We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS BOOK CITATION INDEX INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

8

Efficient VLSI Architecture for Memetic Vector Quantizer Design

Chien-Min Ou[1] and Wen-Jyi Hwang[2,*]
[1]*Department of Electronics Engineering, Ching-Yun University, Chungli,*
[2]*Department of Computer Science and Information Engineering,*
National Taiwan Normal University, Taipei,
Taiwan

1. Introduction

Memetic algorithms (MA) (Eiben & Smith, 2003; Molina, Lozano & Herrera, 2010; Moscato, 1999; Santos & Alves, 2010) have been found to be effective for evolutionary computation (Areibi, Moussa & Abdullah, 2001; Merz & Freisleben, 1999). It can be viewed as the hybrid genetic algorithms (GA) (Eiben & Smith, 2003) consisting of local refinement to genetic search results. Because the algorithms involve both the global and local searches, one challenging issue of the MAs is to reduce the computational complexity. One simple way to lower the computational time is to reduce the population size. Nevertheless, this approach is not favorable because small population size usually traps the MA in poor local optimum.

Another way to accelerate the execution of the MA is to implement the algorithm in hardware (Hwang, Hsu, Li, Weng & Yu, 2010). However, existing hardware architectures are mostly designed only for GA. No architecture for local refinement is available. In addition, existing GA hardware implementations (Choi & Chung, 2000; Nedjah & Mourelle, 2005; Tommiska & Vuori, 1996) have a number of drawbacks. First of all, large storage size is required for processing the genetic strings. Usually two set of population memories are used for the regeneration process. One memory contains the parent strings; the other stores the child strings after the regeneration. Moreover, there is overhead for switching one memory to another at the beginning of a new generation. The second drawback is that the regeneration process is based on the fitness function. The selection of parents therefore may need large chip area for hardware implementation. The mutation and crossover operations also result in high area cost when concurrent processing over all the genetic strings is desired. The third drawback is that the existing GA architectures (Choi & Chung, 2000; Nedjah & Mourelle, 2005; Tommiska & Vuori, 1996) contains only single population. Distributed or parallel evolutions are usually desired for attaining a near global optimal performance.

The objective of this paper is to present a novel hardware architecture for fast parallel MA optimization. In this chapter, we consider the applications of MA for vector quantizer (VQ) (Gersho & Gray, 1992) design. When applied for VQ training, the MA requires large storage

* Corresponding Author

size and long training time (Hwang & Hong, 1999). Therefore, the VQ design is a good example for verifying the effectiveness of the proposed MA architecture.

In the proposed architecture, each population of the parallel MA is associated with a hardware module for independent memetic evolutions. Each hardware module consists of population memory unit, mutation and crossover unit, C-Means (Gersho & Gray, 1992) unit, and survival test and update unit. In our design, the mutation and crossover unit is used for global search, while the C-Means unit is used for local refinement.

Each hardware module contains only one population memory for reducing the area cost. Both the mutation and crossover operations are performed concurrently for accelerating the MA. In addition, a pipeline architecture with direct memory access (DMA) operation is adopted for the C-Means operation. A hardware sorting structure is adopted for survival test. The proposed architecture has been implemented on field programmable gate array (FPGA) devices (Hauck & Dehon, 2008) so that it can operate in conjunction with a softcore CPU (NIOS II Processor Reference Handbook, 2008). Using the reconfigurable hardware, we are then able to construct a system on programmable chip (SOPC) system for the genetic VQ design. As compared with its software counterparts, numerical results reveal that the proposed FPGA-based MA architecture attains higher performance with significantly lower training time for VQ design. These fact demonstrate the effectiveness of our design.

2. Preliminaries

The goal of a VQ for data clustering is to partition a large data set $\mathbf{X} = \{\mathbf{x}_1,...,\mathbf{x}_t\}$ into N non-overlapping clusters $C_1,...,C_N$, where $N << t$. The partitioning process is based on a set of codewords $\{\mathbf{y}_1,...,\mathbf{y}_N\}$, where the codewords and the vectors in $\mathbf{X}$ are of the same dimension w. Given a vector $\mathbf{x} \in \mathbf{X}$, the $\mathbf{x}$ will be assigned to the cluster C_i when

$$i = \alpha(\mathbf{x}) = \arg \min_{1 \le j \le N} d(\mathbf{x}, \mathbf{y}_j), \tag{1}$$

where $d(\mathbf{u}, \mathbf{v})$ denotes a distance measure between two vectors $\mathbf{u}$ and $\mathbf{v}$. In this paper, the squared distance is adopted as the distance measure. When applied for data reduction applications such as data compression, a vector $\mathbf{x}$ will be represented by the codeword $\mathbf{y}_i$ when $i = a(\mathbf{x})$. One cost function for the data reduction is the average distortion for representing $\mathbf{x}$ by $\mathbf{y}_i$, as shown below

$$D = \frac{1}{wt} \sum_{i=1}^{t} d\left(\mathbf{x}_i, \mathbf{y}_{\alpha(\mathbf{x}_i)}\right) \tag{2}$$

Given a data set $\mathbf{x}$, the objective of the VQ design is to find a set of codewords $\{\mathbf{y}_1,...,\mathbf{y}_N\}$ minimizing D in eq.(2).

In the basic MA (termed MA (I)) for VQ design, there are P genetic strings for the genetic operations. Each string r represents a set of N codewords $\{\mathbf{y}_1,...,\mathbf{y}_N\}_r$. Note that these strings are strings of vectors, not strings of binary numbers.

Let $S(k)$ and $D(k)$ denote the set of P strings and the value of current minimum distortion D after the execution of the k-th generation of the basic MA, respectively. Let $s*$ be the current optimum string during the course of genetic operations. Suppose that the $(k-1)$-th iteration is completed, and the execution of the k-th ($k \geq 1$) is to be done. We then perform the following operations sequentially on the strings in $S(k-1)$.

Regeneration: Since each string in $S(k-1)$ for the genetic operations is in fact a codebook of VQ, its corresponding D can be computed using eq.(2). The inverse of D is used as fitness function for each string. The regeneration process is then conducted using the roulette-wheel technique. Once a string has been selected for reproduction, an exact replica of it is made as a regeneration string. In the algorithm, P regeneration strings are created after the regeneration operation.

Crossover: On each regeneration string r, $\{\mathbf{y}_1,\mathbf{y}_2,...,\mathbf{y}_N\}_r$, one point crossover is applied with probability P_c. Out of the total population, a partner string r', $\{\mathbf{z}_1,\mathbf{z}_2,...,\mathbf{z}_N\}_{r'}$, is randomly chosen. Then an integer random number n between 1 and N is generated. Both strings are cut into two portions at position n and the portions $\{\mathbf{y}_{n+1},...,\mathbf{y}_N\}$ and $\{\mathbf{z}_{n+1},...,\mathbf{z}_N\}_r$ are mutually exchanged.

Mutation: Mutation is performed on each codeword of each string with a small probability P_m. Suppose now the string $r=\{\mathbf{y}_1,\mathbf{y}_2,...,\mathbf{y}_N\}_r$ is to be mutated. One of the N codewords, $\mathbf{y}$, is chosen at random. Among the w numbers in $\mathbf{y}$, we also select one number at random. Then a random number, taking the binary values b or $-b$, is generated, and is added to the chosen component.

