applied sciences

MDPI

*Article*

# A RTL Implementation of Heterogeneous Machine Learning Network for French Computer Assisted Pronunciation Training

**Yanjing Bi [1], Chao Li [2,3,*], Yannick Benezeth [4] and Fan Yang [4]**

1 School of Foreign Studies, Capital University of Economics and Business, Beijing 100070, China
2 State Key Laboratory of Acoustics, Institute of Acoustics, Beijing 100190, China
3 University of Chinese Academy of Sciences, Beijing 100049, China
4 Laboratory of ImViA, University of Burgundy-Franche-Comté, 21078 Dijon, France
* Correspondence: chao.li@mail.ioa.ac.cn

**Abstract:** Computer-assisted pronunciation training (CAPT) is a helpful method for self-directed or long-distance foreign language learning. It greatly benefits from the progress, and of acoustic signal processing and artificial intelligence techniques. However, in real-life applications, embedded solutions are usually desired. This paper conceives a register-transfer level (RTL) core to facilitate the pronunciation diagnostic tasks by suppressing the mulitcollinearity of the speech waveforms. A recently proposed heterogeneous machine learning framework is selected as the French phoneme pronunciation diagnostic algorithm. This RTL core is implemented and optimized within a very-high-level synthesis method for fast prototyping. An original French phoneme data set containing 4830 samples is used for the evaluation experiments. The experiment results demonstrate that the proposed implementation reduces the diagnostic error rate by 0.79–1.33% compared to the state-of-the-art and achieves a speedup of 10.89× relative to its CPU implementation at the same abstract level of programming languages.

**Keywords:** computer-assisted pronunciation training; high-level synthesis; embedded designs; machine learning; FPGA

check for updates

**Citation:** Bi, Y.; Li, C.; Benezeth, Y.; Yang, F. A RTL Implementation of Heterogeneous Machine Learning Network for French Computer Assisted Pronunciation Training. *Appl. Sci.* **2023**, *13*, 5835. https://doi.org/10.3390/app13105835

Academic Editor: Douglas O'Shaughnessy

Received: 29 March 2023
Revised: 1 May 2023
Accepted: 5 May 2023
Published: 9 May 2023

## 1. Introduction

Phoneme pronunciation is one of the most important basic skills for foreign language learning. Practicing pronunciations in a computer-assisted way is helpful in self- directed or long-distance learning environments [1]. The computer-assisted pronunciation training (CAPT) programs record and analyze user speech acoustically, comparing their pronunciation and prosody with a native speaker sample using visual feedback. Although users often require additional training to ensure that they can interpret the feedback, such programs can be used to improve their prosody and vowel pronunciation [2,3].

Recent research projects indicate that machine learning provides nice opportunities to improve CAPT systems. From the view point of natural language processing, pronunciation diagnostic tasks are essentially acoustic pattern recognition problems [4], which have made great progress [5–16]. For example, Gulati et al. [12] achieved a 1.9% word error rate on clean test data by using more than 900 hours of labeled speech training data. Turan and Erzin [13] address the close-talk and throat microphone domain mismatching problem by using a transfer learning approach based on stacking denoising auto-encoders, which allows improvement of the acoustic model by mapping the source domain representations and the target domain representations into a common latent space. Sun and Tang [14] propose a method for supporting automatic communication error detection through integrated use of speech recognition, text analysis, and formal modeling of airport operational processes. It is hypothesized that it could form the basis for automating communication error detection and preventing loss of separation. Badrinath and Balakrishnan [15] present an automatic speech recognition model tailored to the air traffic control domain that can transcribe air traffic control voice to text. The transcribed text is used to extract operational information such as call-signs and runway numbers. The models are based on recent improvements in machine learning techniques for speech recognition and natural language processing. Jiang et al. [16] applied the recent state-of-the-art DNN-based training methods to the automatic language proficiency evaluation system that combines various kinds of non-native acoustic models and native ones. The reference-free rate is used as the machine score to estimate the second-language proficiency of the English learners. The evaluations based on the English-read-by-Japanese database demonstrate that it is an effective method to improve the language proficiency assessment techniques.

Despite many significant theoretical achievements in the terms of speech recognition algorithms, the utility value of today's CAPT modalities is limited to the hardware devices, especially in the aspects of portability, maintainability, and resource consumption. Usually, the developments and evaluations of CAPT tools are realized by using general-purpose processors, which can hardly satisfy these requirements entirely; some more efforts therefore need to be made to prototype them embeddedly. E. Manor et al. [17] point out the possibility of efficiently running networks on a Field Programmable Gate Array (FPGA) using a microcontroller and hardware accelerator. In the work of Silva et al. [18], a support vector machine multi-class classifier is implemented within the asynchronous paradigm in a 4-stage architecture. It is claimed that a reduced power consumption of 5.2 mW, a fast average response time of 0.61 µs, and the most area-efficient circuit of 1315 LUTs are obtained as a result. Chervyakov et al. [19] propose a speech-recognition-available CNN architecture based on the Residue Number System (RNS) and the new Chinese Remainder Theorem with fractions. According to the simulations based on the Kintex7 xc7k70tfbg484-2 FPGA, the hardware cost is reduced by 32% compared to the traditional binary system. Paula et al. [20] apply the long short-time memory network to the task of spectral prediction and propose a module generator for an FPGA implementation. Evaluations demonstrate that a prediction latency of 4.3 µs on a Xilinx XC7K410T Kintex-7 FPGA is achievable. Up to present, mature embedded machine learning toolkits like OpenVINO have been developed and widely used in real-life research and developments [21–25]. These successful cases significantly improved the products in different scenarios.

This work focuses on the French CAPT embedded solutions with the goals of high development productivity and running efficiency performance. It is conducted by the Algorithm-Architecture Adequation (AAA) methodology, first introduced by the AOSTE team of INRIA (French National Institute for computer science and applied mathematics) [26]. The key feature of AAA is the ability to rapidly prototype complex real-time embedded applications based on automatic code generation. The concerned algorithm and its hardware architecture are studied simultaneously within a Software/Hardware co-design framework, which allows an embedded implementation optimized both in the algorithm and hardware level.

Concerning the pronunciation diagnosis algorithm, a high-accuracy and low-consumption classifier is desired to balance accuracy and efficient performance. A recently-proposed heterogeneous machine learning CAPT framework [27] is therefore selected. For the reason that the phoneme utterances are made from the base vibrations of vocal cords through resonance chambers (buccal, nasal, and pharyngeal cavities) [28,29], the predictors of the phoneme feature vectors are highly probably collinear, resulting in a multicollinearity problem. The multicollinearity problem means that one of the predictor variables in a classification model can be linearly predicted from the others with a substantial degree of accuracy. Although it is usually difficult to figure out a precise mathematical model to explain the fundamentals ofal least squar a certain pattern recognition problem, research indicates that suppressing the multicollinearity by using some suitable method is helpful to improve the pattern discriminability [30–32]. Yanjing et al. [27] estimate the condition indices of a French phoneme utterance spectrum set, and 87.27% of its elements exceed 10. This means that the predictor dependencies start to affect the regression estimates [33]. The framework of this work first suppresses the multicollinearity among the predictors of the phoneme sample vectors by using the partial least square (PLS) regression algorithm and then classifies them via soft-margin SVMs. Considering that FPGA is one of the most commonly used embedded devices for its benefits in terms of running cost, power consumption, and flexibility [34–41], our team therefore prototyped it as a hardware core within the register-transfer level for FPGA-available solutions.

The main challenge of this project is how to implement the desired algorithm behavior at the register transfer level efficiently with acceptable running efficiency and resource cost performance. For the purpose of high development productivity and maintainability, high-level synthesis techniques are developed. The work of E. Manor [42] demonstrates that this method is an important and effective solution for fast embedded prototyping with efficient performance. This work uses a recently-proposed very high-level synthesis (VHLS) based SW/HW co-design flow [43,44] to facilitate the implementing process from Matlab to RTL. Moreover, different interface and parallel optimizations are made to accelerate the implementations. The evaluation experiment in this paper is conducted using a data set including 35 phonemes $\times 6$ sessions $\times 23$ persons $= 4830$ samples. The experiment results show that the outputs of the final RTL implementation are exactly the same as its Matlab prototype, implying that the Matlab-to-RTL synthesis process of this work is reliable. Comparing to the PLS regressor, SVMs, and deep neural network models, the proposed method achieves the lowest diagnostic error rate in the experiment of this paper. Additionally, the hardware performance evaluations of the RTL implementation indicate that the optimizations used in this paper achieve a speed up of $10.89\times$ relative to that of the CPU.

The main novelties of this work are summarized as follows:

(a) An FPGA-suitable CAPT framework is conceived and trained, in which the phoneme pronunciation diagnostic algorithm is based on the partial least squares regression method and an improved support vector machine, so that it could raise the accuracy performance of the framework by suppressing the collinearity problem among the predictors.

(b) The phoneme diagnostic core is implemented at the register-transfer level (RTL) via the recently-proposed Matlab-to-RTL SW/HW co-design flow for the purpose of high development productivity and maintainability. The implementation is further accelerated at the instruction-level, and a speedup of $10.89\times$ is achieved relative to its CPU implementation.

(c) The proposed RTL implementation of the CAPT framework is functionally verified and evaluated by using a French phoneme utterance database, demonstrating its application values.

The remainder of this paper is organized as follows: Section 2 describes the proposed embedded CAPT framework and explains how it is trained; Section 3 presents the implementation and optimization processes of the proposed CAPT framework; Section 4 analyzes the evaluation experiment results; and finally, Section 5 gives the final conclusion of this work.

