1-20-2006

Hardware Implementation of a Novel Image Compression Algorithm

Vikas Kumar Reddy Sanikomm
University of New Orleans

HARDWARE IMPLEMENTATION OF A NOVEL IMAGE COMPRESSION ALGORITHM

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Engineering

by

Vikas Kumar Reddy Sanikommu

B.S.,Jawaharlal Nehru Technological University,Hyderabad,2002

December 2005

ACKNOWLEDGEMENTS

I would like to express my gratitude to my Advisor, Dr Dimitrios Charalampidis, for his support and timely advice through out my research. I appreciate and value his consistent feedback on my progress, which was always constructive and encouraging.

I would also like to express my sincere thanks to the other committee members, Dr. Vesselin Jilkov and Prof. Kim D Jovanovich for their willingness to be on my thesis committee and helping me in the dire moments of Katrina. Their invaluable suggestions and insightful comments have made my work more presentable. I would really like to appreciate them for allowing me get some of their valuable time in the time of Katrina for my defence.

Finally I express my gratitude to my friends for their encouragement and motivation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Image-related communications are forming an increasingly large part of modern communications, bringing the need for efficient and effective compression. Image compression is important for effective storage and transmission of images. Many techniques have been developed in the past, including transform coding, vector quantization and neural networks. In this thesis, a novel adaptive compression technique is introduced based on adaptive rather than fixed transforms for image compression. The proposed technique is similar to Neural Network (NN)–based image compression and its superiority over other techniques is presented

It is shown that the proposed algorithm results in higher image quality for a given compression ratio than existing Neural Network algorithms and that the training of this algorithm is significantly faster than the NN based algorithms. This is also compared to the JPEG in terms of Peak Signal to Noise Ratio (PSNR) for a given compression ratio and computational complexity. Advantages of this idea over JPEG are also presented in this thesis.

CHAPTER 1

INTRODUCTION

Digital images contain large amounts of data. Therefore, a high image quality implies that the associated file size is large. For the sake of storage and transmission over channels at high efficiency, a high quality, highly compressive algorithm for image compression is at demand. Despite the existence of image compression standards such as JPEG and JPEG 2000, image compression is still subject to a worldwide research effort. Image compression is used to minimize the amount of memory needed to represent an image. Images often require large number of bits to represent them, and if the image needs to be transmitted or stored, it is impractical to do so without somehow reducing the number of bits. The problem of transmitting or storing an image affects all of us daily. Namely, TV and fax machines are both examples of image transmission, and digital video players and web pictures are examples of image storage. By using data compression techniques, it is possible to remove some of the redundant information contained in the images, requiring less storage space and less time to transmit. Another issue in image compression and decompression is increasing processing speed, without losing the image quality especially in real-time applications.

Many compression techniques have been developed in the past, including transform coding, vector quantization, pixel coding and predictive coding. Most recently, Neural Network based techniques have being used for compression. Next, the compression types are discussed.

1.1 Lossless Compression

These techniques generally are composed of two relatively independent operations: (1) devising an alternative representation of the image in which its interpixel redundancies are reduced; and

(2) coding the representation to eliminate coding redundancies. Typical schemes are based on Huffman encoding. They normally provide compression ratios of 2 to 10 [6].

1.2 Lossy Compression

Unlike the Lossless Compression approaches, Lossy encoding is based on the concept of compromising the accuracy of the reconstructed image in exchange for increased compression. Many lossy encoding techniques are capable of reproducing recognizable images from data that have been compressed by more than 30:1, and images indistinguishable from the originals at 10:1 to 20:1 [6].

Transform-based coding techniques have proved to be the most effective in obtaining large compression ratios while retaining good visual quality. In low noise environments, where the bit error rate is less than 10^{-6}, the JPEG [3]-[4] picture compression algorithm, which employs cosine-transforms, has been found to obtain excellent results. However, an increased number of real-time applications require that compression be performed in highly noisy environments with high bit error transmission rates [12]. In this cases, JPEG and other compression techniques involving fixed transforms are not capable of maintaining high image quality.

Many recently developed techniques such as the wavelet Based, JPEG2000 technique, show some adaptability. With the involvement of Neural Networks in image compression, adaptive techniques have evolved [1]-[8]. Neural Networks have been proved to be useful in image compression because of their parallel structure, flexibility, robustness under noisy conditions and simple decoding [15]-[20]. However, there is a reduction in the quality of the decompressed image for the same compression efficiency.

Compression using Neural Networks suffers from several drawbacks, including:

(1) Slow Compression,

(2) High Computational complexity,

(3) Moderate Compression ratio and

(4) The reconstructed image quality is training dependent.

These techniques involve Single Structure and Parallel Structure Neural Network architectures. Parallel structures result in a higher decoded image quality than the single structured techniques.

This thesis introduces a different approach to adaptive image compression, which overcomes the above drawbacks. This architecture consists of a cascade of simple adaptive linear units. This technique is similar to the parallel NN technology in that image blocks can be encoded by different parts of the architecture. An image block is encoded using only a subset of stages and hence the total number of learning parameters is small with their estimation happens to be faster than that of NN techniques. This makes training a part of coding. The technique adapts to the content in order to calculate a set of transforms that code the image in a block manner similar to JPEG. The adaptive nature of this technique allows the extraction of the image information from the blocks through the calculation of their transforms and their associated coefficients.

CHAPTER 2

TECHNIQUES OF IMAGE COMPRESSION

As mentioned earlier, image compression techniques can be classified mainly as lossless and lossy.

2.1 Lossless Compression:

Lossless compression is the method in which the original data can be preserved after reconstruction without any loss, which means that the decompressed image looks nearly the same as the original image. These methods are preferred where high value content is required. Some of such applications include medical imaging or images made for archival consent. Some of the lossless coding techniques are:

- Run length coding
- Huffman coding
- Entropy coding

2.1.1 Run length Coding:

Run length encoding (RLE) is a sequential data compression technique used to reduce the size of a repeating string of characters. This repeating string is called a *run.* RLE encodes a run of symbols into two bytes, a count and a symbol. RLE compresses any type of data regardless of its content but the content of data to be compressed affects the compression ratio.

Consider a character run of 16 characters as:

'*aaacccccvvvtttt*'

Which normally would require 16 bytes to store but with RLE this would only take 8 bytes of data to store, the count is stored as the first bit and the symbol as the second bit which is: 3(a)6(c)3(v)4(t)

In this case RLE yields a compression ratio of 16:8 that is 2:1.

Images with repeating grey values along the rows (or columns) can be compressed by storing runs of identical grey values as:

Grey value 1	Repetition 1	Grey value 2	Repetition 2

Run length coding is fast and can easily be implemented but the compression is very limited as it requires a minimum of two characters worth of information to encode a run.

2.1.2 Huffman coding:

Huffman coding is the most popular technique for removing coding redundancy. This code provides a variable length code with minimal average code–word length (least possible redundancy). This yields the smallest possible number of code symbols per source symbol. The first step in this technique is to create a series of source reductions by sorting the message symbols under consideration in the increasing order and combining the lowest probability symbols into a single symbol until the entire message is finished.

The second step in this technique is to code each reduced symbol starting with the smallest and back to the original symbol. Example for this procedure is:

Original Source			Source Reduction			
Sym.	Prob.	Code	1	2	3	4
V2	0.4	1	0.4 1	0.4 1	0.4 1	0.6 0
V6	0.3	00	0.3 00	0.3 00	0.3 00	0.4 1
V1	0.1	011	0.1 011	0.2 010	0.3 01	
V4	0.1	0100	0.1 0100	0.1 011		
V3	0.06	01010	0.1 0101			
V5	0.04	01011				

2.1.3 Entropy coding:

The implementation of this method is a variable-to-fixed length code. This is an adaptive based technique. This technique uses the previous data to build a dictionary. This dictionary consists of all the strings in a window from the previously read data stream. The window is divided into two parts fixed-size window and look-ahead buffer. The fixed-size window consists of large blocks of decoded data and the look-ahead buffer consists of data read in from the data read but not yet encoded. The data in the buffer is then compared with the data in the fixed-size window. In the dictionary based method, the phrases are replaced with the tokens and if the number of bits in the token are less than the number of bits in the phrase compression occurs. Consider the following example:

Implementing in code composer studio	Code composer studio is

Fixed-size window of previously read data (32 bytes)

Look –ahead buffer (20 bytes)

In this example, if any matches are found then a token is sent to the output stream describing the match. When this symbol is matched, the data is shifted by an amount equal to the length of the symbol represented by the previous token. Data is pushed out of the token and new data is read into the buffer as shown:

Implementing in code composer studio	, is

Here the code composer studio is added to the dictionary and every time it is seen after this a token is passed in the phrase.

The three items taken care of in the token are:

- Length of an offset to a phrase in the fixed-size window.

- The length of the phrase
- The first symbol in the look-ahead buffer that follows the phrase
- The idea of this method is to build a dictionary of strings while encoding. The encoder and decoder start with an empty dictionary in this method. The dictionary is added with each character as long as it matches a phrase in the dictionary. The compression of this method is poor but, as the strings is re-seen and replaced in the dictionary the performance increases.

2.2 Lossy compression:

Lossy encoding is based on the concept of compromising the accuracy of the reconstructed image in exchange for increased compression. Lossy techniques provide far greater compression ratios than lossless techniques. In many applications, the exact restoration of the image is not always necessary. The restored image can be different to the original without the differences being distinguishable by the human eye. Some of the lossy coding methods are:

- Vector Quantization
- Transform Coding
- Fractal Compression

2.2.1 Vector Quantization:

Quantization is the procedure of approximating the continuous values with discrete values. The goal of quantization usually is to produce a more compact representation of the data while maintaining its usefulness for a certain purpose. There are two basic quantization types, namely scalar quantization and vector quantization. In scalar quantization, both the input to and output from the quantizer are scalars. In vector quantization, the vectors from a continuous input set are replaced by that of a much sparser set. This sparse set is called a codebook, and the vectors in it

are called codewords. Scalar quantization is not optimal as successive samples in an image are usually correlated or dependent, thus it has been observed that a better performance is achieved by coding vectors instead of scalars. Several vector quantization schemes exist to take advantage of different characteristics in the image.

In image compression using vector quantization, the procedure followed for codebook design is one of the essential parts of the technique, and directly affects the compressed image quality. First, a training image is split into blocks, and each block is represented in the form of a vector. The set of vectors obtained from the training image is called the set of training vectors. The codebook design procedure uses the training vectors in order to construct the codebook. Essentially, the codewords are a set of vectors that approximate the larger set of vectors obtained from the training image.

Let us assuming that training has already been performed, this implies that the codebook has been obtained. Then, a vector quantizer consists of the following:

(1) The encoder that compares the vectors **x** extracted from the image to be compressed, with the codebook vectors. If a vector **x** is found to be "closest" to one of the codewords, then the codeword's corresponding index is assigned to the vector **x**. The closeness of a vector to a codeword is defined by a distortion measure. One such commonly used distortion measure is the Euclidean distance.

(2) The decoder that maps an index back to the corresponding codeword in order to obtain an approximation of the original vector **x**.

The standard approach to construct the codebook is by Linde-Buzo-Gray's algorithm (LBG) [10].

2.2.2 Transform Coding:

In transform coding, a reversible linear transform is used to map the image into a set of transform coefficients, which are then quantized and coded. The decoder is just reverse of the encoder. A

typical transform coding system has four steps: (1) Decomposition of image into sub-images, (2) Forward transformation, (3) Quantization, (4) Coding.

An N x N image is subdivided into sub images of size n x n (mostly 8 x 8), which are then transformed; so as to collect as much information into a smaller number of transform coefficients. The quantization then quantizes the coefficients that contain the least information. The encoding process ends with the coding of the transformed coefficients. The most famous transform coding systems are Discrete Cosine Transform (DCT), and Karhunen-Loeve (KLT).

Discrete Transform Coding is preferred over all the transform coding techniques as it involves less number of calculations. The standard presently accepted, created by the Joint Photographic Experts Group (JPEG) for lossy compression of images uses DCT as the transformation. More about the DCT is discussed in the background chapter of this thesis.

2.2.3 Predictive Coding:

Predictive image coding techniques take advantage of the correlation between adjacent pixels. This mainly consists of the following three components:

(1) Prediction of the current pixel value

(2) Calculating the prediction error

(3) Modeling the error distribution

The value of the current pixel is predicted based on the pixels that have already been coded. Due to the correlation property among adjacent pixels in an image, the use of predictor can reduce the amount of information bits required to represent an image.

In predictive coding also called Differential Coding such as Differential Pulse Code Modulation (DPCM), the transmitter and receiver process the image in some fixed order. The current pixel is predicted from the preceding pixels which have been reconstructed. The difference between the

current pixel P(x, y) and its predicted value $P^1(x, y)$, the *prediction error* d(x, y), is then quantized, encoded and transmitted to the receiver. If the prediction is well defined, then the distribution of the prediction error is concentrated near zero and has lower first order entropy than the entropy of original image.

The design of this predictive coding scheme involves two main stages, predictor design and quantizer design. Although general predictive coding is classified into AR, ARMA etc, auto-regressive model has been applied to image compression. Predictive coding can further be classified into Linear and Non-linear AR models. Non-linear predictive coding, however, is very limited due to the difficulties involved in optimizing the coefficients extraction to obtain the best possible predictive values.

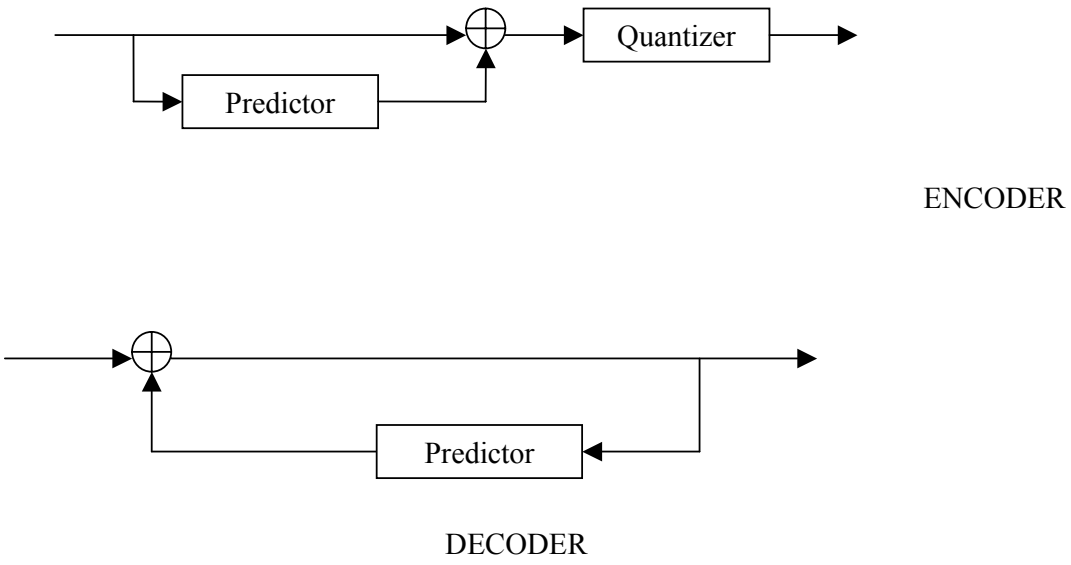


Fig 2.1 Predictive Coding (a) Encoder (b) Decoder

2.2.4 Fractal Coding:

Fractal Coding is a lossy compression method which uses fractals to compress images. Fractal parameters, like fractal dimension, lacunarity, and others provide efficient methods of describing imagery in a highly compact fashion for both intra and inter frame applications. This method is used to compress the natural scenic images where certain parts of the image resemble the other parts of the same image. Fractal compression is much slower than the JPEG compression.

CHAPTER 3

IMAGE COMPRESSION USING JPEG STANDARD

JPEG stands for Joint Photographic Experts Group (collaboration between ITU and ISO) which is a group of experts working with the gray scale and color images. JPEG corresponds to the international standard for digital compression of continuous tone (multilevel) still images. JPEG compression algorithms involves eliminating redundant data, the amount of loss is determined by the compression ratio, typically about 16:1 with no visible degradation. For more compression where noticeable degradation is acceptable compression ratios of up to 100:1 can be employed. The JPEG algorithm performs analysis of image data in such an order to generate an equivalent image which can be represented in a much compact form and needs less space to store. JPEG has two schemes of compression the lossy JPEG, the compressed image when decompressed back is not the same here and lossless JPEG compression does not lose any image data when decompressed back. JPEG usually works by discarding the data information a human eye cannot recognize like slight changes in color are not perceived by the image as the changes in the intensity are. Due to this fact we can see that JPEG compresses color images more efficiently than the gray scale images. Thus, most multimedia systems use compression techniques to handle graphics, audio and video data streams and JPEG forms the important compression standard with various compression techniques as building blocks.

3.1 JPEG Image Compression System:

The main components of the image compression system are:

- Source encoder (DCT based)
- Quantizer
- Entropy encoder

In the image compression algorithm the image is first split into 8x8 non overlapping blocks and the two-dimensional DCT is calculated for each block. The DCT coefficients are then ordered in a zigzag pattern so that the coefficients corresponding to the lower frequencies are ordered first. The DCT coefficients are then quantized. Insignificant coefficients are set to zero and since not all the coefficients are used a delimiter are used to indicate the end of required coefficient sequence in a block. The coefficients are then coded and transmitted. The JPEG receiver or decoder then decodes the quantized DCT coefficients, performs inverse two-dimensional DCT of each block and then puts the blocks together into an image.

3.2 Discrete Cosine Transformation (DCT):

The DCT is closely related to the DFT in that they can be considered to perform the same task of converting a signal into elementary frequency components. DCT is superior to DFT in that DCT is real valued and provides better approximation of the signal with fewer coefficients. DCT helps separate the image into parts of differing importance.

