any and all functional specifications given to the HLS. Figure 4 also presents the general architecture of the proposed solution showing the interaction between a CPU (Host) and the FPGA, via PCI-Express. The CPU transfers to the FPGA (*Block RAMs*) the two sequences to be compared and the circuit generated by the HLS (Co-Processor LCS) already in the FPGA calculates the DP matrix *D* (using only three rows) and returns to the CPU the maximum LCS score between the two sequences.

The algorithm implemented in the LCS Co-Processor is an adaptation of the traditional LCS algorithm, calculating the DP matrix diagonal by diagonal, and using only three rows of the DP matrix at a time. Figure 5 exemplifies the workflow of the algorithm for two sequences of length 5 (ACGTA and CCGTT). The calculation of the DP matrix is done diagonally using three vectors that store the computation results as it progresses within the matrix, such that the vector with the oldest diagonal becomes the current diagonal, as depicted in steps ii., iii., and iv. of Figure 5.

The main code snippet that specifies LCS using Vivado HLS is presented in Algorithm 1. Initially, a static matrix of three rows L[3][MAX_N+1] is defined in line 2, where the intermediate results will be stored as the execution of the LCS algorithm progresses. The Vivado-HLS tool supports arbitrary precision data types, so that the FPGA resources can be used more efficiently. Thus, the type *u16* represents unsigned 16-bit integer data, which can store score values up to 65,536. Then, we present the LCS routine (lines 4 to 34).

Algorithm 1. Excerpt from LCS functional specification using Vivado HLS

```
1  #define MAX_N 50000
2  static u16 L[3][MAX_N + 1]; //only 3 lines matrix to optimize BRAM usage
3
4  void lcs(volatile uchar X[MAX_N],volatile uchar Y[MAX_N],int m,int n,int *res) {
5
6    #pragma HLS INTERFACE s_axilite port=m,n,res,return bundle=Ctrl
7    #pragma HLS ARRAY_PARTITION variable=L block factor=3 dim=1
8
9    u2 bi = 0;
10   for (int line=1; line <= ((m + 1) + (n + 1) - 1); line++) {
11     int start_col = max(0, line - (m + 1));
12     int count = min3(line, ((n + 1) - start_col), (m + 1));
13
14     if (bi > 2) bi = 0; //alternates between one of the 3 L lines
15
16     for (int k = 0; k < count; k++) { #pragma HLS PIPELINE
17       int i = (min((m + 1), line) - k - 1);
18       int j = (start_col + k);
19
20       if (i == 0 || j == 0) { L[bi][j] = 0; }
21       else if (X[i - 1] == Y[j - 1]) {
22         if (bi == 0)       L[bi][j] = L[1][j - 1] + 1;
23         else if (bi == 1)  L[bi][j] = L[2][j - 1] + 1;
24         else               L[bi][j] = L[0][j - 1] + 1;
25       } else {
26         if (bi == 0)       L[bi][j] = max(L[2][j - 1], L[2][j]);
27         else if (bi == 1)  L[bi][j] = max(L[0][j - 1], L[0][j]);
28         else               L[bi][j] = max(L[1][j - 1], L[1][j]);
29       }
30     }
31     if (start_col < m) bi ++; //increments if there are diagonals to process
32   }
33   *res = L[bi][n];
34 }
```

The synthesis of interfaces is shown in line 6, which specifies that arguments *m*, *n*, and **res* will obey the protocol *AXI-Lite slave*, and that will still be grouped in a Ctrl (Control) port. Therefore, each of these arguments will be registers that can be accessed by memory mapping through a host machine.

Line 7 specifies an optimization of array partitioning, indicating that matrix L should be partitioned into three blocks in dimension 1, that is, three vectors. This is important so that the HLS synthesizer can instantiate memory elements with separate read/write addresses for each row in the matrix, allowing access in parallel to any of the three rows. Otherwise, the entire matrix could be mapped into a set of BlockRAMs with a single read/write address, which can generate a bottleneck in the system.

