VECTOR QUANTIZER ACCELERATION FOR AN AUTOMATIC SPEECH RECOGNITION APPLICATION

A. J. Araújo[1,2], *V. C. Pera*[1], *M. N. Souza*[3]

[1]FEUP – Faculdade de Engenharia da Universidade do Porto
[2]INESC – Instituto de Engenharia de Sistemas e de Computadores
Pr. da República, 93 – 4007 Porto Codex – Portugal
E-mail: aaraujo@picasso.inescn.pt, vpera@fe.up.pt
[3]UFRJ – Universidade Federal do Rio de Janeiro

ABSTRACT

For a real-time application of an automatic speech recognition system, hardware acceleration can be the key to reduce the execution time. Vector quantization is an important task that a recognizer based on discrete hidden Markov models must perform. Due to the amount of floating point operations executed, the vector quantizer is an excellent candidate to be accelerated by customized hardware. The design, implementation and obtained results of a hardware solution based on field programmable gate array devices are presented.

1. INTRODUCTION

The usage of Automatic Speech Recognition (ASR) in real-time applications demand increasingly faster processing. In particular, the most computationally intensive algorithms need to be accelerated as much as possible. This is the case of Vector Quantization (VQ), a time consuming task in ASR applications based on discrete hidden Markov models (DHMMs) [1]. One way of achieving this acceleration is by using optimized custom computing machines in terms of data formats and arithmetic operators needed. This approach was used in the Vector Quantization Processor (VQP) based on Field Programmable Gate Arrays (FPGAs) we developed.

Next section describes the ASR application implemented emphasizing the VQ step. Section 3 presents the architecture of the VQP and describes its implementation in a board equipped with XILINX FPGAs. Obtained results are presented and discussed in section 4. Finally, section 5 shows some conclusions about the work presented and remarks on future improvements that can be made.

2. THE AUTOMATIC SPEECH RECOGNIZER

2.1 General Description

The recognizer can be viewed as performing a sequence of operations according to the block diagram of figure 1. Speech is first submitted to anti-aliasing filtering, sampled at 11025 Hz and pre-emphasized with a first-order filter of the form $(1\text{-}0.97z^{-1})$. This signal is segmented into 23 ms Hamming windowed frames with an overlap of 14 ms between consecutive frames. For each frame, a set of 22 mel-scaled triangular bandpass filters is applied to the short-time power obtained by a 512-point FFT. Using a discrete-cosine transform, 12 mel-frequency cepstral coefficients are computed from these filterbank log-energies and then submitted to a sine-liftering window [4]. After the cepstrum mean subtraction, these coefficients plus the normalized energy term and their first derivatives form each 26-D vector.

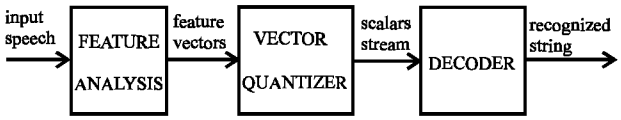

Figure 1: A block diagram of the speech recognizer.

After the quantization (section 2.2), the decoding is carried out by the *one-stage* algorithm extended with a *level building* procedure [5] to account for a previously known string length. For each digit, modeled by one DHMM with 5-state Bakis topology, duration statistics assuming a gaussian distribution are used.

For training, each model was first seeded running 4 iterations of the Baum-Welsh algorithm [1] over 100 samples from the TI database [6]. Then, an embedded 4 iterations of the Baum-Welsh algorithm was performed over 600 5-digits strings. Tests done on 100 5-digits strings from the same database resulted on 94.2% digit accuracy.

The ASR was implemented in C language and the results that we will later present were obtained running it on a Pentium MMX PC at 266 MHz.

2.2 Vector Quantization

In the ASR application the VQ operation deserved our particular attention. It transforms each feature vector **V** into a scalar i^* using a codebook with dimension N=256 previously calculted by the LBG clustering algorithm [1]. For each vector **V** this module outputs the index i^* of the closest codevector $\mathbf{C}^{(i)}$. As a distance measure on the D=26 dimension space the Euclidean norm was used.

$$i^* = \arg \min_{1 \le i \le N} \left\{ \sqrt{\sum_{j=1}^{D} \left(v_j - c_j^{(i)} \right)^2} \right\}$$

3. DESIGN AND IMPLEMENTATION OF THE VECTOR QUANTIZER

3.1 Introduction

The time spent with the VQ can be decreased with a dedicated hardware implementation. The *float* type used in the ASR C source code is the shortest available format for real numbers, presenting an excess of bits for the required precision of data values. Using a reduced format for floating point values allows an acceleration of the algorithms that implement the floating point operators.

VQP was developed to implement the VQ operation working as an auxiliary processor of a host computer where the ASR application runs.

3.2 Architecture

Figure 2 shows the designed architecture. It closely maps in hardware the VQ algorithm: initially, the host stores the codebook into a RAM to be used by the VQP and, for each feature vector, the squared Euclidean distance to each codevector is computed and the index of the closest one is returned to the host.