A two-dimensional DCT of an N1xN2 matrix is defined as follows:

The series form of the 2D discrete cosine transform (2D DCT) pair of formulas is

$$X[k_1,k_2] = c[k_1]c[k_2]\sum_{n_1=0}^{N-1_1}\ \sum_{n_2=0}^{N_2-1} x[n_1,n_2]\cos\left(\frac{\Pi(2n_1+1)k_1}{2N_1}\right)\cos\left(\frac{\Pi(2n_2+1)k_2}{2N_2}\right)$$

for $k_1 = 0,1,2,.....N_1-1$ and $k_2 = 0,1,2,.....N_2-1$ (1)

$$c[n_1,n_2] = \sum_{k_1=0}^{N-1_1}\ \sum_{k_2=0}^{N_2-1} c[k_1,k_2]X[k_1,k_2]\cos\left(\frac{\Pi(2n_1+1)k_1}{2N_1}\right)\cos\left(\frac{\Pi(2n_2+1)k_2}{2N_2}\right)$$

for $n_1 = 0,1,2,.....N_1-1$ and $n_2 = 0,1,2,.....N_2-1$ (2)

$\alpha[k]$ is defined as:

$$\alpha\,[\mathrm{k}] = \begin{cases} \sqrt{\frac{1}{N}} & for\ \ k = 0 \\ \sqrt{\frac{2}{N}} & for\ \ k = 1,2,....,N-1 \end{cases} \tag{3}$$

Where the input image is $N_1 \times N_2$, $x[n_1,n_2]$ is the intensity of the image and $X[k_1,k_2]$ is the DCT coefficient in the k_1th row and k_2th column.

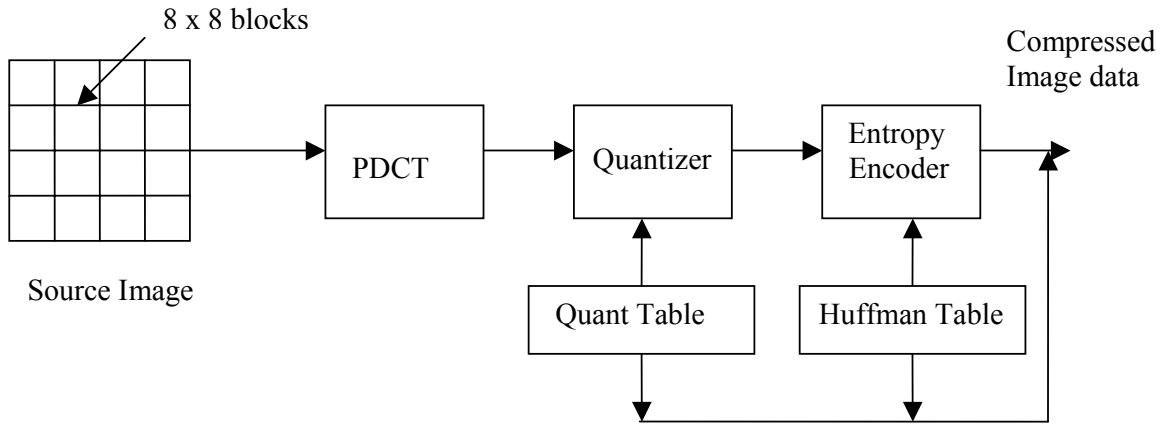


Fig 3.1 JPEG encoder Block Diagram

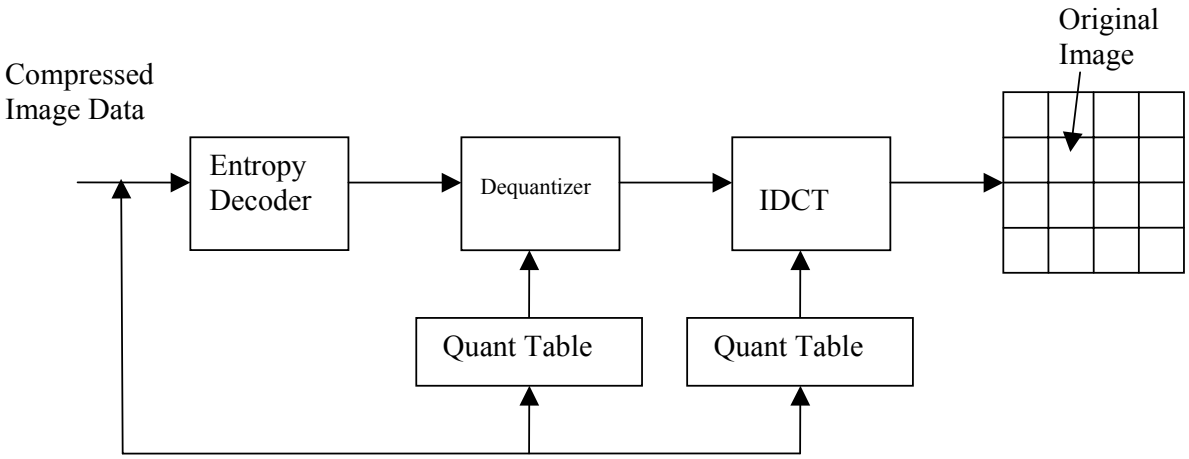


Fig 3.2 JPEG Decoder Block Diagram

Each 8x8 block is compressed into a data stream in the DCT encoder and because the adjacent image pixels are highly correlated the forward DCT concentrates most of the data in the lower spatial frequencies. The output from the FDCT is then uniformly quantized by using a designed quantization table. For a typical 8x8 sample block from a source image, most of the spatial frequencies have zero or near-zero amplitude and need not be encoded. The DCT introduces no loss to the source image samples it simply transforms them to a domain in which they can be more efficiently encoded. The output from the FDCT (64 DCT coefficients) is uniformly quantized with a designed quantization table. At the decoder the quantized values are multiplied with the quantization table elements to retrieve the original image. After quantization the coefficients are placed in a 'zig – zag' manner which helps in entropy coding by placing the low-frequency non-zero coefficients before high frequency coefficients. The DC coefficients are differently encoded. The zig zag sequence is as shown below:

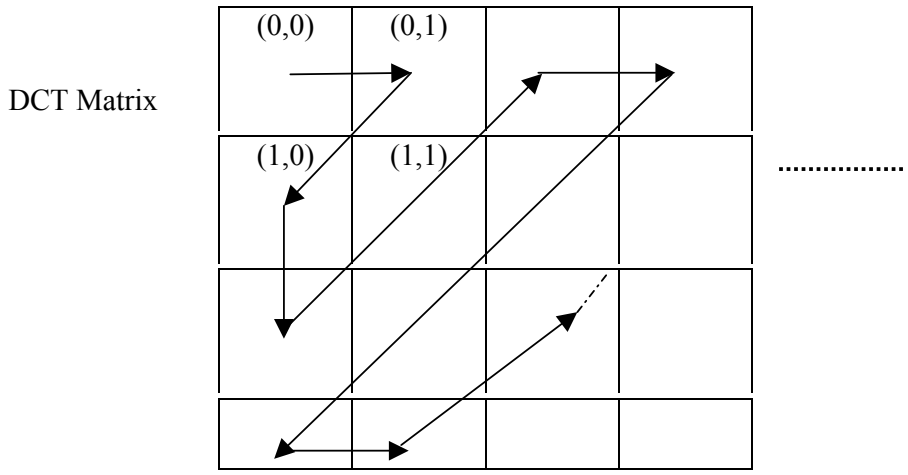


Fig 3.3 Zig – Zag Sequence for Binary Coding

3.3 Quantization:

Each output from the DCT is divided by a quantization coefficient and rounded to an integer to further reduce the values of the DCT coefficients. Each of the DCT output coefficients has its own quantization coefficient with the higher order terms having the larger quantization coefficients which make the higher order coefficients quantized more profoundly than the lower order coefficients. The quantization table is sent along with the compressed data to the decoder. The quality factor set affects the amount of quantization performed on the image. Too much and too little quantization effect the image by making it either blurry or causes loss of information.

3.4 Entropy Encoder:

Entropy coding achieves additional compression by encoding the quantized DCT coefficients based on their statistical characteristics. Huffman coding or Arithmetic coding can be used. Arithmetic compression involves mapping every sequence of pixel values to a region on an imaginary number line between 0 and 1. This region is then represented as a binary fraction of

variable precision (number of bits). Less common sequences are represented with a binary fraction of higher precision.

3.5 Differential Pulse Code Modulation (DPCM):

In a DPCM model a certain number of pixels in the neighborhood of the current pixel are considered to estimate/predict the pixel's value. In the sense the coding of the DC coefficient is done by the differential between the quantized DC coefficient of the current block and the quantized DC coefficient of the previous block. The inverse DPCM returns the current DC coefficient value of the quantized block being processed by summing the current DPCM code with the previous DC coefficient value of the previous quantized block.

3.6 RLE:

The AC coefficients usually contain a number of zeros. Therefore RLE is used to encode these zero values by keeping *the skip* and *the value* where skip is the number of zeros and value is the next non-zero component.

CHAPTER 4

NEURAL NETWORK FOR IMAGE COMPRESSION

Artificial Neural Networks are hardware or software systems based on the simulation of a structure similar to that a human brain has. The important element here is the information processing unit which is composed of one or more layers of groups of processing elements called neurons. The neurons are connected in such a way that the output of each neuron goes as input to one or more neurons, to solve the problems. There are three layers in each Neural Network system: Input layer, Hidden layer and output layer. The input layer and the output layer are of the same size while the hidden layer is smaller in size. The encoded input goes to the hidden layer and then to the output layer where it is processed (decoded) in the network of neurons. The ratio of size of input layer and the intermediate hidden layer is the compression ratio. In most neural network image compression techniques importance is given to the quality of the image for a give compression ratio. But image compression using neural network techniques is considered because of certain advantages of NN such as: their ability to adaptability to learn from the data given for the training and faster decoding. The process of Neural Networks can be divided into: the arrangement of neurons into various layers, deciding the type of connection between neurons inter-layer and intra-layer, finding the way the output is produced from the neurons depending on the input received the learning for adjusting the weight connections.

4.1 Connection Between Neurons:

The type of connection between neurons can be: inter–layer, connection between two layers and intra-layer, connection between neurons in one layer. Inter-layer connections can be classified into: fully connected where each neuron from a layer is connected to each neuron from another layer; partially connected where each neuron from a layer need not be connected to every neuron

in another layer; feed-forward where neurons in a layer sends output to neurons in another layer but don't receive any feedback which makes the connection between neurons one-directional; bi-directional where there is a feedback from the neurons in the other layer when they send their output back to the first layer; hierarchical where the neurons in a layer are connected only to the neurons in the next neighboring layer; resonance-two directional connection where neurons continue to send information between layers until a certain condition is satisfied. Neural Networks can also be classified based on the connection between input and output: auto associative, input vector is same as the output. These can be used in pattern recognition, signal processing, noise filtering, etc; heteroassociative, output vector differs from the input vector.

4.2 Basic Principles of NN Architectures:

The value of the input to a neuron from its previous layer can be computed by an input function normally called a summation function. This function can simply be said as the sum of weighted inputs to the neuron. There are also two specific types of inputs in a network: external input where a neuron receives input from external environment and bias input where a value is sent for neuron activation controls in some networks. Normally the input to many NNs is normalized. After the input from the summation function is received the output of a neuron is computed and sent to the other connected neurons. This output to a neuron is normally computed using a transfer function which can be a linear or non-linear function of its input. The transfer function is chosen according to the problem specified. Some of the transfer functions are the step function, signum function, sigmoid function, hyperbolic-tangent function, linear function, and threshold linear function.

4.3 Learning:

The process of calculation of weights among neurons in a network is called Learning. The network can be considered by supervised learning and unsupervised learning. In the supervised learning the network is given with a set of input and a desired output set. The network weights here are adjusted by comparing these outputs with the desired outputs. In the unsupervised learning the network is given only a set of input without any desired output set making the network to adjust the weights by itself to improve the clustering of data. This is commonly used for pattern recognition and clustering.

Depending on the number of layers Neural Network designs can be: single layered only with the output layer or multi-layered with one or more hidden layers. A NN can be deterministic where the impulses to the other neurons are sent when a neuron gets to a certain activation level and stochastic where it is done by a probabilistic distribution. A NN can also be a static network where the inputs are received in a single pass and dynamic network where the inputs are received over a time interval.

Two major techniques of NN for image compression are the single – structure NN technique and the parallel – structure NN technique.

4.4 Single Structure NN-based Image Compression:

Many approaches have been implemented for a single structure NN like: Basic back – propagation algorithm, Hebbian Learning based algorithm, Vector Quantization NN algorithm and Predictive Coding NN algorithm. In this the image is split into J blocks $\{B_j, j=1, 2\ldots J\}$ of size M x M pixels. The pixel values in each block are rearranged to form a M^2 length pattern $\boldsymbol{C}_j = \{C_{1,j}, C_{2,j}, \ldots, C_{M^2,j}\}$, where $j = 1, 2, \ldots, J$ ($C_{i,j}$ is the ith element of the jth pattern). The input patterns are normalized with the transformation $\boldsymbol{P}_j = f(\boldsymbol{C}_j) = (\boldsymbol{C}_j - m\boldsymbol{c}_j)/\sigma\boldsymbol{c}_j$ where $m\boldsymbol{c}_j$ and $\sigma\boldsymbol{c}_j$ are,

respectively, the average and standard deviation of $\boldsymbol{C}_j$ because the NNs operate more efficiently when the input is in the range [0, 1]. These patterns are used both as inputs and outputs in the training of NNs. The NN consists of three layers as said above the input, hidden and the output with M^2, H and M^2 nodes in each.

4.4.1 Training:

The configuration of Neural Networks for the applications in which the parameters of the network are adjusted such that the network exhibits the desired properties is called training. The network is trained so as to minimize the mean square error between the output and the input values to maximize peak signal to noise ratio. Based on the way the weights are updated the training can be done by online training, the weights here are updated for each input and Batch wise training, the weights here are updated when a complete batch is input to a neural network.

The Neural Network acts as coder/decoder. The coder consists of input-to-hidden layer weights $v_{i,k}$ { i=1,…,M^2 and k=1,…,H},and the decoder consists of the hidden-to-output layer weights $w_{k,m}$ {k =1,…,H and m =1,…,$M2$} where $v_{i,k}$ is the weight from the i^{th} input node to the k^{th} hidden node and $w_{k,m}$ is the weight from the k^{th} hidden node to the m^{th} output node. Compression is achieved by setting the number of hidden nodes smaller than the number of input nodes and propagation of the patterns through the weights *vi,k.* Considerable compression is achieved when the hidden layer nodes are quantized. The coding product of the j_{th} pattern $\boldsymbol{P}_j$ is the hidden layer output $\mathbf{O}_j$ = {$O_{1,j}$, $O_{2,j}$, . . ., $O_{H,j}$}. The set {$\mathbf{O}_j$, mCj,$\sigma\boldsymbol{C}_j$} together with weights $w_{k,m}$ is sufficient for reconstructing an approximation $\boldsymbol{C}_j$ of the original pattern $\boldsymbol{C}_j$ in the decoding phase.

4.5 Parallel – Structure NN-based Image Compression:

The parallel structure is considered as a single structure with multiple hidden layers. Four networks $\{NET_k, k = 1, 2, 3, 4\}$ with different number of hidden nodes (4, 8, 12, and 16) are used to achieve a compression ratio 8:1 with each NN trained as a single structure NN. Each pattern is associated with a NET_k. It is assumed that the larger the number of hidden nodes of the NN to which a pattern is assigned, the smaller the associated error $e_j^2 = (\hat{p}_j - p_j)(\hat{p}_j - p_j)^T$ between the original p_j and estimated patterns $\hat{p}_j$. It is an iterative procedure after this. At each iteration the aim is to reduce the total error $E^2 = \sum e_j^2$ without changing the compression ratio. If $de^2_{j,k}$ is the error caused due to reassigning pattern p_j from NET_k to NET_{k+1} then the error is reduced by reassigning a pair of patterns. Pattern $\boldsymbol{P}_{j1}$ is reassigned from NET_{k1} to NET_{k1+1} if the reassignment causes a maximum error decrease $de^2_{ji,ki} = \max_{jk}(de^2_{j,k})$, and pattern $\boldsymbol{P}_{j2}$ is reassigned from NET_{k2} to NET_{k2-1} if this results in a minimum error increase $de^2_{ji,ki} = \min_{jk}(de^2_{j,k})$. Iterations continue as long as the error E^2 decreases.

This parallel architecture has the advantage of providing better image quality for a given CR than other methods like JPEG in terms of error E^2, including both single and parallel structures. Coding is faster than previous parallel architectures. But, training is still significantly slow due to the use of multiple networks. Moreover, the total number of weights is large. Thus, training cannot be part of the coding process. Thus, the compression quality depends on the training data and their similarity to the test images.