Local Refinement: The C-Means algorithm is used for the local refinement in the MA. Each string r in the population after mutation is used as the initial codebook to C-Means algorithm. The output codebook of C-Means algorithm then is result of local refinement of r. All the strings after local refinement then form the set $S(k)$. The average distortion of each string in $S(k)$ is then computed for updating $D(k)$.

In the MA algorithm, the iteration continues until the convergence of the sequence $D(k)$. The current optimum string $s*$ after the completion of MA algorithm is then chosen as the desired result.

It may be difficult to implement the MA (I) algorithm in hardware. This is because two population memories are required in the algorithm. One population memory contains the parent strings. The other is used for storing the strings after the regeneration process. The roulette-wheel technique may also become the bottleneck for the hardware. In addition, the crossover, mutation and C-Means operations should operate over all the P regenerated strings. The corresponding hardware complexity therefore may be very high.

3. The proposed MA architecture

In this chapter, two alternative MA algorithm (termed MA (II) and MA (III) algorithms) is adopted for the VQ design. The MA (II) is based on the steady-state GA for global search, which has superior performance over the basic generational GA for a number of applications (Rasheed & Davisson, 1999). There is no concept of generation in steady-state

GA. Let S be the population of P genetic strings, which are called the parent strings. Initially, the P strings in S are randomly generated. Two strings (denoted by r_1 and r_2) in S will be selected for mutation and crossover for creating a new child string (denoted by c). The new string then is used as the initial codebook to the C-Means algorithm for local refinement.

The fitness value of the child string after local refinement is then evaluated and compared with the fitness value of all the parent strings in S. If the new string is inferior to all the parent strings in S, no parent string will be removed. Otherwise, the parent string with lowest fitness value is replaced by the child string.

Note that because each string for the VQ design is actually a codebook, the memory access time for string retrieval may be long. Consequently, the retrieval process for r_1 and r_2 may be time-consuming. To reduce the memory access time, in the algorithm, the previous r_1 becomes the new r_2 and then the new r_1 is chosen randomly from S. This selection scheme reduces the memory access time by half.

As the process of selection, crossover, mutation, local refinement, and survival/replacement continues, the overall fitness of population will increase and the survival rate of new offspring will diminish. At some point, the offspring survival rate will drop to zero. At this point, evolution has probably ceased and the algorithm may be terminated. The MA (II) algorithm is more effective for the hardware design. Only one population memory is required. In addition, crossover and mutation operations only operate on r_1 and r_2 instead of all strings in the population memory. Finally, C-Means operation only operate on the new child string c. Therefore, only one crossover and mutation module, and one C-Means module are necessary for hardware implementation. These facts effectively reduce the area cost for FPGA design.

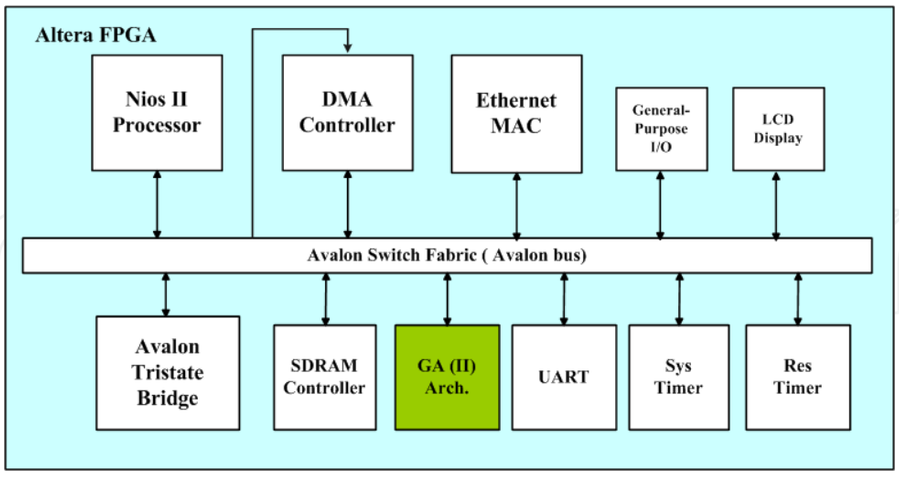


Fig. 1. The employment of SOPC for the MA (II) architecture

In the MA (III) algorithm, there are M populations. Each population evolves independently using the MA (II) algorithm. After all the populations are converged, the optimal strings from different populations are compared. The string having the highest fitness value is then the selected string for the VQ design.

The hardware architecture of the MA (II) and MA (III) can be viewed as a user logic in the NIOS-based SOPC system (NIOS II Processor Reference Handbook, 2008), as shown in Figure 1. Because the proposed architecture is used for the VQ design, the training data is required. The goal of using the SOPC is to provide the training data for the hardware architecture. The training data can be stored in a SDRAM, and delivered to our architecture via the Avalon bus. The DMA can be used to speed up the delivery. Alternatively, the training data can be obtained from a remote host via the internet.

Figure 2 shows the hardware architecture of the MA (II) algorithm. It contains population memory, crossover & mutation unit, C-Means unit, survival test & update unit, and Avalon bus interface. Both the population memory and crossover & mutation unit contain random number generators (RNGs).

In this architecture, the population memory unit is devoted for storing the genetic strings. Moreover, the random selection of parent strings for subsequent crossover and mutation operations is also included here. This selection is based on the RNG inside the population memory unit. All the crossover and mutation operations are performed concurrently in the crossover & mutation unit for producing a new child string c. The fitness value of the resulting string is then evaluated by the fitness evaluation unit.

Based on the fitness value, the goal of the survival test & update unit is to determine whether the child string c will survive. If it is the case, the parent string in the population memory unit with the worst fitness value will be replaced by the child string. Each unit in Figure 2 will be described in detailed as shown below.

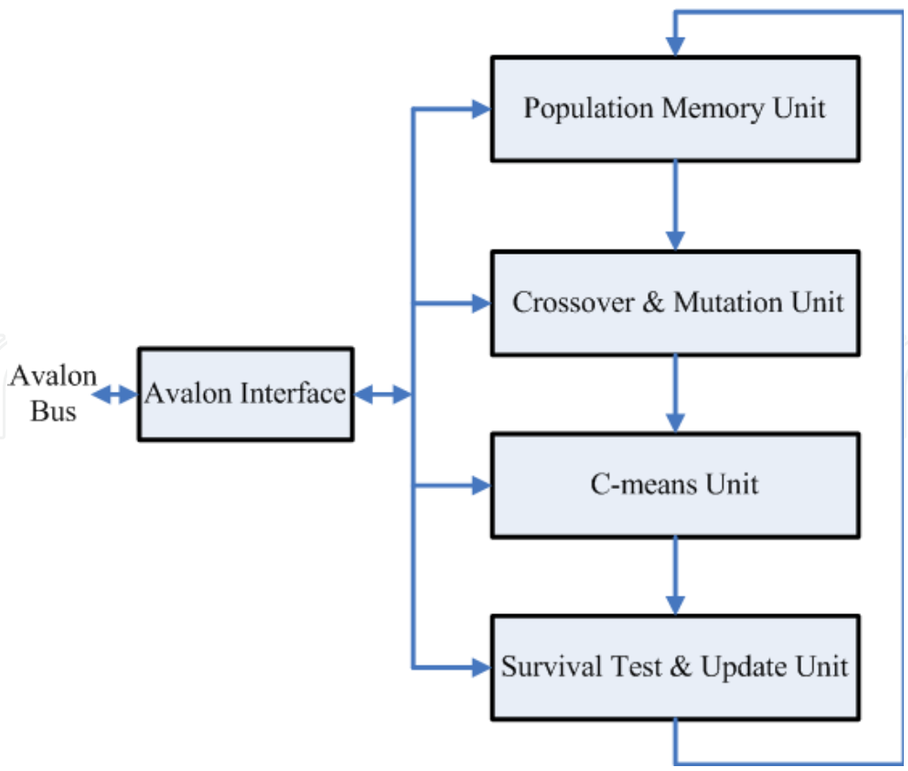


Fig. 2. The proposed hardware architecture for MA (II)

