DSP IMPLEMENTATION OF LATTICE VECTOR INDEXING FOR IMAGE COMPRESSION

R. R. Khandelwal[1], P. K. Purohit[2] and S. K. Shriwastava[3]

[1]Shri Ramdeobaba College Of Engineering and Management, Nagpur
richareema@rediffmail.com
[2]National Institute of Technical Teachers' Training & Research, Bhopal
purohit_pk2004@yahoo.com
[3]Shri Balaji Institute of Technology & Management, Betul
skshriwastava@gmail.com

ABSTRACT

Image-related interactions 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. Image processing applications are increasingly being done on embedded systems because image processing involves high computation and data requirements. This paper work describes the implementation of indexing of vectors quantized using lattice structure on TMS320C6713 Digital Signal Processor board. In this work for quantization of vectors D_4 lattice structure is selected. The proposed indexing technique is based on direct assignment of indices to the vectors. Generation of the codebook and the indexing of leader are not required in this method . The code is written in C in Code Composer Studio (CCS) in order to program the DSP processor. With the help of profile statistic available in CCS, statistical analysis is also carried out.

KEYWORDS

Lattice Vector Quantization (LVQ), Indexing, Digital Signal Processor (DSP), Code Composer Studio (CCS)

1. INTRODUCTION

Digital images are full of large amount of data. Therefore, an image with high quality implies that the associated file size is large. For 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. Digital signal processors are found in embedded environments like cellular phones, fax/modems, disk drives, radio, printers, hearing aids, MP3 players, high-definition television (HDTV), digital cameras, and so on. These processors have become the products of choice for a number of consumer applications, since they have become very cost-effective [1]. They can handle different tasks, since they can be reprogrammed readily for a different application. DSP techniques have been very successful because of the development of low-cost software and hardware support. For example, modems and speech recognition can be less expensive using DSP techniques. Embedded processors need more computing power as demand of multimedia stream processing ability is increasing. Currently, adoption of VLIW (Very Long Instruction Word) architecture becomes the trend of high-performance DSP processor since VLIW supports instruction level parallelism by low-cost compiler compared to the dynamically hardware-scheduled superscalar processors. TMS320C6x, which is a high-performance DSP popularly adopted for many multimedia applications, is also based on VLIW architecture [2].

Many techniques have been developed in the past, including transform coding and vector quantization for image compression. Transform-based coding techniques have proved to be the most effective in obtaining large compression ratios while retaining good visual quality. Scalar or vector quantizers use discrete codebook similar to lossless compression technique but compression is achieved by limiting the number of possible codes. The main advantage of lattice quantization is the very fast encoding and decoding algorithms proposed in [3]. Such algorithms exploit the symmetry of the root lattices to find the closest lattice point for a given input vector with minimum computational effort. Nearest-neighbor search is carried out only among a limited number of points, as opposed to the exhaustive full codebook search as used in conventional vector quantization techniques.

In [4] Embedded Zerotree Wavelet (EZW) Image compression method on 8x8 image matrix is implemented using digital signal processor. In our paper, proposed indexing method is applied on the 8x8 matrix. Vectors are quantized using quantization algorithm for D_4 lattice [3], and then the indexes are assigned to the quantized vectors. In this case sixteen indexes are generated at the encoder output. Then decoder from these indexes generates the vectors and finally the reconstructed matrix is obtained.

2. QUANTIZATION USING LATTICE STRUCTURE

Quantization, involved in image processing, is a lossy compression technique achieved by compressing a range of values to a single quantum value. When the number of discrete symbols in a given stream is reduced, the stream becomes more compressible. The symmetrical geometric structure associated with a regular lattice and the fast quantization algorithms that this structure implies, have been stimulating for the use of lattices as possible block quantizers. In lattice-based vector quantization, the codebook is constructed from the finite subset of regular lattice points. The symmetry of regular lattice makes the encoding and decoding very simple, as selection and indexing of the nearest codeword for a given input point becomes very straightforward. Furthermore, the codewords are not necessarily stored; they can be generated by a set of simple algebraic rules. Thus, lattice quantization offers a vast reduction in the encoding complexity and eliminates the storage requirement, the two inherent problems in implementing a vector quantizer. D_4 lattice is found to be the densest in four dimension [5], which is used in this work with spherical truncation. It is defined to be the set of all integral points whose coordinates have even sum:

