Accelerating Charged Single α-helix Detection on FPGA

Sam Khozama, Zoltán Nagy and Zoltán Gáspári

Pázmány Péter Catholic University
Faculty of Information Technology and Bionics
Budapest, Hungary
Email: {khozama.sam, nagy.zoltan, gaspari.zoltan}@itk.ppke.hu

***Abstract*—Novel biological sequences are determined at an extreme pace producing a huge amount of data each day. Implementing and speeding up the bioinformatics algorithms, which need very fast and accurate results, is the main advantage of reconfigurable architectures like Field-Programmable Gate Array (FPGA), in addition to the low precision input data for these kinds of algorithms, which can be stored in 2-5 bits. Detecting structural motifs such as Charged Single α-Helixes (CSAH) is a computationally intensive task which can be accelerated by using FPGA. The goal of our research is to further improve the processing speed of our algorithm called FT_CHARGE to analyze large databases in a reasonable time. Using the largest state-of-the-art FPGA devices, either the number of processing units or the parallelism inside the processing units can be increased. In this paper, we provide details and compare the two design approaches in terms of speed, implementation and accuracy. We propose a new architecture that can perform search for CSAH 32 times faster compared to our previous FPGA implementation.**

Keywords–FPGA; FT_CHARGE; CSAH; Charged Single α-helix; Hardware acceleration.

I. INTRODUCTION

The Charged Single Alpha-Helix (CSAH or simply SAH) is a unique structural motif in proteins [1]. It has been experimentally characterized only in a small number of proteins, therefore, its recognition by prediction methods is of high importance. Generally, more accurate algorithms tend to be slow, thus, efforts are concentrated to speed up these methods to make them applicable to large sequence sets [2]. Here, we present the speedup of FT_CHARGE, one of the earliest methods based on Fourier transformation. We have previously implemented an FPGA-based version of this algorithm [3] ,which is further improved here. Our novel implementation offers higher computational speed to allow processing of very large protein sequence sets, such as full genomes, or even metagenomic samples, within hours.

The goal of using FPGA is to accelerate different sequence searches and sequence matching in bioinformatics algorithms such as the Smith-Waterman (SW) algorithm. The parallel nature of FPGA, with its huge number of fine-grained blocks (configurable logic blocks) provides a convenient architecture for the additional implementation of bioinformatics algorithm; this way, FPGA and the bioinformatics algorithms have parallelism at a fundamental level [4]. So, a large amount of small and simple functional units can be implemented [5]. The Arithmetic Logic Unit (ALU), which is responsible for performing instructions in conventional computers, which is limited by the fact that it can only perform one instruction at a time [6]. Our task is to implement in parallel any function or circuit that meets the requirements of an application rather than sequentially, to achieve optimal performance.

II. FT_CHARGE ALGORITHM

Both SCAN4CSAH [1] and FT_CHARGE are used to detect CSAHs motifs and these two algorithms use completely different computational methods for analyzing protein sequences conceptually. Because the FT CHARGE algorithm is a very computationally intensive algorithm, it is suitable to accelerate it on FPGA. The standard FAST-All (FASTA) format is the input format for both algorithms. The analysis of biopolymer sequences utilizes Fourier transformation regularly. To implement the FT_CHARGE algorithm, we downloaded the database to the host computer and pre-processed it by using only 2-bit encoding per sequence element. Charges are assigned as follows: -1 for Asp (Aspartic acid) and Glu (Glutamic acid) , +0.5 for His (Histidine), +1 for Arg (Arginine) and Lys (Lysine) and zero for any other amino acid residue.[1]. This encoding is done by the host computer. The host computer sends this encoded data to the FPGA, where it is processed, and we receive back only the filtered results. Consequently, only the candidate sequences expected to have α-helix will be sent back to the host computer. As shown in Figure 1, the four consecutive steps of the algorithm are (1) the Charge Correlation Computation, (2) the Fast Fourier Transform (FFT) Computation, (3) the Maximum Finder and (4) Extreme Value Distribution Computation. The Charge Correlation function is defined by the following function:

$$R(k,n) = \sum_{i=k}^{k+m-n} c(i)c(i+n) \tag{1}$$

where $c(i)$ is the charge assigned to the i^{th} amino acid, m is the length of the window, $1 \leq k \leq l-m$ is the starting position of the current window and l is the length of the sequence. In our case, windows of 32 or 64 elements are examined during the analysis of the full sequences.

