Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2010

MSc THESIS

Optimization of Texture Feature Extraction Algorithm

Tuan Anh Pham

CE-MS-2010-21

Abstract

Texture, the pattern of information or arrangement of the structure found in an image, is an important feature of many image types. In a general sense, texture refers to surface characteristics and appearance of an object given by the size, shape, density, arrangement, proportion of its elementary parts. Due to the signification of texture information, texture feature extraction is a key function in various image processing applications, remote sensing and content-based image retrieval. Texture features can be extracted in several methods, using statistical, structural, model-based and transform information, in which the most common way is using the Gray Level Co-occurrence Matrix (GLCM). GLCM contains the second-order statistical information of spatial relationship of pixels of an image. From GLCM, many useful textural properties can be calculated to expose details about the image content. However, the calculation of GLCM is very computationally intensive and time consuming. In this thesis, the optimizations in the calculation of GLCM and texture features are considered, different approaches to the structure of GLCM are compared. We also proposed parallel computing of GLCM and texture features using Cell Broadband Engine Architecture (Cell Processor). Experimental results show that our parallel approach reduces impressively the execution time for the GLCM texture feature extraction algorithm.

Faculty of Electrical Engineering, Mathematics and Computer Science

Optimization of Texture Feature Extraction Algorithm

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Tuan Anh Pham
born in Hanoi, Vietnam

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Optimization of Texture Feature Extraction Algorithm

by Tuan Anh Pham

Abstract

Texture, the pattern of information or arrangement of the structure found in an image, is an important feature of many image types. In a general sense, texture refers to surface characteristics and appearance of an object given by the size, shape, density, arrangement, proportion of its elementary parts. Due to the signification of texture information, texture feature extraction is a key function in various image processing applications, remote sensing and content-based image retrieval. Texture features can be extracted in several methods, using statistical, structural, model-based and transform information, in which the most common way is using the Gray Level Co-occurrence Matrix (GLCM). GLCM contains the second-order statistical information of spatial relationship of pixels of an image. From GLCM, many useful textural properties can be calculated to expose details about the image content. However, the calculation of GLCM is very computationally intensive and time consuming. In this thesis, the optimizations in the calculation of GLCM and texture features are considered, different approaches to the structure of GLCM are compared. We also proposed parallel computing of GLCM and texture features using Cell Broadband Engine Architecture (Cell Processor). Experimental results show that our parallel approach reduces impressively the execution time for the GLCM texture feature extraction algorithm.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2010-21

Committee Members :

Advisor:	Dr. Koen Bertels, CE, TU Delft
Advisor:	Dr. Asadollah Shahbahrami, CE, TU Delft
Chairperson:	Dr. Koen Bertels, CE, TU Delft
Member:	Dr. Koen Bertels, CE, TU Delft
Member:	Dr. Ir. Zaid Al-Ars , CE, TU Delft
Member:	Dr. Todor Stefanov, Leiden University

Contents

List of Figures **viii**

List of Tables **x**

Acknowledgements **xi**

1 Introduction **1**
- 1.1 Thesis Objective 2
- 1.2 Thesis Organization 2

2 Texture Features **3**
- 2.1 Image Analysis 3
- 2.2 Texture Feature 5
  - 2.2.1 Definition of texture 5
  - 2.2.2 Texture Analysis 5
  - 2.2.3 Application of Texture 7
  - 2.2.4 Texture Feature Extraction Algorithms 7
- 2.3 Grey Level Co-occurrence Matrix and Haralick Texture Features 9
  - 2.3.1 Gray-level Co-occurrence Matrix 9
  - 2.3.2 Haralick Texture Features 10
- 2.4 C++ Implementation for Calculating Co-occurrence Matrix and Texture Features 17
  - 2.4.1 General Structure of the Implementation 17
  - 2.4.2 Data Structure 17
  - 2.4.3 Measure Execution Time 20
- 2.5 Results 23
  - 2.5.1 Comparing Execution Time of Different Sizes 23
  - 2.5.2 Execution Time for Different Gray-levels 24
  - 2.5.3 The Impact of the Distance d on the Execution Time 25
  - 2.5.4 Conclusion 26