$$D_4 = \{(x_1, x_2, x_3, x_4) \in Z^4\}, where \sum_{i=1}^{4} x_i = even$$

The encoding and decoding process for LVQ is as follows-

Encoding Process-

1. Covert the input matrix of size MxN into Rx4 matrix, where MxN should be equal to Rx4. In our case MxN=8x8 it is converted into 16x4. Since D_4 lattice is selected for quantization. Consider one row of this matrix as $x = (x_1, x_2, x_3, x_4)$ vector.
2. Normalized a source vector x by the scaling factor c, so that the normalized vector x_l= x/c
3. x_l is then mapped to the nearest lattice point using fast quantization algorithm described in [3].
4. Assign index to the lattice point.
5. Same process will be done for all rows of the Rx4 matrix, so an Rx1 matrix will be generated which has all the indexes. This matrix is transmitted to the decoder.

Decoding Process-

1. Recover all the vectors from the received indexes.
2. Obtain decoded vector matrix of order Rx4.
3. Recovered matrix = c X (decoded vector matrix).
4. Apply reshaping to convert Rx4 matrix into MxN matrix, with condition that MxN = Rx4.

3. A REVIEW OF INDEXING METHODS FOR LATTICE VECTORS

Quantization using lattice structure is very fast because of the availability of fast quantization algorithms explained in [3] for different lattices. But indexing of lattice vectors is not simple because of the large size of the codebooks. In a system, indexing process must be as fast as possible at the coding stage as well as at the decoding one. The crucial problem in lattice encoding technique is to represent any quantization value by a uniquely decodable binary word. To solve this problem, product code approach is used, in which a lattice vector x can be defined by an index pair (i, j) where i is the index of the radius or norm defined by

$$l_p = \sum_{i=1}^{n} \left| x_i^p \right| \cdots\cdots\cdots\cdots (1)$$

And called prefix of the x and j is the index of the position of x on the shell of radius R called suffix of x. The binary word constituted in two parts, a binary prefix code for the energy and a binary suffix code for the position of the codevector on the shell corresponding to this energy. The complete index for a lattice vector is formed by multiplexing the codes of the prefix and the suffix. The prefix part that represents the energy of the vector can be easily indexed and encoded. Indexing of the suffix part is a difficult operation because of the large size of codebooks. The indexing of the suffix can be done in two different ways.

1. **Indexing based on enumerating lattice points-** In this method index taken into account the total number of vectors lying on a given hyper-surface. Several enumeration techniques have been proposed for the case of Laplacian and Gaussian distributions and for different lattices. A recursive formula to compute the total number of lattice vectors lying on a l_1 norm hyper-pyramid in case of Laplacian distribution and for a Z^n lattice has been introduced in [6]. This enumeration formula has been extended in [7] for generalized Gaussian source distributions.

For practical implementations the total number of lattice vectors lying on a hyper surface can quickly reach non tractable values, the index value based on the overall number of vectors [6], [7] can quickly overflow the computer precision.

2. **Indexing based on leader's addressing -** The method proposed in [8] is based on the knowledge of some lattice vectors, called leaders. Leaders are the vectors of a hyper-surface, from which operations of permutations and sign changes lead to all the other vectors lying on this hyper-surface. Indeed, leaders are vectors with positive coordinates stored in decreasing order. The idea behind the method proposed in [8] is to create a suffix index based on the construction of a look up table and the use of symmetries of the lattice. This look-up-table contains the index of the leaders from which all the other vectors of the hyper- surface can be assigned. The symmetries correspond to two basic operations viz. sign changes and permutations of the vector coordinates. Therefore, instead of indexing directly all the vectors over an hyper-surface, this indexing method assigns to each vector a set of three indices, one corresponding to its leader and the other two corresponding to its permutation and the sign changes from the leader. In [9] vector indexing with absolute leaders and signed leaders are done. With signed leaders small index values can achieved as compared with the absolute leaders. Since in these methods size of the codebook is reduced but a look-up-table containing all the leaders must be constructed for hyper surface of a dimension. To construct this table, it is necessary to know or to generate all of the leaders, which remains a complex operation.

