DYNAMIC MEMORY MODEL BASED OPTIMIZATION OF SCALAR AND VECTOR QUANTIZER FOR FAST IMAGE ENCODING

Gene Cheung, Steven McCanne

Department of EECS, University of California, Berkeley

ABSTRACT

The rapid progress of computers and today's heterogeneous computing environment means computation-intensive signal processing algorithms must be optimized for performance in a machine dependent fashion. In this paper, we present formal machine-dependent optimizations of scalar and vector quantizer encoders. Using a dynamic memory model, the optimal computation-memory tradeoff is exploited to minimize the encoding time. Experiments show marked improvements over existing techniques.

1. INTRODUCTION

If the computer evolution has matured to a stage where computers are ubiquitous and homogeneous, and improvements are asymptotic, then implementation of a signal processing algorithm needs only be painstakingly hand-coded once for optimal performance. Unfortunately, computers continue to progress at an exponential rate, and computing environments are extremely diverse. Clearly, hand-coding an algorithm for every possible platform is impractical. On the other hand, simply compiling a fixed algorithm written in a high level language for each machine is sub-optimal, since machine dependent information such as memory hierarchy is unexploited at the algorithmic level. A fundamental question surfaces: how to re-target an algorithm onto different machine platforms optimally *and* automatically? In light of this problem, recent research [1], [2] has looked at the distortion-computation tradeoffs of particular algorithms, thus providing formal analyses of algorithm tuning for machines with different computational budgets.

Our work [3] differs from previous work in that instead of minimizing the number of computational units for a given distortion, we search for the optimal computation-memory tradeoff to minimize running time: divide the processing so that the optimal subset is implemented as simple data memory retrievals of pre-computed values (*pre-compute*), and the other is implemented as on-the-fly computations (*compute*). *pre-compute* requires only a single data memory lookup but may lead to memory blow-up; *compute* can be slow but avoids the memory implosion problem. If the tradeoff between compute and pre-compute is correctly exploited for a given machine, it can lead to enhanced performance over techniques that completely ignore one or the other. In this paper, we will demonstrate this is indeed the case for scalar and vector quantizer encoding algorithms.

The importance of quantization is paramount, as just about every compression algorithm includes quantization of some form. But while scalar quantizer (SQ) has been widely used, vector quantizer (VQ) has been less popular, partly due to its inherently high encoding complexity. We will show that by optimally dividing the processing into compute and pre-compute for a particular machine, we can improve both encoders' performance. We begin in section 2 and 3 where we review the machine model and its associated optimization framework from [3]. In section 4 and 5, we discuss how the framework is used to optimize SQ and VQ encoders respectively. We present results in section 6.

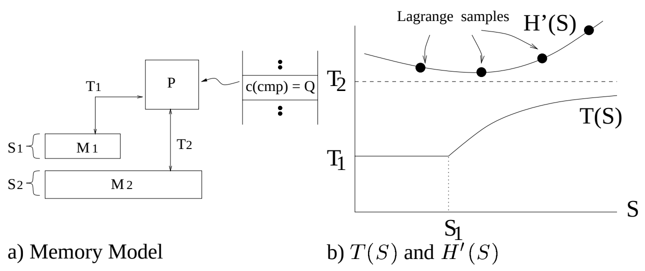


Fig. 1. Machine Model and Optimization Framework

2. DYNAMIC MEMORY MODEL (DMM)

Modern processors use hierarchical memories to enhance performance, where small, fast memories are located near the CPU and larger, slower memories are situated further away. Consequently, the execution speed of a machine instruction that accesses memory depends on the level of memory referenced. The machine model we adopt in Figure 1a reflects this characteristic. If the processor $\mathbf{P}$ accesses a datum residing in level 1 memory $\mathbf{M}_1$ (level 2 memory $\mathbf{M}_2$), it incurs memory access time T_1 (T_2). If the instruction does not involve memory access, then the execution time depends on the complexity of the instruction itself; for example, we denote the cost of a logical comparison (`cmp`) as Q. We assume the size of $\mathbf{M}_1$ ($\mathbf{M}_2$) is S_1 (∞).

Most processors, however, does not have the flexibility of assigning data blocks statically to the processors' memory hierarchy. What data blocks reside in what level of memory hierarchy at any given time depends on the processor's caching policy and the data access patterns. Instead of tracking all this information, we approximate each memory access cost as follows.