Population Memory Unit: The population memory contains a 2-port RAM and a RNG unit. The 2-port RAM contains S, the set of P genetic strings. In our design, the implementation of the RAM is based on the embedded memory, which is provided by some FPGA devices such as Altera Stratix II. The goal of RNG unit in the population memory unit is to select randomly a string for the subsequent crossover and mutation operations. In our design, the cellular automata (CA) is adopted for the VLSI implementation of random number generator due to its simplicity and regularity of the design.

Mutation and Crossover Unit: Figure 3 shows the basic structure of the mutation and crossover unit, which contains three shift registers for storing the strings r_1, r_2 and c, respectively. A number of RNGs, comparators, multiplexers and counters are then used for crossover and mutation. The major advantage of this architecture is that the crossover and mutation can be performed concurrently with low are a cost.

As shown in Figure 3, SHIFT REGISTER 1 and SHIFT REGISTER 2 contain strings r_1 and r_2, respectively. Note that the architecture does not randomly select new r_1 and r_2 from the population memory. In fact, only new r_1 is chosen from population memory. The new r_2 is actually the previous r_1. The memory access time and routing overhead can then be significantly reduced. Based on the algorithm, in the architecture, The SHIFT REGISTER 1 obtains r_1 from the population memory unit. The SHIFT REGISTER 2 obtains r_2 from SHIFT REGISTER 1.

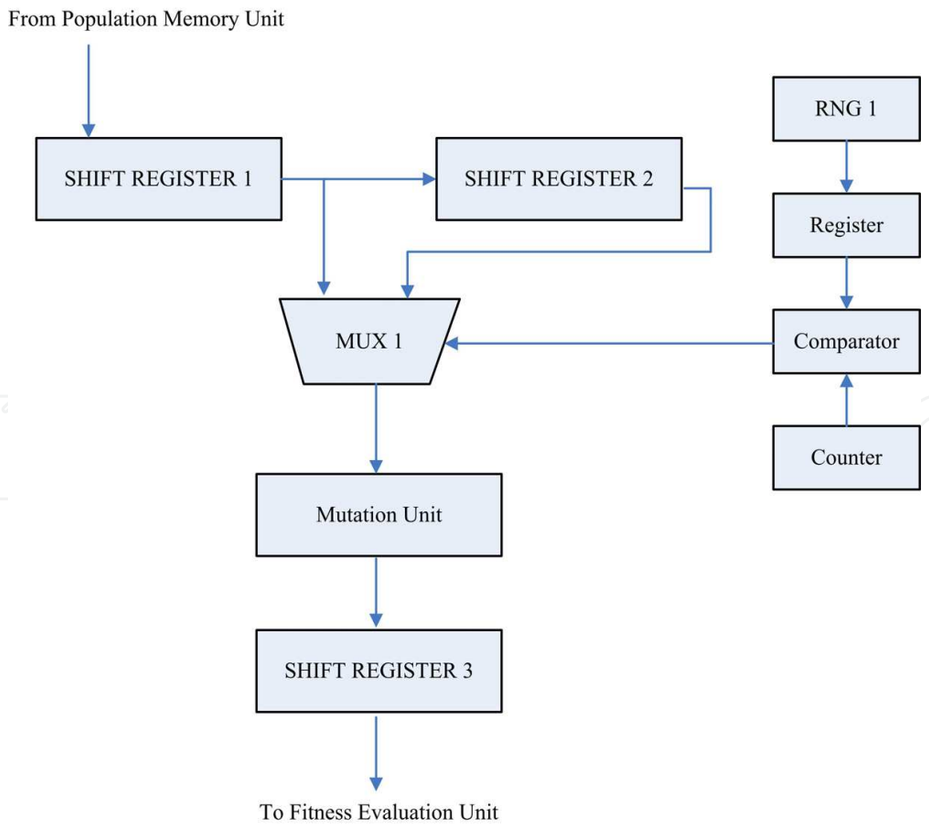


Fig. 3. The architecture of crossover and mutation unit

The crossover operations are accomplished by concurrently shifting the strings in SHIFT REGISTER 1 and SHIFT REGISTER 2 to MUX 1. Each shift register will shift one codeword at a time. As shown in Figure 3, MUX 1 is a switch selecting the codewords of either r_1 or r_2, and route them to SHIFT REGISTER 3, which contains the resulting child string c. The control line of MUX 1 is connected to a comparator, which compares the value of RNG 1 to that of a counter. The counter records the number of shifts made by the shift registers. The value of RNG 1 serves as a threshold here. When the counter value is less than the threshold, codewords of SHIFT REGISTER 1 (i.e., r_1) goes to SHIFT REGISTER 3. Otherwise, codewords of will be selected. Consequently, the value of RNG 1 determines the crossover point. The value will be randomly generated prior to the shifting operations.

We also observe from Figure 3 that the output codeword of MUX 1 will pass through the mutation unit before arriving the SHIFT REGISTER 3. Figure 4 shows the architecture of the mutation unit. As shown in the figure, all w components of the output codeword mutate concurrently. The mutation circuit for each component i consists of 2 RNGs (termed RNG ia and RNG ib), one register (termed register i), one comparator (termed comparator i), one multiplexer (termed mux i).

The probability for mutation P_b is stored in a separate register, and is broadcasted to all the mutation circuits. In the mutation circuit for each component i, the value of RNG ia is first compared with the P_b. The component i will be mutated when the value of RNG ia is less than P_b. The mutated value is then determined by RNG ib.