CHAPTER 5

PROPOSED ADAPTIVE ALGORITHM

The novel Image compression technique implemented in this thesis is based on a cascade of adaptive transforms. The small number and fast estimation of learning parameters make the training of data, to be a part of encoding process which makes this technique apt for image compression. This method uses a cascade of adaptive units called adaptive cascade architecture (ACA). Each unit is equivalent to a feed-forward neural network with a single node at the hidden layer. The coding is independent of training data. The encoder and decoder of this technique are shown in the figures below:

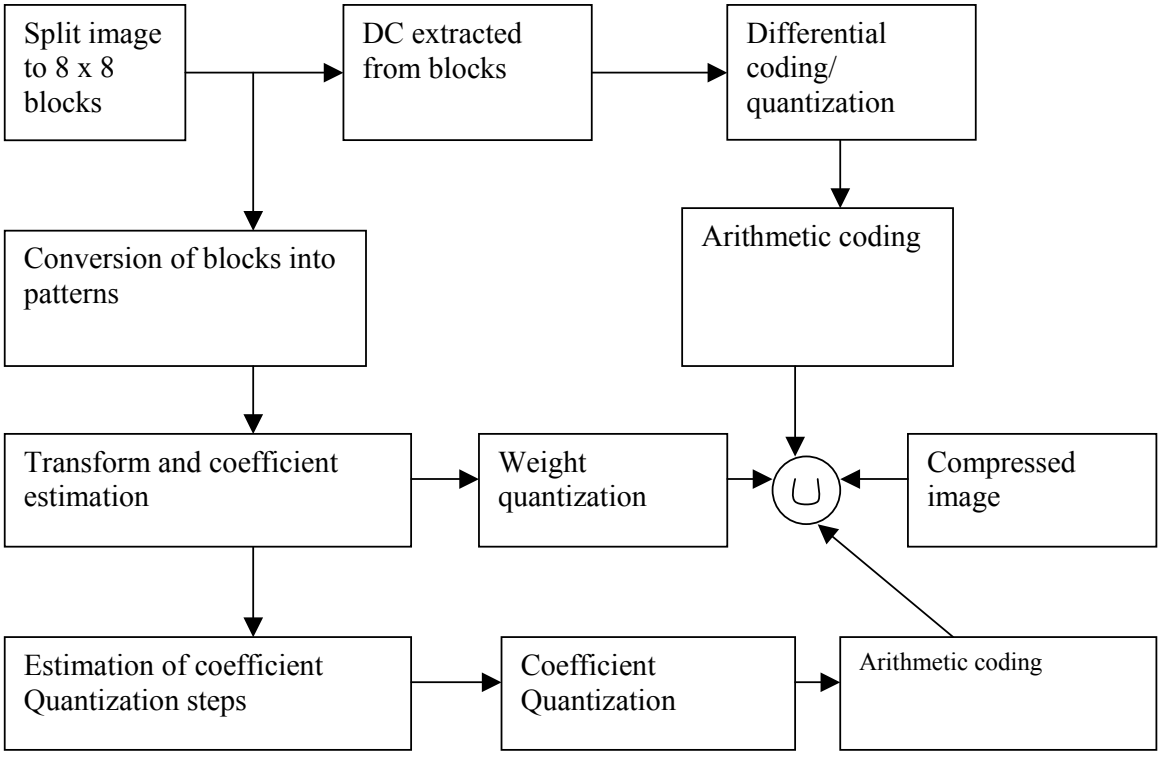


Fig. 5.1 Encoding Phase of the proposed architecture

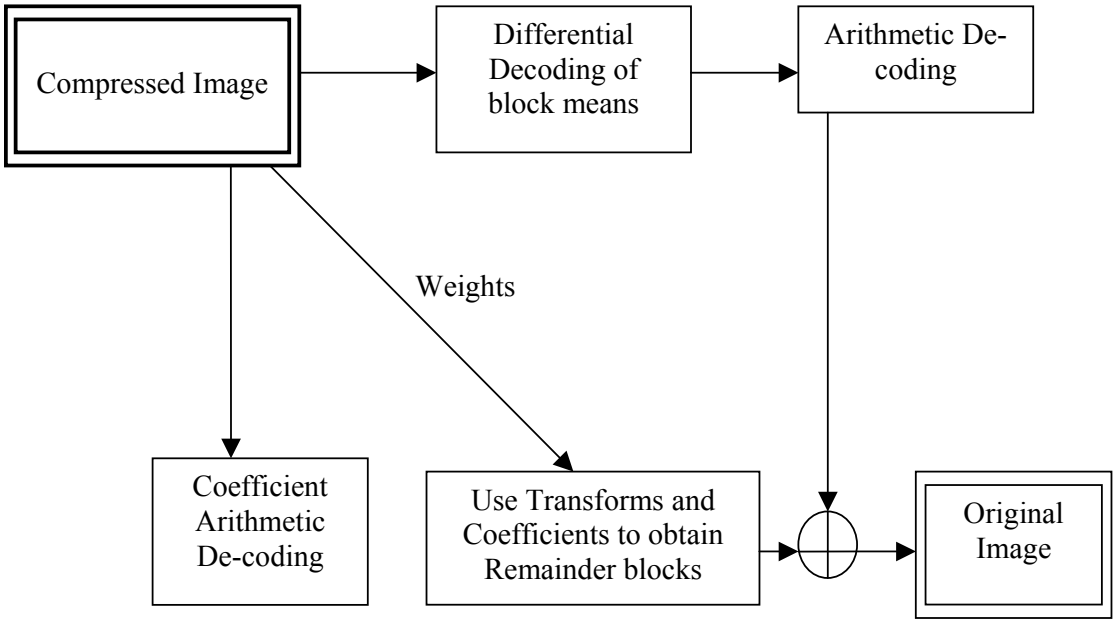


Fig 5.2 Decoding Phase of the proposed algorithm

5.1 Encoding Process:

This section describes the details of each block in the encoder:

Image Splitting into Blocks:

The image is split into J blocks (B_j, j =1, 2, 3… J) of size $M \times M$ pixels. Large processing blocks provide a smaller number of transform coefficients but the transform weights, which need to be stored are increased.

Block means from each block:

The mean of each block j denoted as m_{cj} contains some significant amount of information it is extracted from each block.

Differential coding/quantization of mean values:

The DC values of consecutive blocks j and j+1,namely m_j and m_{j+1} are likely to be similar thus they are differentially encoded: the difference of consecutive mean values $dm_j = m_{j+1} - m_j$ is linearly quantized. The number of quantization levels depends on the compression ratio.

Conversion of remainder blocks into patterns:

The pixel values in each non-overlapping block are arranged to form a M^2 length pattern $C_j = (C_{1,j}, C_{2,j}, ..., C_M{}^2{}_{,j})$, where j=1, 2, ….. J ($C_{i,j}$ is the i^{th} element of the j^{th} pattern). The input patterns are defined by $P_j = f(C_j) = (C_j - m_{Cj})$. The transform and coefficient estimators will be referred to as units. Patterns P_j are used as inputs/desired-outputs to train the first unit.

Weights and Coefficients estimation:

This step is considered as an alterative to DCT, the heart of JPEG. It involves the estimation of weights and their corresponding coefficients. The output of the first unit's hidden node corresponding to the j^{th} pattern is denoted as $O_{j,k}$, k=1. The first unit's estimated weight vector is denoted by

$$W_k = [W_{1,k}, W_{2,k}, ... , W_M{}^2{}_{,k}]; k=1 \qquad (4)$$

This is equivalent to the weight vector between the hidden node and the output layer. W_k is the k^{th} transform i.e. the i^{th} weight of the k^{th} unit. The weight vector between the input node and the hidden layer is not required to be implemented. The output coefficient $O_{j,1}$ and the weight W_1 are necessary in the decoding process.

The first unit's error patterns are denoted as

$$e_{j,1} = [e_{1,j,1}, e_{2,j,1}, \ldots\ldots, e_{M^2,j,1}] \qquad (5)$$

These are the difference between the original p_j and the estimated patterns $\hat{p}_j$ at the output of the first unit when the training is complete. $e_{i,j,k}$ is the i^{th} element of the j^{th} error pattern at unit k. If a set of error patterns at the output of unit k is defined as R_k then only a subset of these error patterns is used as input/output to train the next unit. This subset R_k^s consists of S error patterns $e^s_{j,k}$,j=1,2,…..,S the square sum of whose is larger than a given threshold Q. $R_k^s = \{ e^s_{j,k} \in R_k, e^s_{j,k} e^{s\,T}_{j,k} > Q \}$. Again the second unit's hidden node output coefficients $O_{j,2}$ and weight vectors W_2 are stored for the decoding process.

The second unit's error patterns $e_{2,j}$ are similarly defined as the difference between the actual $e^s_{j,1}$ and the estimated error patterns $\hat{e}^s_{j,1}$ at the second unit's output. Only a subset of these new error patterns $e_{2,j}$ are used as inputs/desired-outputs to train the third unit. This adding and training of units is repeated until the compression ratio does not exceed a given target. As only a subset of error patterns trains each unit, the number of outputs $O_{j,k}$ per unit k is variable. This allows assignment of image-blocks with larger estimation error to more units, with the same CR.

The threshold Q is defined as

$$Q = a\left[\frac{1}{M^2}\sum_{i=1}^{M^2}\mathrm{var}_j\left(e_{i,j,1}\right)\right]CR \qquad (6)$$

From the Equation it can be seen that the threshold Q is proportional to the desired compression ratio CR. The threshold can be set to a low value because a low compression ratio causes a small coding error. The threshold is also proportional to the average standard deviation of the error patterns between the original and the encoded images which is the similarity measure between the error patterns. A small average (ASD) indicates the similarity of the error patterns through out

the image blocks. As a result the additional adaptive units are expected to produce a relatively small coding error and hence the threshold can be ser low. In the Equation a, is a fixed parameter and is given a value of 1.2 for all the experiments.

The advantage of this adaptive cascade technique over the existing parallel NN architectures is that total number of weights is relatively smaller and number of hidden node outputs is variable. The training time is also low because of the low computational complexity of the units. The algorithm converges in 4-5 iterations by using a set of equations:

$$O_{j,k} = W_k \, T_{j,k}^T / W_k \, W_k^T \qquad (7)$$

$$W_k = \sum_j O_{j,k} \, T_{j,k} / \sum_j O^2_{j,k} \qquad (8)$$

$$T_{j,k} = P_j \quad \text{if } k=1$$
$$= e^s_{j,k-1} \quad \text{otherwise} \qquad (9)$$

Equation 7 gives the unit's optimum hidden layer output coefficients $O_{j,k}$ given the unit's weights. This is the result of minimizing the sum of squares (SSE) between the input and output patterns.

$$SSE_k = \sum_j \left(T_{j,k} - \overset{\Lambda}{T}_{j,k} \right) \left(T_{j,k} - \overset{\Lambda}{T}_{j,k} \right)^T \qquad (10)$$

Where $\overset{\Lambda}{T}_{j,k} = O_{j,k} W_k$, for unit k with respect to $O_{j,k}$. Equation (8) gives unit's optimum unit's weights given the hidden layer output coefficients $O_{j,k}$. This is the result of minimizing $SSE_{j,k}$ with respect to W_k. The conditions from which the Equations (7) and (8) are derived are:

$$\frac{\partial SSE_k}{\partial O_{j,k}} = 0 \text{ and } \frac{\partial SSE_k}{\partial W_k} = 0 \qquad (11)$$

since only the transform weights and the hidden layer output coefficients are needed in the decoding process it is necessary to find the optimum set $\{W_k, O_{j,k}\}$ for each transform unit [c]. The algorithm based on equations (7) and (8) directly gets the sub optimum values for both W_k and $O_{j,k}$. The weights and coefficient estimation block is as shown in the figure below:

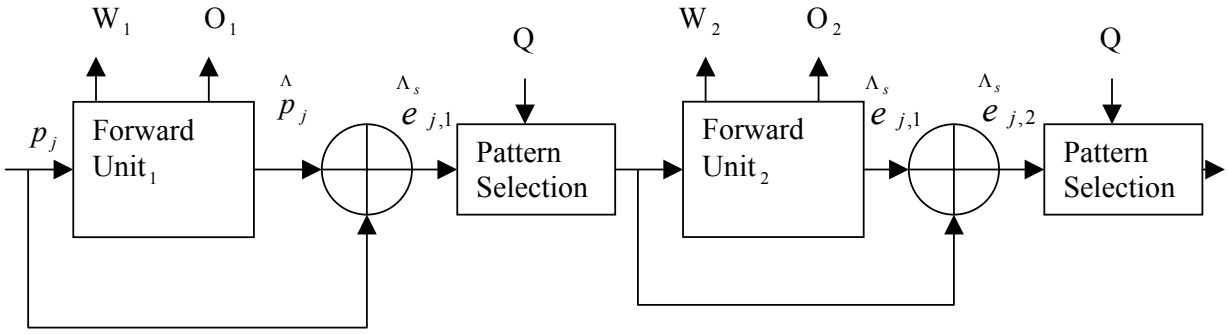


Fig 5.3 Weights and Coefficients estimation Block (Encoding)

Weight Quantization:

After the adaptation the weights are linearly quantized using 256 quantization levels for weight values in the interval [-1, 1]. Before quantization the weights are normalized so that the maximum absolute value equals to 1 as shown below:

$$W_k = W_k / \max_i \left(\left| W_{i,k} \right| \right) \qquad (12)$$

the corresponding hidden output coefficients are scaled to leave the product $O_{j,k}W_k$ unchanged.

$$O_{j,k}^{scaled} = O_{j,k} \cdot \max_i \left(\left| W_{j,k} \right| \right) \quad (13)$$

Coefficient Quantization:

The coefficients estimated and rescaled from the above are linearly quantized. As the algorithm here estimates the best coefficients/weights pair at any given stage k, the coefficients are more likely to obtain lower absolute values as the number of transforms increases. Thus there will be relatively high concentration of coefficient values around 0. Therefore depending on the type of the image the quantization levels can be more or less in order to decrease the quantization error. Thus the coefficients are quantized as

$$O_{j,k}^{quant} = round\left(\frac{15.O_{j,k}^{scaled}}{\Delta.\max\left(\left| O_{j,k}^{scaled} \right| \right)} \right) \quad (14)$$

Where Δ is greater than or equal to 1 and is used to refine the quantization levels. A large Δ implies small values for the quantized coefficients $O_{j,k}^{quant}$. Since the arithmetic coding depends on the probability of occurrence of the symbols higher compression can be achieved if certain coefficients are probable which is done from a large Δ value as it places a considerable amount of coefficient values around zero. The Δ value is estimated as follows:

$$\Delta^t = \begin{cases} (1+\mu^t)\Delta^{t-1}, & if\ SSE\{\Delta = \Delta^t\} < \varepsilon SSE\{\Delta = 1\} \\ (1-\mu^t)\Delta^{t-1}, & if\ SSE\{\Delta = \Delta^t\} > \varepsilon SSE\{\Delta = 1\} \end{cases} \quad (15)$$

and

$$\mu^{t} = \begin{cases} \dfrac{\mu^{t-1}}{2} & \text{if } \left\{SSE\left\{\Delta = \Delta^{t}\right\} > \varepsilon\ SSE\left\{\Delta = 1\right\}\right\} AND \left\{SSE\left\{\Delta = \Delta^{t-1}\right\} < \varepsilon\ SSE\left\{\Delta = 1\right\}\right\} \\ \mu^{t-1}, otherwise \end{cases} \quad (16)$$

where $SSE_{\{\Delta=x\}}$ is the error as in eq (10) but the values of $O_{j,k}$ and W_k are replaced by their quantized values at $\Delta=x$. μ is the learning parameter at iteration t.

Arithmetic coding for block mean values:

This is used for lossless coding of the quantized mean differences m_{cj+1}-m_{cj}.

Arithmetic coding for the coefficients:

The coefficients from each block are placed in order and separated from the next block coefficients by using a delimiter. The symbol histogram can be estimated to globally for small and locally for the large images.

5.2 Decoding Process:

Each block of the image is decoded using the reverse of the encoding process. Each block is decoded using the set $\{O_{j,k}, W_k\}$, with all the units k used to encode the block. Consider, if the first unit produces an estimate of patterns $\hat{p}_j = O_{j,1} W_1^T$ and the second unit an estimate of the first unit's error patterns $\hat{e}^{s}_{j,1} = O_{j,2}\ W_2^T$. The decoded block can be obtained by summing of all

these estimates and the block average m_{cj}. The detailed decoding block is as shown below:Decoder:

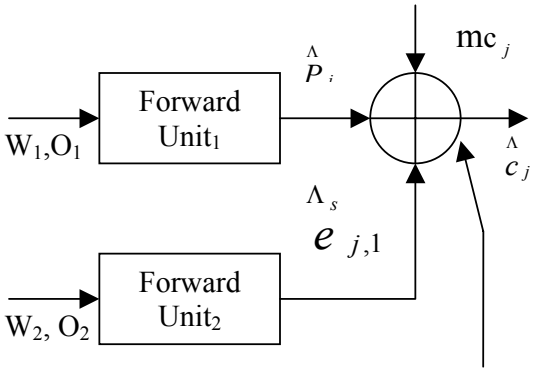


Fig 5.4 Decoding phase of the proposed architecture

CHAPTER 6

IMPLEMENTATION ON THE BOARD

6.1 Overview of the board

The TMS320C6X series of DSP chips are exclusive in that they are the first of their type with Very Long Instruction Word (VLIW). With VLIW the compiler groups together multiple instructions that do not have any dependency into a single VLIW instruction. Since the compiler has done the major task of creating the VLIW instruction stream, the processor now depends on the compiler to provide it with the VLIW, making it simpler. This has two advantages: the clock speed increases and the power consumption remains the same. If there are not enough dependencies between the grouped instructions the compiler pads the VLIW set with NOP instructions but since image processing and signal processing applications are highly repetitive numerical computations of large groups of data are not data independent. If the NOP instructions are padded due to the data dependencies, the processor is not completely utilized so TI came up with VelociTI technology which reduces the problem of memory usage with VLIW making the code size smaller. TI also made another improvement with VelociTI.2 which included specialized instructions for packed data processing for many applications like high – performance and low – cost image processing [8].

6.2 Floating – Point and Fixed – Point:

In the floating – point architecture the set of real numbers is represented by a single-precision float data type or a double precision double data type bits for a fraction containing the number of significant digits, an exponent and a sign bit. These can represent a large range of values.

Floating point arithmetic is very powerful and is also very convenient in programming except for the underflow and overflow conditions to be taken care of. If the function of an arithmetic deals only with the integral values then it is advantageous to use a fixed – point processor as this processor is faster (clock speed increases) with less power consumption due to the simplicity of the hardware.

The above proposed algorithm is implemented on the TI DSP TMS320C6713 DSK board in the floating-point arithmetic. The board is based on high performance, advanced very-long-instruction-word architecture. This board operates at 225MHz, the C6713 delivers up to 1350 million floating point operations per second (MFLOPS), 1800 million instructions per second (MIPS) and with dual fixed/ floating point multipliers up to 450 million multiply- accumulate operations per second (MMACS).

