# Adaptive Coding using Finite State Hierarchical Table Lookup Vector Quantization with Variable Block Sizes*

Sanjeev Mehrotra, Navin Chaddha, and R.M. Gray
Information Systems Laboratory, Stanford University,
Stanford, CA 94305
Tel: (415) 723-2675 Fax: (415) 723-8473 Email: mehrotra@leland.stanford.edu

## Abstract

*In this paper we present an algorithm for performing adaptive vector quantization with memory. By using memory between adjacent blocks which are encoded, we can take advantage of the correlation between adjacent blocks of pixels to reduce redundancy. We use finite state vector quantization to provide the memory. To further improve performance by exploiting nonstationarities in the image, we use variable block sizes in the encoding. This is done by using a quadtree data structure to represent an encoding based on using variable block sizes. To reduce encoding complexity, hierarchical table lookup schemes are used to replace all the full search encoders.*

## 1. Introduction

One of the main problems with memoryless full search vector quantization (VQ) is that the encoding complexity grows exponentially with bit rate and vector dimension [1]. For example if $r$ bits per pixel were used to represent a $k$ dimensional vector, then a codebook of size $2^{rk}$ would be needed. This would mean that a full search encoder would have to compute $2^{rk}$ distortions to find the minimum distortion codeword. However, the decoder would simply be implemented as a table lookup. Therefore VQ typically can only be used where a complex encoder can be implemented, but a relatively low complexity decoder is required, such as software decoding of video from a CD-ROM.

To maintain low codebook design and encoder complexity, VQ can only be used with small vector dimensions if a high bit rate is required. If large vector dimensions are used, then only small codebooks can be used. However, this often causes too much compression and subsequently too much distortion. Therefore a lot of work has gone into achieving the performance of a large vector dimension VQ while keeping a small vector dimension. These algorithms typically achieve better performance by taking advantage of redundancy between adjacent blocks [5].

Even with small codebooks, full search VQ has high complexity. Therefore, we must reduce the encoding complexity at the expense of a slight decrease in performance. To reduce encoding complexity, one can use table lookups to perform the encoding. Since the tables can be built in advance, no arithmetic computations are required to perform the encodings. To make table sizes manageable for large vector dimensions, hierarchical table lookup vector quantization (HVQ) can be used [2, 3]. Once the complexity is reduced using algorithms such as HVQ, VQ with memory, such as FSVQ and PVQ, can be used to improve the performance [4].

Another problem with VQ is that the use of a single quantizer, in the absence of variable rate coding, allocates the same number of bits to all pixels in the image. This is usually not good since images are nonstationary and have some regions which can be highly compressed and other regions with fine detail, such as edges, which cannot be compressed well. To take advantage of nonstationarities in images, one can use bit allocation approaches after identifying regions of greater importance [8]. One such approach is to use variable block size vector quantization [10, 9].

## 2. Hierarchical table lookup VQ

An obvious way to reduce encoding complexity is to precompute the minimum distortion codeword for every possible input. Then the input vector can be used to directly address a lookup table to determine the index of the nearest codeword. Since this table can be built in advance, no arithmetic computations would be required for the encoding. However, since there are many possible input vectors, the size of this table would be immense if the vector dimension were large and would only be possible with what

*Work supported by NSF Graduate Research Fellowship, Kodak Fellowship, and NSF Research Grant MIP-9311190

practically speaking would be an infinite amount of memory. For example, a 16 dimensional vector at 8 bits per pixel would have $2^{128}$ possible input vectors. To fix this problem, we can use hierarchical table-lookup vector quantization (HVQ). By performing the table lookups in a hierarchy, larger vectors can be accommodated in a practical way, as shown in 6.

Another big advantage of table lookup encoding is the ease with which block transforms and complex distortion measures, such as those using perceptual weighting, can be incorporated. This is simply because these things can be incorporated into the table building process itself, rather than in the encoding process. The encoding process will remain exactly the same and the encoding computational complexity will not increase at all. The details of HVQ can be found in [3].

## 3. VQ with Memory

Vector quantization with memory allows us to take advantage of the correlation between adjacent blocks being encoded. This helps in reducing the bit rate needed to achieve a particular distortion thus improving performance. Two common types of VQ with memory are finite state vector quantization (FSVQ) and predictive vector quantization (PVQ). In this paper we consider FSVQ.