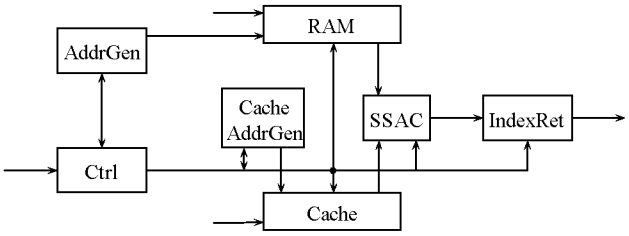

Figure 2: VQP architecture.

The main blocks of QP are described below.

Floating point core. This is the largest module of VQP, where floating point computations are performed. It appears in figure 2 with the name of SSAC (Subtract-Square and ACcumulate). As a measure of distance the square of the Euclidean distance is enough, requiring a subtracter, an adder and a square function. Figure 3 shows the computation chain with registers to permit a pipeline operation with a three clock cycle latency.

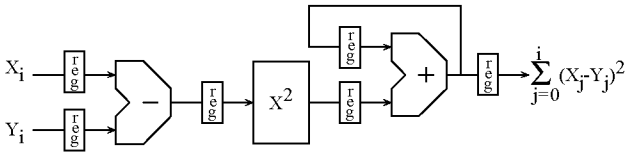

Figure 3: Floating point unit.

The complete algorithms for floating point operators that were implemented are well known and can be find in [8,9,10]. The adder/subtracter units are more complex to implement than the multiplier [9], contrasting with what happens with the corresponding integer arithmetic units. As a consequence of this the implementation area is higher and the speed of the circuit is lower.

The x^2 operator limits the clock frequency of the floating point unit. By this reason several optimizations were introduced. It was implemented by a multiplier that uses an optimized integer square function to perform the square of operand's significand [6]. The critical path occurs in the carry propagation chain of this operator. Another optimization was introduced by using the *fast carry logic* [7] feature available for XILINX 4000E FPGA family. Figure 4 presents an extract of the square operator schematic mapped into the FPGA resources, showing the carry chain (dotted lines) with dedicated input/output (CIN, COUT) ports.

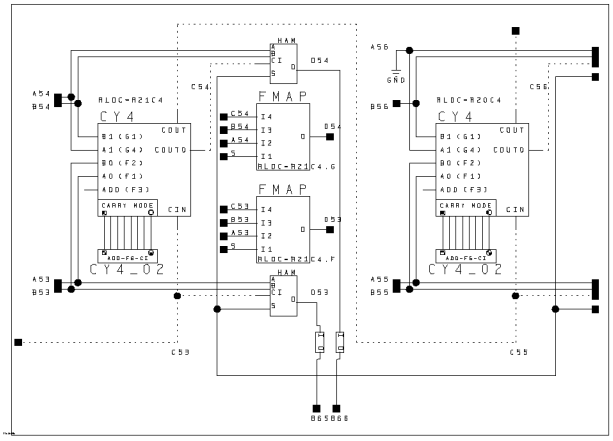

Figure 4: FPGA mapping using *fast carry logic*.

Figure 5 shows the customized 16-bit wide floating point data format consisting on 9 bits for significand field, 6 bits for the exponent and the remaining bit for the signal. With this format the range of representable real numbers is from $\pm 9.313\times10^{-10}$ to $\pm 8.582\times10^{9}$.

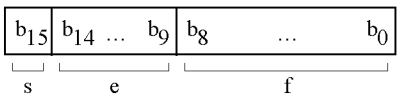

Figure 5: Floating point format.

Memory organization. The RAM block that we see in figure 2 has a size of 13 Kbytes and is used to store de codebook. It is written at system start-up. Due to intensive access of each feature vector during the squared distance computation, it is loaded on a cache memory implemented in the FPGA. This is

possible by using the XILINX 4000E internal RAM feature [7] using the LogiBLOX module generator of XILINX M1 tools. This cache consists of two independent RAMs of $26 \times 16 = 416$ bits. While one of these RAMs holds the vector under processing, the other one is loaded with the next vector. Therefore, when the processing of the current vector finishes, 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.

Input/Output. For each feature vector, VQP returns an index that indicates the closest codevector. The control unit sets the initial index to the first entry in the codebook and updates it whenever a closest codevector is found. The host computer receives the computed indexes needed to the decoding task that is performed by the last phase of the speech recognition application.

3.3 Implementation

The implementation of VQP was done using reconfigurable logic circuits of type FPGA. They offer a short design cycle time and its reconfigurability feature can be explored to adapt the existing hardware to different data formats and to other dimensions of feature vectors.