Suppose the size of data structures used by an implementation of an algorithm, S, is $\leq S_1$. Then the access time of a desired datum, $T(S)$, is T_1, since all data structures can be loaded into $\mathbf{M}_1$. If $S > S_1$, then we do not know whether the desired datum resides in $\mathbf{M}_1$ or $\mathbf{M}_2$. In this case, we estimate the access time as follows. At any given time, a fraction $\frac{S_1}{S}$ ($\frac{S-S_1}{S}$) of the total number of data blocks will be in $\mathbf{M}_1$ ($\mathbf{M}_2$). So assuming all pieces of data are equally probable, we can estimate memory access cost $T(S)$ of a data memory retrieval as:

$$T(S) = \begin{cases} T_1 & \text{if } S \leq S_1 \\ (\frac{S_1}{S})T_1 + (\frac{S-S_1}{S})T_2 & \text{otherwise} \end{cases} \qquad (1)$$

See Figure 1b of an illustration of $T(S)$.

3. OPTIMIZATION FRAMEWORK (DMMOPT)

Using DMM, we can evaluate the execution cost of an implementation l as follows. We first find the size of the implementation's data structures, $R(l)$. This translates to a memory access cost $T(R(l))$ using (1). Knowing the access cost, we can evaluate the execution cost of l, $H_{T(R(l))}(l)$. If $\mathcal{L}$ denotes the set of implementations in the search space, then the optimization problem is:

$$\min_{l \in \mathcal{L}} \left\{ H_{T(R(l))}(l) \right\} \tag{2}$$

Solving (2) is difficult in general (for the VLC decoding instantiation, see [3] for a formal proof of NP-hardness). The reason is twofold: i) while the cost of a memory access is not known till the entire implementation is constructed, the optimal construction of an implementation depends on the cost of memory access — a chicken-and-egg problem; ii) because the cost evaluation depends on non-linear function $T(S)$, the problem is non-linear. Instead of solving (2) directly, we dissect it into easier pieces.

Suppose we know *a priori* that the total data structure size of the optimal implementation l^* is S^*. To find l^*, we only need to search the subset of implementations with total size S^*. Let $H_{T(S)}(l)$ denote the cost of l when the access cost is fixed at $T(S)$. (2) is then the same as:

$$H'(S^*) = \min_{l \in \mathcal{L}} \left\{ H_{T(S^*)}(l) \right\} \quad \text{s.t. } R(l) = S^* \tag{3}$$

We call $H'(S)$ the *sampling function.* Solving $H'(S^*)$ seems easier, since we have eliminated the problems of mutual dependency and non-linearity. Yet we do not know S^* a priori, and so we need to search through all possible values of S for S^*:

$$\min_{l \in \mathcal{L}} \left\{ H_{T(R(l))}(l) \right\} = \min_{\forall S} \left\{ H'(S) \right\} \tag{4}$$

Graphically, for each S, we solve (4) and obtain a sample point on $H'(S)$. S^* is the minimum point on sampling function $H'(S)$. See Figure 1b for an illustration of $H'(S)$. We are now faced with two new difficulties: i) solving (3) for all S is expensive, ii) (3) itself is still hard since it is a constrained problem.

3.1. Lagrangian Approach for DMM

Given that (3) is difficult, we do not solve (3) directly, but we solve its corresponding Lagrangian instead:

$$\min_{l \in \mathcal{L}} \left\{ H_{T(S)}(l) + \lambda R(l) \right\} \tag{5}$$

The two problems, (3) and (5), are related: if there exists a Lagrange multiplier λ, such that the optimal solution to (5), l^o, satisfies $R(l^o) = S$, then l^o is also optimal to (3).

In general, solving the corresponding unconstrained problem (5) is easier than the original constrained one (3). The problem is that for a given target constraint value S, there may not exist a multiplier λ such that the optimal solution to the Lagrangian (5), l^o, has the property $R(l^o) = S$. In that case, we propose the following procedure called *iterative projection method* that converges to a size constraint value S and a multiplier λ such that $R(l^o) = S$: 1) Initialize S. 2) Iteratively solve (5), adjusting λ each time, until $R(l^o) - S$ is minimized while $R(l^o) \geq S$. 3) If $R(l^o) = S$, done. Else, let $S := R(l^o)$, goto step 2.