   To avoid the construction of look up table even for the leaders, a method to directly address the leaders based on partition function is proposed in [10]. A partition function q(r, n) is used in [10] which not only gives the total number of leaders lying on a given hyper-pyramid but can also be used to provide unique indices for these leaders. By this method, indexing a leader can be done even for large vector dimensions. Partition function method is extended for Z^n lattice vectors for generalized Gaussian distribution in [11].

Method proposed in this paper is not based on product code indexing, it is based on the independent indexing of vectors. In proposed method no need to find out the norm of the vector, index of the leader etc.

4. PROPOSED METHOD FOR INDEXING OF QUANTIZED VECTORS

In our work we have taken an input 8x8 matrix, this matrix is reshaped into 16x4 matrix. Each row of the reshaped matrix represents one vector. All the vectors of 16x4 matrix are quantized using quantization algorithm of D_4 lattice structure described in [3]. Then indices are assigned to each quantized vector according to proposed indexing method. This transition method encodes the four dimensional vectors into scalar integer values. It may be noted that, there is no actual codebook present in this method, as compared to traditional vector quantization methods. The transition calculates the possible encoded string for the vector or the vector itself from the encoded string, thus establishing a virtual dynamic codebook. The collection of these scalar encoded string quantities builds the basic compressed image. Decoding is done by first applying reverse transition (vector to index) to obtain the quantized vectors then after reshaping, 8x8 matrix is reconstructed. For example if a vector for quantization is x = (1.8, 0.2, 0.3, 0.4). Then following steps are used to get the index of this vector:

Step 1: Quantization - Conway and Sloane [3] developed a fast quantization algorithm which makes searching of the closest lattice point to a given vector extremely fast. For a given vector x, the closest point of D_4 is, whichever of f(x) and g(x) has an even sum of components (one will have an even sum, the other an odd sum). This procedure works because f(x) and g(x) differ by one in exactly one coordinate, and so precisely one of $\sum f(x_i)$ and $\sum g(x_i)$ is even and the other is odd. After execution of this step the vector will be converted into f(x) and g(x) vectors. f(x) = (2, 0, 0, 0) and $\sum f(x_i)$ = even while g(x) = (2, 1, 0, 0) and $\sum g(x_i)$ = odd since the second component of x is the furthest from an integer, so it is changed from 0 to 1 in g(x). $\sum f(x_i)$ = even therefore f(x) is the point of D_4 closest to x.

Step 2: Indexing - After getting quantized vector, index is assigned to the vector, according to our proposed method. Here we have selected base value 11, vector components are multiplied by the power of base according to their position in a vector. So the index of vector (2, 0, 0, 0) will be calculated as, Index= 2 + 0xbase + 0xbase^2 + 0xbase^3 = 2, 2 bits are required to represent this index..

In literature mainly two methods are there to get the index of a lattice quantized vector, which are explained in section 3 of this paper. So the index of the same vector x calculated by first method (enumerating lattice points) is (2, 1), where 2 represent the norm of the vector and 1 represent the index of that vector. The index of the same vector by second method which is based on leader's addressing is (2, 1, 1), where 2 represent the norm of the vector and first 1 signify the index of the leader and second 1 imply rank of the vector.

5. EXPERIMENTAL ENVIRONMENTS

5.1. DSK board

The above proposed algorithm is implemented on the TI DSP TMS320C6713 DSK board in the floating-point arithmetic. C6713 DSK board is provided with Code Composer Studio (CCS), integrated environments for developing a TI DSP system. The board is based on high performance, advanced very-long-instruction-word architecture. This board operates at 225MHz and 8MB 100MHz SDRAM. 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).

