Neural Comput & Applic (2011) 20:1035–1042
DOI 10.1007/s00521-010-0403-7

ICONIP2009

A GPU implementation for LBG and SOM training

Yi Xiao · Chi Sing Leung · Tze-Yui Ho ·
Ping-Man Lam

Received: 6 May 2010 / Accepted: 3 June 2010 / Published online: 30 July 2010
© Springer-Verlag London Limited 2010

Abstract Vector quantization (VQ) is an effective technique applicable in a wide range of areas, such as image compression and pattern recognition. The most time-consuming procedure of VQ is codebook training, and two of the frequently used training algorithms are LBG and self-organizing map (SOM). Nowadays, desktop computers are usually equipped with programmable graphics processing units (GPUs), whose parallel data-processing ability is ideal for codebook training acceleration. Although there are some GPU algorithms for LBG training, their implementations suffer from a large amount of data transfer between CPU and GPU and a large number of rendering passes within a training iteration. This paper presents a novel GPU-based training implementation for LBG and SOM training. More specifically, we utilize the random write ability of vertex shader to reduce the overheads mentioned above. Our experimental results show that our approach can run four times faster than the previous approach.

Keywords Graphics processing units · Vector quantization · LBG, SOM

1 Introduction

Vector quantization (VQ) [1, 2, 3] is an effective technique applicable in a wide range of areas, such as image compression [4], video compression [5, 6, 7], and bioinformatics [8]. The popularity of VQ comes from its low decoding complexity. However, the computational complexity required for finding a useful codebook is very high. There are some suboptimal searching methods [3, 9, 10] to improve the encoding speed. In [3 and 9], data structures are introduced to the codebooks after it is trained, e.g. Tree-structure VQ [3] and K-D tree VQ [9]. In contrast, [10] achieves the same purpose by using the property codebook, e.g. the ordering property of self-organizing map (SOM) [11]. Although these methods can improve the encoding speed, they are not suitable for codebook training, which is the more time-consuming procedure.

The current generation of consumer-level GPUs has evolved to a stage that supports parallel processing and high-precision computation [12]. Moreover, GPUs become increasingly suitable for general purpose computations, such as sparse matrix solving [13], image-based relighting [14], wavelet transform [15, 16], and neural network simulation [17].

At the time when only non-programmable GPUs are available, there were still some attempts to use GPUs for accelerating the general purpose computations. Bohn [18] proposed an implementation for the online mode training of SOM based on GPUs. However, GPUs cannot handle sequential operations efficiently, which is essential for the online SOM training.

Recently, Takizawa [19] proposed a CPU–GPU co-processing solution for LBG algorithm, where GPU is mainly for Euclidean distance calculation and CPU is for updating the codebook. In their implementation, a training iteration consists of a number of rendering passes, where each rendering pass is associated with a codevector. Within a rendering pass, the temporary nearest codevector index, as well as its corresponding distance, is updated. Afterward, all

Y. Xiao · C. S. Leung · T.-Y. Ho (✉) · P.-M. Lam
Department of Electronic Engineering,
City University of Hong Kong, Kowloon Tong, Hong Kong
e-mail: ee_hty@hotmail.com

rendering passes in a training iteration, the nearest codevector indices, available in the GPU, are transferred to the main memory. Then, the codebook is updated sequentially by CPU. By using their GPU approach, the LBG algorithm can be accelerated by 35 times, compared with a decent CPU implementation. However, there are two bottlenecks in the Takizawa's approach. The first one is the large amount of data transfer between CPU and GPU, which is equal to the number of training vectors. The second one is the large number of rendering passes within a training iteration.

As GPUs evolve, many of the hardware limitations at the time of the Takizawa's method [19] had been relieved. For example, the number of fragment processing units in GPUs was more than that of vertex processing units at that time, but they are unified now. We can use either vertex shader or fragment shader for the computational intensive operations. Also, the maximum number of instructions of shader is greatly increased. As a result, handling only one codevector in one rendering pass to shorten the number of instructions is no longer necessary. Furthermore, vertex shader has an advantage over fragment shader that it allows random write.

