A High Performance FPGA-based Accelerator Design for End-to-End Speaker Recognition System

Mingjun Jiao, Yue Li, Pengbo Dang, Wei Cao*, Lingli Wang
State Key Laboratory of ASIC and System, Fudan University
Shanghai, China
*caow@fudan.edu.cn

***Abstract*—Speaker recognition technique is significant for identification applications. X-vectors, a robust text-independent speaker recognition system, spends plenty of time on extracting voiceprint features due to massive neural network computation and scoring with all the people registered in the database to find the best match person. In this paper, an FPGA-based high-performance accelerator for this end-to-end speaker recognition system is proposed, which contains three parts: Mel Frequency Cepstral Coefficients (MFCC), time delay neural network (TDNN) and probabilistic linear discriminant analysis (PLDA) classifier. A quantitative analysis is presented to balance the bit width and the recognition accuracy. In addition, an optimization strategy to make a trade-off between the system parallelism and the FPGA resource utilization is introduced. As a comparison, the proposed accelerator running on Xilinx XCVU9P FPGA of UltraScale+ VCU118 board can achieve a peak performance of 1.067 TOP/s and 1.30×10^5 voice frames per second (vFPS) with 200MHz, which can obtain 1296× speedup compared with X-vectors software implementation running on a 2.5GHz Intel Xeon E5-2620 processor and 6.42× energy efficiency than Nvidia TITAN Xp GPU solution.**

Keywords—X-vectors, speaker recognition, time delay neural networks, pipelined architecture, end-to-end system

I. Introduction

Biometric authentication is the foundation of the future highly secure identification and personal verification. Among biometric systems, speaker recognition based on voiceprint becomes attractive due to its acceptability, reliability and ease-of-use. To capture the long dependencies in acoustic events and model the temporal dynamics in speech, an acoustic model needs to deal with long temporal contexts efficiently. Numerous algorithms have been proposed to enhance recognition accuracy. The traditional speaker recognition models based on i-vectors [1], which employ unsupervised methods, have demonstrated effectiveness. In recent years, recurrent neural networks (RNNs) [2] and long term short term memory networks (LSTMs) [3] are capable of learning the dependencies of voice signal efficiently. However, according to the analysis in [4], parallelization cannot be easily exploited for RNNs and LSTMs due to dependencies among the time-frames being processed. Thus, the time delay neural network (TDNN) with sub-sampling which is easy to explore its parallelism is employed to extract the features [4]. X-vectors [5], a speaker recognition model based on TDNN, has shown that data augmentation can improve system robustness and achieve superior accuracy. High performance and energy efficiency are very critical for the speaker recognition system because they are typically deployed in the cloud devices to handle massive data from the public security system or other forensics and surveillance applications.

In this paper, an FPGA-based accelerator for the end-to-end speaker recognition system which is based on X-vectors is designed. To the best of our knowledge, this is the first research effort in which the whole end-to-end speaker recognition system based on the neural network is accelerated on the hardware platform. The key contributions are as follows:

- An end-to-end system accelerator which utilizes the pipelined architecture is proposed to accelerate the X-vectors speaker recognition system.
- A quantitative analysis is presented to find out an appropriate bit width which not only keeps high recognition accuracy but also consumes as few resources as possible.
- An optimization strategy is introduced to determine the system parallelism and achieve high performance under the condition of limited resources.
- The hardware implementation of our speaker recognition system accelerator running on the Xilinx XCVU9P FPGA of UltraScale+ VCU118 board can achieve 1.067 TOP/s and 1.30 × 10^5 vFPS with 200MHz, which achieves 1296× speedup compared with X-vectors algorithm running on Intel Xeon E5-2620 processor and 6.42× energy efficiency compared with software solution on GTX TITAN Xp GPU.

This paper is organized as follows. Section II describes the related work. Section III introduces the speaker recognition algorithm based on X-vectors. In section IV, quantitative analysis for hardware implementation and parallelism exploration are presented. Section V describes the hardware accelerator design details. Section VI presents experimental results and comparisons between our design and CPU/GPU solutions. Section VII concludes this paper.

II. Related Work

Recently, various technologies such as support vector machine (SVM), Gaussian mixture model (GMM) and neural networks have been adopted for speaker recognition system. FPGA-based acceleration for MFCC feature extraction algorithm is proposed in [6]. A hardware and software co-design is presented to provide fast training ability while other processes such as speech preprocess, speech feature extraction and SVM voting strategy are still implemented by software in [7].

References [8] and [9] have implemented whole system acceleration based on SVM-based speaker recognition algorithm while GMM-based speaker recognition algorithm is accelerated in reference [10]. However, these previous works are all designed on an embedded platform with traditional speaker recognition algorithm like SVM or GMM.

Considering that for data centers, high throughput and energy efficiency are more important rather than real-time and low power requirements on the embedded devices, technologies oriented by metrics of high performance is needed to fully explore the performance limitation. Furthermore, given that neural networks have been widely applied in voice related applications which can bring advantages such as state-of-art recognition accuracy and better system robustness, this work employs a pipelined system structure based on TDNN to increase the throughput remarkably.

III. BACKGROUND

In this section, a typical speaker recognition flow is briefly introduced, followed by the detail description for each part in the flow.

A. Speaker Recognition Algorithm Flow

The entire flow of the speaker recognition system based on X-vectors is composed of an acoustic feature extraction module (MFCC in Fig. 1), an identification feature extraction module (TDNN in Fig. 2) and a PLDA classifier [11]. The acoustic feature extraction module extracts MFCC acoustic features from raw speakers' utterances. The deep neural network is used to extract identity characteristics called embeddings [5] from MFCC features. Speaker recognition system involves two phases namely the enrollment stage and the evaluation stage. At the time of enrollment, an embedding is generated according to the utterances for each speaker and stored as the database. The PLDA classifier is employed during the evaluation stage to enable the same-or-different speaker decisions by comparing evaluation embedding with enrollment embeddings. The detailed description of each part is introduced below.