FSVQ [7] improves performance by using multiple codebooks to achieve the performance of a larger codebook. It uses many codebooks, called state codebooks, each corresponding to a different state of the encoder. In FSVQ the search for the minimum distortion codeword is limited to searching the codewords in the current state codebook. The current state is a function of the quantized versions of the adjacent blocks. Since the state is a function of only previous states and previous indices (quantized blocks), the decoder can track the state of the encoder without any additional side information once the initial states are known. The current state of the encoder is given by $S_n = f(S_{n-1}, y_{n-1})$ where $S_n$ is the state at time $n$ and $y_n$ is the index output by the encoder at time $n$. Then, the encoder uses the codebook $C_S$ to encode the vector if the encoder is in state $S$. Thus, the output index of the encoder at time $n$ is given by $i_n = \alpha_S(\mathbf{x_n}) = \text{argmin}_i d(\mathbf{x_n}, \beta_S(i))$ where $d$ is the distortion measure, $\alpha_S$ is the encoder for state S, and $\beta_S$ is the decoder for state $S$. In our setup, we use the side pixel values of the adjacent blocks to determine the state, as shown in figure 6. The state is calculated by using VQ to classify quantized versions of the adjacent pixel vectors. This helps preserve edge and grayscale continuities since the current codebook is selected based on the boundary pixel values. So instead of having one codebook which contains global codewords, FSVQ codebooks have codewords which are local to the blocks which map to that state. Thus, there is less variance among the vectors, leading to better clustering and subsequent lowering of the MSE.

## 4. Variable Block Size Encoding

To take advantages of nonstationarities in the image, we use variable block sizes using a quadtree decomposition to perform the encoding [10, 9]. This allows us to adaptively allocate a different number of bits to each pixel depending on the the statistics of the spatial region that we are encoding. If we design quantizers for various block sizes, each with the same number of codewords, then the larger vector dimension quantizers will have a lower rate than the ones for the the small block sizes. If regions of low detail are compressed using large block sizes, and regions of fine detail are encoded by breaking the large block into smaller blocks, then we have adaptively allocated more bits to the important regions. Although the quadtree segmentation information will be sent using added side information, the gains resulting from using multiple quantizers will more then make up for this.

The objective of the algorithm is to encode the image optimally by using the quantizers available [9]. Suppose we have quantizers available for encoding blocks of size $2^l$ x $2^l$, where $l_0 \leq l \leq L$. Then the optimal encoding for an image is the optimal encoding for each of the $2^L$ x $2^L$ blocks in the image. The optimum encoding for each block is determined recursively by comparing the encoding of the block with the encoding of the block using four optimally encoded subblocks.

To find the optimum encoding for a $2^l$ x $2^l$ block, we first encode it using a $2^l$ x $2^l$ quantizer. Let $D_l$ and $R_l$ be the distortion and rate for this quantization. Then, we optimally encode the four subblocks resulting from this block. Let $D_{l-1}$ and $R_{l-1}$ be the sum of the distortions and rates for the four optimally encoded subblocks. If $D_l + \lambda R_l \leq D_{l-1} + \lambda R_{l-1}$, then the optimal encoding of the $2^l$ x $2^l$ block is the quantization using the $2^l$ x $2^l$ block, else it is the optimal encoding of the four $2^{l-1}$ x $2^{l-1}$ blocks. $\lambda$ , the Lagrange multiplier, can be changed depending on the compression desired. The larger the value of $\lambda$ , the more compression we achieve. This comparison is done for all levels of quantization available, $l_0 + 1 \leq l \leq L$. The optimal encoding for the $2^{l_0}$ x $2^{l_0}$ block is simply the encoding of the actual block since the block cannot be split any further as there are no quantizers for smaller block sizes available. The rate for the encodings includes one bit for each node in the quadtree except the last level in the quadtree. A quadtree will be used to represent the the segmentation map telling the decoder how each block has been encoded as shown in figure 6. If the node is a one, then the block is being split, else if it is a zero then that block is being encoded using the index sent. To decode a $2^l$ by $2^l$ block, the decoder simply

looks at the quadtree. If the node corresponding to the block is a zero, then the decoder outputs a reproduction block using the codebook for that block size. If the node is a one, then the decoder decodes each of the four subblocks by recursively using the same procedure.

## 5. HVQ Quadtree Encoding with Memory

The combination of VQ with memory with quadtree encoding schemes is a natural one. If we look at the residuals resulting from prediction using 4 x 4 blocks to predict (prediction is done with absence of quantization), we find that the residuals have large regions of constant intensity such as the background, which can be easily compressed with large block sizes and the only regions which need to be encoded using small block sizes are the edges. By doing both of these things we gain the PVQ advantage of a small codebook performing as well as a large memoryless VQ codebook (because of the fact that the residuals have smaller variance) as well as the quadtree advantage by exploiting the nonstationarities in the residual image. It also makes sense to combine FSVQ with quadtree encoding since FSVQ is like PVQ with a non-linear predictor. The encoder is essentially coding the difference between the block and the average values of blocks mapping to that state.

To combine memory VQ with quadtree encoding, we simply incorporate FSVQ into the quadtree encoding algorithm. There are basically two ways in which we can accomplish this. One is to simply use the largest block size, a $2^L$ x $2^L$ block, to incorporate the memory.