This paper presents an unified GPU implementation for LBG and SOM training algorithms based on the random write ability of vertex shader. The proposed scheme uses GPU as the main computing processor, instead of as a co-processor. In the our approach, all the computations are carried out within GPU, so the data transfer between CPU and GPU is minimized. Moreover, within an iteration, there is only one rendering pass, and the overhead between rendering passes is minimized. Our experimental results show that for the LBG algorithm, our approach can run four times faster than the Takizawa's approach. In addition, by controlling point size in the vertex shader, our approach is able to handle the SOM algorithm.

The rest of this paper is organized as follows. In Sect. 2, we briefly describe the background of GPU, LBG, and SOM algorithms. In Sect. 3, the implementation of both algorithms on GPU is presented. The performance of our scheme and comparison are given in Sect. 4. Section 5 presents our conclusion.

2 Background

2.1 LBG

A VQ model contains a codebook $\mathbb{Y} = \{\vec{c}_1, \ldots, \vec{c}_M\}$ in $\Re^k$. Given an input vector $\vec{x}$, the output is an index i^* whose corresponding codevector $\vec{c}_{i*}$ is the closest codevector to $\vec{x}$. Given a dataset $\mathbb{D} = \{\vec{x}_1, \ldots, \vec{x}_N\}$ in $\Re^k$ and the initial codebook $\mathbb{Y}(0) = \{\vec{c}_1(0), \ldots, \vec{c}_M(0)\}$, the LBG training is summarized as follows:

1. $t = 0$.
2. Set $\Omega_i = \varnothing$, for all $i = 1, \ldots, M$.
3. For each training vector $\vec{x}_j$, find out the nearest codevector $\vec{c}_{i^*}(t)$. Put $\vec{x}_j$ into the subset Ω_{i^*}. Therefore, Ω_{i^*} can be expressed as

$$\Omega_{i^*} = \left\{ \vec{x}_j \middle| \|\vec{x}_j - \vec{c}_{i^*}\|^2 \le \|\vec{x}_j - \vec{c}_i\|^2, \forall i \ne i^* \right\}. \tag{1}$$

Note that the subsets Ω_is are non-overlapped.

4. Update the codebook:

$$\vec{c}_i(t+1) = \frac{1}{|\Omega_i|} \sum_{\vec{x}_j \in \Omega_i} \vec{x}_j, \quad \forall i. \tag{2}$$

5. Set $t = t + 1$ and go to Step 2.

The above iterative procedure repeats until the quantization error is less than a threshold or the number of iterations has reached a pre-defined number.

2.2 SOM

Prior to the SOM training [11, 20, 21, 22], a neighborhood structure is imposed on a codebook. The neighborhood structure is represented by a graph $\mathbb{G} = \{\mathbb{V}, \mathbb{E}\}$, where $\mathbb{V} = \{v_1, \ldots, v_M\}$ is a set of vertices and $\mathbb{E}$ is the set of edges in this graph. In this representation, a vertex v_i is associated with a codevector $\vec{c}_i$.

If codevectors $\vec{c}_i$ and $\vec{c}_{i'}$ are joined by an edge, $\vec{c}_i$ and $\vec{c}_{i'}$ are neighbors. The topological distance between two neighbors is defined as 1. The topological distance between $\vec{c}_i$ and $\vec{c}_j$ is defined as the minimum number of edges connecting $\vec{c}_i$ and $\vec{c}_j$ in $\mathbb{G}$. If vertices $\vec{c}_i$ and $\vec{c}_j$ are separated by no more than u topological distance, their associated codevectors $\vec{c}_i$ and $\vec{c}_j$ are called level-u neighbor. For a given codevector $\vec{c}_i$, the index set of its level-u_t neighbors is denoted by $\mathbb{I}_i(u_t)$, where u_t denotes the level-u neighbors at time t.

Some frequently used structures of SOM are shown in Fig. 1, where each dot represents a vertex and each line represents an edge. Note that the figure is only a graphical representation and it does not reflect any actual geometric information of the codebook.

Since the sequential SOM training [11] is not suitable for parallelization, in this paper, we consider the batch mode SOM [11] instead. The idea of SOM is similar to that of LBG. That is, based on the current codebook $\mathbb{Y}(t)$, we partition the dataset into M overlapped subsets Ω_i. The SOM is summarized as follows:

1. $t = 0$.
2. Set $\Omega_i = \varnothing$, for all $i = 1, \ldots, M$.