B. Acoustic Feature Extraction Module

The acoustic feature extraction module contains a series of computation steps as shown in Fig. 1. The original raw speakers' utterances are sent to the *pre-emphasis* submodule and then each utterance is divided into numerous equal-length frames by *frame blocking* submodule. And after that, each frame needs to be handled through *Windowing*, *Fast Fourier Transformation (FFT)*, *Mel Filters Bank*, *Logarithm* and *Discrete Cosine Transformation (DCT)* submodules. The final outputs are the MFCC features.

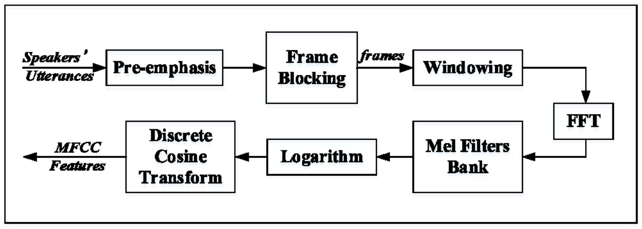

Fig. 1. Acoustic feature extraction flow

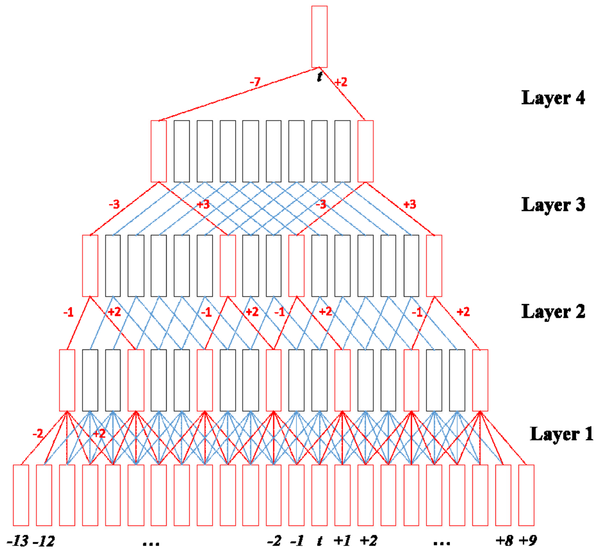

Fig. 2. TDNN structure with sub-sampling

TABLE I. CONFIGURATIONS OF DIFFERENT LAYERS IN TDNN

Layer	Layer context	Total context	Input	Output
Affine1	[t-2, t+2]	5	115	512
Affine2	{t-2, t, t+2}	9	1536	512
Affine3	{t-3, t, t+3}	15	1536	512
Affine4	{t}	15	512	512
Affine5	{t}	15	512	1500
SP	[1, T]	T	1500T	3000
Affine6	-	T	3000	512

C. Time Delay Neural Network

Compared with typical DNNs, the obvious difference is that TDNN can model long term temporal dependencies due to its time-delay characteristics so that it can achieve better accuracy than other feed-forward traditional networks in voice-related applications. A simplified TDNN structure with sub-sampling is illustrated in Fig. 2. In each layer, frame *t* represents the voice frame of the current time while other frames have time offsets relative to frame *t*. Several output frames of the last layer are spliced together to be the input of the current layer. By this means, we find that as the network layer goes deeper, the wider layer context can be obtained. For example, the first layer can only splice together 5 frames directly ([-*2*, +*2*]) while the fourth layer can splice together 23 frames indirectly ([-*13*, +*9*]) in all. Furthermore, with sub-sampling operations, fewer input frames are spliced so that fewer computations and weights are needed. In this way, the TDNN can not only obtain wider temporal dependencies which are beneficial to recognition accuracy, but also reduce the amount of computation so that the overall performance can be enhanced and the energy consumption is decreased simultaneously.

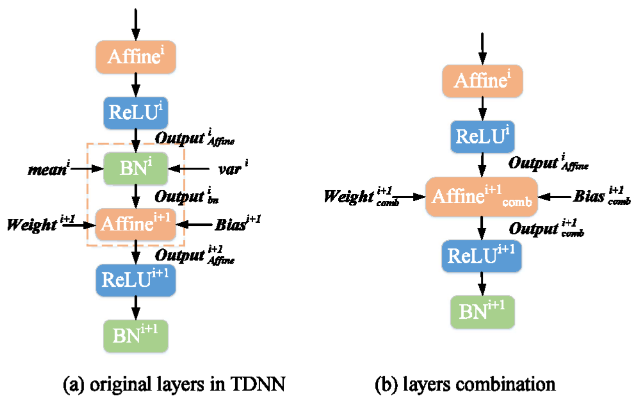

Fig. 3. Layers before and after combination in TDNN

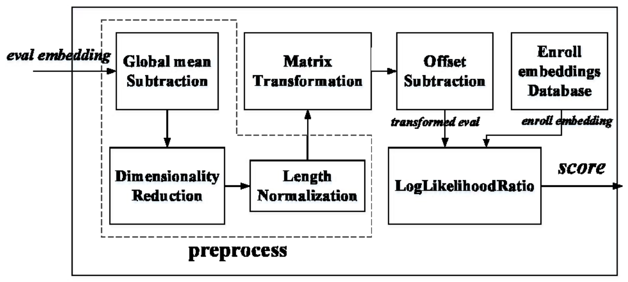

Fig. 4. PLDA classifier flow

The actual configurations of different layers in TDNN of our work are shown in TABLE I, where *Layer context* and *Total context* represent the location and number of spliced frames respectively; *Input* and *Output* are the sizes of the input and output dimensions; *T* represents the total number of frames from one utterance. From this table, we find that the context numbers of the first three layers are more than one, which means that these three layers have frames spliced together as inputs. As a comparison, *Affine4* and *Affine5* have no frame concatenation, which means that these two layers only take the current one frame as their inputs. Owing to this difference, the hardware structure of these two kinds of layers is different, which will be introduced in Section V. It is worth noting that the statistics pooling (*SP*) layer calculates data aggregated over the whole input frames (*Total context* = *T*) from the same utterance. Thus, layers before and after this layer calculate data at the frame and utterance level respectively. The output of the TDNN is a 512-dimensional vector generated by each utterance.

