Available online at www.elixirpublishers.com (Elixir International Journal)

Electrical Engineering

Elixir Elec. Engg. 93 (2016) 39613-39618

ISSN: 2229-712X

ASR For Embedded Real Time Applications

Appavoo Namachivayam[1], Kaliyaperumal Karthikeyan[2] and Dr.M.Peer Mohamed[3]
[1,2]Department of Computer Science, Eritrea Institute of Technology, Asmara, Eritrea.
[3]Department of Mathematics, Eritrea Institute of Technology, Asmara, Eritrea.

ARTICLE INFO

Article history:
Received: 17 March 2016;
Received in revised form:
12 April 2016;
Accepted: 18 April 2016;

Keywords
Automatic Speech Recognition (ASR), Embedded system, Hardware–software code Sign, Real-time system, Soft-core-based system.

ABSTRACT
The system consists of a standard microprocessor and a hardware accelerator for Gaussian mixture model (GMM) emission probability calculation implemented on a field-programmable gate array. The GMM accelerator is optimized for timing performance by exploiting data parallelism. In order to avoid large memory requirement, the accelerator adopts a double buffering scheme for accessing the acoustic parameters with no assumption made on the access pattern of these parameters. Experiments on widely used benchmark data show that the real-time factor of the proposed system is 0.62, which is about three times faster than the pure software-based baseline system, while the word accuracy rate is preserved at 93.33%. As a part of the recognizer, a new adaptive beam-pruning algorithm is also proposed and implemented, which further reduces the average real-time factor to 0.54 with the word accuracy rate of 93.16%. The proposed speech recognizer is suitable for integration in various types of voice (speech)-controlled applications.

I. Introduction

Automatic speech recognition (ASR) on embedded platforms has been gaining its popularity. ASR has been widely used in human–machine interaction, such as mobile robots, consumer electronics, and manipulators in industrial assembly lines, automobile navigation systems, and security systems. More sophisticated ASR applications with larger vocabulary sizes and more complex knowledge sources are expected in the future. As a result, the demand for high performance, accurate, and fast embedded ASR is increasing. This approach enables fast deployment of ASR-based applications. However, the timing performance is constrained by the processing power and memory bandwidth of the target platforms. At another extreme, a speech recognizer can be tailor-made in a *pure hardware-based* system for good timing performance. However, in many human-machine interaction applications, the search space for decoding speech varies dynamically depending on the user's response. Dedicated hardware architecture with a static search space has limited capabilities to deal with the dynamic nature of ASR. In addition, the architecture becomes too application-specific and targets to only ASR applications. It is unlikely that the data path of the hardware can be reused for applications other than ASR. As a compromise, a *hardware–software code sign* approach seems to be attractive. A typical hardware–software co processing system consists of a general purpose processor and hardware units that accelerate time critical operations to achieve required performance. Computationally intensive parts of the algorithm can be handled by the hardware accelerator(s), while sequential and control-oriented parts can be run by the processor core. The additional advantages of the hardware–software approach include the following:

1) Rapid prototyping of applications. Developers can build their applications in software without knowing every detail of the underlying hardware architecture.
2) Flexibility in design modification. The parts of the algorithm which require future modification can be implemented initially in software.
3) Universality in system architecture. The use of the general-purpose processor core enables developers to integrate ASR easily with other applications.

In this paper, we present the development and tradeoffs of a hardware–software co processing ASR system which primarily targets on embedded applications. The system includes an optimized hardware accelerator that deals with the critical part of the ASR algorithm. The final system achieves real-time performance with a combination of software- and hardware implemented functionality and can be easily integrated into applications with voice (speech) control.