Because $R(l^o)$ is inversely proportional to λ, a simple strategy of adjusting λ in step 2 is to use binary search on the real line. Alternatively, a more efficient strategy called *singular value search* can be employed. See [3] for details.

3.2. Lagrangian Sampling

Instead of searching for all possible values S in (4), by finding solutions to (5) using the iterative projection method, we are actually only sampling a relatively small number of points on $H'(S)$, since the method converges to a small subset of points no matter what S is initialized to. We call this phenomenon *Lagrangian sampling*, since each sample point is a solution to the Lagrangian (5). By sampling, however, we may not be able to find the optimal solution $H'(S^*)$; we use the following theorem [3] to bound the error of neighboring sample points from a local minimum point:

Theorem 1 *Let l^* be a locally optimal solution to (2) such that $S^* = R(l^*) \in [S^1, S^2]$, and $H'(S^1)$, $H'(S^2)$ are the two neighboring Lagrangian sample points on $H'(S)$. When S is initialized to S_1 in step 1 of iterative projection method, we can find an optimal solution to (5), l^B, such that one of the* **Bounding Conditions** *is satisfied: 1) ($R(l^B) \geq S^2$) and ($\lambda \geq 0$). 2) ($R(l^B) \leq S^1$) and ($\lambda < 0$). 3) ($S^1 \leq R(l^B) \leq S^2$) and ($\lambda = 0$). The cost of the locally optimal solution l^* is lower bounded by l^B, i.e.: $H'(S^*) = H_{T(S^*)}(l^*) \geq H_{T(S^1)}(l^B)$.*

3.3. Optimization Framework for DMM — DMMOPT

Having developed the above concepts, we are ready to discuss the optimization framework associated with DMM, which we term DMMOPT — it helps us construct a program that finds a near-optimal solution to (2) with *a posteriori* error bound:

> Construct $H'(S)$ by obtaining Lagrangian sampling points. Each Lagrangian sampling point is obtained with the iterative projection method. Among the sampling points, we pick the smallest point as our operating point.

The global error bound is the difference between the best performance sample point and the best performance local bound of all pairs of neighboring sample points.

How the Lagrangian (5) is solved depends on the particular instantiation of DMMOPT. In particular, we will see the SQ instantiation in the next section.

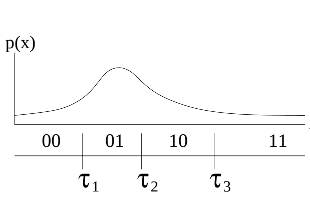
4. SCALAR QUANTIZER

We begin with the non-uniform SQ encoding problem. We first define the search space of implementations $\mathcal{L}$ in the optimization problem. We then discuss the program we developed using DMMOPT that solves SQE-OPT approximately — solving the Lagrangian (5) given λ. We then reduce the complexity of the program by tree pruning. Singular value search used in step 2 of the iterative projection method can be found in [3].

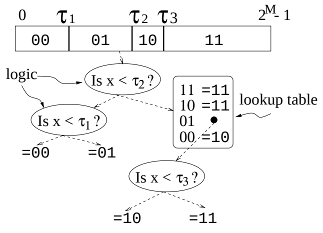
4.1. Problem Formulation

Very often a representation of a signal in a computer needs to be compressed for space-limited storage or for bandwidth-limited transmission. One compression technique that can perform this many-to-few bits mapping of signal is the non-uniform M-to-N bit SQ, where a scalar quantity of M bits is mapped to one of 2^N partitions. Figure 2a shows an example of a M-to-2 bit non-uniform SQ. The optimal design of non-uniform SQs — the selection of partition boundary set $\Gamma = \{\tau_1, \ldots, \tau_{2^N-1}\}$ that minimizes distortion of reconstruction signals — is well-studied [4]. The resulting N-bit quantizer is commonly called the *Lloyd-Max Quantizer*.

Finding an efficient implementation for the M-to-N-bit SQ encoder can involve a tradeoff between computation and memory. Two simple encoding implementations illustrate the extremes of



a) Non-uniform SQ



b) Hybrid Encoding Impl.