There are six Affine layers in the TDNN network. Each Affine layer is followed by a rectified linear unit (ReLU) and batch normalization (BN) layer. There are no pooling layers except a statistics pooling layer after *Affine5*. According to this feature, the output of the *i*-th BN layer, $Output^{i}_{bn}$ can be combined with the next Affine layer's output $Output^{i+1}_{Affine}$ as shown in (1) and (2) respectively. The output of the combined layer $Output^{i+1}_{comb}$ is shown in (3). $Weight^{i+1}_{comb}$ and $Bias^{i+1}_{comb}$ can be calculated as (4) and (5) respectively.

$$Output^{i}_{bn}=\frac{Output^{i}_{Affine}-mean^{i}}{\sqrt{var^{i}}} \quad (1)$$

$$Output^{i+1}_{Affine}=Output^{i}_{bn}\times Weight^{i+1}+Bias^{i+1} \quad (2)$$

$$Output^{i+1}_{comb}=Output^{i}_{Affine}\times Weight^{i+1}_{comb}+Bias^{i+1}_{comb} \quad (3)$$

$$Weight^{i+1}_{comb}=\frac{Weight^{i+1}}{\sqrt{var^{i}}} \quad (4)$$

$$Bias^{i+1}_{comb}=Bias^{i+1}-\frac{mean^{i}}{\sqrt{var^{i}}}\times Weight^{i+1} \quad (5)$$

Through this combination method, the original six Affine layers and six corresponding BN layers can be simplified: the first Affine layer and the last BN layer remain unchanged while the remaining 10 layers are combined to 5 layers. The computation can be reduced dramatically.

D. PLDA Classifier

The PLDA classifier is shown in Fig. 4. The *eval embedding* extracted from the TDNN part is preprocessed by *Global mean Subtraction*, *Dimensionality Reduction* and *Length Normalization*. Afterwards, the dimension of the vector reduces from the original 512 to 150. Then *Matrix Transformation* and *Offset Subtraction* operations are executed. After these steps, a transformed evaluation vector *transformed eval* is obtained. Finally, *transformed eval* and embeddings from registered database *enroll embedding* are sent to the *LogLikelihoodRatio* score generation submodule to generate the final pairwise scores.

IV. Design Space Exploration

In this section, a quantitative analysis to determine the bit width is proposed. Then, we optimize the parallelism strategy to balance each part of this speaker recognition system.

A. Quantitative Analysis for Hardware Implementation

DNNs can be quantized to reduce computation complexity and memory bottleneck with an acceptable accuracy loss. Previous researches [12][13] have shown that 8-bit fixed-point representation for both weights and activations is enough to solve most of the classification tasks in the computer vision field. However, compared with computer vision tasks, voice-related applications usually need higher precision to maintain recognition accuracy. Hence we carry out the relation between quantized bit width and the accuracy. The result is shown in TABLE II. This quantization work is based on the open-source speech recognition toolkit, Kaldi [14] and the AISHELL-ASR009-OS1 [15] dataset. To evaluate the deviation caused by network quantization, we introduce two additional metrics: $Dist_{cosine}$ to represent the cosine distance between two embeddings and $Dist_{euclidean}$ to represent the Euclidean distance. Equal Error Rate (EER) is used to measure the impact on the final recognition accuracy error. The baseline is tested with weights and activations represented in a 64-bit floating-point format.

As can be seen from TABLE II, in Test Case 1, 16-bit fixed-point representation for both weights and activations has little impact on TDNN results and can even get better system accuracy (lower EER) than the baseline. From other cases, we conclude that compared with bit-width of activations, network results and EER are more sensitive to bit-width of weights. To utilize as low bit-width as possible with rarely accuracy loss on

EER, case 4 is adopted with 12-bit fixed-point representation for both weights and activations.

B. Parallelism Exploration

Owing to the massive computation of neural networks and the limitation of hardware resources, it is necessary to determine the parallelism of each part for the full pipelined architecture. In order to balance the computation time and hardware resource utilization, computation and memory overhead are analyzed as shown in TABLE III. It is obvious that the TDNN part is both computation and memory intensive compared with MFCC and PLDA parts, where N_f represents the number of voice frames per utterance. According to statistics for the length of utterances from the AISHELL-ASR009-OS1 dataset, the average value of N_f is about 480. To avoid aggravating mismatch of computation time among these three parts, it is necessary to explore the TDNN parallelism and related analysis is shown as follows.

