文章编号:1002-1175(2011)04-0531-17

# Parallelization and performance optimization of video feature extractions on multi-core systems*

ZHANG Qi[1†], CHEN Yu-Rong[2], LI Jian-Guo[2], HU Yun[1], XU Yin-Long[1]

(1 *Department of Computer Science and Technology, University of Science and Technology of China, Hefei* 230026, *China*;
2 *Intel Labs China, Intel Corporation, Beijing* 100080, *China*)

(Received 14 July 2010; Revised 19 October 2010)

**Zhang Q, Chen Y R, Li J G, et al. Parallelization and performance optimization of video feature extractions on multi-core systems[J]. Journal of Graduate University of Chinese Academy of Sciences, 2011, 28(4): 531 – 547.**

**Abstract** The low-level video feature extractions are the most time-consuming components in content-based video information retrieval systems. In this paper we study parallelization and performance optimization methods of four video feature extractions on multi-core systems. Experiments show that the processing speeds of these programs are 17 times the original processing speed on average when eight cores are used. Besides, detailed performance analysis helps us find bottlenecks and suggest ways to further improve multi-core systems performance in future.

**Key words** program performance optimization, multi-core systems, video feature extraction

**CLC** TP311

Nowadays, with advances in video capture and storage techniques, the amount of video data has exploded not only in enterprises but also at our homes. Concomitantly, there is an increasing demand for a system that can help end user to index massive amounts of video data for further search, browse, and management tasks. Content-based video information retrieval (CBVIR) as a common solution for video retrieval attracts more and more attention both in research community and industry.

CBVIR is a computational technique to index unstructured video information in terms of low-level visual features[1]. An experimental content based video search standard, MPEG-7[2] has been proposed by ISO/IEC, which focuses on multimedia content description interface. MPEG-7 includes a set of visual color, texture, shape, and motion descriptors which describe multimedia content. So that users can search, browse, and retrieve the content more efficiently and effectively than with existing mainly text-based search method. However, some MPEG-7 descriptors are very computationally expensive and time-consuming[3].

Since low-level visual feature extraction is the most time-consuming part in CBVIR systems, these applications are much more compute intensive than traditional video decoding/encoding applications. Although the indexing can be done in off-line mode, there are many more emerging scenarios that require a real-time or even super-real-time processing capability in CBVIR systems. To implement a fast CBVIR system, video feature extractions should be sped up.

* Supported by the National Natural Science Foundation of China(61073038)

† E-mail: qizhang@ustc.edu

In the past decades, the main method to improve the processor speed is continuously increasing the clock rate and transistor integration capacity. However, the power and thermal constraints increase with frequency, which means that there is no endless performance advances by increasing clock rate. Therefore, in order to achieve higher performance in future, processor vendors have turned to multi-core processors[4] where several execution cores are incorporated on a single chip. And multi-core systems have become main stream at present. The video feature extraction can be accelerated by fully utilizing the multi-core's computational power[5-8].

In this paper we improve performance of four video feature extractions on multi-core systems by parallelization and optimization. The four video features are color correlogram, multi-resolution simultaneous auto-regressive (MRSAR) models, Gabor texture and scale invariant feature transform (SIFT), which are widely applied in CBVIR systems. This paper parallelizes each feature extraction program and explores performance optimization methods for them, including serial performance optimization and parallel performance optimization. Experimental results show the serial optimization improves their performance 262% on average; parallelization and parallel optimization contribute another 1440% performance improvement by using eight threads on eight cores. We also conduct detailed performance analysis based on our experiments to identify program and system bottlenecks which limit scalability performance of multi-thread programs on multi-core systems.

With the multi-core processor becoming mainstream, many researchers focus on the methods to improve program and system performance on the multi-core platforms[9-16]. To get high performance on multi-core systems, programs need to be parallelized and run at multiple processes or multiple threads[17]. In the high performance computing, parallel programs normally run at multiple processes on mainframe computers or clusters. But on multi-core systems, we face desktop programs and it needs more fine-grained parallelization. In this case multi-thread programming is the best choice, because of its light overhead[10]. Except parallelizing application programs, some other researchers study on optimizing operating systems for multi-core platforms. Threads co-running on a multi-core processor share and compete for the shared resources on the processor such as functional units and caches. How can the shared resources be efficiently used becomes a critical issue. A lot of thread scheduling policies[12-16] are designed for multi-core systems to co-schedule threads that can efficiently use the shard resources in multi-core processors.

In this paper, we study how to optimize the performance of video feature extraction programs on multi-core systems. Besides scientific and commercial computing, multi-core systems are widely used for desktop and family applications. How to take full advantage of multi-core system in family application is a very important issue for researchers, programmers and customers. Video processing is one key application in desktop, and we take video feature extractions as a case study in this paper. Compared to scientific computing and high performance computer, applications run on multi-core systems do not have very great quantity of computation generally. For optimization on these desktop applications and multi-core systems, even small overhead will take much impact on performance. So we must take as many techniques as possible to reduce overhead of parallelization. In this paper, we do our best to improve load balance between threads, and to reduce synchronization overhead, which is very important for performance of parallel programs. We carefully optimize the cache performance of each parallel program. For example, false sharing is totally removed in these parallel programs, which is a pitfall in shared memory parallel processing. One key feature of multi-core processors is some cores in one processor will share resource such as last level cache, but some others will not. So thread scheduling policy will impact program performance. In our study, we try different thread affinity policies and choose the best one for each parallel program to get the optimal performance.

The contributions of this paper are three-fold. First, we improve performance of four most compute intensive low-level video feature extraction programs. By our parallelization and optimization, the processing speed with eight threads is sped up to about 17 times the original serial programs on average. Specifically, serial optimization improves performance to 2.6 times the original programs; then parallelization and parallel optimization further improve performance to 6.4 times the serial optimized programs by using eight cores. Second, the parallelization and performance optimization methods researched in this paper are representative in video processing applications, and can be further applied in other applications to optimize their performance on multi-core systems. Finally, by our detailed analysis of experimental data on actual multi-core systems, we identify the possible causes of bottlenecks for applications and systems. The parallel load imbalance, cache coherence miss, and available system bandwidth are main factors which limit the speedup performance of parallel programs on multi-core systems. This provides evidences and suggestions for further improving performance of multi-core systems in future.

The remaining of this paper is organized as follows. In section 1, we introduce original algorithms of the four video feature extractions which are widely used in CBVIR systems. Section 2 presents the serial performance optimization applied in each video feature extraction programs. Section 3 parallelizes these programs and section 4 gives the parallel performance optimization to improve the parallel programs performance. The experimental results and performance analysis are reported in section 5. Finally, we conclude this paper in section 6.

# 1 Low-level video feature extractions

In this section, we briefly introduce several low-level visual feature extraction algorithms. Low-level visual feature extractions are the foundation of CBVIR systems. The final representation of an image is a set of feature vectors extracted from image itself, which refers to visual information, such as colors, texture, shapes, localization etc. In this paper, low-level feature extractions are selected for our study based on the following criteria: 1) high computational complexity, 2) widely used with good retrieval performance, 3) representative of different optimization scheme. Finally, four features are emerged out. They are color correlogram feature[18], Markov random filed texture feature (specifically, multi-resolution simultaneous auto-regressive models, MRSAR)[19], Gabor texture feature[20], and scale invariant feature transform (SIFT)[21]. Those four features are widely used in content-based image/video information retrieval systems, and reported very high retrieval performance by many researchers[22-23]. We briefly review their algorithms in this section.

## 1.1 Color correlogram feature

The color histograms, moments, and sets do not involve local relationships among the neighborhood pixels. Fortunately, color correlogram has been proposed recently to characterize how the spatial correlation of pairs of colors is changing with the distance. It has been shown to provide much better performance than color histogram, moments etc[18,22], and been widely used in CBVIR systems[3,24].

In color correlogram feature, let $I$ be an $m \times n$ image, for a pixel of $p = (x, y)$, denote the color by $I(p) = c$. Correlogram adopts infinite norm to measure the distance between two pixels, i. e. $|p_1 - p_2| = \max\{|x_1 - x_2|, |y_1 - y_2|\}$. For a given distance $k \in \{1, \cdots, d\}$, the color correlogram of image $I$ for the finite color set $\{c_i\}$ is defined as

$$\gamma_{c_i,c_j}^{(k)}(I) = \Pr_{\substack{p_1 = c_i \\ p_2}}[p_2 = c_j \,||\, p_1 - p_2| = k], \tag{1}$$

where $p_1$ is a pixel of color $c_i$, $p_2$ is a pixel of color $c_j$, and the distance between $p_1$ and $p_2$ is $k$. So $\gamma_{c_i,c_j}^{(k)}(I)$ gives the probability that a pixel at distance $k$ away from a given pixel of color $c_i$ is of color $c_j$.

When assuming $c_i = c_j$ in the definition, we obtain the auto-correlogram $\alpha_c^{(k)}(I) = \gamma_{c,c}^{(k)}(I)$, which captures spatial information between identical colors. To catch more local spatial information, we can calculate correlogram in a banded distance (the distance between two pixels is in $[k_1, k_2]$) as banded correlogram:

$$\bar{\gamma}_{c_i,c_j}^{(k)}(I) = \sum_{k'=k_1}^{k_2} \gamma_{c_i,c_j}^{(k')}(I), \tag{2}$$

where $k_1$ and $k_2$ are the bounds of the distance band $k$.

In our experiments, all colors are quantized into 36 distinct ones in the LUV color space in order to reduce the size of the feature set, and $\{1,3,5,7\}$ for the possible banded distance $k$.

## 1.2 MRSAR feature

MRSAR has been shown one of the best texture feature descriptors by many performance evaluations[20,22-23]. The MRSAR method models the texture as a second-order non-causal Markov random fields. MRSAR is carried out in a 21x 21 window sliding across the input image with fixed pixel steps (7 pixels in our experiments) on three resolutions. For a given resolution $k$, the model is defined as

$$g(i,j) = \sum_{(m,n)\in N_k} a_k(m,n)g(i-m,j-n) + n_k(i,j), \tag{3}$$

where $N_k$ is the employed neighborhood of the pixel $(i,j)$ at resolution $k$, $g(i,j)$ is the gray level values in the image, $a_k(m,n)$ is the model coefficients, and $n_k(i,j)$ is the error term associated with the model.

MRSAR assumes the model is symmetric, i. e. $a_k(m,n) = a_k(-n,-m)$. Each pixel in the 21x21 window is characterized by an underlying four parameters auto-regressive model at three different resolutions using sub-windows size $5\times5$, $7\times7$ and $9\times9$. The least squares estimations are carried out at each resolution independently. Together with the standard deviation of the error term, five parameters are estimated for each resolution, and concatenated for a 15-dimensional feature vector. The final feature is the mean and covariance matrix of the 15-dimensional feature on all sliding windows.

## 1.3 Gabor texture feature

Gabor filters offer the best simultaneous localization of spatial and frequency information. It has been widely applied in image processing tasks such as edge detection, invariant object recognition, and compression[25]. Gabor texture feature also has emerged as an important visual primitive for search and browsing[18]. The 2-dimensional (2D) Gabor filters are defined as follows when assumes $\sigma_x = \sigma_y = \sigma$[25]:

$$\begin{aligned}\Psi(z,\sigma_s,\theta_o) &= \frac{1}{2\pi\sigma_s^2}\exp\left\{-\frac{\|z\|^2}{2\sigma_s^2}\right\}\left[\exp\left\{\frac{\mathrm{j}x'\kappa}{\sigma_s}\right\}-\exp\left\{-\frac{\kappa^2}{2}\right\}\right],\\ z &= (x',y'),\\ \begin{cases}x' = x\cos\theta_o + y\sin\theta_o\\ y' = -x\sin\theta_o + y\cos\theta_o\end{cases},\end{aligned} \tag{4}$$

where $x$, $y$ are pixel position in spatial domain, $\kappa$ is a parameter for filter bandwidth, $\theta_o$ is the filter angle for $o$-th orientation, and $\sigma_s$ are the Gaussian deviation for $s$-th scale, which is proportion to the wavelength of the filters.

The Gabor representation of images is derived by convolving the image with the Gabor filters:

$$G_{s,o}(x,y) = I(x,y) \otimes \Psi(x,y,\sigma_s,\theta_o), \tag{5}$$

where $\otimes$ is the symbol for spatial convolution, and $I(x,y)$ is an input image. This convolution can be fast implemented by fast fourier transformation (FFT):

$$G_{s,o}(I) = \mathrm{IFFT2}\{\mathrm{FFT2}(I) * \mathrm{FFT2}(\Psi_{s,o})\}, \tag{6}$$

where IFFT2 refers to inverse 2D FFT, and $*$ indicates the production between corresponding elements.

The MPEG－7 experimentation standard suggests Gabor filters based homogeneous texture descriptor. The

image is filtered with 6-orientation and 5-scale Gabor filters, and the means and standard deviations of the filtered outputs in the frequency domain are used as the descriptor.

In our implementation, since the filter parameters are fixed for all input images, the frequency domain filters FFT2($\Psi_{s,o}$) can be pre-calculated to save some computations, and the Gabor texture feature extraction requires one forward FFT for each input image, $5 \times 6 = 30$ element-based-production (i.e. the $*$ operation in frequency domain in eq. (6)) between images and filters, and 30 inverse FFTs.

## 1.4 SIFT feature

SIFT is an approach for detecting and extracting local feature descriptors from images. The flow chart of the SIFT algorithm is shown in Fig. 1(a). The major stages of computation used to generate the set of image features are: building Gaussian scale space, keypoint detection and localization, orientation assignment, and keypoint descriptor[12].