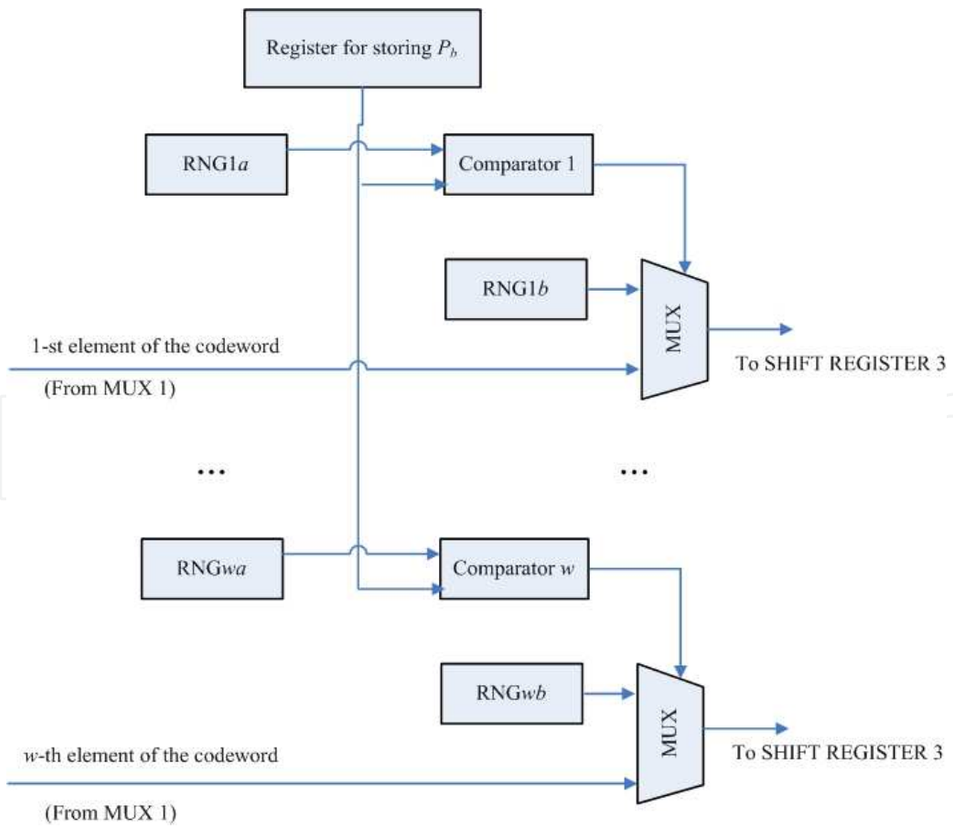


Fig. 4. The architecture of mutation unit

C-Means Unit: The goal of the C-Means unit is to locally refine the mutated child string stored in SHIFT REGISTER 3 using the C-Means algorithm. As shown in Figure 5, the proposed C-Means architecture can be decomposed into two units: the partitioning unit and the centroid computation unit. These two units will operate concurrently for the local refinement process. The partitioning unit uses the codewords stored in the register to partition the training vectors into N clusters. The centroid computation unit concurrently updates the centroid of clusters. Note that, both the partitioning process and centroid computation process should operate iteratively in software. However, by adopting a novel pipeline architecture, our hardware design allows these two processes operate in parallel for reducing the computational time. In fact, our design allows the concurrent computation of N +2 training vectors for the C-Means operations.

Figure 6 shows the architecture of the partitioning unit, which is a N -stage pipeline, where N is the number of codewords (i.e., clusters). The pipeline fetch one training vector per clock from the input port. The i -th stage of the pipeline compute the squared distance between the training vector at that stage and the i -th codeword of the codebook. The squared distance is then compared with the current minimum distance up to the i -th stage. If distance is smaller than the current minimum, then the i -th codeword becomes the new current optimal codeword, and the corresponding distance becomes the new current minimum distance. After the computation at the N -th stage is completed, the current optimal codeword and current minimum distance are the actual optimal codeword and the actual minimum distance, respectively. The index of the actual optimal codeword and its distance will be delivered to the centroid computation unit for computing the centroid and overall distortion.

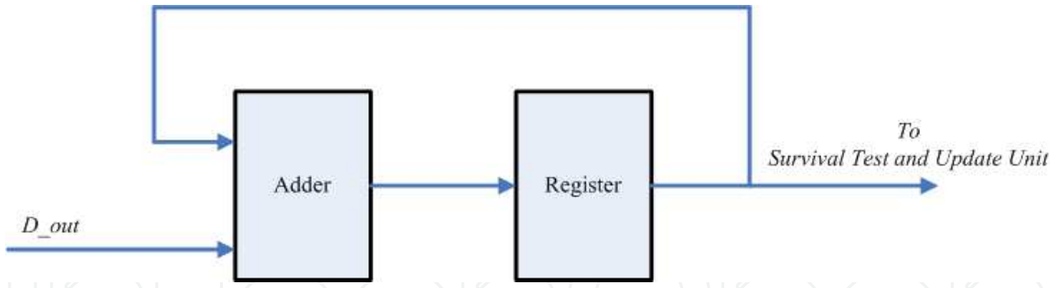


Fig. 5. The architecture of the C-Means unit

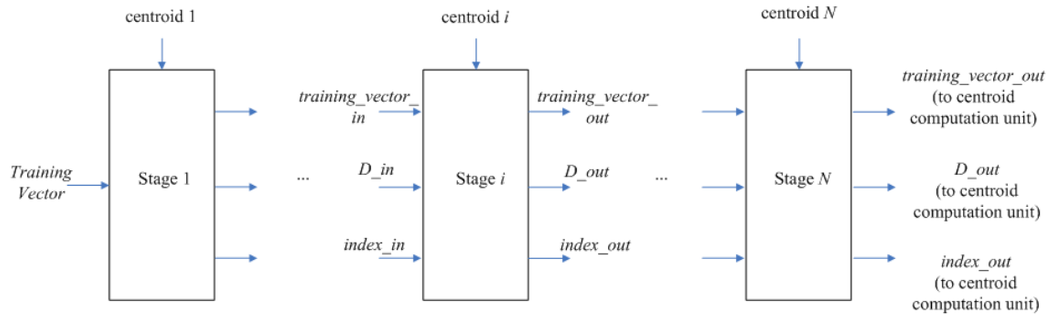


Fig. 6. The architecture of the partitioning unit

Figure 7 depicts the architecture of the centroid computation unit, which can be viewed as a two-stage pipeline. In this paper, we call these two stages, the accumulation stage and division stage, respectively. Therefore, there are $N+2$ pipeline stages in the C-Means unit. The concurrent computation of N +2 training vectors therefore is allowed for the local refinement operations.