Then, in line 8, the variable *bi* indicates the index of the vector (matrix row) that represents the diagonal to be accessed and modified. The type *u2* represents 2-bit integer data, with no sign, since the variable *u2* can only index one of the 3 diagonals.

Starting from line 10, we have the iterations in calculating the DP matrix by diagonal (indicated by the variable *line*) and storing the values in the matrix *L* partitioned into three diagonal vectors. Note that partitioning is automatic, that is, the form of access to matrix L does not need to be modified to indicate access to each of the vectors in particular.

In line 16, the HLS optimization known as PIPELINE was also inserted. The PIPELINE pragma generates in the RTL architecture a pipeline with a certain initiation interval for a function or *loop*, allowing the simultaneous execution of operations in different stages of the pipeline, decreasing the execution time between each of the iterations. During the synthesis, the initiation interval (*Initiation Interval*, II) is set to 1, indicating that at each cycle a new operation of the function or *loop* can be started. If this initial value is prohibitive for circuit production within the time specifications provided by the developer, the value is increased until reaching an acceptable initiation interval for the proposed circuit to operate according to the desired specifications.

Finally, the result (maximum score identified between the two input sequences) will be stored in the last element of the DP matrix L indicated by the variables *bi* and *n* (Line 33).

We used the approach of Reference 13 (section 2.2) partially, with a focus on expressing the dependencies of the LCS problem and determining the complexity of the algorithm we employed to solve it. The LCS dependencies are shown in Equation (1) and illustrated for the 4×4 case in Figure 3A,B. Since each cell (i, j) of the DP matrix ($i = 1..n$, $j = 1..m$) depends on three previously calculated cells $(i-1, j-1)$, $(i-1, j)$, $(i, j-1)$, we can see that, by transitivity, every cell depends on cell (1, 1) and cell (n, m) depends on all previously computed cells. We can also observe that there is parallelism on the computation of every anti-diagonal *di* of the matrix but anti-diagonal $di+1$ must be computed after the computation of anti-diagonal *di*.

More formally, the dependencies can be expressed as[13]: (a) $\forall(i, j)\ A_{i,j} \leftarrow A_{i-1,j}$ and (b) $\forall(i, j)\ A_{i,j} \leftarrow A_{i,j-1}$. Using the notation of Reference 13, the independent elements are expressed as $\forall(i, j, k, l)\ A_{i,j} \nleftrightarrow A_{k,l}$ if $(i + j) = (k + l)$. Please note that this last condition states that the anti-diagonal elements are independent.

In our design, we employed an algorithm that calculates the matrix anti-diagonal by anti-diagonal, with a single loop that iterates over the anti-diagonals, with synchronization at the end of each anti-diagonal computation. Assuming that there is a ring connection between the processing units, that is, P_i is directly connected to P_{i-1} and P_{i+1}, at the end of each anti-diagonal computation, each P_i communicates its previously calculated value to its right neighbor at the same time. So, we can say that the communication time is a constant *c* and communication occurs after the calculation of each anti-diagonal. Assuming that *m* and *n* are the lengths of the sequences, the DP matrix has $m + n - 1$ diagonals. It can be easily seen that this algorithm has $O(m + n)$ time complexity in the best case, where $P \geq (m + n)$ is the number of processing units. In the worst case, for example, $P = 1$, time complexity is $O(mn)$ since there are $m \times n$ cells in the matrix.

4.2 | Design of the CPU+FPGA scheduler

In many situations, the biologists want to do more than one pairwise comparison. So, they define the application as a set of *c* comparisons, where each comparison executes the LCS algorithm for one pair of sequences. This is the target application that we consider in this section.

In order to distribute the comparisons between the FPGA and CPU platforms, a weight-based round robin scheduler was developed. Two weights are assigned to each of the platforms, one related to execution time and one related to energy consumption. These weights are obtained experimentally by running a profiling with sequence comparisons in both platforms. Then, the weights obtained are compared to calculate the proportion of energy consumption and execution time.

