Parallel And SIMD Optimization Of Image Feature Extraction

Ming Qi, Guangzhong Sun, Guoliang Chen

School of Computer Science and Technology, University of Science and Technology of China
qiming@mail.ustc.edu.cn, {gzsun, glchen}@ustc.edu.cn

Abstract

Image feature extraction is widely used in content-based image retrieval (CBIR), computer version and all kinds of image processing applications. In this paper, we introduce some parallel and SIMD optimizations of image-feature extraction to overcome the disadvantages of original methods. We mainly use both thread-parallel optimization and Single Instruction Multiple Data (SIMD) optimizations. And especially, for some hot point, we use SIMD logical operations to eliminate the random conditional branches which cannot be effectively predicted by CPU Branch Prediction. We experimented our optimized implementation on multi-core systems, and various images were used to test the image-feature extraction results and performance. All the parallel and SIMD optimizations work out a good cumulative performance speedup.

Keywords: CBIR, image-feature extraction, optimization, TLP, ILP

1. Introduction

In recent years there has been a rapid increase in the scale of digital image collections on the Internet. At the same time, what growing much faster is video, including a variety of personal video, public video, monitor video and so on.

In science fields, there are satellite remote sensing images or microscope images; in medicine and health care, there are X-ray or infrared or CT images; in security agencies, there are figure prints images and monitor videos. All these images sets are millions, billions or even more.

1.1. Image Feature Extraction Applications

There are many applications in which image feature extraction is involved: content-based image retrieval (CBIR), computer version, and all kinds of image processing applications such as in the fields above.

CBIR is to overcome the limits of the traditional tag-based image retrieval. CBIR retrieves images by visible information, such as color, texture, and for some special images, shape. Since the size of image set in CBIR system is always enormously huge, image-feature extraction is a very time-consuming step.

Besides CBIR, image feature extraction has many other applications, especially when we can do it in real-time. As we can see in the experiments (see 4.2), because of our SIMD optimization, image color and edge feature extraction...
can be done in about 6ms per image, which is enough for real-time image processing. With more and more cores in one CPU chip, real-time image-based applications will be more and more popular.

These applications could be:
(a) Image filtering. According to the color and edge feature vector, we can filter out the image with some bad characteristics, such as inadequate exposure or over-exposure photographs, photos with too much noise, and so on.
(b) Automatic image processing. According to the color and edge features, we can automatically treat different image differently, such as change image color contrast, strengthen or weaken the edge of the images, or generate any other visual styles or effects.
(c) Automatic monitoring and capturing. Extracting features of images from surveillance cameras in real-time, will help to store the surveillance images at the right moment as evidence, or to do other responses.
(d) Automatic computer vision. Extracting features of camera images in real-time, and maybe doing some correlative image processing, will help computer image pattern recognition and computer vision.

However, most of the existing image-feature extraction method do not care much about performance[1] [7], and do not take notice on the utilization of modern highly-developed computer architecture. Although many systems parallelize the application by introducing multi-thread, sophisticated SIMD optimizations about image-feature extraction have not been considered before. [2] [9]

In this paper we introduce some optimized methods of image feature extraction, including both thread level parallelism(TLP) and SIMD instruction level parallelism(ILP). We analyze and test the performances before and after the optimizations, and our experiment indicates that a good speedup ratio is gained by TLP, and SIMD optimizations also improve the performance a lot.

1.2. Parallel Computing on Multicore

When we realized that the Moore’s Law comes to a dead end, that is we cannot gain a higher performance by simply rise the frequency of a single core CPU, we transferred our focus to multicore CPU and parallel computing. A multi-core processor is a CPU chip with two or more computing cores, which are independent but connected by very high speed local bus. Most multicore chips have a shared L2 Cache and separated L1 Caches. And all the cores in one chip, of course share the same main memory.

Although many famous operation systems and application softwares can finely support multi-thread and multi-processor, but they have not really utilized the benefit of multicore in one processor chip, and they cannot distributing the computation load to computing cores equilibrately and elastically. Programming models on multicore computers are being earnestly researched but not well developed.

However, The multicore processor hardware is developing much faster and multicore software and application. According the development roadmap of Intel and AMD, CPU chip with about 100 cores is not very far future, and “manycore” chip with 1000 cores is being sketched. All these CPU revolutions will bring a very large challenge to the software architectures and programming methods.

The traditional parallel programming on distributed system is MPI (Message Passing Interface), which is based on message-passing between different computing nodes. However, in a multicore system, OpenMP is the current best choice. OpenMP is a programming model in share-memory parallel structure, which provides a set of platform-independent functions and compiler instructions to generate a flexible share-memory multi-thread program.