## 2. Architecture of the CAPT Framework

The overall framework of the desired French phoneme utterance detectors is shown in Figure 1. Users utter the phoneme to learn it and record it as the input of the system. According to Figure 1a, the normalized frequency spectrum of the utterance waveform $\boldsymbol{x}$ is assigned to the detector as the training or testing predictor vector. Figure 1b zooms into the architecture of the detector unit, which is implemented as an Intellectual Property (IP) core in this paper. This architecture is a 2-layer network whose output $y$ can be mathematically described as

$$y = \delta^{(2)}(h^{(2)}(\boldsymbol{\delta}^{(1)}(h^{(1)}(\boldsymbol{x}^{(1)})))) \tag{1}$$

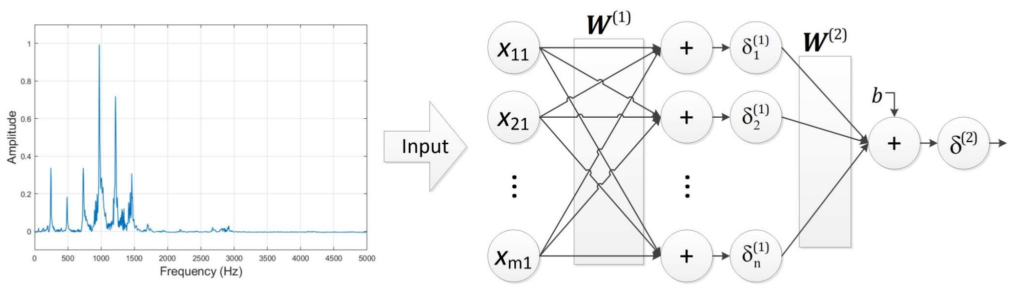
where $\delta^{(1)}$ and $\delta^{(2)}$ are two activation function sets. $h^{(1)}$ and $h^{(2)}$ are the propagation functions of the first and second layers expressed as

$$h^{(1)}(\boldsymbol{x}^{(1)}) = \boldsymbol{x}^{(1)} \times \boldsymbol{W}^{(1)} \tag{2}$$

and

$$h^{(2)}(\boldsymbol{x}^{(2)}) = \boldsymbol{x}^{(2)} \times \boldsymbol{W}^{(2)} + b \tag{3}$$

$\boldsymbol{x}^{(1)} =< x_{11}, x_{21}, \ldots, x_{m1} >$ ($m$ is the vector size and set as 16,384 in this paper) is the input of the detector to which the predictor vector $\boldsymbol{x}$ is assigned directly. $\boldsymbol{W}^{(1)}$ and $\boldsymbol{W}^{(2)}$ are the coefficient matrices of the two layers, respectively. Their sizes are $m$-by-$n$ and $n$-by-1, where $n = 35$ is the phoneme number of the French language. $b$ is the bias value of the second layer. $\boldsymbol{x}^{(2)}$ is the output of the first activation function set $\boldsymbol{\delta}^{(1)}$, whose element functions are rectified linear units (ReLU). For the second layer, the sigmoid function is used to its output as the activation function in order to constrain $y$ into a reasonable range from 0 to 1.



(a) Normalized utterance frequency spectrum (b) Detector architecture

**Figure 1.** Architecture of the phoneme utterance detectors: $\boldsymbol{x}^{(1)} =< x_{11}, x_{22}, \ldots, x_{m1} >$, and $m$ is the size of the input sample vector $\boldsymbol{x}^{(1)}$.

The decision of the system is made by comparing the output of the detector $y$, which is the diagnosis score corresponding to the utterance quality, with a threshold $\eta$ to feedback the diagnosis result. This work trains the detectors through a heterogeneous process presented in [27]. It consists of partial least square (PLS) regression and soft-margin support vector machines.

## *2.1. Training Method of Layer 1*

The diagnosing ability of the design is impacted by the multicollinearity problem among the utterance sample predictors; the partial least square (PLS) regression method is therefore applied to train the feature extraction layer of the French phoneme utterance detectors. PLS is a common class of methods for modeling relations between sets of observed variables by means of latent variables. The underlying assumption is that the observed data is generated by a system or process that is driven by a small number of latent (not directly observed or measured) variables. Its goal is to maximize the covariance between the two parts of a paired data set, even though those two parts are in different spaces. That implies that PLS regression can overcome the multicollinearity problem by modeling the relationships between the predictors. In the case of this paper, we train the first layer of the detector to extract the PLS feature of the samples to facilitate the classifying task of the second layer.

As presented in [32], let $\boldsymbol{X}$ and $\boldsymbol{Y}$ be two matrices whose rows are the predictor vectors $\boldsymbol{x}_i$ and their responses $\boldsymbol{y}_i$ corresponding to the $i$-th sample. According to the nonlinear iterative partial least squares algorithm [31,45], the optimizing problem of PLS regression is to search for some projection directions that maximizes the covariance of the training and response matrices:

$$\max_{\boldsymbol{w}_x, \boldsymbol{w}_y : ||\boldsymbol{w}_x|| = ||\boldsymbol{w}_y|| = 1} \mathbb{C}(\boldsymbol{w}_x, \boldsymbol{w}_y) = \max_{\boldsymbol{w}_x, \boldsymbol{w}_y : ||\boldsymbol{w}_x|| = ||\boldsymbol{w}_y|| = 1} \frac{1}{m} \boldsymbol{w}_x^{\mathrm{T}} \boldsymbol{X}^{\mathrm{T}} \boldsymbol{Y} \boldsymbol{w}_y \tag{4}$$

where $N$ is the number of training samples, $\boldsymbol{w}_x$ and $\boldsymbol{w}_y$ are two unit vectors corresponding to the projection directions. The directions that solve (4) are the first singular vectors $w_x = u_1$ and $w_y = v_1$ of the singular value decomposition of $\mathbb{C}_{xy}$

$$\mathbb{C}_{xy} = U\Sigma V^{\mathrm{T}} \tag{5}$$

where the value of the covariance is given by the corresponding singular value $\sigma_1$. In this paper we apply the same data projecting strategy through deflation in order to obtain multiple projecting direction.

The PLS regression algorithm is programmatically described in Algorithm 1. The inner loop computes the first singular value iteratively, which results in $u_i$ converging to the first right singular vector $Y^T X_i$. Next, the deflation of $X_i$ is computed. Finally, the regression coefficients $\theta_b$ is given by $\boldsymbol{W}^{(1)} = \tilde{\boldsymbol{U}}(\boldsymbol{p}^{\mathrm{T}}\tilde{\boldsymbol{U}})^{-1}\boldsymbol{C}^{\mathrm{T}}$, where $\boldsymbol{C}$ is a matrix with columns $c_i = \boldsymbol{Y}^{\mathrm{T}}\boldsymbol{X}_i u_i/(u_i^{\mathrm{T}}\boldsymbol{X}_i^{\mathrm{T}}\boldsymbol{X}_i u_i)$ [46].

**Algorithm 1** Pseudocode of PLS regression algorithm

**Input:** training matrix $\boldsymbol{X}$, response variables $\boldsymbol{Y}$, projection direction number $k$

**Output:** regression coefficients $\boldsymbol{W}^{(1)}$

```
1:  initialization
2:  for $i = 1, 2, \ldots, k$ do
3:      $u_i \leftarrow$ first column of $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{Y}$
4:      $u_i \leftarrow u_i / ||u_i||$
5:      repeat
6:          $u_i \leftarrow \boldsymbol{X}_i^{\mathrm{T}}\boldsymbol{Y}\boldsymbol{Y}^{\mathrm{T}}\boldsymbol{X}_i u_i$
7:          $u_i \leftarrow u_i / ||u_i||$
8:      until convergence
9:      $p_i \leftarrow \boldsymbol{X}_i^{\mathrm{T}}\boldsymbol{X}_i u_i/(u_i^{\mathrm{T}}\boldsymbol{X}_i^{\mathrm{T}}\boldsymbol{X}_i u_i)$
10:     $c_i \leftarrow \boldsymbol{Y}^{\mathrm{T}}\boldsymbol{X}_i u_i/(u_i^{\mathrm{T}}\boldsymbol{X}_i^{\mathrm{T}}\boldsymbol{X}_i u_i)$
11:     $\boldsymbol{X}_{i+1} \leftarrow \boldsymbol{X}_i(I - u_i p_i^{\mathrm{T}})$
12: end for
13: $\boldsymbol{W}^{(1)} = \tilde{\boldsymbol{U}}(\boldsymbol{p}^{\mathrm{T}}\tilde{\boldsymbol{U}})^{-1}\boldsymbol{C}^{\mathrm{T}}$
```

### 2.2. Training Method of Layer 2

The second layer of the detector is trained by using soft-margin SVMs [47]. SVM is a type of binary classifier that has been widely used [10,47–49] in speech processing. Classical SVMs build the classifier by searching for some hyperplane $(\boldsymbol{W}^{(2)}, b)$ that maximizes the margin between the two target clusters (correct pronunciations or not). This method classifies the utterance samples with a "hard margin" determined by support vectors, which may result in an over-fitting problem. For this issue, based on the SVM model of this paper (see (3)), we propose to use soft-margin SVMs to build the classifier by searching for some hyperplane $(\boldsymbol{W}^{(2)}, b)$ that maximizes the soft margin between the two target clusters:

$$\min_{\boldsymbol{W}^{(2)},b} \frac{1}{2}||\boldsymbol{W}^{(2)}||^2 + C\sum_{i=1}^{N} J_\varepsilon(h^{(2)}(\boldsymbol{x}_i^{(2)}) - y_i^{(2)}) \tag{6}$$

where $\boldsymbol{x}_i^{(2)}$ is the $i$-th predictor vector used to train the second layer, and $C$ is the regularization constant. $J_\varepsilon$ is the insensitive loss function:

$$J_\varepsilon(\boldsymbol{z}) = \begin{cases} 0 & \text{if } |\boldsymbol{z}| \leq \varepsilon \\ |\boldsymbol{z}| - \varepsilon & \text{otherwise} \end{cases} \tag{7}$$

where $\varepsilon$ is the maximum error between the prediction results and the corresponding labels. The problem above can be solved by using the lagrange multiplier method. We introduce two slack variables $\xi_i$ and $\xi_i'$ that correspond to the dissatisfaction degree with the margin constraint, so that

$$\begin{aligned} &\min_{\boldsymbol{W}^{(2)},b,\xi_i,\xi_i'} \frac{1}{2}||\boldsymbol{W}^{(2)}||^2 + C\sum_{i=1}^{N}(\xi_i + \xi_i') \\ &\text{s.t. } h^{(2)}(\boldsymbol{x}_i^{(2)}) - y_i^{(2)} \leq \varepsilon + \xi_i \\ &\quad y_i^{(2)} - h^{(2)}(\boldsymbol{x}_i^{(2)}) \leq \varepsilon + \xi_i' \\ &\quad \xi_i \geq 0 \\ &\quad \xi_i' \geq 0 \end{aligned} \tag{8}$$

with

$$i = 1, 2, \ldots, N$$

The lagrange function of (6) $\mathfrak{L}$ therefor can be written as

$$\begin{aligned} \mathfrak{L}(\boldsymbol{W}^{(2)}, b, \boldsymbol{\alpha}, \boldsymbol{\alpha}', \boldsymbol{\xi}, \boldsymbol{\xi}', \boldsymbol{\mu}, \boldsymbol{\mu}') = {} & \frac{1}{2}||\boldsymbol{W}^{(2)}||^2 + C\sum_{i=1}^{N}(\xi_i + \xi_i') - \sum_{i=1}^{N}\mu_i\xi_i - \sum_{i=1}^{N}\mu_i'\xi_i' \\ & + \sum_{i=1}^{N}\alpha_i(h^{(2)}(\boldsymbol{x}_i^{(2)}) - y_i^{(2)} - \varepsilon - \xi_i) \\ & + \sum_{i=1}^{N}\alpha_i'(y_i^{(2)} - h^{(2)}(\boldsymbol{x}_i^{(2)}) - \varepsilon - \xi_i') \end{aligned} \tag{9}$$

where $\xi_i$ and $\xi_i'$ are the slack variables. $\mu_i \geq 0$, $\mu_i' \geq 0$, $\alpha_i \geq 0$ and $\alpha_i' \geq 0$, which correspond to the columns of $\boldsymbol{\mu}$, $\boldsymbol{\mu}'$, $\boldsymbol{\alpha}$ and $\boldsymbol{\alpha}'$, are the lagrange multipliers and can be solved by building the dual problem of (8) with the Karush-Kuhn-Tucher constraints [27]. The desired coefficient matrix $\boldsymbol{W}^{(2)}$ of the second layer are obtained by computing the partial derivatives of (9) with respects to $\boldsymbol{W}^{(2)}$, $b$, $\xi_i$ and $\xi_i'$. The final bias $b$ is

$$b = \frac{1}{N}\sum_{i=1}^{N} b_i \tag{10}$$

with

$$b_i = \begin{cases} y_i^{(2)} + \varepsilon - \sum_j^N(\alpha_j' - \alpha_j)\boldsymbol{x}_i^{(2)}{\boldsymbol{x}_j^{(2)}}^{\mathrm{T}} & \text{if } 0 < \alpha_i < C \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

where $b_i$ is the bias value corresponding to $(\boldsymbol{x}_i^{(2)}, y_i^{(2)})$.