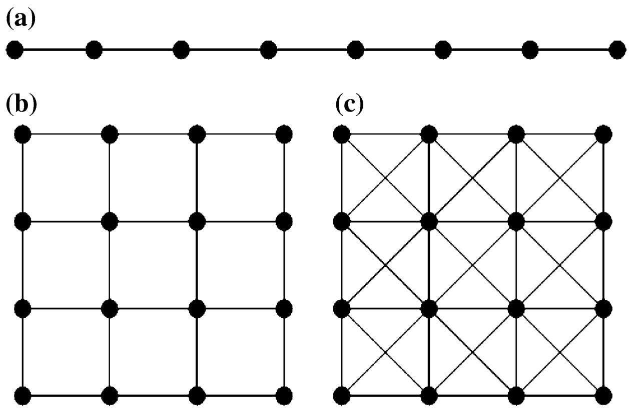

Fig. 1 Three neighborhood structures of SOM. **a** a linear structure; **b** regular grid; **c** diamond-regular grid

3. For each training vector $\vec{x}_j$, find out the nearest codevector $\vec{c}_{i*}(t)$. Put $\vec{x}_j$ into the subsets $\Omega_{i'}$s, where $i' \in \mathbb{I}_{i*}(u_t)$, and u_t linearly decreases to zero during the training process.
4. Update the codebook:

$$\vec{c}_i(t+1) = \frac{1}{|\Omega_i|} \sum_{\vec{x}_j \in \Omega_i} \vec{x}_j. \tag{3}$$

5. Set $t = t + 1$ and go to Step 2.

The above iterative procedure repeats until the quantization error is less than a threshold or the number of iterations has reached a pre-defined number.

If the ordering property of SOM is preserved, the Euclidean distance in the data space between two neighboring codevectors is usually small. Figure 2 shows a typical result of a SOM. As shown in Fig. 2a, the initial codevectors are distributed in a random manner and do not form a good topological ordering. In contrast, the codevectors form a good topological ordering after training as shown in Fig. 2b.

2.3 Background on the GPU

Recently, the demands from the gaming and the film-making industries for the acceleration of 3D rendering have driven the development of graphics processing unit (GPU). The GPU operations are controlled by some user-defined programs called *shaders*. Since GPU is a chip consisting of multiple processing units, we can consider a GPU as a virtual SIMD computer.

Figure 3 shows the graphics pipeline of GPU. It consists of two programmable units, namely the vertex processing unit and the fragment processing unit logically. Programs controlling the vertex processing unit are called vertex shaders, whereas those programs controlling the fragment processing unit are called fragment shaders. Primarily, vertex shaders perform the transformations of vertices, and fragment shaders perform the color calculation of pixels. In particular, a fragment shader can read and modify a texture. Note that a texture can be considered as a 2D array of texels, where each texel has four color channels, namely *R*, *G*, *B*, and *A*.

3 Implementations

3.1 GPU-based LBG implementation

Intuitively, we can consider our implementation as a multiclient–multiserver model, where training vectors are clients and codevectors are servers. For each client (a training vector), it determines its nearest codevector and sends itself to the corresponding server. When the training vectors arrive at a server, the server accumulates the training vectors and updates the number of arrived training vectors. After all clients are served, each server holds the accumulation of arrived training vectors and the number of arrived training vectors. Finally, each server performs a simple division operation to update its codevector.

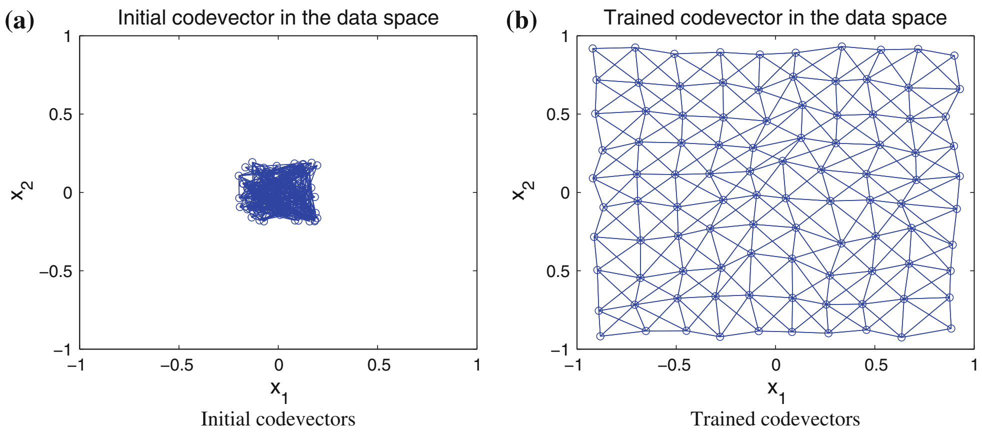

Fig. 2 Codevectors in the data space. **a** Initial codevector before training. **b** A typical result of a 10 × 10 diamond-regular SOM after training. To indicate the neighborhood structure, an edge is added to join two codevectors if they are neighbors of each other in the graph. There are 1024 2D samples in the training set. These samples are uniformly distributed over [−1, 1]

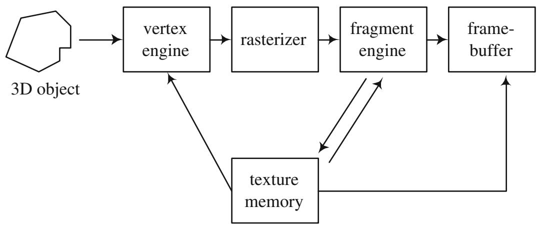

Fig. 3 The graphics pipeline of GPU

In our setting, there are N training vectors and M codevectors, and the dimension of vectors is k. From the shader programming point of view, each training vector is a vertex. A codebook is organized as a number of textures. The logical structure of a codebook in GPU is shown in Fig. 4. The resolution of the textures is equal to $1 \times M$, and each texture position corresponds a codevector. We use $\lceil \frac{k+1}{4} \rceil$ textures to hold a codebook (Note that a texel in a texture have 4 channel, $\{R\ G\ B\ A\}$ and it can hold 4 values.). "An extra channel" is used to store the number of arrived training vectors of a codevector.

Before the GPU training, all training vectors are uploaded to the GPU memory as a *display list* of points, and two sets of textures are initialized. One set of textures is to hold the old codebook for "reading". It initialized the codevectors prepared by CPU. The other set is to hold the updating codebook for "writing". Notice that the read/write ability of textures is realized by using frame buffer objects (FBOs).

The flow diagram of our GPU-based LBG implementation is shown in Fig. 5. For each training vector, the vertex shader performs the nearest codevector searching and finds out the corresponding texel position. Then, a point of size one is rasterized at the corresponding texel position. In the fragment shader, the received training vectors are blended to the writing textures. Besides the received training vectors, a constant "1" is also blended to the extra color channel of the writing textures. Afterward, the writing textures hold the accumulations and the numbers of training vectors of the codevectors. See the appendix for a full example of vertex shader and fragment shader. Finally, an additional fragment shader is performed to update the codevector by dividing the accumulations by the numbers of training vectors.

Our implementation of LBG algorithm is summarized in the following steps:

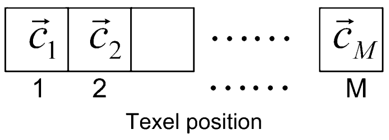

Fig. 4 The logical structure of a codebook in GPU

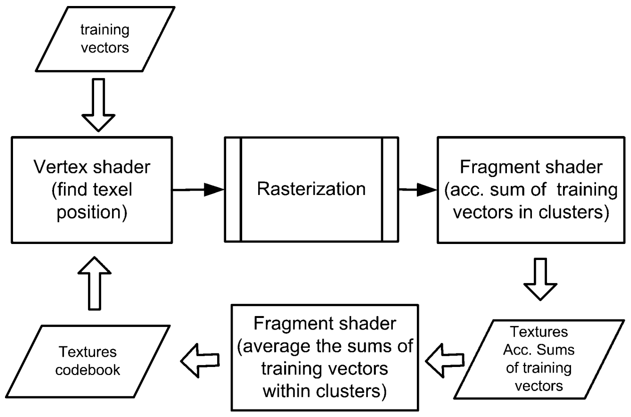
training vectors
Vertex shader (find texel position)
Rasterization
Fragment shader (acc. sum of training vectors in clusters)
Textures Acc. Sums of training vectors
Fragment shader (average the sums of training vectors within clusters)
Textures codebook

Fig. 5 The flow diagram of our GPU based LBG implementation

(1) Pass the training vectors the vertex shader by drawing the *display list* of points.
(2) Use the vertex shader to perform the nearest codevector searching and find out the corresponding texel position, such that a point of size one is rasterized at the corresponding texel position. The training vector is also passed to the fragment shader for further processing.
(3) Use a fragment shader to update the codebook. The received training vectors are blended to the writing textures. Besides the received training vectors, a constant "1" is also blended to the extra color channel of the writing textures.
(4) Swap the reading texture set and writing set.[1]
(5) Use an additional fragment shader to update the codevectors. Firstly, read the accumulated values from the reading texture set. Secondly, divide the accumulated values by the number of training vector of the codevectors. Output the new codevectors to the writing texture set.
(6) Swap the reading textures and writing textures.
(7) Check the termination conditions by CPU. If the conditions are satisfied, stop. Otherwise, goto step 1.

3.2 GPU-based SOM implementation

In the SOM implementation, a codebook is also organized as a number of 2D textures as shown in Fig. 6. For each training vector, the vertex shader finds out its nearest codevector as well as the corresponding texel position. Afterward, instead of a point of size one, a point of size $2u_t + 1$ is rasterized at the corresponding texel position. The remaining two fragment shaders are same as those of the LBG. The main difference is that each training vector

[1] In fact, we do not need to swap the contents of the two sets of textures. We only need to swap their texture IDs.

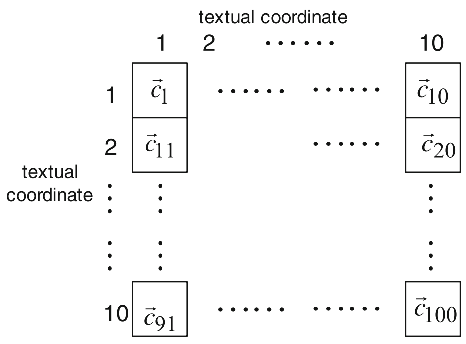

Fig. 6 The logical structure of a 10 × 10 2D SOM codebook in GPU

not only update the nearest codevector but also its u_t-level neighborhood in case of SOM. Hence, the texel positions of codevectors with neighborhood distance less than or equal to u_t are updated as shown in Fig. 7. The size of neighborhood u_t decreases along with t, which can be given by

$$u_t = \begin{cases} 2, & \text{if } 0 \le t < T/8 \\ 1, & \text{if } T/8 \le t < T/2 \\ 0, & \text{otherwise} \end{cases}, \tag{4}$$

where T is the maximum number of iterations.

4 Results and discussion

To evaluate the speed performances of our algorithm, we compared the speed of our approach with that of Takizawa's approach and the CPU approach. All of the experimental results are obtained using a PC with its configuration listed in Table 1.

Table 1 The configuration of the computer used in our experiments

CPU	Intel Core2 Q8200
Memory	4 GB
Display card	GeForce GTX 280
Video memory	1024 MB
Interface	PCI-Express 16 X
System	Windows XP
Shader language	Cg

4.1 LBG algorithm

We have implemented three approaches for the LBG training, i.e. the CPU approach, Takizawa's approach, and our GPU approach. The data vectors dimension k is 16. By varying the number of training vectors N and the number of codevectors M, the number of iterations per second for various settings are measured. The results are summarized in Table 2.

Our GPU-based LBG approach can run around 17–50 times faster than CPU-based LBG approach as shown in Table 2. This speed improvement is larger as the number of codevectors increases. Compared with Takizawa's scheme, our GPU-based LBG approach can run 1.5–4 times faster. The improvement is due to the reduction in the number of rendering passes and the reduction in the amount of data transfer between CPU and GPU memories.

4.2 SOM

We apply our scheme to two scenarios of vector quantization. One is 3-dimensional RGB color quantization, and

