An FPGA Based Architecture for Hidden Markov Models Algorithms

Alejandro Espinosa-Manzo, Aurelio López-López, Miguel O. Arias-Estrada
Graduate Program in Computational Sciences
National Institute of Astrophysics, Optics and Electronics
Luis Enrique Erro No. 1, Sta. Ma. Tonantzintla, Puebla, México. C.P. 72000
P.O. Box 51, 216. Tel: (2) 2 66 31 00, (2) 2 47 20 11 Ext. 8309
aespinos@cseg.inaoep.mx, {allopez, ariasm}@inaoep.mx

Abstract

Hidden Markov Models (*HMMs*) are a powerful tool oriented to the statistical modeling of events. HMMs have been used traditionally in speech recognition and information extraction tasks. These models are considered as a technique in machine learning; in its training phase, they require big volumes of data with several mathematical calculations, consuming considerable computational resources. Similarly, the algorithm for the retrieval phase to produce the answers of the trained model is computationally demanding. The present work proposes the design of an FPGA based architecture, where HMMs algorithms are implemented. These algorithms are highly parallelisable, this feature allows considering them strong candidates to be brought into a hardware architecture with the purpose of improving the implementation and execution of the HMMs.

1. Introduction.

The Hidden Markov Models (*HMMs*) are a powerful machine learning technique. Its use is oriented in the field of the statistical modeling of a series of events, where their statistical properties are characterized depending of the sequence in which they are made. The traditional area of application has been the processing of the speech[1][2], and recently they have also been used in information extraction tasks[3][4]. An HMM is a double stochastic process with an underlying stochastic process that is not observable (it is hidden), but can only be observed through another stochastic process that produce the sequence of observed symbols[5].

There are three algorithms that implement HMMs[5], two of them are used in the training phase of the model, the other one is used in the estimation that makes the model to find the optimal state sequence associated with a given observed sequence. These algorithms are computationally demanding, but have the advantage that they are also highly parallelisable, taking into account these features, they are considered strong candidates to be brought into a hardware architecture, where is possible to implement the techniques of spatial and serial (pipeline) parallel processing, to improve their execution, both techniques are widely used when designing hardware architectures.

Another reason to implement HMMs in a FPGA device is to have available a custom-computing coprocessor for the algorithms related to the HMMs. This coprocessor can free of this task heavy the system where is located, so that the system can attends any others user needs. The designed coprocessor will have a special orientation to the processing of the training phase, where it is necessary to process big volumes of training sequences using different mathematical calculations. In this phase the execution times are high, since the training set is made up of tens of thousands to hundreds of thousands data items. The handling of this high volume of data is necessary because the training data must be representative of the model, since HMMs parameter probabilities have a close relation with the volume of training data. If the resulting trained model used an unrepresentative volume of training data, the final HMM parameters will be far from the event it tries to model.

2. Hidden Markov Models Description.

The elements of a Hidden Markov Model are:

- T - length of the observed sequence (total number of clock cycles)
- O - the observed sequence $O = \{o_1, o_2, \ldots, o_T\}$
- N - the number of states
- Q - set of states $Q = \{1, 2, \ldots, N\}$
- M - the number of symbols
- V - set of symbols $V=\{1, 2, \ldots, M\}$
- A - the state-transition probability matrix.
  $a_{ij} = P(q_{t+1} = j \mid q_t = i) \qquad 1 \le i, j \le N$
- B - observation probability distribution:
  $b_j(k) = P(o_t = k \mid q_t = j) \qquad 1 \le k \le M$
- π - the initial state distribution:
  $\pi_i = P(q_1 = i) \quad 1 \le i \le N$
- λ - the entire model $\lambda = (A, B, \pi)$

The use of the compact notation $\lambda=(A, B, \pi)$ to represent a HMM helps for its implementation, since the specification of a HMM involves the choice of the number of states N, the number of discrete symbols M,

and the specification of the three probability densities A, B, and π[5].