The output of the Charge Correlation Calculation block is connected to the FFT block. The role of the Maximum Finder block is to find the maximum amplitude and its frequency in the FFT spectra, which are used to fit an Extreme Value Distribution (EVD). At last, determining the threshold for signaling a charged single α-helix motif is done by using the fitted distribution.

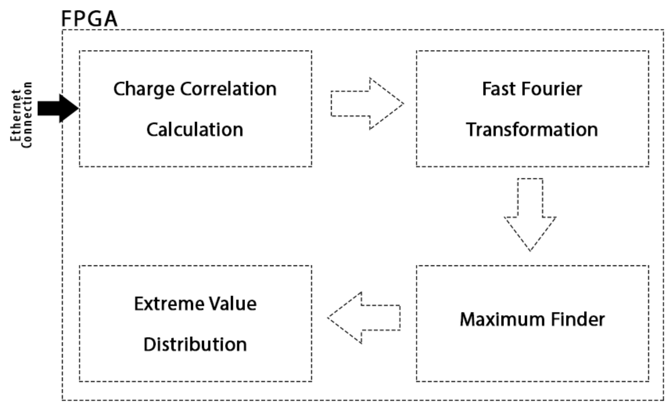
FPGA
Ethernet Connection
Charge Correlation Calculation
Fast Fourier Transformation
Extreme Value Distribution
Maximum Finder

Figure 1. Block diagram of the system implemented on FPGA.

TABLE I. AREA REQUIREMENTS OF OUR PREVIOUSLY IMPLEMENTED SYSTEM [3]

	CLB LUTs	CLB Flip-Flops	BRAM	DSP Slices
Complete system	44662	53122	90	157
ZCU102	274080	548160	912	2520
Estimation	6	10	10	16

III. FPGA IMPLEMENTATION

FPGA manufacturers offer very efficient signal processing libraries to utilize in highly-intensive computation applications. In our previous work [4], we implemented the FT_CHARGE algorithm on a small FPGA. It used the FFT IP core from Xilinx CoreGenerator and the main goal was optimizing the computation of the charge correlation function to feed the FFT core efficiently. The computation time for the whole UniProt TREMBLE database [7] took nearly 24 hours. The stages were computed in parallel and the system was running on 100 MHz clock-frequency. The 32 and 64 element version required a new 2bit wide information sequence element in every 32^{nd} and 64^{th} clock-cycle. The memory bandwidth requirement was 6.25 Mbit/s and 3.125 Mbit/s in the case of 32 and 64 element windows, respectively.

The first way to improve our previous implementation is simply to replicate the system-blocks as much as we can, to fit into a larger ZYNQ device [8]. The FPGA resource usage of our previously implemented system and the estimated number of processing blocks on Xilinx ZCU102 board equipped with a larger FPGA compared to our previous study are shown in Table I.

In this paper, we have the opportunity to increase the performance nearly 6 times, as the limiting factor is the number of Lookup tables (LUTs). In light of this, we should take into consideration what does it mean to have such a huge number of hardware blocks on FPGA. Our previous computing unit connected to the main memory of the board via 6 input and 3 output Advanced Extensible Interfaces (AXI) [9]. These 9 interfaces should be replicated 6 times here. We can say that the area required by the AXI interconnect blocks to connect all the 54 AXI interfaces of these 6 units on the ZYNQ board to the memory interface will use a significant portion of the device. Thus, we could accelerate the implementation, but it would still require a lot of time because of a larger database.

A more efficient way to extend our previous solution is to implement several units in parallel, using a larger Xilinx ZYNQ board [10] [11]. Therefore, we went further to utilize some of the recent advantages of the High-Level Synthesis (HLS). Our suggested solution is to change the output of the Charge Correlation block in order to compute and send all 32 or 64 elements to FFT in each clock-cycle and also to replace the previous FFT module with more parallel processing units. This will definitely help us to improve computing performance in the system.

We proposed a new FFT implementation based on the Cooley-Tukey algorithm [12] that accepts 32 or 64 elements in each clock-cycle. The multiplication processes inside the FFT butterfly diagram normally use floating-point numbers, but this requires a very large area. In addition, we decreased the number of bits and saved a large amount of space because of using fixed-point numbers. The accuracy of the new solution is almost the same with the previous one. The basic idea is to improve the speed of computation by parallelizing the algorithm.