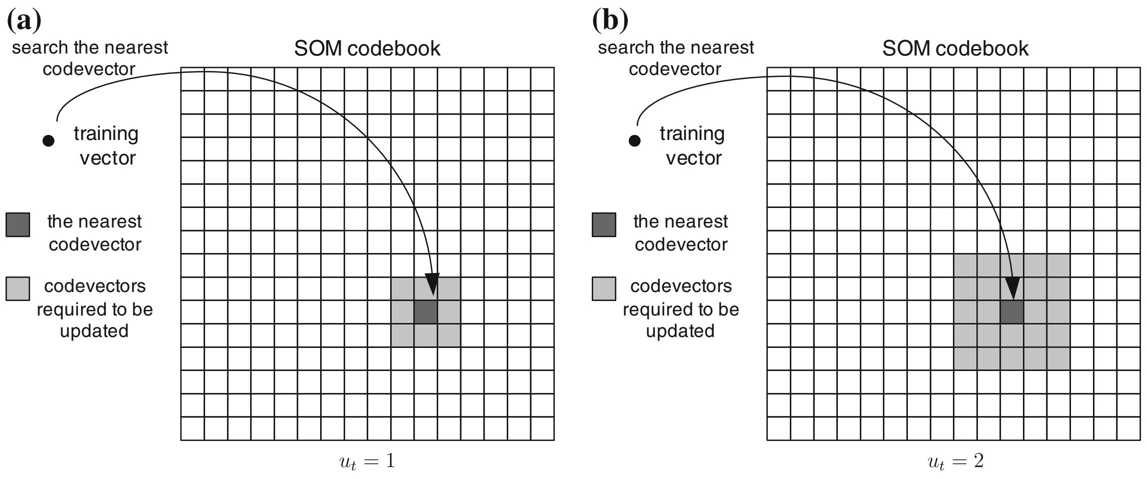
(a)
search the nearest codevector
SOM codebook
training vector
the nearest codevector
codevectors required to be updated
$u_t = 1$
(b)
search the nearest codevector
SOM codebook
training vector
the nearest codevector
codevectors required to be updated
$u_t = 2$

Fig. 7 In our GPU SOM implementation, we update a number of texels by controlling the point size. In (**a**), $u_t = 1$ (the point size is 3). So, a training vector will generate 9 fragments, and the corresponding 9 codevectors are updated. Similarly, in (**b**), $u_t = 2$ (the point size is 5). So, a training vector will generate 25 fragments, and the corresponding 25 codevectos are updated

Table 2 Speed of the LBG scheme

N	CPU approach				Takizawa's approach				Our GPU approach			
	M											
	32	64	128	256	32	64	128	256	32	64	128	256
16384	35.75	20.58	10.54	5.423	354.2	267.3	213.6	133.3	1228	1000	492.6	258.1
65536	9.816	5.111	2.635	1.349	96.89	78.12	61.57	43.25	376.5	293.6	137.3	69.87
262144	2.403	1.273	0.6561	0.338	24.24	21.05	16.32	11.72	105.2	77.29	35.28	17.79

The speed is measured in number of iterations per second. The vector dimension is 16. *N* is the number of training vectors, and *M* is the number of codevectors

Table 3 Speed of CPU SOM Scheme for color quantization

Image size	Point size = 1		Point size = 3		Point size = 5	
	Resolution of SOM		Resolution of SOM		Resolution of SOM	
	8 × 8	16 × 16	8 × 8	16 × 16	8 × 8	16 × 16
256 × 256	13.61	3.67	12.30	3.56	11.48	3.41
512 × 512	3.32	0.92	3.12	0.89	2.84	0.84
1024 × 1024	0.84	0.23	0.78	0.22	0.73	0.22

The speed is measured in number of iterations per second. The vector dimension is equal to 3

Table 4 Speed of our GPU SOM scheme for color quantization

Image size	Point size = 1		Point size = 3		Point size = 5	
	Resolution of SOM		Resolution of SOM		Resolution of SOM	
	8 × 8	16 × 16	8 × 8	16 × 16	8 × 8	16 × 16
256 × 256	1282	492.61	641.02	456.62	304.88	236.97
512 × 512	291.64	133.33	164.20	130.54	75.30	61.53
1024 × 1024	75.3	33.68	39.49	33.50	18.13	15.42

The speed is measured in number of iterations per second. The vector dimension is equal to 3

Table 5 Speed of our CPU SOM Scheme

No. of training vectors	Point size = 1		Point size = 3		Point size = 5	
	Resolution of SOM		Resolution of SOM		Resolution of SOM	
	8 × 8	16 × 16	8 × 8	16 × 16	8 × 8	16 × 16
16384	21.05	5.75	19.00	5.46	17.52	5.16
65536	5.38	1.43	5.09	1.39	4.72	1.34
262144	1.35	0.36	1.27	0.35	1.16	0.34

The speed is measured in number of iterations per second. The vector dimension is equal to 16