Fig. 2. Non-uniform SQ and Encoding Implementation

computation and memory tradeoffs. The first one minimizes computations by performing a single M-bit table lookup, where the resulting table entry contains the corresponding N-bit partition index. However, this requires a memory store of size 2^M, and memory access can be slow if M is large and 2^M elements cannot fit into $\mathbf{M}_1$. An alternative implementation minimizes memory usage by asking a sequence of logic statements "*Is* $x < \tau_i$*?*" until the correct partition has been identified. This corresponds to a binary decision tree of height $h \geq N$. If N is large, the sequence of questions required is long and the implementation is slow. A natural question is: what is the optimal hybrid scheme, using a combination of lookup tables and logic, that minimizes the encoding time? An example of a hybrid implementation is shown in Figure 2b.

Having described the search space of implementations, we can formally define the optimization problem of finding the optimal hybrid implementation for the M-to-N bit SQ encoder, denoted as SQE-OPT, as follows: what is the fastest implementation of a M-to-N bit SQ encoder, given input x distribution $p(x)$, partition boundary set $\Gamma = \{\tau_1, \ldots, \tau_{2^N-1}\}$, and parameters of DMM?

To solve SQE-OPT, we develop a program that finds an implementation with near-optimal processing mixture of compute and pre-compute by applying DMMOPT. We begin development of the program in the next section.

4.2. Program Development

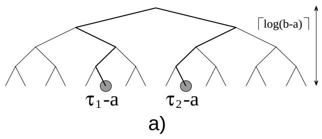
We begin with the definition of the cost function that we are minimizing: let $f(a, b)$ be the minimum Lagrangian encoding cost (5) — $\min_{l \in \mathcal{L}} H_{T(S)}(l) + \lambda R(l)$ — given input $x \in [a, b)$. The optimal initial operation for this input range can potentially be a logic or table lookup, resulting in cost $f_l(a, b)$ or $f_t(a, b)$ respectively:

$$f(a, b) = \min \{ \ f_l(a, b), \ f_t(a, b) \ \} \tag{6}$$

For the logic case, we can choose among all τ_i values that are in the range (a, b) to check against input x. The result of the check is a partition of the original interval into $[a, \tau_i)$ and $[\tau_i, b)$. Let $p(a, b)$ denote the probability that $x \in [a, b)$, and Q denote the cost of a logic operation. We can write $f_l(a, b)$ as:

$$f_l(a, b) = p(a, b)Q + \min_{\tau_i \in (a,b)} [f(a, \tau_i) + f(\tau_i, b)] \tag{7}$$

For the table lookup case, there is an initial cost of $p(a, b)T(S)$. The index used in a h-bit table lookup operation is the h left-most bits of the number $x - a$. The largest value $x - a$ can take on, given $x \in [a, b)$, is $b - a - 1$. Hence, the number of bits needed to describe $x - a$, or the maximum height of a lookup table, is $\lceil \log_2(b - a) \rceil$. For each table height h, the table operation divides the range $[a, b)$ into smaller ranges of width $m = 2^{\lceil \log_2(b-a) \rceil - h}$ each. The number of these smaller ranges, n, is determined by the largest number that the h most significant bits of $x - a$ can take on. The associated penalty $\lambda R(l)$ in (5) is therefore λn. The following equations formalize this:



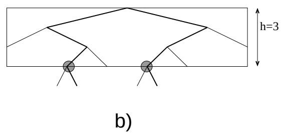


Fig. 3. Tree Pruning Example

$$\begin{aligned} f_t(a, b) &= p(a, b)T(S) + \min_{1 \leq h \leq \lceil \log_2(b-a) \rceil} \Big[\lambda n + \\ & \quad \sum_{i=1}^{n-1} f(a + m(i-1), a + m(i)) + f(a + m(n-1), b) \Big] \\ m &= 2^{\lceil \log_2(b-a) \rceil - h} \qquad n = \left\lfloor \frac{b - a - 1}{m} \right\rfloor + 1 \end{aligned} \tag{8}$$

This recursion is grounded in a base case where there is no τ_i in range (a, b), meaning the input x can only be in one partition:

$$f(a, b) = 0 \qquad \text{if} \ \nexists \, \tau_i \in (a, b) \tag{9}$$

The value of $f(0, 2^M)$ then yields l^o, the optimal solution to (5) given λ.