The C6713 uses a two-level cache-based architecture and has a powerful and diverse set of peripherals. The Level 1 program cache (L1P) is a 4K-Byte direct-mapped cache and the Level 1 data cache (L1D) is a 4K-Byte 2-way set-associative cache. The Level 2 memory/cache (L2) consists of a 256K-Byte memory space that is shared between program and data space. 64K Bytes of the 256K Bytes in L2 memory can be configured as mapped memory, cache, or combinations of the two. The remaining 192K Bytes in L2 serves as mapped SRAM.

The C6713 has a rich peripheral set that includes two Multichannel Audio Serial Ports (McASPs), two Multichannel Buffered Serial Ports (McBSPs), two Inter-Integrated Circuit (I2C) buses, one dedicated General-Purpose Input/Output (GPIO) module, two general-purpose timers, a host-port interface (HPI), and a glueless external memory interface (EMIF) capable of interfacing to SDRAM, SBSRAM, and asynchronous peripherals.

The two McASP interface modules each support one transmit and one receive clock zone. Each of the McASP has eight serial data pins which can be individually allocated to any of the two zones. The serial port supports time-division multiplexing on each pin from 2 to 32 time slots. The C6713 has sufficient bandwidth to support all 16 serial data pins transmitting a 192 kHz stereo signal. Serial data in each zone may be transmitted and received on multiple serial data pins simultaneously and formatted in a multitude of variations on the Philips Inter-IC Sound (I2S) format.

In addition, the McASP transmitter may be programmed to output multiple S/PDIF, IEC60958, AES-3, CP-430 encoded data channels simultaneously, with a single RAM containing the full implementation of user data and channel status fields.

The McASP also provides extensive error-checking and recovery features, such as the bad clock detection circuit for each high-frequency master clock which verifies that the master clock is within a programmed frequency range.

The two I2C ports on the TMS320C6713 allow the DSP to easily control peripheral devices and communicate with a host processor. In addition, the standard multichannel buffered serial port (McBSP) may be used to communicate with serial peripheral interface (SPI) mode peripheral devices.

The TMS320C6713 device has two bootmodes: from the HPI or from external asynchronous rom.

The TMS320C67x DSP generation is supported by the TI eXpressDSP set of industry benchmark development tools, including a highly optimizing C/C++ Compiler, the Code Composer Studio Integrated Development Environment (IDE), JTAG-based emulation and real-time debugging, and the DSP/BIOS kernel.

6.3 DSP Boards in Image Processing:

In many cases, image processing algorithms are characterized by repetitive performance of the same operation on a group of pixels. For example, filtering a signal involves repeated multiply-accumulate operations and filtering of an image involves repeatedly performing these multiply-accumulate operations on each pixel of an image while sliding the mask across the image. These repeated numerical computations require a high memory bandwidth. All these imply that image processing involves high computational and data requirements. This can be considered as a reason why image processing applications are increasingly being done on embedded systems. A digital image is represented as a matrix but in C/C++ the image matrix is often flattened and stored as a one-dimensional array. This flattened image can be done in either a row-major or a column major fashion. In the row major fashion the matrix is stored as an array of rows i.e. all the elements of a row are first ordered, then the elements of the next row and so on, and is used in C/C++. In the column major fashion the matrix is stored in the array of columns i.e. all the elements of a column are ordered first then the elements of the next column and so on, and is used in MATLAB.In the implementation of image processing applications on any embedded systems the main problem is inputting the data into the system. The large size input data of an image cannot be directly initialized into an array. Code Composer Studio provides the functions of basic C like the *fwrite, fread, fopen, fclose* but these functions are so slow that it is not even recommended to transfer even moderate amounts of data like a 64x64 image too. To overcome

this CCStudio is built-in with certain input/output facilities like the Real Time Data Exchange (RTDX) and the High Performance Interface (HPI).

6.4 IMPLEMENTATION OF THE ALGORITHM ON BOARD:

The above proposed algorithm is implemented on this board. The coding part of the algorithm is done in the Code Composer Studio IDE which is done in the visual studio.

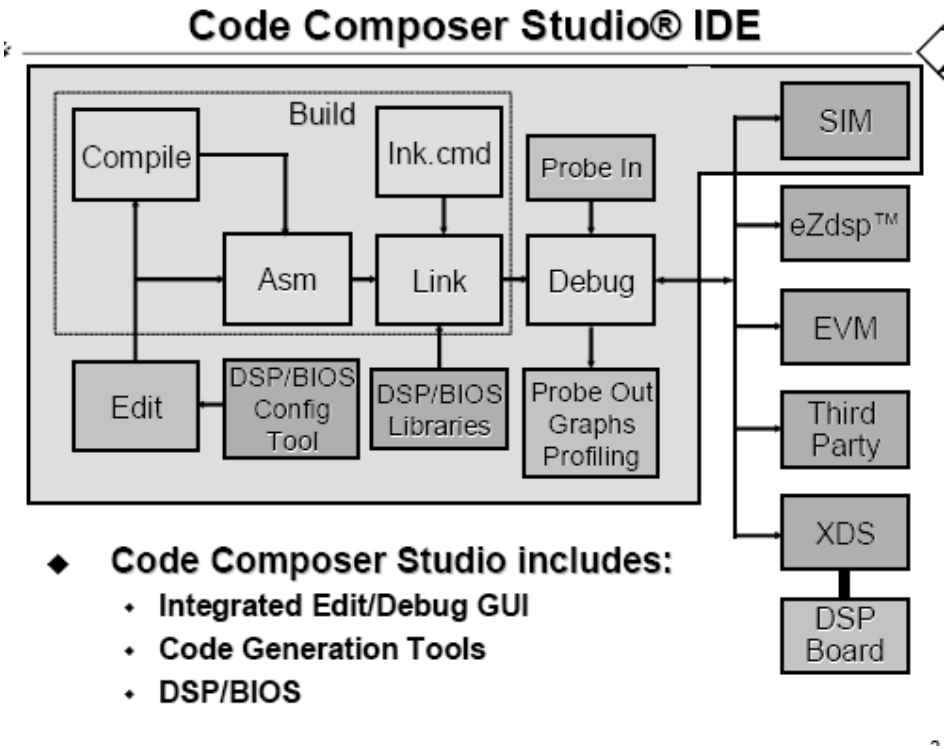


Fig 6.1 Code Composer Studio Architecture from TI Documentation

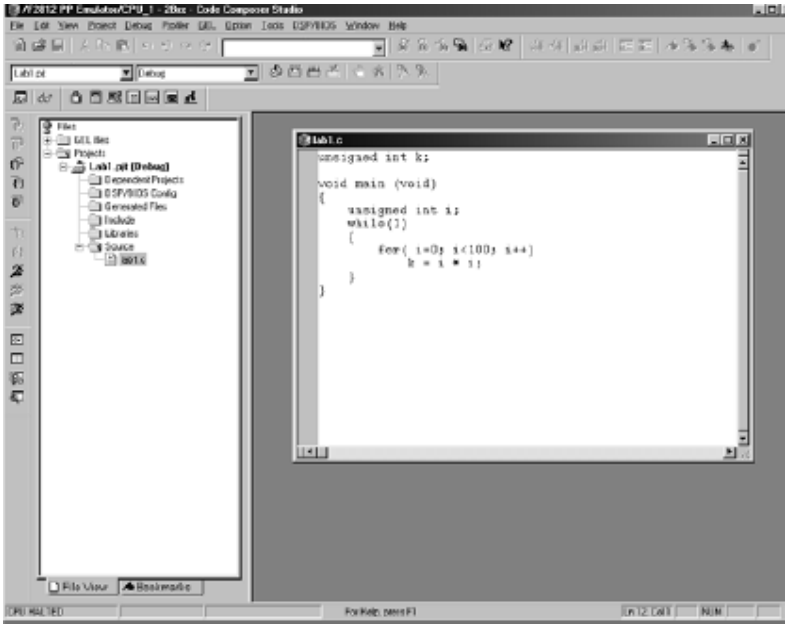
Code composer studio can be used with a simulator (PC) or can be connected to a real DSP system and test the software on a real processor board (DSP). In order to write some type of code on the DSP board a project is to be created, a project consists of source file, library files, DSP/BIOS configuration, and linker command files. The project has the following settings: Build Options (Compiler and Assembler), build configurations, DSP/BIOS and Linker. The build GUI

controls many aspects of the build processes such as: optimization level, target device, compiler/assembly/link options. The step-by-step approach for creating a project with code in the code composer studio is:

- Open code composer studio.
- Create a project based on C.
- Compile, link, download and debug the program.
- Watch variables
- Break points and probe points can be used

To create projects, go to project on the toolbar, **Project->New.** Give the project name, select target type and suitable location of the hard disk. The project creates a subdirectory by itself in the location specified.

To write the source codes, go to file on the toolbar, **File->New->Source file.** Save the file as *.C in the project destination. This looks as

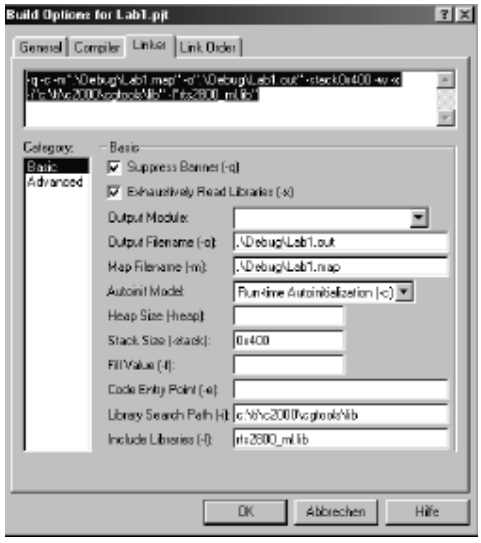


The C source code is now stored in the project subdirectory of the hard disk but it is not a part of the project yet. So, to add it to the project: **Project->Add files to the project,** browse to the location of the source code and say OK.

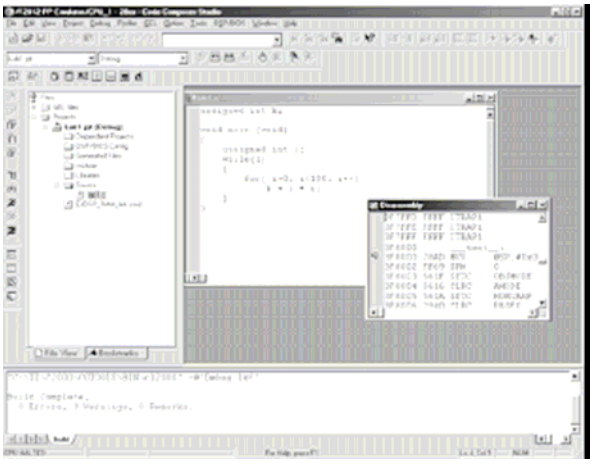
Compile the source code by, **Project->Compile File**. The active source code (the code of the project) will be compiled and if there are any errors they are displayed.

To run the source code the source code is to be linked. Linking is done by adding certain linker library files provided by the TI. This can be done by **Project->Build options->Linker->Library search path** (c:\ti\c2000\cgtools\lib) and **Project->Build Options->Linker->include libraries** (c:\ti\c2000\cgtools\rts2000_lib).

Change the build options as required by the user. The build option box looks as:



The linker puts together the various building blocks needed for the system. This is done with "Linker Command File". This is used to connect the physical memory parts of the DSP's memory with the logical sections of the system. Add the linker command file by: **Project->Add files to the project**. A command file can be written by the user according o his needs. Then Build the project: **Project->Build.**



Linking connects one or more object files (*.obj) into an output file (*.out). This file not only contains the machine code but also information used to debug and to flash DSP.

Now to load the code onto the DSP: **File->Load program->Debug*.out**. To run the code: **Debug->run.**

DSP/BIOS:

DSP/BIOS tool for the Code Composer Studio software is designed to minimize memory and CPU requirements on the target. This can be done by:

- All DSP/BIOS objects can be created in the configuration tool and put into the main program image. This reduces the code size and even optimizes the internal data structures.
- Communication between the target and the DSP/BIOS is performed within the background idle loop such that the DSP/BIOS do not interfere with the main program.
- Logs and Traces used to output, input or for the timed processes are formatted by the host.
- A program can dynamically create and delete objects to be used in the special situations.
- Two I/O models are supported for maximum flexibility and power. Pipes are used for target/host communication and to support simple I/O cases where a thread writes to the pipe and another thread reads from the pipe. Streams are used for more complex I/O operations.

DSP/BIOS is a real time kernel designed for the applications that require scheduling and synchronization, host – to – target communication or real – time instrumentation.

Optimizing C/C++ Code:

A C/C++ program performance can be maximized by using the compiler options, intrinsics and code transformations such as software pipelining and loop unrolling. The compiler has three basic stages when compiling any loop: qualify the loop for software pipelining, collect loop resource and dependency graph information and software pipeline the loop. This combined with

the compiler options make a code fully optimized. The C6x compiler provides intrinsics, functions designed specially to map to the inlined instructions to optimize the code quickly. Intrinsics are specified with an underscore (_) in front of them and are called as any other function is called. For example in the implementation of this thesis there are a few intrinsics used like: *_amemd8* , this is equivalent to the assembly operations Load word/Store word allows aligned loads and stores of 8 bytes to memory and can be used on all the C6x boards. *_extu* is equivalent to extract in the assembly and is used to extract a specified field in src2, sign extended to 32 bits. The extract is performed by a shift left followed by a unsigned shift right. *_itod* is used to create a double register pair from two unsigned integers. *_hi*, returns the high 32 bits of a double as an integer. *_lo* returns the low register of a double register pair as an integer. There are a more of the intrinsics available and can be seen in the TI documentation.

This thesis is put into the DSP processor in the following way: A project compress.pjt is created as said above. Then a C source file compress.c is written. This file contains all the necessary coding for the proposed algorithm. The main part in this implementation is the memory allocation, image loading and memory management. As a large amount of memory is required all the memory allocations and freeing of memory for not so much necessary variables are first done. Then the source file is added to the project. All the necessary library files are also added to the project. The include files are checked to be seen in the project. Then in the DSP/BIOS the memory is assigned. In the DSP/BIOS the system is enlarged and in it the SDRAM is selected. In order to change the size of memory the properties of the SDRAM needs to be checked. In the properties dialog box the starting address, stack size and the heap size are given. Heap is the actual memory we are using out of the stack and hence the heap is always to be smaller than the stack size. The image dimensions of this thesis are taken as 512x512. Due to the size of the resultant memory buffers, not all the image fits into the internal RAM. The DATA_SECTION

pragma (compiler directive) tells the CCStudio to allocate space for the symbols as SDRAM in this case for the input image, image1. So for the linker command file to know this the memory segment in the DSP/BIOS is configured appropriately. A matlab code is written to convert the image into an ASCII text file. This text file is used by the CCS to import data into the DSP. The text file is generated such that each line contains a single word of data. After building the project and loading the executable program onto the DSP, a breakpoint is set in main where the image load is printed. When the program is run the debugger stops at this line. By using the FILE->DATA->LOAD menu item the block of image data is sent into the DSP. When the LOAD dialog box comes up the address where the text file is stored is given, then in the format "Hex" is chosen. By clicking ok on this dialog box the next dialog box allows the user to specify the memory location where it is to be saved, asking for the destination and the data length which in this case is taken as 65536. In order to view the image at any point in the code VIEW->GRAPHS->IMAGE then in the dialog box enter the memory point (variable) to be viewed and the size of the variable(image). To view the contents of any memory location VIEW->MEMORY. The image in the present code then is divided into the blocks then the mean, estimation, encoding and decoding are done as per the algorithm specified in the previous chapter.

CHAPTER 7

RESULTS

The proposed algorithm is tested on four different images and compared with the NN technique and the JPEG technique. The comparisons with the NN do not include lossless step. The advantages of the proposed technique and to justify it being used with JPEG like algorithms than NN. The comparisons are made in terms of the peak-signal-to-noise ratio (PSNR) and computational complexity. The compression ratio is taken as the ratio of the number of pixels in the original image to the number of hidden node output coefficients.

7.1 Comparisons in terms of PSNR:

The table shown below considers the compressing and decompressing of four images with the single structure NN and the proposed adaptive method based on the PSNR. The single structure NN is both trained and tested on the same image in order to avoid the influence of training data on the results. The comparison is done based on the results from the earlier implementations as it is unrealistic due to the high training time requirements. Table 1 shows that the proposed architecture gives higher PSNR for all the tested images and for three different compression ratios (4:1,8:1 and 16:1)

CR	Lena		Baboon		Peppers	
	Single-Structure	Cascade	Single-Structure	Cascade	Single-Structure	Cascade
16:1	28.93	31.36	21.74	21.99	28.28	30.07
8:1	31.87	35.88	23.04	23.42	30.57	35.21
4:1	35.19	38.96	25.07	25.46	33.06	37.35

Comparison between Single-Structure and Proposed algorithm in terms of PSNR

Quality Factor	CR	JPEG		
		Lena	Baboon	Peppers
34.5	16:1	34.24	33.16	33.87
74.9	8:1	38.19	38.79	37.51
92.0	4:1	42.89	43.02	42.47

PSNR for JPEG Algorithm

In JPEG, all transmitted quantized coefficients are entropy coded using DPCM/RLE Huffman coding so it is not possible to directly compare the proposed algorithm with it. Without the entropy coding JPEG gives much lower compression ratio for the same PSNR. It can be seen that PSNR is only about 3 or 4 db below the one from the JPEG.