Table 6 Speed of our GPU SOM Scheme

No. of training vectors	Point size = 1		Point size = 3		Point size = 5	
	Resolution of SOM		Resolution of SOM		Resolution of SOM	
	8 × 8	16 × 16	8 × 8	16 × 16	8 × 8	16 × 16
16384	738.92	202.16	505.90	202.02	237.15	176.16
65536	206.46	53.78	136.18	53.63	64.21	46.49
262144	53.04	13.90	34.97	13.63	16.39	12.25

The speed is measured in number of iterations per second. The vector dimension is equal to 16 the other is 16-dimensional vector quantization. For the sake of speed comparison, we make u_t constant during the training. Tables 3 and 4 show the training speeds of CPU-based SOM and our GPU-based SOM for different point sizes. As shown in the tables, our GPU-based SOM approach can run around 25 to 100 times faster compared to the CPU-based LBG approach. In particular, when the image size is $1,024 \times 1,024$, the SOM resolution is 8×8 (64 codevectors), and the point size is 5, our GPU-based SOM approach runs 25 times faster. When we increase the resolution to 16×16 (256 codevectors), our GPU-based SOM approach can run 70 times faster. In general, we will have a bigger improvement on the speed when the SOM resolution, or the number of codevectors, increases.

We also implement our scheme for clustering for high dimensional data. The data dimension is set to 16. In the experiment, we vary the number N of training vectors and the number M of codevectors. We measure the number of iterations per second for various settings. Tables 5 and 6 show the training speeds of CPU-based SOM and our GPU-based SOM for different point sizes. From the tables, our GPU-based SOM approach can run around 15 to 40 times faster compared to the CPU-based LBG approach. The general trend of the improvement is similar to that of the color quantization case. That is, the improvement on the speed is large when we increase the number of codevectors, i.e., the SOM resolution.

5 Conclusion

In this paper, we proposed a novel GPU implementation for the LBG and SOM training algorithms. By using vertex shader to implement the nearest codevector searching and to do random write, the functionality of GPUs is better utilized. Our experimental results show that the proposed scheme is remarkably better than the conventional GPU approach. Besides, our approach unified the implementation of LBG training and SOM training.

Acknowledgment The work was supported by a research grant from the City University of Hong Kong (Project No.: 7002588).

Appendix

Listing 1 shows the vertex shader of GPU LBG when the vector dimension is 16.

Listing 1. The vertex shader of LBG training for locating the nearest representative codevector index.

```
void cg_vp_findmin(
  float4 trainV0 : POSITION, //4 components of the training
     vector
  float4 trainV1 : TEXCOORD1,
  float4 trainV2 : TEXCOORD2,
  float4 trainV3 : TEXCOORD3,

  out float4 nearest : POSITION,
  out float4 otrainV0 : TEXCOORD0,
  out float4 otrainV1 : TEXCOORD1,
  out float4 otrainV2 : TEXCOORD2,
  out float4 otrainV3 : TEXCOORD3,
  uniform samplerRECT textid[r_ntex]// the codevector texture
){
  int i, j, ind_neareast=0;
  float4 codevector[r_ntex];
  float min_dist=FLT_MAX, dist;

  for( i=0; i<codebook_size; i++ )
  {
      //get the codevector
      for( j=0; j<r_ntex; j++ )
      {
        codevector[j]= texRECT(textid[j], float2(i,0) );
      }
      //calculate the distance of training vector and codevector;
      dist = vec_distance(trainV0,trainV1,trainV2,trainV3,
      codevector[0],codevector[1],codevector[2],codevector[3]);
      if( min_dist>dist )
      {
        min_dist=dist;
        ind_neareast=i;
      }
    }
    //output nearest position as normalized window coordinate
    nearest =
    float4( 2*float(ind_neareast+.5)/codebook_size -1, 0, 0, 1);
    //output the training vector
    otrainV0=trainV0;
    otrainV1=trainV1;
    otrainV2=trainV2;
    otrainV3=trainV3;
  }
```

List 2 shows the fragment shader of GPU LBG when the vector dimension is 16.