4.3. Tree Pruning

While we can solve (5) given λ with call to $f(0, 2^M)$ using recursive calls (6), (7) and (8), the running time is exponential — the call has 2^M recursive calls when a M-bit table lookup operation is tested for $f_t(0, 2^M)$. This means a single execution of (8) has running time $O(2^M M)$. However, we can alter the program to reduce its complexity by pruning off some of the recursive calls in (8). When performing an h-bit table lookup for a given range $[a, b)$, there will be n recursive calls according to (8) corresponding to n branches of a binary tree — the leaves of the tree are the possible values of $x - a$. Figure 3a shows a binary tree for a range $[a, b)$ where $\lceil \log_2(b - a) \rceil = 4$. It also highlights the branches that corresponds to each $\tau_i \in [a, b)$. We call them *τ-branches*.

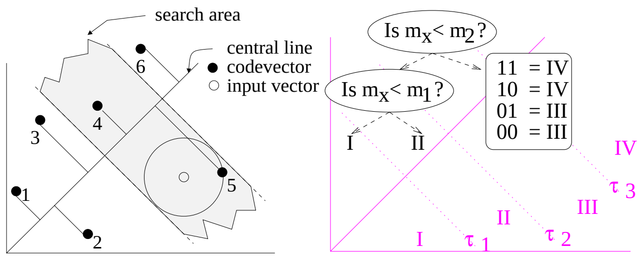
If we perform a 3-bit table lookup on the same range $[a, b)$, as shown in Figure 3b, we see that unless input x follows one of the two τ-branches, we know immediately which partition the input falls into. If input x follows one of the two τ-branches, then further operations are needed to determine the correct partitions. We can generalize the above observation and say that the only recursive calls needed in (8) are these τ-branches. So if we prune off the non-τ-branches during execution of (8), the complexity of (8) is $O(2^N M)$; it is now polynomial in size of the input.

5. VECTOR QUANTIZER

Instead of developing a new optimization, we leverage on the program we developed for SQE-OPT and use it to speed up a pre-processing step of an established VQ encoding technique, called *equal-average nearest neighbor search* (ENNS). We first describe how ENNS works, then discuss how the program developed for SQE-OPT can be used to improve ENNS.

5.1. Equal-average Nearest Neighbor Search

ENNS [5] has been shown to lower VQ encoder's complexity in the average case for image data. The key observation is that there

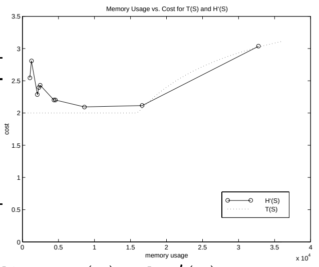


a) Equal-avg VQ Encoding b) Hybrid Encoding Implementation

Fig. 4. Equal-average VQ and Hybrid Encoding

Parameters	Values
S_1	$16k$
T_1	2 cycles
S_2	∞
T_2	4 cycles
Q	3 cycles
$p(x)$	A: $N(8000, 1600^2)$

a) Parameters



b) exp. $T(S)$ and $H'(S)$

Fig. 5. Experiment for SQ

are strong correlations among input vector's individual components for image data. As a result, the majority of the input vectors are distributed along the central line $l = \{\mathbf{x} | x_1 = \ldots = x_k\}$. ENNS proposes that we first pre-sort the codevectors according to their means, then during actual algorithm execution to find the nearest neighbor to input vector $\mathbf{x}$, we can successively eliminate potential nearest neighbors by using this bound:

$$d(\mathbf{x}, \mathbf{y}) \geq \sqrt{k} |m_{\mathbf{x}} - m_{\mathbf{y}}| \tag{10}$$

where $d(\mathbf{x}, \mathbf{y})$ is the l_2 distance between input vector $\mathbf{x}$ and potential nearest neighbor $\mathbf{y}$, k is the number of dimensions, and $m_{\mathbf{x}}$ and $m_{\mathbf{y}}$ are mean of $\mathbf{x}$ and $\mathbf{y}$ respectively.

An example is shown in Figure 4a, where we first test codevector $\mathbf{y}_5$ and compute $d(\mathbf{x}, \mathbf{y}_5)$. We can then eliminate any vector $\mathbf{y}_i$ whose mean $m_{\mathbf{y}_i}$ is such that $\sqrt{k}|m_{\mathbf{x}} - m_{\mathbf{y}_i}| \geq d(\mathbf{x}, \mathbf{y}_5)$. Geometrically, we eliminate all codevectors that lie outside the gray strip that encloses the circle in Figure 4a. In this example, we eliminate $\mathbf{y}_1$, $\mathbf{y}_2$, $\mathbf{y}_3$ and $\mathbf{y}_6$.