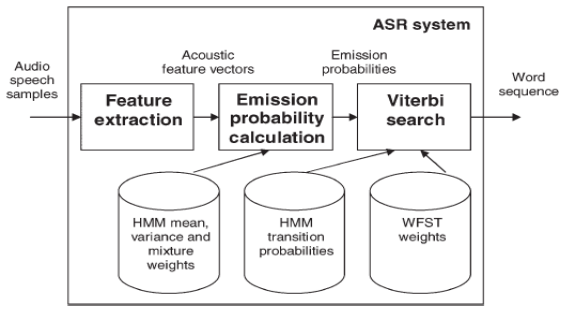

Fig 1. Data flow diagram of a typical ASR system. The input of the system is an audio speech signal. The output is a sequence of words.

II. Automatic Speech Recognition System

In a typical hidden Markov model (HMM)-based ASR system, three main stages are involved. Fig. 1 shows the dataflow within the ASR algorithm. The first stage is *feature extraction*. Its main purpose is to convert a speech signal into a sequence of acoustic feature vectors, $\mathbf{o}^{T}_{1} = \{\mathbf{o}_1, \mathbf{o}_2, \ldots, \mathbf{o}_T\}$, where T is the number of feature vectors in the sequence.

Tele:
E-mail address: kirithicraj@gmail.com

The entire speech signal is segmented into a sequence of shorter speech signals known as frames. The time duration of each frame is typically 25ms with 15ms of overlapping between two consecutive frames. Each frame is characterized by an acoustic feature vector consisting of *D* coefficients. One of the widely used acoustic features is called Mel frequency campestral coefficient (MFCC). Feature extraction continues until the end of the speech signal is reached. The next stage is the calculation of the *emission probability* which is the likelihood of observing an acoustic feature vector. The emission probability densities are often modeled by Gaussian mixture models (GMMs). The last stage is *Viterbi search* which involves searching for the most probable word transcription based on the emission probabilities and the search space. The use of weighted finite state transducers (WFSTs) offers a tractable way for representing the search space. The advantage is that the search space represented by a WFST is compact and optimal. Fig. 2 shows an example of a search space. Basically, a WFST is a finite state machine with a number of states and transitions. As shown in Fig. 2, each WFST transition has an input symbol, an output symbol, and a weight. The input symbols are the triphone or biphone labels. The output symbols are the word labels. In ASR, a word is considered as a sequence of sub word units called phones. Two or three phones are concatenated to form biphones or triphones. Each triphone or biphone label is modeled by an HMM. In other words, each WFST transition in Fig. 2 is substituted by an HMM. The entire WFST is essentially a network of HMM states. The WFST weights are the language model probabilities which model the probabilistic relationship among the words in a word sequence. Usually, a word is grouped with its preceding $(n-1)$ words. The *n*-word sequence called *n*-gram is considered as a probabilistic event. The WFST weights estimate the probabilities of such events. Typical *n*-grams used in ASR are unigram (one word), bigram (two word, also known as word pair grammar), and trigram (three word).For implementation purposes, each HMM state has a bookkeeping entity called *token* which records the probability(*score*) of the best HMM state sequence ended at that state. Each token is propagated to its succeeding HMM states according to the topology of the search space. For example, in Fig. 2, the token in State 3 of the /k-ae+t/ HMM will replicate itself. One will propagate to its own state with HMM transition probability ($a33$ in this example) added to the token's score. Another replicated token will enter State 1 of the /ae-t/ HMM and the WFST weight ($\omega 6$) will be added to its score. When two tokens meet at an HMM state, only the better token with a higher score survives and stays at the HMM state. Other losing tokens are discarded. This method of performing the Viterbi search is known as *token passing*. In addition to as core, tokens also record a sequence of word labels encountered during propagation. The pseudo code of the ASR algorithm is shown in Fig. 3. In the beginning of the algorithm, a token is instantiated in each of HMM states at the start of each word (Line 2). $Q_{word-start}$ is a set of word-starting HMM states. The score of each token is reset (Line 3). After the initialization, the algorithm begins to process each frame of speech. An acoustic feature vector, $\mathbf{o}_t$, is generated by *feature extraction* (Line 6) for each speech frame. In practice, it is intractable to perform a complete Viterbi search over all the HMM states within the search space. Therefore, *pruning* is essential for practical applications with the cost of introducing search errors. One of the common pruning techniques is called *beam pruning*. A *pruning threshold* is determined by subtracting a certain value called *pruning beam width* from the maximum token score (Lines 8–9). A token remains active if its score is above the pruning threshold (Line 13). Otherwise, the token is discarded. Pruning is said to be tight when the pruning beam width is narrow, which reduces the number of active tokens in the search space. Since there are fewer tokens, the decoding time is shorter. However, the word accuracy rate tends to decrease in tight pruning since the token with the correct word transcription has a greater chance to be discarded.