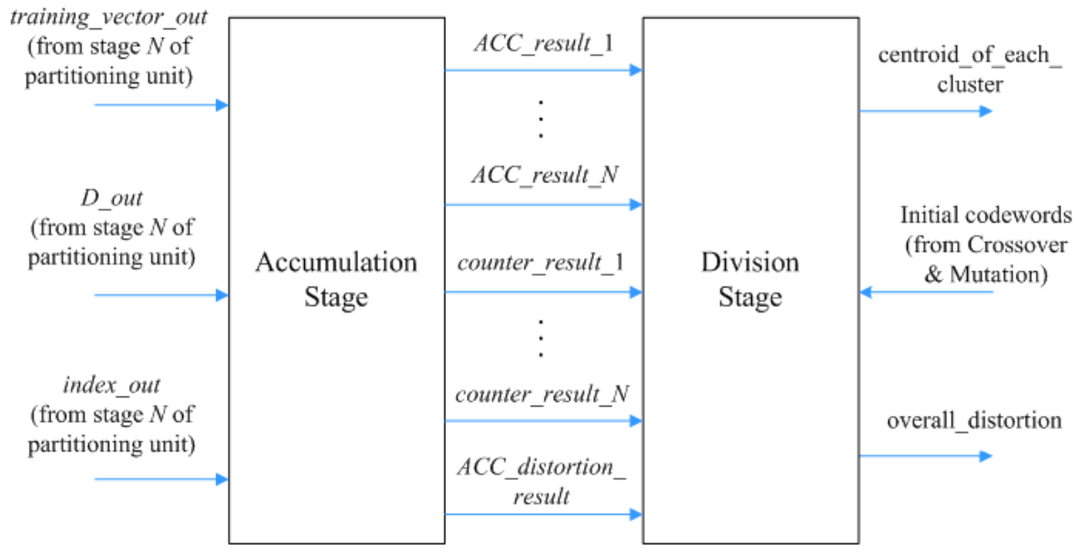


Fig. 7. The architecture of the centroid computation unit

As shown in Figure 8, there are N accumulators (denoted by $ACCi, i = 1,...,N$) and N counters for the centroid computation in the accumulation stage. The i-th accumulator records the current sum of the training vectors assigned to cluster i. The i-th counter contains the current number of training vectors mapped to cluster i. The i-th counter contains the current number of training vectors mapped to cluster i. The training_vector_out, D_out and index_out in Figure 8 are actually the outputs of the N-th pipeline stage of the partitioning unit. The index_out is used as control line for assigning the training vector (i.e. training_vector_out) to the optimal cluster found by the partitioning unit.

The circuit of division stage is shown in Figure 9. There is only one divider in the unit because only one centroid computation is necessary at a time. Suppose the final index_out is i for the i-th vector in the training set. The centroid of the i-th cluster then need to be updated. The divider and the i-th accumulator and counter are responsible for the computation of the centroid of the i-th cluster. Upon the completion of the j-th training vector at the centroid computation unit, the i-th counter records the number of training vectors (up to j-th vector in the training set) which are assigned to the i-th cluster. The i-th accumulator contains the sum of these training vectors in the i-th cluster. The output of the divider is then the mean value of the training vectors in the i-th cluster.

It can be observed from the Figure 9 that the division stage also evaluates the overall distortion of the codebook. This can be accomplished by simply accumulates all the minimum distortion associated with each training vector after the completion of the

partitioning process. The overall distortion is used for both the fitness evaluation and the convergence test of the C-Means algorithm.

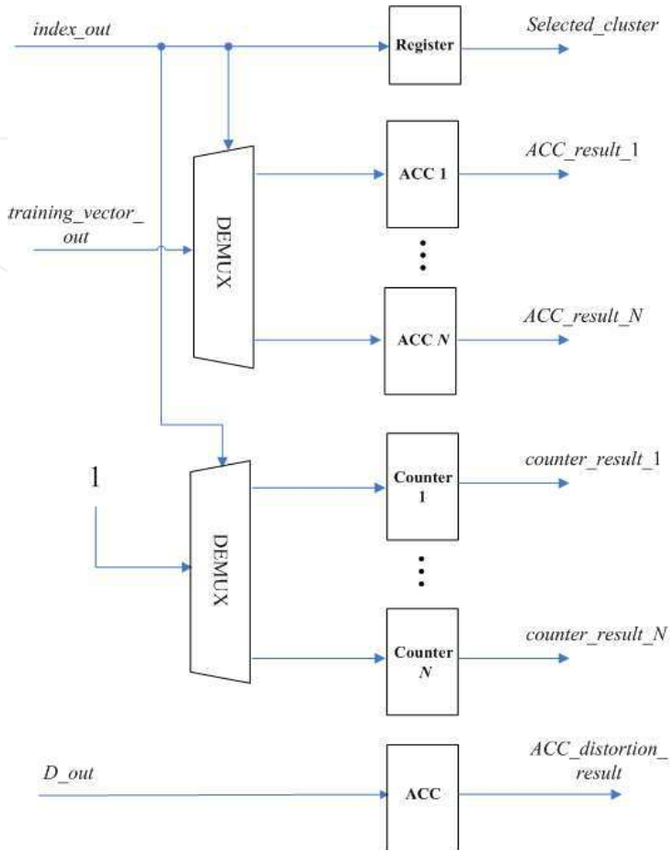


Fig. 8. The architecture of the accumulation stage in the centroid computation unit

Survival Test and Update Unit: This unit contains a hardware sorting circuit (Hwang, Li, Yeh & Chan, 2008), which sorts the parent strings in a descending order according to their fitness values. After the fitness evaluation operation is completed, the fitness value of the child string is used as the input to the sorting circuit. When the distortion of the string is larger than the parent string with lowest fitness value, the child string is not survival, and no updating operation is necessary. Otherwise, the parent string with highest distortion is replaced by the child string. The sorting circuit is then activated to determine the new parent string with the highest distortion.

Figure 10 depicts the architecture of MA (III) algorithm, which contains modules. Each module is a hardware realization of MA (II) algorithm. Therefore, the architecture of each module is shown in Figure 2. Although the genetic strings in different modules (i.e., different populations) evolve independently, they all need the same set of training vectors for C-means algorithm and fitness evaluation. Independent requests for training vector delivery from different modules demand very high memory bandwidth. This may become the bottleneck of the architecture for MA (III) implementation. To solve the problem, the C-means operation of all the architectures operate synchronously. Therefore, training vectors from main memory can be broadcasted to all the modules for C-means training. In addition, the DMA is used for

further accelerating the data delivery. To implement synchronous C-means operations among different modules, we first note that all the string selection, mutation crossover, and survival test operations take fixed number of clock cycles. The same operation will take the same number of clocks in different modules. Consequently, when all the modules start MA operations at the same time, the synchronization among the modules can be achieved.

