J Supercomput (2012) 59:1455–1477
DOI 10.1007/s11227-011-0556-x

Parallel implementation of Gray Level Co-occurrence Matrices and Haralick texture features on cell architecture

Asadollah Shahbahrami · Tuan Anh Pham · Koen Bertels

Published online: 2 February 2011

Abstract Texture features extraction algorithms are key functions in various image processing applications such as medical images, remote sensing, and content-based image retrieval. The most common way to extract texture features is the use of Gray Level Co-occurrence Matrices (GLCMs). The GLCM contains the second-order statistical information of spatial relationship of the pixels of an image. Haralick texture features are extracted using these GLCMs. However, the GLCMs and Haralick texture features extraction algorithms are computationally intensive. In this paper, we apply different parallel techniques such as task- and data-level parallelism to exploit available parallelism of those applications on the Cell multi-core processor. Experimental results have shown that our parallel implementations using 16 Synergistic Processor Elements significantly reduce the computational times of the GLCMs and texture features extraction algorithms by a factor of 10× over non-parallel optimized implementations for different image sizes from 128 × 128 to 1024 × 1024.

Keywords Texture feature extraction · Co-occurrence matrix · Parallel techniques · Cell architecture

1 Introduction

Texture is a significant feature of an image that has been widely used in medical image analysis, image classification, automatic visual inspection, content-based image retrieval, and remote sensing [1–3]. Textures are generally complex visual patterns

A. Shahbahrami (✉)
Department of Computer Engineering, Faculty of Engineering, University of Guilan, Rasht, Iran
e-mail: shahbahrami@guilan.ac.ir

T.A. Pham · K. Bertels
Computer Engineering Laboratory, Faculty of EEMCS, Delft University of Technology, 2628 CD Delft, The Netherlands

composed of entities, or sub-patterns that have characteristics such as brightness, color, slope, and size. Texture features can be extracted in several methods, namely: statistical, structural, model-based, and transform information. Each method has different techniques. A well-known algorithm to extract texture features is the use of Gray Level Co-occurrence Matrices (GLCMs), which belongs to the statistical methods [4]. The GLCM contains the second-order statistical information of spatial relationship the pixels of an image. Haralick [4] has extracted many statistical features known as Haralick texture features using the GLCMs. However, the main drawback is that the computation of the GLCMs and texture features are computationally intensive and time-consuming. For example, the experimental results in [5] showed that the calculation time for GLCMs and Haralick texture features for an image of size 5000×5000 is approximately 350 seconds using Pentium 4 machine running at 2400 MHz. Therefore, implementing the mentioned algorithms using parallel techniques on a parallel architecture such as the Cell multi-core processor is an excellent alternative to provide high performance texture analysis.

The Cell Broadband Engine (Cell BE) [6] is a heterogeneous multi-core architecture with nine processors specialized into two types: one PowerPC Processor Element (PPE) and eight Synergistic Processor Element (SPE). The first type of processor element, PPE, is a 64-bit PowerPC architecture core. The second type of processor element, the SPE, is optimized for running computation-intensive Single Instruction Multiple Data (SIMD) applications. In order to maximize the performance on the Cell architecture the following steps are used. First, multiple SPEs are used in parallel to execute either different programs or different parts of a program and second, SIMD vectorization is applied on each SPE to execute a single instruction on different data. In other words, task- and data-level parallelism can be exploited using multiple SPEs and each SPE, respectively. Our experimental results have shown that the implementing the GLCMs and texture features on the Cell architecture using 16 SPEs the speedups of up to $10\times$ is yielded over non-parallel optimized implementations.

We make the following contributions compared to other research works.

- We propose parallel implementations of GLCMs and Haralick texture features on the Cell processor, which is one of the latest high performance embedded processors. We exploit both task-level parallelism using task partitioning and data-level parallelism using code optimization with SIMD instructions.
- The GLCM is an irregular application, whose computational parts hardly fit into parallel architectures. The access patterns to the data of GLCM are non-aligned and irregular. In order to paralize it using existing SIMD instructions, we apply large data type concept, which defines each element of co-occurrence matrix as a vector type. This technique improves performance compared to the scalar implementation from 1.2 to 1.6 depending on the number of working SPEs.
- Optimized implementations on the Cell architecture using different numbers of SPEs show fast computation of the GLCMs and Haralick texture features and it is almost $10\times$ times faster than the Intel processor.
- We observe that when the size of workloads, image size, number of gray level, is small the communication and synchronization times between different cores become dominant compared to the computational time. On the other hand, when the size of workloads is large, the computational time becomes dominant in comparison with the synchronization time.

This paper is organized as follows. In Sect. 2 we have collected much background information. It describes statistical texture features extraction algorithms, gray level co-occurrence matrix, Haralick texture features, and the Cell architecture. Related work is discussed in Sect. 3. Parallel implementations of the GLCMs and Haralick texture features on the Cell architecture are presented in Sect. 4. Experimental results are presented in Sect. 5. Conclusions are drawn in Sect. 6.

2 Background

Background information about statistical texture features extraction algorithms and Haralick texture features and the Cell architecture are presented in this section.

2.1 Statistical texture features

Statistical texture features extraction algorithms represent textures based on the distributions and relationships between image pixels. These algorithms can be classified into first-, second-, and higher-order statistics. The difference between these classes is that the first-order statistics estimate properties, e.g., average and variance, of individual pixel values by waiving the spatial interaction between image pixels, but in the second-order and higher-order statistics estimate properties of two or more pixel values occurring at specific locations relative to each other. The most popular second-order statistical features for texture analysis are derived from the GLCMs [7, 8]. In the following section, we describe this algorithm in detail.

2.1.1 Gray Level Co-occurrence Matrix

In 1973, Haralick [4] introduced the co-occurrence matrix and his texture features which are the most popular second-order statistical features today. Haralick proposed two steps for texture feature extraction: the first is computing the GLCMs and the second step is calculating texture features using the calculated GLCMs. This technique is useful in a wide range of image analysis applications from biomedical [2, 9] to remote sensing techniques [3].

The GLCMs define the probability of joining two pixels i and j, with distance d and an orientation angular θ. It is denoted by $p_{d,\theta}(i, j)$. The element (i, j) of $p_{d,\theta}$ is the number of occurrences of the pair of gray levels i and j whose distance is d in direction of θ. A GLCM for an image of size $N \times M$ with N_g gray levels is a 2D array of size $N_g \times N_g$. For example, Fig. 1 depicts a GLCM for an image of size 4×4 with $N_g = 5$. This 2D array is computed with $d = 1$ and $\theta = 0$. The co-occurrence matrix can be normalized by dividing each element by the number of pixels in an image. The GLCMs can be defined in eight directions (0, 45, 90, 135, 180, 225, 270, 315) as these directions depicted in Fig. 2.