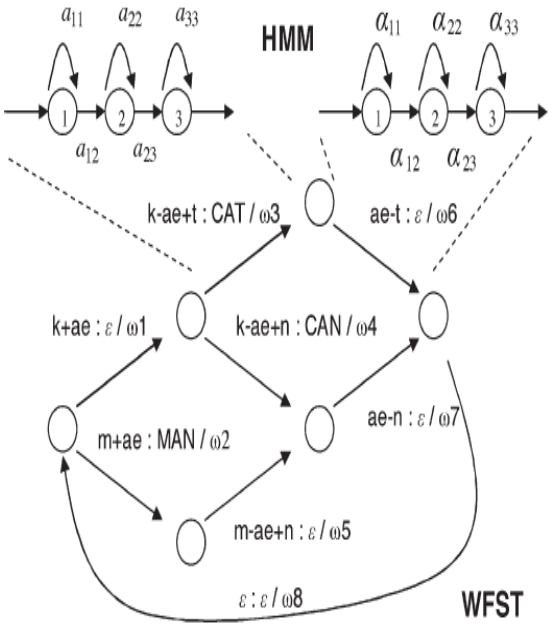

Fig 2. Search space represented by a WFST. Each WFST transition *x: y/z* has three attributes. *X* is an input symbol representing a triphone or biphone label. *y* is an output label representing a word label.

Algorithm 1 Speech recognition algorithm

1: /* $\tilde{Q}_t$ is a set of HMM states which have tokens at time t */
2: $\tilde{Q}_1 \leftarrow Q_{word-start}$
3: $score_{q,1} \leftarrow 0$ for all $q \in \tilde{Q}_1$
4:
5: **for** $t = 1$ to T **do**
6: $\quad \mathbf{o}_t \leftarrow Feature\_extraction(Frame_t)$
7:
8: $\quad max\_score \leftarrow \max(score_{q,t})$ for all $q \in \tilde{Q}_t$
9: $\quad pruning\_threshold \leftarrow max\_score - pruning\_beamwidth$
10: $\quad \tilde{Q}_{t+1} \leftarrow \{\}$
11:
12: $\quad$ **for all** $q \in \tilde{Q}_t$ **do**
13: $\quad\quad$ **if** $score_{q,t} > pruning\_threshold$ **then**
14: $\quad\quad\quad log\_emis\_prob \leftarrow Emission\_prob\_calc(\mathbf{o}_t, q)$
15: $\quad\quad\quad \mathcal{V} \leftarrow Viterbi\_search(log\_emis\_prob, q, t)$
16: $\quad\quad\quad \tilde{Q}_{t+1} \leftarrow \tilde{Q}_{t+1} \cup \mathcal{V}$
17: $\quad\quad$ **end if**
18: $\quad$ **end for**
19: **end for**
20:
21: $\mathcal{Q} \leftarrow \tilde{Q}_{T+1} \cap Q_{word-end}$
22: $best\_token \leftarrow \underset{q \in \mathcal{Q}}{\operatorname{argmax}}(score_{q,T+1})$

Fig 3. Pseudo code of the speech recognition algorithm with beam pruning.