For ENNS to be most effective, the initial candidate codevector should have mean closest to the input vector. To this end, ENNS uses a binary decision tree to first find this closest-input-mean codevector. To speed up this initial search, we use the program for SQE-OPT to find a near-optimal implementation that finds this closest-input-mean codevector.

6. RESULTS

To evaluate the performance of our developed algorithm for SQ, we conducted experiments to compare our optimized implementation to an implementation that use a binary decision tree to encode SQ [4]. For our experiments, we use parameters in Figure 5a, which are estimates of our test machine, a Pentium II 266 MHz processor. It has 16kbyte L1 cache (50-50 split of the 32kbyte data-instruction cache). The input distribution $p(x)$ of Figure 5a is an arbitrarily chosen Gaussian distribution.

(M,N)	impl.	speed
(15,2)	logic-only	3.738 mil/s
(15,2)	hybrid	4.790 mil/s
(15,4)	logic-only	2.484 mil/s
(15,4)	hybrid	4.597 mil/s

a) Comparison of SQ Encoders

size	impl.	time
8	logic-only	.280s/lena
8	hybrid	.255s/lena
16	logic-only	.465s/lena
16	hybrid	.440s/lena
32	logic-only	.465s/lena
32	hybrid	.440s/lena
64	logic-only	.935s/lena
64	hybrid	.895s/lena

b) Comparison of VQ Encoders

Fig. 6. Comparisons of SQ, VQ Encoders

Using the chosen parameters, we generated $T(S)$ (bottom) and $H'(S)$ for 15-to-4 bit SQ, with input distribution A (top) in Figure 5b. The partition boundary set Γ was generated using Lloyd's algorithm.

To compare our hybrid table lookup-logic SQ encoder to a binary decision tree SQ encoder, we generated a workload of 20 million input samples according to the input distribution and encoded them 10 times with each implementation to find an average speed for each case.

For input distribution A, 15-to-2 bit (15-to-4 bit) SQ encoders excluding I/O access time, we see a 28.14% improvement (85.06%) over the binary-tree encoder. As N increases, the improvement of the hybrid encoder over the binary-tree encoder increases. This is expected, since the height of the binary decision tree for binary-tree encoders is larger when as N increases.

We performed experiments to show that ENNS has faster encoding speed when the implementation found by the program solving SQE-OPT is first used to find the codevector with the closest mean to input vector. To generate various VQ codebooks for testing, we used 512*512 gray scale images of `Lena`, `Baboon` and `Tiffany` as training data, and constructed codebooks of size 8, 16, 32 and 64 for dimension 4 using the generalized Lloyd algorithm [4]. We then compared the encoding speed of ENNS using a binary decision tree and ENNS using our generated implementation when encoding the `Lena` image. Excluding I/O access time, we achieved speed improvement of 9.83%, 5.67%, 6.65% and 4.47% respectively for the four codebook sizes.

First, we observe that the improvement for VQ is not as drastic as SQ. This is expected, since we are speeding up only the initial search for closest codevector mean, and the VQ encoding algorithm needs to perform other tasks like computing distortion between input vector and potential candidate vectors. Second, we see that as the size of the codebook increases, the percentage improvement decreases. The reason is that ENNS is increasingly ineffective in ruling out candidate codevectors as the codebook size grows. The bulk of the computation then becomes the computations of distortion between input vector and candidate vectors, and the speed improvement of initial search for closest codevector mean is diminished in the overall picture.

7. REFERENCES

[1] K.Lengwehasatit, A.Ortega, "Distortion/Decoding Time Tradeoffs in software DCT-based Image Coding," *ICASSP 97*, 1997.

[2] V. Goyal and M. Vetterli, "Computation-Distortion Characteristics of Block Transform Coding," *ICIP 97*, pp.2729-2732, 1997.

[3] G.Cheung, "Memory Model based Algorithmic Computational Optimizations for Signal Processing," PhD thesis, May 2000.

[4] A.Gersho, R.Gray, *Vector Quantization and Signal Compression*, Kluwer, 1992.

[5] L.Guan, M.Kamel, "Equal-average Hyperplane Partition Method for Vector Quantization of Image Data," Pattern Recognition Letter 13 (1992) 693-699.