In other words, the co-occurrence matrix can be computed using two techniques. First, image pixels are separated by d and $-d$ for a given direction (θ) in four directions $(0, 45, 90, 135)$. Second, image pixels are separated by distance d in eight

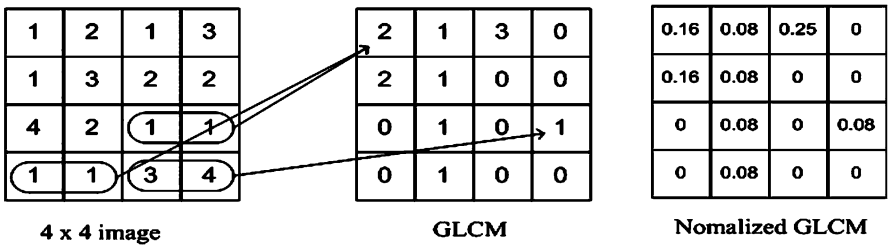

Fig. 1 Calculating a GLCM for an image of size 4×4, which the number of its gray level (N_g) is 5 with $d = 1$ and direction $\theta = 0$

Fig. 2 Eight directions of adjacency

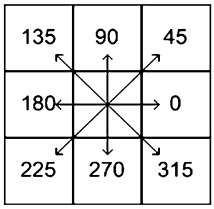

Fig. 3 The C implementation of the GLCM with consideration of 8 neighboring pixels

```
void CoOccurrence_Matrix_Calculation_8() {
 for (i=1; i<=N; i++)
   for (j=1; j<=M; j++) {
      cooccurance[img[i][j]][img[i-d][j-d]]++;
      cooccurance[img[i][j]][img[i  ][j-d]]++;
      cooccurance[img[i][j]][img[i+d][j-d]]++;

      cooccurance[img[i][j]][img[i-d][j]]++;
      cooccurance[img[i][j]][img[i+d][j]]++;

      cooccurance[img[i][j]][img[i-d][j+d]]++;
      cooccurance[img[i][j]][img[i  ][j+d]]++;
      cooccurance[img[i][j]][img[i+d][j+d]]++;
   }
 }
```

directions $(0, 45, 90, 135, 180, 225, 270, 315)$ [8, 10]. Fine textures need small values of d, while coarse textures require large values of d. The C implementation of the GLCM with 8 directions is depicted in Fig. 3 and it is referred to as *array structure* technique in this paper.

Although the computational complexity of the co-occurrence matrix for an image of size $N \times N$ is only $O(N^2)$, the computational power requirements to compute multiple co-occurrence matrices, which are needed in many applications such as medical image processing are significantly large [11].

2.1.2 Haralick texture features

From the computed GLCMs, Haralick proposed a number of useful texture features. These features are as follows:

$$f_1 = \sum_{i=0}^{N_g-1}\sum_{j=0}^{N_g-1} p_{d,\theta}(i,j)^2, \tag{1}$$

$$f_2 = \sum_{n=0}^{N_g-1} n^2 \left\{ \sum_{i=0}^{N_g-1}\sum_{j=0}^{N_g-1} p_{d,\theta}(i,j) \right\}, \quad \text{where } n = |i-j|, \tag{2}$$

$$f_3 = \sum_{i=0}^{N_g-1}\sum_{j=0}^{N_g-1} p_{d,\theta}(i,j)\log(p_{d,\theta}(i,j)), \tag{3}$$

$$f_4 = \sum_{i=0}^{N_g-1}\sum_{j=0}^{N_g-1} (i-\mu)^2 p_{d,\theta}(i,j), \tag{4}$$

$$f_5 = \sum_{i=0}^{N_g-1}\sum_{j=0}^{N_g-1} p_{d,\theta}(i,j)\frac{(i-\mu_x)(j-\mu_y)}{\sigma_x\sigma_y},$$

$$\mu_x = \sum_{i=0}^{N_g-1}\sum_{j=0}^{N_g-1} i.p_{d,\theta}(i,j), \quad \mu_y = \sum_{i,j=0}^{N_g-1}\sum_{i,j=0}^{N_g-1} j.p_{d,\theta}(i,j),$$

$$\sigma_x = \sqrt{\sum_{i=0}^{N_g-1}\sum_{j=0}^{N_g-1} (i-\mu)^2 p_{d,\theta}(i,j)},$$

$$\sigma_y = \sqrt{\sum_{i=0}^{N_g-1}\sum_{j=0}^{N_g-1} (j-\mu)^2 p_{d,\theta}(i,j)}, \tag{5}$$

$$f_6 = \sum_{i=0}^{N_g-1}\sum_{j=0}^{N_g-1} \frac{1}{1+(i-j)^2} p_{d,\theta}(i,j), \tag{6}$$

$$f_7 = \sum_{i=0}^{2(N_g-1)} i.p_{x+y}(i),$$

$$p_{x+y}(k) = \sum_{i=0}^{N_g-1}\sum_{j=0}^{N_g-1} p_{d,\theta}(i,j), \quad k = i+j = \{0,1,2,\ldots,2(N_g-1)\}, \tag{7}$$

$$f_8 = \sum_{i=0}^{2(N_g-1)} (i-f_7)^2 p_{x+y}(i), \tag{8}$$

$$f_9 = -\sum_{i=0}^{2(N_g-1)} p_{x+y}(i) \log p_{x+y}(i), \tag{9}$$

$$f_{10} = \sum_{i=0}^{N_g-1} \left(i - f_{10}^{'}\right)^2 p_{x-y}(i), \tag{10}$$

$$p_{x-y}(k) = \sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} p_{d,\theta}(i,j), \quad k = |i-j| = \{0,1,2,\ldots,(N_g-1)\}, \tag{11}$$

$$f_{11} = -\sum_{i=0}^{N_g-1} p_{(x-y)}(i) \log p_{x-y}(i), \tag{12}$$

$$f_{12} = \frac{HXY - HXY1}{\max(HX, HY)}, \tag{13}$$

$$f_{13} = \left(1 - \exp[-2(HXY2 - HXY)]\right)^{1/2}, \tag{14}$$

$$p_x(i) = \sum_{j=0}^{N_g-1} p_{d,\theta}(i,j),\ p_y(j) = \sum_{i=0}^{N_g-1} p_{d,\theta}(i,j),$$

$$HX = -\sum_{i=0}^{N_g-1} p_x(i) \log(p_x(i)), \quad HY = -\sum_{i=0}^{N_g-1} p_y(i) \log(p_y(i)),$$

$$HXY = -\sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} p_{d,\theta}(i,j) \log\left(p_{d,\theta}(i,j)\right),$$

$$HXY1 = -\sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} p_{d,\theta}(i,j) \log\left(p_x(i) p_y(j)\right),$$

$$HXY2 = -\sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} p_x(i) p_y(j) \log\left(p_x(i) p_y(j)\right).$$

2.2 Cell broadband engine

The Cell Broadband Engine (BE) is a heterogeneous multi-core architecture with nine processors specialized into two types: one PowerPC Processor Element (PPE) and eight Synergistic Processor Element (SPE) [12]. The first type of processor element, PPE, is a 64-bit PowerPC architecture core. It is fully compliant with the 64-bit PowerPC architecture and can run 32-bit and 64-bit operating systems and

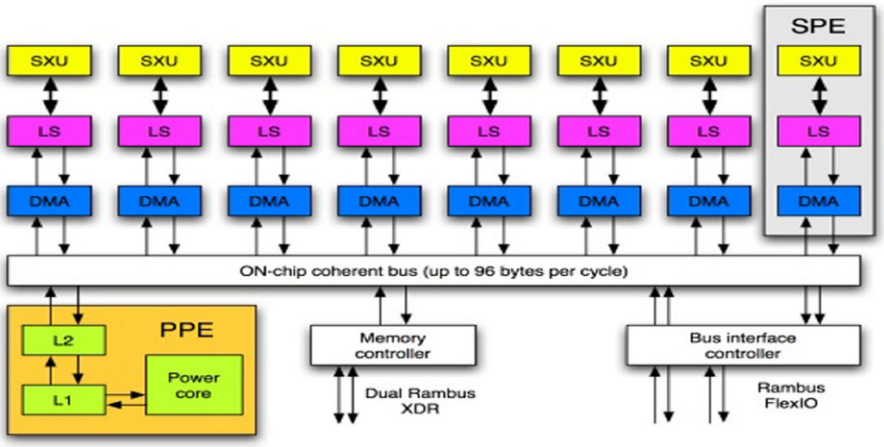

Fig. 4 Block diagram of the Cell BE architecture

applications. The second type of processor element, the SPE, is optimized for running computation-intensive SIMD applications. The SPEs are independent processors, each running its own individual application programs. Each SPE has full access to coherent shared memory, including the memory-mapped I/O space [6]. Figure 4 shows a block diagram of the Cell BE. In this diagram, SXU is Synergistic Execution Unit, Load Store (LS) is 256 KB.

The PPE is a traditional 64-bit PowerPC processor core with a vector multimedia extension (VMX) unit, 32-Kbyte level 1 instruction and data caches and a 512-Kbyte level 2 cache. The PPE is a dual-issue, in-order-execution design, with two-way simultaneous multithreading. The eight SPEs are SIMD processors optimized for data-rich operations allocated to them by the PPE. Each of these identical elements contains a RISC core, 256-KB, software-controlled local store for instructions and data and a large (128-bit, 128-entry) unified register file. The SPEs rely on asynchronous DMA transfers to move data and instructions from main storage and to their local stores. A DMA operation can transfer either a single block area of size up to 16 KB, or a list of 2 to 2048 such blocks. The PPE and SPEs communicate coherently with each other, main storage and I/O through the Element Interconnect Bus (EIB). The EIB is a 4-ring structure (two clockwise and two counterclockwise) for data, and a tree structure for commands. The EIB's internal bandwidth is 96 bytes per cycle with a peak bandwidth of 204.8 GB/s) [13], and it can support more than 100 outstanding DMA memory requests between main storage and the SPEs. The memory interface controller provides a peak bandwidth of 25.6 Gbytes/s to main memory. The I/O controller which provides peak bandwidths of 25 Gbytes/s inbound and 35 Gbytes/s outbound can deliver a sustained bandwidth of 25.6 GB/s.

3 Related work

We discuss the work related to the acceleration of the co-occurrence matrices and Haralick texture features in two parts. The first part presents the FPGAs accelerators and the second displays the Cell and Graphics Processing Units (GPUs).

3.1 FPGAs accelerators

Many researchers have been working on accelerating the process of computation the GLCMs and texture features extraction algorithms on FPGAs platforms [5, 11, 14–17]. Tahir [5] presented an FPGA-based coprocessor for GLCM and texture features and their application in prostate cancer classification. Tahir et al. [14] also presented an FPGA architecture that compute the GLCM of mutispectral images. The computation of the GLCM is performed by one FPGA core, while the computation of the texture features is performed by a second core that is subsequently programmed onto the FPGA. However, the use of this second core results in a time overhead for reprogramming the FPGA, affecting the overall feature extraction performance.

Bariamis et al. [15] presented a hardware implementation to calculate 16 co-occurrence matrices and 4 feature vectors using a single core. The implemented hardware exploited both the symmetry and the sparseness of the matrix They chose $N_g = 32$, the number of gray level for different image sizes from 512×512 to 2048×2048. In order to use floating-point operations to calculate texture features, they used integer arithmetic. Their architecture has 16 co-occurrence matrix computation units. Each input image is divided into different block sizes from 8×8 to 256×256 and is loaded into the corresponding RAM bank. Each pixel is represented using 25-bit, which includes five 5-bit of neighboring pixels at 0°, 45°, 90°, and 135° directions.

Iakovidis et al. [11] presented an FPGA architecture for parallel computation of 16 co-occurrence matrices ($d = 1, 2, 3, 4$) and ($\theta = 0, 45, 90, 135$) that exploits both their symmetry and sparseness. They used 5-bit gray-level representation just like [15]. Bariamis et al. [16] calculated both the GLCMs and features in hardware, while a significant part of the computations relies on software. Their design in [15] allows the computation of GLCM features in hardware, but employed data redundancy in order to achieve high processing throughput. However, the redundancy led to high memory capacity requirements and redundant transfers of data over the PCI bus. They have proposed an FPGA implementation for real-time extraction of GLCM texture features from video frames in [18]. Sieler et al. [17] implemented the GLCMs and Haralick texture features on the FPGA architecture for image size of 128×128. In addition, the whole architecture was implemented into a single FPGA without the use of any external memory and host machine.

The drawbacks of some of those implementations on the FPGAs are the following. First, some implementations such as [5] required large external memory banks, while some processing is performed by a host machine. Second, other implementations such as [11, 15] included symmetry and sparseness matrices, which is not a general implementation to support all kinds of images. Finally, these implementations calculate GLCMs without implementation considerations for improving the performance

of the Haralick texture features. Additionally, some of them [17] used small image sizes such as 128×128.

3.2 GPU and cell accelerators

Gipp et al. [19] accelerated the computation of the GLCMs and Haralick texture features using Graphics Processing Units (GPUs) for biological applications. In biological applications, features are extracted from microscopy images of cells. The computation of features takes several weeks because of processing a larger number of images. They claimed that the development time for GPUs is much less than for FPGAs platforms. In addition, the computing power of GPUs grows much faster than the FPGAs.

Sugano and Miyamoto [20] have implemented good feature extraction for tracking on the Cell processor. They used 6 SPEs to process an image size of 640×480. Their performance improvement was 5 times faster than the computation on Core Duo CPU running at 3.00 GHz. Liu [21] accelerated the color auto-correlogram feature on the Cell processor. The color auto-correlogram is a feature that can be used for image descriptor for both comparison and retrieval. The speedup of 12.16 yielded for that feature over a cache-based microprocessor. Lu et al. [22] indicated that feature extraction is very time consuming process in the content-based image retrieval systems. In addition, for evaluation of an image retrieval system, the response time is a critical factor. Based on these they have discussed using parallel computing techniques to improve the performance of the content-based image retrieval systems.

In this paper, we have chosen the Cell platform for our parallel implementation and performance improvement. The reasons behind this selection compared to the FPGAs and GPUs are as follows. First, a common Cell processor is not expensive and it is available. There are many experiments in the implementation of different applications on the Cell platform. Second, because of available tools and experiments on the Cell architecture, the implementation and mapping phase times are not significant. Finally, the Cell is a parallel programmable processor and this makes the implementation of different algorithms of texture features extraction and their updated versions easier than on non programmable processors.

4 Parallel implementation

In this section, first we discuss different strategies for parallel implementations on the Cell architecture. Second, parallelization of the GLCM and feature extraction algorithms on the Cell platform are presented.

4.1 Different parallelization strategies on the cell architecture

In parallel implementation on the Cell BE, we have to make use of the two following point to maximize the performance of the Cell. First, operate multiple SPEs in parallel to maximize operations that can be executed in a certain time unit. Second, perform SIMD parallelization on each SPE to maximize operations that can be executed for

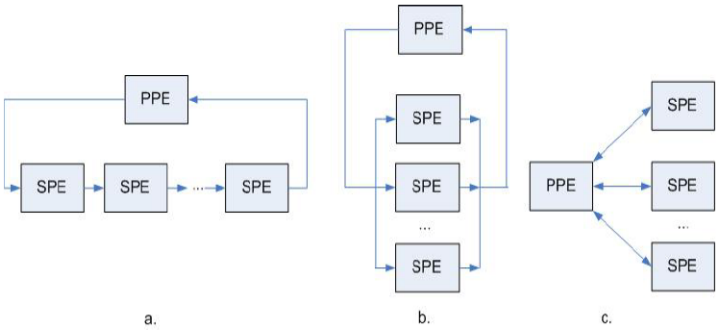

Fig. 5 Parallel programming model. **a** Multistage pipeline model. **b** Parallel stage model. **c** Service model

each instruction. For parallelization the work over SPEs, there are three ways in which the SPEs can be used: the multistage pipeline model, the parallel stage model, and the service model [6]. Figure 5 depicts these parallel models. Multistage pipeline model is suitable when tasks can be divided into sequential stages. Following this model, the stream of data is sent into the first SPE, which performs the first stage of the processing. The first SPE then passes the data to the next SPE for the next stage of processing. At the same time, the next part of data is fed into first SPE to process. After the last SPE has done the final stage of processing on its data, that data is returned to the PPE. As with any pipeline architecture, parallel processing occurs, with various portions of data in different stages of processing. The disadvantage of this model is that the data must be moved for each stage of the pipeline, which slows down the execution of the whole process.

Parallel stage model can be applied when the data is partitionable or tasks can be processed concurrently. Each SPE processes different parts of data or different tasks in parallel. This is the basic and most popular parallel model. This parallel stage model has also been used in [23] where an image is divided into a number of blocks and a transputer is allocated to each block. In the service model, the PPE assigns different services to different SPEs, and the PPE's main process calls upon the appropriate SPE when a particular service is needed. This model is suitable for service-driven applications.

In SIMD processing, data are presented in vector type with the length of 128 bits. Each SIMD instruction processes whole vector simultaneously. However, some applications could not benefit from SIMD processing. This is because not only of data dependency issues but also non-aligned and irregular data access patterns problems. The gray level co-occurrence matrix belongs to this category which cannot benefit directly from SIMD instructions. In the following section, the vectorization of this application is described in details.

4.2 Parallel implementation of Gray Level Co-occurrence Matrix

In the non-parallel implementation as mentioned in Fig. 3, to calculate co-occurrence matrix, each pixel and its neighbors are read sequentially. To parallelize this process, among three discussed models in Sect. 4.1; the multistage pipeline model, the parallel

stages model, and the service model; the parallel stage model is more suitable than the others. This is because we can easily partition the image by the number of working SPEs. Each divided part of the image is processed independently in an SPE. The partitioning technique and its implementation using scalar and SIMD instructions are discussed in the following section.

4.2.1 Parallelization of the co-occurrence matrix using partitioning technique

Computation of the GLCM for an image is vectorized in two steps. The first step is the partitioning the image by the number of working SPEs. Each SPE processes the same operations to compute a sub-cooccurrence matrix. In other words, input image is partitioned as many times as the number of available SPEs and each partitioned part is processed independently in an SPE. The second step is summing up the all computed sub-cooccurrence matrices to provide the final co-occurrence matrix. These steps are depicted in Figs. 6 and 7 using 4 SPEs. We process a sub-part of the

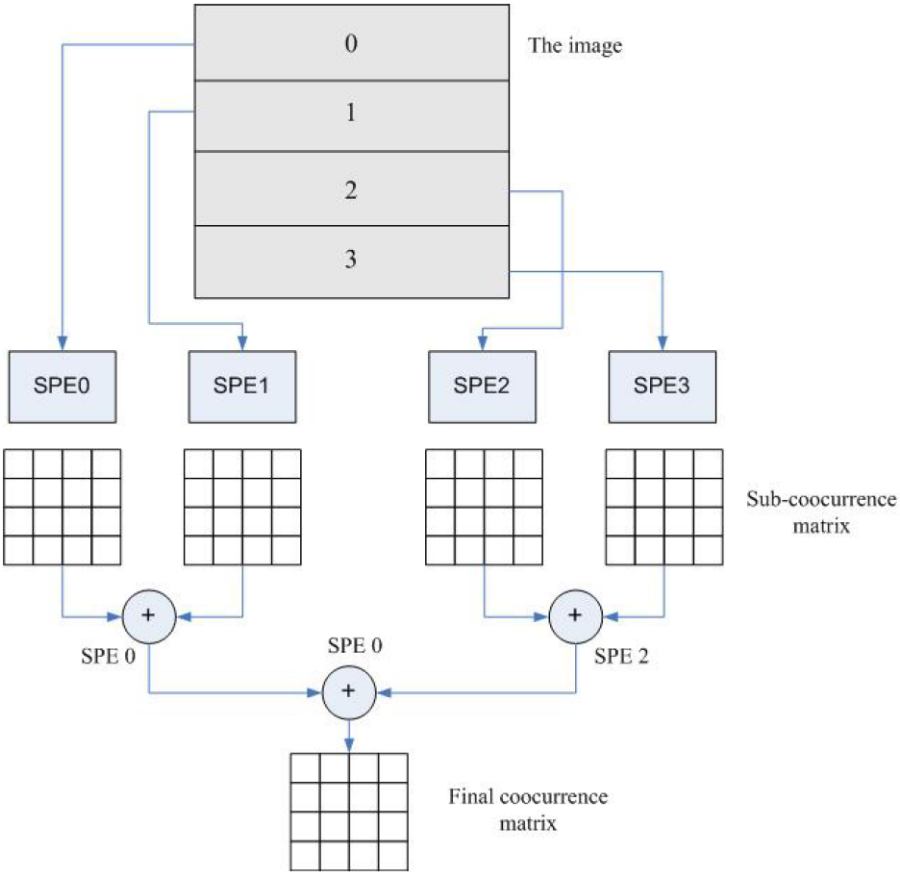

Fig. 6 Method 1: parallel implementation of co-occurrence matrix by splitting an image with 4 SPEs and computing the final co-occurrence matrix using the tree structure

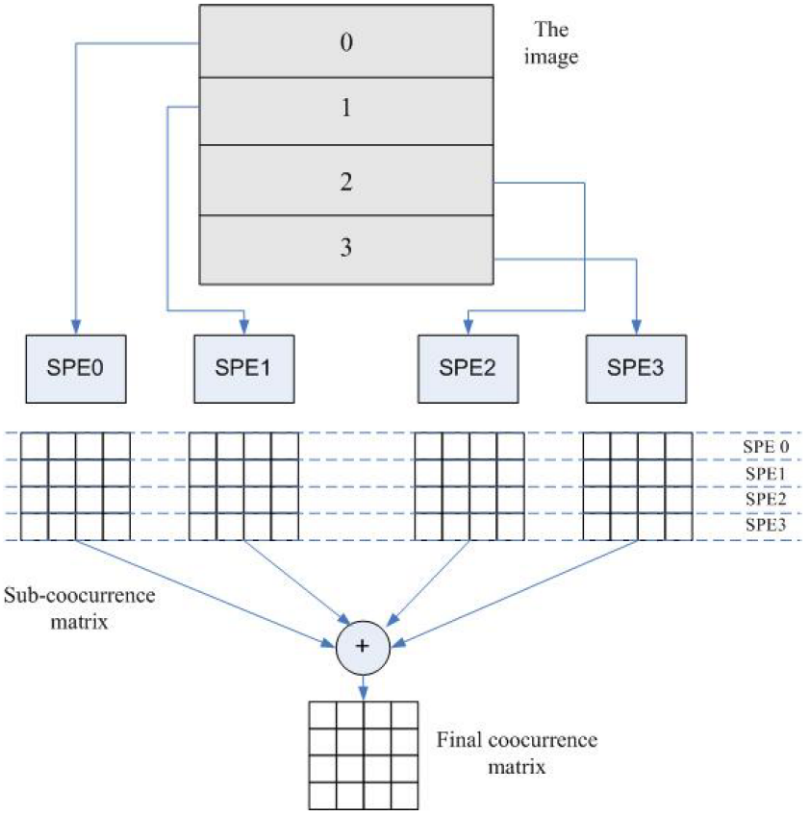

Fig. 7 Method 2: parallel implementation of co-occurrence matrix by splitting an image with 4 SPEs and computing the final co-occurrence matrix using all working SPEs

image in each SPE, computing a sub-cooccurrence matrix. After that, these matrices are summarized to produce final co-occurrence matrix. At this stage, there are two techniques to calculate the sum of all sub-cooccurrence matrices.

The first method is tree structure as it is depicted in Fig. 6. In SPE0, the sub-matrix of SPE0 is added with the sub-matrix of SPE1. In SPE2, the sub-matrix of SPE2 is added with the sub-matrix of SPE3. After that, in SPE0, two result-matrices are added to form the final matrix. However this technique utilizes SPEs inefficiently, because in first step, only SPE0 and SPE2 work, and in second step, only SPE0 works. The second technique is described in Fig. 7. The computed sub-matrix in each SPE is partitioned into N parts (N is the number of working SPEs). Each of these parts is added with corresponding parts from all other sub-matrices in an SPE. Following this method, all SPEs work equally and the final co-occurrence matrix is calculated in one step.

In next section, we discuss implementation of the partitioning technique using scalar and SIMD instructions.

4.2.2 *Implementation of the co-occurrence matrix in an SPE using SIMD instructions*

As stated before, not every parallelizable application benefits significantly from SIMD processing because of data dependency issues, non-aligned, and irregular data access problems. Unfortunately, the co-occurrence matrix belongs to this category. To build a co-occurrence matrix, based on neighboring pair of pixels, elements of the matrix are accessed irregularly to update their values. Because of this irregular access, co-occurrence matrix is not able to parallelize with SIMD, therefore it have to be processed by scalar operations.

However, the SPE is SIMD-only processor, offers no registers dedicated to scalar data and thus uses the same register for both vector and scalar data. The SPE also provides no load, store and arithmetic instructions designed specifically for scalar data. Scalar operations on this processor are performed by using SIMD instructions. In 128-byte register, scalar types are stored in the slot as depicted in Fig. 8. Scalar data only uses a predetermined part of the register called the preferred slot. `char` data type is located to Byte 3, `short` data type to Bytes 2 and 3, `int` data type and `float` type to Bytes 0 to 3, and `long long` type and double type to Bytes 0 to 7. SPE only manipulates scalar data that is allocated in preferred slot. Therefore, before processing scalar data, it is required that they must be aligned in preferred slot by shuffling through `rotqby` and `shufb` instructions [24]. Figure 9 describes SPE scalar operations.

Because data must be loaded and stored in 16-byte units, input data—16 bytes inclusive of scalar data—is loaded to the register and shifted appropriately to the preferred scalar element. When storing the result in the memory, the 16-bytes data inclusive of the location to store is loaded, a part of the 16-bytes data is replaced with the calculated result, and then the whole 16-byte data is stored. This way of SPE in handling scalar operations causes considerable overhead. In [24], Large-Data-Type (LDT) methodology, an alternative approach in scalar processing, is described. In LDT methodology, scalar data is defined as a vector, which means one element of the vector is scalar data, others are zero. Then this vector is manipulated using SIMD instructions. LDT methodology skips rotate and shuffle operations, however, increases the data size, as the scalars are now four times larger. If we still use the scalar computation in calculating sub-cooccurrence matrix, we can use SIMD instructions in

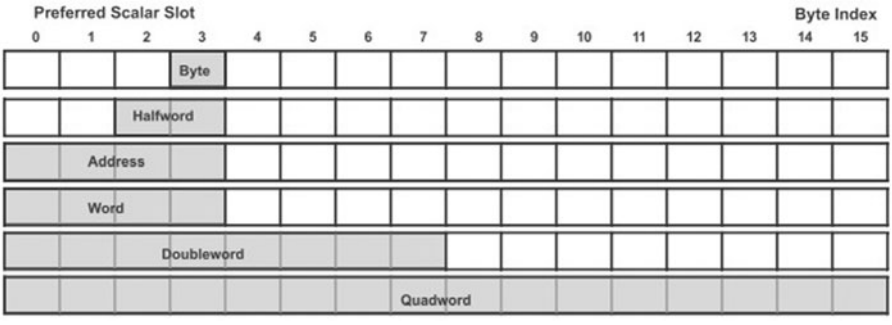

Fig. 8 Register layout of data types and preferred scalar slot

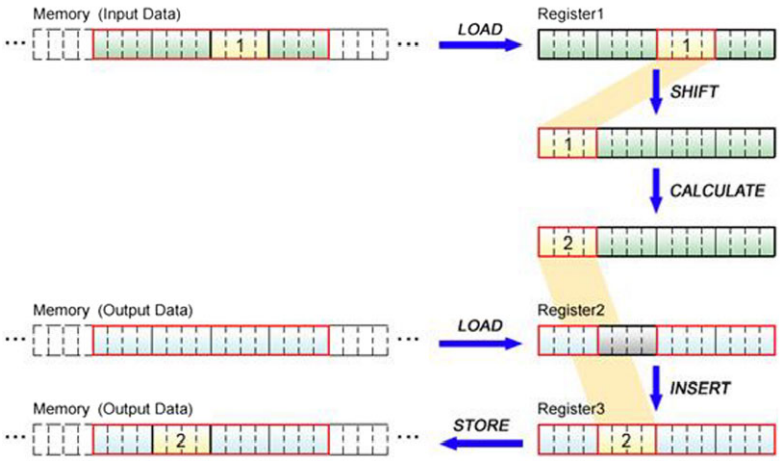

Fig. 9 SPE scalar operations

summing all these matrices. Elements of sub-matrices are aligned and accessed consecutively, which is perfect for SIMD operations.

4.3 Parallel implementations of texture feature extractions algorithms

As discussed in Sect. 2.1.1, Haralick exploited different texture features using the GLCMs in [4]. There are different approaches to parallelize the texture feature extraction stage. First technique is computing one or a group of texture features in each processor. However, this approach has a bottleneck since the computing intensiveness of features is different from each others, which makes the unbalanced load between SPEs. Second approach which can give better balance is partitioning the computed co-occurrence matrix into different separate parts. Each part is processed in an SPE to compute partial texture features. The sum of these partial texture features in all SPEs is the final value. Figure 10 depicts this strategy with 4 SPEs. On the one hand, elements of sub-cooccurrence matrices are aligned and consecutively accessed and we can apply SIMD instructions to compute each partial texture feature. On the other hand, there is data dependency between texture features and we cannot compute all partial texture features in parallel. In other words, most of features depend on other features as well as on intermediate results. In order to avoid the repetition of the computations, we have classified the computation of texture features into following three groups.

- Group 1 consists of texture features of energy $f1$, contrast $f2$, entropy $f3$, homogeneity $f6$, and mean μ, P_{x+y}, P_{x-y}, P_x.
- Group 2 consists of variance $f4$, sum average $f7$, sum entropy $f9$, different entropy $f11$, deviation σ, and different average, HX, HXY, $HXY1$, $HXY2$.
- Group 3 consists of correlation $f5$, sum variance $f8$, different variance $f10$, and information measure of correlation $f12$, and $f13$.

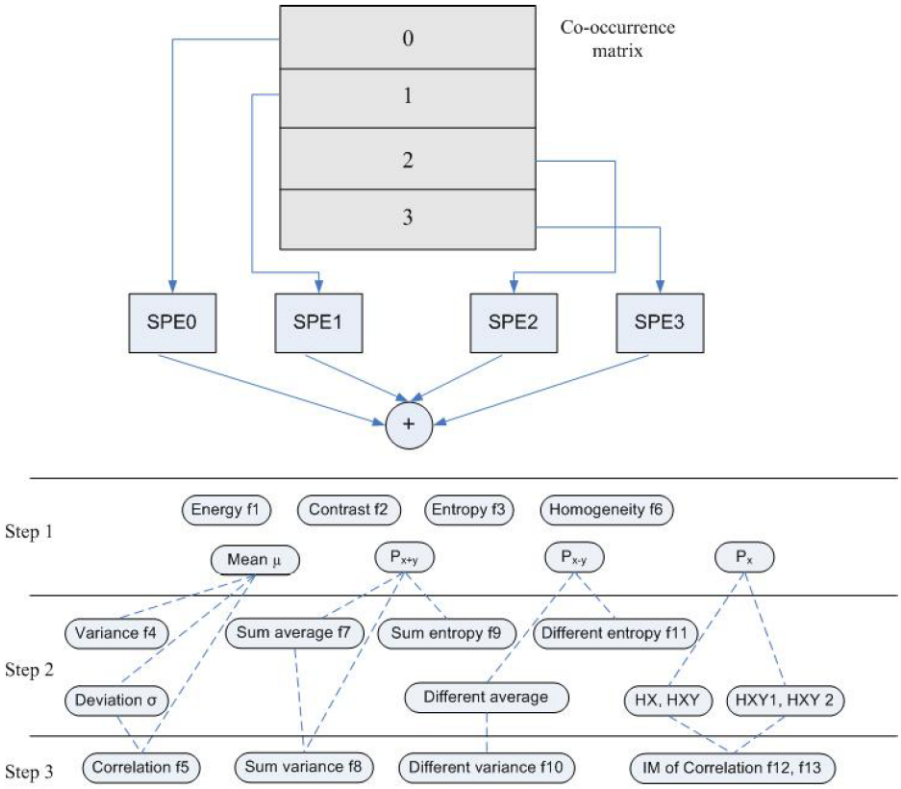

Fig. 10 Partitioning the co-occurrence matrix and compute each partial texture feature in an SPE and the data dependency between texture features

The data dependency between texture features are shown using dash lines in Fig. 10.

As this figure shows the whole feature computation was split in different small computing steps. In the first and second steps, some intermediate results are computed, which can be reused in the other steps.

5 Experimental results and analysis

In this section, first we discuss the experimental environment. Second, we present the obtained performance improvements of our implementations onto the Cell architecture. Finally, the performance and efficiency of the Cell processor is compared with modern state-of-the-art x86-based processors.

5.1 Experimental environment

We have used a development tool to implement the above mentioned algorithms on the Cell architecture. The development tool is Cell BE SDK 3.1 including GCC com-

piler version 4.1.2 for the PPU and the SPU. The optimization compiling flag for the code running on the PPE and SPE is -O3.

The benchmarking environment is Cell Blade located in the Barcelona Supercomputing Center (BSC). The Cell Blade consists of two physical Cell processors linked via the FLEXIO bus, which has a peak throughput of 37.6 GB/s. The amount of external memory is 1 GB and is fully usable. The rated memory bandwidth is 25.6 GB/s. The operating system is Fedora Core 7 with kernel version 2.6.22. With Cell Blade, 16 SPEs are fully usable.

We have used different image sizes such as 128×128, 256×256, 512×512, and 1024×1024 with different gray levels, 16, 32, 64, and 128. In order to obtain the performance, we have measured the execution time for each algorithm. Time is measured using the SPU decrementer. The SPU decrementer 32-bit register ticks at a constant rate which is defined in the Time Base (with the Cell Blade of this experiment, Time Base value is 14.318 MHz). The value of the SPU decrementer is read at the beginning and at the end of the measured interval. In order to increase the accuracy for each experiment, the smallest time was selected from a number of, for example, 1,000 iterations.

In order to evaluate the impact of the increasing number of processors on the performance, we have used different implementations on the PPE, 1 SPE, 2 SPEs, 4 SPEs, 8 SPEs, and 16 SPEs. In addition, the performance improvements of parallel implementations have been compared with non-parallel implementations. For this purpose, we have selected two modern x86 processors, Intel Pentium 4 and Core2 Duo. The specifications of these processors are depicted in Table 1. In order to compare different platforms, the Cell processor and x86 processors, speedups criteria were chosen. The speedups were measured by the ratio of the total execution time of each algorithm for the Core2 Duo processor implementation to the Pentium 4, Cell 8 SPEs, and Cell 16 SPEs implementations.

5.2 Experimental results for co-occurrence matrix

In this section, we present two experimental results for the GLCMs. In the first implementation that is called Scalar, we have implemented the GLCMs on different processors (Cell architecture) using scalar operations, while we have not used any SIMD instructions. In the second implementation, in addition to using different processors we have employed SIMD instructions using the discussed LDT concept.

Table 1 Specification of $\times 86$ processors, which have been used for non-parallel implementations

Specifications	Pentium 4	Core2 Duo
Number of cores	1	2
Processor frequency	3.6 GHz	2.33 GHz
L1 Cache	32 KB	64 KB
L2 Cache	2 MB	4 MB
RAM	2 GB	8 GB
Operating system	Suse 10	Red Hat 4.1.2
32/64-bit	32-bit	64-bit
Kernel version	2.6.27	2.6.18

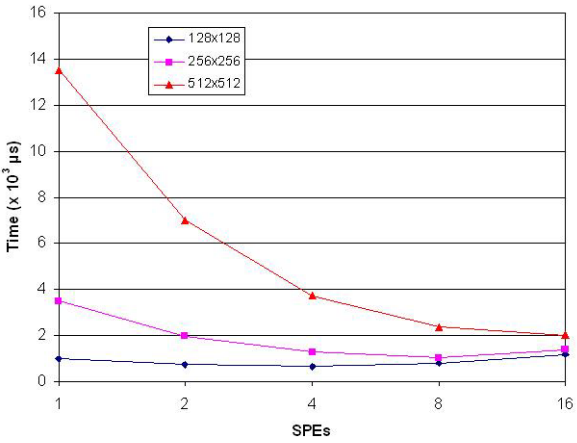

Fig. 11 The total execution time of implementation GLCMs on the Cell processor using scalar operations for different image sizes with 128 gray levels

5.2.1 Implementation of GLCMs on the cell processor using scalar operations

Figure 11 depicts the total execution time of the GLCMs for different image sizes with 128 gray levels on different SPEs from 1 to 16. The results depict that for large image sizes, with increasing the number of processing elements, the total execution time has been significantly reduced. For example, the total execution time for an image size of 512×512 on 1 SPE and 16 SPEs is 13520 and 2024 µseconds, respectively. On the other hand, for small image sizes such as 128×128, with increasing the number of processors, the execution time is not only significantly reduced but also it is increased for 16 SPEs from 986 to 1010 µseconds. The reason for this behavior is that there are two steps to compute the GLCMs, calculation the sub-matrices in SPEs and summing all computed sub-matrices to each other, as already discussed in previous sections.

On the one hand, the execution time to compute the sub-matrices depends on the size of the image and decreases with increasing the number of SPEs. On the other hand, execution time of summing is independent from image sizes. It increases with increasing the number of SPEs due to the synchronization time between different SPEs. For small image sizes, when the number of SPEs is increased, the execution time of summing all sub-cooccurrence matrices becomes dominant compared to the execution time of computing sub-cooccurrence matrices. The results presented in Fig. 12 validate this claim. As this figure shows for small image sizes such as 128×128, the execution time of computation of sub-matrices for smaller usage of SPEs is significantly more than the execution time of summing up all computed sub-matrices. On the other hand, for larger usage of SPEs, the execution time of computation of sub-matrices is significantly less than the execution of summing up all computed sub-matrices. While this behavior is not true for large image sizes such as 512×512.

5.2.2 Implementation of GLCMs on the cell processor using large data type concept

As explained in previous sections, the computation of GLCMs has two steps, calculation the sub-cooccurrence matrices and summing all computed matrices to each

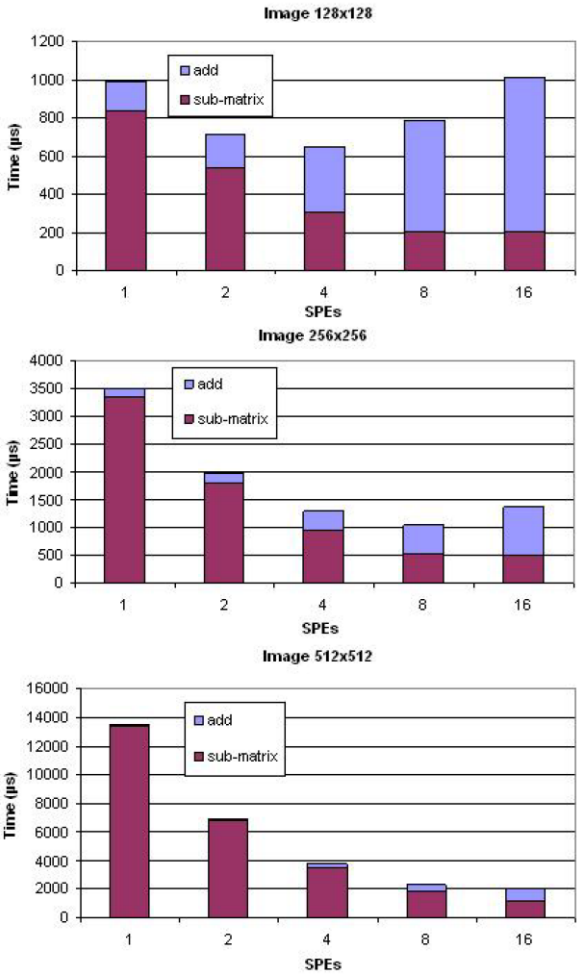

Fig. 12 The execution time of computing sub-cooccurrence matrices using different SPEs and summing all computed sub-cooccurrence matrices to each other for different image sizes

other. The former has been implemented in two different approaches, scalar operations and SIMD instruction using the LDT concept. The latter; the summing all computed matrices; has been implemented using the SIMD instructions due to its regular and consecutively access patterns.

Figure 13 compares these two approaches for an image size of 512×512. The speedup of the LDT implementation over scalar implementation ranges from 1.2 to 1.6. This is because computing sub-cooccurrence matrices in the LDT approach is faster than the corresponding stage in the scalar approach due to use of SIMD implementations. This means that the LDT technique reduces the computational time of the GLCMs more than the using scalar implementation. With increasing the number of SPEs the speedup is reduced. The reason for this is that with increasing the number of SPEs the computational time of summing computed sub-cooccurrence is dominant compared to the computing sub-cooccurrence matrices.

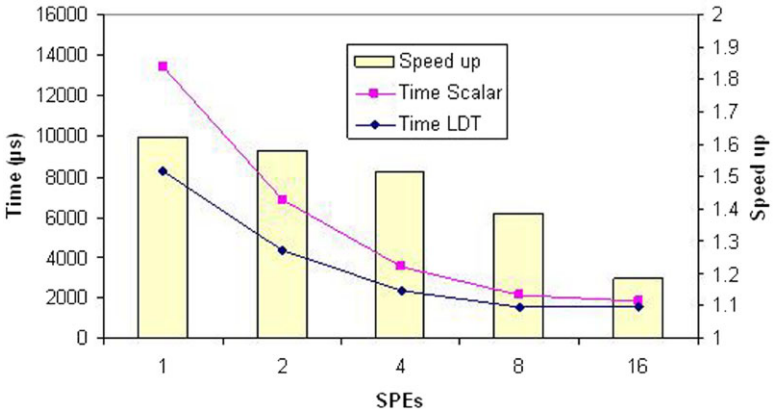

Fig. 13 Comparison of two different implementations, scalar and LDT, to compute GLCMs for an image size of 512 × 512

Table 2 The total execution time in µseconds to compute Haralick texture features for different image sizes with 128 gray levels

Image sizes	PPE	1 SPE	2 SPE	4 SPE	8 SPE	16 SPE
128 × 128	13819	952	555	332	**227**	**250**
256 × 256	17836	951	555	331	**234**	**250**
512 × 512	20302	951	555	330	**223**	**242**
1024 × 1024	21330	952	555	331	**227**	**253**

5.3 Experimental results for texture features

Table 2 depicts the total execution time of computation Haralick texture features for different image sizes with 128 gray levels using PPE, 1, 2, 4, 8, and 16 SPEs. As this table shows, with increasing the number of SPEs from 1 to 8, the execution time is reduced for each image size, while increasing the number of SPEs from 8 to 16, the execution time is increased. For example, the execution time to compute the Haralick texture features for image size of 512 × 512 using 1 SPE and 8 SPEs is 951 and 223 µseconds, while for 16 SPEs it is 242 µseconds. This is because with increasing the number of SPEs from 8 to 16, the synchronization and communication times between different SPEs are dominant compared to the computational time.

In addition, Table 2 shows that the total execution time to compute Haralick texture features for different image sizes on the same number of SPEs is almost the same. For instance, the computational time to process different image sizes using 4 SPEs is almost 330 µseconds. This is because the computation of texture features is independent from image sizes. It depends on the number of gray levels. Table 3 depicts some results for validation. This Table depicts the execution time to compute texture features for an image size of 512 × 512 with different gray levels from 16 to 128 using different numbers of SPEs. This table shows with decreasing the number of gray levels from 128 to 16 the execution time is reduced. As already explained, the

Table 3 Execution time in μseconds to calculate Haralick texture features for an image size of 512 × 512 with different gray levels 16, 32, 64, and 128 using different numbers of SPEs

Gray levels	PPE	1 SPE	2 SPE	4 SPE	8 SPE	16 SPE
128	20302	**951**	555	330	223	242
64	5378	**242**	193	137	123	211
32	1510	**122**	98	84	88	175
16	415	**82**	76	72	88	174

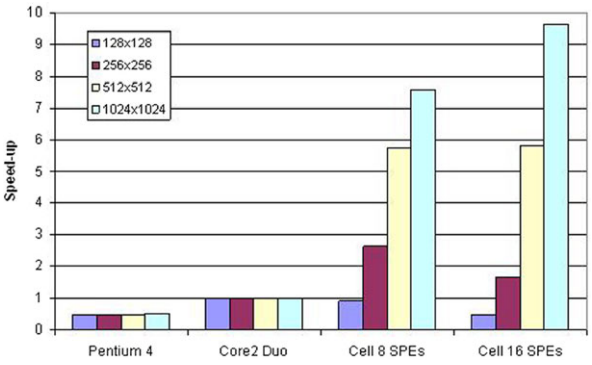

Fig. 14 Speedups of the Pentium 4, Cell 8 SPEs, and Cell 16 SPEs over the Core2 Duo processor for computation of GLCMs using different image sizes with 64 gray levels

reason why the computational time of 16 SPEs is more than the computational time of 8 SPEs is because of increasing the synchronization and communication times between different SPEs.

Tables 2 and 3 depict that execution time of PPE is more than the execution time of 1 SPE. The main reason is that scalar operations have been used to implement algorithms on PPE, while SIMD operations have been employed on SPEs. In other words, computation time on the PPE indicates the performance without optimization for the Cell processor.

5.4 Comparison between parallel and non-parallel implementations

In the previous sections, the performance of parallel implementations of the GLCMs and texture features have been presented. In this section, the performance improvement on the Cell architecture is compared to the performance improvement on the several x86 processors, a Pentium 4 and Core 2 Duo. The specifications of the these platforms have already been presented in Sect. 5.1.

Figures 14 and 15 depict the speedups of the Pentium 4, Cell 8 SPEs, and Cell 16 SPEs over the Core2 Duo processor for computation of GLCMs and texture features using different image sizes with 64 gray levels, respectively. The speedups were measured by the ratio of the total execution time of each algorithm on the Core2 Duo processor implementation to the Pentium 4, Cell 8 SPEs, and Cell 16 SPEs implementations.

It can be seen from Fig. 14 that for calculation of the GLCMs with small image sizes such as 128 × 128, Core2 Duo processor is the most efficient. It is hard for the

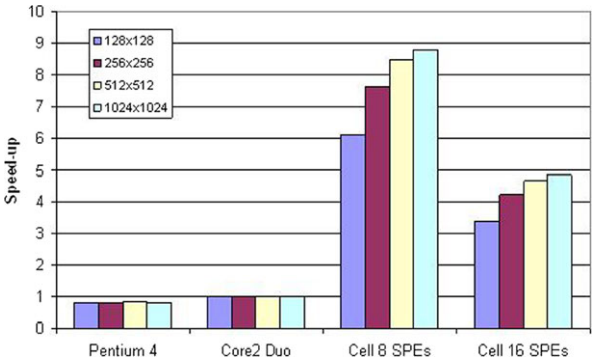

Fig. 15 Speedups of the Pentium 4, Cell 8 SPEs, and Cell 16 SPEs over the Core2 Duo processor for computation of texture features using different image sizes with 64 gray levels

Cell processor to demonstrate its superiority because of synchronization and communication times needed between different SPEs. When the size of workloads increases, the Cell processor with 8 and 16 SPEs yields more performance improvement than other processors. For example, for image size of 1024×1024, the speedup of using 16 SPEs is approximately $10\times$.

As already mentioned, the texture feature extraction algorithms are more computational intensive than the co-occurrence matrices. On the one hand, the computation of texture features is independent from image sizes. On the other hand, it depends on the number of gray levels. As we can see in Fig. 15, texture feature extraction algorithms benefit from the advantages of the Cell processor for different image sizes. For instance, the speedup of 8 SPEs is approximately $9\times$ for image size of 1024×1024.

With increasing the number of gray levels more speedup is obtained. For instance, our experimental results shows that with doubling the gray levels to 128, the speedups is almost doubled.

6 Conclusions

The co-occurrence matrices and Haralick texture features extraction algorithms are computationally intensive. Our experimental results showed that applying some software optimization techniques such as using link list, hash table, and loop unrolling which have been proposed by researchers in the literature, could not provide sufficient performance. In order to exploit available task- and data-level parallelism in those algorithms, we have implemented the gray-level co-occurrence matrices and Haralick texture features on a programmable multi-core, Cell processor. The Cell is a single-chip multi-processor with nine processors: one PowerPC Processor Element (PPE) and eight Synergistic Processor Element (SPE) which are optimized for compute-intensive applications. In order to compute the GLCM, in the first step the input image was partitioned into the number of working SPEs. All divided parts were simultaneously processed by different SPEs. Each SPE computes a sub-cooccurrence matrix. In the second step, all computed sub-cooccurrence matrices have been summed up to compute the final co-occurrence matrix. We could not use the SIMD instructions for the implementation of first stage directly, due to data dependency issues, non-aligned, and irregular data access problems. We have applied Large Data Type (LDT)

concept, which defines each element of co-occurrence matrix as a vector type, to use SIMD instructions. This technique improves performance compared to the scalar implementation from 1.2 to 1.6 depending on number of working SPEs. The speedup of our parallel implementation of using 16 SPEs over x86 Core Duo processor is almost $10\times$ for an image size of 1024×1024.

In addition, for computation of Haralick's texture features, we have exploited both task- and data-level parallelism. The computed GLCM was partitioned into different parts. Each SPE computed the partial texture features. Then in the second stage, all computed partial results have been summed up. In each SPE, the features were calculated using SIMD instructions. Our experimental results showed that the speedup of the parallel implementation over the non-parallel implementation is almost $9\times$ for an image size of 1024×1024.

We observed that when the size of workloads, image size, number of gray level, is small the communication and synchronization times between different cores become dominant compared to the computational time. On the other hand, when the size of workloads is large, the computational time becomes dominant in comparison with the synchronization time. This means that parallel implementation on the Cell architecture is suitable for computational intensive applications such as medical image processing and remote sensing.

References

1. Tuceryan M, Jain AK (1998) Texture analysis. In: Chen CH, Pau LF, Wang PSP (eds) The handbook of pattern recognition and computer vision, 2nd edn. World Scientific, New York, pp 207–248
2. Nguyen NG, Poulsen RS, Louis C (1983) Some new color features and their application to cervical cell classification. Pattern Recognit 16(4):401–411
3. Schroder M, Dimai A (1998) Texture information in remote sensing images: a case study. In: Workshop on texture analysis
4. Haralick RM, Shanmugam K, Denstien I (1973) Textural features for image classification. IEEE Trans Syst Man Cybern 3(6):610–621
5. Tahir MA, Bouridane A, Kurugollu F (2005) An FPGA based coprocessor for GLCM and Haralick texture features and their application in prostate cancer classification. Analog Integr Circuits Signal Process 43(2):205–215
6. IBM (2008) Software Development Kit for Multicore Acceleration version 3.1: Programming Tutorial
7. Ojala T, Pietikaine M (2010) Texture classification. Master's thesis, Machine Vision and Media Processing Unit, University of Oulu, Finland
8. Materka A, Strzelecki M (1998) Texture analysis methods—a review. Technical report, Institute of Electronics, Technical University of Lodz
9. Sutton R, Hall EL (1972) Texture measures for automatic classification of pulmonary disease. IEEE Trans Comput C-21:667–676
10. Hall-Beyer M (2011) The GLCM Tutorial Home Page. http://www.fp.ucalgary.ca/mhallbey/tutorial.htm
11. Iakovidis DK, Maroulis DE, Bariamisa DG (2007) FPGA architecture for fast parallel computation of co-occurrence matrices. Microprocess Microsyst 31(2):160–165
12. IBM (2007) Synergistic processor unit instruction set architecture, January 2007, version 1.2
13. Chen T, Raghavan R, Dale JN, Iwata E (2007) Cell broadband engine architecture and its first implementation: a performance view. IBM J Res Dev 51(5):559–572
14. Tahir MA, Bouridane A, Kurugollu F (2005) An FPGA based coprocessor for GLCM and Haralick texture features and their application in prostate cancer classification. Analog Integr Circuits Signal Process 43:205–215

15. Bariamis D, Iakovidis DK, Maroulis DE (2006) Dedicated hardware for real-time computation of second-order statistical features for high resolution images. In: Lecture notes in computer science, vol 4179. Springer, Berlin, pp 67–77
16. Bariamis DG, Iakovidis DK, Maroulis DE, Karkanis SA (2004) An FPGA-based architecture for real time image feature extraction. In: Proc 17th int conf on pattern recognition
17. Sieler L, Tanougast C, Bouridane A (2010) A scalable and embedded FPGA architecture for efficient computation of Grey Level Co-occurrence Matrices and Haralick textures features. Microprocess Microsyst 34:14–24
18. Maroulis D, Iakovidis DK, Bariamis D (2008) FPGA-based system for real-time video texture analysis. J Signal Process Syst 53(3):419–433
19. Gipp M, Marcus G, Harder N, Suratanee A, Rohr K, Konig R, Manner R (2009) Haralick's texture features computed by GPUs for biological applications. IAENG Int J Comput Sci 36(1)
20. Sugano H, Miyamoto R (2009) Parallel implementation of good feature extraction for tracking on the cell processor with OpenCV interface. In: Proc 5th IEEE int conf on intelligent information hiding and multimedia signal processing, pp 1326–1329
21. Liu Q (2008) Color spatial feature extraction for image indexing—a case study on the cell B. E. processor. In: Proc congress on image and signal processing, pp 709–713
22. Lu Y, Gao P, Lv R, Su Z (2007) Study of content-based image retrieval using parallel computing technique. In: Proc workshop on high performance computing, pp 186–191
23. Arabnia HR, Oliver MA (1987) A transputer network for the arbitrary rotation of digitised images. Comput J 30(5):425–432
24. Filho AA, Juurlink B (2009) Scalar processing overhead on SIMD-only architectures. In: IEEE international conference on application-specific systems, architectures and processors, pp 183–190