Algorithm 2 $\mathcal{V} \leftarrow Viterbi\_search(log\_emis\_prob, q, t)$

1: $\mathcal{V} \leftarrow \{\}$
2: **for all** q_suc states that succeed State q **do**
3: $new\_score \leftarrow score_{q,t} + log\_emis\_prob + trans\_weight(q, q\_suc)$
4: **if** $new\_score > score_{q\_suc,t+1}$ **then**
5: $score_{q\_suc,t+1} \leftarrow new\_score$
6: $path_{q\_suc,t+1} \leftarrow path_{q,t}$
7: $\mathcal{V} \leftarrow \mathcal{V} \cup \{q\_suc\}$
8: **end if**
9: **end for**
10: **return** $\mathcal{V}$

Fig 4. Pseudo code of the viterbi_*search* function

After feature extraction and setting the pruning threshold, the algorithm iterates through all the HMM states that have a token(Lines 12–18). If the token stays above the pruning threshold, the *emission probability* of that state is calculated (Line 14). After that, *Viterbi search* is performed on that HMM state (Line 15).Token-passing takes place during this process. It returns a set of new HMM states, *V*, which are occupied by the new tokens after token passing. The new tokens are accumulated into another set ˜*Qt*+1 which is prepared for the next speech frame. Once all the speech frames have been processed, the best token is found among all the word-end HMM states denoted by *Q* (Lines 21–22). The best token records its propagation path from which the word transcription can be determined. Fig. 4 shows the pseudo code of the *viterbi_search* () function. The for-loop iterates through all the succeeding states of *q* (Lines 2–9). For each succeeding state, *new_score*is calculated (Line 3) where the transition weight can be either the HMM transition probability for within-HMM transitions or the WFST transition weight for cross-HMM transitions. If *new score* is greater than the score at *q_suc*, the *new score* will update the score at *q_suc*(Line 5). The path record of the original token at *q_suc* is replaced by the path record at *q* (Line 6).The pseudo code shows that there are three major levels of iterations in the ASR algorithm: 1) iteration of *T* speech frames (Line 5 in Fig. 3); 2) iteration of ˜*Qt*HMM states in each frame (Line 12 in Fig. 3); and 3) iteration of *qsuc* states for each active HMM state (Line 2 in Fig. 4). Since the search result of each speech frame in the first iteration loop depends on previous frames, only the second and the third loops are suitable for possible parallelism. However, data contention is likely to occur because an HMM state is often a *qsuc* state of multiple HMM states. The impact of contention on timing performance needs to be carefully studied if parallelism is adopted. The performance of an ASR system is often evaluated by two metrics. The first metric is word accuracy rate which is defined as follows [23]:

$$\text{Word accuracy rate} = \frac{n - s - d - i}{n} \times 100\% \quad (1)$$

Where *n* is the total number of words. *s* is the number of word substitutions (incorrectly recognized words). *d* and *i* are the numbers of word deletions and word insertions, respectively. The second metric is real-time factor which measures the timing performance of the ASR system. It is defined as follows:

$$\text{Real-time factor} = \frac{\text{Decoding time}}{\text{Speech duration}}. \quad (2)$$

III.Hardware–Softwarecoprocessing System

The ASR algorithm is partitioned into three main parts: feature extraction, GMM emission probability calculation, and Viterbi search. The speech recognizer is first implemented in software where the 16-b fixed-point implementation of the recognizer is compared with the floating-point implementation. The experimental results show that there is no degradation in recognition accuracy in the fixed-point implementation. Hence, the fixed-point system is chosen as our baseline system for time profiling. It shows that about 69% of the total elapsed time is spent on GMM computation. The proportions of time spent on feature extraction and Viterbi search are 7% and 24%, respectively. Since GMM computation is the most computationally intensive part, a hardware accelerator is designed in order to speed up this part of the ASR algorithm.

A. System Architecture

The architecture of the hardware–software co processing system is shown in Fig. 5. The system consists of an Altera Nios II processor core and a GMM hardware accelerator. The Nios II processor acts as the control unit of the entire system. Feature extraction and Viterbi search are implemented in software. When the system needs to perform a GMM calculation, the processor instructs the accelerator to carry out the computation. The accelerator returns the computation result to the Nios II core. The entire co processing system is synthesized on an Altera Stratix II EP2S60F672C5ES field-programmable gate array (FPGA).