The plot of the PSNR values the compression ratio CR for the lena image using the proposed adaptive method, single-structure NN and the JPEG is as shown below:

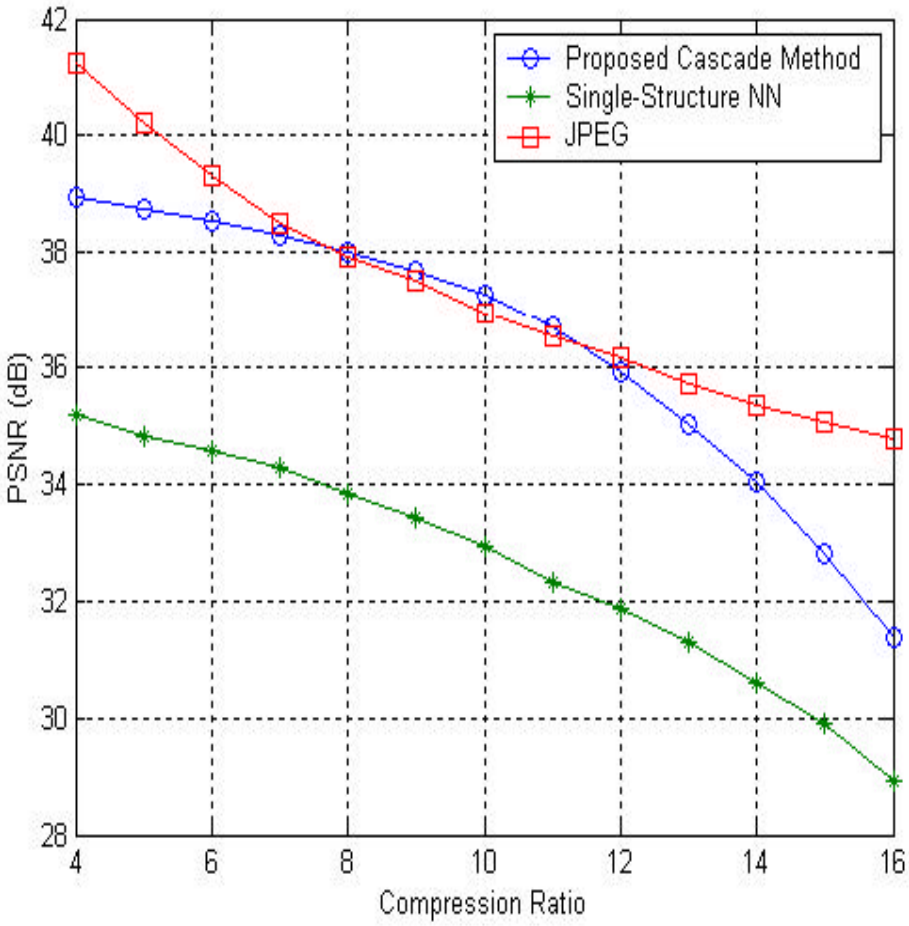


Fig 7.1 PSNR Values for reconstructed Lena Image at Different Compression Ratios

As can be seen the proposed method outperforms the Single-Structure NN but JPEG compression performs better than the proposed method.

The below figures show the same comparison for the pepper and baboon images respectively:

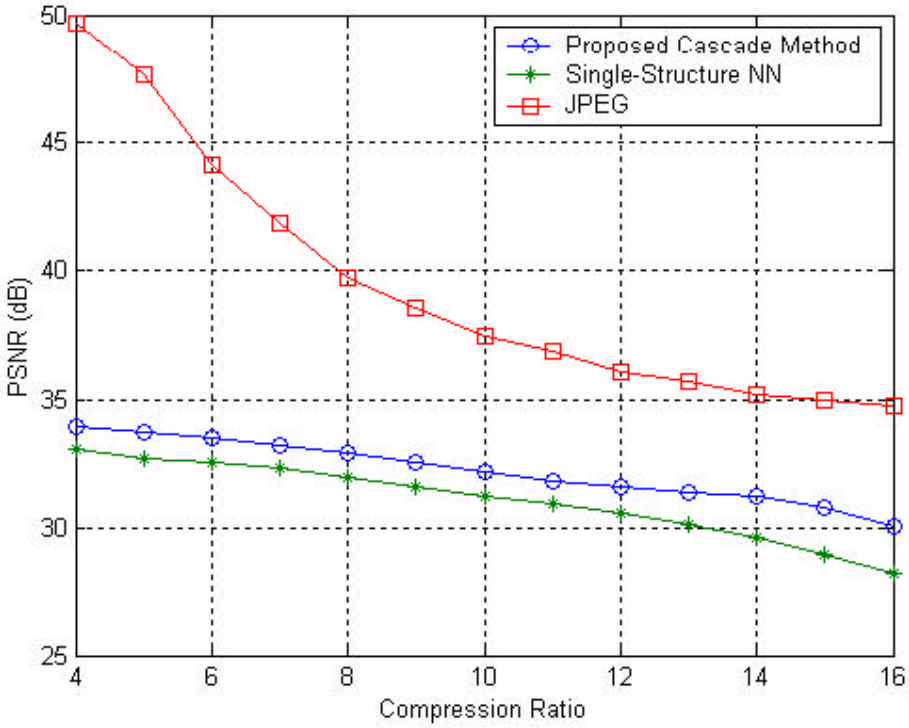


Fig 7.2 PSNR values for reconstructed Pepper image at different Compression Ratios

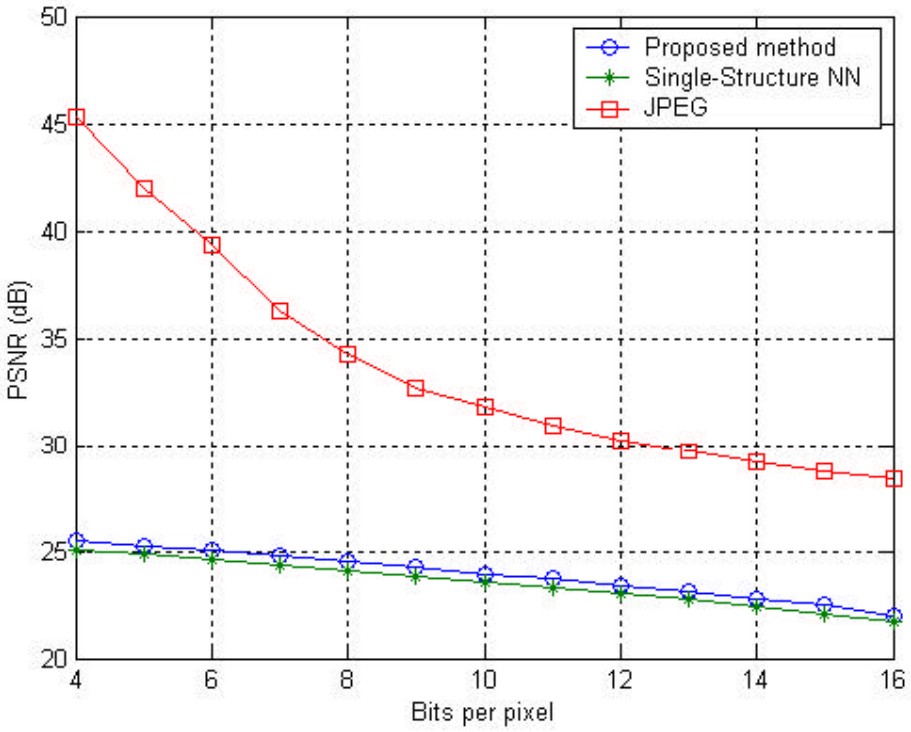


Fig 7.3 PSNR values for reconstructed Baboon Image at different Compression Ratios

The JPEG algorithm provides better PSNR but is visually performance is poor at high compression ratios as can be seen below:

(a) (b)

(c) (d)

Fig 7.4 (a) Original Lena image and reconstructed image using (b) Single-Structure NN (c) Proposed – adaptive method and (d) JPEG algorithms

7.2 Comparison in terms of computational complexity:

JPEG and the proposed algorithm are based on the quantization and lossless coding the computational complexity is compared in terms of their associated transforms. As we have seen in the previous chapters JPEG is based on the DCT and the proposed algorithm is based in the adaptive transforms. The number of multiplications for the two dimensional DCT of an 8x8 block using the fast implementation is 584 [10]. A 512x512 image divided into 4096 blocks requires $2.4*10^6$ operations. In the proposed algorithm the number of operations is variable. Each block requires 64 multiplications and 64 additions as can be seen from equation (5), i.e. a total of 128 operations for a single transformation. For equation (6) it requires approximately 65 multiplications and 65 additions, a total of 130 operations. In addition it requires 64 additions and 64 multiplications, a total of 128 operations for every error calculation of the adaptive quantization step. In all there is a total of approximately NT x IT x (256 + QIT x 128) operations per block, where NT is the average number of transformations per block, IT is the number of iterations per unit for calculation of weights and coefficients and QIT is the number of iterations required in the adaptive estimation of coefficient quantization levels. Therefore although the proposed algorithm includes training in the encoding process it requires fewer operations compared to the parallel NN technique. But it is slower than the JPEG.

CHAPTER 8

CONCLUSIONS

A novel adaptive image compression technique is implemented on the TI's DSP TMS320C6713. The advantage of this algorithm is it requires a smaller number of training parameters and has a fast training phase compared to other adaptive techniques like the NN. Therefore the training can be incorporated into the encoding phase, even after this it is faster in encoding than the parallel structure NN technique. It also has a smaller code error than the single structure and parallel structure NN techniques.

The proposed algorithm is mostly similar to the JPEG algorithm but the DCT in JPEG is replaced with the adaptive transform. The proposed algorithm produces a higher PSNR than JPEG at higher compression ratios. It is also observed that this algorithm can successfully be implemented on the processor boards without much complication.

REFERENCES

[1] “Adaptive Cascade Architectures for Image Compression”, D.Charalampidis ,IEE electronic letters,2003.

[2] S.Carrato ,”Neural Networks for image compression”, The Netherlands; North-Holland,1992.

[3] W.B.Pennebaker and J.L Mitchell,”JPEG still Image Data Compression Standard, New York”.

[4] “Image Compression and segmentation using Neural Networks “,Constantino Carlos Reyes and Ana Laura Aldeco.

[5] G.Qui , M.R.Varley, and T.J.Terrel, ”Image Compression by edge Pattern Learning using Multi layer Perceptrons”,Elecron.Lett,1993.

[6] R.C.Gonzalez and R.E.Woods ,”Digital Image Processing”,Addison-Wesley,1992

[7] TI’s TMS320C6713 Floating point digital signal processor documentation, focus.ti.com/general/docs/lit/getliterature.tsp?genericPartNumber=**tms320c6713**&fileType=pdf.

[8] Embedded Image Processing on the TMS320C6000 DSP by Shehrzad Qureshi.

[9] Digital Signal Processing with C and the TMS320C30 by Rulph Chassaing, Ralph Chassaing.

[10] G.Bi, G.Li and K.K.Ma, “On the Computation of two dimensimal DCT”, IEEE Transactions on Signal Processing, April 2000.

[11] Digital Signal Processing: Laboratory Experiments Using C and TMS320C31 DSK by Rulph Chassaing.