3 Software Optimization of Texture Features **27**
- 3.1 Optimization in Co-occurrence Matrix 27
  - 3.1.1 Gray Level Co-occurrence Link List 27
  - 3.1.2 Gray Level Co-occurrence Hybrid Structure 29
- 3.2 Optimization in Texture Features 31
- 3.3 Testing and Result 33
  - 3.3.1 Optimization Co-occurrence with Different Structures 33
  - 3.3.2 Optimization Texture Teatures 34
- 3.4 Conclusions 37

4 Parallel Implementation of Texture Feature Extraction 39
4.1 Overview of the Cell Broadband Engine 39
4.2 The Cell Instruction Set Architecture 41
  4.2.1 The PPE Instruction Set 41
  4.2.2 The SPE Instruction Set 44
4.3 Data Transfer and Communication 44
  4.3.1 Sequential DMA Transfer 44
  4.3.2 List DMA Transfer 47
  4.3.3 Mailbox Communication Mechanisms 47
4.4 Parallel Implementation 48
4.5 Parallelism of Co-occurence Matrix on Multi-SPEs 49
  4.5.1 Parallel Strategy 50
  4.5.2 Implementation on the Cell BE 51
4.6 Implementation of the Co-occurrence Matrix in the SPE using SIMD Instructions 56
4.7 Parallel Implementations of Texture Feature Extraction 59
  4.7.1 Parallel Strategy 59
  4.7.2 Implementation on the Cell BE 59
4.8 Implementation of Co-occurrence Matrix and Texture Feature Extraction with Non-zero Elements 63
4.9 Experimental Results and Analysis 64
  4.9.1 Experiment Environment 64
  4.9.2 Implementation of Co-occurrence Matrix 65
  4.9.3 Implementation of Texture Features 68
  4.9.4 Implementation of Co-occurrence Matrix and Texture Features with Non-Zero Elements 69
  4.9.5 Comparison 70
  4.9.6 Conclusion 71

5 Conclusion and Future Work 75
5.1 Conclusions 75
5.2 Future Work 76

Bibliography 79

List of Figures

2.1 Examples of texture features 6
2.2 Different steps in the texture analysis process 6
2.3 GLCM of a 4×4 image for distance $d = 1$ and direction θ=0 9
2.4 Eight directions of adjacency 10
2.5 4x4 gray-scale image(a); Co-occurrence matrix(b); Normalized Co-occurrence matrix (c) 13
2.6 The structure of the program 18
2.7 The image Wood 23
2.8 Graph of execution time of the image in different sizes with gray-level=256, d=1 24
2.9 Graph of execution time with different gray-levels, image size =2048 × 2048, d=1 25

3.1 4x4 gray-scale image(a); Co-occurrence matrix(b); Normalized Co-occurrence matrix (c) 27
3.2 GLCLL structure for the image in Figure 3.1 28
3.3 Array of linked-list structure 29
3.4 Hash table - linked list structure 30
3.5 Hash table - array structure 31
3.6 The tested image 'Stone' 33
3.7 Speed up of 3 structures with different images sizes (in comparison with array structure) 35
3.8 Speed up in calculating texture features of 4 structures after optimization in texture features (in comparison with normal implementation) 36
3.9 Speed up in calculating texture features of 3 structures in comparison with array structure after optimization 37
3.10 Speed up with different non-zero elements, hash table - linked list structure in comparison with array structure, image size 128x128, gray level =256 . 38