1.3. SIMD Optimization

Along with increasing the number of computing cores to run more parallel threads, the CPU instruction level parallelism(ILP) is also further developed. Single Instruction Multiple Data(SIMD) instruction sets, such as MMX(i.e. MultiMedia eXtensions), SSE(i.e. Streaming SIMD Extensions), SSE2 and SSE3, are commonly supported by x86 architecture CPUs.

Inside CPU cores, SIMD registers with 64 or 128 bits are used to manipulate multiple data with single instruction in one instruction cycle. Intel IA-32 CPU has eight 64-bit MMX registers and eight 128-bit SSE registers, while Intel EM64T 64-bit CPU contains 16 128-bit registers. AMD CPU support similar SIMD instruction in 3DNow! instruction set. These SIMD instruction sets provide a huge resource to explore ILP parallel to the application program, especially for multimedia processing programs, which are to process multimedia data in a very large scale.
However, there are very few multimedia applications really utilized the advantage of multicore thread-parallel as well as SIMD parallel in instruction level. The mainly reason is that both the two level parallelism are not easy for common programmers, so it is even harder to combine these two advantages together.

The latest Intel C++ compiler attempts to provide multi-threaded programming and SIMD optimization at the same time. This compiler supports the OpenMP syntax, as well as SIMD code generation (including MMX, SSE, SSE2, SSE3), by supporting embedded assemble language statements and a set of intrinsics functions.[4]

1.4. Outline of the Paper

In Section 1, we simply introduced the application background of image feature extraction and the background of parallel computing on multicore system. Section 2 describes the methods and principles of image feature extracting and its serval problems, which are bottlenecks of its performance. To overcome the problems, in Section 3, we mainly describe four optimization points including both thread and SIMD parallelization and optimization. In Section 4, we analyze, test and compare the performances before and after the optimizations. And in Section 5, we briefly conclude our work.

2. Image Feature Extracting Methods and Existing Problems

Image feature extraction is one of the most CPU consuming steps in the many image processing applications, because this process is for every image, and there are billions of images to be done.

Figure 1: Image Sections for Image Feature Extraction

Looking at a common picture, for example a photo, the main theme figure is usually on the foreground and in the middle of the picture, which usually is what we want to see in the picture. And the background scenes above and below the horizon, and scenes on the left and right side, are usually different. Look at the Figure 1, in the middle of the photo there is the main figure, that’s a house, and the scenes in left and right, up and down are rather different in color and texture.

Besides, we cannot afford advanced image recognition in a general-used application like CBIR. So, we simply segment the image into five different sections, that is: left-top, right-top, left-bottom, right-bottom, and the central
section (the central section overlaps other sections). Each of the 5 sections is \((1/2) \times (1/2)\) the size of the origin picture.

To extract the edge features, we use a well known method called Canny operator. [5] The steps of Canny operator edge feature extraction are like this:

Step 1: Image Smoothing, which is to filter high frequency noises in the image. This is a preparing step of step 2, since high frequency noises can destroy gradient-based edge enhancing.

Step 2: Differential Gradient Generating, which is to compute differential gradient between each pixel and its neighbors in \(x\) and \(y\) directions, \(\Delta x\) and \(\Delta y\). And the absolute grad value is \(\sqrt{\Delta x^2 + \Delta y^2}\).

Step 3: Non-maxima Suppression. The gradient of the edge pixels must be max value in a certain direction of its 8-neighbor domain. So, suppress the non-maxima pixels can reveal the pixels along the edges and help to find the right edge. We use a high-value threshold and a low-value threshold to disperse the pixel strength into two categories. Pixels whose value under the low-value threshold are neglected.

Step 4: Edge detecting, which is to trace along the edge. The tracing begins with pixel above high-value threshold and among the pixels which satisfy non-maxima suppression. When tracing an edge, the strength and direction of the edge pixels, are to be recorded.

Step 5: Text feature vector building. There are 8 direction categories with angles dividing 360-circle into 8 equal parts. And, edge strength values are dispersed into 8 categories. So the entire edge feature vector dimensions count \(8 \times 8\times 5\).

To extract the color feature is easier. We disperse 24-bits RGB colors into 64 different color ranges. So the dimension of entire color feature vector counts \(64 \times 5\).