APPENDIX

COMPRESSION OF AN IMAGE:

```
#include <std.h>
#include <log.h>
#include <mem.h>
#include<math.h>

#include "vikascfg.h"

#define X_SIZE 512
#define Y_SIZE 512
#define N_PIXELS X_SIZE*Y_SIZE
#define ADAPT 1


#pragma DATA_SECTION(image1, "SDRAM");
#pragma DATA_ALIGN (image1, 8);
unsigned char image1[N_PIXELS];
unsigned char *image=&image1[0];


float maxo;
int olevels=126;


unsigned char mean(unsigned char*);
unsigned char block1(unsigned char*,int,int,int,int);
unsigned char block2(unsigned char*,int,int,int,int);
unsigned char block3(unsigned char*,int,int,int,int);
unsigned char block4(unsigned char*,int,int,int,int);
void s_array(int, unsigned char*,unsigned char*,short*);
void estimste(float*,float*,float*,short*,int);

extern Int SEG0;

void main()
{
    Ptr
px,ps,pf1,pf4,pf3,pnets_per_block,poj,pojj,pchoose,py,pytemp,pw,po,potemp,pxte
st_short,pxtest_unsigned,pwc,
pmem,pc1,pc2,ppr,ppr1,pimagex;
    short
*ptf1,*frame,*ptf4,*ptf3,*ytemp,*otemp,oc,*xtest_short,xxd,xx1,mut;
    int
wtest,i,j,x,y,rx,cx,shift1=0,shift2=0,mean_diff,smean,total_nets=3,f1=4096,net
,f2,k,done,NSYM2,NSYM3,*pac1,
*pac2;
    int
*choose,stdp,meanp,mxo[5],compr,f3,mb,nbytes=0,numofcoef,pp,wlevels=256,p,coun
t,FLAGE,n,sym,m;
    unsigned char
*image=&image1[0],mn,*mean_array,*test,*current_block,*s,*s1,*sx,mx,*xtest_uns
igned;
    char *ojj,*oj,*nets_per_block,*wc,round,*mem,*enit;
    float *w,*o,fround,fc,*yy,fx1,fx2,err,xcmp,ffr;
```

```
    unsigned r1=0,r2=0;
    double dx;
   // int *pr11,*pr12;
   // int pr[60][34],pr1[60][34];
    int (*pr)[60][128],(*pr1)[60][128],(*rpr),(*rpr1),*zrx1,*zrx2,vi,sys,z;

     int (*imagex)[512][512];
   unsigned char (*ins)[512][512];


/// xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
   int iz,jz,ffx=0,kx,mx;

   short (*pxr)[4096][64];
//xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx



   float fsym,ffw;

    px= MEM_alloc(SEG0, 4096, 8);// for mean
    ps = MEM_alloc(SEG0, 512*512, 8);
    pf1=MEM_alloc(SEG0, 2*512*512, 8);

    pnets_per_block=MEM_alloc(SEG0, 4096, 8);
    poj=MEM_alloc(SEG0, (512*512)+64, 8);
    pchoose=MEM_alloc(SEG0, 4*4104, 8);
    pw=MEM_alloc(SEG0, sizeof(float)*64, 8);
    po=MEM_alloc(SEG0, sizeof(float)*4096, 8);
    py=MEM_alloc(SEG0, sizeof(float)*512*512, 8);
   // pytemp=MEM_alloc(SEG0, 2*512*512, 8);
    pwc= MEM_alloc(SEG0, 4096, 8);
      pc1=MEM_alloc(SEG0, 128*sizeof(int), 8);
     pc2=MEM_alloc(SEG0, 129*sizeof(int), 8);
     pimagex=MEM_alloc(SEG0, 512*512*sizeof(int), 8);



    mean_array=(unsigned char*)px;
   // LOG_printf(&trace, " Load");
     LOG_printf(&trace, " Loaeddddd");

     for(i=0;i<64;i++)
       {
        for(j=0;j<64;j++)
         {
          *mean_array++=mean(image);
          image=image+8;
         }
          image=image+3584;
       }
    LOG_printf(&trace, "Finding Mean done");
     mean_array=(unsigned char*)px;
      s=(unsigned char*)ps;

      for(rx=0;rx<512;rx++)
   {
     //s=sx;
     for(cx=0;cx<512;cx++)
```

```
  {
   x=rx>>3;
   y=cx>>3;
   i=rx-(x<<3);
   j=cx-(y<<3);
   current_block=mean_array+((x<<6)+y);
   if((i<4) & (j<4))
     mx=block1(current_block,x,y,i,j);
   if((i<4) & (j>=4))
     mx=block2(current_block,x,y,i,j);
   if((i>=4) & (j<4))
     mx=block3(current_block,x,y,i,j);
   if((i>=4) & (j>=4))
     mx=block4(current_block,x,y,i,j);
    *s++=mx;

  }

}

 LOG_printf(&trace, "Blocks done");

 // Smoothing surface
s=(unsigned char*)ps;
mean_array=(unsigned char*)px;
frame=(short *)pf1;
image=&image1[0];
for(i=0;i<64;i++)
    {
     for(j=0;j<64;j++)
      {
       smean=(int)mean(s);
       mean_diff=(*mean_array++) - smean;
       s_array(mean_diff,s,image,frame);
       //LOG_printf(&trace, "mean1 %d:",mean_diff);
       s=s+8;
       image=image+8;
       frame=frame+64;
      }
       s=s+3584;
      image=image+3584;
    }

   LOG_printf(&trace, " Smooth done");
       LOG_printf(&trace, " Smooth done");
// MEM_free(SEG0, ps, 512*512);
    //s=(unsigned char*)ps;
    //frame=(short *)pf1;
  nets_per_block=( char *)pnets_per_block;
  oj=( char *) poj;

   for (i=0;i<262208;i++)// use 8 store
    {
     if (i<4096)
        *nets_per_block++=0;
        *oj++=0;
     }

  nets_per_block=( char *)pnets_per_block;
```

```
     oj=(char*) poj; //
     //ojj=( char *) pojj;
     pf4=MEM_alloc(SEG0, 2*512*512, 8);
     pf3=MEM_alloc(SEG0, 2*512*512, 8);
     ptf1=(short *)pf1;
     ptf4=(short *)pf4;
     ptf3=(short *)pf3;
     choose=(int *)pchoose;
     w =(float *)pw;
     o =(float *)po;
     net=0;
     k=0;
     f2=1000;
     compr=4096;
     wtest=1;
     done=1;
     yy=(float *)py;
     //ytemp =(short *)pytemp;
     wc=(char *)pwc;

  while((compr<total_nets*f1)&&(f2>100))
// while(wtest<3)
    {
       wtest++;
       f2=0;
       LOG_printf(&trace, " in while");
      if (net==0)
         {
           stdp=0;
           for(j=0;j<64;j++)
            {
              meanp=0;
              for(i=0;i<4096;i++)
               {
                 meanp=meanp+*(ptf1+(i<<6)+j);
               }
            //   LOG_printf(&trace, " Meanp before %d",meanp);
               meanp=meanp>>5;// meanp=(meanp)/4096
            //   LOG_printf(&trace, " Meanp After %d",meanp);
               for (i=0;i<4096;i++)
               {
                  mut=(*(ptf1+(i<<6)+j));
                  xxd=((mut<<7)-meanp)>>7;
                  stdp=stdp+(xxd*xxd);
                  *(ptf4+(i<<6)+j)= mut;
                  *(ptf3+(i<<6)+j)=mut;
                  //*(ptf4+(i<<6)+j)= *(ptf1+(i<<6)+j);
                  //*(ptf3+(i<<6)+j)=*(ptf4+(i<<6)+j);
                  if(done)
                  {
                    f2++;
                    *(choose+i)=1;
                  }

              } // end of i<4096
              done=0;
            }// end of for j<64
            // LOG_printf(&trace, " stdp before %d",stdp);
             stdp=stdp>>18;//stdp=stdp=std/64
```

```
          xcmp=((1.3)*((float)stdp))/total_nets;
         //LOG_printf(&trace, " stdp %d",stdp);
      }// end of if net==0
 else
     {
       LOG_printf(&trace, " IN ELSE....net!=0");
       for(i=0;i<4096;i++)
       {
         err=0;
         for(j=0;j<64;j++)
         {
           ffr=(float)((*(ptf4+(i<<6)+j)));
        //   LOG_printf(&trace, " parttn2 %f",ffr);
         //  LOG_printf(&trace, " yy err %f",(*(yy+(i<<6)+j)));

           fx1= ffr-(*(yy+(i<<6)+j));
       //    LOG_printf(&trace, " fx1 %f",fx1);

            *(ptf4+(i<<6)+j)=(short)fx1;
            err=err+(fx1*fx1);
       //    LOG_printf(&trace, " CHECK xxx");
          // xx1= *(ptf4+(i<<6)+j);
          // *(ptf4+(i<<6)+j)=(((xx1<<7)-(*(yy+(i<<6)+j)))>>7);
          // err=err+((*(ptf4+(i<<6)+j))*(*(ptf4+(i<<6)+j)));
         }
      //  LOG_printf(&trace, " CHECKerr before %f",err);
          err=err/64;
        //LOG_printf(&trace, " CHECKerr before %f",err);
        //LOG_printf(&trace, " CHECK err");
       if(err>xcmp)
          {
            *(choose+i)=1;
            for (j=0;j<64;j++)
                *(ptf3+(f2<<6)+j)=*(ptf4+(i<<6)+j);

            *(nets_per_block+i)=(*(nets_per_block+i))+1;
            f2++;
          }

       else
          *(choose+i)=0;
       } // end of for i<4096
      //LOG_printf(&trace, " VALUE Of f2 INSIDE ELSE %d",f2);
       //LOG_printf(&trace, " ELSE DONE");
     } // end of else

estimste(yy,w,o,ptf3,f2);

for(j=0;j<64;j++)
{
  fround=(wlevels/2)*(*(w+j));
  round=(char)fround;
  fc=fround-round;

  if((fabs(fc)>=0.5))
    {
    if(round>=0)
    *(wc+(net<<6)+j)=round+1;
    else
```

```
        *(wc+(net<<6)+j)=round-1;
         }
      else

        *(wc+(net<<6)+j)=round;

   }

   for(j=0;j<f2;j++)
   {
     fround=((olevels>>1)*(*(o+j))/maxo);
     round=(char)fround;
     fc=fround-round;

     if(fabs(fc)>=0.5)
       {
       if(round>=0)
       *(oj+k)=round+1;
       else
       *(oj+k)=round-1;
       }
     else
       *(oj+k)=round;
   // LOG_printf(&trace, " Oj =%d",*(oj+k));
      k=k+1;

   }

   mxo[net]=maxo;
   compr=compr+f2;
   net++;
   LOG_printf(&trace, " END of while %d",net);



  } // end of while




//xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxx



for(i=0;i<4096;i++)
{
    for(j=0;j<64;j++)
   {
      ffr=(float)(*(ptf1+(i<<6)+j)-*(ptf4+(i<<6)+j));

      ffr= ffr+(*(yy+(i<<6)+j));
      *(ptf1+(i<<6)+j)=(short)ffr;

    }

   }
```

```
    ins=(unsigned char (*)[512][512])(&image1[0]);
    mean_array=(unsigned char*)px;
    pxr=(short (*)[4096][64])pf1;

    ffx=0;

   for(i=0;j<512;i=i+8)

      for(j=0;j<512;j=j+8)
       {

          iz=(i>>3);
          jz=(j>>3);

        for(kx=0;kx<8;kx++)
         for(mx=0;mx<8;mx++)

              (*ins)[i+kx][j+mx] = (*pxr)[ffx][kx+(mx<<3)]+(*mean_array);
              ffx++;
         mean_array++;
       }
```

```
//xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxx
      LOG_printf(&trace, " HIIIIII");
        LOG_printf(&trace, " HIIIIII");

}// end of main

// Finding Mean
unsigned char mean(unsigned char *pt)
{


        int k;
        unsigned char*im1=pt;
        unsigned  m1,m2,m3,m4,m5,m6,m7,m8,m9,m10,rowmean=0,mt,r1,r2;
        for(k=0;k<8;k++)
         {
           m1=_hi(_amemd8(im1));
           m2=_lo(_amemd8(im1));

           m3= _extu(m1,24,24);
           m4= _extu(m1,16,24);
           m5= _extu(m1,8,24);
           m6= _extu(m1,0,24);

           m7= _extu(m2,24,24);
           m8= _extu(m2,16,24);
           m9= _extu(m2,8,24);
           m10= _extu(m2,0,24);
```

```
            r1=m3+m4+m5+m6;
            r2=m7+m8+m9+m10;

            rowmean=rowmean+r1+r2;
            im1=im1+512;
          }
           mt=rowmean>>6;
           rowmean=0;

        return (unsigned char)mt;
} // End Of Finding Mean

unsigned char block1(unsigned char *current_block, int x,int y,int i, int j)
{
  int d;
  unsigned char mx,c1,c2,c3,c4,x1,x2,x3,x4;
  //LOG_printf(&trace, "Calling Block 1:");

          if(x==0)
               {
                 if(y==0)// Block x =0,y=0 & point i<4,j<4
                 {
                   mx=*current_block;
                 }
                 else // Block x = 0 ,y (1 to 63) & point i<4,j<4
                  {
                   c1=*(current_block-1);
                   c2=*current_block;
                   x1=(unsigned char)((4-i)*(4-j) + (4+i)*(4-j));
                   x2=(unsigned char)((4-i)*(4+j) + (4+i)*(4+j));
                   d=(c1*x1)+(c2*x2);
                   mx=(unsigned char)(d>>6);

                  }
               } // end of if x==0
               else // x >0 (1to 63)
               {
                if(y==0)// x= 1to 63, y=0,  point i<4,j<4
                 {
                  c1=*(current_block-64);
                  c2=*current_block;
                  x1=(unsigned char)((4-i)*(4-j) + (4-i)*(4+j));
                  x2=(unsigned char)((4+i)*(4-j) + (4+i)*(4+j));
                  d=(((c1*x1)+(c2*x2))>>6);
                  mx=(unsigned char)d;

                 }
                 else // x= 1to 63, y=1 to 63,  point i<4,j<4
                  {
                    c1=*(current_block-65);
                    c2=*(current_block-64);
                    c3=*(current_block-1);
                    c4=*(current_block);
                    x1=(unsigned char)((4-i)*(4-j));
                    x2=(unsigned char)((4-i)*(4+j));
                    x3=(unsigned char)((4+i)*(4-j));
                    x4=(unsigned char)((4+i)*(4+j));
                    d=(c1*x1)+(c2*x2)+(c3*x3)+(c4*x4);
```

```
                    mx=(unsigned char)(d>>6);
                  }

               }
           LOG_printf(&trace, "mean1 %d:",mx);
            return mx;

}

unsigned char block2(unsigned char *current_block, int x,int y,int i, int j)
{
  int d;
  unsigned char mx,c1,c2,c3,c4,x1,x2,x3,x4;
   //LOG_printf(&trace, "Calling Block 2:");
            if(x==0) //
                 {
                   if(y==63)// Block x=0,y=63 point i=<4,j>=4
                    {
                     mx=*current_block;
                    }
                    else // Block x=0,y=0 to 62 point i=<4,j>=4
                    {
                     c1=*(current_block);
                     c2=*(current_block+1);
                     x1=(unsigned char)((4-i)*(12-j) + (4+i)*(12-j));
                     x2=(unsigned char)((4-i)*(j-4) + (4+i)*(j-4));
                     d=(c1*x1)+(c2*x2);
                     mx=(unsigned char)(d>>6);
                    }
                 }
                 else // x= 1to 63
                  {
                   if(y==63)// Block x=1 to 63,y=63 point i=<4,j>=4
                    {
                     c1=*(current_block-64);
                     c2=*(current_block);
                     x1=(unsigned char)((4-i)*(12-j) + (4-i)*(j-4));
                     x2=(unsigned char)((4+i)*(12-j) + (4+i)*(j-4));
                     d=(c1*x1)+(c2*x2);
                     mx=(unsigned char)(d>>6);
                    }
                    else // Block x=1 to 63 ,y=0 to 62 point i=<4,j>=4
                    {
                     c1=*(current_block-64);
                     c2=*(current_block-63);
                     c3=*(current_block);
                     c4=*(current_block+1);
                     x1=(unsigned char)((4-i)*(12-j));
                     x2=(unsigned char)((4-i)*(j-4));
                     x3=(unsigned char)((4+i)*(12-j));
                     x4=(unsigned char)((4+i)*(j-4));
                     d=((c1*x1)+(c2*x2)+(c3*x3)+(c4*x4))>>6;
                     mx=(unsigned char)(d);
                    }
                  }

           LOG_printf(&trace, "mean2 %d:",mx);
           return mx;
```

```
}
unsigned char block3(unsigned char *current_block, int x,int y,int i, int j)
{
  int d;
  unsigned char mx,c1,c2,c3,c4,x1,x2,x3,x4;
  //LOG_printf(&trace, "Calling Block 3:");
           if(x==63)
               {
                 if(y==0)// Block x = 63, y== 0 point i>=4 & j<4
                  {
                   mx=*current_block;

                  }
                  else// Block x=63,y = 1to63 point i>=4 & j<4
                  {
                   c1=*(current_block-1);
                   c2=*current_block;
                   x1=(unsigned char)((12-i)*(4-j) + (i-4)*(4-j));
                   x2=(unsigned char)((12-i)*(4+j) + (i-4)*(4+j));
                   d=(c1*x1)+(c2*x2);
                   mx=(unsigned char)(d>>6);
                  }
               }
               else // x = 0 to 62
               {
                  if(y==0)// block x = 0 to 62,y = 0 point i>=4 & j<4
                  {
                   c1=*(current_block);
                   c2=*(current_block +64);
                   x1=(unsigned char)((12-i)*(4-j) + (12-i)*(4+j));
                   x2=(unsigned char)((i-4)*(4-j) + (i-4)*(4+j));
                   d=(c1*x1)+(c2*x2);
                   mx=(unsigned char)(d>>6);
                  }
                  else// block x = 0 to 62,y 1 to 63 point i>=4 & j<4
                  {
                    c1=*(current_block-1);
                    c2=*(current_block);
                    c3=*(current_block+63);
                    c4=*(current_block+64);
                    x1=(unsigned char)((12-i)*(4-j));
                    x2=(unsigned char)((12-i)*(j+4));
                    x3=(unsigned char)((i-4)*(4-j));
                    x4=(unsigned char)((i-4)*(4+j));
                    d=(c1*x1)+(c2*x2)+(c3*x3)+(c4*x4);
                    mx=(unsigned char)(d>>6);
                  }
               }
           LOG_printf(&trace, "mean3 %d:",mx);
           return mx;
}
unsigned char block4(unsigned char *current_block, int x,int y,int i, int j)
{
  int d;
  unsigned char mx,c1,c2,c3,c4,x1,x2,x3,x4;
// LOG_printf(&trace, "Calling Block 4:");
            if(x==63)
               {
                 if(y==63) // Block x=63,y=63 point i>=4,j>=4
```

```
                {
                 mx=*current_block;
                }
                else   // Block x=63,y= 0 to 62 point i>=4,j>=4
                {
                 c1=*(current_block);
                 c2=*(current_block+1);
                 x1=(unsigned char)((12-i)*(12-j) + (i-4)*(12-j));
                 x2=(unsigned char)((12-i)*(j-4) + (i-4)*(j-4));
                 d=(c1*x1)+(c2*x2);
                 mx=(unsigned char)(d>>6);
                }
              }
              else
              {
               if(y==63) // Block x=0 to 62,y=63 point i>=4,j>=4
                {
                 c1=*(current_block);
                 c2=*(current_block+64);
                 x1=(unsigned char)((12-i)*(12-j) + (12-i)*(j-4));
                 x2=(unsigned char)((i-4)*(12-j) + (i-4)*(j-4));
                 d=(c1*x1)+(c2*x2);
                 mx=(unsigned char)(d>>6);
                }
                else // Block x=0 to 62 ,y=0 to 62 point i>=4,j>=4
                {
                  c1=*(current_block);
                  c2=*(current_block+1);
                  c3=*(current_block+64);
                  c4=*(current_block+65);
                  x1=(unsigned char)((12-i)*(12-j));
                  x2=(unsigned char)((12-i)*(j-4));
                  x3=(unsigned char)((i-4)*(12-j));
                  x4=(unsigned char)((i-4)*(j-4));
                  d=(c1*x1)+(c2*x2)+(c3*x3)+(c4*x4);
                  mx=(unsigned char)(d>>6);
                }
              }
            LOG_printf(&trace, "mean4 %d:",mx);
             return mx;
}


void s_array(int mean_diff,unsigned char *s,unsigned char *image,short
*frame)
{
        int
k,m1,m2,m3,m4,m5,m6,m7,m8,m9,m10,hi,lo,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10;
        unsigned char*im1=s,*img=image,ch;
        for(k=0;k<8;k++)
         {
           m1=_hi(_amemd8(im1));
           m2=_lo(_amemd8(im1));

           x1=_hi(_amemd8(img));
           x2=_lo(_amemd8(img));

           m3= (int)_extu(m1,24,24) + mean_diff;
           m4= (int)_extu(m1,16,24)+ mean_diff;
```

```
            m5= (int)_extu(m1,8,24)+ mean_diff;
            m6= (int)_extu(m1,0,24)+ mean_diff;

            x3= (int)_extu(x1,24,24) ;
            x4= (int)_extu(x1,16,24);
            x5= (int)_extu(x1,8,24);
            x6= (int)_extu(x1,0,24);

            m7= (int)_extu(m2,24,24)+ mean_diff;
            m8= (int)_extu(m2,16,24)+ mean_diff;
            m9= (int)_extu(m2,8,24)+ mean_diff;
            m10= (int)_extu(m2,0,24)+ mean_diff;

              x7= (int)_extu(x2,24,24);
            x8= (int)_extu(x2,16,24);
            x9= (int)_extu(x2,8,24);
            x10=(int) _extu(x2,0,24);

            x7=x7- m7;
            x8=x8-m8;
            x9=x9-m9;
            x10=x10-m10;

            x3=x3-m3;
            x4=x4-m4;
            x5=x5-m5;
            x6=x6-m6;

            *(frame+k)=(short)x7;
            *(frame+8+k)=(short)x8;
            *(frame+16+k)=(short)x9;
            *(frame+24+k)=(short)x10;

            *(frame+32+k)=(short)x3;
            *(frame+40+k)=(short)x4;
            *(frame+48+k)=(short)x5;
            *(frame+56+k)=(short)x6;

            hi=m3 | (m4<<8) | (m5<<16)|(m6<<24);
            lo=m7 | (m8<<8) | (m9<<16) |(m10<<24);

            _amemd8(im1)=_itod(hi,lo);
            im1=im1+512;
            img=img+512;
          }

}

void estimste(float *yy,float *w,float *o,short *ptf3, int f2)
  {
    int i,j,it;
    float sw,wp,so,op,maxw,sig,squ,f1,f3,f4,f5,fm;
    short chx,sf1,sf;

    /*for(i=0;i<4096;i++)
     {
       for(j=0;j<64;j++)
        chx=*(ptf3+(i<<6)+j);
     }*/
```

```
    for (i=0;i<64;i++)
       *(w+i)=i/(float)64;

    for(it=0;it<10;it++)
    {
     sw=0;
     for(j=0;j<64;j++)
          sw=sw+((*(w+j))*(*(w+j)));

     for (i=0;i<4096;i++)
        {
          wp=0;
          for (j=0;j<64;j++)
              wp=wp+((*(w+j))*((float)(*(ptf3+(i<<6)+j))));

          *(o+i)=(wp)/sw;
          sig=*(o+i);
         }

      so=0;

      for (i=0;i<4096;i++)
       {
        // sig=*(o+i);
        // squ=sig*sig;
        // so=so+squ;

        so=so+((*(o+i))*(*(o+i)));
       }


      for (j=0;j<64;j++)
         {
           op=0;
            for (i=0;i<4096;i++)
              op=op+((*(o+i))*(*(ptf3+(i<<6)+j))); // op is multiple of 128
only

            *(w+j)=(op)/so; // w is multiple of 128
          }
      } // end of it =10

      maxw = -1000000;
      for(j=0;j<64;j++)
       if (fabs(*(w+j))>maxw)
          maxw=fabs(*(w+j));

      for (j=0;j<64;j++)
         *(w+j)=(*(w+j))/maxw; // w is multiple of 128

      for(i=0;i<4096;i++)
         *(o+i)=((*(o+i))*maxw); //

      maxo=-1000000;
    for (i=0;i<4096;i++)
      if (fabs(*(o+i))>maxo)
          maxo=fabs(*(o+i));
```

```
   //   if(maxo<0.00001) maxo=0.1;
    LOG_printf(&trace, " In ESTIMATE MAX %f",maxo);
      so=0;
      for (i=0;i<4096;i++)
        so=so+((*(o+i))*(*(o+i)));

      for (j=0;j<64;j++)
        {
         op=0;
         for (i=0;i<4096;i++)
           op=op+((*(o+i))*((*(ptf3+(i<<6)+j))));

         *(w+j)=(op)/so;

         /* if (fabs(*(w+j))>1)
            *(w+j)=(0.99*(*(w+j)))/fabs(*(w+j));  */
        }
      LOG_printf(&trace, "w done :");
      f4=(maxo/(olevels>>1));
     for (i=0;i<4096;i++)
         {
           f1=(((*(o+i))*(olevels>>1))/maxo);
           sf1=(short)f1;
           f3=f1-sf1;
           if(fabs(f3)>=0.5)
             {
              if (sf1>=0)
                sf1++;
              else
                 sf1--;
             }
             fm=(maxo/(olevels>>1))*sf1;
           for(j=0;j<64;j++)
            {
              f5=(fm)*(*(w+j));
             // sf=(short)f5;
              *(yy+(i<<6)+j)=f5;

              // *(yy+(i<<6)+j)=((int)((*(w+j))*(*(o+i))))<<7;  // yy is
multiple of 128
            }
          }

     LOG_printf(&trace, "estimate done :");

  }
```

