Vector Quantization Parallelization

Hai Zhu

New York University

1 Introduction

Quantization is the process of representing a large set of input values with a much smaller set. In signal processing and image processing, Vector Quantization is a classical quantization which extends the scalar quantization to multi-dimensional space. It is widely used in many applications such as data compression, data correction, pattern recognition, and density estimation. This project proposes a parallel implementation of Vector Quantization for image compression processing.

Vector Quantization works by dividing a large set of source vectors into groups having approximately the same number of vectors closest to them [1] and each group is represented by the vector offering the lowest distortion among all the vectors in a predesigned set of vectors (codebook). For image compression, a large amount of computation is required. The most time-consuming factor on compression is "nearest neighbor search" [2] to find the vector nearest to a source input among a large number of predesigned vectors.

Vector Quantization involves the comparison of all distances between a source vector and vectors in a codebook. Due to the fact that comparisons of distances between each source vector and all vectors in the same codebook are independent in the processes of Vector Quantization, parallelization of the algorithm can be achieved.

The work of Vector Quantization can be divided into three parts as codebook generation, encoding procedure and decoding. Since codebook generation is an important part of Vector Quantization, which involves most algorithms and computations in the processes of VQ image compression, this report mostly focuses on the parallel implementation of LBG algorithm, one of codebook generation algorithms.

2 Background

This section presents the theoretical background of image compression, vector quantization, and codebook generation.

2.1 Image compression

The objective of data compression is reducing the number of bytes stored or transmitted, without an appreciable loss of information. There are two types of data compression, lossless and lossy. In lossless data compression, the original information can be completely recovered from the compressed data. However, since some error was introduced, the original data cannot be completely recovered in lossy data compression.

In Vector Quantization, compression is achieved by transmitting or storing the indices associated to the vectors in the codebook instead of the vectors because of the far fewer bits required for the indices.

2.2 Vector Quantization

Vector Quantization is one of various data compression techniques. It exploits the correlation that exists between neighboring data of the input values. The quantization can be divided into two distinguishable part: a lossy encoder and a reproduction decoder. The following Figure 1 [3] shows the principle of the resulting encoder and decoder.

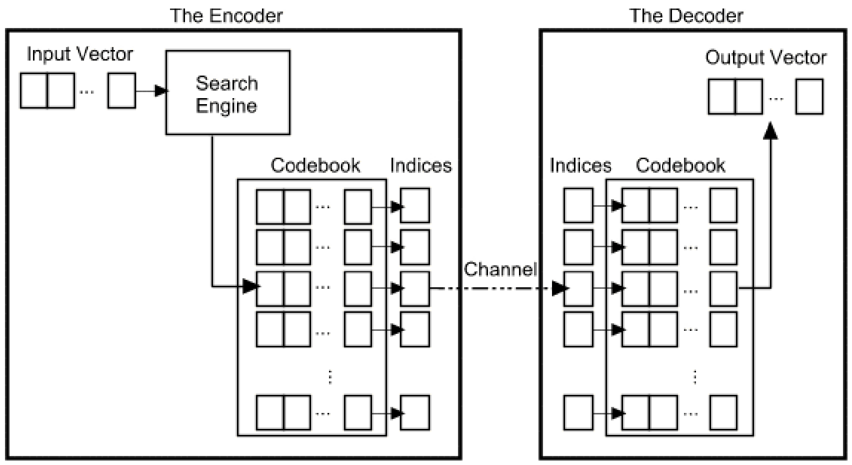

Figure 1: The principle of the encoder and the decoder used in vector quantization.

The image to be encoded is broken into blocks or vectors. The encoder maps each input vector to one of a finite set of predesigned vectors. This set of predesigned vectors is the codebook and the vectors in it is called codevectors. The encoder finds the closest matching codevector from the codebook by calculating the distortion between the input vector and each of the codevectors and outputs the index of that codevector. Then the index is transmitted to the decoder. The decoder has the same codebook as encoder and uses the codevectors identified by the indices to piece the image back.

2.3 Codebook and codebook generation

Codebook consist of a set of vectors, called codevectors. VQ algorithms use codebook to map an input vector to the codevector closest to it, thus codebook is the representative of the input data.

Codebook generation plays a fundamental role in the performance of data compression based on VQ, because the image quality is the result of the comparison between input vectors with codebook. A good codebook should be able to represent the variety of source vectors within the minimum distortion.

Codebook generation problem can be stated as follows:

Given a vector source with its statistical properties known, given a distortion measure, and given the number of codevectors, find a codebook and a partition which result in the lowest average distortion.