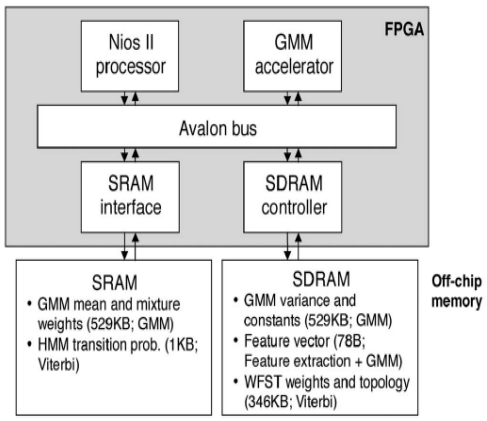

Fig 5. System architecture of the hardware–software co processing recognizer with the GMM hardware accelerator. Inside the brackets, it shows the data size and the ASR sub stages in which the data are accessed. The Nios II processor performs feature extraction and Viterbi search, while the GMM accelerator is used for GMM computation.

B. GMM Emission Probability Hardware Accelerator

1) Data path: The GMM hardware accelerator calculates the log emission probability of an observation vector given an HMM state. Given an observation feature vector $\mathbf{o}_t$, the emission probability function in an HMM state *j* is modeled by a sum of weighted Gaussian mixtures

(3)

$$b_j(\mathbf{o}_t) = \sum_{m=1}^{M} b_{jm}(\mathbf{o}_t)$$

$$= \sum_{m=1}^{M} c_{jm}\mathcal{N}(\mathbf{o}_t, \boldsymbol{\mu}_{jm}, \boldsymbol{\Sigma}_{jm})$$

Where *bjm*(**o***t*) is the probability density function of the weighted *m*th Gaussian mixture. *N*(.) denotes a Gaussian mixture. The mean vector and the covariance matrix of the Gaussian mixture are denoted by ***μ****jm*and **Σ***jm*, respectively. Since the coefficients of a feature vector are assumed to be independent, **Σ***jm*is a diagonal matrix. The total number of Gaussian mixtures is*M*per HMM state. The weight of the *m*thGaussian mixture is *cjm*. The logarithm of a weighted Gaussian mixture, log *bjm*(**o***t*),can be expressed by the following equation:

(4)

$$\log b_{jm}(\mathbf{o}_t) = C_{jm} + g_{jm} + \sum_{d=0}^{D-1} \left(o_t^{(d)} - \mu_{jm}^{(d)}\right)^2 v_{jm}^{(d)}.$$

In the equation, *o*(*d*)*t* is the *d*th dimension of the observation vector at time *t*. *D* is the dimension of the observation vector. In many ASR applications, the typical value of *D* is 39, which is commonly adopted by the research community. *μ*(*d*)*jm*is the *d*th dimension of the ***μ****jm* mean vector. *Cjm*, *v*(*d*)*jm*, and *gjm* are constants defined as follows:

(5)

$$C_{jm} = \log c_{jm}$$

(6)

$$v_{jm}^{(d)} = \frac{-1}{2\left(\sigma_{jm}^{(d)}\right)^2}$$

(7)

$$g_{jm} = -\frac{1}{2}\left(D\log(2\pi) + \sum_{d=0}^{D-1} \log\left(\sigma_{jm}^{(d)}\right)^2\right)$$

Where the symbol (*σ*(*d*)*jm*)2 is the *d*th feature variance, which is the *d*th diagonal element of the covariance matrix. The log emission probability, log *bj*(**o***t*), can be evaluated recursively by the following equation:

(8)

$$\log b_j(\mathbf{o}_t) = (((\log b_{j1}(\mathbf{o}_t) \oplus \log b_{j2}(\mathbf{o}_t)) \oplus \ldots) \oplus \log b_{jM}(\mathbf{o}_t).$$

The $\oplus$symbol represents the *log-add* operator, which has the following definition and approximation:

(9)

$$x \oplus y = \log\left(\exp(x) + \exp(y)\right)$$

$$\approx \begin{cases} y & z < -16 \\ y + \log\left(1 + \exp(z)\right) & -16 \le z < 0 \\ x + \log\left(1 + \exp(-z)\right) & 0 \le z < 16 \\ x & z \ge 16 \end{cases}$$

Where $z = x - y$. When $|z|$ is greater than a threshold, the difference between exp(*x*) and exp(*y*) is large enough to just consider only the greater number. The threshold value of 16 is chosen because it shows no degradation in recognition accuracy and also it is a power of two. Several different thresholds (8, 16,and 32) are tested. The word accuracy rates stay at 93.33% for threshold values of 16 and 32, whereas there is a slight decrease in word accuracy (about 0.03%) when the threshold is 8. The log(1 + exp(.)) function can be calculated offline and stored in a lookup table. The |*z*| value can be used as the look-up index of the table. It can be seen from (4) that there is a summation of *D* interim values. Since these values are independent of each other, it is possible to compute *N* of them at the same time in parallel, where $1 \le N \le D$. For example, if $N = D$, *D* interim values are calculated in one go. However, if $1 < N < D$, *N* interim values are calculated each time and it requires _*D*/*N*_ iterations to calculate all the values. In contrast, there is no parallelism if $N = 1$. In other words, the degree of parallelism is governed by *N*, and it is a design variable which needs to be optimally chosen. In order to avoid pipeline stalls, the hardware accelerator adopts a double-buffering scheme as shown in Fig. 6. Each buffer contains the GMM parameters of an HMM state. Since the Avalon bus is 32-b wide and there are two separate memories (SRAM and SDRAM), 8 B of parameters can be loaded to the buffer in each clock cycle. GMM calculation and Viterbi search are performed during the retrieval of the next HMM state parameters from the off-chip memories. The accelerator only needs to store the parameters of two HMM states, which are about 1280 B in the internal memory of the FPGA chip. observation vector only needs to be loaded once for each speech frame. The size of the observation vector buffer is 78 B.

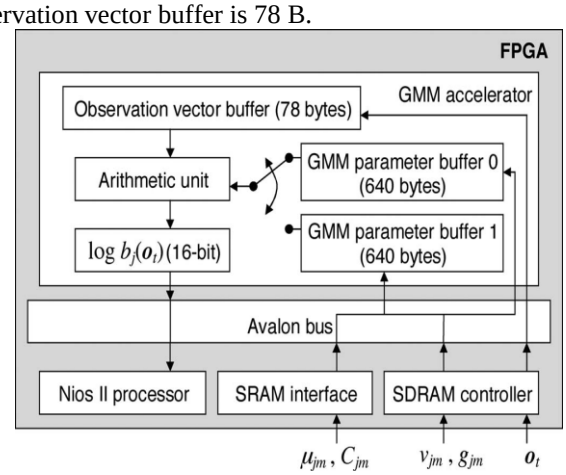

Fig 6. Double-buffering inside the GMM hardware accelerator. The arithmetic unit is reading from one buffer while another buffer is retrieving GMM parameters from off-chip memories.

The major differences between the proposed system and the other co processing system are as follows:

a) The GMM accelerator has only one computation unit for calculating one dimension. We argue that this architecture is not optimized. The proposed system includes *N* computation units and a parallel adder block to further employ data parallelism.

b) The accelerator in their system computes log $b_{jm}(\mathbf{o}_t)$only. The summation of Gaussian mixtures is done by the general purpose processor in software, while the proposed accelerator includes a hardware log-add unit and the final output is log $b_j(\mathbf{o}_t)$.

c) The accelerator in their system internally stores 128 kB of HMM parameters, which is about 20% of the total amount.

This makes the architecture infeasible for larger vocabulary tasks. In addition, the parameters are predetermined. The parameters of the most probable HMM states, which are found by offline profiling on the test speech data, are stored inside the accelerator. In contrast, our proposed accelerator only stores two HMM states (1280 B).Furthermore, we do not make any assumptions on which HMM states should be stored.