2.1 Forward Algorithm.

The forward algorithm calculates $P(O|\lambda)$ known as forward probability using the following procedure[5]:

- Define forward variable $\alpha_t(i)$ as:
  $\alpha_t(i)=P(o_1, o_2, \ldots, o_t, q_t=i|\lambda)$
- $\alpha_t(i)$ is the probability of observing the partial sequence $(o_1, o_2, \ldots, o_t)$ such that the state q_t is i.
- Induction:

1. Initialization:
$$\alpha_1(i)= \pi_i b_i(o_1)$$
2. Induction:
$$\alpha_{t+1}(j)=\left[\sum_{i=1}^{N}\alpha_t(i)a_{ij}\right]b_j(o_{t+1}) \qquad 1\le t\le T-1 \qquad (1)$$
3. Termination:
$$P(O\mid\lambda)=\sum_{i=1}^{N}\alpha_T(i) \qquad (2)$$

The final result is known as the probability that the HMM assigns to a particular observed sequence, establishing how well the model fits the input.

Also, just as with the forward algorithm, there is another algorithm called backward algorithm. This second algorithm is the mirror image of the first one and is not necessary for the calculation of the forward probability, but is needed for HMMs training algorithm. The backward algorithm is described as follow[5]:

- Define backward variable $\beta_t(i)$ as:
  $\beta_t(i)= P(o_{t+1}, o_{t+2}, \ldots, o_T, q_t=i|\lambda)$
- $\beta_t(i)$ is the probability of observing the partial sequence $(o_{t+1}, o_{t+2}, \ldots, o_T)$ such that the state q_t is i.
- Induction:
  1. Initialization:
  $$\beta_T(i)=1$$
  2. Induction:
  $$\beta_t(i)=\sum_{j=1}^{N}a_{ij}b_j(o_{t+1})\beta_{t+1}(j) \qquad T-1\le t\le 1 \qquad (3)$$

2.2 Viterbi Algorithm.

A formal technique for finding the state sequence that best explains an observation is called the Viterbi algorithm. The key part in the implementation of this algorithm is to recall the best way for each state and its associated probability. The Viterbi algorithm is detailed as follows[5]:

- Initialization:
$$\delta_1(i)=\pi_i b_i(o_1) \qquad 1\le i\le N$$
$$\psi_1(i)=0$$
- Recursion:
$$\delta_t(j)=\max_{1\le i\le N}\left[\delta_{t-1}(i)a_{ij}\right]b_j(o_t) \qquad (4)$$
$$\psi_t(j)=\arg\max_{1\le i\le N}\left[\delta_{t-1}(i)a_{ij}\right] \qquad (5)$$
- Termination:
$$P^*=\max_{1\le i\le N}\left[\delta_T(i)\right]$$
$$q_T^*=\arg\max_{1\le i\le N}\left[\delta_T(i)\right]$$
- Path (state sequence) backtracking:
$$q_t^*=\psi_{t+1}\left(q_{t+1}^*\right) \qquad T-1\le t\le 1$$

2.3 Baum-Welch Algorithm.

The algorithm that adjusts the model parameters to maximize the probability of an observed sequence, is called the forward-backward algorithm or the Baum-Welch algorithm[5]. This algorithm, given a training sequence, adjusts the HMM parameter probabilities to make that training sequence as likely as possible.

The next two steps describe the top-level algorithm for HMM training:

1. Let the initial model be λ_o.
2. Compute new λ based on λ_o and observation O.

The description of the Baum-Welch algorithm is the following[5].

- Define $\xi_t(i,j)$ as the probability of being in state i at
$$\xi_t(i,j)=\frac{\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{P(O\mid\lambda)} \qquad (6)$$
time t and in state j at time t+1.
- Define $\gamma_t(i)$ as probability of being in state i at time t,
$$\gamma_t(i)=\sum_{j=1}^{N}\xi_t(i,j) \qquad (7)$$
given the observed sequence.
- $\sum_{t=1}^{T-1}\gamma_t(i)$ is the expected number of times state i is visited.
- $\sum_{t=1}^{T-1}\xi_t(i,j)$ is the expected number of transitions from state i to state j.

$$\bar{\pi}_i = \gamma_1(i) \qquad 1 \le i \le N \tag{8}$$

$$\bar{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)} \tag{9}$$

$$\bar{b}_j(k) = \frac{\sum_{t=1,o_t=k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)} \tag{10}$$

The reestimation formulas for π, A, and B are[5]:

3. Proposed Architecture.

The designed hardware architecture has been implemented in a single FPGA device, which works with two external memory banks, where the HMM and the observed sequence to evaluate are stored. The architecture was prototyped with a new hardware description language called Handel-C[10]; this language based in ISO/ANSI-C offers a novel approach for hardware architectures design. Using this language, a design can be compiled from Handel-C to an FPGA net list without an intermediate HDL step, reducing the costly HDL design loop.

The architecture is integrated in two main schemes: the first scheme implements the forward and Viterbi algorithms that can be executed separately, but not simultaneously. The second scheme is fully dedicated to the training algorithm implementation. Both schemes consider the use of a 32 bits floating point representation (IEEE 754 standard) for values calculated with HMM algorithms, and also is considered the design of modules that implement floating point arithmetic; where the operations of addition, multiplication and division are executed according to the algorithms requirements.

3.1 Hardware Design of Forward and Viterbi Algorithms.

The first scheme describes the full implementation of forward and Viterbi algorithms, running separately. The architecture top level of this scheme is shown in figure 1. As one can see in the figure, a control signal manipulated by the user selects which algorithm will be executed, using during this execution, the HMM and the observed sequence stored in the external memory.

If the Viterbi algorithm is selected by the user, the FPGA based architecture implement the 3 necessary stages for the algorithm: the first stage applies equations 4 and 5, the second stage finds the index of the sequence with highest probability, and the final stage establishes the sequence of states corresponding to this index. On the other hand, if the forward algorithm is selected, the architecture implements the algorithm in 3 stages: the first stage initializes the values of the matrix, the second stage calculates the rest of the matrix values (equation 1), and the third stage calculates the resulting forward probability (equation 2).

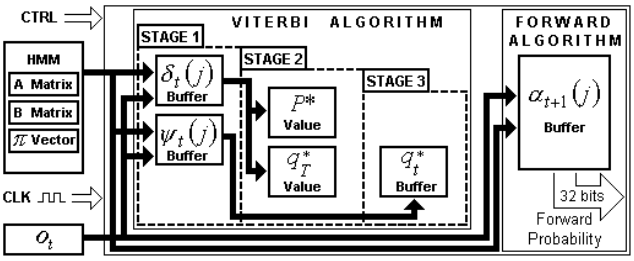

Fig. 1. Architecture top level. The FPGA implements the Viterbi and forward algorithms.

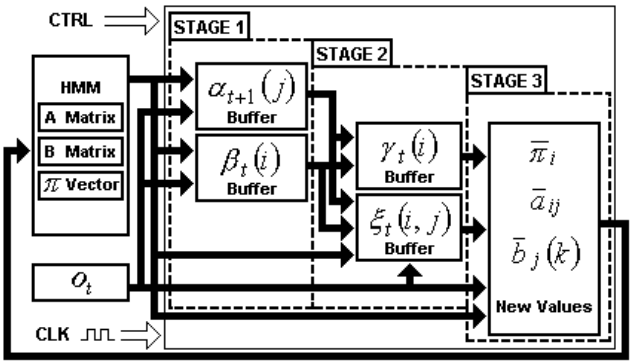

Fig. 2. Architecture top level. The FPGA implements the training algorithm (Baum-Welch).

3.2 Hardware Design of Training Algorithm.

The figure 2 shows the hardware architecture designed for the second scheme, where the training algorithm is implemented. The FPGA based architecture describes the three necessary stages to implement the Baum-Welch algorithm: The first stage calculates the forward algorithm (equations 1 and 2) and the backward algorithm (equation 3). The second stage calculates the intermediate values defined with equations 6 and 7. The third stage considered as the most important on the algorithm is fully executed in parallel, here the adjusts of the HMM parameter probabilities is made with the equation 8, 9 and 10. The

FPGA device receives by a control signal the user indication to start the algorithm execution.

4. Implementation and Results.

The proposed architecture was synthesized for a XCV2000E Virtex family device from Xilinx, using the Xilinx ISE 4 tool. The designed custom-computing processor was prototyped in a RC1000 card made by Celoxica Ltd[11]. The architecture has the limit to implement models with a maximum of 126 states (*N*) and 126 symbols (*M*). The evaluation of these models is limited to use pieces of observed sequences with a maximum length of *T*=126.