Due to the need for multi-dimensional integration, the design of a VQ is considered to be a challenging problem. However, since in 1980, Linde, Buzo, and Gray proposed a VQ design algorithm based on a training sequence [4], which bypasses the need for multi-dimensional integration, vector quantization has been widely studied.

LBG Algorithm [5]: This is an iterative clustering descent algorithm which yields a locally optimum codebook for a given source or distribution. Its basic function is to minimize the distance between the vectors in the training sequence and the codevector that is closest to it.

Since LBG algorithm is the most famous and widely used algorithm for vector quantization, the algorithm of vector quantization chosen for parallelizing is the LBG algorithm in this project. The next section describes the LBG algorithms in more details.

3 Design problem and Algorithm implementation

3.1 Distortion

The codebook used for the encoding and decoding is optimal in the sense that the overall distortion is minimal. There the distortion between an input vector and a codevector is measured by the mean squared error. The process of calculate a distortion described as follows [4]:

1. Given a training sequence consisting of M source vectors and each vector is k dimensional:
   $T = \{\mathrm{X}_1, \mathrm{X}_2, \dots, \mathrm{X}_\mathrm{M}\}; \quad \mathrm{X}_\mathrm{m} = (\mathrm{x}_{\mathrm{m},1}, \mathrm{x}_{\mathrm{m},2}, \dots, \mathrm{x}_{\mathrm{m},\mathrm{k}}),\ \mathrm{m} = 1,2,\dots,\mathrm{M}$
   The training sequence can be obtained from input source data. M should be sufficiently larger so that all the statistical properties of the source can be captured by training sequence.
2. A codebook with N codevectors. Each codevector is k dimensional
   $\mathrm{C} = \{\mathrm{c}_1, \mathrm{c}_2, \dots, \mathrm{c}_\mathrm{N}\}; \quad c_n = (\mathrm{c}_{\mathrm{n},1}, \mathrm{c}_{\mathrm{n},2}, \dots, \mathrm{c}_{\mathrm{n},\mathrm{k}}),\ \mathrm{n} = 1,2,\dots,\mathrm{N}$
3. Let S_n be the encoding region associated with codevector c_n and $\mathrm{P} = \{\mathrm{S}_1, \mathrm{S}_2, \dots, \mathrm{S}_\mathrm{N}\}$ denote the partion of the space. If the source vector X_m is in the region S_n, then its approximation (denoted by $Q(X_m)$) is c_n:
   $$\mathrm{Q}(\mathrm{X}_\mathrm{m}) = \mathrm{c}_\mathrm{n}, \qquad \text{if } \mathrm{X}_\mathrm{m} \in \mathrm{S}_\mathrm{n}$$
4. Using squared-error distortion measure, the average distortion is:
   $$\mathrm{D}_{\mathrm{ave}} = \frac{1}{\mathrm{Mk}} \sum_{\mathrm{m}=1}^{\mathrm{M}} \|\mathrm{X}_\mathrm{m} - \mathrm{Q}(\mathrm{X}_\mathrm{m})\|^2 = \frac{1}{\mathrm{Mk}} \sum_{\mathrm{m}=1}^{\mathrm{M}} \sum_{j=1}^{k} (\mathrm{X}_{\mathrm{m}j} - c_{n\mathrm{j}})^2$$

3.2 LBG algorithm for codebook generation

The LBG algorithm is an exhaustive search algorithm. The algorithm requires an initial codebook $C_{0.}$ This initial codebook can be obtained by the splitting or random chosen method. In this project, we use splitting method. An initial code vector is obtained from the average of the entire training sequence and then this codevector is split into two. The iterative algorithm is run with these two vectors as the initial codebook. Then two codevectors are split into four and the process is repeated until the desired number of codevector is obtained. Split Codebook is the average amount of Training Vectors population. The result is reducing scatter data better than random sample. The algorithm is summarized below [4].

1. Given training set T and fixed $\epsilon > 0$ to be a “small” number.
2. Let N=1 and

$$c_1{}^* = \frac{1}{M} \sum_{m=1}^{M} X_m$$

   Calculate

$$D_{ave}{}^* = \frac{1}{Mk} \sum_{m=1}^{M} \|X_m - c_1{}^*\|^2$$

3. Splitting: For i=1, 2, …, N,

$$c_i{}^{(0)} = (1+\epsilon)c_i{}^*; \ \ c_{N+i}{}^{(0)} = (1-\epsilon)c_i{}^*$$

   Set N=2N
4. Iteration: Let $D_{ave}{}^{(0)} = D_{ave}{}^*$. Set the iteration index i=0
   i. For m= 1, 2, …, M, find the minimum value of $\left\|X_m - c_n{}^{(i)}\right\|^2$ (n=1,2,…, N)
      Let n^* be the index which achieves the minimum.
      Set $Q(X_m) = c_{n*}{}^{(i)}$
   ii. For n=1, 2, …, N, update the codevector (centroid)

$$c_n{}^{(i+1)} = \frac{\sum_{Q(X_m)=c_n{}^{(i)}} X_m}{\sum_{Q(X_m)=c_n{}^{(i)}} 1}$$

   iii. Set i=i+1
   iv. Calculate

$$D_{ave}{}^{(i)} = \frac{1}{Mk} \sum_{m=1}^{M} \|X_m - Q(X_m)\|^2$$

   v. If $D_{ave}^{(i)} - \frac{D_{ave}^{(i)}}{D_{ave}^{(i-1)}} > \epsilon$, repeat from step(i)
   vi. Set $D_{ave}^* = D_{ave}^{(i)}$. For n=1, 2, …, N, set $c_n^* = c_n^{(i)}$ as the final codevectors
5. Repeat Steps 3 and 4 until the desired number of codevector is obtained.

3.3 Complexity

In code generation and encoding process, the nearest neighbor search is most time-consuming factor. The simplest way to find the nearest neighbor codevector is to compute for a given source vector the distortion for each of the N possible codevectors and to choose the codevector with the minimal distortion. The exhaustive search algorithm and the partial distance search algorithm exploit this idea.

The complexity can be easily derived from the algorithm. We can see given M input vectors, N codevectors, and each vector is in k dimensions, then the number of multiplies is kMN, the number of additions and subtractions is MN((k - 1) + k) = MN(2k-1), and the number of comparison is M(N-1) . The number of total operation is 3MN (k-1).

e.g. For an 256x256 image, a codebook of 256, and vectors in 16 dimensions, there is 1,048,576 times Squired Euclidean Distance (including 32,505,856 additions and 16,777,216 multiplication).

4 Parallel Implementation

Although VQ offers more compression for the same distortion rate as scalar quantization and PCM, yet is not as widely implemented. This due to two things. The first is the time it takes to generate the codebook, and second is the speed of the search. So to parallelize VQ for getting better performance will help the applicability of VQ algorithm.

Due to the fact that all input vectors are compared with the same codebook independently, division of work is clearly visualized and parallelization of the algorithm can be achieved. Taking the advantage of SIMD processors, we divided input training data into chunks and share a single copy of codebook in the primary memory of each processors.

The following illustrates the psudo-code of parallelized code in codebook generation.

```
Input: training data T with M vectors, each vector is k dimensional, and the
desired number of codevector N
Output: codebook with N codevectors, each codevector is k dimensional.

Initialize codebook $c_0$ = centroid of all training data
while (c.size < N){
  Split codebook:
    c.size = 2*c.size;
    foreach $c_i$=$c_{i/2}$+e; $c_{i-1}$=$c_{i/2}$
  while(ave_dist>threshold) {
    :parallel execution at for loop
    foreach x in T do
      mindist<-min(dist(x,c0),dist(x, c1),…);
      index <- codebookindex(mindist);
      $centroid_{index}$+=x;
      $count_{index}$ ++;
      dist+=mindist;
    end
    update c = centroid/count
    ave_dist =dist/t.size
  }
}
```

In above code, the parallelized partial is the algorithm for nearest neighbor search, which also involved in codebook searching. Significant speedup can be achieved when this partial is parallelized because its complexity.

5 Experiments and Results

This project implemented VQ algorithm is sequential and parallel model. The parallelization implemented by using OpenMP. For obtaining the complete perspective of our experiments, we compared the results of the sequential program with the results of parallel program with varying number of threads. The test platform is crunchy1.cims.nyu.edu, which consists of 4 16-Core 2.1GHz AMD Opteron 6272 CPUs and 256GB memory.

We use a training set taken from 512*512 8bpp monochrome image of Lena.pgm. The image is partitioned into 4*4 blocks and the blocks are used to design the codebook by LBG algorithm. The desired size of codebook is 256.

Our program parallelized several parts of image compression processes, including codebook generation, encoding and decoding. Although the codebook generation is the most time-consuming process, it's difficult to measure the performance of codebook generation because of the complexity of codebook generation, increasing size of codebook and the amount of calculation somehow decided by the distortion threshold. So we use elapsed time to make some approximate evaluation.

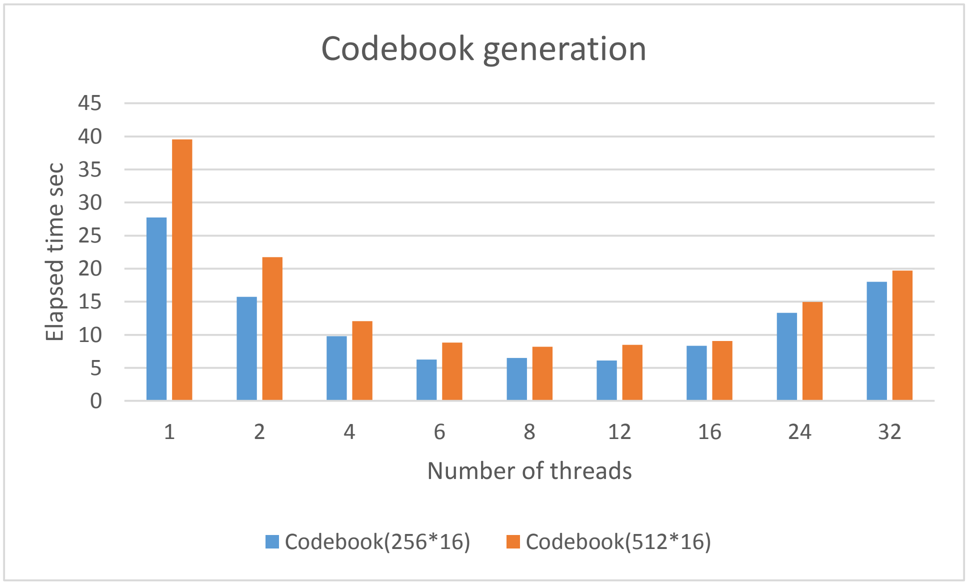

Figure 2 Elapsed time of Codebook generation

The above figure shows that speedup can be achieved if the codebooks are generated in parallel. However the elapsed time began to increase after the number of threads is more than 8. The reason is that there is a critical section in the parallelized code, the probability of entering this section increases along with the increase of the number of threads. The overhead of critical section will damage the speedup obtained by the increase number of processors.

In the encoding process, the nearest neighbor search is a fix scale computation and it is a core computation of the vector quantization. We test the performance of this part on varying number of threads and different problem sizes, namely the Codebook 256 *16, 512*16, 256*64, and 512*16, where 256 and 512 are the number of codevectors in codebook, 16 and 64 are dimension of codevector. The operation in this measurement takes account into all the operations involved in nearest neighbor search, including additions, subtractions multiplies and comparisons. The number of total operation should be 3MN (k-1). Figure 3 shows the performance of nearest neighbor search given various codebooks and various number of threads.

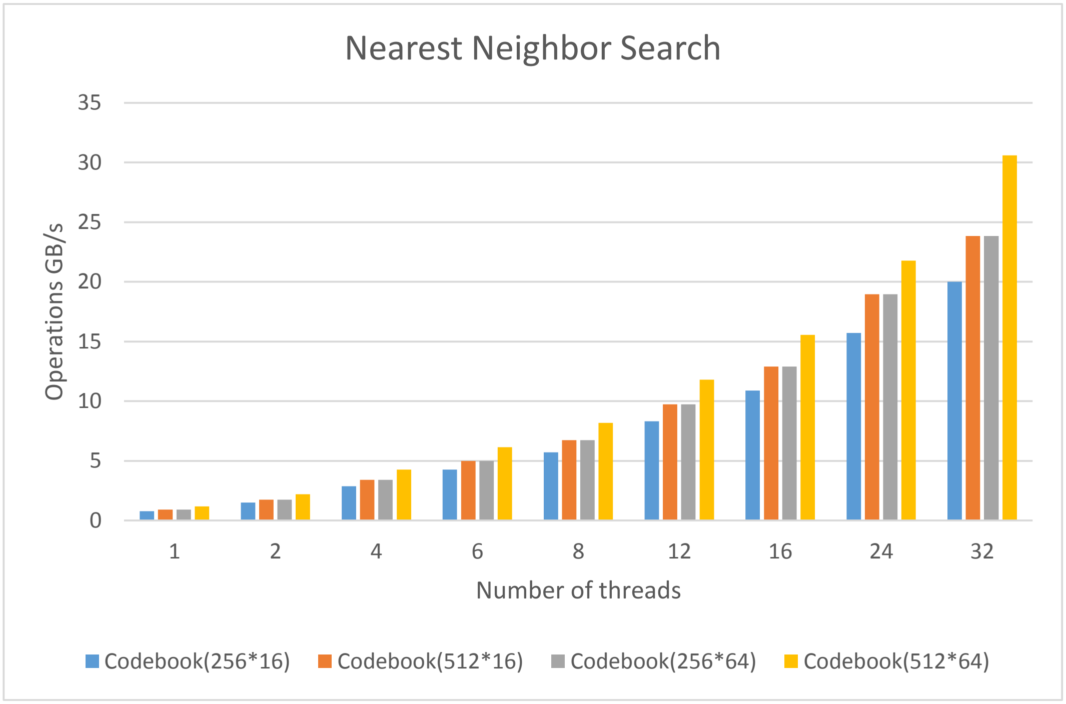

Figure 3 Performance of nearest neighbor search

As we can see, the algorithm scales very well on the increase of the number of threads. The figure also shows that larger number of dimensions will obtain the better performance. The reason is that the large dimensions will block the image into fewer groups, thus the number of iteration decreased and the workloads of each threads in the parallel execution is decreased too. However, the large dimension usually offering higher distortion, so it is not practical in vector quantization. The figure also displays that the larger size of codebook with same dimension gains better performance. Since the parallel algorithm in our code is divided the input vectors (training set) into chucks to exploit the benefits of parallelization, the increase of the size of codebook will not change the granularity of parallelization but only increases the amount of computation in each parallelized unit. So the appropriate amount of computation (workload) in a parallelized unit will exploit the benefits of parallelization completely and get better performance.

6 Appendix – Source Code

6.1 Availability

The source code is available on the forge repository: http://forge.tiker.net/p/hpc12-final-hz575

6.2 Building

The included makefile should build the code on any generic Linux system using GCC compilers. The OpenMP version should be 3.0 or above.

There are two parameters for defining different level of information output.

DEBUG for more details level output, including parameters of structure or function, performance statistics of code snippets.

DEBUG0 for more high level output, including the performance statistics of the most important algorithms in VQ, such as overall codebook generation fraction (Func img_codebook_p()), nearest neighbor search (codebook mapping), and decoding (decompression)

e.g. make DEBUG0=1

Running generated program will output the performance statistics of the most algorithms.

6.3 Running code

For the convenience, included test.sh can be used to run program on various number of threads in a batch model. Modify the contents of the bash script can change the parameters of program.

Executing the program without any arguments or invalid arguments will give a detail usage or an explanation.

The following table shows usage and the parameters of program with explanations.

usage	`vqdemo cmd [-i imagefile][-o outfile][-c codebook][-a compressedfile][-s cbsize][-w width][-h height][-n numthr][-?]`
Parameters	**Notes**
cmd	The operation, could be codebook, encode, decode, and all.
imagefile	Image file to be used for training or compression.
outfile	Decompressed image file.
codebook	Codebook file to be used or saved.
compressedfile	Compressed file, to be used or saved.
cbsize	Size of the codebook.
width	Width of the block of pixels making up a vector.
height	Height of the block of pixels making up a vector.
numthr	The number of threads

Eaxamples:

$ vqdemo codebook [-i imagefile][-c codebook][-s cbsize][-w width][-h height][-n 4]

- Generate a codebooke using given imagefile and save as codebook.

$ vqdemo all [-i imagefile][-o outfile][-s cbsize][-w width][-h height]

- Generate codebook with given imagefile, compress image, and save decompressed image as outfile. The generated codebook has cbsize codevectors.

References

[1] "Vector quantization," [Online]. Available: http://en.wikipedia.org/wiki/Vector_quantization.

[2] K. Kobayashi, M. Kinoshita, M. Takeuchi, H. Onodera, K. Tamaru, "A Memory-based Parallel Processor for Vector Quantization," in *Solid-State Circuits Conference, 1996. ESSCIRC '96.*, 1996.

[3] "Vector quantization," [Online]. Available: http://www.mqasem.net/vectorquantization/vq.html.

[4] "Vector Quantization," [Online]. Available: http://www.data-compression.com/vq.html.

[5] Y. Linde, A. Buzo, and R. M. Gray, "An algorithm for vector quantizer design," *IEEE Trans. Commun.,* Vols. COM-28, pp. 84-95, 1980.