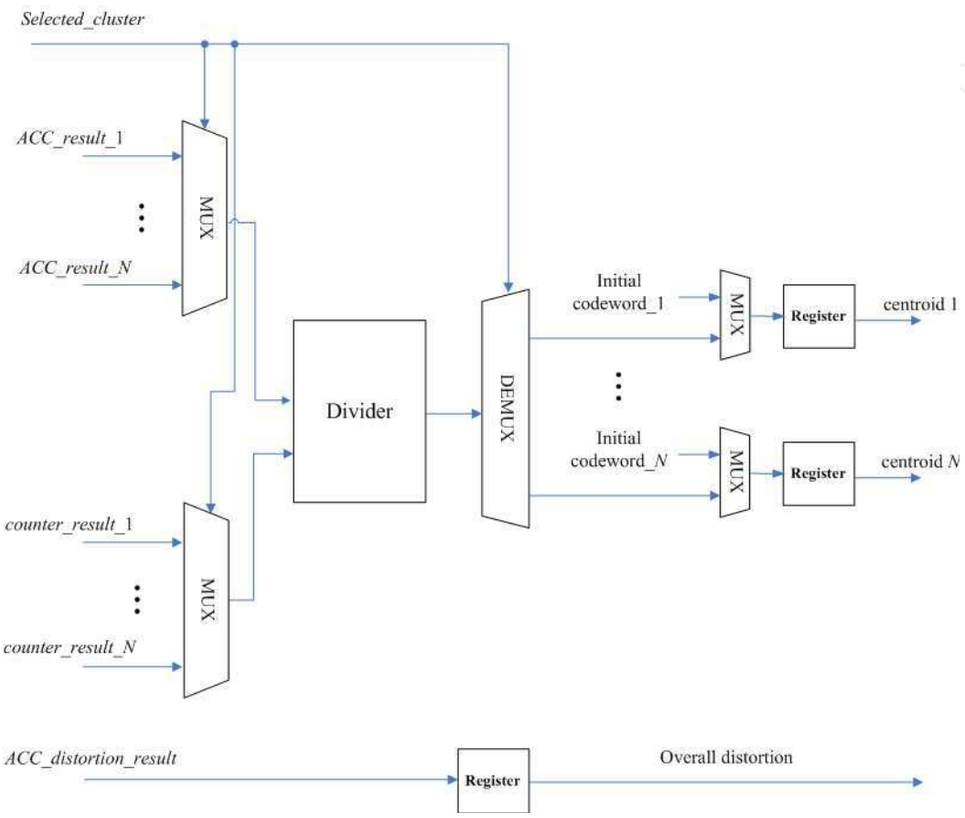


Fig. 9. The architecture of the division stage in the centroid computation unit

3. Experimental results

This section presents some physical performance measurements of the proposed FPGA implementation. The target FPGA device for the hardware design is Altera Stratix II 2S60 (Stratix II Device Handbook, 2008). The Altera Quartus II version 7.2 with SOPC Builder is used as the platform for the system development. The vector dimension of codewords is $w = 2 \times 2$. The mutation probability is $P_b = 0.03125$.

Figure 11 compares the average distortion of the proposed MA (II) implementation with that of basic MA(I) under the same population size $P = 64$ and number of codewords $N = 64$. The software implementation of MA (I) is executed on the 3-GHz Pentium D CPU. In the experiment, we execute each implementation 300 times independently. The training set contains 65536 training vectors from the image "Lena."

Based on the training set, the average distortion of each execution is computed according to eq.(2). Figure 11 then reveals the distribution of the average distortion. It can be observed

from Figure 11 that both implementations have similar average distortion distributions. Therefore, our architecture simplifying the string random selection process does not degrade the performance of memetic VQ design.

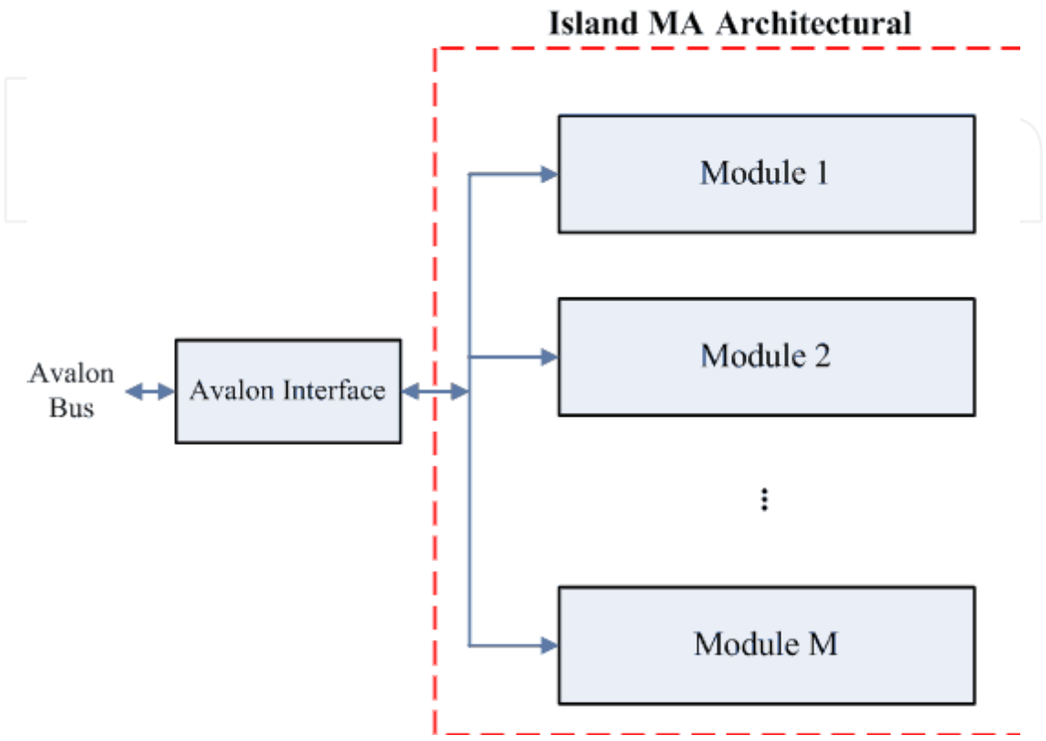
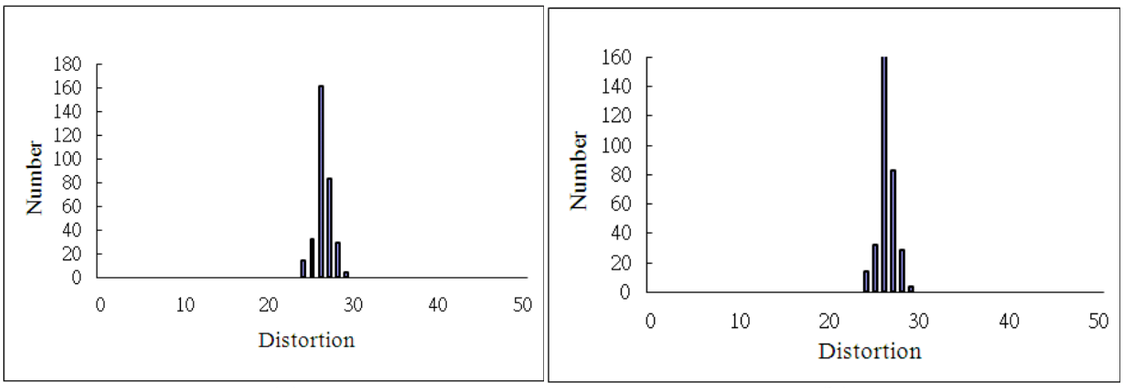


Fig. 10. The architecture of the MA (III) Algorithm.



(a) (b)

Fig. 11. Distribution of average distortion of various MA algorithms:(a) MA(I) algorithm,(b) MA(II) algorithm

Figure 12 shows the distribution of the mean-squared distortion of MA(II) algorithm implemented by our proposed architecture for 100 independent runs with $N = 64$ and $P = 64$. Each run starts with different set of genetic strings randomly selected from training images. The distribution of distortion of C-Means algorithm and steady-state GA algorithm

for 100 independent runs are also included in Figure 12 for comparison purpose. From Figure 12, it can be observed that the C-Means algorithm has a broad distribution of local optima. On Figure 12: Distortion Distribution of proposed MA(II) architecture, C-Means architecture and steady-state GA architecture.

On the other hand, from Figure 12, we see that the distribution of distortion of the MA(II) has a better concentration. The worst case of MA(II) has distortion Only 9 of the distortion of VQs designed by C-Means algorithm are lower than that of the worst case of the MA(II) algorithm. The best case of the MA(II) has The difference between the worst and best cases is only 2. Moreover, we can observe from Figure 12 that the MA(II) algorithm significantly outperforms the steady-state GA algorithm for 100 independent runs.

