A Vector Quantizer Architecture for an Automatic Speech Recognizer

A. J. Araújo[†‡], V. C. Pera[†], C. C. Espain[†], J. S. Matos[†‡]

[†]FEUP – Faculdade de Engenharia da Universidade do Porto
[‡]INESC – Instituto de Engenharia de Sistemas e de Computadores
Pr. da República, 93 – Apartado 4433 – 4007 Porto CODEX – Portugal
Tel:+351-2-2094220 Fax:+351-2-2084172
E-mail: aaraujo@picasso.inescn.pt, vpera@fe.up.pt

Abstract

Computationally intensive algorithms are excellent candidates for acceleration using custom computing machines optimized for a particular application. This is the case of vector quantization (VQ), a time consuming task in an automatic continuous speech recognizer (CSR), due to the floating point arithmetics involved. For a real-time application of a CSR system, hardware acceleration is the key to achieve the high performance required.

The architecture of a specific processor to perform the VQ task is presented. The implementation using FPGAs in a reconfigurable computing platform is also described. Finally, a set of results are presented and discussed. Some remarks are included about the modularity and scalability of the developed architecture, supported by the obtained results.

1 Introduction

State of the art CSR systems use hidden Markov models (HMMs) [1]. Although it is accepted there is some advantage, often not significant, on the recognition performance if the system uses continuous or semi-continuous HMMs, those models are more complex than discrete type ones (DHMMs). Therefore, DHMMs are often used [2]. CSR systems based on those models require a vector quantizer, which introduces a significant volume of floating point computation. In order to handle it, we developed a modular and scalable architecture that we call Vector Quantization Processor (VQP). An implementation was done on a general purpose reconfigurable system based on XILINX FPGAs.

Next section presents an overview of the CSR that was implemented in order to do a set of experiments in speech recognition. The VQP architecture for the VQ module is presented in section 3. Practical considerations about the implementation are made in section 4. The results of both software and hardware implementations are discussed in section 5. Last section outlines the main conclusions and points out future developments.

This work was partially supported by the Portuguese government under the program PRAXIS XXI Proj. 2/2.1/TIT/1558/95.

2 The CSR system

The CSR system we used recognizes strings of digits continuously spoken in English (digits 'zero' to 'nine', plus the word 'oh') [3]. It is based on DHMM modelling at word level and uses a connected speech approach. Figure 1 shows the sequence of steps performed by the system.

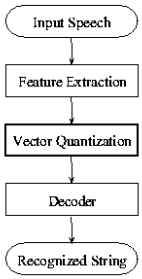

Figure 1: Modular structure of the CSR system.

The feature extraction module transforms the speech signal into a convenient representation [4]. Every 9 ms intervals, the signal is sampled and represented by a feature vector of 26 floating point elements. It contains a normalized energy term, 12 mel-frequency cepstrum coefficients [1] obtained from the short-time power spectrum, and the respective first derivatives (delta terms). The number of feature vectors we have to process grows linearly with the signal duration, and a total of 300 vectors are collected for a speech signal with a duration of 2.7 s.

The following module in the CSR chain deserves our particular attention. It transforms each feature vector into a scalar, using a codebook with 256 codevectors previously calculated by the LBG clustering algorithm [1]. For each vector V, this module outputs the index of the closest codevector, according to the squared Euclidean distance. The VQ algorithm is as follows:

Algorithm 1: VECTOR QUANTIZATION ALGORITHM
Input: Features vector $V = \{v_0, v_1, \ldots, v_{25}\}$.
Output: The codebook index of the closest vector.

```
SEARCH(V)
    Index ← 0
    MinDist ← ED(CodeBook(0), V)
    for i ← 1 to 255
        Dist ← ED(CodeBook(i), V)
        if Dist < MinDist then
            Index ← i
            MinDist ← Dist
    return (Index)
ED(X,Y)
    Sum ← 0
    for i ← 0 to 25
        Sum ← Sum + (X(i) − Y(i))²
    return (Sum)
```

Now that the input speech is given by a sequence of scalars, the recognizer can use DHMMs. The recognition is performed by the *one stage* algorithm, optionally extended with a *level building* procedure [5] to account for a fixed number of digits per string. To improve the recognition rate, duration statistics (mean and variance) of each one of the spoken digits were used.