DECOMPRESSION OF AN IMAGE:

```
#include <std.h>
#include <log.h>
#include <mem.h>
#include<math.h>

#include "vikascfg.h"
#include "ac.c"
#include  "ac.h"

#define X_SIZE 512
#define Y_SIZE 512
#define N_PIXELS X_SIZE*Y_SIZE
#define ADAPT 1


#pragma DATA_SECTION(image1, "SDRAM");
#pragma DATA_ALIGN (image1, 8);
unsigned char image1[N_PIXELS];
unsigned char *image=&image1[0];
float maxo;
int olevels=126;


unsigned char mean(unsigned char*);
unsigned char block1(unsigned char*,int,int,int,int);
unsigned char block2(unsigned char*,int,int,int,int);
unsigned char block3(unsigned char*,int,int,int,int);
unsigned char block4(unsigned char*,int,int,int,int);
void s_array(int, unsigned char*,unsigned char*,short*);
void estimste(float*,float*,float*,short*,int);

extern Int SEG0;

void main()
{
    Ptr
px,ps,pf1,pf4,pf3,pnets_per_block,poj,pojj,pchoose,py,pytemp,pw,po,potemp,pxte
st_short,pxtest_unsigned,pwc,
pmem,pc1,pc2,ppr,ppr1,pimagex;
    short
*ptf1,*frame,*ptf4,*ptf3,*ytemp,*otemp,oc,*xtest_short,xxd,xx1,mut;
    int
wtest,i,j,x,y,rx,cx,shift1=0,shift2=0,mean_diff,smean,total_nets=3,f1=4096,net
,f2,k,done,NSYM2,NSYM3,*pac1,
*pac2;
    int
*choose,stdp,meanp,mxo[5],compr,f3,mb,nbytes=0,numofcoef,pp,wlevels=256,p,coun
t,FLAGE,n,sym,m;
    unsigned char
*image=&image1[0],mn,*mean_array,*test,*current_block,*s,*s1,*sx,mx,*xtest_uns
igned;
    char *ojj,*oj,*nets_per_block,*wc,round,*mem,*enit;
    float *w,*o,fround,fc,*yy,fx1,fx2,err,xcmp,ffr;
    unsigned r1=0,r2=0;
    double dx;
   // int *pr11,*pr12;
   // int pr[60][34],pr1[60][34];
```

```
 int (*pr)[60][128],(*pr1)[60][128],(*rpr),(*rpr1),*zrx1,*zrx2,vi,sys,z;
  int (*imagex)[512][512];
unsigned char (*ins)[512][512];
float fsym,ffw;
 ac_encoder ace;
 ac_model acm[6];
 ac_decoder acd;
 px= MEM_alloc(SEG0, 4096, 8);// for mean
 ps = MEM_alloc(SEG0, 512*512, 8);
 pf1=MEM_alloc(SEG0, 2*512*512, 8);

 pnets_per_block=MEM_alloc(SEG0, 4096, 8);
 poj=MEM_alloc(SEG0, (512*512)+64, 8);
 pchoose=MEM_alloc(SEG0, 4*4104, 8);
 pw=MEM_alloc(SEG0, sizeof(float)*64, 8);
 po=MEM_alloc(SEG0, sizeof(float)*4096, 8);
 py=MEM_alloc(SEG0, sizeof(float)*512*512, 8);
// pytemp=MEM_alloc(SEG0, 2*512*512, 8);
 pwc= MEM_alloc(SEG0, 4096, 8);
   pc1=MEM_alloc(SEG0, 128*sizeof(int), 8);
  pc2=MEM_alloc(SEG0, 129*sizeof(int), 8);
  pimagex=MEM_alloc(SEG0, 512*512*sizeof(int), 8);



 mean_array=(unsigned char*)px;
// LOG_printf(&trace, " Load");
  LOG_printf(&trace, " Loaeddddd");

  for(i=0;i<64;i++)
    {
     for(j=0;j<64;j++)
      {
       *mean_array++=mean(image);
       image=image+8;
      }
       image=image+3584;
    }
 LOG_printf(&trace, "Finding Mean done");
  mean_array=(unsigned char*)px;
   s=(unsigned char*)ps;

   for(rx=0;rx<512;rx++)
{
  //s=sx;
  for(cx=0;cx<512;cx++)
  {
   x=rx>>3;
   y=cx>>3;
   i=rx-(x<<3);
   j=cx-(y<<3);
   current_block=mean_array+((x<<6)+y);
   if((i<4) & (j<4))
     mx=block1(current_block,x,y,i,j);
   if((i<4) & (j>=4))
     mx=block2(current_block,x,y,i,j);
   if((i>=4) & (j<4))
     mx=block3(current_block,x,y,i,j);
   if((i>=4) & (j>=4))
```

```
     mx=block4(current_block,x,y,i,j);
    *s++=mx;

  }

}

 LOG_printf(&trace, "Blocks done");

 // Smoothing surface
s=(unsigned char*)ps;
mean_array=(unsigned char*)px;
frame=(short *)pf1;
image=&image1[0];
for(i=0;i<64;i++)
    {
     for(j=0;j<64;j++)
      {
       smean=(int)mean(s);
       mean_diff=(*mean_array++) - smean;
       s_array(mean_diff,s,image,frame);
       //LOG_printf(&trace, "mean1 %d:",mean_diff);
       s=s+8;
       image=image+8;
       frame=frame+64;
      }
       s=s+3584;
      image=image+3584;
    }

   LOG_printf(&trace, " Smooth done");
       LOG_printf(&trace, " Smooth done");
// MEM_free(SEG0, ps, 512*512);
    //s=(unsigned char*)ps;
    //frame=(short *)pf1;
  nets_per_block=( char *)pnets_per_block;
  oj=( char *) poj;

   for (i=0;i<262208;i++)// use 8 store
    {
     if (i<4096)
        *nets_per_block++=0;
        *oj++=0;
     }

  nets_per_block=( char *)pnets_per_block;
  oj=(char*) poj; //
  //ojj=( char *) pojj;
  pf4=MEM_alloc(SEG0, 2*512*512, 8);
  pf3=MEM_alloc(SEG0, 2*512*512, 8);
  ptf1=(short *)pf1;
  ptf4=(short *)pf4;
  ptf3=(short *)pf3;
  choose=(int *)pchoose;
  w =(float *)pw;
  o =(float *)po;
  net=0;
  k=0;
  f2=1000;
```

```
     compr=4096;
     wtest=1;
     done=1;
     yy=(float *)py;
     //ytemp =(short *)pytemp;
     wc=(char *)pwc;

  while((compr<total_nets*f1)&&(f2>100))
// while(wtest<3)
    {
       wtest++;
       f2=0;
       LOG_printf(&trace, " in while");
      if (net==0)
         {
           stdp=0;
           for(j=0;j<64;j++)
            {
              meanp=0;
              for(i=0;i<4096;i++)
               {
                 meanp=meanp+*(ptf1+(i<<6)+j);
               }
            //   LOG_printf(&trace, " Meanp before %d",meanp);
               meanp=meanp>>5;// meanp=(meanp)/4096
            //   LOG_printf(&trace, " Meanp After %d",meanp);
               for (i=0;i<4096;i++)
               {
                  mut=(*(ptf1+(i<<6)+j));
                  xxd=((mut<<7)-meanp)>>7;
                  stdp=stdp+(xxd*xxd);
                  *(ptf4+(i<<6)+j)= mut;
                  *(ptf3+(i<<6)+j)=mut;
                  //*(ptf4+(i<<6)+j)= *(ptf1+(i<<6)+j);
                  //*(ptf3+(i<<6)+j)=*(ptf4+(i<<6)+j);
                  if(done)
                  {
                    f2++;
                    *(choose+i)=1;
                  }

              } // end of i<4096
              done=0;
            }// end of for j<64
            // LOG_printf(&trace, " stdp before %d",stdp);
             stdp=stdp>>18;//stdp=stdp=std/64
             xcmp=((1.3)*((float)stdp))/total_nets;
            //LOG_printf(&trace, " stdp %d",stdp);
         }// end of if net==0
    else
        {
          LOG_printf(&trace, " IN ELSE....net!=0");
          for(i=0;i<4096;i++)
          {
            err=0;
            for(j=0;j<64;j++)
            {
              ffr=(float)((*(ptf4+(i<<6)+j)));
           //   LOG_printf(&trace, " parttn2 %f",ffr);
```

```
          //  LOG_printf(&trace, " yy err %f",(*(yy+(i<<6)+j)));

             fx1= ffr-(*(yy+(i<<6)+j));
        //    LOG_printf(&trace, " fx1 %f",fx1);

              *(ptf4+(i<<6)+j)=(short)fx1;
              err=err+(fx1*fx1);
        //    LOG_printf(&trace, " CHECK xxx");
            // xx1= *(ptf4+(i<<6)+j);
            // *(ptf4+(i<<6)+j)=(((xx1<<7)-(*(yy+(i<<6)+j)))>>7);
            // err=err+((*(ptf4+(i<<6)+j))*(*(ptf4+(i<<6)+j)));
           }
       //  LOG_printf(&trace, " CHECKerr before %f",err);
            err=err/64;
         //LOG_printf(&trace, " CHECKerr before %f",err);
         //LOG_printf(&trace, " CHECK err");
        if(err>xcmp)
           {
             *(choose+i)=1;
             for (j=0;j<64;j++)
                 *(ptf3+(f2<<6)+j)=*(ptf4+(i<<6)+j);

             *(nets_per_block+i)=(*(nets_per_block+i))+1;
             f2++;
           }

        else
           *(choose+i)=0;
        } // end of for i<4096
       //LOG_printf(&trace, " VALUE Of f2 INSIDE ELSE %d",f2);
        //LOG_printf(&trace, " ELSE DONE");
     } // end of else

estimste(yy,w,o,ptf3,f2);

for(j=0;j<64;j++)
{
  fround=(wlevels/2)*(*(w+j));
  round=(char)fround;
  fc=fround-round;

  if((fabs(fc)>=0.5))
    {
    if(round>=0)
    *(wc+(net<<6)+j)=round+1;
    else
    *(wc+(net<<6)+j)=round-1;
     }
  else

    *(wc+(net<<6)+j)=round;

}

for(j=0;j<f2;j++)
{
  fround=((olevels>>1)*(*(o+j))/maxo);
  round=(char)fround;
  fc=fround-round;
```

```
  if(fabs(fc)>=0.5)
    {
    if(round>=0)
    *(oj+k)=round+1;
    else
    *(oj+k)=round-1;
    }
  else
    *(oj+k)=round;
// LOG_printf(&trace, " Oj =%d",*(oj+k));
   k=k+1;

}

mxo[net]=maxo;
compr=compr+f2;
net++;
LOG_printf(&trace, " END of while %d",net);



} // end of while

MEM_free(SEG0, ptf3,2*512*512);
  LOG_printf(&trace, "Modal Init Donek %d",k);
   numofcoef=k;
   mb=net;
    LOG_printf(&trace, " END of while :%d",mb);
   pojj= MEM_alloc(SEG0, 512*512, 8);
   //nbytes=nbytes+mb*64;
   //nbytes=nbytes+mb*4+18;
   ojj=(char *)pojj;
   k=0;
   for (j=0;j<mb;j++)
   {
     p=0;
      for (i=0;i<4096;i++)
      {
        if (*(nets_per_block+i)>j)
        {
          *(ojj+j+p)=*(oj+k);
          k++;
        }
        p=p+(*(nets_per_block+i))+1;
      }
   }

   p=0;
   for (i=0;i<4096;i++)
   {
     ojj[p+nets_per_block[i]]=(olevels>>1)+1;
     p=p+nets_per_block[i]+1;
   }

   i=numofcoef+4095;
     printf("i is: %d\n",i);
   LOG_printf(&trace, "value of i:%d ",i);
while(i>0)
```

```
    {
      if (ojj[i]==((olevels>>1)+1))
      {
        i=i-1;
        j=i-3;
        if(j<0)
        j=0;
        while (ojj[i]==0)
        {
          ojj[i]=-100;
          i=i-1;
        }
      }
      else i=i-1;
    }
   printf("i is: %d\n",i);
  MEM_free(SEG0,pnets_per_block,4096);
  MEM_free(SEG0,poj,(512*512)+64);
  MEM_free(SEG0,pw,sizeof(float)*64);
  //MEM_free(SEG0,po,sizeof(float)*4096);
  //MEM_free(SEG0,pwc,sizeof(char)*4096);


   //pr = MEM_alloc(SEG0, 60*34, 8);
   //pr1 = MEM_alloc(SEG0, 60*34, 8);

  ppr=MEM_alloc(SEG0, sizeof(int)*60*128, 8); // NEWWWWWWWWW
  ppr1=MEM_alloc(SEG0,sizeof(int)*60*128, 8); // NEWWWWWWWW
  pr=(int(*)[60][128])ppr;// NEWWWWWWWWWWWWW
  pr1=(int(*)[60][128])ppr1;// NEWWWWWWWWWWWW
  rpr = (int(*))ppr; // NEWWWWW
  rpr1 = (int(*))ppr1; // NEWWWWW

  //pr11=(int *)pr;
  // pr12=(int *)pr1;

    NSYM2=olevels+2;
    NSYM3=NSYM2+1;
     pmem=MEM_alloc(SEG0, 512*512, 8);// NEEEEED For DECODERRRRR
    enit=(char *)pmem;
  ac_encoder_init(&ace,enit);
   LOG_printf(&trace, "Encoder Init Done ");




/*   for (i=0;i<60;i++)
       for(j=0;j<NSYM2;j++)
        {
          *(pr11+(i<<7)+j)=1;
          printf("pre :\n%d",*(pr11+j+i));
          *(pr12+(i<<7)+j)=1;
          //printf("\n%d",*(pr12+j+i));

        }*/



     for (i=0;i<128;i++)
```

```
      for (j=0;j<60;j++)
       {
        (*pr)[j][i]=1; // NEWWWWWWWWW
        //printf("pre :\n%d",(*pr)[i][j]);
          (*pr1)[j][i]=1; //NEWWWWWWWWW
       }
      LOG_printf(&trace, "Encoder2 Init Done ");
      count=0;
     FLAGE=0;
            pc1=MEM_alloc(SEG0, 128*sizeof(int), 8); //NEWWWW
  pc2=MEM_alloc(SEG0, 129*sizeof(int), 8);// NEWWWW
  pac1=(int *)pc1; // NEWWWW
  pac2=(int *)pc2; // NEWWWWW

  vi=(numofcoef);
  printf("vi is: %d\n,no : %d\n",vi,numofcoef);
 for(i=0;i<vi;i++)
    {

      if(i>0){if((ojj[i-1]==olevels/2+1)||(count>mb)) count=0;}
      if(ojj[i]!=-100) count=count+1;
     //LOG_printf(&trace, "Modal Init Done %d",count);

      if((ojj[i]!=-100)&&(count<mb))
      {


         if(FLAGE==0)
          {

              zrx1=rpr+count;
               LOG_printf(&trace, "Modal Initisl Done ");
            ac_model_init(&acm[count],NSYM2,zrx1,ADAPT,pac1,pac2);
               LOG_printf(&trace, "Modal Init Done ");
              sys=(ojj[i]+(olevels>>1));
               ac_encode_symbol(&ace,&acm[count],sys);
              // LOG_printf(&trace, "Symbol done:%d ",i);
              // printf("ccc:%d,vvv:%d\n",i,ojj[i]);

          }
       else
           {
               zrx2=rpr1+count;

               ac_model_init(&acm[count],NSYM2,zrx2,ADAPT,pac1,pac2);
               LOG_printf(&trace, "Modal Init Done ");
               ac_encode_symbol(&ace,&acm[count],ojj[i]+(olevels>>1));
               LOG_printf(&trace, "Symbol done1 :%d ",i);

           }

         (*pr)[count][ojj[i]+(olevels>>1)] =
(*pr)[count][ojj[i]+(olevels>>1)]+5;
          if((*pr)[count][ojj[i]+(olevels>>1)] > 500)
            for(j=0;j<NSYM2;j++)
              (*pr)[count][j]=(2*(*pr)[count][j])/3+1;

             if(ojj[i]!=(olevels>>1)+1)
              {
```

```
(*pr1)[count][ojj[i]+(olevels>>1)]=(*pr1)[count][ojj[i]+(olevels>>1)]+15;

                 if((*pr1)[count][ojj[i]+(olevels>>1)] >500)
                  for(j=0;j<NSYM2;j++)
                    (*pr1)[count][j]=(2*(*pr1)[count][j])/3+1;
                }
                if(ojj[i]==0) FLAGE=1;
                 else FLAGE =0;
        }
      }// end of for(i=0;i<numofcoef+4096;i++)

  ac_decoder_init(&acd,(char *)pmem);

    imagex=(int (*)[512][512])pimagex;
    ins=(unsigned char (*)[512][512])ps;


    for (i=0;i<512;i++)
       for (j=0;j<512;j++)
       {
         (*imagex)[i][j]=0;
       }

       for (i=0;i<60;i++)
        for (j=0;j<128;j++)
         {
          (*pr)[i][j]=1; // NEWWWWWWWWW
           //printf("pre :\n%d",(*pr)[i][j]);
            (*pr1)[i][j]=1; //NEWWWWWWWWW
         }

         FLAGE=0;

         for (i=0;i<64;i++)
           for (n=0;n<64;n++)
           {
             m=0;
             sym=0;
             count=0;

             while((sym!=olevels+1) && (count<mb))
             {
               count=count+1;
               if (FLAGE==0)
               {
                zrx1=rpr+count;
                ac_model_init(&acm[count],NSYM2,zrx1,ADAPT,pac1,pac2);
                sym=ac_decode_symbol(&acd,&acm[count]);
               }
               else
               {
                 zrx2=rpr1+count;
                ac_model_init(&acm[count],NSYM2,zrx2,ADAPT,pac1,pac2);
                sym=ac_decode_symbol(&acd,&acm[count]);

               }
                (*pr)[count][sym] = (*pr)[count][sym]+5;
                    if((*pr)[count][sym] > 500)
```

```
                    for(j=0;j<NSYM2;j++)
                    (*pr)[count][j]=(2*(*pr)[count][j])/3+1;

                  if(sym!=olevels+1)
                {

                  (*pr1)[count][sym]=(*pr1)[count][sym]+15;

                 if((*pr1)[count][sym] >500)
                  for(j=0;j<NSYM2;j++)
                    (*pr1)[count][j]=(2*(*pr1)[count][j])/3+1;
                }

                 if (sym==(olevels>>1))
                      FLAGE=1;
                 else
                      FLAGE=0;

                 if (sym!=olevels+1)
                   for (j=0;j<8;j++)
                      for (k=0;k<8;k++)
                      {