2) Timing Profile: After synthesis and place and route, the proposed system is implemented on the target FPGA board. The first experiment is to investigate the relationship between the speedup in GMM calculation and the number of parallel computation units (*N*). The aim is to find the smallest number of computation units with maximum speed up. Fig. 9shows the number of clock cycles for GMM calculation versus the number of computation units. The task is the Resource Management (RM1) task, which consists of 1200 test utterances. The vocabulary size is 993. Triphone HMM models with three emitting states and four Gaussian mixtures per state are trained on 2880 utterances. Acoustic features are 39-D MFCCs with the zeroth coefficient plus their delta and delta–delta coefficients. The language model is word-pair grammar (bigram).In terms of word accuracy, the GMM accelerator is the exact implementation of the algorithm. Hence, the word accuracy rate is 93.33% which is the same as that of the pure software-based system.

3) Resource Usage: Table II shows the resource usage of the GMM hardware accelerator. Adaptive Logic Module (ALM), which can be programmed to perform logic functions, is the building block of a Stratix II FPGA device. M4K RAM blocks on the FPGA provide on-chip memory storage. Hardware multipliers are also embedded on the FPGA.

IV. Adaptive Pruning

Our goal is to reduce the decoding time of those utterances which have a relatively greater real-time factor, while keeping the recognition accuracy of the other utterances. In order to fulfill this goal, an *adaptive* pruning scheme is proposed, where the pruning beam width is adaptive according to the number of active tokens.

A. Algorithm

Fig. 11 shows the pseudo code of the ASR algorithm with adaptive pruning. In the beginning, the beam width is initialized to a value (Line 4). Before token passing, the algorithm modifies the pruning beam width according to the number of active tokens, $n(\tilde{Q}t)$. If the number of tokens is greater than a threshold, upper, a tighter beam width is adopted. The beam width is decreased by a certain amount denoted by δ (Lines 11–12).However, if the number of active tokens is smaller than another threshold, τlower, and also if the beam width is tightened previously, the beam width will be relaxed and its value will be increased by δ (Lines 13–16). The rest of the algorithm is the same as the one shown in Fig. 3.The proposed pruning scheme is more flexible than the narrow and fixed pruning scheme. The number of active tokens is often time varying in the duration of an utterance. The fixed pruning scheme applies a tight beam width throughout the entire utterance regardless of the number of active tokens. on the other hand, the adaptive scheme allows relaxation of the beam width in parts of the utterance where the workload is less heavy. In terms of implementation, the proposed adaptive scheme is simpler than histogram pruning. Implementing histogram pruning requires a sorted list of the token scores. For each token, the recognizer needs to perform an insertion sort which involves searching for the token's ranking in a sorted list of the previously iterated token scores. Maintaining the tokens in a sorted order is computationally intensive. In contrast, the adaptive pruning scheme only requires to record the number of active tokens and a few decision-making statements (if-statements) for adjusting the beam width once for every speech frame.

B. Timing Profile

Fig. 12 shows the real-time factor of the co processing system. Fixed beam pruning and adaptive beam pruning are compared. The beam width is held constant at 170 for the fixed beam-pruning scheme. In adaptive beam pruning, the *original beam width* variable is also set to 170. The thresholds, τlower and τupper, are 1900 and 2300, respectively. The beam width adjustment value is 10 (δ = 10). These parameters are determined empirically. In the fixed beam-pruning scheme, about 94% of the utterances have a real-time factor below one. When the adaptive beam-pruning scheme is used, this percentage increases to99.75%. Only 3 out of 1200 utterances have a real-time factor above one. Compared with the fixed beam-pruning scheme, there is a small degradation in recognition accuracy which decreases from 93.33% to 93.16%.We have also tried to tighten the adaptive pruning scheme by adjusting τupper and τlower to smaller values (τupper = 1700, τlower = 1250), so that the real-time factors of all the utterances are below 1. The word accuracy rate reduces to 92.62%.