4.1 Block diagram of the Cell Broadband Engine architecture. 40
4.2 Concurrent execution of integer, floating-point, and vector units 42
4.3 Array-summing using SIMD instructions 43
4.4 Latency of DMA get transfer (LS-Main memory and LS-LS) with different DMA message sizes 45
4.5 Bandwidth of DMA get transfer (LS-Main memory and LS-LS) with different DMA message sizes 45
4.6 Bandwidth of LS- main memory DMA get transfer with different SPEs and DMA message sizes 46
4.7 Bandwidth of LS-LS DMA get transfer with different SPEs and DMA message sizes 46
4.8 Bandwidth of list and sequential LS- main memory DMA get transfer with different SPEs 47

4.9 Bandwidth of list and sequential LS- LS memory DMA get transfer with different SPEs 48
4.10 Parallel programming model: a.Multistage pipeline model b.Parallel stages model c.Services model 49
4.11 Method 1: Parallel implementation of co-occurrence matrix by splitting an image with 4 SPEs 51
4.12 Method 2: Parallel implementation of co-occurrence matrix by splitting an image with 4 SPEs 52
4.13 Parallel implementation of co-occurrence matrix in eight directions with 8 SPEs 52
4.14 Register layout of data types and preferred scalar slot 57
4.15 SPE scalar operations 57
4.16 parallel model in computing texture features 60
4.17 Implementation of Co-occurrence Matrix with Non-zero Elements 64
4.18 Scalar implementation of building co-concurrence matrix for three image sizes: 128 × 128, 256 × 256, 512 × 512 and the number of gray level is 128 66
4.19 Components of building time of co-concurrence matrix for three image sizes: 128 × 128, 256 × 256, 512 × 512 and the number of gray level is 128 67
4.20 Comparison of the implementation of the co-occurrence matrix using scalar and LDT techniques, image size 512 × 512, gray level Ng =64 . . . 68
4.21 Execution time of the calculation 13 texture features for different gray levels when the image size is 512 × 512 70
4.22 Execution time of normal array and non-zero element approach, the image size is 128 × 128, the gray level is 64 72
4.23 Performance speed-up (in comparison with Core2 Duo) of texture feature extraction algorithm (including the calculation of co-occurrence matrix and texture features) when gray level is 64 73

List of Tables

2.1 texture features of the 4 × 4 image 17
2.2 Specification of running environment: Intel Core 2 Duo E6550 23
2.3 Execution time of the image Stone in different sizes 24
2.4 Execution time with different gray-levels, image size 2048 × 2048, d=1 . . 25
2.5 execution time with different distances d, image size 1024 × 1024, gray level 256 26

3.1 Building time of co-occurrence matrix of 5 structures, gray level Ng =256 ($\times 10^6$ cycles) 33
3.2 Calculating time of texture features of 5 structures, gray level Ng =256 ($\times 10^6$ cycles) 34
3.3 Total execution time of 5 structures, gray level Ng =256 ($\times 10^6$ cycles) . . 34
3.4 Calculating time of texture features of 5 structures after texture feature optimization, gray level Ng =256 ($\times 10^6$ cycles) 36
3.5 Total execution time of 5 structures after texture feature optimization, gray level Ng =256 ($\times 10^6$ cycles) 36
3.6 Calculation time of texture features of 2 structures with different non-zero elements, image size =128x128 gray level Ng =256 37

4.1 Average latency of mailbox communication in us 48
4.2 specification of running environment 65
4.3 Scalar implementation of building co-concurrence matrix for different image sizes and the number of gray level is 128 66
4.4 Building time of co-concurrence matrix with different gray level, image size 512x512 67
4.5 Large data type implementation of building co-concurrence matrix of the image of size 512 × 512, gray level Ng =64 68
4.6 Execution time of the calculation 13 texture features for different image sizes when the number of the gray level is 128 69
4.7 Execution time of the calculation 13 texture features for different gray levels when the image size is 512 × 512 69
4.8 Execution time of building co-concurrence matrix of non-zero element (in bold text) and normal array approach, the number of gray level is 64 . . . 70
4.9 Execution time of calculating texture features of non-zero element (in bold text) and normal array approach, the number of gray level is 64 71
4.10 Total execution time of calculating co-concurrence matrix and texture features of non-zero element (in bold text) and normal array approach, the number of gray level is 64 71
4.11 Performance(time in μs and speed-up in comparison with Core2 Duo) of co-occurrence matrix with different platforms when gray level is 64 72
4.12 Performance(time in μs and speed-up in comparison with Core2 Duo) of texture features with different platforms when gray level is 64 74

4.13 Performance(time in μs and speed-up in comparison with Core2 Duo) of texture features with different platforms when gray level is 128 74

Acknowledgements

First and foremost, this thesis is dedicated to my parents and other family members, who have unconditionally given me all of their support on all fronts. Without your love and encouragement, I would not have reached this far.

I would like to take this opportunity to express my special thanks to Dr. Asadollah Shahbahami for his advice, support, and encouragement during my MSc project.

My special thanks goes to my supervisor Dr. Koen Bertels for his time and help.

I wish to thank helpful and friendly classmates, PhD students and professors at the CE group, Arnaldo Azevedo, Cor Meenderinck, Vikram etc,. for giving me valuable advices during the time I work on the thesis.

Special thanks to Viet Phuong, Hung V.X my housemate for sharing thoughts, culture, cuisine, movies and enjoying life together. I appropriate Ba Thang, my Vietnamese classmate, for his help in study an in life as well during two years in Delft. I would also like to say thank you to Lan T.H for believing and encouraging me all the time we know each other.

Last but not least, I owe many thanks to all my friends in Vietnam for always giving me supports and encouragements whenever I need.

Tuan Anh Pham
Delft, The Netherlands
July 29, 2010

Introduction

1

Texture is a significant feature of an image that has been widely used in medical image analysis, image classification, automatic visual inspection and remote sensing [36] [25] [20]. Generally speaking, textures are complex visual patterns composed of entities, or sub-patterns, that have characteristic brightness, color, slope, size, etc. A basic stage to collect such features through texture analysis process is texture feature extraction. Texture features can be extracted in several methods, using statistical, structural, model-based and transform information, in which a well-known method is using a Gray Level Co-occurrence Matrix (GLCM) [26]. GLCM contains the second-order statistical information of spatial relationship of pixels of an image. From GLCM, Haralick [26] proposed 13 common statistical features, known as Haralick texture features.

GLCM is a useful tool in texture analysis [36] [37], however the overall calculations for the computation of GLCM and texture features are computationally intensive and time-consuming. In [18], Tahir described an example of calculations of GLCM and texture features in a medical application. With an image of size 5000×5000 pixels with 16 bands, the time required is approximately 350 seconds using Pentium 4 machine running at 2400 MHz. 75% of the total time spent is for the calculation of GLCM, 5% for the normalisation, 19% for the calculation of texture features while 1% is for the classification using classical discrimination method. Due to such heavy computation, there were many reasearches on accelerating the process of calculation GLCM and texture features. In one method the image is represented by four or five bits instead of eight bits that makes to reduce the size of GLCM but it removes some information about the image. Another method is to reduce the size of GLCM by storing just non-zero values, using the Gray Level Co-occurrence Linked List (GLCLL) [3], or Gray Level Co-occurrence Hybrid Structure (GLCHS) [5]. Eizan Miyamoto [24] also proposed some techniques to fast calculate Haralick texture features.

With the emerging of parallel processing, there are several researches about computing GLCM and texture features in parallel. Khalaf et al [27] proposed a hardware architecture following odd-even network topology to parallelize GLCM. Markus Gipp et al [19] accelerated the computation of Haralicks texture features using Graphics Processing Units (GPUs). Tahir [18] presented an FPGA based co-processor for GLCM and texture features and their application in prostate cancer classification. Results from these researches demonstrated that parallel processing could provide significant increase in speed for GLCM and texture feature computation.

In this thesis, computation of GLCM and texture features are implemented in C++ to demonstrate its time consumption. Several methods of optimization of GLCM and texture features are also investigated and compared. Finally, parallel computing of GLCM and texture features is proposed using Cell Broadband Engine Architecture (Cell Processor). Cell processor, developed by IBM, Sony and Toshiba, is a heterogeneous chip-multiprocessor (CMP) architecture to offer very high performance, especially on multimedia applications. It consists of a traditional microprocessor (PPE) that controls eight SIMD co-processing units (SPE), which are optimized for heavy computing applications. Using Cell processor gives a significant speed-up in calculating GLCM and texture features.

Section 1.1 and 1.2 declare the objective and organization of the thesis.

1.1 Thesis Objective

- To investigate different approaches to extract texture features extraction.
- To introduce gray-level co-occurrence matrix as a effective technique in texture feature extraction. In addition, to show that this algorithm is time-consuming.
- To study and propose improvements in optimization of calculation co-occurrence matrix and texture features.
- To parallelize the computation of co-occurrence matrix and texture features using Cell Broadband Engine Architecture.

1.2 Thesis Organization

The rest of thesis is organized as follows. In Chapter 2, the first section describes texture features and different techniques that can be used for texture feature extraction. Then in the next section, GLCM and Haralick texture features are implemented in C++. In Chapter 3, several optimization methods in calculating GLCM and Haralick texture features are implemented and compared. In Chapter 4, parallel processing on Cell processor is described. Finally, in Chapter 5 the work is concluded and future work is discussed.

2

Texture Features

This chapter contains an overview of texture features, gray-level co-occurrence matrix and feature extraction. Section 2.1 introduces about image analysis. The following section provides a discussion about texture and different methods to extract texture features, in which co-occurrence matrix and Haralick texture features are focused. Finally, section 2.3 is a C++ implementation of texture feature extraction based on co-occurrence matrix and experimental results.

2.1 Image Analysis

Today, images play a crucial role in fields as diverse as science, medicine, journalism, advertising, design, education and entertainment. Therefore image analysis with the aid of computer becomes more and more substantial in all research fields. Image analysis involves investigation of the image data for a specific application. Normally, the raw data of a set of images is analyzed to gain insight into what is happening with the images and how they can be used to extract desired information. The image analysis involves image segmentation, image transformation, pattern classification, and feature extraction [37]

- Image segmentation: It divides the input image into multiple segments or regions, which show objects or meaningful parts of objects. It segments image into homogeneous regions thus making it easier to analyze them.
- Image transformation: It is used to find the spatial frequency information that can be used in the feature extraction step.
- Pattern classification: It aims to classify data (patterns) based either on a priori knowledge or on statistical information extracted from the image.
- Feature extraction: It is the process of acquiring higher level information of an image, such as color, shape, and texture. Features contain the relevant information of an image and will be used in image processing (e.g. searching, retrieval, storing). Features are divided into different classes based on the kind of properties they describe. Some important features are as follows.

Color

Color is a visual attribute of things that results from the light they emit or transmit or reflect. From a mathematical viewpoint, the extension from luminance to color signals is an extension from scalar-signals to vector-signals. Two major advantages of using color vision are: 1. color provides extra information which allows the distinction between various physical causes for color variations in the world, such as changes due to shadows, light source reflections, and object reflectance variations; 2. Color is an important discriminative property of objects, allowing us to distinguish between fresh water and coca-cola. Color features can be derived from a histogram of the image. The weakness of color histogram is that the color histogram of two different things with the same color, can be equal. However, color features are still useful for biomedical image processing, such as for cell classification, and cancer cell detection [25], or for content-based image retrieval (CBIR) systems. In CBIR, every image added to the collection is analyzed to compute a color histogram. At search time, the user can either specify the desired proportion of each color or submit an example image from which a color histogram is calculated. Either way, the matching process then retrieves those images whose color histograms match those of the query most closely. The matching technique most commonly used, histogram intersection, was first developed by Swain and Ballard [34]. Variants of this technique are now used in a high proportion of current CBIR systems. Methods of improving on Swain and Ballard's original technique include the use of cumulative color histograms [32], combining histogram intersection with some elements of spatial matching [31] and the use of region-based color querying [7].

Texture

Texture is a very general notion that is difficult to describe in words. The texture relates mostly to a specific, spatially repetitive structure of surfaces formed by repeating a particular element or several elements in different relative spatial positions. John R. Smith defines texture as visual patterns with properties of homogeneity that do not result from the presence of only a single color such as clouds and water [29]. Texture features are useful in many applications such as in medical imaging [25], remote sensing [20] and CBIR. In CBIR, there are many techniques to measure texture similarity, the best-established rely on comparing values of what are known as second-order statistics calculated from query and stored images. Essentially, they calculate the relative brightness of selected pairs of pixels from each image. From these, it is possible to calculate measures of image texture such as the degree of contrast, coarseness, directionality and regularity [9], or periodicity, directionality and randomness [17]. Alternative methods of texture analysis for retrieval include the use of Gabor filters [21] and fractals [10].

Shape

Unlike texture, shape is a fairly well-defined concept. The shape of an object located in some space is the part of that space occupied by the object, as determined by its external boundary abstracting from other properties such as color, content, and material composition, as well as from the object's other spatial properties. Mathematician Kendall [16] defined shape as all the geometrical information that remains when location, scale and rotational effects are filtered out from an object. Shape features can be used for medical applications for example for cervical cell classification or for CBIR. Two main types of shape feature are commonly used - global features such as aspect ratio, circularity and moment invariants [8] and local features such as sets of consecutive boundary segments [23].

2.2 Texture Feature

2.2.1 Definition of texture

Texture is a conception that is easy to recognize but very difficult to define. This difficulty is demonstrated by the number of different texture definitions attempted by vision researchers, some of them are as follows.

- Texture is visual patterns with properties of homogeneity that do not result from the presence of only a single color such as clouds and water [29].
- A region in an image has a constant texture if a set of local statistics or other local properties of the picture function are constant, slowly varying, or approximately periodic [28].
- An image texture is described by the number and types of its (tonal) primitives and the spatial organization or layout of its (tonal) primitives... A fundamental characteristic of texture: it cannot be analyzed without a frame of reference of tonal primitive being stated or implied. For any smooth gray-tone surface, there exists a scale such that when the surface is examined, it has no texture. Then as resolution increases, it takes on a fine texture and then a coarse texture [11].
- The notion of texture appears to depend upon three ingredients: (i) some localorder is repeated over a region which is large in comparison to the orders size,(ii) the order consists in the nonrandom arrangement of elementary parts, and (iii) the parts are roughly uniform entities having approximately the same dimensions everywhere within the textured region [12].

Figure 2.1 depicts some examples of different texture features. Textures might be divided into two categories, namely, touch and visual textures. Touch textures relate to the touchable feel of a surface and range from the smoothest (little difference between high and low points) to the roughest (large difference between high and low points). Visual textures refer to the visual impression that textures produce to human observer, which are related to local spatial variations of simple stimuli like color, orientation and intensity in an image.

2.2.2 Texture Analysis

Major goals of texture research in computer vision are to understand, model and process texture. Four major application domains related to texture analysis are texture classification, texture segmentation, shape from texture, and texture synthesis [36]:

- Texture classification: It produces a classification map of the input image where each uniform textured region is identified with the texture class it belongs.
- Texture segmentation: It makes a partition of an image into a set of disjoint regions based on texture properties, so that each region is homogeneous with respect to certain texture characteristics. Results of segmentation can be applied to further image processing and analysis, for instance, to object recognition.