The system has implemented on the ZCU102 board. There are 2 options when preparing the input-output interfaces: (1) either load the input sequence by sequence or (2) load many sequences into a large buffer. Before processing each sequence, we need to load some special parameters to the control registers of the FT_CHARGE processing block. These parameters include the address of the sequence, the buffer where the result should be saved, and the length of the sequence. Sending these parameters take time since they should be written through the relatively slow AXI lite interface. According to the co-simulation results, sending one parameter takes around 5-6 clock-cycles, so, setting all the needed parameters will require a long time (in case of short sequences, it will be comparable with computation time of the whole sequence).

Here, our improvement consists of working on larger buffers, where one buffer contains several sequences concatenated one after each other and, instead of giving the length of one sequence, we give an array of lengths to the FT_CHARGE algorithm. For example, if we have 4 MB buffer, we can place several sequences in this large buffer and the cumulative length of these sequences is in the order of 16 million. In 1 byte, we can store 4 sequence elements because 2 bits can encode 1 element from the sequence. In this case, we have a task, which may last for 16 million clock cycles and, when we start the operation of this block, we need nearly 100 clock-cycles to set up the system with the required parameters. After that initial delay, the processing unit can work for a long time, consequently, the utilization of the hardware will be greatly increased.

IV. FAST FOURIER TRANSFORMATION BLOCK

For efficient computation of the spectra of the charge correlated window, the FFT can be used. A well known computing method of the FFT is the Cooley-Tukey Algorithm [12]. The Discrete Fourier Transform (DFT) of a signal is defined as follows:

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi i}{N} nk} \tag{2}$$

where x_n is an element of the input vector and k is an integer ranging from 0 to $N-1$, where N is the size of the transform.

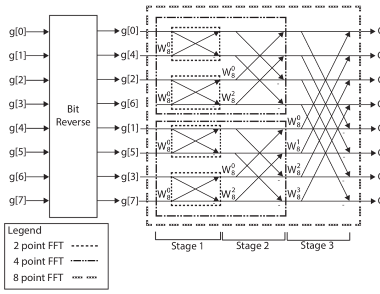

Figure 2. Cooley-Tukey algorithm: An 8 point FFT built recursively.

We can rewrite the computation of X_k as:

$$X_k = E_k + e^{\frac{-2\pi i}{N}k} O_k \tag{3}$$

$$X_{k+N/2} = E_k - e^{\frac{-2\pi i}{N}k} O_k \tag{4}$$

A block used to compute these equations is called butterfly, which is a small DFT. The Butterflies can be connected systematically to build larger FFT blocks, as shown in Figure 2 for an 8 input FFT. The number of stages is $log_2 N$ and $N/2$ Butterflies are required in each stage. As a result, the number of operations during the Fourier Transformation computation is reduced to $\mathcal{O}(n \log n)$ from $\mathcal{O}(n^2)$. Also, this structure provides a convenient way to implement parallel FFT by using $N \times log_2 N$ Butterflies.

Our previously implemented system used Xilinx FFT IP, which does the transformation serially. So, 32 or 64 clock-cycles are required to load the samples. The whole computation is done in 32 or 64 clock-cycles. The previous system has one processing the pipeline for 32 elements window and two processing pipelines for the 64 elements window. Therefore, the time required for 32 and 64 element window computations is roughly the same, if we have a large group of sequences.

In this paper, we modified the charge correlation part to provide these data elements in parallel, which means computation speed could be increased 32 times. So, by implementing FFT and Charge Correlation computation in parallel 32 times, a speedup can be expected because all the 32 or 64 outputs of the Charge Correlation block can be computed in one clock-cycle and shifted to the FFT. After some delay, the FFT block will provide all the 32 or 64 transformed results in one clock-cycle. The Maximum Finder should also compute the maximum value of the 32 or 64 element window in one clock cycle. The Extreme Value distribution block is not modified because it works on the maximum value of the FFT spectra and its position, just like in the previous implementation.

Our system's expectations are that each pipeline will require one input stream and one output stream. Therefore, 4 AXI interfaces are required all together. If these blocks are replicated 2 or 3 times, we still need only 8 or 12 memory ports, which is more manageable than the 54 ports of the previous serial implementation.

For easier testing, we use the MATLAB code and C code from the previous implementation. In this work, create a complete test bench in Vivado HLS [10] to load a valid sequence data to the system, process it and check whether the result is good or not.

TABLE II. AREA REQUIREMENTS OF THE CURRENT SYSTEM WITH DIFFERENT NUMBER REPRESENTATIONS

	CLB LUTs	CLB Flip-Flops	CLBs	Block RAM	DSPs
ZCU102	274 080	548 160	599 550	912	2 520
Fixed-point 18bit	44 043	42 344	8 155	13.5	668
Fixed-point 23bit	51 572	50 378	9 417	13.5	768
Floating-point	225 157	321 469	34 092	13.5	2 146

TABLE III. RUNTIME PROCESSING OF THE SPROT DATABASE (SECONDS), FPGA AND SOFTWARE IMPLEMENTATIONS

	with communication	without communication
Previous System	275.857	-
Current System		
Fixed-point	20.218	2.24159
Floating-point	23.1604	5.15008
AMD Ryzen 5 3400G	223.52	-
INTEL Core i5-4590	371.731	-

Table II shows three different representations that could be used in the current system. The original used an 18 bit word length inside FFT. The second one used 23 and 24 bits in the cases of 32 and 64 element windows. These designs were running on a 250MHz clock-frequency and were processing one window in each clock-cycle. The last one was a single-precision floating-point version.

By choosing the fixed-point 18 bit representation, the limiting factor will be the number of Digital Signal Processor (DSP) slices. In this case only, 30% of the resources are used and we can replicate the system-blocks three times.

Using single-precision floating-point numbers requires 6 times more DSP slices, because one floating-point multiply add (MADD) unit requires 4 DSP slices for the multiplier and 2 DSP slices for the adder, compared to a single DSP slice in the 18 and 23 bit cases. To fit the floating-point version into the ZCU102 board, the Initiation Interval of the FFT block is increased to two. In this case, the FFT is computed in two clock-cycles and the floating-point multiply-add units are shared between two operations, effectively halving the DSP slice requirement of the circuit. The FPGA resource utilization in this case, is over 90%, which might cause timing issues during implementation. Therefore, the clock-frequency of the design is reduced to 150MHz for stable operation. With the floating-point representation, the more accurate results, the more DSP slices are needed. The computation speed of the floating point solution is reduced to 75 million windows/s due to the slower 150MHz clock frequency and the two clock cycle processing time of the FFT. The computation times of the different solutions for the UniProt SPROT database [7] are summarized in Table III.

V. SYSTEM TESTING

After finished testing all the methods, which represented the functionality of the system in a synthesizable form, we started preparing the Xilinx environment to execute our implementation on a real circuit. On the host PC, we started preprocessing the database of amino acid sequences. The data is downloaded from the UniProt website [7] in FASTA format and converted to charges. We are using a ZCU102 board

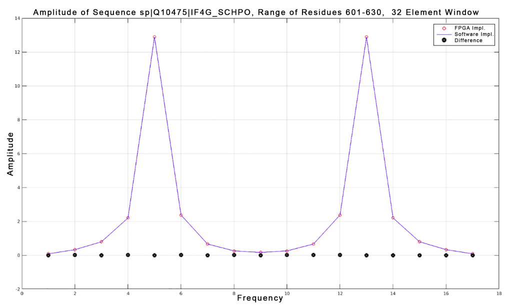

Figure 3. Amplitude of FFT, 32 elements window computation.

from Xilinx, which, in turn, has ARM Central Processing Unit cores. We are utilizing them for running Linux to handle the communication between the host computer and the ZCU102 board over gigabit Ethernet. After accepting the data, the ARM cores are sending it to the FPGA with Direct Memory Access (DMA). Our reference solutions for the FT_CHARGE algorithm are computed using MATLAB and C++. The result of these two were compared to the FPGA output. The comparison procedure included the number of hits, which represent the candidates windows expected to contain α-helix. Vivado Software Development Kit (SDK) was used for developing this software. While the circuit was running, we experienced a few cases where the sequences were detected on the software side but not in the hardware one. Examination of the undetected sequences showed two types of errors; in case of the first error type, the amplitude was close to the amplitude threshold (7.0). Some elements were not found because the maximum amplitude computed on the FPGA was smaller than the threshold due to the rounding error of the fixed-point computation. The other error has been noticed after plotting the FFT amplitudes in MATLAB. The representation of the amplitude in case of the 32 element window is plotted in Figure 3. A similar behaviour can be observed in the case of some 64 element windows too.