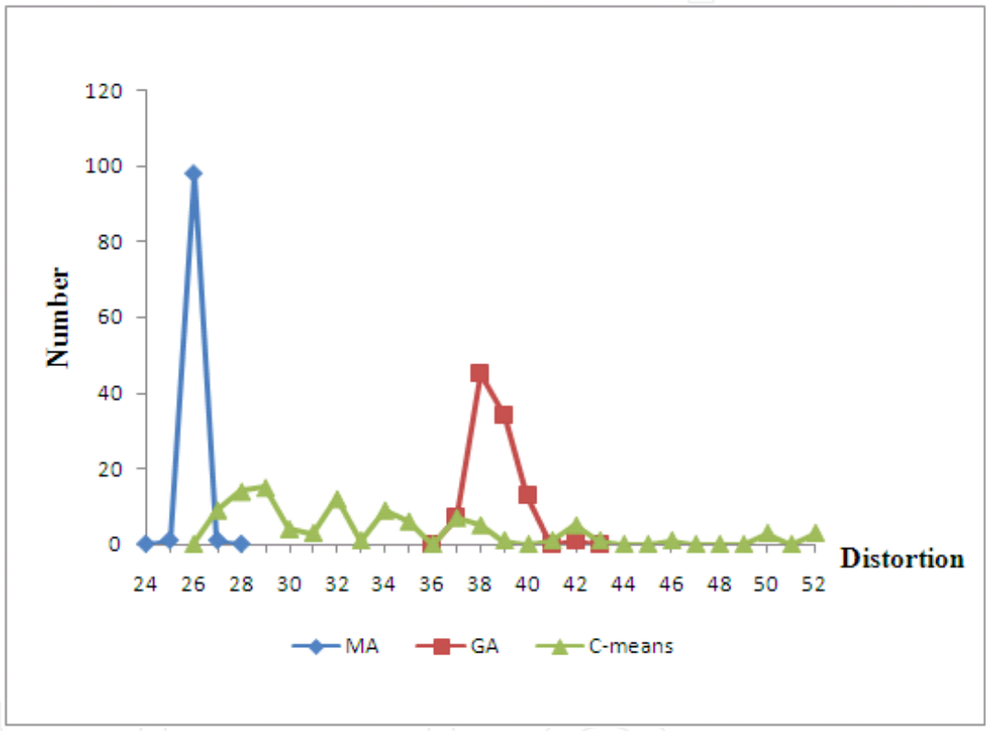


Fig. 12. Distortion Distribution of proposed MA(II) architecture, C-Means architecture and steady-state GA architecture.

Table 1 shows the area costs, average distortion and execution time of the proposed SOPC system for various population size We fix for the experiment. The distortion and execution time are obtained by averaging those of 100 independent executions. It can be observed from Table 1 that the area cost of the entire system becomes only slightly higher as increases. The average execution time grows linearly with In addition, the average distortion can be effectively reduced as becomes larger.

Table 1 shows the area costs, average distortion and execution time of the proposed SOPC system for various population size P. We fix $N = 64$ for the experiment. The distortion and execution time are obtained by averaging those of 100 independent executions. It can be observed from Table 1 that the area cost of the entire system becomes only slightly higher as

P increases. The average execution time grows linearly with P. In addition, the average distortion can be effectively reduced as P becomes larger.

Table 2 investigates the impact of the number of codewords N on the area cost, average distortion and execution time of the proposed SOPC system for MA (II) implementation. The population size P is fixed to 64 for this experiment. The distortion and execution time are obtained by averaging those of 100 independent executions. From Table 2, we see that the number of ALMs and embedded memory bits consumed by the circuit grows with N. The average distortion is effectively lowered as N increases.

P	Average Distortion	Average CPU Time	ALMs (Entire SOPC)	Embedded Memory Bits(Entire SOPC)
16	26.88	366.8ms	22,487	629,504
32	26.65	580.5ms	22,793	662,272
64	26.50	1001.0ms	23,456	727,808
128	26.48	1801.4ms	24,935	858,880

Table 1. The area costs, average distortion and execution time of the proposed SOPC system for MA (II) algorithm for various population size P

N	Average Distortion	Average CPU Time	ALMs (Entire SOPC)	Embedded Memory Bits (Entire SOPC)
8	85.15	909.8ms	9,068	613,120
16	54.49	933.7ms	10,444	629,504
32	37.12	971.0ms	14,569	662,272
64	26.50	1001.0ms	23,456	727,808

Table 2. The area costs, average distortion and execution time of the proposed SOPC system for MA (II) for various number of codewords N

We can also observe that the average CPU time only slightly increase with N. Note that the computational complexity of C-Means algorithm may be large when the number of codewords is high, due to the fact that the partitioning process is based on exhaustive search. However, in the proposed architecture, the C-Means algorithm is implemented by

an efficient $(N+2)$-stage pipeline. Therefore, the average CPU time for the VQ design can still be low even when N increases, as shown in Table 2.

Table 3 shows the average CPU time and average distortion of various SOPC systems for VQ design. The measurements are based on the average values of 100 independent runs. All the SOPC systems are running on the same NIOS II softcore CPU with 50 MHz operating frequency. The number of codewords is $N = 64$. Moreover, the MA(II) and steady-state GA have the same population size $P = 64$.

For comparison purpose, we also implement the software counterpart of each SOPC system using only C code running on Pentium D 3.0 GHz CPU. Note that, the SOPC system and its software counterpart may not have the same distortion for each algorithm shown in the table. This is because the SOPC system and its software counterpart are based on different RNG for initial codewords selection and genetic operations. Nevertheless, the difference in distortion is very small.

	SOPC systems			Software		
	MA (II)	C-Means (Hwang, Hsu, Li, Weng & Yu, 2010)	Steady-State GA(Ou, 2010a)	MA(II)	C-Means	Steady-State GA(Ou, 2010b)
CPU Time (*m*s)	1001.0	7.36	400.2	366669.3	1922.35	37495.47
Distortion	26.50	35.10	37.62	27.67	37.40	37.48

Table 3. The average CPU time and average distortion of various SOPC systems and software programs for VQ design

It can be observed from the table that each SOPC system has significantly lower CPU time than its software counterpart. In particular, the CPU time of the SOPC system and software for MA(II) design are 1001.0 ms and 366669.3 ms, respectively. The speedup of the proposed SOPC over its software counterpart is 366.3. In addition, as shown in Table 3, the MA(II) algorithm has lowest average distortion as compared with the C-Means and steady-state GA algorithms.

The performance of MA(II) algorithm can be further improved by the employment of MA(III) algorithm. Table 4 compares the performance of the proposed MA(II) and MA(III) architectures, and their software counterparts. The number of modules for MA(III) implementation is $M = 3$. The CPU time and average distortion measurements are based on the average values of 100 independent runs. The area cost of the hardware implementations are also included in the table for comparison purpose. To achieve meaningful comparisons, both the architectures have the same number of codewords $N = 16$ and the same number of total genetic strings $P = 24$. They are also implemented on the same target FPGA device Altera Stratix II 2S60. The software counterpart executes on the processor 4GHz Intel I7.

It can be observed from Table 4 that the MA(III) architecture has lowest average CPU time and lowest average distortion. The average CPU time of the MA(III) architecture is 0.47 second, which is only 43.93 % and 1.36 % of the CPU time of the MA(II) architecture and the software counterpart of MA(III), respectively. Note that, each module for MA(III) architecture has only 8 genetic strings. By contrast, the MA(II) architecture has 24 genetic strings. Therefore, because each module for MA(III) architecture has smaller population, its memetic operations are able to achieve faster convergence. In addition, the best string is selected from multiple populations. Its average distortion is lower than that of the basic memetic algorithm containing only one population, as shown in Table 4.