For FSVQ this would mean that the side pixels of the adjacent quantized blocks of size $2^L$ x $2^L$ are used to determine the state of the current $2^L$ x $2^L$ block. Then, this block can simply be encoded using the quadtree decomposition described in the previous section using quantizers designed for the state. This will be referred to as the quadtree FSVQ with single classifier method. However, this method is not as good at taking advantage of the full correlation between blocks. This is because if the optimal encoding of a $2^L$ x $2^L$ block involves the use of subblocks to encode, then it makes better sense to encode the subblocks by using the adjacent subblocks to determine the state of the subblocks rather than by simply using the big blocks to determine the state. So in this method not only is the encoding done recursively to find the optimum encoding, but the state classification is also done recursively. This will be called the recursive FSVQ and quadtree decomposition method. In this method, to find the optimal FSVQ encoding of a $2^l$ x $2^l$ block, we first encode the block using the state codebook using the adjacent $2^l$ x $2^l$ quantized blocks to determine the state. This is then compared with the optimal encoding of the subblock resulting from using nearby optimally encoded subblocks to determine the state of each of the subblocks. The only drawback of this method is the increased complexity due to repeated state classifications and the design of classifiers for each of the possible block sizes. Both these methods are shown in figure 6.

## 6. Simulation Results and Conclusion

In this section, we give simulation results for encoding the 512 x 512 monochrome image Lena using the various techniques described in this paper. The original image is at 8 bpp (bits per pixel). All the codebooks were designed using the GLA algorithm on 30 training images. The final transmitted channel symbols were generated by mapping the final indices using a code matched to their probabilities. This gives us a variable rate code. Each codebook was designed with 256 codewords. For quadtree decompositions this means that a 16 x 16 quantizer corresponds to a compression of 256:1 and a 2 x 2 quantizer corresponds to a compression of 4:1.

In figure 5, we show PSNR curves for compression using VQ, FSVQ, VQ with optimal quadtree decomposition, FSVQ with quadtree using a single classifier, and FSVQ with the recursive classification and quadtree encoding. The block size listed for the quadtree results is the largest block size used in the quantization. The smallest block size for the quadtree encoding is 2 x 2 for all cases. The various rates are achieved by simply changing the value of $\lambda$ in the quadtree encoding. For low values of $\lambda$ most of the blocks are being encoded as 2 x 2 and for high values of $\lambda$ , they are being encoded as the largest block size available. Although we are using relatively large block sizes for the maximum block size quantizer in the quadtree encoding, this is possible since the codebooks are very small and HVQ is used.

As one can see from the graphs, FSVQ gains about 3 dB over memoryless VQ. VQ with the optimal quadtree decomposition performs about 2.2-2.6 dB better then regular memoryless VQ but about 0.15 dB worse than FSVQ. The recursive FSVQ with quadtree gives about a 4.4 dB improvement over regular memoryless VQ at 0.4 bpp. It is interesting to note that at the very low rates FSVQ seems to be performing slightly worse then or comparable to regular VQ. This might be because of the fact that at low rates, the distortions in the adjacent blocks leads to faulty state calculations which hurts the encoding of the current block.

Figure 6 shows the compressed image resulting from recursive FSVQ with quadtree at a rate of 0.327 bpp at a PSNR of 32.27 dB. Figure 7 shows the same image using standard JPEG (unoptimized) at a comparable rate. As one can see, the JPEG image has the same PSNR, but is more blocky than the recursive FSVQ with quadtree image. Also shown in figure 8 is the corresponding segmentation map for the encoding. From the segmentation map, it is relatively easy to see how we are taking advantage of the nonstationarities in the

image to improve our encoding. Basically each block, regardless of size, is being assigned 8 bits (i.e. 256 codewords in the codebook). The blocks which correspond to regions of detail are being encoded using small block sizes, thus allocating more bits to them than to the large block sizes.

## References


[1] A. Gersho and R.M. Gray, *Vector Quantization and Signal Compression*, Kluwer Academic Pub., Boston, MA, 1992.

[2] P.-C. Chang, J. May and R.M. Gray, "Hierarchical Vector Quantization with Table-Lookup Encoders," *Proc. Intl. Conf. on Communications*, Chicago, IL, June 1985, pp. 1452-55.

[3] N. Chaddha, M. Vishwanath and P.A. Chou, "Hierarchical Vector Quantization of Perceptually Wieghted Block Transforms," *Proc. Data Compression Conference*, March 1995.

[4] N. Chaddha, P.A. Chou and R.M. Gray, "Constrained and Recursive Hierarchical Table-Lookup Vector Quantization," *Proc. Data Compression Conference*, March 1996.

[5] R. Arvind and A. Gersho, "Image Compression Based on Vector Quantization with Memory," *Optical Engineering*, July 1987, vol. 26, pp. 570-580.