Listing 2. The fragment shader of LBG training for the accumulation and the counting of training vectors .

```
void cgfl_fp_blend(
  float4 otrainV0 : TEXCOORD0,
  float4 otrainV1 : TEXCOORD1,
  float4 otrainV2 : TEXCOORD2,
  float4 otrainV3 : TEXCOORD3,
  out float4 outsum0  : COLOR,
  out float4 outsum1  : COLOR1,
  out float4 outsum2  : COLOR2,
  out float4 outsum3  : COLOR3,
  out float4 outsum4  : COLOR4
){
  //blend to the framebuffer
  outsum0 = otrainV0;
  outsum1 = otrainV1;
  outsum2 = otrainV2;
  outsum3 = otrainV3;
  //to record the number of vectors accumulated
  outsum4 = float4(1.0, 1.0, 1.0, 1.0);
}
```

References

1. Linde Y, Buzo A, Gray RM (1980) An algorithm for vector quantizer design. IEEE Trans Commun 28(1):84–95
2. Gray RM (1984) Vector quantization. IEEE Acoust Speech Signal Process Mag 1(1):4–29
3. Gersho A, Gray RM (1991) Vector quantization and signal compression. Kluwer, Norwell
4. Lu Z-M, Xu D-G, Sun S-H (2000) Multipurpose image watermarking algorithm based on multistage vector quantization. IEEE Trans Image Process 14(6):822–831
5. Tsang PWM, Tsang WH (1996) Preservation of interlaced patterns in encoding video signals using side match vector quantization smoothing regularization of MLP for Jacobian stabilization. IEEE Trans Consu Electron 42(1):112–120
6. Mukherjee D, Chae JJ, Mitra SK, Manjunath BS (2000) A source and channel-coding framework for vector-based data hiding in video. IEEE Trans Circuits Syst Video Technol 10:630–645
7. Sim J-Y, Lee S-U (2008) Compression of 3D point visual data using vector quantization and rate-distortion optimization. IEEE Trans Multimed 10(3):305–315
8. Pham T (2003) Alignment-free sequence comparison with vector quantization and hidden markov models. In CSB '03: Proceedings of the IEEE computer society conference on bioinformatics, Washington, DC, USA, 2003, p 534, IEEE Computer Society
9. Ramasubramanian V, Paliwal KK (1992) Fast k-dimensional tree algorithms for nearest neighborsearch with application to vector quantization encoding. IEEE Trans Signal Process 40(3):518–531
10. Laha A, Chanda B, Pal NR (2006) Accelerated codebook searching in a som-based vector quantizer. IJCNN 2006. 2006, pp 3306–3311
11. Kohonen T (2001) Self-organizing maps. Springer, Berlin
12. Mark WR, Glanville RS, Akeley K, Kilgard MJ (2003) Cg: A system for programing graphics hardware in a C-like language. ACM Trans Graph 22(3):896–907
13. Bolz J, Farmer I, Grinspun E, Schreoder P (2003) Sparse matrix solvers on the GPU: conjugate gradients and multigrid. ACM Trans Graph 22(3):917–924
14. Leung CS, Wong TT, Lam PM, Choy KH (2006) An RBF-based image compression method for image-based rendering. IEEE Trans Image Process 15(1):1031–1041
15. Garcia A, Shen HW (2005) GPU-based 3d wavelet reconstruction with tileboarding. Vis Comput 21(8–10):755–763
16. Wong TT, Leung CS, Heng PA, Wang J (2007) Discrete wavelet transform on consumer-level graphics hardware. IEEE Trans Multimed 9(3):668–673
17. Ho TY, Lam PM, Leung CS (2008) Parallelization of cellular neural networks on gpu. Pattern Recogn Lett 41(8):2684–2692
18. Bohn CA (1998) Kohonen feature mapping through graphics hardware. In: Proceedings of 3rd international conference on computational intelligence and neurosciences, pp 64–67
19. Takizawa H, Kobayashi H (2006) Hierarchical parallel processing of large scale data clustering on a pc cluster with gpu co-processing. J Supercomput 36:219–234
20. Togneri R, Lai E, Attikiouzel Y (1990) Kohonen's algorithm for the numerical parametrisation of manifolds. Pattern Recognit Lett 11(5):313–319
21. Leung CS, Chan LW (1997) Transmission of vector quantized data over a noisy channel. IEEE Trans Neural Netw 8(3):582–589
22. Leung CS, Chan LW (1999) Design of trellis vetor quantizers using kohonen maps. Neural Netw 12(6):907–914