Although each module has smaller population for MA(III) architecture, the total number ofgenetic strings of the MA(III) architecture is identical to that of the MA(II) architecture. As a result, we can see from Table 4 that both the hardware architectures consume the same number of embedded memory bits. On the other hand, the MA(III) architecture uses more ALMs and DSP blocks for hardware implementation. This is because the architecture consists of 3 modules, and each module has independent mutation and crossover unit, and C-means units. These units utilizes large number of ALMs and DSP blocks. Therefore, when both the area cost and speed are the important concerns, the MA (II) architecture can be used. All these facts demonstrate the effectiveness of the proposed architectures.

Algorithms	ALMs	Embedded Memory Bits	DSP Blocks	Average CPU Time	Average Distortion
MA (III) Architecture	19281	12288	288	0.47 (sec)	54.62
MA (II) Architecture	7203	12288	96	1.07 (sec)	55.27
Software				34.55(sec)	55.23

Table 4. Comparisons of the performance of various MA implementations

4. Conclusion

The proposed MA (II) and MA (III) architectures have been shown to be effective for fast VQ training. Selections of genetic strings one at a time for crossover and mutation are able to reduce area cost for hardware implementation while maintaining the performance for VQ training. Moreover, the DMA for training vector delivery and pipeline architecture for C-Means algorithm are beneficial for local refinement and fitness evaluation. Experimental results show that the proposed architectures attains high speedup over its software counterpart. It also has lower average distortion as compared with C-means and steady-state GA algorithms.

5. References

Areibi, S., Moussa, M., & Abdullah, H., (2001). A Comparison of Genetic/Memetic Algorithms and Heuristic Searching, *Proc. International Conference on Artificial Intelligence,* pp.660-666, June 2001.

Choi, Y. H., & Chung, D.J., (2000) VLSI Processor of Parallel Genetic Algorithm, *IEEE Asia Pacific Conf. on ASICs*, pp.143-146, 2000.

Eiben, A. E., & Smith, J. D., (2003) Introduction to Evolutionary Computing, *Springer*, 2003.

Gersho, A., and Gray, R.M., (1992). Vector Quantization and Signal Compression, Kluwer, *Norwood, Massachusetts*, 1992.

Hwang W.J., Hsu C.C., Li H.Y., Weng S.K., & Yu T.Y., (2010). High Speed C-means Clustering in Reconfigurable Hardware, *Microprocessors and Microsystems*, pp.237-246, 2010. (SCI)

Hwang, W.J., & Hong, S. L., (1999). Genetic entropy-constrained vector quantization, *Optical Engineering*, Vol. 38, pp.233-239, 1999.

Hwang, W.J., Li, H.Y., Yeh, Y.J. & Chan, K.F., (2008). FPGA Implementation of Competitive Learning with Partial Distance Search in the Wavelet Domain, *Progress in Neurocomputing Research*, pp.203-221, NOVA Science Publisher, 2008.

Hauck, S., & Dehon, A., (2008). Reconfigurable Computing, *Morgan Kaufmann*, 2008.

Merz, P., & Freisleben, B., (1999). A Comparison of Memetic Algorithms, Tabu Search, and Ant Colonies for the Quadratic Assignment Problem, *Proc. the IEEE Congress on Evolutionary Computation*, pp.2063-2070, 1999.

Molina, D.; Lozano, M., & Herrera, F., (2010). MA-SW-Chains: Memetic algorithm based on local search chains for large scale continuous global optimization, *the IEEE Congress on Evolutionary Computation*, pp.1-8, 2010.

Moscato, P., (1999). Memetic Algorithms: A Short Introduction," *New Ideas in Optimization*, pp.219-234, McGraw-Hill, 1999.

Nedjah, N., & Mourelle, L., (2005). Hardware Architecture for Genetic Algorithms, *Lecture Notes in Computer Science*, pp. 554-556, Vol. 3533, 2005.

Ou, C.M., (2010a). FPGA implementation of genetic vector quantizers, *Neurocomputing*, pp. 2125-2131, vol. 73, no. 10-12, Jun. 2010.

Ou, C.M., (2010b). Vector Quantization Based on Steady-State Memetic Algorithm, *Journal of Marine Science and Technology (JMST)*, vol. 18, no. 4, pp. 553-557, Aug. 2010.

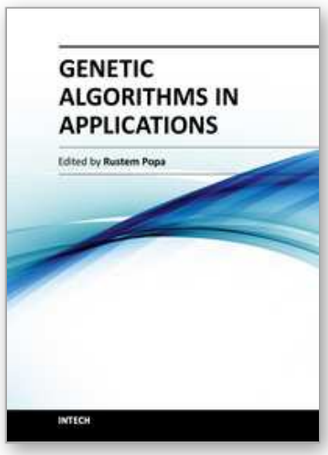
Rasheed, K., & Davisson, B.D., (1999). Effect of global parallelism on the behave of a steady state genetic algorithm for design optimization, In Proceedings of the Congress on Evolutionary Computation, Washington, DC, 1999.

Tommiska, M., & Vuori, J., (1996). Implementation of genetic algorithms with programmable logic devices, *Proc. 2nd Nordic Workshop on Genetic Algorithms and Their Applications*, pp.111-126, 1996.

Santos, P.V.; Alves, J.C., (2010). FPGA Based Engines for Genetic and Memetic Algorithms, *Field Programmable Logic and Applications (FPL)*, pp.251-254, 2010.

Stratix II Device Handbook, 2008, Altera Corporation. http:// www.altera.com/ literature/ lit-nio2.jsp.

NIOS II Processor Reference Handbook, 2008, Altera Corporation. http://www.altera.com/ literature/ lit-nio2.jsp.



Genetic Algorithms in Applications

Edited by Dr. Rustem Popa

ISBN 978-953-51-0400-1
Hard cover, 328 pages
Publisher InTech
Published online 21, March, 2012
Published in print edition March, 2012

Genetic Algorithms (GAs) are one of several techniques in the family of Evolutionary Algorithms - algorithms that search for solutions to optimization problems by "evolving" better and better solutions. Genetic Algorithms have been applied in science, engineering, business and social sciences. This book consists of 16 chapters organized into five sections. The first section deals with some applications in automatic control, the second section contains several applications in scheduling of resources, and the third section introduces some applications in electrical and electronics engineering. The next section illustrates some examples of character recognition and multi-criteria classification, and the last one deals with trading systems. These evolutionary techniques may be useful to engineers and scientists in various fields of specialization, who need some optimization techniques in their work and who may be using Genetic Algorithms in their applications for the first time. These applications may be useful to many other people who are getting familiar with the subject of Genetic Algorithms.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Chien-Min Ou and Wen-Jyi Hwang (2012). Efficient VLSI Architecture for Memetic Vector Quantizer Design, Genetic Algorithms in Applications, Dr. Rustem Popa (Ed.), ISBN: 978-953-51-0400-1, InTech, Available from: http://www.intechopen.com/books/genetic-algorithms-in-applications/efficient-vlsi-architecture-for-memetic-vector-quantizer-design

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen