


# Online Signature Verification Systems on a Low-Cost FPGA

Enrique Cantó Navarro <sup>1,\*</sup>, Rafael Ramos Lara <sup>2</sup>  and Mariano López García <sup>2</sup>

<sup>1</sup> Electrical, Electronic and Automation Engineering Department, Universitat Rovira i Virgili, Avda. Països Catalans 26, 43007 Tarragona, Spain

<sup>2</sup> Electronics Engineering Department, Universitat Politècnica de Catalunya, Avda. Victor Balaguer 1, 08800 Vilanova i la Geltrú, Spain; rafa.ramos@upc.edu (R.R.L.); mariano.lopez@upc.edu (M.L.G.)

\* Correspondence: enrique.canto@urv.cat

**Abstract:** This paper describes three different approaches for the implementation of an online signature verification system on a low-cost FPGA. The system is based on an algorithm, which operates on real numbers using the double-precision floating-point IEEE 754 format. The double-precision computations are replaced by simpler formats, without affecting the biometrics performance, in order to permit efficient implementations on low-cost FPGA families. The first approach is an embedded system based on MicroBlaze, a 32-bit soft-core microprocessor designed for Xilinx FPGAs, which can be configured by including a single-precision floating-point unit (FPU). The second implementation attaches a hardware accelerator to the embedded system to reduce the execution time on floating-point vectors. The last approach is a custom computing system, which is built from a large set of arithmetic circuits that replace the floating-point data with a more efficient representation based on fixed-point format. The latter system provides a very high runtime acceleration factor at the expense of using a large number of FPGA resources, a complex development cycle and no flexibility since it cannot be adapted to other biometric algorithms. By contrast, the first system provides just the opposite features, while the second approach is a mixed solution between both of them. The experimental results show that both the hardware accelerator and the custom computing system reduce the execution time by a factor  $\times 7.6$  and  $\times 201$  but increase the logic FPGA resources by a factor  $\times 2.3$  and  $\times 5.2$ , respectively, in comparison with the MicroBlaze embedded system.

**Keywords:** online signature; FPGA; biometrics verification; DTW; hardware accelerator; fixed-point



**Citation:** Cantó Navarro, E.; Ramos Lara, R.; López García, M. Online Signature Verification Systems on a Low-Cost FPGA. *Appl. Sci.* **2022**, *12*, 378. <https://doi.org/10.3390/app12010378>

Academic Editor: Andrew Teoh  
Beng Jin

Received: 16 November 2021

Accepted: 29 December 2021

Published: 31 December 2021

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:** © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

A handwritten signature requires a conscious action by the user, and therefore, it is one of the most usual methods by which persons claim their identity verification and accept responsibility for a signed document. Nonetheless, handwritten signatures have certain disadvantages, which have hindered their widespread use as biometric verification. The main challenge is the large intra-class and small inter-class variability. Samples taken from the genuine user may have large variability in their shapes. In addition, trained forgery signatures carried out by impostors may look very similar to signatures from the genuine user. Online signatures provide advantages for automatic biometric verification due to the additional information obtained from the capturing device, an electronic pen on a touch screen. Capturing angle and pressure samples, along with the shape coordinates, provides additional information about signature dynamics that increases inter-class variability.

Biometric systems are typically developed on general-purpose microprocessors due to the development comfortability provided by programming languages and compilers. Implementation issues, such as power consumption, data format, processing speed, or memory requirements, are not generally considered since they are assumed to run on high-performance computers. In contrast, portable systems are generally based on low-cost embedded microprocessors featured by memory and computing limitations. In such cases,

algorithm optimizations, hardware accelerators or custom computers can be explored to provide an efficient implementation.

Biometric algorithms are usually composed of a set of computing stages that are used not only in different biometric modalities but also in computer vision, cryptography, and other fields. The computational requirements at some stages may hinder their applicability for real-time systems on general-purpose microprocessors. In such cases, the acceleration of some calculations by using FPGAs is an interesting possibility. Most of the works related to accelerators on FPGAs focus on a particular computing stage, such as the simplified FFT (Fast Fourier Transform) presented in [1], SVM (Support Vector Machine) implementations in [2] or the three-dimensional DTW (Dynamic Time Warping) architecture shown in [3]. However, complete biometric systems accelerated on FPGAs are scarce due to the development effort that must be carried out for their implementation. The work presented in [4] describes a pyramid-pipeline architecture of a convolutional neural network on FPGA for speaker recognition, and a complete biometric speaker verification system was presented in [5]. Other examples are the hardware accelerator of cryptosystem for iris, presented in [6], and the iris identification system on FPGA based on hardware-software co-design [7]. Related work about online signature biometrics on FPGA is even more scarce. The FPGA role described in [8] is to act as a wireless slave, which is connected to a computer executing a very simple signature algorithm. The online signature verification system proposed in [9] is based on a hardware accelerator, which has been deprecated by the second system presented in this paper.

The paper is organized into six sections. The next section of the paper briefly describes the online signature verification algorithm, which is implemented in the presented systems. Section 3 starts describing the adopted replacements of the IEEE 754 double-precision format in order to efficiently implement the algorithm on a low-cost FPGA family. The section continues with an architecture overview of the three implemented systems, their main advantages and drawbacks. Section 4 shows a detailed explanation about the DTW stage implemented on these systems, not only because it vastly dominates the total execution time of the signature verification but also because it is widely adopted in other biometric modalities, such as speaker [10], electrocardiogram [11], iris [12], and many others. The experimental results obtained on the three systems are reported in Section 5. Finally, discussions are presented in the last section.

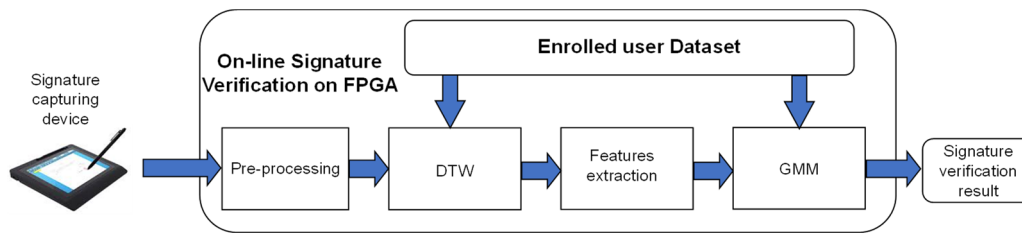
## 2. Signature Verification Algorithm

Signature biometric systems are usually based on two phases: enrollment and verification. At the enrollment phase, the system is trained for a genuine user who provides several signatures to extract a set of distinguishing features, which is stored as the user dataset. During the verification phase, a new signature is presented, and its features are extracted and matched against the previously stored one. Then, by computing a similarity score that is compared with a threshold value, the signature is classified as genuine or counterfeit.

Most online signature verification techniques can be classified as template, statistical or structural matching [13,14]. Template matching is a straightforward comparison of samples from the genuine user and the presented signature, which are usually pre-aligned by a DTW algorithm [15]. In statistical matching, the most salient behavioral signature features are compared and scored by means of a neural network, a hidden Markov model or a GMM (Gaussian Mixture Models). Structural matching is based on syntactic approaches for representing signatures that are compared through graph or tree techniques.

The signature verification algorithm consists of four stages: pre-processing, DTW, extraction of features and GMM matching, as depicted in Figure 1. The initial pre-processing applies a set of transformations on the captured signature, as resampling, centering and rotating the shape, and normalization. The resulting data are a set of 256-width vectors containing normalized samples. Then, a DTW algorithm computes the optimal alignment between the time-dependent samples of the processed signature and the enrolled template.