DSP processors are basically designed to perform fast calculations. SOP (Sum Of Product) is the key algorithm for most of the DSP algorithms. Hence DSP processor are basically designed to perform these operations faster than other processors, so that we can conduct real time processing on the received data and get the desired output.

To achieve this goal, additional functional units, other than 32 bit registers, are provided in DSP processors which work simultaneously to carry out both multiplication and addition at the same time [12]. This is shown in Figure 1.

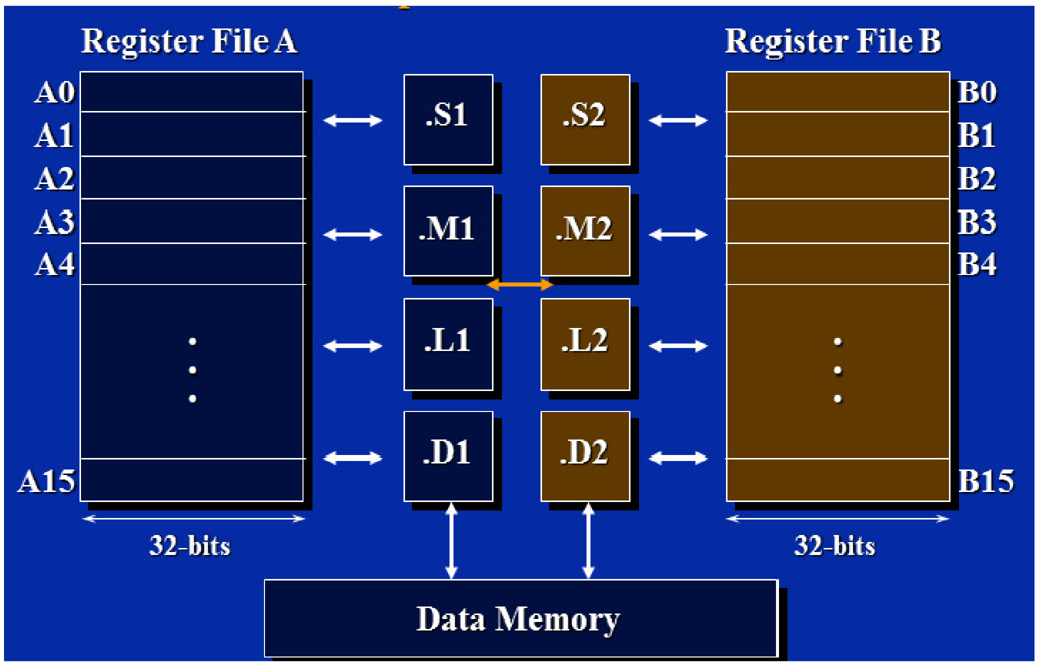


Figure 1. TMS320C6713 data path

Here two sets of general purpose 32 bit register A and B are used to store the operand data. These registers cannot be uploaded directly from memory. Hence the ".D" functional unit provided above performs this operation. It is used to exchange data between registers and external memory. To achieve fast calculation of SOP a separate ".M" and ".L" units that perform multiplication and addition respectively are in the processor. They perform their operations independently and simultaneously. The ".S" unit is used to perform the operation of addition and multiplication in a loop. Both registers sets A and B have their separate functional units that can perform operations independently and they can also exchange data between them via internal data link buses. Hence these special features of DSP processor enable it to perform faster calculations of SOP than other processors.