Algorithm 3 Speech recognition algorithm with adaptive beam pruning

```
1: /* Q̃_t is a set of HMM states which have tokens at time t */
2: Q̃_1 ← Q_word−start
3: score_{q,1} ← 0 for all q ∈ Q̃_1
4: pruning_beamwidth ← original_beamwidth
5:
6: for t = 1 to T do
7:    o_t ← Feature_extraction(Frame_t)
8:
9:    max_score ← max(score_{q,t}) for all q ∈ Q̃_t
10:
11:   if n(Q̃_t) > τ_upper then
12:      pruning_beamwidth ← pruning_beamwidth − δ
13:   else if n(Q̃_t) < τ_lower then
14:      if pruning_beamwidth < original_beamwidth then
15:         pruning_beamwidth ← pruning_beamwidth + δ
16:      end if
17:   end if
18:
19:   pruning_threshold ← max_score − pruning_beamwidth
20:   Q̃_{t+1} ← {}
21:
22:   for all q ∈ Q̃_t do
23:      if score_{q,t} > pruning_threshold then
24:         log_emis_prob ← Emission_prob_calc(o_t, q)
25:         V ← Viterbi_search(log_emis_prob, q, t)
26:         Q̃_{t+1} ← Q̃_{t+1} ∪ V
27:      end if
28:   end for
29: end for
30:
31: Q ← Q̃_{T+1} ∩ Q_word−end
32: best_token ← argmax_{q∈Q} score_{q,T+1}
```

Fig 7. Speech recognition algorithm with adaptive beam pruning.

V. Conclusion

The proposed ASR system shows much better real-time factors than the other approaches without decreasing the word accuracy rate. Other advantages of the proposed approach include rapid prototyping, flexibility in design modifications, and ease of integrating ASR with other applications. These advantages, both quantitative and qualitative, suggest that the proposed co processing architecture is an attractive approach for embedded ASR. The proposed GMM accelerator shows three major improvements in comparison with another co processing system. First, the proposed accelerator is about four times faster by further exploiting parallelism. Second, the proposed accelerator uses a double-buffering scheme with a smaller memory footprint, thus being more suitable for larger vocabulary tasks. Third, no assumption is made on the access pattern of the acoustic parameters, whereas the accelerator has a predetermined set of parameters. Finally, we have presented a novel adaptive pruning algorithm which further improves the real-time factor. Compared with other conventional pruning techniques, the proposed algorithm is more flexible to deal with the time-varying number of active tokens in an utterance. The performance of the proposed system is sufficient for a wide range of speech-controlled applications. For more complex applications which involve multiple tasks working with ASR, further improvement of timing performance, for example, by accelerating the Viterbi search algorithm, might be required.

References

[1] A. Green and K. Eklundh, “Designing for learn ability in human–robot communication,” *IEEE Trans. Ind. Electron.*, vol. 50, no. 4, pp. 644–650,Aug. 2003.

[2] M. Imai, T. Ono, and H. Ishiguro, “Physical relation and expression: Joint attention for human-robot interaction,” *IEEE Trans. Ind. Electron.*,vol. 50, no. 4, pp. 636–643, Aug. 2003.

[3] B. Jensen, N. Tomatis, L. Mayor, A. Drygajlo, and R. Siegwart, “Robots meet humans—Interaction in public spaces,” *IEEE Trans. Ind. Electron.*,vol. 52, no. 6, pp. 1530–1546, Dec. 2005.

[4] H. Lam and F. Leung, “Design and training for combinational neurologic systems,” *IEEE Trans. Ind. Electron.*, vol. 54, no. 1, pp. 612–619,Feb. 2007.

[5] A. Chatterjee, K. Pulasinghe, K. Watanabe, and K. Izumi, “A particles warm-optimized fuzzy-neural network for voice-ontrolled robot systems,”*IEEE Trans. Ind. Electron.*, vol. 52, no. 6, pp. 1478–1489,Dec. 2005.