The validation of the results was made by comparing against a HMMs software implementation running in a PC with a Pentium processor at 500MHz, keeping some others processes simultaneously in the same system when executing the HMM algorithms.

The synthesis results of the first scheme (Viterby & forward algorithms) are summarized as follows: the scheme requires 7503 out of 19200 (39%) slices, 116Kgates consumed, with an operation clock frequency of 40MHz. The performance of the FPGA implementation of this scheme did not beat the software implementation, evaluating in both a model with 126 states and 126 symbols, using an observed sequence with *T*=126. The Viterbi algorithm running in the PC was 9.71% faster, the FPGA consumed 3.645 seconds, with respect to forward algorithm the PC was 39.44 % faster, the FPGA consumed 9.644 seconds. But it is important to note that the scheme only use the 39% of the FPGA device, allowing a second implementation of the algorithms, taking into account this, it is possible to evaluate two models in parallel.

The second scheme (training algorithm) during its synthesis gave the next results: 19198 out of 19200 (99%) slices required, 297Kgates consumed, and an operation clock frequency of 20MHz. In this case the performance of the FPGA implementation was 3.1% faster that the software execution, evaluating the same model described previously. The FPGA execution used 11.106 seconds. It is important to mention that training is the heavy process when applying HMM to model events, since requires big volumes of data.

5. Conclusions.

This paper describes a complete implementation of HMM algorithms, through an FPGA based architecture, used as the foundation for the complete design of a dedicated processor or coprocessor. The resulting custom-computing processor will be dedicated to provide the HMMs evaluation services to users who want to model processes using the HMMs approach.

This work also gives a clear idea about how the implementation of a specific task or algorithms through hardware architectures, establishes good improvements in its performances, especially if these algorithms have parts that allow an execution in parallel. This feature can be highly exploited with the parallel process techniques widely used in the designs of hardware architectures.

An important contribution obtained through the HMMs hardware implementation is related to its training phase, since this work define a device which can be entirely dedicated to this procedure, allowing with this the evaluation of high volumes of data with a minimum consumption of time, obtaining a better adjust of the HMM parameter probabilities.

6. Acknowledgement.

This work was sponsored by a research grant of Conacyt México (R31886-A).

References.

[1] L. R. Rabiner and B. H. Juang, “An Introduction to Hidden Markov Models”, IEEE ASSP Magazine, January 1986.
[2] Eugene Charniak, “Statistical Language Learning”, MIT Press, 1993. ISBN 0-262-03216-3.
[3] Kristie Seymore, Andrew McCallum, and Roni Rosenfeld , “Learning Hidden Markov Model Structure for Information Extraction”, AAAI'99 Workshop on Machine Learning for Information Extraction, 1999.
[4] Ion Muslea. “Extraction Patterns for Information Extraction Tasks: A Survey”, In the AAAI Workshop, pag. 1-6, Orlando, Florida, 1999.
[5] Lawrence R. Rabiner, “A tutorial on hidden Markov models and selected applications in speech recognition”, Proceedings of the IEEE , 77(2):257-286, 1989.
[6] Christopher D. Manning and Hinrich Schutze, “Foundation of Statistical Natural Language Processing”, MIT Press, 1999. ISBN 0-262-13360-1.
[7] S.J. Melnikoff, P.B. James-Roxby, S.F. Quigley and M.J. Russell, “Reconfigurable Computing for Speech Recognition: Preliminary Findings”, Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing, Springer, pp. 495-504, 2000.
[8] D. Patterson, J. Hennessy. “Computer Organization & Design: The Hardware/Software Interface”, 2nd Ed. Morgan Kaufmann Publishers, 1997.
[9] Stephen Brown and Jonathan Rose, "Architecture of FPGAs and CPLDs: A Tutorial", IEEE Design and Test of Computers, Vol. 13,No. 2, pp. 42-57, 1996.
[10] “Handel-C Language Reference Manual version 2.1”, Celoxica Ltd, Oxfordshire U. K, http://www.celoxica.com
[11] “RC1000-PP Hardware Reference Manual”, Celoxica Ltd, Oxfordshire U. K, http://www.celoxica.com