5.2. Code Composer Studio

The Code Composer Studio (CCS) application provides an integrated environment with the following capabilities:

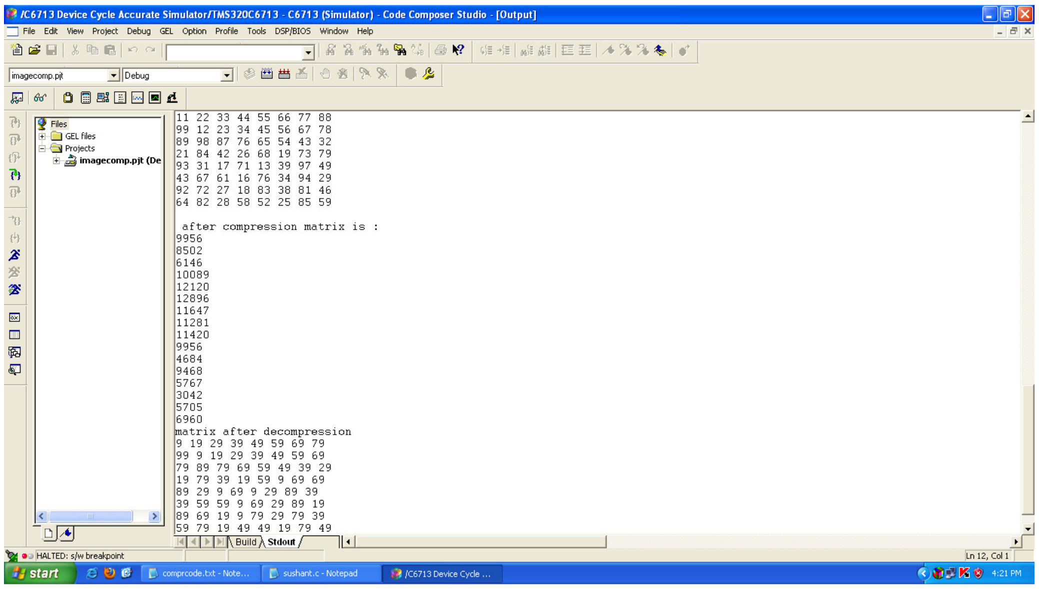
- Integrated development environment with an editor, debugger, project manager, profiler, etc.
- 'C/C++' compiler, assembly optimizer and linker (code generation tools).
- Simulator.
- Real time operating system (DSP/BIOS).
- Real time data exchange between the host and target.
- Real time analysis and data visualization.

6. RESULTS

6.1. Execution of algorithm

The proposed indexing method is applied on 8x8 size matrix, which is shown in the upper part of the CCS window of Figure 2, after applying proposed indexing algorithm on it, a 16x1 matrix is obtained at the output of encoder, shown in the middle part of the same window. For decoder this 16x1 matrix works as input and from these indexes decoder reconstructs the matrix, which is shown in the lower part of the window. Error in the reconstructed data is calculated by subtracting reconstructed matrix from original matrix, as shown below as error matrix.

$$\text{Error Matrix} = \begin{pmatrix} 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 0 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 \\ 2 & 5 & 3 & 7 & 9 & 10 & 4 & 10 \\ 4 & 2 & 8 & 2 & 4 & 10 & 8 & 10 \\ 4 & 8 & 2 & 7 & 7 & 5 & 5 & 10 \\ 3 & 3 & 8 & 9 & 4 & 9 & 2 & 7 \\ 5 & 3 & 9 & 9 & 3 & 6 & 6 & 10 \end{pmatrix}$$



```
11 22 33 44 55 66 77 88
99 12 23 34 45 56 67 78
89 98 87 76 65 54 43 32
21 84 42 26 68 19 73 79
93 31 17 71 13 39 97 49
43 67 61 16 76 34 94 29
92 72 27 18 83 38 81 46
64 82 28 58 52 25 85 59

 after compression matrix is :
9956
8502
6146
10089
12120
12896
11647
11281
11420
9956
4684
9468
5767
3042
5705
6960
matrix after decompression
9 19 29 39 49 59 69 79
99 9 19 29 39 49 59 69
79 89 79 69 59 49 39 29
19 79 39 19 59 9 69 69
89 29 9 69 9 29 89 39
39 59 59 9 69 29 89 19
89 69 19 9 79 29 79 39
59 79 19 49 49 19 79 49
```