Once the weights are configured, the scheduler generates allocations for the CPU and FPGA, dividing the comparisons between the platforms, which occur simultaneously, after the division.

Algorithm 2 presents scheduler pseudo-code. In line 3, the files containing the sequences to be compared are read. In line 4, the weights for CPU and FPGA are obtained. Then, in line 5, the scheduler uses the weights to assign comparisons to the devices (CPU or FPGA) by writing the ids of the comparisons into the CPU or FPGA local queues. After that, two separate threads (*threadCPU* and *threadFPGA*) access their local queues and execute the comparisons in parallel (lines 8 and 9). We also show the directives for obtaining the execution time in lines 7 and 10. When energy consumption is considered, in line 7 we start measuring the power and line 10 marks the end of the measurement.

Algorithm 2. Excerpt from the scheduler using C++

```
1  std::vector<std::string> files;
2  std::string path = PATH;
3  read_files_on(files,PATH);
4  obtain_weights(&cpuWeight, &fpgaWeight);
5  weighted_scheduler(files, cpuQueue, fpgaQueue, cpuWeight, fpgaWeight);
6  // both threads are executed in parallel
7  int timeExecCPU, timeExecFPGA,timeExecTotal;
8  thread(threadCPU,cpuQueue,timeExecCPU);
9  thread(threadFPGA,fpgaQueue,timeExecFPGA);
10 getTime(timeExecTotal);
```

5 | EXPERIMENTS

The test environment used for the experiments was a dedicated machine with an Intel Core i7-3770 CPU 3.40GHz and an Alpha-Data ADM-PCIE-KU3 card that has an Xilinx Kintex UltraScale XCKU060 FPGA. The circuit generated by the implementation of our proposed FPGA component (Section 4.1) has a frequency of 250MHz and uses the resources of the board as shown in Table 2.

In order to create the set of comparisons, we run an automated script that generated 40 random DNA sequences of each size (10k, 20k, and 50k), with a total of 20 comparisons for each sequence size.

In the case of the FPGA solution, the *pipeline* initiation interval was automatically set to 4 by the tool, that is, every 4 cycles it starts computing a new diagonal element. We tried to set the initiation interval to 1, 2, and 3 but the Vivado tool was not able to handle it at 250 MHz due to data dependencies.

5.1 | FPGA and CPU standalone results

In this section, the power was measured using the sensors available on the Alpha-Data board. The sensors measure the amperage and voltage of the two main power rails that power the board components, including the FPGA chip. The *sysmon* tool is part of the reference design of the *Alpha-Data* board, in which it is possible to view the amperage and voltage recorded by the sensors during the time of execution of the algorithm. The energy is the same for all FPGA tests, as a single circuit has been implemented with the capacity to store sequences of up to 50,000 characters each. For the CPU, the *powerstat*[24] tool was used, which measures the electrical energy of the processor when the algorithm is running.

The execution time was measured through the *host* as shown in Algorithm 2 (Lines 7 and 10) so, we start measuring the time when the CPU and FPGA threads are created and stop measuring when both threads finish execution, that is, the execution time is the time of executing the 20 comparisons. In the case of the FPGA, it includes the time of data transfer to the FPGA to the return of the score to the *host*. Since the tests were done in a dedicated environment, the variance in repeated executions is negligible.

Table 3 shows the results obtained for each test performed in the FPGA and CPU separately. In this table, the energy (*energy*) was calculated by multiplying the *power* by the execution time.

As can be seen in Table 3, the execution time of the three comparisons (10k, 20k, and 50k) is shorter on the CPU, being around 25% faster when compared to our FPGA design. However, when we consider the energy spent running on both platforms, our FPGA solution consumes about 15% of the energy spent on CPU. For example, we have 17.02 J in FPGA and 108.40 J in CPU for a single 10k comparison.