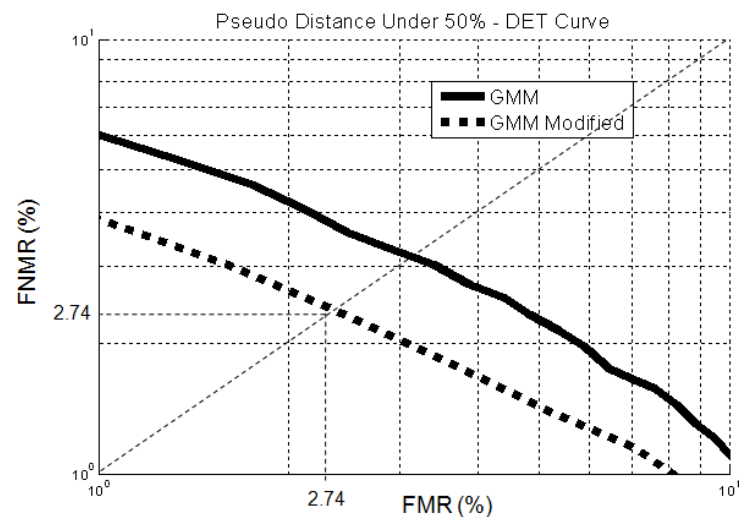
Finally, a set of statistical features are extracted from the aligned signature, known as pseudo-distances, to execute a GMM model that calculates a distance score.



**Figure 1.** Overview of the online signature verification algorithm.

The dataset of an enrolled user on the implemented online signature system [16] is composed of a signature template and its GMM parameters. The signature template consists of 256-width vectors containing normalized samples (x-y coordinates, pressure, angles). A set of distinctive features from the signature are selected to generate a GMM model. The parameters of the optimal GMM model for each enrolled user are computed by the expectation-maximization (EM) algorithm [17] with a public signature database [18]. The database contains 25 genuine signatures and 25 skilled forgeries for 100 different users. Basically, the implemented algorithm is an improved version of the presented one [19] by adding dynamic features, such as speeds and accelerations, at the GMM matching.

The algorithm performance was tested by the public database available in [18]. The DET (Detection Error Tradeoff) curve, shown in Figure 2, represents the relationship between FMR (False Match Rate) and FNMR (False Non-Match Rate) for different threshold decision values [16,19]. The EER (Equal Error Rate) achieved by the proposed online signature algorithm is 2.74%, which is a competitive performance when compared to other biometric modalities presented in some competitions [20].



**Figure 2.** DET curves of the signature verification for algorithms [16,19] (GMM Modified and GMM, respectively).

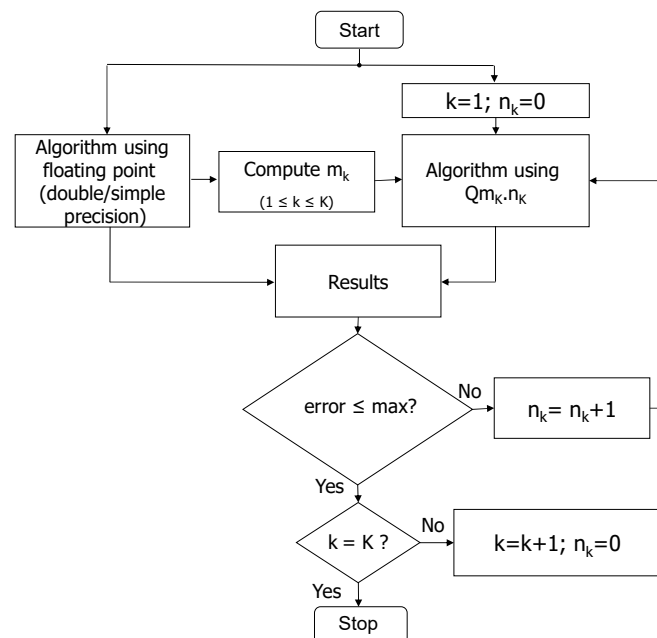
### 3. Hardware Implementations

This paper proposes the implementation of three different approaches for the signature verification system using an FPGA. The first one is an embedded system based on a 32-bit microprocessor, which includes a single-precision floating-point unit (FPU). In the second one, the FPU is substituted by a hardware accelerator especially designed for resolving calculations on floating-point vectors. The last one is a custom computing system built by means of a large set of fixed-point arithmetic circuits.

The algorithm was initially developed on MATLAB where, by default, real numbers are represented by IEEE 754 double-precision format. This standard is a 64-bit floating-point format, which provides both a large dynamic range of representable data and high resolution [21]. However, arithmetic circuits based on this format are significantly complex, and usually, only 64-bit high-performance processors include double-precision FPU. Therefore, to achieve high efficiency when the algorithm is implemented on low-cost FPGAs, the double-precision data must be replaced to simplify the arithmetic circuits but without affecting the biometric performance. Moreover, the format replacement also reduces the required capacity of the involved memories.

As mentioned, the systems based on the 32-bit embedded microprocessor require an IEEE 754 single-precision FPU or the hardware accelerator. However, the implementation of the customized system using the single-precision format is not practical due to the large number of required arithmetic circuits. The IEEE 754 arithmetic circuits require a large number of hardware resources and execution clock cycles due to denormalization, normalization and rounding steps that are not carried out on integer arithmetic circuits. Alternatively, real numbers can be represented by binary fixed-point formats, which are basically integers scaled by a constant factor. Consequently, the implementation of arithmetic circuits in such format is much simpler and faster than their floating-point counterparts. However, the number of bits in the fixed-point format must be properly adjusted to avoid overflow or underflow problems that would affect the algorithm performance.

As the algorithm includes several stages that require different ranges and precisions, a variable fixed-point format is chosen. Using the Q-format [22] and the notation  $Q_{m_k, n_k}$  to represent a format,  $m_k$  and  $n_k$  stands for the number of bits for the integer and fractional parts, respectively. If  $K$  is the total number of computations involved in the whole algorithm, the  $m_k$  and  $n_k$  values in the  $k$ -th computation are selected according to the methodology shown in Figure 3. Note as  $m_k$  can be easily calculated by logging the range of results to avoid overflow errors. On the other hand,  $n_k$  is selected as the minimum value that does not provide excessive error due to lack of precision.



**Figure 3.** Methodology used to adjust the fixed-point formats on MATLAB.

The algorithm is initially executed in double-precision and single-precision formats to record results and verify that both versions provide the same biometric performance. The final distance scores ( $r_{\text{double-precision}}$ ,  $r_{\text{single-precision}}$ ) are used to calculate the maximum error criterium for the fixed-point version. Then, the methodology continues executing the

fixed-point version of the algorithm, obtaining  $m_k$  from the logged results and incrementing  $n_k$  until the distance score ( $r_{Q_{m,n}}$ ) meets the error criterium at each computation on the entire signature database, according to the following equation.

$$\text{error} = \left| \frac{r_{\text{double-precision}} - r_{Q_{m,n}}}{r_{\text{double-precision}}} \right| \leq \left| \frac{r_{\text{double-precision}} - r_{\text{single-precision}}}{r_{\text{double-precision}}} \right| \quad (1)$$

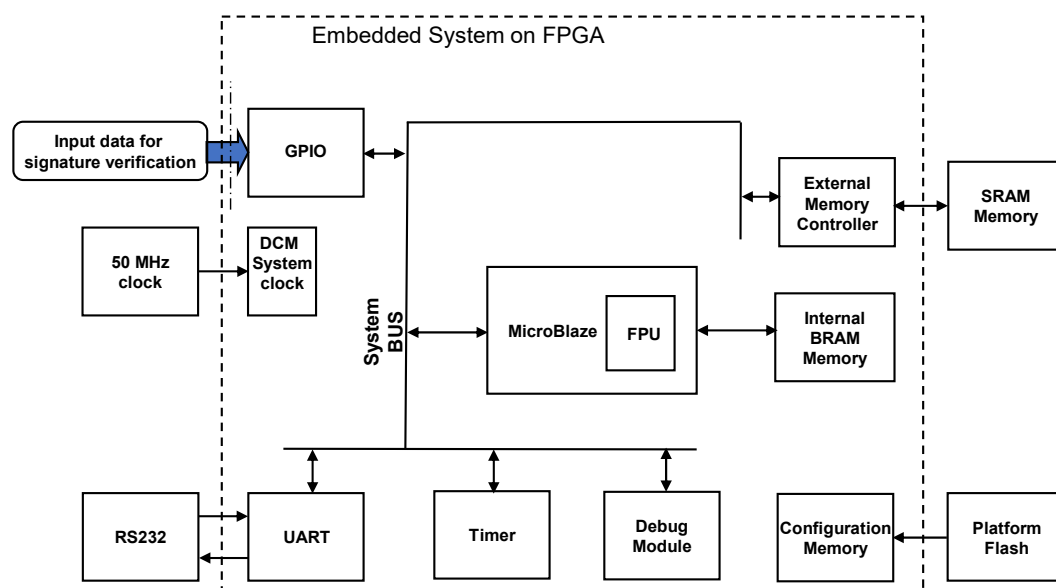
The fixed-point formats obtained range from the simpler Q10.0 format at the pressure samples acquisition to the largest Q7.45 format associated with the pseudo-distances. Table 1 shows the statistical analysis of the errors obtained from both algorithm versions on the database.

**Table 1.** Statistical analysis of data formats.

Data format	Average Error	Standard Deviation	Variance	Median
Single-precision	$1.292 \times 10^{-3}$	$7.768 \times 10^{-3}$	$6.034 \times 10^{-5}$	$2.272 \times 10^{-4}$
Fixed-point	$3.353 \times 10^{-4}$	$7.229 \times 10^{-4}$	$5.227 \times 10^{-7}$	$9.541 \times 10^{-5}$

### 3.1. MicroBlaze Embedded System

Figure 4 shows an overview of the embedded system based on MicroBlaze, a 32-bit soft-core general-purpose microprocessor for Xilinx FPGAs. It can be configured to better fit the computational requirements with the possibility of providing a single-precision FPU to accelerate floating-point operations. In this approach, the algorithm is written in C programming language, and single-precision IEEE 754 format is used. The C compiler automatically translates floating-point code to FPU instructions, and, therefore, the implementation or modification of the algorithm is reduced to a programming problem. Consequently, this implementation provides the lowest development effort but the worst execution time when compared against the other alternatives. The flexibility of the system is very high since the hardware can be reused for other biometric algorithms.



**Figure 4.** Overview of the embedded system.

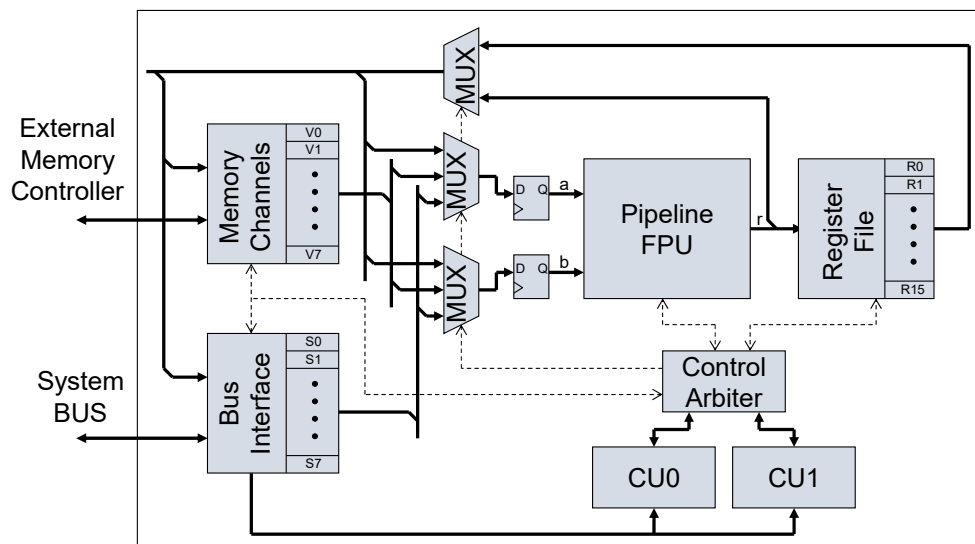
### 3.2. Hardware Accelerator

The second approach attaches a hardware accelerator, named FPBA (Floating-Point Biometric Accelerator) [23], to the embedded system. The FPBA is a programmable accel-

ator designed to speed up operations on floating-point vectors, commonly used in several biometric algorithms. MicroBlaze configures the FPBA at runtime, setting pointer registers to the vectors stored on external memory and the floating-point instructions that must be executed. Then, the FPBA launches the instructions, automatically retrieving/writing vectors from/to memory when required through memory channels, independent of MicroBlaze. Starting from the first approach, the implementation of the algorithm is also reduced to a programming problem but replacing the C code that must be accelerated by FPBA instructions on floating-point vectors.

The FPBA is basically a programmable processor that controls a pipelined FPU attached to memory channels, improving the execution time of biometric algorithms on floating-point vectors. A vector is a set of sequentially ordered data, which is stored in the memory. Vector lengths are configurable, launching a set of FPBA instructions repeatedly in hardware-controlled loops. Partial results of a vector computation can be temporarily stored in the internal registers and can be retrieved when needed. Any FPBA instruction defines the operation type (MIN, ADD, SQRT, etc.) on the involved source and destination operators, which can be vectors (V0 to V7), registers (R0 to R15) or scalars (S0 to S7).

Figure 5 depicts an overview of the architecture, which is mainly focused on the fast execution of single-precision operations on vectors, commonly used in biometrics. Firstly, the pipelined FPU increases the throughput to one instruction per clock cycle of the most frequently used operations (as the maximum, addition/subtraction and multiplication). The throughput of more complex operations, such as the square root and exponential, is greatly decremented to one instruction per 28 clock cycles. Secondly, a register file temporarily stores partial results of operations that form a computation over vectorized data. Finally, the double control unit (CU0 and CU1) permits two sets of instructions virtually executed in parallel, multiplexing on time the driving of signals for the FPU and registers through an arbiter. The CU0 can continuously launch those instructions with the faster throughput; meanwhile, CU1 is waiting for the result from instructions with a slower throughput.



**Figure 5.** Overview of the FPBA architecture.

FPBA instructions should be properly ordered and grouped, reducing latencies due to operand dependencies. Moreover, instructions can also be separated to take profit of the double control unit, enhancing the execution time. This approach improves the execution time when compared with the previous embedded system, maintaining the flexibility of the hardware reusability for other biometrics. Nonetheless, it requires a deep knowledge of the FPBA architecture and timing details, and, therefore, a higher development effort when compared with the first approach.

### 3.3. Custom Computing System

The last implementation approach is a custom computing system, which does not require the MicroBlaze embedded processor, as depicted in Figure 6. The PicoBlaze, which is a very simple 8-bit microcontroller, is primarily used to trigger the starting of the verification and to retrieve the final score through an RS232 communication port, as well as for debugging tasks. All computations are carried out using a set of specialized arithmetic circuits that are provided with internal BRAM (Block RAM) memories to share data between them. The DTW circuit additionally requires external memory to store the large result matrix. The feature extraction circuit is embedded in the GMM circuit, and it permits the parallel computing of the pseudo-distances along the optimal warping path, in order to reduce the execution time.

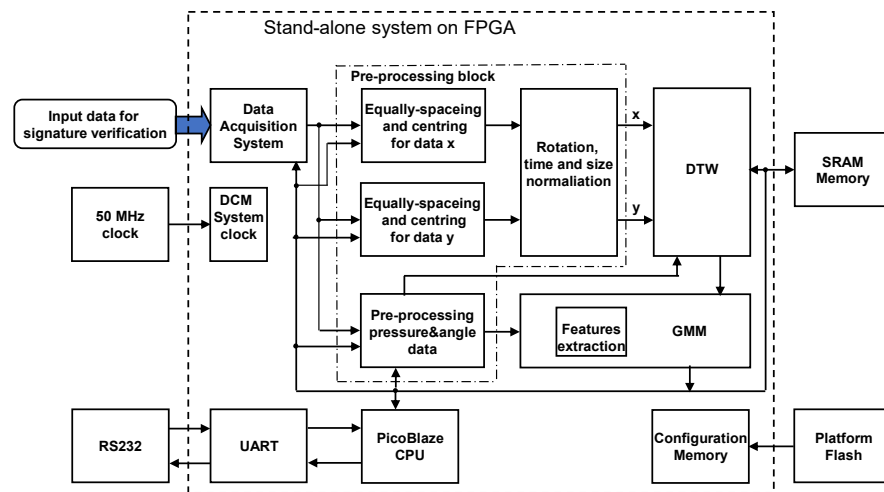


Figure 6. Architecture overview of the customized system.

### 4. DTW

The DTW result is used to find the optimal time alignment between signals  $s$  and  $t$  by computing a  $G$  matrix, which is composed of  $g(i, j)$  points. The optimal warping path is the set of points with minimal  $g(i, j)$ , pairing each  $s(j)$  sample with another  $t(i)$  sample. Figure 7 shows the bidimensional DTW implemented in the signature algorithm, which aligns 256-width  $x$ - $y$  samples from the acquired signature  $\{s_x, s_y\}$  to the enrolled template  $\{t_x, t_y\}$ . The DTW computes the result matrix  $G$  by pairing shape samples  $\{s_x(j), s_y(j)\}$  and  $\{t_x(i), t_y(i)\}$  on a permitted region  $R$ . Typically, DTW regions can be the Sakoe–Chiba band or the Itakura parallelogram [15,24], which is the one chosen in the work presented. The algorithm starts declaring the initial condition and continues computing each new  $g(i, j)$  on  $R$ , until the last  $g(255, 255)$  is calculated.

The initial condition is declared at the point  $i = j = -1$ . Alignment outside the region is not permitted by declaring  $g(i, j) = \infty$  on the points that do not belong to  $R$ .

$$g(-1, -1) = 0 \tag{2}$$

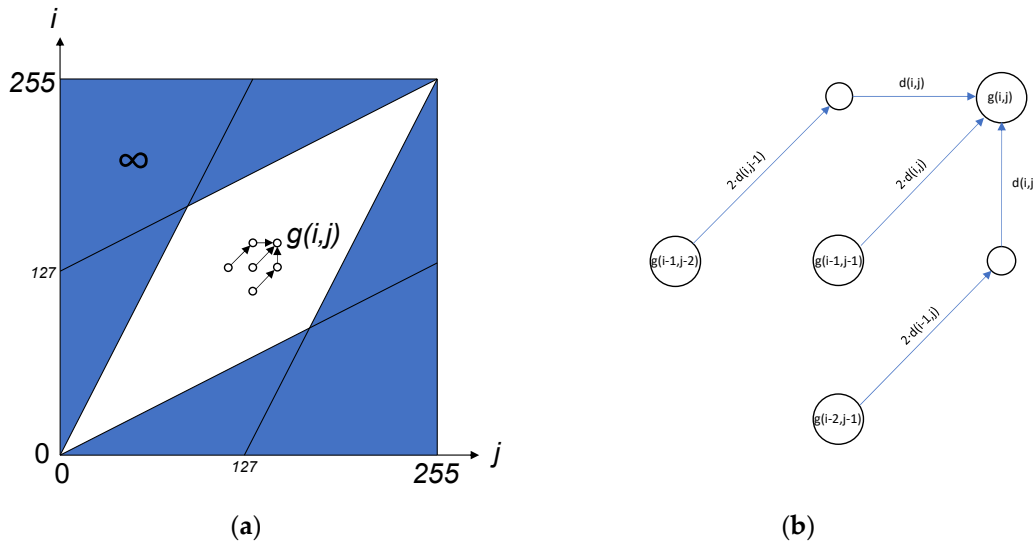
$$g(i, j) = \infty; (i, j) \notin R \tag{3}$$

Each  $g(i, j)$  on  $R$  is selected as the minimum value of three possible paths, depending on both previously computed points of the  $G$  matrix and three distances.

$$g(i, j) = \min \left\{ \begin{array}{l} g(i-1, j-2) + 2 \cdot d(i, j-1) + d(i, j) \\ g(i-1, j-1) + 2 \cdot d(i, j) \\ g(i-2, j-1) + 2 \cdot d(i-1, j) + d(i, j) \end{array} \right\}; (i, j) \in R \tag{4}$$

The  $d(i, j)$  denotes the Euclidean distance between the bidimensional samples, which involves a square root operation.

$$d(i, j) = \sqrt{(t_x(i) - s_x(j))^2 + (t_y(i) - s_y(j))^2} \tag{5}$$



**Figure 7.** (a) The Itakura parallelogram DTW region  $R$  to compute the  $G$  matrix, and (b) the computation of each  $g(i, j)$  element at the region.

In order to enhance the execution time of the DTW algorithm, all three approaches minimize the number of square root operations by an auxiliary matrix  $D$ , which is previously calculated. The distances required to calculate  $g(i, j)$  are retrieved from the  $D$  matrix, avoiding repeating the distance computation on the same point. Furthermore, the runtime of the DTW algorithm can be optimized by parallelizing the computations of the matrices  $D$  and  $G$ , which are adopted in the last two approaches. Finally, a high throughput square root calculation on the previous optimizations can greatly reduce the DTW execution time, as in the custom circuit approach.

#### 4.1. MicroBlaze Embedded System

The next programming pseudo-C code shows the main details about the implementation of the algorithm on the MicroBlaze microprocessor. The  $G$  matrix will be computed only at the  $(i, j)$  points belonging to the region  $R$ . Consequently, an initializing function writes  $g(i, j) = \infty$ , excepting at the starting point  $g(-1, -1)$ . In order to improve execution time, a distance  $D$  matrix is previously computed to avoid repeated computations of distances on the same point. The  $D$  matrix stores  $d(i, j)$  at points that belong to  $R$ , and required distances are retrieved from  $D$  during the  $G$  matrix computation. Each matrix is composed of  $256 \times 256$  single-precision data, requiring 256 KB of storage capacity on external memory. As it is shown, matrices  $D$  and  $G$  are processed row by row, and each row is calculated by a set of operations on sequentially ordered data.



```

DTW_Init_G(); //Sets G-matrix g(i,j)=∞, except g(-1,-1)=0

for(i=0, i<256; i++) { //D-matrix computation, from row 0 to 255
    DTW_Region_D(i,&j0,&j1); //Computes column bounds for D-matrix at row i
    for(j=j0, j<j1; j++) //Computes d(i,j) in all of the columns of region R
        x=tx(i)-sx(j);
        y=ty(i)-sy(j);
        d(i,j)=sqrt(x*x+y*y); }

for(i=0, i<256; i++) { //G-matrix computation, from row 0 to 255
    DTW_Region_G(i,&j0,&j1); //Computes column bounds for G-matrix at row i
    for(j=j0, j<j1; j++) //Computes g(i,j) from previously computed D-matrix
        g1=g(i-1,j-2)+2*d(i,j-1)+d(i,j); // and previously computed points of G-matrix
        g2=g(i-2,j-1)+2*d(i-1,j)+d(i,j);
        g3=g(i-1,j-1)+2*d(i,j);
        g(i,j)=min(g3,min(g1,g2)); }

```

#### 4.2. Hardware Accelerator

In the previous programming code, column data from the  $D$  and  $G$  matrices are sequentially allocated in external memory, as in floating-point vectors. Consequently, the FPBA can accelerate computations carried out on columns by launching a set of instructions. The throughput of all required instructions is one per clock cycle, except the square root instruction in the  $D$  matrix computation. The execution time on the FPBA is enhanced by the double control unit, which launches instructions for  $G$  and  $D$  matrices in a time-multiplexed way. Since  $G$  computation requires the previous calculus of  $D$ , the computation of the  $G$  matrix at row  $i$  is virtually executed in parallel with the  $D$  matrix at the next row  $i + 1$ . For the sake of simplicity, the programming code for the first and last row is omitted as they are slightly different from the code shown.

```

DTW_Init_G(); //Sets G-matrix g(i,j)=∞, except g(-1,-1)=0
CoproFPU_Scalar0(2.0); //Writes the constant 2 into the scalar S0

for(i=1, i<255; i++) { //G-matrix computation, from row 1 to 254
    DTW_Region_D(i+1,&j0,&j1); //Computes column bounds for D-matrix at row i+1
    FPBA_Length(j1-j0,CU1); //Sets vector length for CU1
    FPBA_V7(&d(i+1,j0)); //Output vector V7 pointer to D-matrix
    FPBA_V5(&sx(j0)); FPBA_V6(&sy(j0)); //Input vectors V5,V6 pointers to sx,sy
    FPBA_S5(tx(i+1)); FPBA_S6(ty(i+1)); // and scalars S5,S6 to store tx,ty

    DTW_Region_G(i,&j0,&j1); //Computes column bounds for G-matrix at row i
    FPBA_Length((j1-j0)/2,CU0); //Sets vector length for CU0
    FPBA_V4(&g(i,j0)); //Output vector V4 pointer to G-matrix
    FPBA_V0(&d(i,j0-1)); FPBA_V1(&d(i-1,j0)); //Input vectors V0,V1 pointers to D-matrix
    FPBA_V2(&g(i-1,j0-2)); FPBA_V3(&g(i-2,j0-1)); // and V2,V3 pointers to G-matrix

    FPGA_Run(CU0,CU1); //Start CU0 and CU1
}

```

The  $D$  matrix computation is performed by the following CU1 code, which consists of six instructions in a loop, generating  $(j1 - j0)$  iterations by controlling a  $k$  index, which starts at 0. The first instruction stores in register R14 the subtraction result from a scalar S5, which stores the  $t_x(i + 1)$  sample, and a sample  $s_x(j)$  from the vector V5. The  $s_x(j)$  values are sequentially retrieved from memory according to  $j = j0 + k$ , since the vector pointer was configured to start at the  $s_x(j0)$  element. Similarly, another instruction stores in R15 the other required subtraction, and the following instructions complete the calculation. The last instruction stores the result  $d(i + 1, j)$  sequentially on the V7 pointer, which was configured by MicroBlaze to the first element  $d(i + 1, j0)$ .

```

BEGIN_LOOP:                                     //Loop for vectors (0<=k<j1-j0; j=j0+k) at row i+1
  R14←S5-V5;   R15←S6-V6;                       //R14=tx(i+1)-sx(j)       R15=ty(i+1)-sy(j)
  R14←R14*R14; R15←R15*R15;                       //R14=(tx(i+1)-sx(j))2   R15=(ty(i+1)-sy(j))2
  R15←R14+R15;                                     //R15=(tx(i+1)-sx(j))2+ (ty(i+1)-sy(j))2
  V7←SQRT(R15);                                    //d(i+1,j)=sqrt((tx(i+1)-sx(j))2+ (ty(i+1)-sy(j))2)
END_LOOP:                                         //End of loop for vectors

```

In order to enhance the  $G$  matrix computation, the CU0 instructions are ordered to reduce latencies due to operand dependencies and grouped to calculate two consecutive  $g(i, j)$  and  $g(i, j + 1)$  elements at each loop iteration. Consequently, the vector length was configured to  $(j_1 - j_0)/2$  and the last two instructions write  $g(i, j)$  and  $g(i, j + 1)$  at each iteration in the V4 vector. The V0 and V1 vector pointers are set to the first required elements of  $D$  at the rows  $i$  and  $i - 1$ , to read  $d(i, j - 1)$  and  $d(i - 1, j)$ , respectively. Similarly, V2 and V3 vectors point the first elements of  $G$  elements to be retrieved from memory, reading a new  $g(i - 1, j - 2)$  and  $g(i - 2, j - 1)$  at each iteration. After completing a set of multiplications and additions, instructions (\*) in the following code store the three possible  $g(i, j)$  values in registers R4, R5, R6. Similarly, R9, R10, R11 are written with the possible  $g(i, j + 1)$  values at (\*\*). The last instructions choose the minimum values to be stored in  $g(i, j)$ ,  $g(i, j + 1)$ , and write R0, R1 with the data required for the next iteration. Due to the fast throughput of these instructions, the CU0 launches many of them meanwhile CU1 is calculating the square root result.

```

R0←V0; R1←V2;                                     //R0=d(i, j-1) R1=g(i-1, j-2)
BEGIN_LOOP:                                       //Loop for vectors (0<=k<(j1-j0)/2; j=j0+k) at row i
  R2←V0; R7←V0; R3←V2; R8←V2;                   //R2=d(i, j) R7=d(i, j+1) R3=g(i-1, j-1) R8=g(i-1, j)
  R4←S0*R2; R9←S0*R7;                             //R4=2d(i, j) R9=2d(i, j+1)
  R5←S0*R0; R10←S0*R2;                            //R5=2d(i, j-1) R10=2d(i, j)
  R6←S0*V1; R11←S0*V1;                           //R6=2d(i-1, j) R11=2d(i-1, j+1)
  R5←R1+R5; R10←R3+R10;                          //R5=g(i-1, j-2)+2d(i, j-1) R10=g(i-1, j-1)+2d(i, j)
  R6←V3+R6; R11←V3+R11;                          //R6=g(i-2, j-1)+2d(i-1, j) R11=g(i-2, j)+2d(i-1, j+1)
  R5←R2+R5;                                       //R5 =g(i-1, j-2)+2d(i, j-1)+d(i, j) (*)
  R10←R7+R10;                                    //R10=g(i-1, j-1)+2d(i, j)+d(i, j+1) (**)
  R6←R2+R6;                                       //R6 =g(i-2, j-1)+2d(i-1, j)+d(i, j) (*)
  R11←R7+R11;                                    //R11=g(i-2, j)+2d(i-1, j+1)+d(i, j+1) (**)
  R4←R3+R4;                                       //R4 =g(i-1, j-1)+2d(i, j) (*)
  R9←R8+R9;                                       //R9 =g(i-1, j)+2d(i, j+1) (**)

  R12←MIN(R5, R6); R13←MIN(R10, R11);             //Partial minimum value for g(i, j) and g(i, j+1)
  R0←R7; R1←R8;                                   //R0=d(i, j+1) R1=g(i-1, j) for the next iteration
  V4←MIN(R4, R12); V4←MIN(R9, R13);              //Writes g(i, j) and g(i, j+1)
END_LOOP:                                         //End of loop for vectors

```

### 4.3. Custom Computing System

The DTW computing circuit is a highly parallelized architecture able to simultaneously access several double-port internal BRAM memories, performing a throughput of one  $g(i, j)$  calculus per clock cycle after an initial latency. It is composed of two dedicated circuits built from fixed-point arithmetic circuits and a controller unit, as depicted in Figures 8 and 9. The first circuit is devoted to the calculus of the  $D$  matrix at the next row  $i + 1$ , while the second circuit computes the  $G$  matrix on the current row  $i$ , similarly as in the FPBA. The controller not only drives the multiplexers, demultiplexers and BRAMs at the computing circuits but also maintains a proper synchronization of both circuits and generates the required indexes for the region  $R$ . The region bounds  $j_0, j_1$  at the  $i$  and  $i + 1$  rows are retrieved from two BRAM memories (acting a ROMs), generating the required range of  $j$  indexes during the calculation of  $D$  and  $G$  at each row.

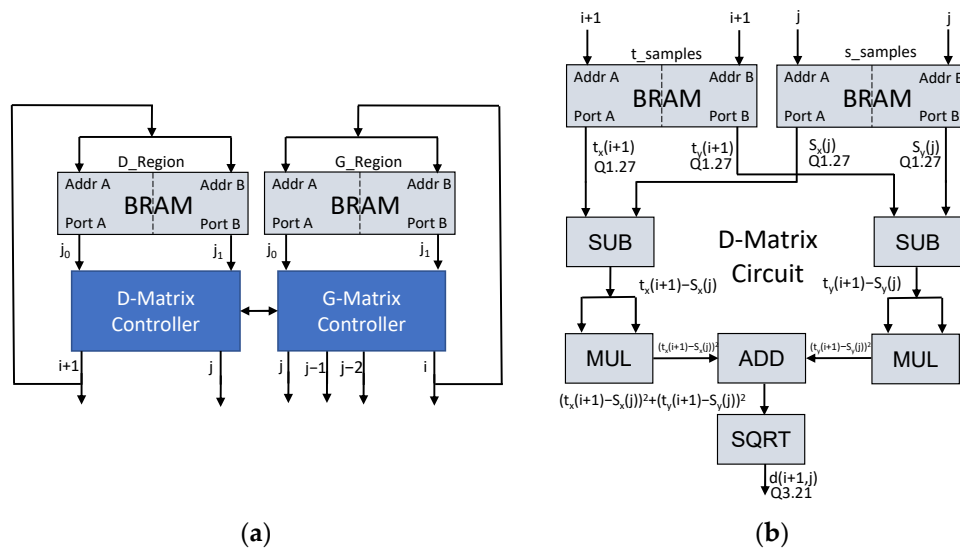


Figure 8. (a) Controller unit; (b) datapath of the  $D$  matrix computing circuit.

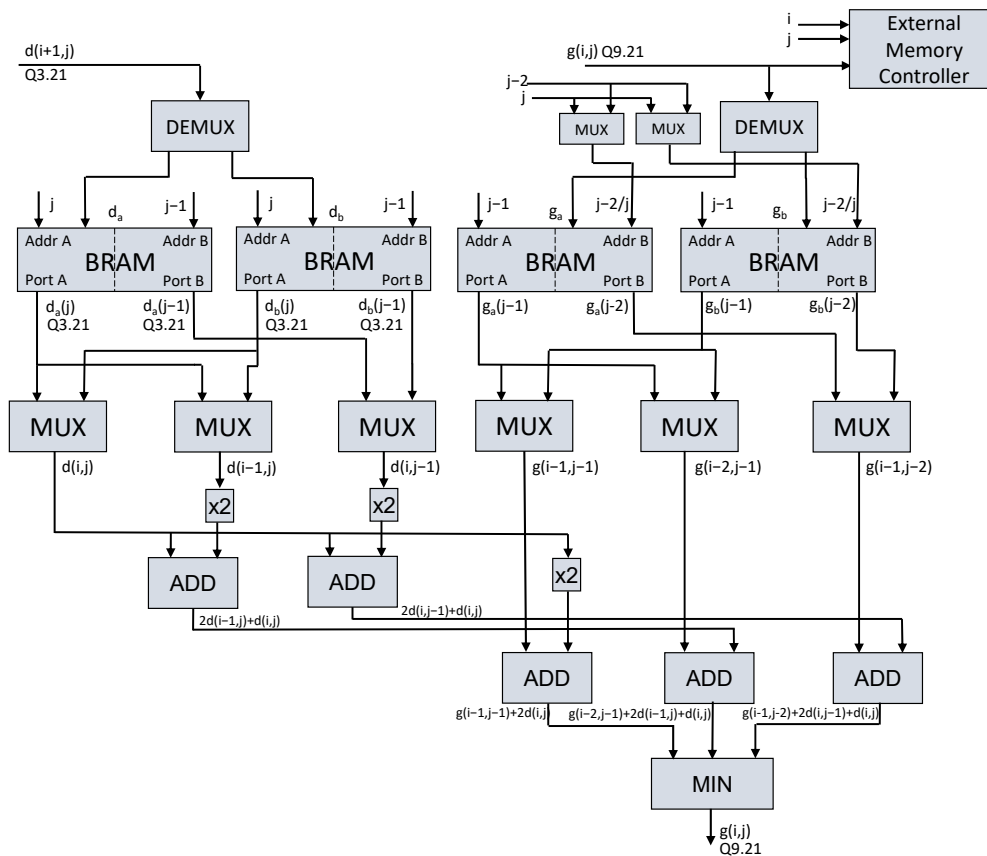


Figure 9. Datapath of the  $G$  matrix computing circuit.

The  $D$  matrix circuit is quite simple. It devotes two BRAMs to retrieve the  $t_x(i + 1)$ ,  $t_y(i + 1)$ ,  $s_x(j)$ ,  $s_y(j)$  samples and a few arithmetic fixed-point circuits to calculate the distances  $d(i + 1, j)$ . The square root circuit is the CORDIC core from Xilinx’s LogiCORE [25], which is configured to enhance the performance by fully parallel architecture with a single-cycle data throughput. The core is able to calculate one square root per clock cycle in fixed-point format after an initial latency of 22 clock cycles. Distance results are temporarily stored in BRAMs instead of the external RAM, since only the previous two rows of  $D$

are required during the computation of a row of  $G$ . The double-port BRAMs allow to simultaneously store a new result and retrieve a previously calculated distance.

The  $G$  matrix circuit is significantly more complex. It devotes two BRAMs ( $d_a, d_b$ ) and a set of multiplexers and demultiplexers to store the result  $d(i+1, j)$  and to retrieve the three distances  $d(i, j), d(i-1, j), d(i, j-1)$  in a single clock cycle, as required for the calculation of each  $g(i, j)$ . During a row computation,  $d(i, j)$  and  $d(i, j-1)$  are read from BRAM  $d_b$ ,  $d(i-1, j)$  is retrieved from BRAM  $d_a$ , and meanwhile it stores the result  $d(i+1, j)$ . The roles of  $d_a$  and  $d_b$  are alternatively swapped on each new row calculation. Similarly, only two rows of  $G$  are required during the  $g(i, j)$  calculation, and two BRAMs simultaneously retrieve all the necessary data and store the result. During a row calculation,  $g(i-1, j-1)$  and  $g(i-1, j-2)$  are read from  $g_b$ ,  $g(i-2, j-1)$  is retrieved from  $g_a$  meanwhile it stores the result  $g(i, j)$ . On each new row, the  $g_a, g_b$  roles are swapped, and a set of fixed-point arithmetic circuits calculate the  $g(i, j)$  result. The results are also written into the external memory since the following feature extraction stage requires the complete  $G$  matrix to compute the pseudo-distances on the optimal warping path.

## 5. Experimental Results

### 5.1. Speed Processing

The execution times, expressed in clock cycles, for the three implementations are experimentally obtained and shown in Table 2 at each of the stages of the algorithm. These results represent the average execution time obtained from all the available signatures of a single user. The execution times are not dependent on the presented signature, except a slight variation in the features extraction stage due to the different warping paths used to compute the pseudo-distances, which is negligible in the total execution time. Between brackets, the speed-up factors of the last two implementations are reported when compared with the embedded system. The table also shows the execution time in milliseconds at the clock frequencies used in the implemented systems, according to the synthesis results.

**Table 2.** Execution times, expressed in clock cycles (speed-up factor) and milliseconds.

	MicroBlaze Embedded System 40 MHz	FPBA and Embedded System 40 MHz	Custom System 50 MHz
Pre-processing	$236.80 \times 10^3$ 5.92 ms	$36.47 \times 10^3$ ( $\times 6.5$ ) 0.91 ms	$1.38 \times 10^3$ ( $\times 171.6$ ) 0.028 ms
DTW	$4786.80 \times 10^3$ 119.7 ms	$632.64 \times 10^3$ ( $\times 7.6$ ) 15.8 ms	$21.87 \times 10^3$ ( $\times 218.8$ ) 0.437 ms
Features extraction	$15.15 \times 10^3$ 0.38 ms	$2.92 \times 10^3$ ( $\times 5.2$ ) 0.073 ms	$1.76 \times 10^3$ ( $\times 8.6$ ) 0.035 ms
GMM	$146.26 \times 10^3$ 3.66 ms	$11.54 \times 10^3$ ( $\times 12.7$ ) 0.29 ms	$0.75 \times 10^3$ ( $\times 195$ ) 0.015 ms
Total	$5185.01 \times 10^3$ 129.6 ms	$683.57 \times 10^3$ ( $\times 7.6$ ) 17.1 ms	$25.76 \times 10^3$ ( $\times 201.3$ ) 0.515 ms

The total execution time is clearly dominated by the DTW stage since it requires both calculations on large matrices and square root operations. Experimental results show that the speed-up factors are  $\times 7.6$  and  $\times 201$  for the FPBA and custom system, respectively. The acceleration of the custom DTW circuit is very noticeable due to the high throughput achieved when calculating  $g(i, j)$  (one computation per clock cycle).

As stated in Section 1, there is a lack of online signature systems on FPGAs. However, we can compare the acceleration of the execution time of our custom DTW circuit against other DTW-based FPGA accelerators in Table 3. The 3D-DTW accelerator [3] for action recognition is designed to be connected to the ARM microprocessor embedded in the target Zynq-7000 device, a high-performance SoC from Xilinx. The core running at 100 MHz

can accelerate the execution time  $\times 40$  times faster than the software counterpart. A closer comparison can be achieved with the 2D-DTW accelerator for embedded platforms [26]. As the previous accelerator, it is designed to be connected to an ARM microprocessor on a Cyclone V device from Altera, achieving a  $\times 7.5$  speed-up running at 60 MHz.

**Table 3.** Comparison of the DTW algorithms, execution and cycle time (in clock cycles and milliseconds/nanoseconds), and speed-up.

	Frequency (MHz)	Dim	Points in R	Execution Time	Cycle Time	Speed-Up
[3]	100	3D	331,776	$11.6 \cdot 10^6$ 116 ms	34.9 349 ns	$\times 40$
[26]	60	2D	3125	7200 0.12 ms	2.3 38.4 ns	$\times 7.5$
FPBA	40	2D	21,845	$632.6 \times 10^3$ 15.8 ms	29.0 723.28 ns	$\times 7.6$
Custom System	50	2D	21,845	21,870 0.44 ms	1.0 20.01 ns	$\times 218.8$

We compare the execution time of different DTW implementations since the result depends on several parameters. The dimension of samples affects the calculation of the distance, the sample width and region  $R$  determine the number of points to compute  $g(i, j)$ , and the data format affects the latency of the arithmetic circuits. Therefore, the cycle time of calculating  $g(i, j)$  is a better comparative measurement.

The core shown in [3] presents important differences when compared with the DTW algorithm implemented in this work. First, it does not compute the optimal warping path since it only requires the minimum  $g(i, j)$  value at the last row, which scores the similarity of the aligned samples. The algorithm is repeated in 12 different templates in order to classify the captured action. Moreover, each distance is calculated from the addition of 3 square root operations. Additionally, it does not use any specific region, calculating  $g(i, j)$  at all possible points. Finally, to improve the execution time, the DTW stops if the minimum value of  $g(i, j)$  at any point in a row is greater than a threshold condition. The execution time for 576-width floating-point samples ranges from  $8.7 \times 10^6$  to  $11.6 \times 10^6$  clock cycles at 100 MHz. The higher time should be considered since it calculates  $g(i, j)$  on all the points (331,776 points), providing a cycle time of 349 ns (34.9 clock cycles).

The DTW accelerator [26] is much closer to the proposed version used in this paper since it implements the Euclidean distance for 2D samples. However, the resolution of samples is limited to 8-bit, and it uses the Sakoe–Chiba region. The execution time for 250-width samples is 0.12 ms (7200 clock cycles) by computing only the 5% of the  $G$  matrix (3125 points) due to the parameters of the region. Therefore, the cycle time when calculating  $g(i, j)$  is about 38.4 ns (2.3 clock cycles).

The custom computing system works with 28-bit samples using a Q1.27 fixed-point format. The DTW execution time, for 256-width samples on an Itakura region that computes 33.3% of the  $G$  matrix (21,845 points), is 0.437 ms (21,870 clock cycles). Therefore, the cycle time of  $g(i, j)$  achieves 20.01 ns, which is very close to the theoretical one (1 clock cycle).

The FPBA is not specifically designed to calculate the DTW but rather to speed up several computing stages of biometric algorithms. Therefore, the performance is not as competitive as in the DTW accelerators. The 256-width samples are in 32-bit floating-point format, taking 15.8 ms to calculate the DTW in the same region as in the custom system. The cycle time is 723.28 ns (29.0 clock cycles).

### 5.2. FPGA Resources

The devoted FPGA hardware resources and clock frequency are obtained from the synthesized circuits on a XC3S2000 device and reported in Table 4. The incremented ratio

of FPGA resources for the FPBA and the custom systems, when compared with the first implementation, are also reported in brackets. The FPBA is attached to the embedded system, and therefore, the logic resources are incremented. The custom computing system is built from a large set of dedicated arithmetic circuits and internal memories, significantly incrementing the number of FPGA resources needed. Mainly, the CLBs (Complex Logic Block), which mainly consists of LUTs (LookUp Table) and FFs (Flip-Flop), are incremented by factors  $\times 2.3$  and  $\times 5.2$  for the FPBA and the custom system, respectively. The clock frequency of the custom system is slightly increased due to the simplified architecture of the fixed-point arithmetic circuits.

**Table 4.** Hardware FPGA resources (area ratio) and clock frequency.

	MicroBlaze Embedded System	FPBA and Embedded System	Custom System
CLB Slices	2393	5434 ( $\times 2.27$ )	12,537 ( $\times 5.23$ )
LUTs	4148	8999 ( $\times 2.17$ )	20,416 ( $\times 4.92$ )
FFs	2591	5663 ( $\times 2.19$ )	10,482 ( $\times 4.05$ )
MULT 18 $\times$ 18	7	11 ( $\times 1.57$ )	18 ( $\times 2.57$ )
BRAM 18 Kbit	16	16 ( $\times 1.00$ )	29 ( $\times 1.81$ )
Clock frequency	40 MHz	40 MHz	50 MHz

## 6. Discussion

The paper presents three different approaches to implementing an online signature verification system on a low-cost FPGA family from Xilinx. Firstly, the IEEE 754 double-precision data were replaced by simpler formats without degrading the algorithm biometric performance. The simple-precision floating-point format is adequate for embedded systems based on the 32-bit microprocessor MicroBlaze configured with internal FPU. The same data format is used for the hardware FPBA, which accelerates floating-point computations on data vectors. An alternative representation of real numbers is the fixed-point format since arithmetic circuits are simpler and faster than their floating-point counterparts. This data format is used by the custom computing system, which is built from a large set of devoted arithmetic circuits.

Each of the three tested alternatives has its advantages and drawbacks. The MicroBlaze embedded system provides a simpler and faster development cycle since the compiler automatically translates single-precision operations to FPU instructions. The system provides a high flexibility because it could be adapted to different biometric algorithms by including a new programming code. Nonetheless, execution time is limited due to the floating-point operations on large data. The FPBA accelerates the processing time ( $\times 7.6$ ) on floating-point vectors, also providing a high flexibility. However, the programming effort is increased since it is required to rewrite the code that must be accelerated as vector operations and write FPBA instructions properly. Another drawback is the increase in FPGA resources ( $\times 2.3$ ). The last alternative is a custom computing system built from a large set of devoted fixed-point arithmetic circuits and internal BRAM memories. This proposal features the opposite advantages and drawbacks when compared to the embedded system, providing a very large acceleration ( $\times 201$ ) but no system flexibility. Furthermore, the implementation area is noticeably incremented ( $\times 5.2$ ), and the development effort is enormous due to the large set of customized computational circuits and memories that must be managed.

Depending on the target, one proposal is preferred against the others. Taking a multi-modal biometric system into perspective, a custom computing system could provide a very fast execution time at the expense of a huge increase in both the FPGA resources and development effort, which may not be justifiable. In such a case, the FPBA could be the preferred option since it allows accelerating the processing time without increasing the hardware resources, along with a reasonable development effort.

**Author Contributions:** All authors wrote and reviewed specific sections of the paper. E.C.N. focused on the algorithm porting to C and the FPBA system and experimental results. R.R.L. mainly contributed to the custom computing system and experimental results. M.L.G. mainly contributed to the MATLAB algorithm and optimizations. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was funded by Spanish MCIN/AEI/10.13039/501100011033, grant number PID2019-107274RB-I00.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Not applicable.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Reynoso-Godinez, R.; Rodriguez-Donate, C.; Lopez-Ramirez, M.; García-Guevara, F.M. FPGA-based parallel process for Walsh-Hadamard transform. *Electron. Lett.* **2020**, *56*, 1039–1041. [[CrossRef](#)]
2. Afifi, S.; GholamHosseini, H.; Sinha, R. FPGA Implementations of SVM Classifiers: A Review. *SN Comput. Sci.* **2020**, *1*, 133. [[CrossRef](#)]
3. Vidhaypathi, C.M.; Raj, J.; Noel, A.; Sundar, S. The 3D-DTW Custom IP based FPGA Hardware Acceleration for Action Recognition. *J. Imaging Sci. Technol.* **2021**, *65*, 10401-1–10401-10. [[CrossRef](#)]
4. Xu, J.; Li, S.; Jiang, J.; Dou, Y. A Simplified Speaker Recognition System Based on FPGA Platform. *IEEE Access* **2019**, *8*, 1507–1516. [[CrossRef](#)]
5. Ramos-Lara, R.; López-García, M.; Cantó-Navarro, E.; Puente-Rodríguez, L. Real-Time Speaker Verification System Implemented on Reconfigurable Hardware. *J. Signal Process. Syst.* **2013**, *71*, 89–103. [[CrossRef](#)]
6. McGuffey, C.; Liu, C.; Schuckers, S. Hardware Accelerator Approach Towards Efficient Biometric Cryptosystems for Network Security. *J. Comput. Inf. Technol.* **2015**, *23*, 329–340. [[CrossRef](#)]
7. López, M.; Daugman, J.; Cantó, E. Hardware-software co-design of an iris recognition algorithm. *IET Inf. Secur.* **2011**, *5*, 60–68. [[CrossRef](#)]
8. Sandeep, D.; Chandrakanth, C. Online Signature Verification by Using FPGA. *Int. J. Mag. Technol. Manag. Res.* **2017**, *4*, 372–377.
9. López-García, M.; Ramos-Lara, R.; Miguel-Hurtado, O.; Cantó-Navarro, E. Embedded System for Biometric Online Signature Verification. *IEEE Trans. Ind. Electron.* **2014**, *10*, 491–501. [[CrossRef](#)]
10. Mansour, A.H.; Salh, G.Z.A.; Mohammed, K.A. Voice Recognition using Dynamic Time Warping and Mel-Frequency Cepstral Coefficients Algorithms. *Int. J. Comput. Appl.* **2015**, *116*, 34–41. [[CrossRef](#)]
11. Shen, Y.; Bao, S.; Yang, L.; Li, Y. The PLR-DTW method for ECG based biometric identification. In Proceedings of the 2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society, Boston, MA, USA, 30 August–3 September 2011.
12. Bernadelli, C.R.; da Silva, P.R. Dynamic Time Warping in Iris Biometric Recognition Process. *IEEE Lat. Am. Trans.* **2021**, *19*, 42–49. [[CrossRef](#)]
13. Impedovo, D.; Pirlo, G. Automatic signature verification: The State of the Art. *IEEE Trans. Syst. Man Cybern. Part C Appl. Rev.* **2008**, *38*, 609–635. [[CrossRef](#)]
14. El-Henawy, I.; Rashad, M.; Nomir, O.; Ahmed, K. Online Signature Verification: State of the art. *Int. J. Comput. Technol.* **2005**, *4*, 664–678. [[CrossRef](#)]
15. Sakoe, H.; Chiba, S. Dynamic programming algorithm optimization for speaker word recognition. *IEEE Trans. Acoust. Speech Signal Process.* **1978**, *26*, 43–49. [[CrossRef](#)]
16. Miguel Hurtado, O. Online Signature Verification Algorithms and Development of Signature International Standards. Ph.D. Dissertation, Universidad Carlos III Madrid, Getafe, Spain, 2011.
17. Dempster, A.P.; Laird, N.M.; Rubin, R.B. Maximum likelihood from incomplete data via the EM algorithm. *J. R. Stat. Soc. Ser. B (Methodol.)* **1977**, *39*, 1–22.
18. Ortega, J.; Fierrez, J.; Simón, D.; González, J.; Faundez, M.; Espinosa, V.; Satue, A.; Hernaez, I.; Igarza, J.; Vavaracho, C.; et al. MCyT base corpus: A bimodal biometric database. *IEE Proc. Vision Image Signal Process.* **2003**, *150*, 395–401. [[CrossRef](#)]
19. Miguel-Hurtado, O.; Mengibar-Pozo, L.; Pacut, A. A new algorithm for signature verification system based on DTW and GMM. In Proceedings of the 2008 42nd Annual IEEE International Carnahan Conference on Security Technology, Prague, Czech Republic, 13–16 October 2008; pp. 206–213. [[CrossRef](#)]
20. Cappelli, R.; Ferrara, M.; Franco, A.; Maltoni, D. Fingerprint verification competition 2006. *Biom. Technol. Today* **2007**, *15*, 7–9. [[CrossRef](#)]
21. IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*; IEEE STD 754-2019; IEEE Computer Society: Washington, DC, USA, 2019; pp. 1–84.

22. Erick, L.O. Fixed-Point Representation & Fractional Math. Oberstar Consulting. 2007. Available online: <http://darcy.rsgc.on.ca/ACES/ICE4M/FixedPoint/FixedPointRepresentationFractionalMath.pdf> (accessed on 20 September 2021).
23. Cantó Navarro, E.; López-García, M.; Ramos-Lara, R. Floating-point accelerator for biometric recognition on FPGA embedded systems. *J. Parallel Distrib. Comput.* **2018**, *112*, 20–34. [[CrossRef](#)]
24. Rabiner, L.; Juandg, B. *Fundamentals of Speech Recognition*; Prentice-Hall: Hoboken, NJ, USA, 1993.
25. Xilinx LogiCORE IP CORDIC v4.0 (DS249). Available online: [https://www.xilinx.com/support/documentation/ip\\_documentation/cordic\\_ds249.pdf](https://www.xilinx.com/support/documentation/ip_documentation/cordic_ds249.pdf) (accessed on 20 September 2021).
26. Zhou, H.; Xu, X.; Hu, Y.; Yu, G.; Yan, Z.; Lin, F.; Xu, W. Energy-efficient Pipelined DTW Architecture on Hybrid Embedded Platforms. In Proceedings of the Sixth International Green and Sustainable Computing Conference, Las Vegas, NV, USA, 14–16 December 2015.