Figure 2. Encoder and decoder output for 8x8 matrix

6.2. Code Analysis

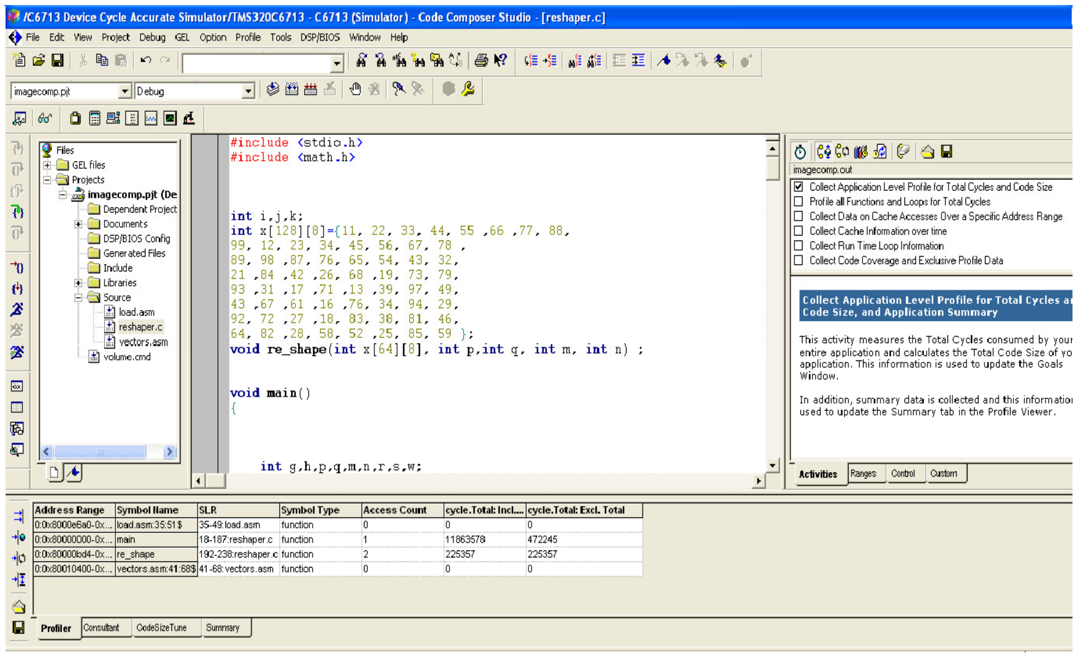
Statistical analysis is done by using the profile statistic of Code Composer Studio (CCS). Figure 3 shows the result of profiling in the profile viewer window. Results show-

i) Access count which is the number of times that CCS profiled the function.
ii) Inclusive average which is the average number of cycles needed to run the function including any calls to subroutines.

iii) Exclusive average which is the average number of cycles needed to run the function excluding any calls to subroutines.

6.3. Comparison with MATLAB execution

When the execution of same algorithm is performed on MATLAB with 2.1 GHz processor it requires approximately 0.03 seconds for execution. It means 63000000 CPU cycles are required. While in case of DSP implementation total number of CPU cycles including calls to subroutines is 12088935.



```
#include <stdio.h>
#include <math.h>



int i,j,k;
int x[128][8]={11, 22, 33, 44, 55 ,66 ,77, 88,
99, 12, 23, 34, 45, 56, 67, 78 ,
89, 98 ,87, 76, 65, 54, 43, 32,
21 ,84 ,42 ,26, 68 ,19, 73, 79,
93 ,31 ,17 ,71 ,13 ,39, 97, 49,
43 ,67 ,61 ,16 ,76, 34, 94, 29,
92, 72 ,27 ,18, 83, 38, 81, 46,
64, 82 ,28, 58, 52 ,25, 85, 59 };
void re_shape(int x[64][8], int p,int q, int m, int n) ;


void main()
{



    int g,h,p,q,m,n,r,s,w;
```

☑ Collect Application Level Profile for Total Cycles and Code Size
☐ Profile all Functions and Loops for Total Cycles
☐ Collect Data on Cache Accesses Over a Specific Address Range
☐ Collect Cache Information over time
☐ Collect Run Time Loop Information
☐ Collect Code Coverage and Exclusive Profile Data

Collect Application Level Profile for Total Cycles a Code Size, and Application Summary

This activity measures the Total Cycles consumed by your entire application and calculates the Total Code Size of yo application. This information is used to update the Goals Window.