## 3. Prototyping of the Proposed CAPT Framework

This paper prototypes the proposed CAPT framework by using a VHLS-based SW/HW co-design flow. As shown in Figure 2, this workflow allows one to synthesize the algorithm behavior from a high-abstract program language level (Matlab) down to low ones (register transfer languages) via intermediate C++ code. More precisely, the algorithm behavior is specified in Matlab, then automatically transformed to intermediate C++ code by using Matlab Coder. The generated C++ function is further verified and optimized manually in

GCC. Finally, the desired RTL implementation is generated and evaluated in Vivado HLS (formerly AutoPilot from AutoESL) [50]. The methodology of this work was approved in writing by the Ethics Committee of the Foreign Language College, Capital University of Economics and Business.

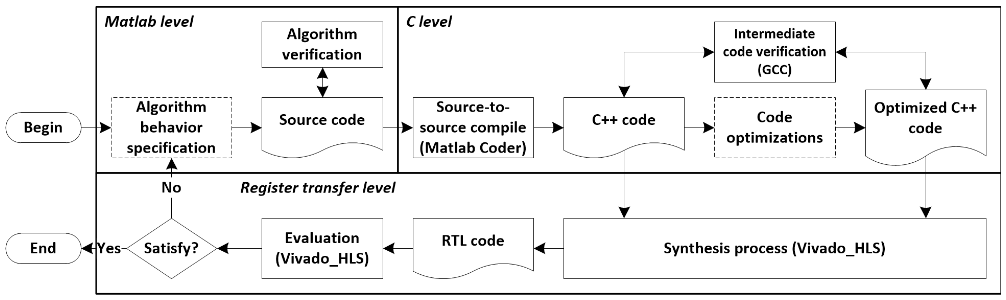


**Figure 2.** VHLS based SW/HW co-design flow: the full- and dotted-line blocks represent the automatic and manual development cycles, respectively.

## 3.1. Original Implementation

Considering that the diagnosing tasks are activated when the interested signals are segmented and pre-processed, the proposed CAPT detector is prototyped as a slave IP core, allowing a master module to invoke it at any time. In order to facilitate the updating of the network parameters, parameter ports are designed. Moreover, the data ports should be parallelizable, allowing parallelizing the computing by accessing multiple data sets simultaneously. The final interface protocol is shown in Table 1. It includes a group of logical control signals ($CLK$, $RST$, $START$, $DONE$, $IDLE$, $READY$, and $RETURN$) and the data ports ($x\_*$, $W1\_*$, $W2\_*$, $B$ and $ETA$). The parameter ports are implemented in the classical memory protocol, allowing the core to access the external memory with an index when needed. $x\_*$ is the utterance sample to be diagnosed, and its size is determined by the sampling precision of the system. In the case of this paper, we set $\overline{n}$ as 14, allowing a sampling frequency of $2^{\overline{n}} = 16384$ Hz. $W1\_*$ and $W2\_*$ are the parameter matrices of the first and second layers of the proposed framework, respectively. The size of $W2\_*$ is set as $2^6 = 64$, and $W1\_*$ is therefore a $2^{\overline{n}+6} = 1048579$-element array, which can cover the 35 phonemes in the case of this paper. If desired, the data and parameter ports can be expanded to accelerate the processing speed by communicating parallelly. $B$ is the bias value of the second layer. $ETA$ is the threshold value of the decision cycle. If the diagnosis result is positive, $RETURN$ output *true*, otherwise *false* when negative.

Algorithm 2 shows the Matlab pseudocode of the proposed CAPT implementation. Because Matlab provides a vector-available programming environment, the target behavior can be efficiently described. The algorithm's behavior starts the diagnostic processing after parameter initializations. Lines 2 and 4 correspond to the regression operations of the first and second layers (see (2) and (3)), whereas Lines 3 and 5 correspond to the ReLU and sigmoid active functions. The final decision is made in Line 6. The operator ".$\times$" returns a matrix whose elements are the products of the corresponding elements of the two input matrices.

Next, we transform the algorithm behavior from Matlab into C++ code via the source-to-source compiler Matlab Coder. For the reason that C++ is only available for scalar variables and operations, the matrix computations in Matlab are mapped into loops line by line. The pseudocode of the generated C++ behavior is shown in Algorithm 3. The first nest loop (Lines 2–8) computes the PLS regressions (see (2)) with the help of a newly-allocated register. The second loop with a "if" body (Lines 9–15) corresponds to the ReLU function. Lines 17–20 compute the SVM regressions (see (3)). Here, the buffer of the first loop is reused to save the hardware sources. Line 21 is the sigmoid function, and the decision is made within Lines 22–26.

From this point, we can start to synthesize the C++ code down to the register level through the HLS process. HLS extracts the source code as a control-and-datapath flow graph (CDFG) [51] and then represents it as a finite state machine (FSM) [52]. Figure 3 shows the diagram of the extracted FSM, in which each node is a state including a series of subsequent operations, and the arrows are the operating orders. $L*$ is the line number of Algorithm 3. As presented in [44], this method allows us to formally implement it at RTL in a standardized way.

**Table 1.** Interface protocol of the original CAPT IP core.

| RTL Ports | I/O | Bits | Protocol |
|---|---|---|---|
| *CLK* | I | 1 | ap_ctrl_hs |
| *RST* | I | 1 | ap_ctrl_hs |
| *START* | I | 1 | ap_ctrl_hs |
| *DONE* | O | 1 | ap_ctrl_hs |
| *IDLE* | O | 1 | ap_ctrl_hs |
| *READY* | O | 1 | ap_ctrl_hs |
| *RETURN* | O | 1 | ap_ctrl_hs |
| *x_ADDRESS* | O | $2^{\overline{n}}$ ($\overline{n} \in \mathbb{N}^+$) | ap_memory |
| *x_CE* | O | 1 | ap_memory |
| *x_Q* | I | 32 | ap_memory |
| *W1_ADDRESS* | O | $2^{\overline{n}+6}$ ($\overline{n} \in \mathbb{N}^+$) | ap_memory |
| *W1_CE* | O | 1 | ap_memory |
| *W1_Q* | I | 32 | ap_memory |
| *W2_ADDRESS* | O | $2^6$ | ap_memory |
| *W2_CE* | O | 1 | ap_memory |
| *W2_Q* | I | 32 | ap_memory |
| *B* | I | 32 | ap_none |
| *ETA* | I | 32 | ap_none |

**Algorithm 2** Pseudocode of the original CAPT implementation

```
Input: utterance sample $\boldsymbol{x}$, coefficients $\mathbf{W1}$, $\mathbf{W2}$, $B$ and $ETA$
Output: decision $RETURN$
1: Initialization
2: $\boldsymbol{h1} \leftarrow \boldsymbol{x} \times \mathbf{W1}$
3: $\boldsymbol{\delta1} \leftarrow (\boldsymbol{h1} > 0). \times \boldsymbol{h1}$
4: $h2 \leftarrow \boldsymbol{\delta1} \times \mathbf{W2} + B$
5: $\delta2 \leftarrow 1/(1 + \exp(-h2))$
6: $RETURN \leftarrow \delta2 \geq ETA$
```

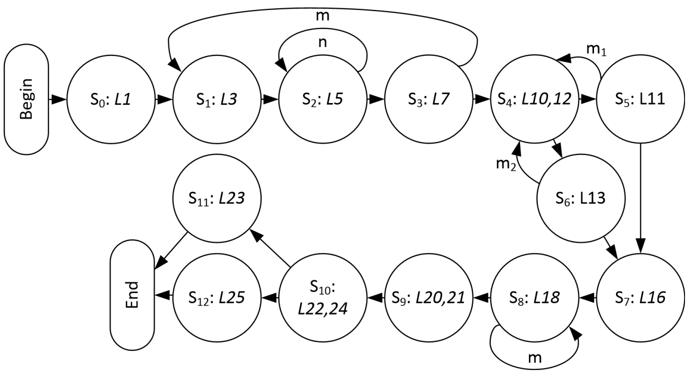


**Figure 3.** Diagram of the finite state machine of the original C++ implementation: $S*$ is the state of identification, $L*$ is the line number of Algorithm 3, $n$ and $m = m_1 + m_2$ are the iteration numbers.

**Algorithm 3** Pseudocode of the C++ CAPT implementation

**Input:** utterance sample $\boldsymbol{x}[m]$, coefficients $\mathbf{W1}[n \times m]$, $\mathbf{W2}[n]$, $B$ and $ETA$

**Output:** decision $RETURN$

```
1: Initialization
2: for $i \in \{1, 2, \ldots, m\}$ do
3:     $reg \leftarrow 0$
4:     for $j \in \{1, 2, \ldots, n\}$ do
5:         $reg \leftarrow reg + \boldsymbol{x}(j) \times \mathbf{W1}(i, j)$
6:     end for
7:     $\boldsymbol{h1}(i, j) = reg$
8: end for
9: for $i \in \{1, 2, \ldots, m\}$ do
10:     if $\boldsymbol{h1}(i) \leq 0$ then
11:         $\boldsymbol{\delta 1}(i) \leftarrow 0$
12:     else
13:         $\boldsymbol{\delta 1}(i) \leftarrow \boldsymbol{h1}(i)$
14:     end if
15: end for
16: $reg \leftarrow 0$
17: for $i \in \{1, 2, \ldots, m\}$ do
18:     $reg \leftarrow reg + \boldsymbol{\delta 1}(i) \times \mathbf{W2}(i)$
19: end for
20: $h2 \leftarrow reg + B$
21: $\delta 2 \leftarrow 1/(1 + \exp(-h2))$
22: if $\delta 2 \geq ETA$ then
23:     $RETURN \leftarrow true$
24: else
25:     $RETURN \leftarrow false$
26: end if
```

## *3.2. Optimizations*

Automatically synthesizing the behavior from high to low abstract levels allows fast algorithm prototyping, but there are still performance gaps between it and manual implementations in terms of time control, execution speed, consumption, and other factors [53,54]. As indicated in ref. [52], the quality of the HLS-based implementations is impacted by the following three factors: high-level description of language, optimization forms, and applying orders of optimization forms [55,56]. In the case of this paper, multiple optimization forms are made successively at the interface, loop, and instruction levels in order to accelerate the implementation by improving the parallelism of the code.

According to the estimations, the first loop nest costs almost all the clock cycles in the original implementation (7454790 vs. 7455344 cycles), so the main goal of the optimization work is to accelerate this part. First, the memory ports of the target core are optimized in order to mitigate access conflicts and reduce consumption due to the logic controls. According to Algorithm 3, $\boldsymbol{x}$ and $\mathbf{W1}$ are the frequently-activated ports in the first loop nest, and this will lead to access conflicts when parallelizing. We therefore partition the related arrays from a single to multiple ones to expand the bus width. An index $D_{\text{opt}} = 4, 8, 16, \ldots$ is defined to present the optimization depth. The optimized memory port protocol is shown in Table 2, demonstrating that the band of the two ports is multiplied by $D_{\text{opt}}\times$. With Vivado HLS, this optimization is made by inserting the directives of *array_partition* into the code (see Lines 1 and 2 of Algorithm 4). The option *cyclic* creates smaller arrays by interleaving elements from the original array. The array is partitioned cyclically by putting one element into each new array before coming back to the first array to repeat the cycle until the array is fully partitioned. The option $factor$ specifies the number of smaller arrays that will be created.

**Table 2.** $x$ and $W1$ ports of the optimized CAPT IP core: $D_{\text{opt}}$ is the optimization depth index.

| RTL Ports | I/O | Bits | Protocol |
|---|---|---|---|
| … | | | |
| $x\_\{1,2,\ldots,D_{\text{opt}}\}\_ADDRESS$ | O | $2^{\overline{n}}$ ($\overline{n} \in \mathbb{N}^+$) | ap_memory |
| $x\_\{1,2,\ldots,D_{\text{opt}}\}\_CE$ | O | 1 | ap_memory |
| $x\_\{1,2,\ldots,D_{\text{opt}}\}\_Q$ | I | 32 | ap_memory |
| $W1\_\{1,2,\ldots,D_{\text{opt}}\}\_ADDRESS$ | O | $2^{\overline{n}+6}$ ($\overline{n} \in \mathbb{N}^+$) | ap_memory |
| $W1\_\{1,2,\ldots,D_{\text{opt}}\}\_CE$ | O | 1 | ap_memory |
| $W1\_\{1,2,\ldots,D_{\text{opt}}\}\_Q$ | I | 32 | ap_memory |
| … | | | |

**Algorithm 4** Pseudocode of the Optimized CAPT implementation at $D_{opt}$

```
Input: utterance sample $\boldsymbol{x}[m]$, coefficients $\mathbf{W1}[n \times m]$, $\mathbf{W2}[n]$, $B$ and $ETA$
Output: decision $RETURN$
1: #pragma HLS ARRAY_PARTITION variable=$\boldsymbol{x}$ cyclic factor=$D_{\text{opt}}$ dim=1
2: #pragma HLS ARRAY_PARTITION variable=$\mathbf{W1}$ cyclic factor=$D_{\text{opt}}$ dim=2
3: Initialization
4: for $i \in \{0,1,,2\ldots,n-1\}$ do
5:     for $j \in \{0,1,2,\ldots,m/D_{\text{opt}}-1\}$ do
6:         #pragma HLS OCCURRENCE
7:         $ind\_\{1,2,\ldots,D_{\text{opt}}\} \leftarrow \{j \times D_{\text{opt}}, j \times D_{\text{opt}}+1,\ldots,j \times D_{\text{opt}}+D_{\text{opt}}-1\}$
8:         $reg\_\{1,2,\ldots,D_{\text{opt}}\} \leftarrow \boldsymbol{x}(ind\_\{1,2,\ldots,D_{\text{opt}}\}) \times \mathbf{W1}(i, ind\_\{1,2,\ldots,D_{\text{opt}}\})$
9:         $\boldsymbol{h1}(i) \leftarrow \sum_{k=1}^{D_{\text{opt}}}(reg\_\{k\})$
10:    end for
11: end for
12: for $i \in \{0,1,2,\ldots,n\}$ do
13:    #pragma HLS PIPELINE
14:    if $\boldsymbol{h1}(i) \geq 0$ then
15:        $reg \leftarrow reg + \boldsymbol{h1}(i) \times \mathbf{W2}(i)$
16:    end if
17: end for
18: $h2 \leftarrow reg + B$
19: $\delta 2 \leftarrow 1/(1+\exp(-h2))$
20: if $\delta 2 \geq ETA$ then
21:    $RETURN \leftarrow true$
22: else
23:    $RETURN \leftarrow false$
24: end if
```

Next, the FSM control flow is simplified via loop manipulations. As shown in Algorithm 4, we first move all the register allocation operations of Algorithm 3 to the beginning of the routine, which fuses States 0 and 7 into a single one during synthesis. The isolation between the second and third loops is therefore broken, so we can further simplify the control flow via loop mergence. The merged loop is shown in Lines 13–18 of

Algorithm 4, and the new loop control is shown in Figure 4, whose size is reduced from 13 down to 9 states. We insert the "pipeline" directives into the loops of the optimized code (see Lines 6 and 13 of Algorithm 4) in order to accelerate the iterations.

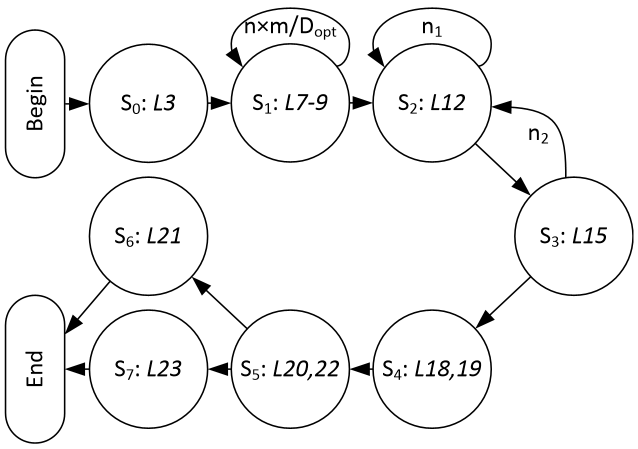
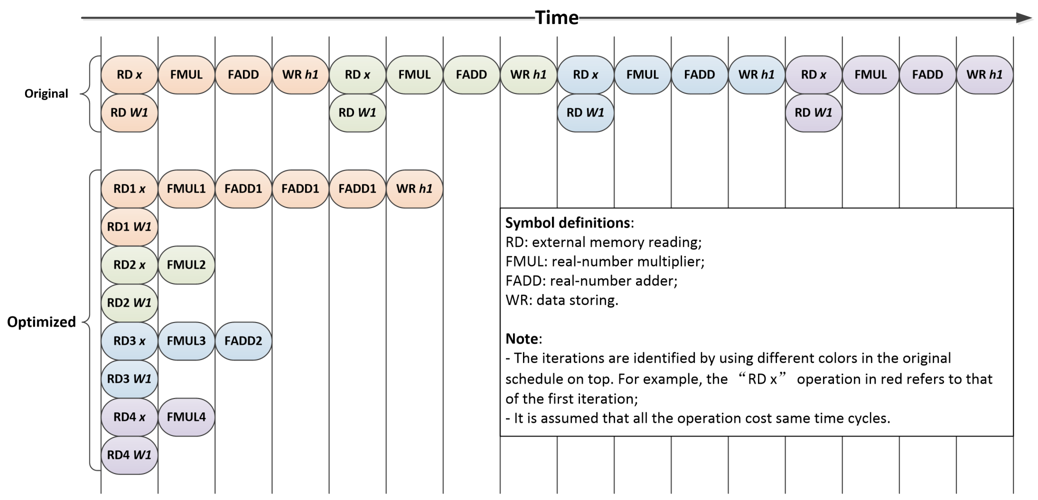


**Figure 4.** Diagram of the finite state machine of the optimized implementation: $S*$ is the state of identification, $L*$ is the line number of Algorithm 4, $n = n1 + n2$ and $m$ are the iteration numbers.

Finally, the code of Algorithm 4 is optimized at the instruction level depending on the symbolic expression manipulation strategies presented in [52]. In the original C++ code, the iterations of Line 5 have dependent relationships between each other, so they cannot be parallelized only via loop unroll. We therefore partially unroll the loops with a factor of $D_{\text{opt}}$, and then re-specify the iteration information and body manually. As shown in Lines 5–10 of Algorithm 4, the loop body is repeated $D_{\text{opt}}$ times during each iteration, and the iteration number is reduced by $D_{\text{opt}}$ accordingly. The body operations are described in polynomial form:

$$\boldsymbol{h1} = \sum_{k=i \times D_{\text{opt}}}^{(i+1) \times D_{\text{opt}}-1} \boldsymbol{x}(k) \times \mathbf{W1}(i,k) \tag{12}$$

The re-specified code avoids the dependence between the loops, enabling parallelization of the operation schedule. Figure 5 compares the original and optimized loop body operation schedules at $D_{\text{opt}} = 4$. Despite higher hardware resource consumption, the optimizations achieve a speedup of 2.67×.



**Symbol definitions**:
RD: external memory reading;
FMUL: real-number multiplier;
FADD: real-number adder;
WR: data storing.

**Note**:
- The iterations are identified by using different colors in the original schedule on top. For example, the "RD x" operation in red refers to that of the first iteration;
- It is assumed that all the operation cost same time cycles.

**Figure 5.** Comparison of the original and optimized loop body operation schedule at $D_{\text{opt}} = 4$.

It should be noted that in Figure 5, the time-cycle consumptions of each operation are equal, but the realities are not. The operators available for different devices are usually different. The speedup gain of this case $G_{\text{opt}}$ is therefore formulated as the running time ratio of the original and optimized implementations:

$$G_{\text{opt}} = \frac{T_{\text{ori}}}{T_{\text{opt}}} \tag{13}$$

with

$$\begin{aligned} T_{\text{ori}} &= D_{\text{opt}} \times (T_{\text{RD}} + T_{\text{FMUL}} + T_{\text{FADD}} + T_{\text{WR}}) + T^{\text{ctrl}}_{\text{ori}}(D_{\text{opt}}) \\ T_{\text{opt}} &= T_{\text{RD}} + T_{\text{FMUL}} + (\log_2 D_{\text{opt}} + 1) \times T_{\text{FADD}} + T_{\text{WR}} + T^{\text{ctrl}}_{\text{opt}}(D_{\text{opt}}) \end{aligned} \tag{14}$$

where $T_*$ is the time cost of the operator $* = \text{RD}, \text{FMUL}, \text{FADD}$ or WR (see Figure 5). $T^{\text{ctrl}}_{\text{ori}}$ and $T^{\text{ctrl}}_{\text{opt}}$ return the time cost values due to control operations of the original and optimized implementations at different $D_{\text{opt}}$, respectively. The consumptions of the $p$-th device element for the optimized implementation are estimated as

$$C^p_{\text{opt}} = \sum_{i=1}^{N_{\text{ope}}} C^p_i \tag{15}$$

where $N_{\text{ope}}$ is the operator number and $C^p_i$ is the consumptions of the $p$-th element for the $i$-th operator. The final goal of this optimization is to maximize $G_{\text{opt}}$ with resource constraints:

$$\begin{aligned} &\min G_{\text{opt}} \\ &\text{s.t.} \ \ \forall p, C^p_{\text{opt}} \leq C^p_{\text{av}} \end{aligned} \tag{16}$$

where $C^p_{\text{av}}$ is the number of the $p$-th available element.

## 4. Experiments

This section evaluates the proposed embedded CAPT framework. First, the function of the phoneme diagnostic IP core is verified experimentally. Next, the resource consumption and speedup gains of the optimized RTL implementation are estimated.

### *4.1. Function Verifications*

This subsection verifies the function of the VHLS implementation. The experiments are conducted by the CUEB French Phoneme Database 1.0 established by the Capital University of Economics and Business and the Institute of Acoustics CAS [27], which includes 35 phonemes $\times$ 6 sessions $\times$ 23 participants $=$ 4830 samples. The participants in the data collection are asked to read the French phonemes shown in Table 3 six times to perform six different data sessions.

**Table 3.** French phoneme table.

| **15 Vowels** | |
|---|---|
| Vowel: | [ɑ], [i], [e], [ɛ], [y], [u], [o], [ɔ], [ə], [ø], [œ] |
| Nasal vowel: | [ɑ̃], [ɔ̃], [ɛ̃], [œ̃] |
| **3 semi vowels** | [j], [w], [ɥ] |
| **17 consonants** | |
| Deaf consonants: | [p], [t], [k], [f], [s], [ʃ] |
| Sound consonants: | [b], [d], [g], [v], [z], [ʒ] |
| Lateral consonants: | [l], [r] |
| Nasal consonants: | [m], [n], [ŋ] |

Figure 6 shows the test bench of the experiments. The preprocessing cycles of the input phoneme waveform include band filtering, Fourier transforming, and normalizing. The normalized frequency spectrum is used as the predictor vector for the detectors. We verify the functions of the proposed implementation (*capt_vhls*) by comparing it with its Matlab prototype (*capt_matlab*) and C++ routine (*capt_cpp*). Moreover, in order to evaluate the diagnostic error rate of the selected algorithm, the PLS regressor (*pls_matlab*), hard-margin SVM (*hmsvm_matlab*), soft-margin SVM (*smsvm_matlab*), and deep neural network (*dnn_matlab*) are implemented in Matlab as references. All the reference implementations are specified in Table 4. All the implementations are considered to be constructed by input, hidden, and output layers. The hidden layer numbers of *capt* implementations and *dnn_matlab* are two and three, and those of the other implementations are one. The lost value and maximal iteration number are set as $1 \times 10^{-6}$ and 1000, respectively. The ReLU and sigmoid functions are used as the activate functions of *dnn_matlab*. Considering that machine learning modules necessitate high data precision and the values of speech signals usually have a large dynamic range, the data type of the proposed implementation is set to floating-point one. We also tried to use the fixed-point data to realize the same function and found that all the threshold values must be carefully set if the similar accuracy performance are desired. That is, fixed-point implementation may lead to unknown risks in the case of this paper.

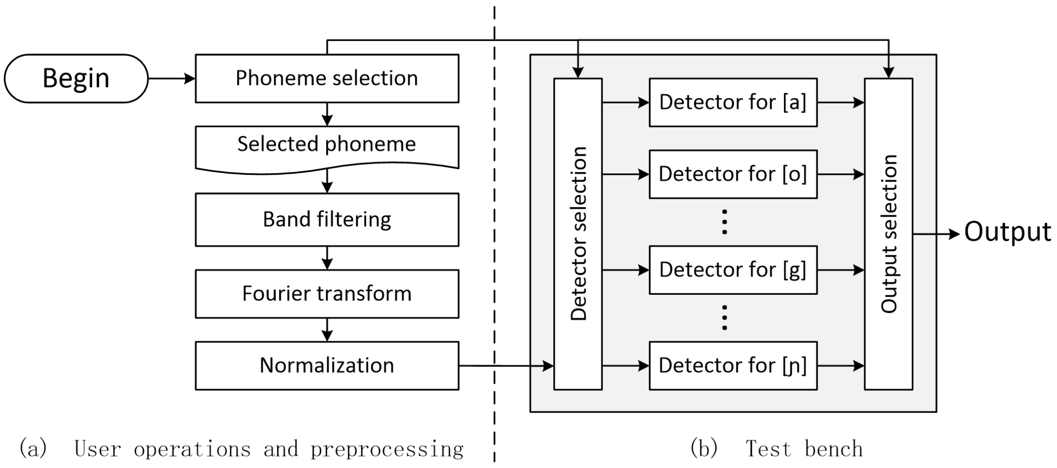


**Figure 6.** Testbench of the proposed phoneme diagnostic IP cores.

**Table 4.** Implementation specification I.

| Implementations | Layer Sizes (Input/Hidden/Output) | Descriptions |
|---|---|---|
| *pls_matlab* | 16,384/16,384/1 | Matlab implementation of PLS regressor |
| *hm_matlab* | 16,384/16,384/1 | Matlab implementation of hard-margin SVM |
| *sm_matlab* | 16,384/16,384/1 | Matlab implementation of soft-margin SVM |
| *dnn_matlab* | 16,384/2048/512/128/1 | Matlab implementation of neural network |
| *capt_matlab* | 16,384/16,384/64/1 | Matlab prototype of Algorithm 2 |
| *capt_cpp* | 16,384/16,384/64/1 | C++ routine of Algorithm 3 |
| *capt_vhls* | 16,384/16,384/64/1 | VHLS implementation of Algorithm 2 |

We divide the data set into two groups for training and testing, each with different size ratios. The size ratios are $R$ = 1:5, 2:4, 3:3, 4:2 or 5:1. The average diagnostic error rate of three measurements is used to determine the final evaluation result. Figure 7 compares the diagnostic error rates of all the implementations. First, it demonstrates that the diagnostic error rates decrease with the increases in the training set size. This is because enough training data improves the machine learning classifiers by overcoming the over-fitting problems. Second, when the different implementations have the same training set size, the implementations of $capt\_*$ achieve similar diagnostic results, indicating that the VHLS procedure used in this paper correctly synthesize the Matlab prototype to the register-transfer level automatically. Third, among the PLS-only, SVM and DNN implementations, $smsvm\_matlab$ achieves the best accuracy performance at $R$ = 1:5, 2:4 and 3:3, whereas the $dnn\_matlab$ at $R$ = 4:2 and 5:1. Comparing to them, $capt\_vhls$ improves it by 1.24%, 0.79%, 1.33%, 0.97% and 0.89%. That is, the CAPT framwork of this paper possesses the best accuracy performance in this experiment. Finally, it should be noted that it shows that $capt\_vhls$ achieves similar diagnostic error rate corresponding to $capt\_matlab$ and $capt\_cpp$, but that does not mean that the new implementation possesses lower accuracy performance. This tiny difference is caused by dividing the data set randomly into the training and testing groups, which introduces random factors in the evaluations.

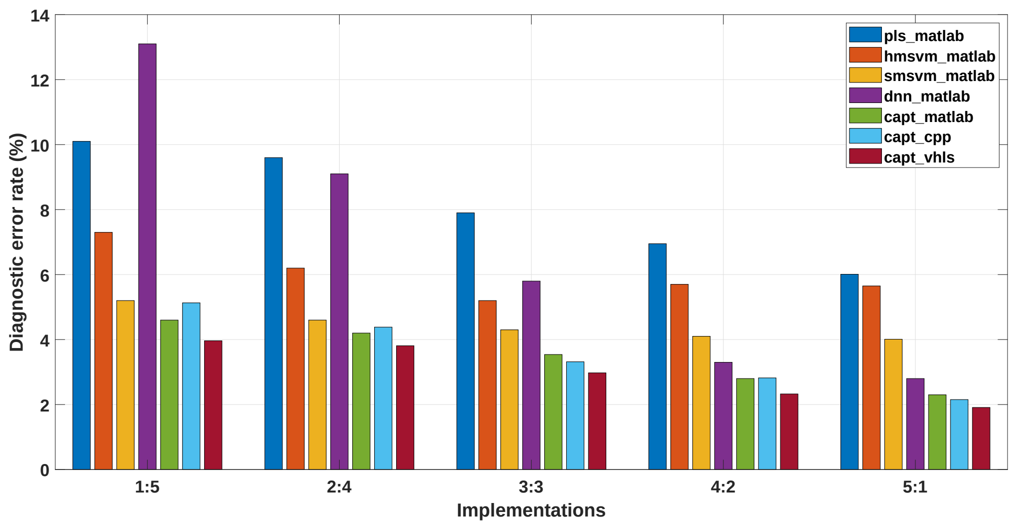


**Figure 7.** Diagnostic error rate of different implementations (see Table 4 for the definition of acronyms).

### 4.2. Hardware Resource Consumptions

This subsection analyzes the hardware resource consumptions of the original C++ implementation ($capt\_vhls\_ori$) with that of the optimized versions ($capt\_vhls\_opt\_*$, $* = 1, 2, 3, \ldots, D_{\text{opt}}$). The Zynq development and evaluation board (part: xc7z020clg484-1) of Xilinx is used as the target device. The element utilizations of different implementation versions are listed in Table 5. According to the element utilization percentage, the implementation optimizing is mainly constrained by the number of DSP48Es, so the optimizing goal of (16) can be rewritten as

$$\begin{aligned} &\min G_{\text{opt}} \\ &\text{s.t.}\ \ C_{\text{opt}}^{\text{DSP48E}} \leq C_{\text{av}}^{\text{DSP48E}} \end{aligned} \tag{17}$$

where $C_{\text{opt}}^{\text{DSP48E}}$ is the overall DSP48E consumptions, $C_{\text{av}}^{\text{DSP48E}}$ is the available DSP48E number of the target device. In this case the DSP48E elements are used to generate the single- or double-precision floating point operating instances. Table 6 specifies the DSP48E-based instances of different optimizing versions. $fadd$ and $fmul$ refer to single-precision floating point adders and multiplexers, and they are allocated for PLS regressions (Lines 7–9 in Algorithm 4). The number of these two elements are multiplied with the raising of $D_{\text{opt}}$. $dadd$ is the double-precision floating point adder, and $dexp$ outputs the value of $e$ raised to the input value's power. Despite of these operations are frequently invoked, HLS enables hardware reuse by binding one instance to multiple operations, so the resource cost can be well economized. The relationship between the DSP48E consumptions $C_{\mathrm{opt}}^{DSP48E}$ and the optimizing depth $D_{\mathrm{opt}}$ in this paper can be formulated as

$$C_{\mathrm{opt}}^{\mathrm{DSP48E}} = \frac{1}{2}C_{\mathrm{fadd}}^{\mathrm{DSP48E}}D_{\mathrm{opt}} + C_{\mathrm{fmul}}^{\mathrm{DSP48E}}D_{\mathrm{opt}} + C_{\mathrm{dadd}}^{\mathrm{DSP48E}} + C_{\mathrm{dexp}}^{\mathrm{DSP48E}} \tag{18}$$

**Table 5.** Element utilizations.

| Implementations | BRAM_18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| *capt_vhls_ori* | 1(≈0%) | 34(15%) | 6565(6%) | 9691(18%) |
| *capt_vhls_opt_4* | 1(≈0%) | 45(20%) | 7571(7%) | 11,130(20%) |
| *capt_vhls_opt_8* | 1(≈0%) | 61(27%) | 9004(8%) | 13,242(23%) |
| *capt_vhls_opt_16* | 1(≈0%) | 93(42%) | 11,867(11%) | 17,456(32%) |
| *capt_vhls_opt_32* | 1(≈0%) | 157(71%) | 17,590(16%) | 25,850(48%) |
| Available | 280 | 220 | 106,400 | 53,200 |

**Table 6.** Instance list.

| Instance | DSP48E | FF | LUT | Quantity | | | |
|---|---|---|---|---|---|---|---|
| | | | | $D_{opt} = 4$ | $D_{opt} = 8$ | $D_{opt} = 16$ | $D_{opt} = 32$ |
| *fadd* | 2 | 205 | 390 | 2 | 4 | 8 | 16 |
| *fmul* | 3 | 143 | 321 | 4 | 8 | 16 | 32 |
| *dadd* | 3 | 445 | 1149 | 1 | 1 | 1 | 1 |
| *dexp* | 26 | 1549 | 2599 | 1 | 1 | 1 | 1 |

According to Table 6, the DSP48E number costed by *fadd*, *fmul*, *dadd* and *dexp* instances are $C_{\mathrm{fadd}}^{\mathrm{DSP48E}} = 2$, $C_{\mathrm{fmul}}^{\mathrm{DSP48E}} = 3$, $C_{\mathrm{dadd}}^{\mathrm{DSP48E}} = 3$ and $C_{\mathrm{dexp}}^{\mathrm{DSP48E}} = 26$, so (18) can be simplified to $4 \times D_{\mathrm{opt}} + 29$. Considering that the $D_{\mathrm{opt}}$ must be an integer divisor of the iteration number $m = 16384$, according to (17), the optimal optimizing depth in this case is $D_{\mathrm{opt}} = 32$.

### *4.3. Running-Time Performance*

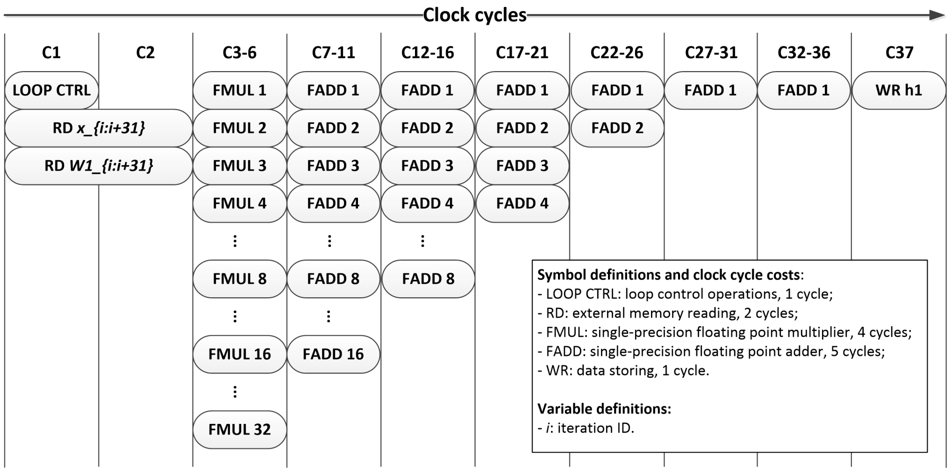
This subsection first analyzes the operation schedules of the proposed implementation, then evaluate its running-time performance by comparing it with multiple reference implementations. For the purpose of fairness, all the references are implemented in high abstract environments, including Matlab and C++. This allows us to evaluate the implementations within the similar development productivity constraint. The optimizing depth is set as $Dopt = 32$ in order to well balance the hardware consumptions and running time efficiency.

For better understanding, the optimized behavior code, Algorithm 4, is divided into the following four scopes to analyze: (a) Scope 1: Line 3, (b) Scope 2: Lines 4–11, (c) Scope 3: Lines 12–17 and (d) Scope 4: Lines 18–24. The four scopes execute in sequence, so the total latency cost $L_{\mathrm{opt}}$ can be estimated as

$$L_{\mathrm{opt}} = L_{\mathrm{s1}} + L_{\mathrm{s2}} + L_{\mathrm{s3}} + L_{\mathrm{s4}} \tag{19}$$

where $L_{\mathrm{s1}}$, $L_{\mathrm{s2}}$, $L_{\mathrm{s3}}$ and $L_{\mathrm{s4}}$ are the latency costs of the four scopes. According to the scheduling results of Vivado HLS, the first scope includes only two constant data reading accesses (*B* and *ETA*). The memory access operations cost one cycle and they are parallelly scheduled, so Scope 1 costs one cycle totally.

The second scope is a perfect loop nest corresponding to the PLS regression. The scheduling results of the inner loop body is shown in Figure 8, as well as the latency cost of each operator. All the operations are scheduled as indicated in Figure 5. The latency cost of the loop body is 37 cycles, and that of Scope 2 therefore is $L_{s2} = n \times m \times 37/D_{\text{opt}} + n \times L^{\text{loop}}_{\text{interval}} = 663110$ cycles, where $L^{\text{loop}}_{\text{interval}} = 2$ is the interval latency of the outer loop.

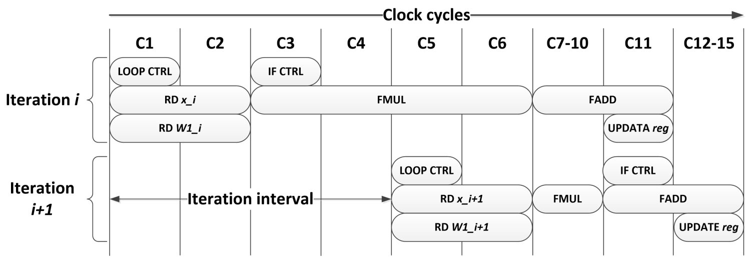


**Symbol definitions and clock cycle costs:**
- LOOP CTRL: loop control operations, 1 cycle;
- RD: external memory reading, 2 cycles;
- FMUL: single-precision floating point multiplier, 4 cycles;
- FADD: single-precision floating point adder, 5 cycles;
- WR: data storing, 1 cycle.

**Variable definitions:**
- *i*: iteration ID.

**Figure 8.** Schedule result: Lines 7–9 of Algorithm 4.

The third scope is a dependent loop whose iterations are pipelined. Figure 9 shows the scheduling result of the *i*- and $(i+1)$-th iterations of this scope. The latency cost of a single iteration is 11 cycles. Because this is a dependent loop, the iteration interval latency due to the pipelining optimization is $L^{\text{PIPELINE}}_{\text{interval}} = 4$ cycles. Thus, its total latency cost is $L_{s3} = (n-1) \times L^{\text{PIPELINE}}_{\text{interval}} + 11 = 147$ cycles.

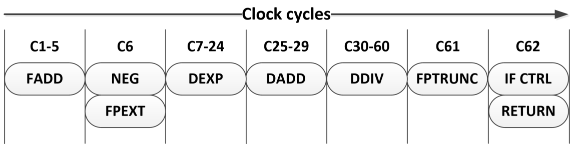


**Symbol definitions and clock cycle costs:**
- LOOP CTRL: loop control operations, 1 cycle;
- IF CTRL: switch control operations, 1 cycle;
- RD: external memory reading, 2 cycles;
- FMUL: single-precision floating point multiplier, 4 cycles;
- FADD: single-precision floating point, 5 cycles;
- UPDATE: register updating, 1 cycle.

**Variable definitions:**
- *i*: iteration ID.

**Figure 9.** Schedule result: Lines 14–16 of Algorithm 4.

The fourth scope is a series of operations executing in sequence. They cannot be parallelized due to the data dependencies. As shown in Figure 10, the latency cost of this part is $L_{s4} = 62$ cycles. It should be noted that the instance library available to the target device provide only double-precision floating point exponent and dividing operators ($dexp$ and $ddiv$), so $fpext$ and $fptrunc$ operators are necessitated for data format transforming between $float$, which is pre-defined, and $double$ data.



**Symbol definitions and clock cycle costs:**
- NEG: single-precision floating point negator, 1 cycle;
- FPEXT: single-precision floating point sign-extension, 1 cycle;
- FADD/DADD: single/double-precision floating point adder, 5 cycles;
- DDIV: double-precision floating point divider, 31 cycles;
- FPTRUNC: Single-precision floating point truncation, 1 cycle;
- IF CTRL: switch control operations, 1 cycle;
- RETURN: return the computing result, 1 cycle.

**Figure 10.** Schedule result: Lines 18–24 of Algorithm 4.

According to (19), the latency cost of this optimized implementation is $L_{\text{opt}}$ = 663,320 cycles. The running time costs of RTL implementations can be estimated as $T_{\text{opt}} = P_{\text{opt}} \times L_{\text{opt}}$, where $P_{\text{opt}}$ is the estimated clock period. It should be noted that the value of $P_{\text{opt}}$ varies with the optimizing depth $D_{\text{opt}}$. In this case, the evaluation report for Vivado HLS indicates that $P_{\text{opt}} = 8.63$ ns at $D_{\text{opt}} = 32$, so its running time is $T_{\text{opt}} = 8.63 \text{ ns} \times 663{,}320 \text{ cycles} = 5.72$ ms.

For the purpose of unbiased evaluation results, we compared the running-time performance of the final implementation with multiple references. Table 7 specifies their developing environments. $capt\_matlab$ is the very original version for algorithm verifications. $capt\_cpp$ is developed from the Matlab-to-C transforming version, which is used as the input of the high-level synthesis process. $capt\_vhls\_ori$ is generated from the $capt\_cpp$ code without any optimizations. $capt\_vhls\_opt\_D_{\text{opt}}$ ($D_{\text{opt}} = 4, 8, 16$, or 32) are the versions optimized from $capt\_vhls\_ori$ with different optimizing depths. $capt\_vhls\_opt\_32$ is the proposed CAPT prototype of this paper.

**Table 7.** Implementation specifications II.

| Implementations | Clock Period (ns) | Routines | Environments | Devices |
|---|---|---|---|---|
| $capt\_matlab$ | 0.55 | Algorithm 2 | Matlab 2017b Win10-64bit | Intel Core i7, 1.80 GHz 32 GB RAM |
| $capt\_cpp$ | 0.55 | Algorithm 3 | VS 2015 Win10-64bit | Intel Core i7, 1.80 GHz 32 GB RAM |
| $capt\_vhls\_ori$ | 8.62 | Algorithm 4 | Vivado HLS 2017 Windows 10-64bit | xc7z020clg484-1 |
| $capt\_vhls\_opt\_4$ | 8.63 | | | |
| $capt\_vhls\_opt\_8$ | 8.63 | | | |
| $capt\_vhls\_opt\_16$ | 8.63 | | | |
| $capt\_vhls\_opt\_32$ | 8.63 | | | |

Figure 11 plots the acceleration ratios over the different implementing versions, in which $capt\_matlab$ is set as the base. It illustrates that the C++ version achieves a speedup of 1.8×, whereas the original VHLS version 0.97×. This result means that synthesizing the Matlab code directly down to register-transfer levels do not obtain acceleration gains corresponding to today's commonly-used processors and developing environments. However, the optimizing methods used in this paper effectively improve the running-time efficiency of the VHLS implementations. Comparing with $capt\_vhls\_ori$, the optimizing forms made in this paper accelerate the implementations by 2.36×, 3.85×, 6.50× and 11.28× at $D_{\text{opt}} = 4, 8, 16$ and 32. Comparing with the two CPU-based implementations $capt\_matlab$ and $capt\_cpp$, the proposed CAPT prototype $capt\_vhls\_opt\_32$ achieves speedups of 10.89× and 6.02×, respectively. Meanwhile, it should be noted that the VHLS implementations can be further accelerated by raising the optimizing depth value, but that will cost more hardware resources. Within the device xc7z020cl484-1 used in this paper, we have maximized the resource cost. If desired, the implementation can be further accelerated by using a bigger device.

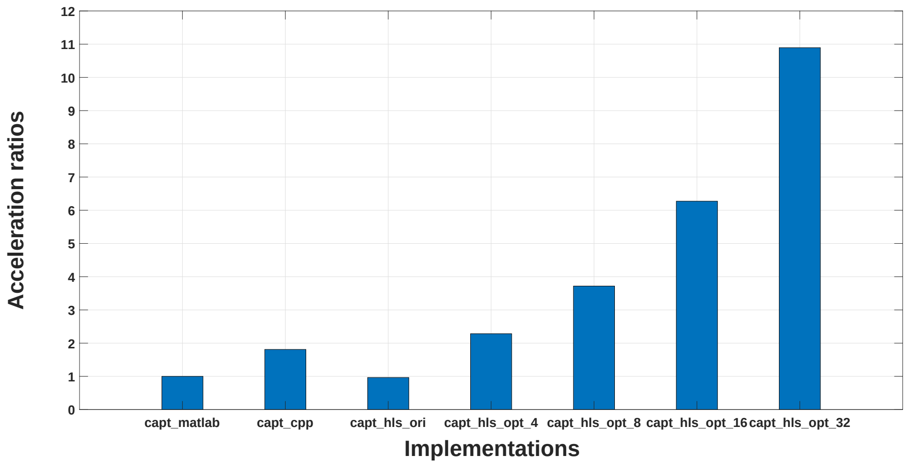


**Figure 11.** Running time comparison (see Table 7 for the definition of acronyms in this figure).

## 5. Discussions and Conclusions

This paper implements a new-developed phoneme pronunciation diagnostic framework for French CAPT modalities as a register-transfer level core. Classical machine learning networks are impacted by the multicollinearity problem among the predictors of the utterance sample vectors, the PLS algorithm is therefore applied in to the desired network as the feature extracting layer to suppress the collinearity. Next, the soft-margin SVM is used to perform the second network layer to enhance the classifying ability of the network. Experiments results demonstrate that this method possesses better accuracy performance than the state-of-the-art. Yet, we must claim that the performance of the DNN implementations are constrained by training data size, so the experiments of this paper cannot prove that the algorithm of this paper inevitably leads to the best performance. Considering that a classical DNN model include 5 layers at least (1 input, 3 hidden and 1 output layers), whereas the proposed one only 4 (1 input, 1 PLS feature extracting, 1 SVM classifying and 1 output layers), the latter is more suitable for the systems on chips.

As far to the register-transfer level implementing of the design, we prototype it via a new-proposed VHLS SW/HW co-design flow in order to facilitate the development works and maintenances. During this work, it is found that synthesizing directly the behavior from Matlab down to RTL prevents the implementation from benefiting from the running-efficiency advantages of FPGAs, a series optimizing forms are therefore made in the loop and instrument levels. The CUEB French Phoneme Database is used to evaluate the achievements of this work. The experiments results verify the basic function of the new implementation by comparing it with its Matlab and C++ implementations. The hardware evaluation experiments demonstrate that the prototype of this paper make efficient use of the given hardware resources, and achieves a speedup of 10.89×, which making better use of the hardware resources. Despite of many benefits of development productivity and easy maintenances, it should note that high level synthesis seriously constrain the performance of FPGA implementations in the terms of hardware costs and running-efficiency comparing to the low abstract level implementations. If high performance is desired, some more bottom optimizations are still required, especially when the constraints of Placing and Routing cycle is taken into account.

In the future research, we will further improve the methods of this paper. The PLS methods and hardware implementing experiences will be considered as a potential solution of sparse learning to solve the data-hungry problems, which may also benefit the embedded CAPT applications from deep learning methods. Meanwhile, there exists still some other hardware solutions worth trying, such as MicroBlaze, which may provide nice performance if well optimized.

**Author Contributions:** Conceptualization, Y.B. (Yanjing Bi) and C.L.; methodology, Y.B. (Yanjing Bi), C.L., Y.B. (Yannick Benezeth) and F.Y.; software, C.L.; validation, C.L.; formal analysis, Y.B. (Yanjing Bi) and C.L.; investigation, Y.B. (Yanjing Bi) and C.L.; resources, Y.B. (Yanjing Bi), C.L., Y.B. (Yannick Benezeth) and F.Y.; data curation, C.L.; writing-original draft preparation, Y.B. (Yanjing Bi) and C.L.; writing-review and editing, Y.B. (Yannick Benezeth) and F.Y.; project administration and funding acquisition, Y.B. (Yanjing Bi), C.L., Y.B. (Yannick Benezeth) and F.Y. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was partially funded by National Natural Science Foundation of China (ID: 62171440), Chinese Academy of Sciences and Jiangxi Provincial Social Sciences "14th Five-Year Plan" (2021) Fund Project (ID: 21YY33).

**Institutional Review Board Statement:** The study was approved by the Academic Council of School of Foreign Studies in the Capital University of Economics and Business (8 May 2023).

**Informed Consent Statement:** Informed consent was obtained from all subjects involved in the study.

**Data Availability Statement:** The data presented in this study are available in [27].

**Acknowledgments:** We would like to thank our colleague, Mooney Cormac, for his valuable help in proofreading.

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

## References

1. Golonka, E.M.; Bowles, A.R.; Frank, V.M.; Richardson, D.L.; Freynik, S. Technologies for foreign language learning: a review of technology types and their effectiveness. *Comput. Assist. Lang. Learn.* **2014**, *27*, 70–105. [CrossRef]
2. Carey, S. The Use of WebCT for a Highly Interactive Virtual Graduate Seminar. *Comput. Assist. Lang. Learn.* **1999**, *12*, 371–380. [CrossRef]
3. Bonneau, A.; Camus, M.; Laprie, Y.; Colotte, V. A computer-assisted learning of English prosody for French students. In Proceedings of the Instil/Icall Symposium NLP & Speech Technologies in Advanced Language Learning Systems, Venecia, Italia, 17–19 June 2004.
4. Zhang, L.; Zhao, Z.; Ma, C.; Shan, L.; Gao, C. End-to-End Automatic Pronunciation Error Detection Based on Improved Hybrid CTC/Attention Architecture. *Sensors* **2020**, *20*, 1809. [CrossRef] [PubMed]
5. Piotrowska, M.; Korvel, G.; Kostek, B.; Ciszewski, T.; Cyzewski, A. Machine Learning–based Analysis of English Lateral Allophones. *Int. J. Appl. Math. Comput. Sci.* **2019**, *29*, 393–405. [CrossRef]
6. Long, Z.; Li, H.; Lin, M. An adaptive unsupervised clustering of pronunciation errors for automatic pronunciation error detection. In Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012), Tsukuba, Japan, 11–15 November 2012.
7. Almajai, I.; Cox, S.; Harvey, R.; Lan, Y. Improved speaker independent lip reading using speaker adaptive training and deep neural networks. In Proceedings of the 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Shanghai, China, 20–25 March 2016; pp. 2722–2726. [CrossRef]
8. Yin, S.; Liang, W.; Liu, R. Lattice-based GOP in automatic pronunciation evaluation. In Proceedings of the 2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE), Singapore, 26–28 February 2010.
9. Brocki, U.; Marasek, K. Deep Belief Neural Networks and Bidirectional Long-Short Term Memory Hybrid for Speech Recognition. *Arch. Acoust.* **2015**, *40*, 191–195. [CrossRef]
10. Zehra, W.; Javed, A.R.; Jalil, Z.; Gadekallu, T.R.; Kahn, H.U. Cross corpus multi-lingual speech emotion recognition using ensemble learning. *Complex Intell. Syst.* **2021**, *7*, 1845–1854. [CrossRef]
11. Abdel-Hamid, O.; Mohamed, A.R.; Jiang, H.; Deng, L.; Penn, G.; Yu, D. Convolutional Neural Networks for Speech Recognition. *IEEE/ACM Trans. Audio Speech Lang. Process.* **2014**, *22*, 1533–1545. [CrossRef]
12. Gulati, A.; Qin, J.; Chiu, C.C.; Parmar, N.; Zhang, Y.; Yu, J.; Han, W.; Wang, S.; Zhang, Z.; Wu, Y. Conformer: Convolution-augmented Transformer for Speech Recognition. In Proceedings of the Interspeech 2020, Shanghai, China, 25–29 October 2020.
13. Turan, M.; Erzin, E. Improving phoneme recognition of throat microphone speech recordings using transfer learning. *Speech Commun.* **2021**, *129*, 25–32. [CrossRef]
14. Sun, Z.; Tang, P. Automatic Communication Error Detection Using Speech Recognition and Linguistic Analysis for Proactive Control of Loss of Separation. *Transp. Res. Rec.* **2021**, *2675*, 1–12. [CrossRef]
15. Badrinath, S.; Balakrishnan, H. Automatic Speech Recognition for Air Traffic Control Communications:. *Transp. Res. Rec.* **2022**, *2676*, 798–810. [CrossRef]
16. Jiang, F.; Chiba, Y.; Nose, T.; Ito, A. Automatic assessment of English proficiency for Japanese learners without reference sentences based on deep neural network acoustic models - ScienceDirect. *Speech Commun.* **2020**, *116*, 86–97.
17. Manor, E.; Greenberg, S. Custom Hardware Inference Accelerator for TensorFlow Lite for Microcontrollers. *IEEE Access* **2022**, *10*, 73484–73493. [CrossRef]
18. Silva, W.; Batista, G.C.; Saotome, O.; Oliveira, D. A Low-power Asynchronous Hardware Implementation of a Novel SVM Classifier, with an Application in a Speech Recognition System. *Microelectron. J.* **2020**, *105*, 104907.
19. Chervyakov, N.I.; Lyakhov, P.A.; Deryabin, M.A.; Nagornov, N.N.; Valuev, G.V. Residue Number System-Based Solution for Reducing the Hardware Cost of a Convolutional Neural Network. *Neurocomputing* **2020**, *407*, 439–453. [CrossRef]
20. Pardo, P.C.; Tilbrook, B.; van Ooijen, E.; Passmore, A.; Neill, C.; Jansen, P.; Sutton, A.J.; Trull, T.W. Surface ocean carbon dioxide variability in South Pacific boundary currents and Subantarctic waters. *Sci. Rep.* **2019**, *9*, 7592. [CrossRef]
21. Castro-Zunti, R.D.; Yépez, J.; Ko, S.B. License plate segmentation and recognition system using deep learning and OpenVINO. *IET Intell. Transp. Syst.* **2020**, *14*, 119–126. [CrossRef]
22. Andriyanov, N.A. Analysis of the Acceleration of Neural Networks Inference on Intel Processors Based on OpenVINO Toolkit. In Proceedings of the 2020 Systems of Signal Synchronization, Generating and Processing in Telecommunications (SYNCHROINFO), Svetlogorsk, Russia, 1–3 July 2020; pp. 1–5. [CrossRef]
23. Zunin, V.V. Intel OpenVINO Toolkit for Computer Vision: Object Detection and Semantic Segmentation. In Proceedings of the 2021 International Russian Automation Conference (RusAutoCon), Sochi, Russia, 5–11 September 2021; pp. 847–851. . [CrossRef]
24. Bernabé, S.; González, C.; Fernández, A.; Bhangale, U. Portability and Acceleration of Deep Learning Inferences to Detect Rapid Earthquake Damage From VHR Remote Sensing Images Using Intel OpenVINO Toolkit. *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.* **2021**, *14*, 6906–6915. [CrossRef]
25. Gupta, S. Real Time Face Recognition on an Edge Computing Device. In Proceedings of the ICSCA 2020: 2020 9th International Conference on Software and Computer Applications, Langkawi Malaysia, 18–21 February 2020.
26. team, A. *The AAA Methodology and SynDEx*; Technical report; INRIA Paris-Rocquencourt Research Center France: Le Chesnay-Rocquencourt, France, 2017.

27. Yanjing, B.; Chao, L.; Yannick, B.; Fan, Y. Impacts of multicollinearity on CAPT modalities: An heterogeneous machine learning framework for computer-assisted French phoneme pronunciation training. *PLoS ONE* **2021**, *16*, e0257901. [CrossRef]
28. Boersma, P. An articulatory synthesizer for the simulation of consonants. In Proceedings of the Third European Conference on Speech Communication and Technology, EUROSPEECH 1993, Berlin, Germany, 22–25 September 1993.
29. Wong, K.; Lo, W.; Meng, H. Allophonic variations in visual speech synthesis for corrective feedback in CAPT. In Proceedings of the 2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Prague, Czech Republic, 22–27 May 2011; pp. 5708–5711.
30. Nguyen, D.V.; Rocke, D.M. Tumor classification by partial least squares using microarray gene expression data. *Bioinformatics* **2002**, *18*, 39. [CrossRef]
31. Uzair, M.; Mahmood, A.; Mian, A. Hyperspectral Face Recognition With Spatiospectral Information Fusion and PLS Regression. *IEEE Trans. Image Process.* **2015**, *24*, 1127–1137. [CrossRef]
32. Li, C.; Benezeth, Y.; Nakamura, K.; Gomez, R.; Yang, F. A robust multispectral palmprint matching algorithm and its evaluation for FPGA applications. *J. Syst. Archit.* **2018**, *88*, 43–53. [CrossRef]
33. Belsley, D.A.; Kuh, E.; Welsch, R.E. *Conditioning Diagnostics: Collinearity and Weak Data in Regression*; Wiley-Interscience: Hoboken, NJ, USA, 2005. [CrossRef]
34. Musavi, S.; Chowdhry, B.; Kumar, T.; Pandey, B.; Kumar, W. IoTs Enable Active Contour Modeling Based Energy Efficient and Thermal Aware Object Tracking on FPGA. *Wirel. Pers. Commun.* **2015**, *85*, 529–543. [CrossRef]
35. Sukhwani, B.; Thoennes, M.; Min, H.; Dube, P.; Brezzo, B.; Asaad, S.; Dillenberger, D. A Hardware/Software Approach for Database Query Acceleration with FPGAs. *Int. J. Parallel Program.* **2015**, *43*, 1129–1159. [CrossRef]
36. Colodro-Conde, C.; Toledo-Moreo, F.J.; Toledo-Moreo, R.; Martínez-Álvarez, J.J.; Guerrero, J.G.; Ferrández-Vicente, J.M. Evaluation of stereo correspondence algorithms and their implementation on FPGA. *J. Syst. Archit.* **2014**, *60*, 22–31. [CrossRef]
37. Sidiropoulos, H.; Siozios, K.; Soudris, D. A novel 3-D FPGA architecture targeting communication intensive applications. *J. Syst. Archit.* **2014**, *60*, 32–39. [CrossRef]
38. Toledo-Moreo, F.J.; Martínez-Álvarez, J.J.; Garrigós-Guerrero, J.; Ferrández-Vicente, J.M. FPGA-based architecture for the real-time computation of 2-D convolution with large kernel size. *J. Syst. Archit.* **2012**, *58*, 277–285. [CrossRef]
39. Lyberis, S.; Kalokerinos, G.; Lygerakis, M.; Papaefstathiou, V.; Mavroidis, I.; Katevenis, M.; Pnevmatikatos, D.; Nikolopoulos, D.S. FPGA prototyping of emerging manycore architectures for parallel programming research using Formic boards. *J. Syst. Archit.* **2014**, *60*, 481–493. [CrossRef]
40. Li, T.; He, B.; Zheng, Y. Research and Implementation of High Computational Power for Training and Inference of Convolutional Neural Networks. *Appl. Sci.* **2023**, *13*, 1003. [CrossRef]
41. Milik, A.; Kubica, M.; Kania, D. Reconfigurable Logic Controller—Direct FPGA Synthesis Approach. *Appl. Sci.* **2021**, *11*, 8515. [CrossRef]
42. Manor, E.; Greenberg, S. Using HW/SW Codesign for Deep Neural Network Hardware Accelerator Targeting Low-Resources Embedded Processors. *IEEE Access* **2022**, *10*, 22274–22287. [CrossRef]
43. Bi, Y.; Li, C.; Yang, F. Very High Level Synthesis for image processing applications. In Proceedings of the 10th International Conference on Distributed Smart Cameras (ICDSC 2016), Paris France, 12–15 September 2016.
44. Li, C.; Bi, Y.; Marzani, F.; Yang, F. Fast FPGA prototyping for real-time image processing with very high-level synthesis. *J. Real-Time Image Process.* **2017**. [CrossRef]
45. Shawe-Taylor, J.; Cristianini, N. *Kernel Methods for Pattern Analysis*; Cambridge University Press: New York, NY, USA, 2004.
46. Wold, H. Soft modelling: The Basic Design and Some Extensions. *Systems Under Indirect Observation, Part II*; North-Holland: Amsterdam, The Netherlands, 1982; pp. 36–37.
47. Cortes, C.; Vapnik, V. Support-Vector Networks. *Mach. Learn.* **1995**, *20*, 273–297. [CrossRef]
48. Schuller, B.; Vlasenko, B.; Eyben, F.; W?llmer, M.; Stuhlsatz, A.; Wendemuth, A.; Rigoll, G. Cross-Corpus Acoustic Emotion Recognition: Variances and Strategies. *IEEE Trans. Affect. Comput.* **2010**, *1*, 119–131. [CrossRef]
49. Albornoz, E.; Milone, D. Emotion recognition in never-seen languages using a novel ensemble method with emotion profiles. *IEEE Trans. Affect. Comput.* **2017**, *8*, 43–53. [CrossRef]
50. XILINX. *Vivado Design Suite User Guide*, ug902(2012.2) ed.; XILINX: San Jose, CA, USA, 2012.
51. Daniel D., G.; Nikil D., D.; Allen C-H, W.; Steve Y-L, L. *High-Level Synthesis: Introduction to Chip and System Design*; 1st ed.; Springer: New York, NY, USA, 1992; pp. XV, 359. [CrossRef]
52. Li, C.; Bi, Y.; Benezeth, Y.; Ginhac, D.; Yang, F. High-level synthesis for FPGAs: code optimization strategies for real-time image processing. *J. Real-Time Image Process.* **2018**, *14*, 701–712. [CrossRef]
53. Rupnow, K.; Liang, Y.; Li, Y.; Min, D.; Do, M.; Chen, D. High level synthesis of stereo matching: Productivity, performance, and software constraints. In Proceedings of the 2011 International Conference on Field-Programmable Technology (FPT), New Delhi, India, 12–14 December 2011; IEEE: Piscataway, NJ, USA, 2011.
54. Liang, Y.; Rupnow, K.; Li, Y.; Min, D.; Do, M.N.; Chen, D. High-Level Synthesis: Productivity, Performance, and Software Constraints. *J. Electr. Comput. Eng.* **2012**, *2012*, 649057.

55. Cong, J.; Liu, B.; Prabhakar, R.; Zhang, P. A Study on the Impact of Compiler Optimizations on High-Level Synthesis. In *Languages and Compilers for Parallel Computing*; Kasahara, H., Kimura, K., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2013; Volume 7760, pp. 143–157. [CrossRef]
56. Huang, Q.; Lian, R.; Canis, A.; Choi, J.; Xi, R.; Calagar, N.; Brown, S.; Anderson, J. The Effect of Compiler Optimizations on High-Level Synthesis-Generated Hardware. *ACM Trans. Reconfigurable Technol. Syst.* **2015**, *8*, 14:1–14:26. [CrossRef]