Equations (6)-(8) can determine the TDNN parallelism P_i under the constraints of computing resources, where t_i denotes the number of computation clock cycles of processing one frame in layer I; N^i_{in} and N^i_{out} are the input and output dimensions respectively; P^i_{in} and P^i_{out} represent the parallelism of input and output channels in layer $i, P_i=P^i_{in}\times P^i_{out}$. To achieve a full pipelined architecture, it is necessary that t_i of each layer is approximately equal to t, where t is the maximum computation cycles among all the L layers.

$$t_i=\frac{N^i_{in}\times N^i_{out}}{P^i_{in}\times P^i_{out}}=\frac{N^i_{in}\times N^i_{out}}{P_i} \tag{6}$$

$$t=max\{t_1,t_2,\ldots,t_L\}\approx t_1\approx t_2\approx\cdots\approx t_L \tag{7}$$

$$\sum_{i=1}^{L} P_i \leq DSP_{total} \tag{8}$$

$$\frac{Num^i_{BRAM}}{2}=\left[\frac{max\{P^i_{in},P^{i-1}_{out}\}\times B_a}{36}\right]\times\left[\frac{N^i_{in}}{max\{P^i_{in},P^{i-1}_{out}\}\times 1024}\right] \tag{9}$$

$$Num^i_{URAM}=\left[\frac{P_i\times B_w}{72}\right]\times\left[\frac{N^i_{in}\times N^i_{out}}{P_i\times 4096}\right] \tag{10}$$

$$\sum_{i=1}^{L} Num^i_{BRAM} \leq BRAM_{total} \tag{11}$$

$$\sum_{i=1}^{L} Num^i_{URAM} \leq URAM_{total} \tag{12}$$

In equations (8)-(12), DSP_{total}, $BRAM_{total}$ and $URAM_{total}$ are the upbound of FPGA hardware resources; B_a and B_w represent bit-width of activations and weights respectively, which is determined by previous quantitative analysis. Since t_i and P_i are determined based on (6)-(8), it is necessary to check out storage resource constraints (11) and (12) according to (9) and (10). If constraints cannot be met, the determined t_i needs to be increased and P_i will be reduced meanwhile. The minimal t under all these constraints is the final computation cycles of the full pipeline.

After the parallelism of the TDNN part is determined, the balance of time cost among MFCC, TDNN and PLDA parts is considered below.

C. Implementation of the Speaker Verification System

Given that the FPGA chip we use contains three super logic regions (SLRs) and there is some delay penalty when crossing from one SLR to another, we decide to implement our work within one SLR to avoid some timing issues. Therefore, the DSP_{total}, $BRAM_{total}$ and $URAM_{total}$ are just one-third of the resources in the whole chip.

Due to little computation, the clock cycle cost on the MFCC part is around 500 even when the parallelism of this part is not explored, which is faster than the later TDNN part. Based on the above analysis, 1536 is adopted as the value of t. If this value becomes larger, which means the parallelism of TDNN is reduced, imbalance of computation time between MFCC and later two parts will increase so that the system performance will be affected. If this value becomes smaller, the TDNN part will run faster and match better with the MFCC part, but the resource bottleneck will be met. Owing to this, 1536 is taken as the number of clock cycles in a pipelined stage to balance the resource and performance. TABLE IV shows the final parallelism configuration for each layer of TDNN based on the above constraints. To balance the utilization of the BRAMs and

TABLE II. COMPARISON AMONG DIFFERENT QUANTITATIVE CASES

Test Case	Bit-Width		$Dist_{cosine}$	$Dist_{euclidean}$	EER
	Weight	*Activation*			
baseline	64	64	1	0	5.031%
1	16	16	1	0.169	5.003%
2	16	8	0.9997	1.2278	5.522%
3	8	16	0.9827	9.6846	6.037%
4	12	12	0.9999	0.645	5.037%
5	12	10	0.9999	0.7195	5.476%
6	10	12	0.9990	2.4171	5.679%
7	10	10	0.9990	2.4067	5.732%
8	12	8	0.9997	1.3133	5.601%
9	8	8	0.9823	9.8007	6.211%

TABLE III. NUMBER OF OPERATIONS AND COEFFICIENTS PER UTTERANCE

	Speaker Recognition System		
	MFCC	*TDNN*	*PLDA*
Operations	3.0K×N$_f$	8.4M×N$_f$	0.2M
Weights/ Coefficients	1.5K	4.2M	0.1M

TABLE IV. CONFIGURATION FOR TDNN HARDWARE IMPLEMENTATION

Layer	P_i	t_i	Number of BRAMs	Number of DSPs	Number of URAMs
Affine1	5×8	1472	5	40	7
Affine2	8×64	1536	24	512	86
Affine3	8×64	1536	70+257	512	0
Affine4	8×32	1024	59	256	43
Affine5	128×4	1500	88	512	86
BN5	1	512	5	1	0
SP	2	1500	2	2	0
Affine6	2	1500 ×512	5	2	64
Total	-	-	515	1837	286
Percent	-	-	23.7%	26.9%	29.8%

URAMs, 86 URAMs of Affine3 are replaced by 257 BRAMs. It is worth noting that Affine6 carries out the data processing at the utterance level because it is followed by a statistics pooling layer (*SP*) which aggregates over all of the voice frames from one utterance. Furthermore, when the PLDA part is considered, it takes about 512×150 clocks to generate an eval embedding which needs to be sent to the score generation submodule. Given that only 150 clocks are needed to generate pairwise scores, the score generation submodule can deal with 512 speakers' enroll embeddings so that these two parts can match perfectly. If the number of speakers is less than 512, the score generation submodule can be idle sometimes; if the number of speakers is more than 512, the scoring submodule can be expanded to match the former parts due to its little resource consumption.

V. HARDWARE ACCELERATOR DESIGN

In this section, the overall architecture of our accelerator is presented firstly. Then, the key functional modules are introduced.

A. *Accelerator Architecture Overview*

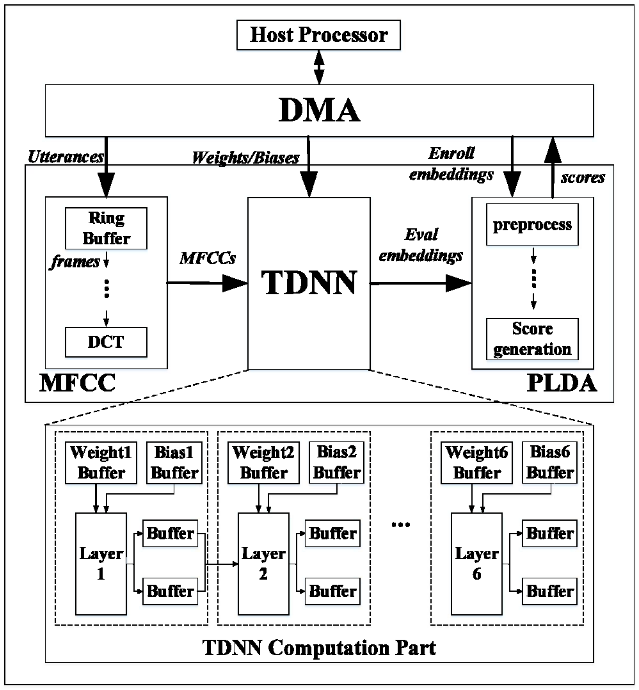

Fig. 5. An overview of the proposed accelerator architecture

The system architecture of the speaker recognition accelerator is shown in Fig. 5. The coefficients in MFCC and PLDA parts, weights and biases in TDNN are all stored in the on-chip buffers. Due to a large number of weights, ultra-rams (URAMs) are used to store weights of each layer while others including biases, coefficients and intermediate data are all stored in the block-rams (BRAMs). DMA is applied to transfer data between the external DDR4 memory and on-chip buffers. All the needed weighs and coefficients are transferred to external DDR4 memory from the host processor via PCI-E interface and then they are allocated to on-chip memory of each module after being read from DDR4 during the initialization process. After parameters preloading operation has been done, the host processor sends utterances as the input of the accelerator. Then a ring buffer is used to divide each utterance into numerous frames and send each frame to the latter computation modules. After operations in MFCC and TDNN parts, embeddings are obtained from each utterance. Besides, a ping-pong structure which includes two buffers is needed at each pipeline stage, so that the latter stage can read data from the other buffer before the previous stage finished writing a buffer. Owing to that most modules process data at the frame level, which means only one frame data is handled each time, the memory size of the intermediate data is quite small compared with other storage consumption. *Enroll embeddings* are transferred from external DDR4 memory to an FIFO in the PLDA part. If *Eval embedding* is extracted from the previous parts, it will be sent to the PLDA part to generate similarity scores when each *Enroll embedding* is read from the storage FIFO. The final scores are stored in the output buffer and finally transferred to the external DDR4 memory.

B. *Ring Buffer Unit*

For the frame blocking phase in Fig. 1, each utterance is divided into frames according to its length, which is shown in Fig. 6 (a). The length of each frame is 25*ms* and the stride is 10*ms*, which means there is 15*ms* overlap between two neighboring frames. A dedicated concurrent read/write memory is constructed based on the BRAMs to adapt the data overlapping characteristics. To reuse the overlapping data stored in the memory, the read-port address will increase in a loopback pattern, which is shown in Fig. 6 (b). For the convenience of reading and writing control, two flags *head* and *tail* are used to mark the start addresses of the next reading and writing operations respectively. Furthermore, an additional bit is used to

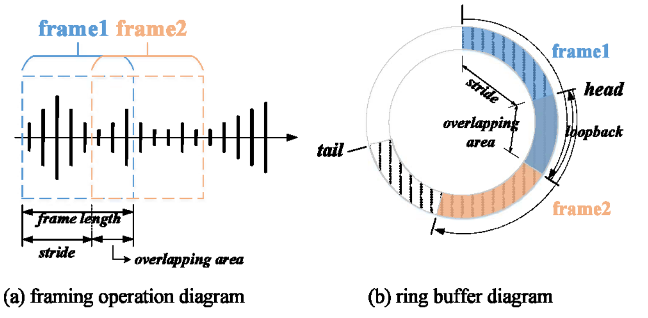

Fig. 6. Diagram of (a) frame blocking (b) read operation of ring buffer

distinguish reading empty and writing full states. If the numbers of *head* and *tail* except the flag bit are the same and the flag bit is different, which means *tail* catches up with *head* flag, the buffer writing operation will stop; and if two flags are all the same, the reading operation will pause.

C. *Mel Filters Bank Unit*

In Fig. 1, the results generated from FFT should be processed by Mel filters bank, which is illustrated in Fig. 7. To guarantee that each filter matches the correct input data, the location and length of each filter are saved in a dual-port BRAM. Hence, these two messages and corresponding coefficients of each filter can be fetched at the same time.

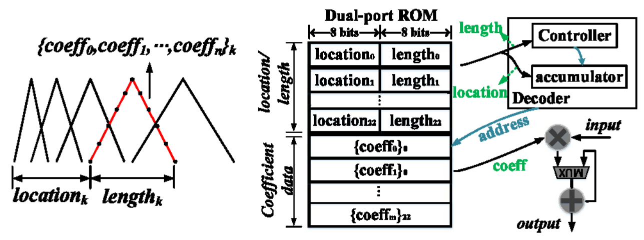

Fig. 7. Diagram of Mel filters bank

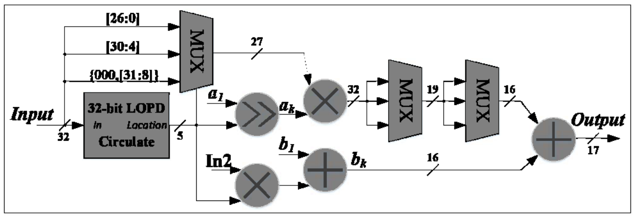

Fig. 8. Diagram of logarithm architecture

D. Logarithm Module

For the logarithm module in Fig. 1, a high speed, low-cost architecture is proposed, which is based on the Mitchell approximation [16] with a piecewise linear approximation. This method divides the input range of the logarithm operation into intervals of powers-of-two integers and uses piecewise linear approximation to estimate final results. According to this division, the slope a_k and intercept b_k of each interval have recursive relationships as shown in (13) and (14). Thus, the corresponding slope a_k and intercept b_k can be calculated if the input interval k and a_1, b_1 are all known. Then the logarithm result can be calculated by applying a linear equation. To find the interval k of input data, a leading one position detector (LOPD) method [17] is employed. The detailed architecture of the logarithm is given in Fig. 8.

$$a_k=\frac{1}{2}\times a_{k-1} \tag{13}$$

$$b_k=b_{k-1}+\ln 2 \tag{14}$$

E. TDNN Module

As mentioned above, the TDNN architecture contains three kinds of computational layers, which are Affine layers without the frame concatenation, Affine layers with input frames spliced together and a statistics pooling layer which calculates mean and variance value of all the frames from one utterance.

For Affine layers without frame concatenation, the computation mode is the same as a fully connected (FC) layer in typical deep neural networks. FC layer can be regarded as a matrix-vector multiplication [18]. The dataflow of this operation is illustrated in Fig. 9. Firstly, P_{in} input activations and $P_{in}\times P_{out}$ weights are sent to processing element units and generate the first intermediate partial sum of output which is stored in the on-chip memory. Secondly, to reuse input data, other $P_{in}\times P_{out}$ new weights shifting in the first direction are loaded to compute other partial sum parts of the output vector. After $\frac{N_{out}}{P_{out}}$ iterations, the partial sum of the whole output vector is obtained. Then the next P_{in} new input activations are loaded, which will shift in the

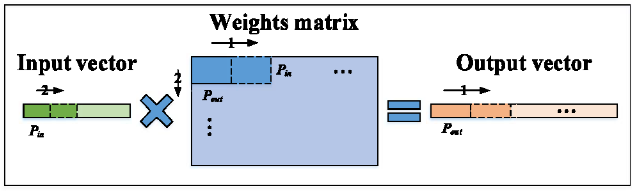

Fig. 9. Dataflow during computation of a basic Affine layer

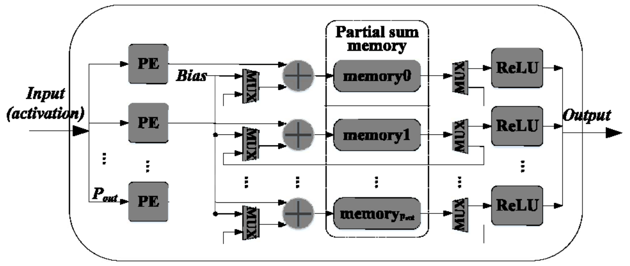

Fig. 10. Architecture of a basic Affine layer

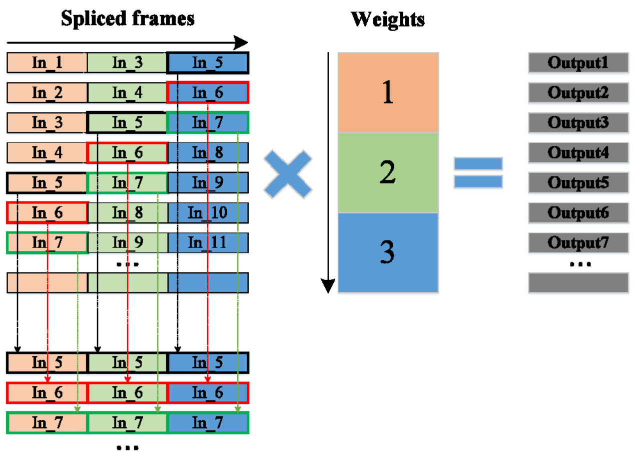

Fig. 11. Computation order reorganization of a Affine layer with spliced frames

second direction as shown in Fig. 9. Finally, after $\frac{N_{in}}{P_{in}}$ iterations, the final results can be achieved. The architecture of this basic layer is shown in Fig. 10. Each PE will complete a vector-vector inner product and generate one partial sum result. Then a corresponding bias or partial sum stored in the memory are selected to add with this newly generated partial sum result and the cumulative result will be sent to the memory to replace the old one. When all the activations of one input frame are added, the final outputs are achieved by doing an extra ReLU operation.

For Affine layers with frames spliced from the last layer, a customized architecture is needed to match the concatenation operation. One method is simply to add a concatenation module at the input. Here the Affine2 layer can be taken as an example. In Affine2 layer, the output frames t-2, t, t+2 coming from the output of Affine1 layer are spliced together and are used to generate output frame t of Affine2 layer. Even though only three frames need to be spliced, the concatenation memory needs to store five frames from t-2 to t+2. The reason is the sub-sampling operation which helps to get a wider context. Considering that a ping-pong buffer structure is used, the overhead of storing input frames will be much higher than the Affine layer without frames concatenation. The second method is to reuse the input frame and store the intermediate partial sum results of other frames. Each frame except the first several frames is needed to be multiplied by three different parts of the weight matrix, which is shown in Fig. 11. In this way, each frame does not need to be spliced with other frames but needs to be loaded three times and each time is multiplied by one-third of the weight matrix. Compared with the first solution, this method can avoid the waiting for all the frames in the context but require additional storage for partial sum results from different output frames, which is acceptable compared with storage for input frames.

For a statistic pooling layer, the variance calculation formula is transformed from (15) to (16) so that data can be handled at the frame level to match former layers rather than be processed at the utterance level which needs all of the frames data stored to get the final results.

$$D(x)=\frac{1}{N}\sum_{n=1}^{N}\left(x(n)-E(x)\right)^2 \tag{15}$$

$$D(x)=E(x^2)-E(x)^2 \tag{16}$$

Fig. 12 shows the architecture of the statistic pooling layer. The input *iData* is added to the previous partial sum value until all the frames are added together. Then the accumulated sum is divided by *Num_frames* to generate the output mean value *E(x)*. *E(x²)* can be calculated in the same way simultaneously by doing an extra square operation. In this way, the input *iData* only needs to be loaded once and can be discarded after it is handled. Thus, there is no need to store all the input data for the second use and numerous storage overhead can be saved.

F. Score Generation Module (LogLikelihoodRatio)

The score generation module computes the similarity *score* of two embeddings after preprocessing shown in Fig. 4, one is *iEnroll* from the person registered in the database and the other is *iEval* which is from person needed to be evaluated. *IEnroll* vector is stored in the FIFO while *iEval* vector is stored in the BRAM. There are two alternative hypotheses: H_s represents that two vectors share the same speaker identity and H_d represents the hypothesis that they are from different identities. The verification *score* can be computed as the loglikelihood ratio for this hypothesis test as

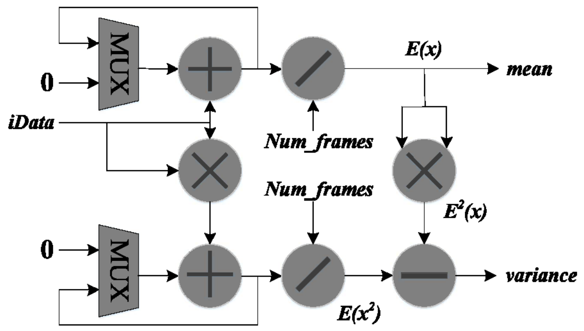

Fig. 12. Architecture of the statistic pooling layer

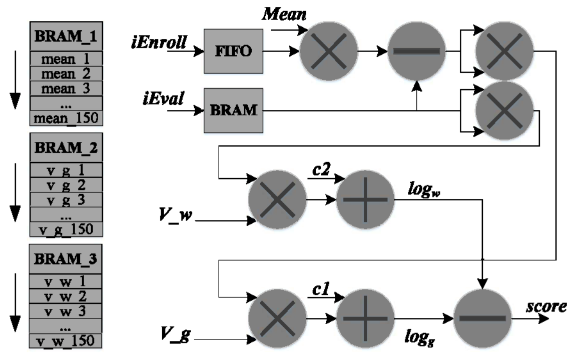

Fig. 13. Architecture of the score generation module

$$score=log\frac{p(n_1,n_2|H_s)}{p(n_1|H_d)p(n_2|H_d)}=log_g-log_w \tag{17}$$

IEnroll and *iEval* are respectively to replace n_1, n_2 in the formula. The architecture of this module is shown in Fig. 13. *Mean*, *V_w*, and *V_g* are all constant vectors that computation steps need and they are all stored in the BRAMs while *c1*, *c2* are two constants stored in the distributed ram. When one *score* is obtained, another new *iEnroll* vector will be fetched from FIFO and calculated with the same *iEval* vector to get a new score. Only when *iEnroll* vectors that need to be matched in the database are all computed, the new *iEval* vector can be loaded from BRAM.

As mentioned above, owing to its little resource consumption, this scoring submodule can be expanded according to the number of people registered in the database to match the processing speed of TDNN part so that the idle waiting time can be reduced to improve the whole system's performance.

VI. Experimental results

We implement this architecture on the Xilinx VCU118 board in Verilog, which contains an XCVU9P FPGA and dual DDR4 memory blocks. Xilinx Vivado Design Tools (v2018.2) is used for the hardware implementation. The whole system runs at 200MHz with 0.095*ns* timing slack. We evaluate our design in terms of resource utilization and efficiency. The resource utilization after routing is shown in TABLE V. The pre-trained network model and parameters are downloaded from the Kaldi official website. The dataset we use is an Open Source Mandarin Speech Corpus called AISHELL-ASR009-OS1. It contains 400 speakers' utterance recordings with total of 178-hour duration. To the best of our knowledge, there are no related papers about high-performance speaker recognition hardware accelerators before, so we can only compare our work with X-vectors software implementation running on Intel Xeon E5-2620 CPU and GTX TITAN Xp GPU. The results are shown in TABLE VI. It is worth noting that for X-vectors implementation on GPU platform in TABLE VI, only the TDNN part is accelerated by GPU, which means the MFCC and PLDA parts are still running on Xeon E5-2620 CPU. Voice frame per second (vFPS) is used as the evaluation metrics based on real measurements. As described in part A of Section V, we record the start time when the parameters preloading operation has been done and the host processor sends the first utterance data. The end time is recorded when the last score received by the host. The time difference is regarded as the total processing time and the vFPS can be calculated by the division operation.

In our design, the processing time per voice frame is 7.72×10^{-3} *ms* under the peak performance. That is, the corresponding vFPS is 1.30×10^{5}, which is 1296× higher than a 2.5GHz Intel Xeon-2620 CPU and 540× higher than work accelerated by GPU operation mode of Kaldi framework on GTX TITAN Xp GPU. Given that each frame costs about 8.4 million operations, the peak performance of this work can achieve 1.067 TOP/s. It is worth noting that the performance of GPU solution is only about 2.4× higher than that of CPU solution. The reason may be lack of optimization of parallelism on GPU when the program runs on the Kaldi framework. To get better performance on GPU, we implement the second TDNN part in the Tensorflow framework to do the acceleration and can get the 650 GOP/s performance. Compared with the throughout, our design can also achieve 1.64× speedup. In addition, the energy consumption of Tensorflow GPU solution is 3.91× than our work. In this design, the power consumption is measured on the Xilinx UltraScale+ VCU118 board during the processing, and the peak power is 36.39W, which includes 22.56W no-load standby power. By comparing energy efficiency, our design can achieve 6.42× efficiency than Tensorflow GPU solution.

TABLE V. RESOURCE UTILIZATION

	Resource Utilization on Xilinx VCU118				
	FF	*LUT*	*DSP48*	*BRAM*	*URAM*
Utilization (Interface)	72469	72311	0	104.5	0
Utilization (Accelerator)	149886	88974	1861	562	286
Utilization (Total)	222355	161285	1861	666.5	286
Available	2364480	1182240	6840	2160	960
Percent(%)	9.40	13.64	27.21	30.86	29.79

TABLE VI. COMPARISON WITH OTHER IMPLEMENTATIONS

	Kaldi CPU	**Kaldi GPU**	**Tensorflow GPU**	**This Work**
Platform	Intel Xeon E5-2620	GTX TITAN Xp	GTX TITAN Xp	Xilinx VCU118
Processing time/frame (ms)	10.15	4.23	1.23×10^{-2}	7.72×10^{-3}
vFPS	98	236	8.07×10^{4}	1.30×10^{5}
Performance (GOP/s)	0.824	1.98	650	1.067×10^{3}
Power(W)	56.22	121.32	142.26	36.39
Energy Efficiency (GOP/s/W)	1.47×10^{-2}	1.64×10^{-2}	4.57	29.32

VII. CONCLUSIONS

In this paper, we propose an end-to-end speaker recognition system accelerator based on X-vectors. A quantitative analysis of X-vectors algorithm and an optimization strategy aimed to determine the parallelism are proposed. Based on the above analysis, we implement our accelerator (MFCC, TDNN, and PLDA classifier) with a pipelined structure to make all the functional modules work concurrently. This work is implemented on the Xilinx XCVU9P FPGA of UltraScale+ VCU118 board and can achieve 1.067 TOP/s and 1.30×10^{5} vFPS with 200MHz clock frequency, which gets a 1296× speedup compared with the algorithm running on Intel Xeon E5-2620 processor and 6.42× energy efficiency compared with GTX TITAN Xp GPU solution.

REFERENCES

[1] Douglas A Reynolds, Thomas F Quatieri, and Robert B Dunn, "Speaker verification using adapted gaussian mixture models," Digital signal processing, vol. 10, no.1-3, pp. 19–41, 2000.

[2] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on. IEEE, 2013, pp. 6645–6649.

[3] H. Sak, A. Senior, and F. Beaufays, "Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition," Tech. Rep. 14021128v1 [cs.NE], Feb. 2014, arXiv.

[4] Vijayaditya Peddinti, Daniel Povey, and Sanjeev Khudanpur, "A time delay neural network architecture for efficient modeling of long temporal contexts," in the Proceedings of INTERSPEECH, 2015b.

[5] D. Snyder, D. Garcia-Romero, G. Sell, D. Povey, and S. Khudanpur, "X-vectors: Robust DNN embeddings for speaker recognition," in Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, 2018.

[6] P. Ehkan, F. F. Zakaria, M. N. M. Warip, Z. Sauli and M. Elshaikh, "Hardware Implementation of MFCC-Based Feature Extraction for Speaker Recognition," in Advanced Computer and Communication Engineering Technology. Springer, 471-480.

[7] F. Jhing, S. Jr, C. Jia, C. Po and W. Ta, "Hardware/software co-design for fast-trainable speaker identification system based on SMO," in Proceedings of the 2011 IEEE International Conference on Systems, Man, and Cybernetics (SMC'11). 1621-1625.

[8] C. Jia, X. Lian, Y. Yan and H. Jia, "VLSI Design for SVM-Based Speaker Verification System," IEEE Trans. Syst. 99, 2014.

[9] R. R. Lara, M. L. Garcia, E. C. Navarro and L. P. Rodriguez, "SVM speaker verification system based on a low-cost FPGA," in Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), 2009. IEEE, 582-586.

[10] Phaklen EhKan, Timothy Allen and Steven F. Quigley, "FPGA Implementation for GMM-Based Speaker Identification," Int. J. Reconfig. Comput. 2011, Article 3 (Jan. 2011).

[11] S. Ioffe, "Probabilistic linear discriminant analysis," Computer Vision–ECCV 2006, pp. 531–542, 2006.

[12] V. Vanhoucke, A. Senior, and M. Z. Mao, " Improving the speed of neural networks on CPUs," In Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop, volume 1, page 4, 2011.

[13] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," arXiv preprint arXiv:1712.05877 (2017).

[14] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motl´ı̌cek, Y. Qian, P. Schwarz, et al., "The Kaldi speech recognition toolkit," in Proceedings of the Automatic Speech Recognition & Understanding (ASRU) Workshop, 2011.

[15] H. Bu, J. Du, X. Na, B. Wu, and H. Zheng, “Aishell-1: An opensource mandarin speech corpus and a speech recognition baseline,” arXiv preprint arXiv:1709.05522, 2017.

[16] J. N. Mitchell, “Computer multiplication and division using a binary logarithms”, IEEE Trans. Electron. Comput., vol. 11, pp. 512-517, 1962.

[17] Kunaraj, K. and R. Seshasayanan, “Leading one detectors and leading one position detectors–an evolutionary design methodology.” Canadian Journal of Electrical and Computer Engineering 36.3(2013):103-110.

[18] H. Li, X. Fan, L. Jiao, W. Cao and X. Zhou, “A high performance FPGA-based accelerator for large-scale convolutional neural networks,” in Field Programmable Logic and Applications (FPL), 2016, pp. 1-9.