                        fsym=((float)sym-(olevels/2))*mxo[m]/((olevels/2));
                        ffw=((float)*(wc+(m<<6)+k*8+j))/128;

(*imagex)[i*8+j][n*8+k]=(float)(*imagex)[i*8+j][n*8+k]+(fsym*ffw);

//(*imagex)[i*8+j][n*8+k]=(*imagex)[i*8+j][n*8+k]+(float)(sym-
(olevels/2))*(*(wc+(m<<6)+k*8+j))*mxo[m]/((olevels/2+1));
                      }
                      m=m+1;
             }

        for (j=0;j<8;j++)
             for (k=0;k<8;k++)
             {

(*ins)[i*8+j][n*8+k]=(int)(*ins)[i*8+j][n*8+k]+(*imagex)[i*8+j][n*8+k];
             }
           }



      LOG_printf(&trace, " HIIIIII");
        LOG_printf(&trace, " HIIIIII");

}// end of main

// Finding Mean
unsigned char mean(unsigned char *pt)
{


        int k;
        unsigned char*im1=pt;
        unsigned  m1,m2,m3,m4,m5,m6,m7,m8,m9,m10,rowmean=0,mt,r1,r2;
        for(k=0;k<8;k++)
         {
           m1=_hi(_amemd8(im1));
```

```
            m2=_lo(_amemd8(im1));

            m3= _extu(m1,24,24);
            m4= _extu(m1,16,24);
            m5= _extu(m1,8,24);
            m6= _extu(m1,0,24);

            m7= _extu(m2,24,24);
            m8= _extu(m2,16,24);
            m9= _extu(m2,8,24);
            m10= _extu(m2,0,24);

            r1=m3+m4+m5+m6;
            r2=m7+m8+m9+m10;

            rowmean=rowmean+r1+r2;
            im1=im1+512;
          }
           mt=rowmean>>6;
           rowmean=0;

        return (unsigned char)mt;
} // End Of Finding Mean

unsigned char block1(unsigned char *current_block, int x,int y,int i, int j)
{
  int d;
  unsigned char mx,c1,c2,c3,c4,x1,x2,x3,x4;
  //LOG_printf(&trace, "Calling Block 1:");

          if(x==0)
               {
                 if(y==0)// Block x =0,y=0 & point i<4,j<4
                 {
                   mx=*current_block;
                 }
                 else // Block x = 0 ,y (1 to 63) & point i<4,j<4
                  {
                   c1=*(current_block-1);
                   c2=*current_block;
                   x1=(unsigned char)((4-i)*(4-j) + (4+i)*(4-j));
                   x2=(unsigned char)((4-i)*(4+j) + (4+i)*(4+j));
                   d=(c1*x1)+(c2*x2);
                   mx=(unsigned char)(d>>6);

                  }
               } // end of if x==0
               else // x >0 (1to 63)
               {
                if(y==0)// x= 1to 63, y=0,  point i<4,j<4
                 {
                  c1=*(current_block-64);
                  c2=*current_block;
                  x1=(unsigned char)((4-i)*(4-j) + (4-i)*(4+j));
                  x2=(unsigned char)((4+i)*(4-j) + (4+i)*(4+j));
                  d=(((c1*x1)+(c2*x2))>>6);
                  mx=(unsigned char)d;

                 }
```

```
                else // x= 1to 63, y=1 to 63,  point i<4,j<4
                 {
                   c1=*(current_block-65);
                   c2=*(current_block-64);
                   c3=*(current_block-1);
                   c4=*(current_block);
                   x1=(unsigned char)((4-i)*(4-j));
                   x2=(unsigned char)((4-i)*(4+j));
                   x3=(unsigned char)((4+i)*(4-j));
                   x4=(unsigned char)((4+i)*(4+j));
                   d=(c1*x1)+(c2*x2)+(c3*x3)+(c4*x4);
                   mx=(unsigned char)(d>>6);
                 }

              }
          LOG_printf(&trace, "mean1 %d:",mx);
           return mx;

}

unsigned char block2(unsigned char *current_block, int x,int y,int i, int j)
{
  int d;
  unsigned char mx,c1,c2,c3,c4,x1,x2,x3,x4;
   //LOG_printf(&trace, "Calling Block 2:");
           if(x==0) //
                {
                  if(y==63)// Block x=0,y=63 point i=<4,j>=4
                   {
                    mx=*current_block;
                   }
                   else // Block x=0,y=0 to 62 point i=<4,j>=4
                   {
                    c1=*(current_block);
                    c2=*(current_block+1);
                    x1=(unsigned char)((4-i)*(12-j) + (4+i)*(12-j));
                    x2=(unsigned char)((4-i)*(j-4) + (4+i)*(j-4));
                    d=(c1*x1)+(c2*x2);
                    mx=(unsigned char)(d>>6);
                   }
                }
                else // x= 1to 63
                 {
                  if(y==63)// Block x=1 to 63,y=63 point i=<4,j>=4
                   {
                    c1=*(current_block-64);
                    c2=*(current_block);
                    x1=(unsigned char)((4-i)*(12-j) + (4-i)*(j-4));
                    x2=(unsigned char)((4+i)*(12-j) + (4+i)*(j-4));
                    d=(c1*x1)+(c2*x2);
                    mx=(unsigned char)(d>>6);
                   }
                   else // Block x=1 to 63 ,y=0 to 62 point i=<4,j>=4
                   {
                    c1=*(current_block-64);
                    c2=*(current_block-63);
                    c3=*(current_block);
                    c4=*(current_block+1);
                    x1=(unsigned char)((4-i)*(12-j));
```

```
                    x2=(unsigned char)((4-i)*(j-4));
                    x3=(unsigned char)((4+i)*(12-j));
                    x4=(unsigned char)((4+i)*(j-4));
                    d=((c1*x1)+(c2*x2)+(c3*x3)+(c4*x4))>>6;
                    mx=(unsigned char)(d);
                   }
                 }

          LOG_printf(&trace, "mean2 %d:",mx);
          return mx;

}
unsigned char block3(unsigned char *current_block, int x,int y,int i, int j)
{
  int d;
  unsigned char mx,c1,c2,c3,c4,x1,x2,x3,x4;
  //LOG_printf(&trace, "Calling Block 3:");
           if(x==63)
               {
                 if(y==0)// Block x = 63, y== 0 point i>=4 & j<4
                  {
                   mx=*current_block;

                  }
                  else// Block x=63,y = 1to63 point i>=4 & j<4
                  {
                   c1=*(current_block-1);
                   c2=*current_block;
                   x1=(unsigned char)((12-i)*(4-j) + (i-4)*(4-j));
                   x2=(unsigned char)((12-i)*(4+j) + (i-4)*(4+j));
                   d=(c1*x1)+(c2*x2);
                   mx=(unsigned char)(d>>6);
                  }
               }
               else // x = 0 to 62
               {
                  if(y==0)// block x = 0 to 62,y = 0 point i>=4 & j<4
                  {
                   c1=*(current_block);
                   c2=*(current_block +64);
                   x1=(unsigned char)((12-i)*(4-j) + (12-i)*(4+j));
                   x2=(unsigned char)((i-4)*(4-j) + (i-4)*(4+j));
                   d=(c1*x1)+(c2*x2);
                   mx=(unsigned char)(d>>6);
                  }
                  else// block x = 0 to 62,y 1 to 63 point i>=4 & j<4
                  {
                    c1=*(current_block-1);
                    c2=*(current_block);
                    c3=*(current_block+63);
                    c4=*(current_block+64);
                    x1=(unsigned char)((12-i)*(4-j));
                    x2=(unsigned char)((12-i)*(j+4));
                    x3=(unsigned char)((i-4)*(4-j));
                    x4=(unsigned char)((i-4)*(4+j));
                    d=(c1*x1)+(c2*x2)+(c3*x3)+(c4*x4);
                    mx=(unsigned char)(d>>6);
                  }
               }
```

```
           LOG_printf(&trace, "mean3 %d:",mx);
           return mx;
}
unsigned char block4(unsigned char *current_block, int x,int y,int i, int j)
{
  int d;
  unsigned char mx,c1,c2,c3,c4,x1,x2,x3,x4;
// LOG_printf(&trace, "Calling Block 4:");
            if(x==63)
               {
                 if(y==63) // Block x=63,y=63 point i>=4,j>=4
                  {
                   mx=*current_block;
                  }
                  else   // Block x=63,y= 0 to 62 point i>=4,j>=4
                  {
                   c1=*(current_block);
                   c2=*(current_block+1);
                   x1=(unsigned char)((12-i)*(12-j) + (i-4)*(12-j));
                   x2=(unsigned char)((12-i)*(j-4) + (i-4)*(j-4));
                   d=(c1*x1)+(c2*x2);
                   mx=(unsigned char)(d>>6);
                  }
               }
               else
               {
                if(y==63) // Block x=0 to 62,y=63 point i>=4,j>=4
                  {
                   c1=*(current_block);
                   c2=*(current_block+64);
                   x1=(unsigned char)((12-i)*(12-j) + (12-i)*(j-4));
                   x2=(unsigned char)((i-4)*(12-j) + (i-4)*(j-4));
                   d=(c1*x1)+(c2*x2);
                   mx=(unsigned char)(d>>6);
                  }
                  else // Block x=0 to 62 ,y=0 to 62 point i>=4,j>=4
                  {
                    c1=*(current_block);
                    c2=*(current_block+1);
                    c3=*(current_block+64);
                    c4=*(current_block+65);
                    x1=(unsigned char)((12-i)*(12-j));
                    x2=(unsigned char)((12-i)*(j-4));
                    x3=(unsigned char)((i-4)*(12-j));
                    x4=(unsigned char)((i-4)*(j-4));
                    d=(c1*x1)+(c2*x2)+(c3*x3)+(c4*x4);
                    mx=(unsigned char)(d>>6);
                  }
               }
            LOG_printf(&trace, "mean4 %d:",mx);
             return mx;
}


void s_array(int mean_diff,unsigned char *s,unsigned char *image,short
*frame)
{
        int
k,m1,m2,m3,m4,m5,m6,m7,m8,m9,m10,hi,lo,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10;
```

```
        unsigned char*im1=s,*img=image,ch;
        for(k=0;k<8;k++)
         {
           m1=_hi(_amemd8(im1));
           m2=_lo(_amemd8(im1));

           x1=_hi(_amemd8(img));
           x2=_lo(_amemd8(img));

           m3= (int)_extu(m1,24,24) + mean_diff;
           m4= (int)_extu(m1,16,24)+ mean_diff;
           m5= (int)_extu(m1,8,24)+ mean_diff;
           m6= (int)_extu(m1,0,24)+ mean_diff;

           x3= (int)_extu(x1,24,24) ;
           x4= (int)_extu(x1,16,24);
           x5= (int)_extu(x1,8,24);
           x6= (int)_extu(x1,0,24);

           m7= (int)_extu(m2,24,24)+ mean_diff;
           m8= (int)_extu(m2,16,24)+ mean_diff;
           m9= (int)_extu(m2,8,24)+ mean_diff;
           m10= (int)_extu(m2,0,24)+ mean_diff;

             x7= (int)_extu(x2,24,24);
           x8= (int)_extu(x2,16,24);
           x9= (int)_extu(x2,8,24);
           x10=(int) _extu(x2,0,24);

           x7=x7- m7;
           x8=x8-m8;
           x9=x9-m9;
           x10=x10-m10;

           x3=x3-m3;
           x4=x4-m4;
           x5=x5-m5;
           x6=x6-m6;

           *(frame+k)=(short)x7;
           *(frame+8+k)=(short)x8;
           *(frame+16+k)=(short)x9;
           *(frame+24+k)=(short)x10;

           *(frame+32+k)=(short)x3;
           *(frame+40+k)=(short)x4;
           *(frame+48+k)=(short)x5;
           *(frame+56+k)=(short)x6;

           hi=m3 | (m4<<8) | (m5<<16)|(m6<<24);
           lo=m7 | (m8<<8) | (m9<<16) |(m10<<24);

           _amemd8(im1)=_itod(hi,lo);
           im1=im1+512;
           img=img+512;
         }

}
```

```
void estimste(float *yy,float *w,float *o,short *ptf3, int f2)
  {
    int i,j,it;
    float sw,wp,so,op,maxw,sig,squ,f1,f3,f4,f5,fm;
    short chx,sf1,sf;

    /*for(i=0;i<4096;i++)
     {
       for(j=0;j<64;j++)
        chx=*(ptf3+(i<<6)+j);
     }*/


    for (i=0;i<64;i++)
       *(w+i)=i/(float)64;

    for(it=0;it<10;it++)
    {
     sw=0;
     for(j=0;j<64;j++)
          sw=sw+((*(w+j))*(*(w+j)));

     for (i=0;i<4096;i++)
        {
          wp=0;
          for (j=0;j<64;j++)
              wp=wp+((*(w+j))*((float)(*(ptf3+(i<<6)+j))));

          *(o+i)=(wp)/sw;
          sig=*(o+i);
         }

      so=0;

      for (i=0;i<4096;i++)
       {
        // sig=*(o+i);
        // squ=sig*sig;
        // so=so+squ;

        so=so+((*(o+i))*(*(o+i)));
       }


      for (j=0;j<64;j++)
         {
           op=0;
            for (i=0;i<4096;i++)
              op=op+((*(o+i))*(*(ptf3+(i<<6)+j))); // op is multiple of 128
only

            *(w+j)=(op)/so; // w is multiple of 128
          }
      } // end of it =10

      maxw = -1000000;
      for(j=0;j<64;j++)
       if (fabs(*(w+j))>maxw)
          maxw=fabs(*(w+j));
```

```
      for (j=0;j<64;j++)
         *(w+j)=(*(w+j))/maxw; // w is multiple of 128

      for(i=0;i<4096;i++)
         *(o+i)=((*(o+i))*maxw); //

      maxo=-1000000;
     for (i=0;i<4096;i++)
      if (fabs(*(o+i))>maxo)
          maxo=fabs(*(o+i));
    //   if(maxo<0.00001) maxo=0.1;
     LOG_printf(&trace, " In ESTIMATE MAX %f",maxo);
      so=0;
      for (i=0;i<4096;i++)
        so=so+((*(o+i))*(*(o+i)));

      for (j=0;j<64;j++)
        {
         op=0;
         for (i=0;i<4096;i++)
           op=op+((*(o+i))*((*(ptf3+(i<<6)+j))));

         *(w+j)=(op)/so;

         /* if (fabs(*(w+j))>1)
            *(w+j)=(0.99*(*(w+j)))/fabs(*(w+j));  */
        }
      LOG_printf(&trace, "w done :");
      f4=(maxo/(olevels>>1));
     for (i=0;i<4096;i++)
         {
           f1=(((*(o+i))*(olevels>>1))/maxo);
           sf1=(short)f1;
           f3=f1-sf1;
           if(fabs(f3)>=0.5)
             {
              if (sf1>=0)
                sf1++;
              else
                 sf1--;
             }
             fm=(maxo/(olevels>>1))*sf1;
           for(j=0;j<64;j++)
            {
              f5=(fm)*(*(w+j));
             // sf=(short)f5;
              *(yy+(i<<6)+j)=f5;

              // *(yy+(i<<6)+j)=((int)((*(w+j))*(*(o+i))))<<7;  // yy is
multiple of 128
             }
          }

     LOG_printf(&trace, "estimate done :");
  }
```

VITA

Vikas Kumar Reddy Sanikommu was born in Hyderabad, Andhra Pradesh, India. He graduated from Ratna Junior College in 1998. In the same year he joined Sri Venkateswara Engineering College he began studies in Electronics and Communications Department and the following year he transferred to Srinidhi Institute of Science and Technology and graduated from the same with a Bachelor of Engineering degree in May 2002. The following FALL he came to the University of New Orleans to pursue graduate studies in Electrical Engineering.