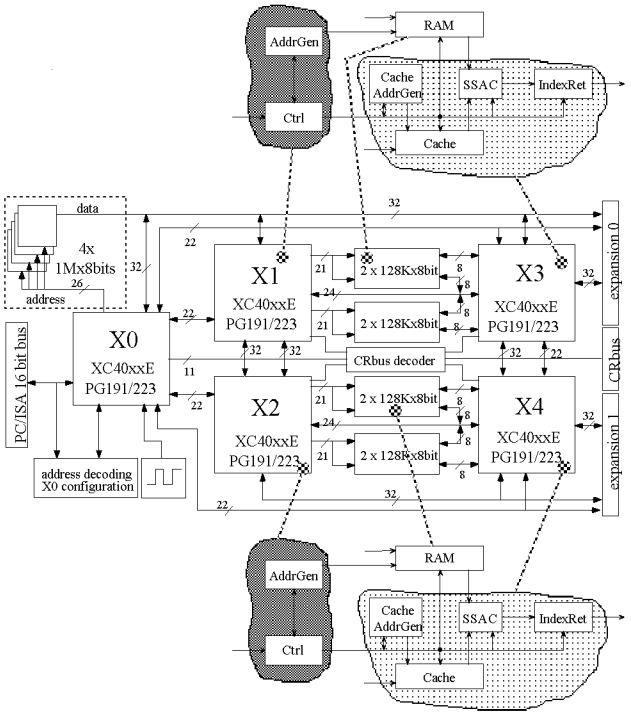

Figure 6: The RVC board supporting two instances of VQP.

The RVC board. For VQP's implementation we used RVC (Reconfigurable Vector Co-processor), a general purpose reconfigurable system, essentially developed for vector processing applications [11,12]. In figure 6 we can see the organization of RVC. It is composed of a PC expansion board with five XILINX FPGAs, two 4010E-4 and three 4013E-4 [7], and is equipped with 1 Mbyte of fast RAM intended to work as vector memories providing a fast access for computation units running in the FPGAs.

The system is configured by a program running in the host computer, that downloads the configuration bit-stream for each FPGA, whenever hardware reconfiguration is needed. Although equipped with the referred FPGAs, the board is compatible with any FPGA of the XILINX 4000E family.

Partitioning on RVC. The RVC available resources allow two VQP instances to operate in parallel. Figure 6 illustrates the instantiation of VQP blocks in RVC FPGAs. While X1 and X3 hold one VQP, the other VQP instance is distributed by X2 and X4. In the top of the hierarchy, X0 generates the top level control for the whole system and interfaces with the host bus. An ISA interface was used, and in spite of its low transfer data rate, it does not introduces any time constraint on data transfers because the time spent with computation is greater than the one wasted with data flow. To give an idea of FPGA occupation, X3 and also X4 are about 80% occupied, with the arithmetic operators, corresponding to approximately three quarters of the total area.

Task scheduling. Figure 7 helps to understand how the VQP units work in parallel, showing the main data flow that occurs in FPGAs X0, X3 and X4.

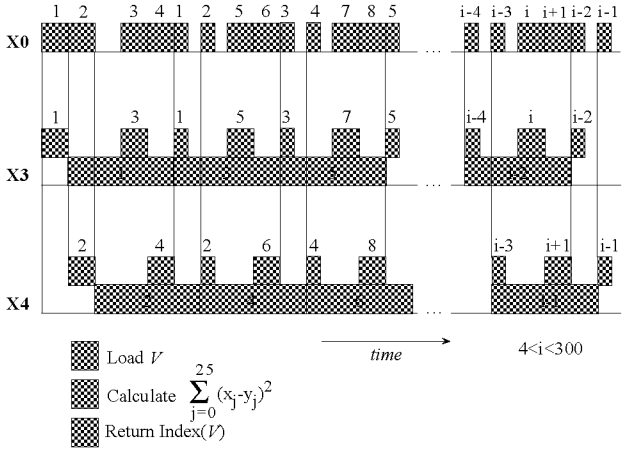

Figure 7: Sequencing the main operations in the FPGAs.

The influence of cache usage (see section 3.2 about memory organization) can be observed in this temporal sequence. While a vector V_i is loaded into X3, the vector V_{i-2} is being processed, and at the end of this, the processing of V_i starts in parallel with the return of the V_{i-2} computed index.

4. RESULTS

The experiments we used to compare both software and hardware solutions consist 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 $256 \times (26+4) \times 143\,\text{ns} = 11\,\text{ms}$. 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 6 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 total time (330 ms) needed to process the set of 300 feature vectors.

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.

Module	Time (s)		Reduction (%)
	SW	HW	
VQ	2.510	0.115	95
ASR	7.240	4.885	33

Table 1: Improvements on the execution time of VQ and ASR.

Table 1 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 ASR system, where the VQ effort is approximately one third.

5. CONCLUSION

A hardware solution was presented to accelerate execution time of an ASR application. An implementation of the architecture was performed 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. Features like modularity and scalability of the architecture of VQP were emphasized, since they can be explored to obtain a VQ with several VQPs working concurrently to increase the system performance.

The main system clock limitation is due to the floating point arithmetic operators. Future work is planned to increase the performance by using pipelining techniques that will improve the data throughput.

6. REFERENCES

1. J. Deller and J. Proakjs, 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. Ficher, "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. 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.
10. A. R. Omondi, Computer Arithmetic Systems - Algorithms, Architecture and Implementations. Prentice Hall, 1994.
11. 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.
12. 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.