In addition, summary data is collected and this informatio used to update the Summary tab in the Profile Viewer.

Address Range	Symbol Name	SLR	Symbol Type	Access Count	cycle.Total: Incl....	cycle.Total: Excl. Total
0:0x8000e6a0-0x...	load.asm:35:51$	35-49:load.asm	function	0	0	0
0:0x80000000-0x...	main	18-187:reshaper.c	function	1	11863578	472245
0:0x80000bd4-0x...	re_shape	192-238:reshaper.c	function	2	225357	225357
0:0x80010400-0x...	vectors.asm:41:68$	41-68:vectors.asm	function	0	0	0

Figure 3. Statistical Analysis in Code Composer Studio

7. CONCLUSION

Encoder and decoder using lattice vector quantization with proposed indexing method are successfully implemented on floating point digital signal processor TMS320C6713. The program is written in C. Error matrix is calculated by subtracting reconstructed matrix from original matrix. The number of cycles required to execute the function including calls to subroutines and excluding calls to subroutines are calculated using profile statistic of Code Composer Studio. The number of cycles required to run functions are less using processor as compare to execution on MATLAB. The number of cycles required to run functions can still be reduced with optimization techniques.

REFERENCES

[1] Rulph Chassaing, Digtal Signal Processing and Applications with the C6713 and C6416 DSK, John Wiley & Sons, Inc. publisher.

[2] Hui-Jae You, Sun-Tae Chung, and Souhwan Jung (2008), "Optimization of SAD Algorithm on VLIW DSP", World Academy of Science, Engineering and Technology 37.

[3] J. H. Conway and N. J. A. Sloane (1982), "Fast Quantizing and Decoding Algorithms for Lattice Quantizers and Codes", IEEE Transaction on Information Theory, Vol. IT-28, pp 227-232.

[4] Shamika M. jog, Shashikant D. Lokhande, (2009), "A DSP Implementation of Embedded Zerotree Wavelet (EZW) Image CODEC in Image Compression System", International Journal of recent Trends in Engineering, Vol 2, No. 4, pp 162-164.

[5] J. H. Conway and N. J. A. Sloane (1982), "Voronoi Region of Lattices, Second Moments of Polytopes, and Quantization", IEEE Transaction on Information Theory, Vol. IT-28, pp 211-226.

[6] T. R. Fiser (1986), "A Pyramid Vector Quantizer," IEEE Transaction on Infomation. Theory, Vol. 32, pp 568-583.

[7] Z. G. F. Chen and J. Villasenor (1997), "Lattice Vector Quantization of Generalized Gaussian Sources", IEEE Transaction on Information Theory, Vol. 43, pp 92-103.

[8] J. Moureaux, P. Loyer, and M. Antonini (1998), "Low Complexity Indexing Method for Zn and Dn Lattice Quantizers", IEEE Transaction on Communication, Vol. 46, No. 12, pp 1602–1609.

[9] Patrick Rault and Christine Guillemot (2001), "Vector Indexing and Lattice Vector quantization with Reduced or without Look up Table"

[10] Mahmoud Mejdoub, Leonardo Fonteles, Chokri BenAmar and Marc Antonini (2007), "Fast Algorithm for Image Database Indexing Based on Lattice", in Proc. of EUSIPCO.

[11] L. H. Fonteles and M. Antonini (2007), "Indexing Zn Lattice Vectors for Generalized Gaussian Distributions", in Proc. of IEEE International Symposium on Information Technology, pp 241-245.

[12] Texas Instrument, TMS320C6713 datasheet