TABLE 2 Table of resources used by the circuit in the FPGA

Resource	Used	Available	Used(%)
LUTS	26129	331680	7.88
Registers	42208	663360	6.36
Blockram	223	1080	20.65
DSPs	64	2760	2.32
IOB	53	520	10.19
IO	49	104	47.12

5.2 | Combined FPGA-CPU results

In this section, we used the same method for energy measurement explained in the beginning of Section 5.1 and added the scheduler to divide out the comparisons on the two platforms using the weighted round robin as described in Section 4.2. The weights used were 1 and 6, for the CPU and FPGA, respectively, meaning that the FPGA will compute 6x more comparisons than the CPU. As stated at the beginning of Section 5, the application consisted of 20 comparisons of each size (10k, 20k, and 50k).

The results are shown in Table 4. The second column presents the size of the comparisons. The third column presents the execution time in each platform (CPU or FPGA) whereas the forth column presents the total execution time (CPU+FPGA). The fifth and sixth columns present the power and energy for each platform (CPU or FPGA) and the seventh column shows the total energy of the heterogeneous solution (CPU+FPGA).

Analyzing Tables 3 and 4, we can see that the execution time of CPU+FPGA solution is lower than the CPU-only and FPGA-only solutions in the three sequence sizes (10k, 20k, and 50k). In terms of energy, the CPU+FPGA solution spends about 4x less energy than the CPU-only solution and about 2x more energy than the FPGA-only solution. For the sizes of sequences compared (10k, 20k, and 50k), we can see that the heterogeneous CPU+FPGA solution provides a very good tradeoff between execution time and energy consumption.

5.3 | Comparison to the CPU state of art

In this section, we compare our FPGA results to the state-of-the-art CPU multithreaded tool proposed by Shikder et al,[25] called CPU-OpenMP in this article. In CPU-OpenMP, there is a pre-processing phase to determine the maximum displacement in each row, with time complexity $O(n)$. Then, a modified LCS recurrence relation[25] that does not have dependencies on the same row is executed. In this phase, all the elements in the same row may be computed in parallel and, thus, processing is done row by row, instead of the traditional antidiagonal by antidiagonal.

Table 5 shows the results obtained with CPU+FPGA, FPGA, and CPU-OpenMP (4 cores) with the same DNA sequences (10k, 20k, and 50k) described in Section 5. For ease of comparison, the results from CPU+FPGA and FPGA were copied from Tables 3 and 4 to Table 5. To measure power consumption, the tool *powerstat*[24] was used once more.

Since the CPU-OpenMP is a highly optimized version of LCS that runs in multiple cores, it obtains very good speedups (from 3.7x to 3.9x), when compared to our FPGA versions (CPU+FPGA and FPGA). This comes at the expense of high energy consumption. The energy consumption of the CPU-OpenMP tool is much higher than the energy consumption of our FPGA version (about 2.5x higher). So, if energy consumption is a concern, the FPGA version should be used. The CPU+FPGA version provides an intermediate option, considering both execution time and energy consumption.

TABLE 3 Execution time, power, and energy results for 20 comparisons in the CPU-only and FPGA-only platforms

	Size	Time (s)	Power (W)	Energy (J)
CPU-sequential	10k	28.30	99.24	2808.49
FPGA-only	10k	32.09	10.69	343.04
CPU-sequential	20k	113.42	101.88	11, 555.23
FPGA-only	20k	128.35	10.69	1372.06
CPU-only	50k	706.35	101.96	72019.44
FPGA-only	50k	801.94	10.69	8572.74

TABLE 4 Execution time, power, and energy results for 20 comparisons in the CPU+FPGA platform

	Size	Time (s)	Total time(s)	Power (W)	Energy (J)	Total energy (J)
CPU	10k	4.28	27.28	98.43	421.28	712.9
FPGA		27.28		10.69	291.62	
CPU	20k	17.06	109.08	99.38	1695.42	2861.48
FPGA		109.08		10.69	1166.06	
CPU	50k	106.14	681.65	101.13	10,733.94	18,020.78
FPGA		681.65		10.69	7286.84	

TABLE 5 Comparison to the CPU-OpenMP tool (execution time and energy results)

	Size	Time (s)	Energy (J)
CPU+FPGA	10k	27.28	712.90
FPGA	10k	32.09	**343.04**
CPU-OpenMP (4 cores)[25]	10k	**7.21**	890.07
CPU+FPGA	20k	109.08	2861.48
FPGA	20k	128.35	**1372.06**
CPU-OpenMP (4 cores)[25]	20k	**29.33**	3478.92
CPU+FPGA	50k	681.65	18,020.78
FPGA	50k	801.94	**8572.74**
CPU-OpenMP (4 cores)[25]	50k	**171.32**	21569.19

5.4 | Case study: Covid-19

On 12 December 2019, a set of cases of severe acute respiratory syndrome caused by a newly identified coronavirus was announced in Wuhan, China. This coronavirus was initially named as the new coronavirus of 2019 (2019-nCoV) on 12 January 2020 by the World Health Organization (WHO). An epidemic of acute respiratory tract infection quickly spread, with WHO officially naming the disease as coronavirus disease 2019 (coronavirus disease 2019 - Covid-19), and the new coronavirus as the coronavirus severe acute respiratory syndrome virus 2 (severe acute respiratory syndrome coronavirus 2 - SARS-CoV-2). As an emerging acute respiratory infectious disease, SARS-CoV-2 spreads mainly through the respiratory tract, droplets, respiratory secretions, and direct contact.[26]

In this section, we perform a case study on the Covid-19 disease.

In order to do this case study, we compared 20 real SARS-CoV-2 sequences from several locations retrieved from NCBI (National Center for Biotechnology Information) at www.ncbi.nlm.nih.gov/sars-cov-2 to the reference SARS-CoV-2 sequence from Wuhan, China, retrieved from the same site. The sequences selected were of equal length (29,903 characters) and randomly selected from all five continents—at least one sequence per continent.

The accession numbers, name, length, and location of the sequences used in this test are listed in Table 6. The first sequence of the table is the reference sequence and all remaining sequences were compared to this one.

Table 7 presents the execution times and energy results for the Covid-19 study. These results are quite consistent with the ones presented in Tables 3 and 4, where the best execution times are obtained by the CPU-only version and the FPGA-only solution has the lowest energy consumption. The CPU-FPGA solution provides a good tradeoff between execution time and energy.

Table 8 shows, for each sequence compared to the reference sequence *NC_045512.2*, the accession number, the LCS score obtained by our tool, the country of the sequence, and the date that it was last updated in the public NCBI database. In the table, the results are ordered from the highest LCS score to the lowest one.

The first observation is that, as expected, all sequences are very similar to the reference sequence. Considering that all sequences have 29,903 characters and that the value assigned for matches is 1, the maximum LCS score is 29,903. This means that the similarity of sequences in Table 8 is in the 99.98%-99.94% range.

In such a scenario, our LCS tool can be used as a filter, helping to determine which sequences to analyze in more detail. Particularly, biologists would target the sequences that have the lower similarity with the reference sequence, in search of possible mutations. Our study indicates that sequences MT750353.1 and MT762396.1 are of special interest since they have the lowest score and are quite recent (last updated on July 14th and July 15th, respectively).

6 | CONCLUSION AND FUTURE WORK

This article proposed and evaluated an HLS solution in FPGA to compare biological sequences with the LCS algorithm and a heterogeneous CPU+FPGA strategy that uses a weighted scheduler and incorporates our HLS FPGA design.

Our FPGA solution (a) used *Block RAMs* to store the sequences, which are accessed through the FPGA DMA buses; (b) optimized the iterations of the calculation of the DP matrix using pipeline; and (c) reduced the cost of memory space using only three vectors to store the values and decreased the execution time by calculating diagonally. We also proposed a CPU+FPGA strategy which uses a weighted round-robin scheduler to distribute the LCS comparisons between the devices (CPU or FPGA).

TABLE 6 SARS-CoV-2 sequences

Accession	Description	Country
Asia		
NC_045512.2	*Severe acute resp syndrome coronavirus 2 isolate Wuhan-Hu-1*	China
MT135044.1	*Severe acute resp syndrome coronavirus 2 isolate SARS-CoV-2/human/CHN/235/2020*	China
MT374104.1	*Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/TWN/CGMH-CGU-08/2020*	Taiwan
MT415321.1	*Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/IND/GMCKN318/2020*	India
MT762396.1	*Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/BGD/BCSIR-NILMRC-254/2020*	Bangladesh
MT740381.1	*Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/BGD/BCSIR-NILMRC-145/2020*	Bangladesh
MT428552.1	*Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/KAZ/NCB-2/2020*	Kazakhstan
MT371047.1	*Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/LKA/COV38/2020*	Sri Lanka
MT755883.1	*Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/SAU/477/2020*	Saudi Arabia
Oceania		
MT459985.1	*Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/GUM/GU-NHG-01/2020*	Guam
Europe		
MT358638.1	*Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/DEU/FFM1/2020*	Germany
MT328032.1	*Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/GRC/10/2020*	Greece
MT511066.1	*Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/POL/PL-P10/2020*	Poland
America		
MT350282.1	*Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/BRA/SP02cc/2020*	Brazil
MT738101.1	*Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/BRA/HIAE-SP03/2020*	Brazil
MT466071.1	*Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/URY/Mdeo-1/2020*	Uruguay
MT470219.1	*Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/COL/Cali-01/2020*	Colombia
MT679205.1	*Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/USA/NYU-VC-017/2020*	USA
MT750353.1	*Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/USA/CA-CZB-1829/2020*	USA
Africa		
MT731285.1	*Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/MAR/RMPS01/2020*	Morocco
MT324062.1	*Severe acute respiratory syndrome coronavirus 2 isolate SARS-CoV-2/human/ZAF/R03006/2020*	South Africa

Note: All sequences have 29903 characters.

TABLE 7 Results from SARS-CoV-2 sequence comparisons

	Time (s)	Total time (s)	Power (W)	Energy (J)	Total energy (J)
CPU-only	230	230	100.52	23, 119.60	23, 119.60
FPGA-only	287	287	10.69	3068.03	3068.03
CPU	35	244	99.86	3495.10	6103.46
FPGA	244		10.69	2608.36	

The experimental results showed that the solution proposed in FPGA is capable of consuming much less energy than the solution in CPU. When comparing 20 pairs of 50k sequences, the energy used was 8572.74 J in FPGA while the same comparisons used 72,019.44 J in CPU. With that, note that the CPU consumes 8.4x more energy to perform the application compared to the FPGA. Therefore, with the FPGA-only solution, energy consumption was significantly lower at FPGA. It is also noted that the execution time in FPGA increases about 1.13x in relation to the CPU.

We also showed that our heterogeneous FPGA+CPU solution is able to have the best execution time for the sizes of sequence considered (10k, 20k, and 50k), when compared to the FPGA-only and CPU-only solutions. If we compare the FPGA+CPU solution with the FPGA-only solution, we observe a reduction of about 4x in the execution time and an increase of about 2x in energy consumption. If the comparison is done with the CPU-only solution, the decrease in the execution time is about 1.13x and the reduction in energy consumption is about 4x.

TABLE 8 Scores of SARS-CoV-2 sequences compared to the NC_045512.2 reference sequence from China uploaded in the public NCBI database at 31-DEC-2019 with last modification date 30-MAR-2020

Accession	LCS score	Country	Date of update
MT135044.1	29,899	China	06-APR-2020
MT415321.1	29,899	India	30-APR-2020
MT466071.1	29,899	Uruguay	16-MAY-2020
MT324062.1	29,897	South Africa	13-APR-2020
MT350282.1	29,897	Brazil	17-APR-2020
MT358638.1	29,897	Germany	14-MAY-2020
MT428552.1	29,897	Kazakhstan	05-MAY-2020
MT374104.1	29,896	Taiwan	24-APR-2020
MT470219.1	29,896	Colombia	14-MAY-2020
MT679205.1	29,896	USA	02-JUL-2020
MT731285.1	29,896	Morocco	08-JUL-2020
MT459985.1	29,895	Guam	13-MAY-2020
MT328032.1	29,894	Greece	13-APR-2020
MT371047.1	29,893	Sri Lanka	23-APR-2020
MT511066.1	29,892	Poland	26-MAY-2020
MT738101.1	29,891	Brazil	09-JUL-2020
MT740381.1	29,891	Bangladesh	10-JUL-2020
MT755883.1	29,891	Saudi Arabia	14-JUL-2020
MT750353.1	29,889	USA	14-JUL-2020
MT762396.1	29,886	Bangladesh	15-JUL-2020

Note: Higher scores mean more similarity.

There are still improvements to be made in the FPGA implementation of the LCS algorithm both in the part of the *host* and in the *design* of the HLS, such adding an asynchronous transfer strategy between the *host* and the FPGA. There is still room for optimizations in the HLS code, especially regarding the use of *AXI-Stream Interface* to increase the throughput of the LCS input/output ports.

We also intend to make a complementary study to determine the impact of the resource consumption of the board, as well as the circuit generated by the tool, increasing the size of the sequences to be compared in the FPGA. Preliminary results indicate that it is possible to fit two LCS blocks in the FPGA, so that two pairs of 50k sequences can be processed in parallel, with almost no impact to the overall energy consumption.

It can be noted as well, that using an heterogeneous system can improve both execution time and energy consumption. However, this approach still poses a number of challenges especially on the software side, in which there is still room for improvement. As future work, we intend to further investigate which characteristics of the application and the platform have most impact in execution time and energy consumption and how they should be used in the definition of the weights assigned to each platform.

Finally, we intend to implement other scheduler strategies such as work stealing and dynamic workload distribution and compare them to our weighted scheduler.

ACKNOWLEDGMENTS

This research is funded by the Capes/PROCAD 183794 project, the CNPq 426729/2018-8 project, and the FAPDF 00193-00002139/2018-79 project. Last but not least, the authors would like to thank Xilinx University Program for the donation of the licenses that allowed the development of this work.

ORCID

Carlos A. C. Jorge https://orcid.org/0000-0003-3842-3263
Alba C. M. A. Melo https://orcid.org/0000-0001-5191-5209
Alfredo Goldman https://orcid.org/0000-0001-5746-4154

REFERENCES

1. Bucak IO, Uslan V. Sequence alignment from the perspective of stochastic optimization: a survey. *Turk J Electr Eng Comput Sci*. 2011;19:157-173.
2. Mount David W. *Bioinformatics: sequence and genome analysis*. 1. 2. Cold Spring Harbor, N.Y: Cold Spring Harbor Laboratory Press; 2001.

3. Arslan AN. Sequence alignment. Telefoncu A, Sahin F, Kilinc A, *Biyoinformatik-II*. Bioinformatics Graduate Summer School II 01. 2004:101–114. ISBN = 975-483-637-X.
4. Wagner RA, Fischer MJ. The string-to-string correction problem. *J ACM*. 1974;21(1):168-173. https://doi.org/10.1145/321796.321811.
5. Smith TF, Waterman MS. Identification of common molecular subsequences. *Journal of Molecular Biology*. 1981;147(1):195-197. https://doi.org/10.1016/0022-2836(81)90087-5.
6. Needleman Saul B, Wunsch Christian D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*. 1970.48(3):443–453. https://doi.org/10.1016/0022-2836(70)90057-4.
7. Hirschberg DS. A linear space algorithm for computing maximal common subsequences. *Commun ACM*. 1975;18(6):341-343. https://doi.org/10.1145/360825.360861.
8. Xilinx Corporation. UltraFast High-Level Productivity Design Methodology Guide. 2019.
9. Jorge CAC, Nery A, Melo ACMA. Uma implementacao do algoritmo LCS em FPGA usando high-level synthesis. Paper presented at: Simposio de Sistemas Computacionais de Alto Desempenho (WSCAD); 2019:1-8.
10. Papadimitriou C, Ullman J. A communication-time tradeoff. *SIAM J Comput*. 1987;16(4):639-646.
11. Bampis E, Delorme C, Konig J. Optimal schedules for d-D grid graphs with communication delays. *Parallel Comput*. 1998;24:1653-1664.
12. Daoudi EM, Trystram D, Wagner F. Scheduling 2-dimensional grids with large communication delays. *RAIRO-Oper Res*. 2015;49:369-381.
13. D'Amore L, Mele V, Romano D, Laccetti G. A multilevel approach for the performance analysis of parallel algorithms. *Comput Inform*. 2019;38:817-850.
14. Mele V, Romano D, Constantinescu E, Carracciuolo L, D'amore L. Performance evaluation for a PETSc parallel-in-time solver based on the MGRIT algorithm. In: Springer International Publishing; 2018:716-728.
15. D'amore L, Mele V, Laccetti G, Murli A. Mathematical approach to the performance evaluation of matrix multiply algorithm. In: Springer International Publishing; 2016:25-34.
16. Al Junid S, Haron A, Majid AZ, et al. Optimization of DNA sequences data for accelerate DNA sequences alignment on FPGA. Paper presented at: 2010 Fourth Asia International Conference on Mathematical/Analytical Modelling and Computer Simulation, Bornea; 2010:231-236. https://doi.org/10.1109/AMS.2010.54.
17. Chen H, Cheng F, Hu S. A reconfigurable embedded system for sequence alignment problem. Paper presented at: 2011 International Conference on Machine Learning and Cybernetics, Guilin; 2011;3:1345-1351. https://doi.org/10.1109/ICMLC.2011.6016879.
18. Mousavi SR, Tabataba F. An improved algorithm for the longest common subsequence problem. *Comput Oper Res*. 2012;39(3):512-520. https://doi.org/10.1016/j.cor.2011.02.026.
19. Blum C, Blesa MJ, López-Ibáñez M. Beam search for the longest common subsequence problem. *Comput Oper Res*. 2009;36(12):3178-3186. https://doi.org/10.1016/j.cor.2009.02.005.
20. Ozsoy A, Chauhan A, Swany M. Achieving TeraCUPS on longest common subsequence problem using GPGPUs. Paper presented at: Proceedings of the 2013 International Conference on Parallel and Distributed Systems, Seoul; 2013:69-77. https://doi.org/10.1109/ICPADS.2013.22.
21. Cinti A, Bianchi FM, Martino A, Rizzi A. A novel algorithm for online in exact string matching and its FPGA implementation. *Cognitive Computation* . 2018;12 369–387. https://doi.org/10.1007/s12559-019-09646-y.
22. Alser M, Hassan H, Kumar A, Mutlu O, Alkan C. Shouji: a fast and efficient pre-alignment filter for sequence alignment. *Bioinformatics*. 2019;35(21):4255–4263. https://doi.org/10.1093/bioinformatics/btz234.
23. Xilinx. Vivado High-Level Synthesis. *Xilinx* 2016. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.
24. Canonical. Ubuntu Manpage: powerstat - a tool to measure power consumption. *Ubuntu* 2015. http://manpages.ubuntu.com/manpages/xenial/man8/powerstat.8.html.
25. Shikder R, Thulasiraman P, Irani P, Hu P. An OpenMP-based tool for finding longest common subsequence in bioinformatics. *BMC Research Notes*. 2019;22(1):220:1–220:6. https://doi.org/10.1186/s13104-019-4256-6.
26. Guo YR, Cao QD, Hong ZS, et al. The origin, transmission and clinical therapies on coronavirus disease 2019 (COVID-19) outbreak – an update on the status. *Military Medical Research*. 2020;7(1):11:1–11:10. https://doi.org/10.1186/s40779-020-00240-0.

How to cite this article: Jorge CAC, Nery AS, Melo ACMA, Goldman A. A CPU-FPGA heterogeneous approach for biological sequence comparison using high-level synthesis. *Concurrency Computat Pract Exper*. 2020;e6007. https://doi.org/10.1002/cpe.6007