[6] H.-M. Hang and J.W. Woods, "Predictive Vector Quantization of Images," *IEEE Trans. Communications*, COM-33, Nov. 1985, pp. 1208-1219.

[7] J. Foster, R.M. Gray and M.O. Dunham, "Finite State Vector Quantization for Images," *IEEE Trans. Information Theory*, May 1985, vol. IT-31, pp. 348-355.

[8] Y. Shoham and A. Gersho, "Efficient Bit Allocation for an Arbitrary Set of Quantizers," *IEEE Trans. Acoust. Speech Signal Processing*, Sept. 1988, vol. 36, pp. 1445-1453.

[9] G.J. Sullivan and R.L. Baker, "Efficient Quadtree Coding of Images and Video," *IEEE Trans. Image Processing*, May 1994, vol 3, pp. 327-331.

[10] J. Vaisey and A. Gersho, "Image Compression with Variable Block Size Segmentation," *IEEE Trans. Signal Processing*, Aug. 1992, vol. 40, pp. 2040-2060.


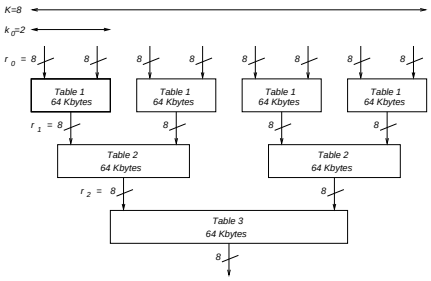


**Figure 1. A 3 stage HVQ encoder**

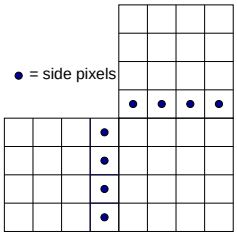


**Figure 2. Side pixels for incorporating memory**

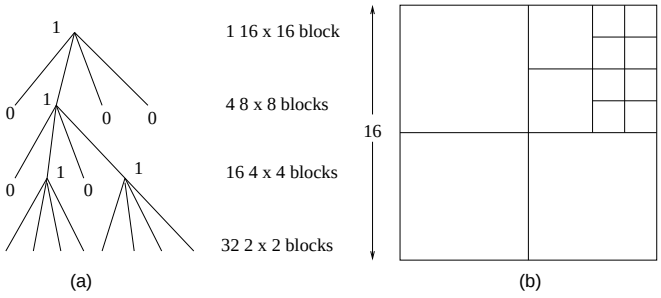


**Figure 3. (a) Quadtree and (b) corresponding block decomposition. Quadtree adds 9 bits of side information (1 bit/node except leaves).**

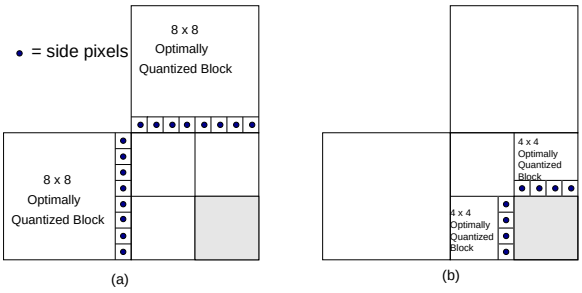


**Figure 4. Side pixels used for state calculation; In (a), the largest block size is used to incorporate memory into the subblocks whereas in (b), the adjacent subblocks are used to incorporate memory into the subblocks.**

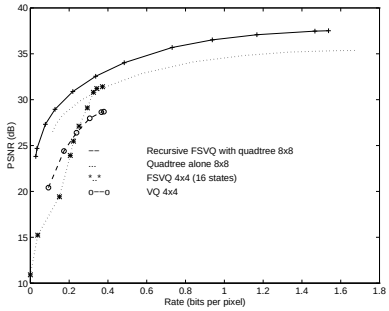


**Figure 5. PSNR results for FSVQ with quadtree.**

**Figure 6. Lena compressed with recursive FSVQ quadtree at .327 bpp with PSNR = 32.6 dB; 2x2 to 8x8 quantizers used.**

**Figure 7. Lena compressed at .33 bpp using JPEG; PSNR = 32.6 dB.**

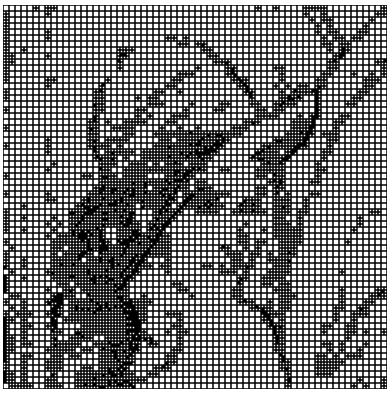

**Figure 8. Segmentation map for Lena shown in figure 6.**