There are several problems in these methods. Generally, these methods are neither optimized for the parallel implementation, nor suitable for multi-core or multi-threaded environment. At the same time, algorithms of methods are not designed for the SIMD instruction, and can not use the powerful CPU instruction-level parallelism. The SIMD ignoring problem is rather prevalent in almost all the steps.

In the step “Image Smoothing”, the origin algorithm is to calculate pixels’ average in its \(3 \times 3\) neighborhood. There are some redundancy in the algorithm, and the calculation is hard to use SIMD.

In the step “Non-maxima Suppression”, we need to compare every pixel’s gradient value between its 8 neighbors in 4 directions. It involves too many random conditional branches, which cannot be predicted effectively by CPU hardware branch prediction, and of course, are not suitable for SIMD. This problem make “Non-maxima Suppression” the bottleneck and hottest spot of the process.

In color feature extracting, problems are very similar, random conditional branches must be involved during color reduction, and the work must be done serially, one pixel after another.

As we can see in next section, all these problems can be overcome. We can eliminate the random conditional branches, and they can be done in a SIMD way, at least four pixels at a time. And at the same time in high level, thread parallelism is used flexibly and scalability on multi cores.

3. Optimization Methods of Image Feature Extraction

The primary idea of the optimization is to take both advantages of TLP and ILP together. On thread level, we use OpenMP thread parallel on multicore system; and on instruction level, we use CPU SIMD instructions. The high level of the optimized methods of image feature extracting is as Figure 2.

3.1. Adaptive Coarse-grained TLP

The main idea of thread-level parallel is coarse-grained thread parallelism, that is, each working CPU thread deals with one image’s feature extraction. Since all the images are independent, there is no share information among parallel threads, and no synchronizing control is needed. At the same time, the dependence between the threads is minimum, and the program has a good scalability. So the overall process can achieve the largest throughput on the multi-core systems, while the speed-up ratio has been basically equal the number of threads.

We use OpenMP dynamic thread allocation to schedule parallel threads among CPU cores. In OpenMP methods, the number of working threads can always be equal to the number of available CPU cores, so that the TLP can adapt and scale automatically.
Figure 2: High level process of optimized image feature extracting
However, when the image size is larger but with small amount of images to deal with, there are two possible strategies: one is to scale the image within a normal range (typically 600 × 500 pixels), which will lose some details of the image; another scheme is to do a fine-grained thread-level parallelization, which could have a fast response as well as a complete image feature information.

3.2. **SIMD optimization on Image Smoothing and Gradient Generating**

Image Smoothing is to eliminate the high-frequency noise of an image and prepare for the gradient generating. The original image smoothing algorithm is a classical Gaussian Filter, which needs high computational cost and returns a good visual effect. But for the image edge extraction, the image smoothing job does not require good visual effect, and can be simplified to an Average Filter, which is a special case of Gaussian Filter.

The origin algorithm is to calculate pixels’ average in its 3 × 3 neighborhood. There are some redundancies in the algorithm, and the calculation is hard to use SIMD.

We can replace the 2-dimensional pixel average operation by 1-dimensional 3 pixel average in both horizontal and vertical directions. And in the average calculating, we can detach the multiplication and addition, that is, multiplied all the pixel by 1/3, and later sum the center pixel and its two neighbors. In this way, all the multiplication and addition can be in a SIMD style, that is, we treat four pixels in one CPU SIMD instruction.

We can implement this SIMD optimization manually by SIMD CPU instruction or Intel SIMD intrinsics functions.[4]

More intelligently, if we can sure the memory of the image pixel structure is 128-bit aligned, the compiler can automatic parallel the multiplication and addition calculation into SIMD code. So, we take the advantage of this automatic SIMD parallelism in our implementation.

In the step “Differential Gradient Generating”, the calculation is similar. Because in our optimization the image pixel structure is specially designed 128-bit aligned by line, the differential calculation in fact is a SIMD 128-bit subtraction, which contains four 32-bit signed integer subtractions.

3.3. **Eliminate conditional branch and SIMD optimization on Non-maxima Suppression**

The edge pixels must have max gradient value in a certain direction among its eight neighbors, that is horizontal, vertical, or either two diagonal directions. The pixel’s symbol expression and maxima comparisons directions are shown in Figure 3.

![Figure 3: 8-neighborhood and 4 directions in Non-maxima Suppression](image)

As we discussed before, the traditional way of maxima comparisons in the four directions is:

```plaintext
if... else....
```

Conditional branches must be involved in this implementation. Even worse, these conditional branches are almost random because of the fluctuation of the image, which eventually make “Non-maxima Suppression” the hottest spot of the edge feature extraction process.

In fact, we can eliminate conditional branch or conditional assignment by SIMD bit logical operations. There are 128-bit SIMD integer comparison operations in SSE2, for example, **PCMPGTQ** instruction compares the 4 signed 32-bit integers in operand A and the 4 signed 32-bit integers in operand B for greater than. And the return of the instruction is a 128-bit mask, with four 32-bit data with all 1s or 0s, depending on the result of the logic comparison. Base on the mask of the SIMD comparison operations, we can use SIMD logical operations, such as **PAND** and **POR**, to finally work out the Non-maxima Suppression result according the algorithm.

In our SIMD optimization, pixels in Figure 3 are all represent four pixels, we call it “packed pixel”, that is, 128-bit data for four integers. Because the pixel data structure is aligned by lines, the “packed pixel” C, U and D are already
128-bit aligned, and the “packed pixel” UL, UR, DL, DR can come from C, U, D and the neighbor pixels by shifting and coping.

![Figure 4: Logical Operations to the Non-maxima Suppression Result](image)

Figure 4 shows the SIMD logical operations to get the Non-maxima Suppression results. The whole logical expression is:

\[
\begin{align*}
((L > C) \text{ and } (U > C)) \text{ or } \\
((U > C) \text{ and } (D > C)) \text{ or } \\
((UL > C) \text{ and } (DR > C)) \text{ or } \\
((DL > C) \text{ and } (UR > C)) \text{ or }
\end{align*}
\]

The SIMD logical operations returns the mask of Non-maxima Suppression, whose all 1s represents the pixels could be on the edge, and all 0s represents not.

As Figure 4 shows, the SIMD logical operations treats 4 pixels at one time. So, we can eliminate all the random conditional branches and use SIMD ILP at the same time.

To be convenient for next step edge detecting, we change the mask result into a constant pixel value 128, by SIMD and operations:

\[
\text{mask and vec128}
\]

3.4. SIMD optimization on Image Color Feature Extraction

In color feature extraction, we assume that the image are all 24-bit color, that is, three 8-bit unsigned integers(0-255) representing the primitive colors of red, green, and blue. If all the 24-bit color information is recorded, the feature vector will be too long and has too much details. In fact, we reduce 24-bit color into 64 different color ranges, that is 2-bit data for each primitive color.

To avoid conditional branches and do more optimization, we use SSE2 logical shift-operation instructions do this job in SIMD style. First, we transport the 24-bit color data into three different aligned bite arrays, each of which represents one color red, green or blue. Then, all bytes in the red color array logical shift right 6 bits and shift left 4 bits, that is, only the first two bits in the original byte are reserved. Similarly, all bytes in the green color array logical shift right 6 bits and shift left 2 bits, and all bytes in the blue color array logical shift right 6 bits without shifting left.
Finally, reduce all the three color arrays into the result array, using logic OR operation. This process is shown in Figure 5.

Figure 5: Bit Shifting for Color Reducing

All the shift-operation and logical operation can be done in SIMD. By using 128-bit SSE2 instruction PSLLW, PSRLW and POR, we can treat 8 pixels at the same time.

4. Performance Modeling and Experiments

4.1. Performance Modeling

We can simply model the performance before and after the optimization.

In the high level, we can simply use the Amdahl Law. In Amdahl’s equation, where $W_s$ and $W_p$ is respectively the serial and parallelizable part of the application program, and $p$ is the number of the parallel processing units, the speedup radio of parallelization is

$$R_{TLP} = \frac{W_s + W_p}{W_s + \frac{W_p}{p}}.$$

In our optimization scheme $W_s$ consists of the thread dispatching operation, which can be effectively implemented by OpenMP dynamic thread allocation and scheduling, and serial operations of input and output data source, which may be not exist if images input is not from input stream but file system and data output file is parallel writable. Therefore, $W_s$ is almost neglectable, and the high level speedup will almost equal to the number of working CPU cores.

In SIMD instruction level parallel, as we can see in our scheme, things are different, because the SIMD parallel program is not serial version accelerated by a factor of processing units, but some instructions instinctively different. In some SIMD parallelization, SISD arithmetic instructions can be replaced by SIMD instructions, which pack four or eight data units into a 128-bit “packed data” and execute in one instruction cycle. Sometimes, however, additional computing has to be introduced, and in some special circumstances, for example while using SIMD to eliminate random conditional branches, the serial program transforms to a completely different form, where the speedup factor is unknown before test.

In SIMD optimization scheme, let $S$ denotes the sum of all the serial operations, which is immutable, and $t_i$ is the $i$th part of operation to be optimized, and $t'_i$ is the corresponding SIMD optimized version. Then, the speedup ratio of SIMD optimization is

$$R_{SIMD} = \frac{S + \sum t_i}{S + \sum t'_i}.$$
If SIMD instructions come from packing original data, which usually involves loop unrolling, data packing and data aligning, the workload (or cost time) can be regarded as

\[ t'_i = t_i / q + e_i, \]

where \( q \) is the data packing ratio, often 4 (four 32-bit data packed) or 8 (eight 16-bit data packed), and \( e_i \) denotes the additional workload because of SIMD optimization, for example, data aligning. In this case, the optimization is reasonable if and only if

\[ e_i < (1 - 1/p) t_i. \]

Usually, \( e_i \) is a constant, and data packing SIMD optimization is worthwhile.

If the SIMD optimization are more speculated, such as eliminate conditional branches or remove unnecessary slow operations, the optimization makes the original program reformed, so that there is no obvious analytic relations between the programs before and after optimization. In this case, we should test both \( t_i \) and \( t'_i \) and make sure the optimization is worthwhile.

### 4.2. Experiments and Results

In our experiment, we used a common 2-CPU server with shared memory; each CPU is 4-core Intel Xeon E5430 2.66GHz. The reason why we use a common server rather a super computer is that, in Internet applications dealing with massive data set, using a cluster of common servers usually receives higher cost-effectiveness.

#### Table 1. Performance speedup.

<table>
<thead>
<tr>
<th></th>
<th>1000 images</th>
<th>edge feature</th>
<th>color feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-optimized</td>
<td>10055.38 ms</td>
<td>1825.78 ms</td>
<td></td>
</tr>
<tr>
<td>only SIMD</td>
<td>5299.89 ms</td>
<td>723.21 ms</td>
<td></td>
</tr>
<tr>
<td>only TLP</td>
<td>1222.83 ms</td>
<td>233.48 ms</td>
<td></td>
</tr>
<tr>
<td>TLP and SIMD</td>
<td>936.90 ms</td>
<td>230.01 ms</td>
<td></td>
</tr>
</tbody>
</table>

We timed the implementation with both TLP and SIMD optimization, only TLP, only SIMD and non-optimized version. Each of the test set has 1000 images, and Table 1 shows how much time used on different optimization level when the image size is 256 \( \times \) 384. The edge and color feature information extraction on different optimization level and image size is shown in Figure 6. By the image size, we mean the size of image to process the feature extraction.
In the real application, if the size of the image is larger than the threshold, this image will be scaled to be within a proper size.

TLP optimization generally speedup ratio is 8 and 7.8, nearly the linear speedup, because the adaptive coarse-grained TLP used. In edge feature extraction compared with TLP only, about 24% time are saved by SIMD ILP optimization, because the original hot spot is optimized.

In the level of thread parallel parallelization, the partition is based on image processing of tasks. Since every image is processed by only one thread, there is no data dependency between threads, and thus does not require synchronization between threads; and so in this experiment, the communications overhead among threads can be ignored. In the equation of Amdahl’s law, the serial parts of the program can be seen as $W_s = 0$, so that the parallel program will be linear speedup ($R_{TLP} = p$), which is verified by the experiment results, and this means that the multicore parallel computing resources are fully utilized.

In the color feature extraction, the SIMD speedup is not markable when the image size is relatively small. The main reason is that conditional branches in color feature extraction are not so random as they in edge feature extraction, and the advantage of SIMD optimization is mostly counteracted by its cost, when the image size is small. When the image size is larger, SIMD optimization shows a good improvement.

5. Conclusions

Image feature extraction is involved as a key step by various applications, and how to utilize modern multicore CPU computers to accelerate this process is important. In our paper, we overcome some performance problems of image feature extraction and put forward an overall optimized method, which uses both high-level multi-core TLP optimization and ILP optimization by CPU SIMD instructions. We analyzed, tested and proofed its favorable performance acceleration, so that all the applications related will benefit from our optimization.

At the same time, we have successfully tried a methodology which is integrating TLP and SIMD ILP optimization, to utilize both multi-core feature and SIMD feature of modern CPU.

Acknowledgements

The authors would like to thank reviewers for many helpful advice and assistance. This work is supported by the National Natural Science Foundation of China (No.61033009 and No.60873210). This work is also supported by “the Fundamental Research Funds for the Central Universities”.

References