3 The VQP architecture

The VQP is intended to work as a co-processor for a host computer, where the entire CSR system runs. The communication between them is done by an additional module that establishes the interface with the PC bus. The designed architecture closely follows the VQ algorithm presented above. Initially, the host stores the codebook into local VQP memory. For each feature vector, the squared Euclidean distance computation to every codevectors $\sum_{i=0}^{25}(X_i - Y_i)^2$ is performed. After the corresponding index is calculated, the VQP module returns this value.

The VQP architecture is presented in figure 2 and its main blocks are detailed in the next sections.

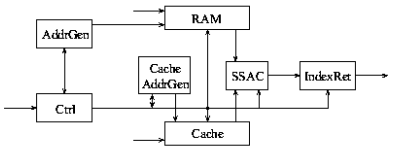

Figure 2: VQP architecture.

3.1 The SSAC Unit

This is the largest and time critical part of the VQP architecture. The Subtract-Square-ACcumulate (SSAC) unit (see figure 3) performs the floating point operations needed to compute the squared Euclidean distance. It is implemented using a pipelined scheme presenting a latency of three clock cycles, and the included arithmetic units are not internally pipelined.

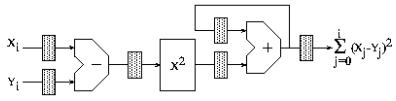

Figure 3: Subtract-square-accumulate unit.

The subtracter and the adder are identical units. The first one uses an adder where the sign bit of the second operand is inverted. The x^2 operator is based on a floating point multiplier that uses an optimized integer square function to perform the square of operand's significand [6]. In this arithmetic unit the critical path is the carry propagation chain. To reduce this propagation delay we used the *fast carry logic* [7] feature available for XILINX 4000E family. All of these operators use a 16-bit wide floating point format to store the operands. The 9 LSBs are used for the significand, the exponent uses the next 6-bits and the sign bit occupies the MSB. With this format the range of representable floating point numbers is from $\pm 9.313 \times 10^{-10}$ to $\pm 8.5815 \times 10^{9}$.

Because floating point numbers are stored in sign and magnitude form, the multiplier needs only to deal with unsigned integer numbers and normalization. Contrasting with it, in the adder/subtracter, a lot of operations must be done before the addition of the significands. So, it is necessary to align the significands of the operands, which requires comparing the absolute values of the operands and subtracting the exponents in order to calculate the number of shifts needed to do the alignment. As a result, the adder/subtracter units are more complex to implement than the multiplier, contrasting with what happens with the correspondent integer arithmetic units. The results that we later present reflect this on the implementation area

and speed obtained. The complete algorithms used to implement the arithmetic units are well known and can be find in [8, 9, 10, 11].

3.2 Cache and External RAM

The cache was implemented in the FPGA itself using the XILINX 4000E internal RAM feature [7] with LogiBLOX module generator of M1 tools. This cache consists of two independent RAMs of $26 \times 16 = 416$ bytes, and is intended to hold the vector V that is repeatedly accessed during it's processing. While one of these RAMs holds the vector V for processing, the other one is loaded with the next V (figure 4). In this way, when the computation for the current V terminates, the roles of the two RAMs are inverted and the VQP doesn't need to wait for a new incoming vector to initiate a new computation.

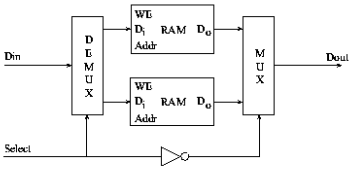

Figure 4: Cache organization.

The RAM block that we see in figure 2, consisting of 13 Kbytes, is used to permanently hold the codebook. It is written once at system start-up.

3.3 The Index Return Unit

The purpose of this module is to maintain an actualized index of the closest codevector during the codebook search. For each new vector V, the control unit sets the initial index to the first entry in the codebook. This index is always changed when a new closest codevector is achieved. After an entire scan of the codebook, the index return unit signals to the control unit that the search operation is finished and the result is available. The host computer collects the indexes, one for each feature vector, to perform the decoding task in the last stage of the speech recognition sequence (figure 1).

4 Implementation

We used Abel HDL to specify the control unit. All of the other logic circuits, including the floating point operators, were designed with the ViewLogic schematic capture and simulation tools, and implemented with XILINX M1 tools.

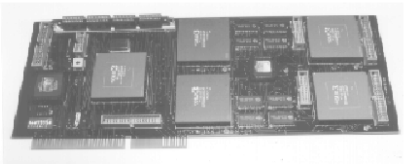

Figure 5: The RVC board.

To implement and test the described architecture we used RVC, a reconfigurable system for vector processing applications [12, 13]. It consists of a PC expansion board (figure 5) with five XILINX FPGAs, two 4010E-4 and three 4013E-4 [7], equipped with 1 Mbyte of fast RAM intended to work as vector memories, providing a fast access for computation cores running in the FPGAs. The system is configured by a program running in the host computer, that downloads the configuration bit-streams for every FPGAs. The board is actually equipped with the referred FPGAs, but it is compatible with any FPGA of the XILINX 4000E family.

4.1 Partitioning of VQP

Looking at figure 6 we can see the organization of the system, and the mapping of the VQP blocks described in the previous section into actual devices on the RVC board. X1 and X2 are 4010E-4, and X0, X3 and X4 are 4013E-4 FPGAs.

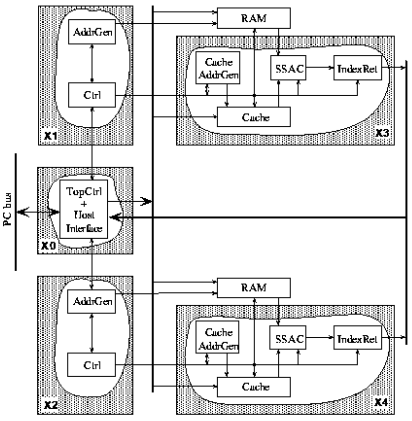

Figure 6: The double VQP implementation on the reconfigurable system.

When the system is initiated, the codebook is loaded into the board memory, and after this it is configured to perform the VQ tasks. Figure 6 shows that the whole system is composed of two instances of the VQP working in parallel. Together, X1 and X3 hold one VQP, while the other instance is distributed by X2 and X4. X0 generates the system's top level control and performs the interface with the host bus. At the moment an ISA interface is used, which does not introduce any time constraints on the data flow, because the time spent with computation is greater than the one spent with data transfers. The address generation units are not located in the same FPGA of the VQ cores (X3 and X4) due to the fact that the address generation for on-board memories must be physically done by X1 and X2[1]. Placing the local control out of the VQP core simplifies the interface needed between the FPGAs and turns the VQP unit modular and easier to simulate. This distributed control contributes to a high modularity and scalability of the whole system, facilitating an expansion when required.

4.2 Data flow

Figure 7 shows the activity of FPGAs X0, X3 and X4, and helps to understand the sequence of events. X1 and X2 are used for address generation purposes and to control the overall operation, and are omitted from this diagram. The impact of the cache organization that was described can be observed. For example, while a vector V_i is loaded into X3, the vector V_{i-2} is being processed, and at the end of this, starts the processing of V_i in parallel with the return of the V_{i-2} index.

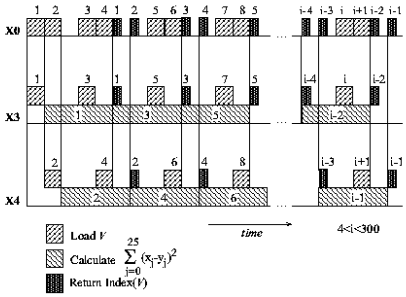

Figure 7: Temporal diagram for the system operations.

[1]This constraint was motivated by the amount of free I/O pins available on the X3 and X4 FPGAs.

5 Results

Data about the size of the most significant blocks, relative to an entire VQP, can be seen in table 1. It presents also the post-layout delay time obtained for the floating point operators used in SSAC and VQP. We collect these results for both 4013E-4 and 4013E-1 FPGAs to show the speed-up that we can obtain if we use faster devices. To give an idea of size, the part of the VQP implemented on X3 (a 4013E, with 576 CLBs), as shown in figure 6, occupies about 72% of it.

Block	*Relative size(%)*	$t_d(ns)$	
		4013E-4	4013E-1
$+,-$	26.9	141.2	83.0
x^2	20.2	100.9	56.7
SSAC	80.4	143.7	87.6
VQP	100	143.2	84.4

Table 1: Some implementation results.

For different placement and routing solutions, we can observe small variations on the maximum delay time. For example, the SSAC unit, that limits the maximum frequency of the system to about 7 MHz, presents a delay time of 143.7 ns while the adder presents a corresponding time of 141.2 ns.

The experiments we used to compare both software and hardware solutions consists of a set of 300 feature vectors, extracted from a speech signal corresponding to a string of five continuously spoken digits. Each one of these vectors consists of 26 floating point numbers. The time spent to process each vector (compute the squared distance to each codevector and generate the index of the closest one) is approximately $256 \times (26+4) \times 143\,\text{ns} = 1.1$ ms. The term 4 is due to the three pipeline stages included in the SSAC unit and one additional clock cycle needed to output the index.

An important conclusion that we can take from the temporal diagram shown in figure 7 is that the initial loads of V_1 and V_2, and the final result retrieved for V_{300}, are operations that consume an insignificant time when compared with all the other operations. So, we can ignore this time to deduce the complete time needed to process the set of 300 feature vectors, which will be 330 ms. Since the implementation described uses two VQP units working in parallel the total time to process this entire set is 115 ms. In each clock cycle of a VQP, three operations $(-, x^2, +)$ are performed, so a total of 42 MFLOPS is achieved by the entire system.

The CSR system was implemented in C language, where float *type* variables were used for feature vectors. The *float* type uses a floating point format with

32-bit wide, where 23 bits are used to store the significand. Analyzing the codebook contents, we see that a significand with 9 bits is enough. So, this customized floating point format and the pipelined SSAC unit contribute for the good results that the hardware presents.

Table 2 shows the execution time results for software and hardware implementations of the VQ module. It also shows the impact of these results on the overall CSR system, where the VQ effort is approximately one third. The execution times for the SW approach were obtained by running the CSR system on a Pentium MMX PC at 266 MHz.

Module	*Time(s)*		*Gain(%)*
	SW	HW	
VQ	2.510	0.115	95
CSR	7.240	4.885	33

Table 2: Time comparison results.

The most significative of the obtained results is the evident hardware acceleration of the VQ module, by a factor of about 22, in spite of the current implementation of arithmetic operators having not included pipelining. This shows that the performance reached can still be improved. We are studying now how many levels of pipelining can be inserted, and the impact this brings on area requirements within the FPGAs.

6 Conclusion

The VQ computation time is an important part of the overall execution time of the CSR system. With the goal of decreasing this host's computation effort we presented a modular architecture for the VQ module. An implementation of this architecture was made on an FPGA based platform. The resources available on this system allow an implementation with two computation cores working in parallel. A very significant reduction of the execution time was achieved, allowing the VQ module to be 22 times faster than the software solution. This results in a scalable solution, as additional computation cores may be added for increased performance.

In order to enhance the actual performance we are planning to implement pipelined versions of the arithmetic operators.

References

[1] J. Deller and J. Proakis, *Discrete-Time Processing of Speech Signals*. Macmillan Publishing Company, 1993.

[2] K.-F. Lee, *Automatic Speech Recognition: The Development of the SPHINX System*. Kluwer Academic Publishers, 1989.

[3] R. G. Leonard, "A database for speaker independent digit recognition," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, March 1984.

[4] P. Loizou and A. Spanias, "High-performance alphabet recognition," *IEEE Transactions on Speech and Audio Processing*, vol. 4, pp. 430–445, November 1996.

[5] L. Rabiner and B.-H. Juang, *Fundamentals of Speech Recognition*. Prentice Hall, 1993.

[6] A. Eshraghi, T. S. Fiez, K. D. Winters, and T. R. Fischer, "Design of a new squaring function for the Viterbi algorithm," *IEEE Journal of Solid-State Circuits*, vol. 29, pp. 1102–1107, September 1994.

[7] Xilinx, *The Programmable Logic Data Book*, 1996.

[8] D. Goldberg, "What every computer scientist should know about floating point arithmetic," *ACM Computing Surveys*, vol. 23, March 1991.

[9] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, ch. Appendix A. Morgan Kaufmann Publishers, 1990.

[10] N. Shirazi, A. Walters, and P. Athanas, "Quantitative analysis of floating point arithmetic on FPGA based custom computing machines," in *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 155–162, IEEE Computer Society Press, April 1995.

[11] A. R. Omondi, *Computer Arithmetic Systems – Algorithms, Architecture and Implementations*. Prentice Hall, 1994.

[12] J. C. Alves, A. Puga, L. Corte-Real, and J. S. Matos, "FPGA implementation of a vector processor for the estimation of higher-order moments," in *Proceedings of the XII Design of Circuits and Integrated Systems Conference*, pp. 759–763, November 1997.

[13] J. C. Alves and J. S. Matos, "RVC – a reconfigurable coprocessor for vector processing applications," in *Proceedings of the 6th Annual IEEE Symposium on FPGA Custom Computing Machines*, April 1998.