The figure shows that the difference can not be determined by the human eye, because the results of the software and the FPGA computation are correct to at least three or four decimal digits. The question raised here is: why are these sequences recognized only in software?

After examining the numbers in greater depth, we determined that we have two nearly equal peaks for half of the sequences in the software. In the hardware, we have a difference in the third or fourth decimal value, as mentioned before and, because of the rounding error of this very tinny difference, the output of the hardware is changed and no longer in sync with the software. We can easily overcome this problem by increasing the width of the registers storing the partial results in the FFT block by adding more fractional bits. Unfortunately this solution requires more resources because the multipliers in the DSP slices are working on 25 bit and and 18 bit signed inputs. Therefore two DSP slices required, instead of one, when on one of the inputs of the multiplication is in the $(26 - 49)$ bit range.

The consequences of the differences in numerical precision are to be investigated on large biological sequence sets to determine whether they cause any negative consequences for SAH detection. The double peaks, observed for some sequences, likely come from regular larger repeating units in the protein sequences. This issue and its relevance are also under investigation.

VI. Conclusion

In this paper, we improved the previously proposed FPGA based system for speeding up the α-helix detection algorithm by replicating the main three blocks as much as possible to fit a larger FPGA board. In addition, implementing these processing units in parallel enables fast search on larger protein databases and runs the whole system at a speed 30 times higher than the previous implementation. We have finished modifying the code of the whole system to be fitted with FPGA specifications represented in the 2-bit representation for the Charge Correlation Calculation module and also the transition from floating-point to fixed-point during the FFT module. This is required to use FPGA resources more efficiently.

On one hand, we were able to significantly reduce the area required to implement the circuit on FPGA by using fixed-point representation inside the FFT module. On the other hand computing performance is increased by a factor of three, while the accuracy of the results is similar to the accuracy of the floating-point solution. We have also tested this code, implemented on FPGA with real sequence data, and we obtained the same results as with the previous version on MATLAB. So, the error is in an acceptable range and the hardware version is working properly.

References

[1] D. Süveges, Z. Gáspári, G. Tóth, and L. Nyitray, "Charged single α-helix: a versatile protein structural motif," Proteins, vol. 74, 2009, pp. 905–916, doi: 10.1002/prot.22183.

[2] D. Simm and M. Kollmar, "A command-line tool for predicting stable single α-helices (SAH-domains), and the SAH-domain distribution across eukaryotes." PLoS one, vol. 13, no. 2, 2018, p. e0191924, doi: 10.1371/journal.pone.0191924.

[3] Á. Kovács, D. Dudola, L. Nyitray, G. Tóth, Z. Nagy, and Z. Gáspári, "Detection of single α-helices in large protein sequence sets using hardware acceleration," Journal of structural biology, vol. 204, no. 1, 2018, pp. 109–116, doi: 10.1016/j.jsb.2018.06.005.

[4] Z. Nagy, Z. Gáspári, and A. Kovács, "Accelerating a charged single α-helix search algorithm in protein sequences using FPGA," in CNNA 2016; 15th International Workshop on Cellular Nanoscale Networks and their Applications. VDE, 2016, pp. 1–2.

[5] D. G. Bailey, Design for embedded image processing on FPGAs. John Wiley & Sons, 2011.

[6] D. Abramson, A. de Silva, M. Randall, and A. Posutla, "Parallel special purpose architectures for high speed optimisation," in Proceedings of the Second Australasian Conference on Parallel and Real Time Systems, 1995, pp. 13–20.

[7] "Uniprot," URL: https://www.uniprot.org/.

[8] F. Albu et al., "Implementation of (Normalised) RLS lattice on Virtex," in International Conference on Field Programmable Logic and Applications. Springer, 2001, pp. 91–100, doi: 10.1007/3-540-44687-7_10.

[9] Xilinx, AXI4-Stream, Infrastructure IP Suite v3.0, LogiCORE IP Product Guide, Xilinx, December 2018.

[10] ——, Introduction to FPGA Design with Vivado High-Level Synthesis, UG998, Xilinx, Jan. 2019.

[11] ——, ZCU102 Evaluation Board User Guide UG1182 (v1.6), Xilinx, June 2019.

[12] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," Math. of Computation, vol. 19, 1965, pp. 297–301, doi.org/10.1090/S0025-5718-1965-0178586-1.