Building Gaussian scale space Interest points for SIFT features correspond to local extreme of difference-of-Gaussian (DoG) images at different scales. An efficient approach to construction of DoG is shown in Fig. 1(b). For each octave of scale space, the initial image is repeatedly convolved with Gaussians to produce the set of scale space images as shown on the left of the Fig. 1(b). Adjacent Gaussian images are subtracted to produce the DoG images as shown on the right of the Fig. 1(b). After each octave, the Gaussian image is down-sampled by a factor of 2, and the process repeated. The convolved images are grouped by octave, and we obtain a fixed number of DoG per octave.

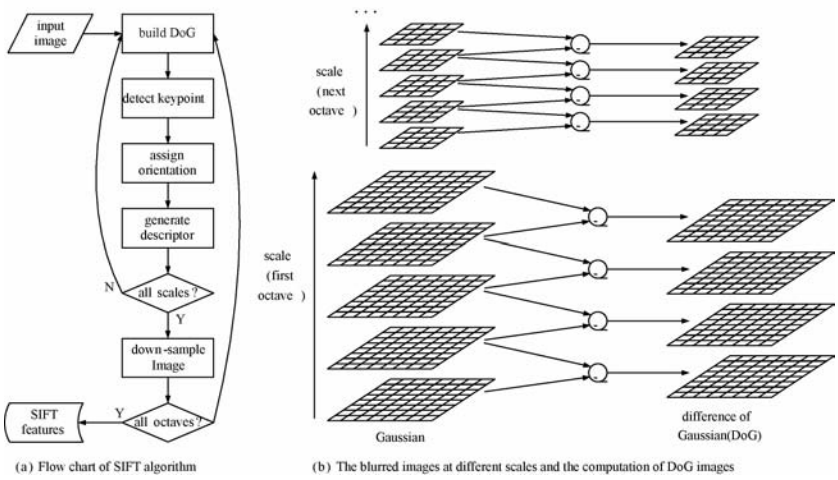



**Fig. 1 SIFT feature extraction**

Keypoint detection and localization Keypoints are identified as local maxima or minima of the DoG images across scales. Each pixel in the DoG images is compared to its 26 neighbors in 3x3 regions at the current and adjacent scales. If the pixel is a local maximum or minimum, it is selected as a candidate keypoint. Then we remove low contrast points and edge responses, and finally get some keypoints.

Orientation assignment To determine the keypoint orientation, a gradient orientation histogram is computed in the neighborhood of the keypoint. The contribution of each neighboring pixel is weighted by the gradient magnitude and a Gaussian window with a $\sigma$ that is 1.5 times the scale of the keypoint. Peaks in the histogram correspond to dominant orientations. A separate keypoint is created for the direction corresponding to the histogram maximum, and any other direction within 80% of the maximum value.

Keypoint descriptor Once a keypoint orientation has been selected, the feature descriptor is computed as a set of orientation histograms on 4x4 pixel neighborhoods. The orientation histograms are relative to the

keypoint, the orientation data comes from the Gaussian image closest in scale to the keypoint's scale. Histograms contain 8 bins each, and each descriptor contains an array of 4 histograms around the keypoint. This leads to a SIFT feature vector with $4 \times 4 \times 8 = 128$ elements. This vector is normalized to enhance invariance to changes in illumination[25].

## 2 Serial performance optimization

Recently, most of the video analysis applications including CBVIR are expected to be applied in real-time environments, but the processing speed can not meet the demand sometimes. To optimize the whole video retrieval system, each video feature extraction kernel must be speeded up. In this section, we give several optimization methods to improve the serial program performance.

### 2.1 Serial performance optimization for color correlogram

Removing reduplicative computation and unnecessary computation is the main way to optimize performance of color correlogram feature extraction serial program.

In color correlogram feature extractions, the major operation is accumulating color co-occurrence times at the neighborhood of each pixel. We observe that a pixel $x$ is at distance $k$ away from another pixel $y$, meanwhile the pixel $y$ is at the same distance from pixel $x$. A redundancy exists since the contact is calculated twice for each two pixels. We eliminate the redundant computation by merging the two statistics into one. This reduces the statistical operations to half without any payment.

While examining the profile data collected by the Intel VTune Performance Analyzer[27], we observe that the hotspot is the computation of infinite norm distance between two pixels. In the original implementation, each pixel is related to a sliding window, which is a square matrix and the center is this pixel. The distances between the central pixel and all other pixels in this window need to be calculated. Fortunately, we detect that once given a distance, all the pixels can be determined which are at the certain distance away from the central pixel. So we change the loop's variable from the coordinate into the distance, then the computation of infinite norm distance is not required.

### 2.2 Serial performance optimization for MRSAR

The performance of MRSAR feature extraction serial program can be optimized mainly through improving program locality to increase cache hits.

In MRSAR, we find that a 2-dimensional array named sum needs to be accessed in two orders, sometimes in row order and sometimes in column order. As we know, there will be poor cache performance due to accessing array in column order. So we keep a reverse copy of the array sum, and replace the sum with the reverse version when we have to access the array in column order. This operation improves cache reuse of this program, and the tradeoff is more memory requirement.

### 2.3 Serial performance optimization for Gabor

In this section, highly optimized library and single-instruction multiple-data technique are applied to improve performance of Gabor feature extraction serial program.

In Gabor, we find the hotspot is the function fftwf_execute which executes discrete Fourier transform for the Gabor filters. To achieve better performance we modify the linked library from open sourced FFTW library[28] to intel math kernel library (MKL) Version 9.0[29]. The FFTs in MKL are highly optimized for the newest Intel dual-core and quad-core processors and can provide significant performance gains over alternative libraries for medium and large transform sizes.

We also observe that there are abundant floating point operations around the function fftwf_execute. Since almost all modern processors have the ability of processing streaming data, we conduct some SIMD (single-

instruction multiple-data) optimization[30] to accelerate the element-production between images and filters. We align data structures on 16-byte boundary, and reconstruct the code to vector operation. Then Intrinsics[30] functions are applied to fully take advantage of SIMD instructions.

### 2.4 Serial performance optimization for SIFT

To optimize performance of SIFT feature extraction serial program, we mainly focus on improve program locality and reduce bandwidth demand.

In SIFT, we widely use loop fission in the serial optimization. This technique breaks a big loop into two or more smaller loops to improve memory locality and eliminate data dependences.

In addition, the cache performance and bandwidth requirement heavily influence the application's performance on multi-core systems[31]. We applied cache-conscious optimization to improve data locality. For example, the program needs convolving each row and each column of the input image with Gaussian filter. Since the nested loop in the columns order causes bad cache performance, we transform it into access data in the rows order. We also try our best to reduce bandwidth demand and contest, for example removing memory copy operation. In SIFT algorithm the flow of computation is very complex, and there are frequent memory copy operations in the original program. As we know the memory copy requires high bandwidth and hard to scale well, so we change the data flow carefully to avoid memory copy.

Finally, to take advantage of the data level parallelism (DLP) architecture features provided by the modern processor, we also utilize SIMD optimization for the float-point computations around the SIFT program.

### 2.5 Some other serial performance optimization

Besides the techniques mentioned above, we also apply some other optimization techniques to improve performance of these video feature extractions, such as using pre-malloc to reduce operating system expense and achieve good data locality, using temporary result caching and sub-expression optimization to remove unnecessary computations, using data blocking, loop unrolling and memory alignment[30] to reduce cache misses.

## 3 Parallelization

With the boom of multi-core processor and the prevalence of shared memory processing, it is important to exploit thread level parallelism within application to fully take advantage of multi-core processing capability. Thread-level parallelism can be exploited in different ways for video feature extractions. We could choose processing several images at the same time, or choose processing one image in parallel internally. However, in order to implement the real time or on-line processing of these video applications, it is important to extract the fine-grained parallelism in each frame. So in this paper, we focus on how to process each frame as fast as possible. In this section, we design parallel algorithms for video feature extractions and implement them with OpenMP programming model[32].

### 3.1 Parallelization for color correlogram

In color correlogram, the major job is counting the color histogram for each pixel. The straightforward parallel method of data segmentation can be applied to this condition. We attach the #pragma omp for directive to the loop of pixel. Then the OpenMP runtime environment can assign data to threads automatically. However, there must be serious contention between those threads, since each thread will access the shared histogram array. And our tests show that as the number of threads increase, the contention grows drastically. So we assign independently histogram array memory to each thread and call a data reduction at the end of thread by #pragma omp critical directive. In this way, we reduce synchronization overhead, and achieve better performance of scalability. In addition, this optimization requires only about 20 KB memory for each thread in

our experiments. The pseudo code of parallel Color Correlogram feature extraction is shown in Fig. 2(a).

### 3.2 Parallelization for MRSAR

In MRSAR, we try to process several sliding windows in parallel. In the original source code, there is a 2-dimensional loop to scan all the windows. There are only thirty to forty iterations for the first level loop. It will cause serious load imbalance when the ratio of iteration number to thread number is very low and the number of iterations is not a multiple of the thread number. So we apply the loop collapsing technique to replace the 2-dimensional loop with a 1-dimensional loop. Thus, there are more iterations which can be assigned to threads, and the runtime of each iteration becomes shorter. In this way, the load imbalance between threads is significantly decreased. The pseudo code of parallel MRSAR feature extraction is shown in Fig. 2(b).

### 3.3 Parallelization for Gabor

In Gabor, the parallelization can be conducted with different granularities, such as filter level and FFT level. But for FFT level parallelization, by checking with Intel Thread Profiler we find that there is a sequential region in the fftwf_execute function which downgrades the scaling performance. So we choose parallelize the Gabor feature extractions on filter level.

As we have multiple filters to process in Gabor, the most straightforward parallelization scheme is to perform several filters in parallel. Using this scheme, all the filters are assigned to each thread averagely. So this filters level parallelization scheme can fully utilize the underlying processing capabilities with the least effort. But we have to prepare private memory space and construct individual FFT plan for each filter. This requires a much larger memory consumption to store the input and output of each filter. The pseudo code of parallel Gabor feature extraction is shown in Fig. 2(c).

```
# pragma omp parallel
{
  malloc_local_histogram_array();
  # pragma omp for schedule(dynamic)nowait
  for(int y=0; y<height; y++){
    for(int x=0; x<width; x++){
      calc _correlogram(y, x);
    }
  }
  # pragma omp critical
  {
    merge _result_to_global_histogram_array();
  }
  free _local_histogram_array();
}
```

(a) color correlogram

```
int ny = height/ win_inc_step;
int nx = width/ win_inc_step;
int n = ny*nx;
# pragma omp parallel
{
  malloc_local_data_structure();
  #pragma omp for schedule(dynamic) nowait
  for(int z=0; z<n; z++)
  {
    int win_y = (z/nx)*win_inc_step;
    int win_x = (z%nx)*win_inc_step;
    mrsar_calculation(win_y, win_x);
  }
  free_local_data_structure();
}
```

(b) MRSAR

```
# pragma omp parallel for
 schedule(static)
for(int i=0; i<filter_number; i++)
{
  for(int k=0; k<image_size; k++)
  {
    convolution(i,k);
  }
  fftwf_execute(inverse_FFT_plans[i]);
  for(int k=0; k<image_size; k++)
  {
    compute_magnitude(i,k);
  }
}
```

(c) Gabor

**Fig. 2 Pseudo code of parallel color correlogram, MRSAR and Gabor feature extraction algorithm**

### 3.4 Parallelization for SIFT

Due to the complexity of SIFT algorithm, we take a conventional approach to parallelize it. That is, we first prioritize the modules in the application according to their importance, and then parallelize them in sequence. After analyzing the serial algorithm, we determine the most compute-intensive modules and select four key modules (i.e. the most time-consuming ones) from the SIFT algorithm. These key modules make up more than 99.8% of the whole SIFT execution time. We describe these key modules and related parallel schemes as follows.

Build Gaussian scale space (BGSS) The BGSS module convolves the input image with Gaussian filters. It is a two-dimensional convolution. So each thread can process one part of the input image data. Since in the original program this module writes result into the input image immediately, there are some contests between threads. Meanwhile, to calculate the DoG after convolution, the original program requires keep a copy of the

input image. So we change the BGSS module to save result to a new data array. As a result, the thread contest and the memory copy operation are eliminated.

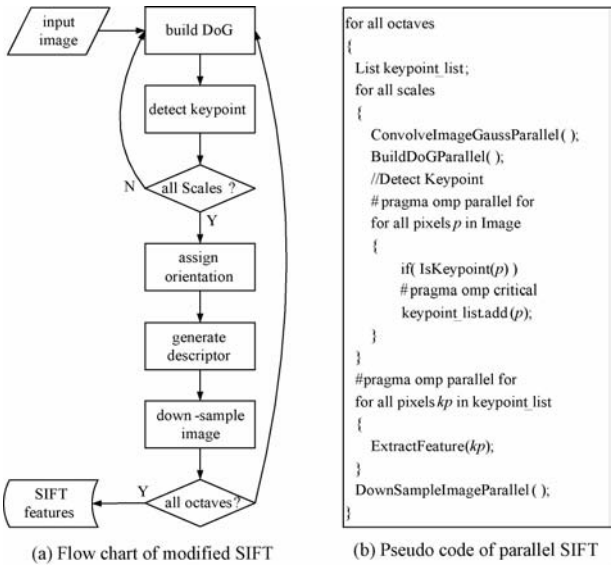
Keypoint detection and localization (KDL) The KDL module detects the local maxima or minima points of the DoG image, and removes points with low contrast from the extreme points. Finally, it saves the localizations of keypoints to a keypoint-list. In this module each pixel in the DoG image is identified whether it is a keypoint. Obviously, data partition method is suitable for this module, and when pushing a keypoint to the result list a synchronization between threads is necessary.

Orientation assignment and keypoint descriptor (OAKD) The OAKD module assigns orientation and computes feature descriptors for keypoints. However, the number of keypoints and the computational effort required for each keypoint are uncertain. So in this module we dynamically schedule the keypoints to the work threads to achieve parallel processing.

Matrix operations (MO) The MO module includes the matrix operations for image processing, such as matrix subtraction and image down-sample. Since the loop iterations in those operations are independent, this module can be easily parallelized by using the data parallelism.

Through above methods, SIFT feature extraction is parallelized totally. But in some experiments, we find the load imbalance is serious by characterizing it with Intel Thread Profiler. In the original SIFT algorithm flow (as shown in Fig. 1(a)), each keypoint will be assigned an orientation and generated a descriptor just after detecting keypoints for one scale. In this case, the load imbalance will occur in steps of "Assign Orientation" and "Generate Descriptor" (the OAKD module), since there may be very few keypoints generated in one scale. Furthermore, as the image is down-sampled in each octave, the number of keypoints detected from each scale is decrease gradually. As a result, the load imbalance will become more serious in the late stage of program execution.

To get load balance in parallel processing, we usually need enough tasks for scheduling. Sometimes we can merge several task sets to obtain a larger one. Motivating by this idea, we carefully analyze the original SIFT algorithm flow and find that for different input images, although the number of octaves is variable, but the number of scales in each octave is constant. Thus, we can gather keypoints detected from all scales of one octave, and then calculate their features in parallel. In this way, we got a modified SIFT algorithm and its flow chart is shown in Fig. 3(a).




(a) Flow chart of modified SIFT (b) Pseudo code of parallel SIFT

**Fig. 3 Flow chart of modified SIFT and pseudo code of parallel SIFT algorithm**

Based on the modified algorithm, we obtained an improved parallel SIFT algorithm. The pseudo code of the improved parallel SIFT algorithm is shown in Fig. 3(b). In this algorithm, we assign one keypoint-list to an octave. Once detecting keypoints from each scale in this octave, we collect those keypoints to the keypoint-list and then detect keypoints from next scale, instead of calculating features for those keypoints immediately. After gathering all keypoints for one octave, we get a larger size keypoint-list and schedule these keypoints to threads for feature extraction. In this way, the load imbalance of OAKD module is reduced significantly.

# 4 Parallel performance optimization

After study and implement the parallel algorithms of the video feature extractions, we further enhance their performance on multi-core systems. We use several Intel software tools to analyze the parallel programs. For instance, we use the Intel Thread Checker[33] to test the correctness of the program, and the Intel Thread Profiler[27] to collect parallel metrics for bottleneck identification. Furthermore, to understand the cache behavior, we use the Intel VTune Performance Analyzer[27] to collect different levels of cache data. We identify the parallel bottlenecks, and the following optimization techniques are employed in our parallel implementations.

## 4.1 Load balance improvement

The load imbalance is one of the most important factors significantly influencing the scalability performance of parallel applications because some processor resources will be idle. The load imbalance status is a function of the size of the tasks and the number of tasks. In parallel video feature extractions, we use following techniques to improve the load balance performance.

Generally, the more tasks number, the better load balance performance. So we need increase tasks number in a parallel region sometimes. For example in MRSAR, a 2-dimension loop is merged into one dimension to enlarge the independent tasks number and get better load balance performance.

In addition, for almost all the parallel regions, we use the dynamic task scheduling policy of OpenMP to minimize the load imbalance. Normally, the dynamic scheduling policy has better load balance performance than the default static policy.

## 4.2 Synchronization overhead reduction

Often threads are not totally independent, which forces the program to add synchronization to guarantee the execution order of the threads. The frequent synchronization calls and the associated waiting operations will degrade the scaling performance on multi-core systems. Generally, the synchronization is presented in the form of critical section, lock, and barrier in the OpenMP implementation. In parallel video feature extractions, we also have to employ some synchronization operations and tune them carefully. We largely eliminate the locks by carefully selecting the proper parallelism techniques.

For example, in the KDL module of SIFT, all threads push the keypoints to one shared point list, and a critical section is necessary for synchronization. Every time threads push a keypoint into the list, a lock has to be employed and this consumes too much time. So we design a lock-free mechanism to reduce the synchronization overhead. The shared keypoint list is replicated into several private lists. Each thread operates on its local list exclusively to avoid the mutual access of the shared list. And these local lists are merged at the end of parallel region.

In section 4.1 we mentioned that to reduce load imbalance, we would choose dynamic task distribution policy of OpenMP for the parallel executions. But dynamic policy will generate much overhead when there are a mount of tasks. Fortunately, we have another choice: guided tasks distribution policy. For example, in SIFT we manually use the guided policy, and the task size is chosen depending on the iteration number in the parallelization loop. Since the task size varies greatly as the image downscaled, the guided policy helps to balance the load balance performance and parallel overhead.

In addition, we make careful use of buffer manipulation for each thread, since frequent memory allocation/free operations will cause severe lock contentions in the heap, and these requests are essentially running in serial even in a parallel region. So we widely use buffer management to reduce frequent memory allocation/free operations in the parallel programs.

### 4.3 Cache efficiency optimization

Good cache efficiency becomes even more important when using multi-core processors, since all cores collectively share a fixed memory bandwidth and several cores share a last level cache. Thus, it is necessary to design algorithms that are cache-conscious and can efficiently utilize the multi-core processing capability.

For video feature extractions, we design the parallel programs with the cache performance in mind. We choose the most favorable granularity in terms of cache performance, where fine-grained threads are more cache-friendly than the coarse-grained ones, because more often they can make fully use of data sharing instead of replicating cache data for each thread.

We widely use cache blocking technique to improve the temporal data locality. We segment the whole data set into several tiles. This subset of data which can fit in cache is operated on all at once before moving on to the next set. Since the block of data can be processed several times before moving on to the next block, this can significantly improve the cache locality performance.

Besides, we also observe that the False Sharing is a common pitfall in shared memory parallel processing. It occurs when two or more cores/processors are updating different bytes of memory that happen to be located on the same cache line. Since multiple cores cannot cache the same line of memory at the same time, when one thread writes to this cache line, the same cache line referenced by the other thread is invalidated. Any new references to data in this cache line by the second thread result in cache misses and potentially huge memory latencies. Therefore, it is important to make sure that the memories referenced by different threads are to different non-shared cache lines. We manually resolve false sharing issues in the parallel video feature extractions by padding each thread's data element to ensure that elements owned by different threads all lie on separate cache lines. For example, in SIFT we dynamically schedule keypoints to different threads for computing features, and the size of one keypoint feature vector is 532bytes. There must be some false sharing between threads. So we expand each feature vector with a blank space of 128bytes to force these threads never to share cache lines.

### 4.4 Thread affinity scheduling

The thread affinity mechanism[34] attaches one thread to a specific core in multi-core or multi-processor systems. It is used to improve the cache performance, and minimize the thread migrations and context switches among cores. It also improves the data locality performance and mitigates the impact of maintaining the cache coherency among the cores/processors. Since multi-core processors are likely to have a non-uniform cache architecture (NUCA), the communication latency between different cores varies depending on its memory hierarchy. When a group of threads has high data sharing behavior, we can schedule them to the same cluster to utilize the shared cache for data transfer (A cluster is a collection of closely-coupled cores, e.g. two cores sharing the same L2 cache in an Intel Core 2 quad-core processor is termed a cluster). On the other hand, for applications with high bandwidth demands, we prefer to schedule these threads on different clusters to utilize the aggregated bandwidth.

We carefully select the thread affinity policy for the parallel video feature extractions. For instance, in the SIFT application, after row-based parallelization the image chunk assigned to one thread/core will be used by the other threads. Significant coherence traffic occurs when the image data does not reside in cores sharing the same last-level cache. Therefore, scheduling threads to the same cluster will maximally mitigate the data transfer between cores.

## 5 Experimental results and performance analysis

This section presents our experimental results of the video feature extractions' parallelization and

optimization on multi-core systems. To characterize the performance of these programs running on multi-core systems, we investigate them from different aspects, including the processing speed, scalability performance and memory behavior.

## 5.1 Experiment Platform

Our experiment and analysis works are based on two Intel multi-core systems. The first one is a quad-socket dual-core system with total 8 cores and 8GB shared main memory. The second one is a dual-socket quad-core system (HP ProLiant DL380 G5) with total 8 cores and 4GB shared main memory. The first one uses Intel Xeon 7130M CPUs, and the second one uses Intel Xeon E5345 CPUs. The detailed processor information is shown in Table 1.

**Table 1 CPU information of the two multi-core systems**

| CPU model | Intel Xeon Processor 7130M | Intel Xeon Processor E5345 |
|---|---|---|
| CPU type | dual-core | quad-core |
| Core frequency | 3.20GHz | 2.33GHz |
| Bus speed | 800MHz | 1333MHz |
| L1 data cache | 16KB per core | 32KB per core |
| L2 cache | 1MB per core | 2 x 4MB (4MB shared by 2 cores) |
| L3 cache | 8MB shared by 2 cores | none |

These four video feature extraction programs studied in this paper are implemented by using OpenMP programming model with C language. We apply Intel C/C++ Compiler Version 9.1 to compile the four applications into 64-bit binaries with full compiler optimization. For software configuration, the first system use Windows Server 2003 operating system, and the second system use Linux kernel 2.6.5-7.283 – smp (x86_64) operating system. The performance data is collected by Intel performance analysis tools such as the VTune Performance Analyzer and the Thread Profiler[26].

In following experiment we run color correlogram, MRSAR and Gabor feature extractions on the first multi-core system, and run SIFT feature extraction on the second system.

## 5.2 Experiment input data set

For color correlogram, MRSAR and Gabor feature extractions, experiments are based on the TRECVID 2005 developing data sets. The 141-th and 142-th video sequences are drawn to evaluate the performance, which adds up to about one hour MPG-1 (352 × 240 in resolution) videos and contains 791 key frames. Our evaluation is directly running on the 791 extracted key frames.

For SIFT feature extraction program, we use three different kinds of data sets in our experiments as shown in Table 2. One is a MPG-2 image. The F series is 5 images with fixed image size and increased number of keypoints. And the S series is another 5 images with almost the equal quantity of keypoints (about 1000 keypoints for each image) and increased image size.

**Table 2 Label, image size and keypoints number of three kinds of input data set**

| label | image size | keypoints number |
|---|---|---|
| MPG2 | 720 × 576 | 509 |
| F200 | 640 × 480 | 200 |
| F400 | 640 × 480 | 394 |
| F600 | 640 × 480 | 620 |
| F800 | 640 × 480 | 802 |
| F1000 | 640 × 480 | 1028 |
| S600 | 600 × 600 | 1015 |
| S700 | 700 × 700 | 1005 |
| S800 | 800 × 800 | 1001 |
| S900 | 900 × 900 | 1019 |
| S1000 | 1000 × 1000 | 990 |

Our evaluation of SIFT program is based on the 11 images.

To get the data shown in following experiments, we run each experiment 10 times and get the average value as result.

## 5.3 Performance improvement

This section evaluates the performance improvement generated by the parallelization and optimization on these four video feature extractions.

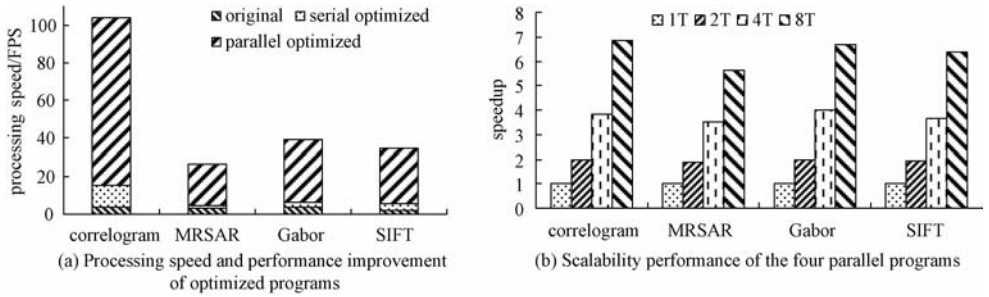
Figure 4(a) shows the processing speeds of original, serial optimized and parallel optimized programs. The parallel programs are running with 8 threads on 8 cores. For color correlogram program, the processing speed of original program is about 3.4 FPS (frames per second) on average; after serial optimization, the processing speed is 15.1 FPS; and after parallelization and parallel optimization, the processing speed achieves 103.7 FPS. For MRSAR program, the processing speeds of original, serial optimized and parallel optimized are 3.2 FPS, 4.7 FPS and 26.3 FPS. For Gabor program, the processing speeds of original, serial optimized and parallel optimized are 3.3 FPS, 5.9 FPS and 39.2 FPS. For SIFT program, the processing speeds of original, serial optimized and parallel optimized are 2.0 FPS, 5.5 FPS and 34.5 FPS.

In summary, after parallelization and optimization in this paper, performance of these four video feature extraction programs achieves 51.0 FPS on average when running on eight cores, which is 17x speedup of original performance. In details, serial optimization in section 2 improves performance 262%. With eight cores, parallelization in section 3 and parallel optimization in section 4 achieve 6.4x speedup averagely based on the serial optimized program, and further contribute another 1440% performance improvement. So in total of serial optimization, parallelization and parallel optimization, performance of these programs running on eight cores is improved to 17x of original programs on average.

## 5.4 Scalability performance analysis

In this section, we evaluate scalability performance of these four video feature extraction parallel programs, and give the time breakdown analysis.

Figure 4(b) shows the scalability performance of the four video feature extraction parallel programs. In this figure, all programs reach almost linear speedups with 2 to 4 threads. As the thread number increases to 8, the speedups of parallel color correlogram, MRSAR, Gabor, and SIFT are 6.8x, 5.6x, 6.7x and 6.4x.




(a) Processing speed and performance improvement of optimized programs

(b) Scalability performance of the four parallel programs

**Fig. 4 Processing speed and scalability performance of the four parallel programs**

To deeply understand the limiting factors of scalability performance, we characterize the parallel performance with the general parallel overheads. The runtime of parallel programs using 8 threads is split to parallel region, sequential region, load imbalance, synchronization penalties, and overheads in Fig. 5. These data is collected by Intel Thread Profiler.

From Fig. 5, we can see that parallel regions dominate in the executions. When these four parallel programs run with 8 threads, parallel regions account for 92.3% of runtime on average while sequential regions account for only 1.4%. This shows the parallelization in this paper covers almost all parts of the

programs. But load imbalance still is a main limiting factor for these parallel programs. Time spent in load imbalance achieves 4.7% of execution time on average. Besides, times spent in synchronization and other overheads are 1.1% and 0.5% of execution time on average.

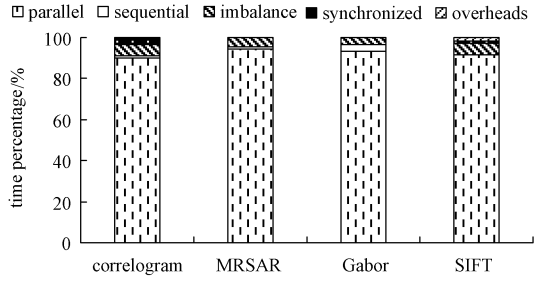



**Fig. 5 Breakdown of runtime spent in parallel, sequential and other overhead when programs run with 8 threads**

As shown in Fig. 5, the time percentage of additional expense for parallelization (including load imbalance, synchronization and overheads) reaches up to 8% for color correlogram and 4% for Gabor. This causes the speedup of the two programs is about 7x with 8 threads.

For MRSAR, the most serious parallel limiting factor is the load imbalance part. This arises from some loops which are consisted of a small number of iterations and are not enough to be evenly distributed between eight threads. However, the parallel region still occupies 94.6% in execution time. The percentage of load imbalance and synchronization is not so large to lead to 5.6x speedup on 8 threads. Based on other experiments, we identify that when running with multiple threads, the number of cache misses of parallel MRSAR is increasing, and this limits parallel MRSAR obtaining higher speedup performance. We will describe its details in next section.

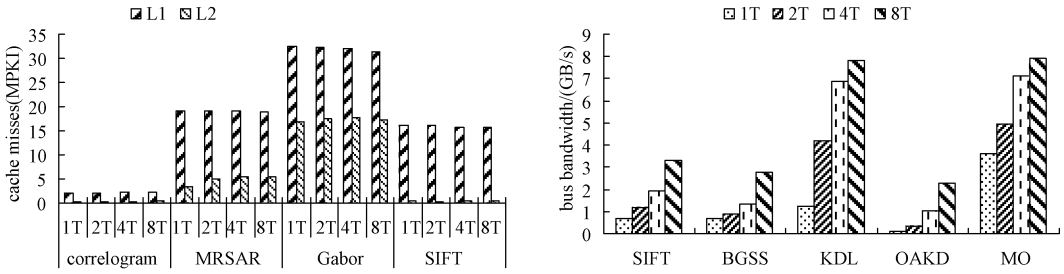
For SIFT, there is a same situation as MRSAR. Although load imbalance occupies 5.5% in execution time, but parallel region still achieves 91.5%. This cannot limit speedup to 6.4x when running with 8 threads. As we can see that the aggregate running time of the parallel regions increases from the single-thread running to the multi-thread running. It is highly possible that some operations run slower in the multi-core configuration than in the single-core configuration for MRSAR and SIFT parallel programs.

## 5.5 Memory behavior analysis

In this section, we show experimental results of memory system performance. Besides the general scalability performance factors shown in section 5.4, memory system also plays an importance role in identifying the scaling performance bottlenecks. For further assurance, we get the memory-hierarchy micro-architectural statistics with the Intel VTune Performance Analyzer, and show them in Fig. 6. Note that we can not get the L3 cache miss data and the bandwidth data on the first multi-core system, since there is no related hardware counters on the Intel Xeon Processor 7130M CPU.

Figure 6(a) shows the L1 and L2 cache misses per kilo instructions (MPKI) of these four video feature extraction parallel programs with different number of threads. We can observe that the cache misses are constant with different threads numbers for color correlogram, Gabor and SIFT. But for MRSAR the L2 cache misses grow as the threads number increases, especially when changing from one thread to two threads. This is because we apply dynamic scheduling for the threads to reduce the load imbalance, but this dynamic scheduling destroys the data locality and raises the cache misses. We speculate the L3 cache misses will increase accordingly, since the working set of each thread is larger than 8MB L3 cache. For this reason the average latency to access data is slower in the multi-thread running than single-thread running. And due to the increase of cache misses and load imbalance, the scalability of MRSAR is a little poorer.

Generally speaking, memory bandwidth is also a key factor which may potentially limit the speedup of multi-thread programs on multi-core systems. Figure 6(b) shows the bus bandwidth utilization of the SIFT parallel programs and its four key modules with different number of threads when input a MPG-2 image. As shown in this figure, the bandwidth utilization of the whole application is not very high (3.3GB/s with eight threads) and nearly increases linearly with the thread number. But for the KDL module and MO module, the




(a) L1 and L2 cache misses of these four parallel programs with different number of threads

(b) Bus bandwidth utilizations of parallel SIFT programs and its four key modules with different numbr of threads

**Fig. 6 Memory behavior of these four parallel programs**

bandwidth utilization with eight threads achieves about 8GB/s, i.e. 38% of the peak bus bandwidth. And the bandwidth utilization with eight threads does not increase much than with four threads. This proves that the bandwidth demands for these two modules are higher than the saturated bandwidth provided by the system. Available bus bandwidth in this system limits SIFT parallel program's speedup to a certain extent.

# 6 Conclusion

Visual feature extractions are key kernels for future video analysis systems. It can help users extract useful visual information from the explosion of video information they are face with. This paper looks at the parallelization and performance optimization of four low-level video feature extractions in CBVIR system, and analyzes their processing speed, scalability performance and memory behavior on multi-core systems.

In this paper, we first present serial performance optimization method for each video feature extraction program, such as removing reduplicative computation and unnecessary computation, improving program locality to increase cache hits, using highly optimized library and Single-Instruction Multiple-Data technique, and reducing bus bandwidth demand etc. Then, we parallelize these video feature extraction programs to take advantage of computing power of multiple cores. After that, we study parallel performance optimization methods to improve scalability performance of these multi-thread programs on multi-core systems.

Experimental results show that parallelization and optimization in this paper speed up performance of these four video feature extraction programs to 17x of original performance on average, when running with eight threads on multi-core systems. In detail, serial optimization improves performance to 2.6x of the original programs; then parallelization and parallel optimization further improve performance to 6.4x of the serial optimized programs when running with eight cores. This proves parallelization and optimization methods studied in this paper are very effective to improve performance of these video feature extraction programs. Furthermore, these optimization methods are representative for video processing applications, and can be widely applied in other programs' performance optimization works on multi-core systems.

In the experiments we also identify the bottlenecks of these parallel programs which limit their scalability performance on multi-core systems. Based on our analysis, the load imbalance, cache coherence misses, and available system bandwidth are main limiting factors to achieve ideal speedup for multi-thread programs running on multi-core system. These limiting factors need we pay more attention when design systems or optimize programs in the future.

## References


[1] Michael S L, Nicu S, Chabane D, et al. Content-based multimedia information retrieval: state of the art and challenges [J]. ACM Transactions on Multimedia Computing, Communications and Applications, 2006, 2(1):1-19.

[2] Martinez J M. MPEG-7 overview:technical report[R]. N6828 ISO/IEC/JTC1/SC29/WG11, 2004.

[3] Cao J, Lan Y, Li J, et al. Intelligent multimedia group of Tsinghua University ad TRECVID 2006 [C]// Proceedings of TRECVID'06. 2006.

[4] Intel® . Multi-core technology[EB/OL]. [2010-07-01]. http://www.intel.com/multi-core/.

[5] Zhang Q, Chen Y, Li J, et al. Parallelization and performance analysis of video feature extractions on multi-core based systems [C]// Proceedings of the 36th International Conference on Parallel Processing. Xi'an, China, 2007.

[6] Zhang Q, Chen Y, Zhang Y, et al. SIFT implementation and optimization for multi-core systems [C]// Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium. Miami, Florida, United States, 2008.

[7] Feng H, Li E, Chen Y, et al. Parallelization and characterization of SIFT on multi-core systems [C]// Proceedings of IEEE International Symposium on Workload Characterization, 2008. Seattle, USA, 2008.

[8] Miao Q, Chen Y, Li J, et al. Parallelization and optimization of a CBVIR system on multi-core architectures [C]// Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium. Rome, Italy, 2009.

[9] Li E, Li W, Tong X, et al. Accelerating video-mining applications using many small, general-purpose cores [J]. IEEE Micro, 2008, 28(5):8-21.

[10] Li W, Tong X, Wang T, et al. Parallelization strategies and performance analysis of media mining applications on multi-core processors [J]. Journal of Signal Processing Systems, 2009, 57(2):213-228.

[11] Hong C, Chen W, Zheng W, et al. Parallelization and characterization of probabilistic latent semantic analysis [C]// Proceedings of the 37th International Conference on Parallel Processing. 2008:628-635.

[12] Snavely A, Tullsen D M, Voelker G. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor [C]// Proceedings of the SIGMETRICS'02. 2002:66-76.

[13] Fedorova A, Seltzer M, Small C, et al. Performance of multithreaded chip multiprocessors and implications for operating system design [C]// Proceedings of the USENIX'05. 2005:26.

[14] Fedorova A, Seltzer M, Smith M D. Improving performance isolation on chip multiprocessors via an operating system scheduler [C]// Proceedings of the PACT'07. 2007:25-38.

[15] Chen S, Gibbons P B, Kozuch M, et al. Scheduling threads for constructive cache sharing on CMPs [C]// Proceedings of the SPAA'07. 2007:105-115.

[16] Lee R, Ding X, Chen F, et al. MCC-DB: Minimizing cache conflicts in multi-core processors for databases [C]// Proceedings of the VLDB'09. 2009.

[17] Sutter H. The free lunch is over: A fundamental turn toward concurrency in software [J]. Dr Dobb's Journal, 2005, 30(3).

[18] Huang J, Kumar S R, Mitra M, et al. Spatial color indexing and applications [J]. International Journal of Computer Vision, 1999, 35(3):245-268.

[19] Mao J, Jain A K. Texture classification and segmentation using multi-resolution simultaneous autoregressive models [J]. Pattern Recognition, 1992, 25(2):173-188.

[20] Manjunath B S, Ma W Y. Texture features for browsing and retrieval of image data [J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 1996, 18(8):837-842.

[21] Lowe D G. Distinctive image features from scale-invariant keypoints [J]. International Journal of Computer Vision, 2004, 60(2):91-110.

[22] Ma W Y, Zhang H J. Benchmarking of image features for content-based retrieval [C]// Conference Record of the 32nd Asilomar Conference on Signals, Systems & Computers. Pacific Grove, CA, 1998:253-257.

[23] Xu K, Georgescu B, Comaniciu D, et al. Performance analysis in content-based retrieval with textures [C]// Proceedings of the 15th International Conference on Pattern Recognition. 2000:4275-4279.

[24] Campbell M, Ebadollahi S, Naphade M, et al. IBM research TRECVID-2006 Video retrieval system [C]// Proceedings of TRECVID'06. 2006.

[25] Lee T S. Image representation using 2D Gabor wavelets [J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 1996, 18(10):959-971.

[26] Estrada F, Jepson A, Fleet D. Local features tutorial: technique report[R/OL]. (2004) [2010-07-01]. http://www.cs.toronto.edu/~jepson/csc2503/tutSIFT04.pdf.

[27] Intel Corporation. Intel® . VTune™ performance analyzer[CP/OL]. [2010-07-01]. http://www.intel.com/software/products/vtune/.

[28] Matteo F, Steven G J. The design and implementation of FFTW3[C]// Proceedings of the IEEE, 2005, 93(2): 216 – 231.

[29] Intel Corporation. Intel® . Math kernel library[CP/OL]. [2010-07-01] http://www.intel.com/software/products/mkl/.

[30] Intel Corporation. Intel® . 64 and IA-32 architectures optimization reference manual [M]. CA, USA: Intel Corporation, 2006.

[31] Hugher C J, Grzeszczuk R, Sifakis E, et al. Physical simulation for animation and visual effects: parallelization and characterization for chip multiprocessors [C]// Proceedings of the 34th Annual International Symposium on Computer Architecture. 2007.

[32] The OpenMP Architecture Review Board. OpenMP C and C++ application program interface Ver 2.0 [CP/OL]. (2002-03) [2010-07-01]. http://www.openmp.org/mp-documents/cspec20.pdf.

[33] Intel® . Thread checker[CP/OL]. [2010-07-01]. http://software.intel.com/en-us/intel-thread-checker/

[34] Salehi J D, Kurose J F, Towsley D F. The effectiveness of affinity-based scheduling in multiprocessor network protocol processing [J]. IEEE/ACM Transactions on Networking, 1996, 4(4):516-530.

# 基于多核系统的视频特征提取程序并行化及性能优化方法

张 琦[1], 陈玉荣[2], 李建国[2], 胡云[1], 许胤龙[1]

(1 中国科学技术大学计算机科学与技术学院, 合肥 230026;
2 英特尔中国研究院, 北京 100080)

**摘 要** 基于多核系统,对 4 种视频特征的提取程序分别研究了并行算法和性能优化方法. 实验结果表明,通过的并行化和性能优化,当使用 8 个核时,这 4 种视频特征提取程序的处理速度平均提高到原始串行程序的 17 倍. 此外,对实验结果进行了深入的性能分析,寻找和剖析了多核系统的性能瓶颈,为进一步提高多核系统的性能提供了依据和建议.

**关键词** 程序性能优化, 多核系统, 视频特征提取