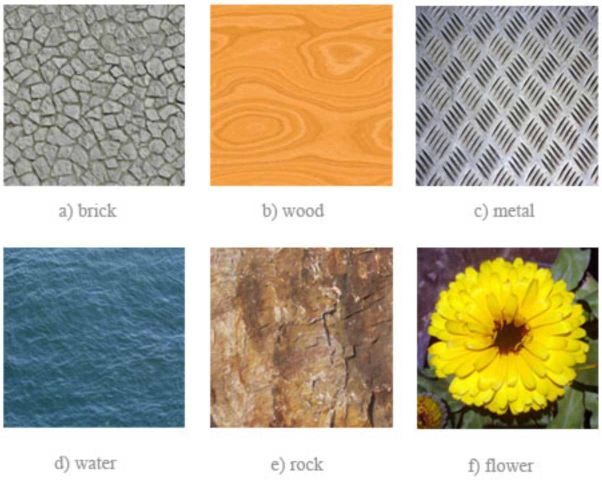


Figure 2.1: Examples of texture features

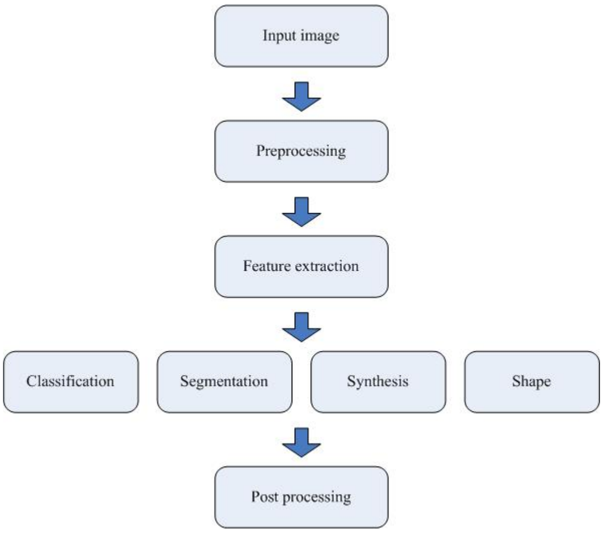


Figure 2.2: Different steps in the texture analysis process

- Texture synthesis: It is a common technique to create large textures from usually small texture samples, for the use of texture mapping in surface or scene rendering applications.

- Shape from texture: It reconstructs 3D surface geometry from texture information.

In all four types of texture analysis, texture extraction is an inevitable stage. A typical process of texture analysis in a computer vision system can be divided into components showed in Figure 2.2

2.2.3 Application of Texture

Texture analysis methods have been utilized in a variety of application domains such as: automated inspection, medical image processing, document processing, remote sensing and content-based image retrieval. In some of the mature domains (such as remote sensing or CBIR) texture has already played a major role, while in other disciplines (such as surface inspection) new applications of texture are being found.

Remote Sensing

Texture analysis has been extensively used to classify remotely sensed images. Land use classification where homogeneous regions with different types of terrains (such as wheat, bodies of water, urban regions, etc.) need to be identified is an important application. Haralick et al [26] used gray level co-occurrence features to analyze remotely sensed images. They computed gray level co-occurrence matrices for a distance of one with four directions. They obtained approximately 80% classification accuracy using texture features.

Medical Image Analysis

Image analysis techniques have played an important role in several medical applications. In general, the applications involve the automatic extraction of features from the image which are then used for a variety of classification tasks, such as distinguishing normal tissue from abnormal tissue. Depending upon the particular classification task, the extracted features capture morphological properties, color properties, or certain textural properties of the image. For example, Sutton and Hall [33] discussed the classification of pulmonary disease using texture features.

2.2.4 Texture Feature Extraction Algorithms

Tuceryan and Jain [36] divided the different methods for feature extraction into four main categories, namely: structural, statistical, model-based and transform domain, which are briefly explained in the following sections.

2.2.4.1 Structural Method

Structural approaches [11] represent texture by well-defined primitives (microtexture) and a hierarchy of spatial arrangements (macrotexture) of those primitives. To describe the texture, one must define the primitives and the placement rules. The choice of a primitive (from a set of primitives) and the probability of the chosen primitive to be placed at a particular location can be a function of location or the primitives near the location. The advantage of the structural approach is that it provides a good symbolic description of the image; however, this feature is more useful for synthesis than analysis tasks. This method is not suitable for natural textures because of the variability both of micro-texture and macro-texture and there is no clear distinction between them.

2.2.4.2 Statistical Method

Statistical methods represent the texture indirectly according to the non-deterministic properties that manage the distributions and relationships between the gray levels of an image. This technique is one of the first methods in machine vision [36]. By computing local features at each point in the image and deriving a set of statistics from the distributions of the local features, statistical methods can be used to analyze the spatial distribution of gray values. Based on the number of pixels defining the local feature, statistical methods can be classified into first-order (one pixel), second-order(pair of pixels) and higher-order (three or more pixels) 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 co-occurrence matrix [22]. Statistical method based on co-occurrence matrix will be discussed in section 2.3.

2.2.4.3 Model-based

Model based texture analysis such as Fractal model and Markov are based on the construction of an image that can be used for describing texture and synthesizing it [36]. These methods describe an image as a probability model or as a linear combination of a set of basic functions. The Fractal model is useful for modeling certain natural textures that have a statistical quality of roughness at different scales [36] , and also for texture analysis and discrimination. This method has a weakness in orientation selectivity and is not useful for describing local image structures. Pixel-based models view an image as a collection of pixels, whereas region-based models view an image as a set of sub patterns. There are different types of models based on the different neighborhood systems and noise sources. These types are one-dimensional time-series models, Auto Regressive (AR), Moving Average (MA) and Auto Regressive Moving Average (ARMA). Random field models analyze spatial variations in two dimensions, global random and local random. Global random field models treat the entire image as a realization of a random field, and local random field models assume relationships of intensities in small neighborhoods. A widely used class of local random field models are Markov models, where the conditional probability of the intensity of a given pixel depends only on the intensities of the pixels in its neighborhood (the so-called Markov neighbors) [22].

2.2.4.4 Transform Method

Transform methods, such as Fourier, Gabor and wavelet transforms represent an image in a space whose co-ordinate system has an interpretation that is closely related to the characteristics of a texture (such as frequency or size). They analyze the frequency content of the image. Methods based on Fourier transforms have a weakness in a spatial localization so they do not perform well. Gabor filters provide means for better spatial localization but their usefulness is limited in practice because there is usually no single filter resolution where one can localize a spatial structure in natural textures [22]. These methods involve transforming original images by using filters and calculating the energy of the transformed images. They are based on the process of the whole image that is not good for some applications which are based on one part of the input image.

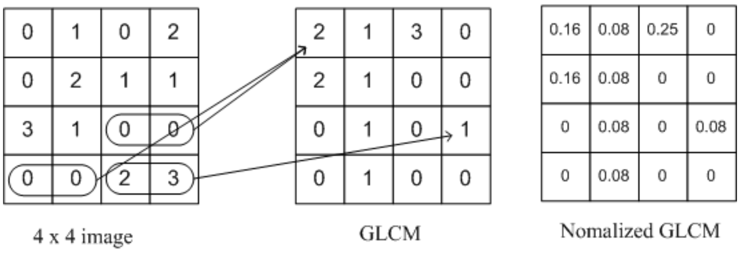


Figure 2.3: GLCM of a 4 × 4 image for distance $d = 1$ and direction θ=0

2.3 Grey Level Co-occurrence Matrix and Haralick Texture Features

In 1973, Haralick [26] 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 co-occurrence matrix and the second step is calculating texture feature base on the co-occurrence matrix. this technique is useful in wide range of image analysis applications from biomedical to remote sensing techniques.

2.3.1 Gray-level Co-occurrence Matrix

One of the defining qualities of texture is the spatial distribution of gray values. The use of statistical features is therefore one of the early methods proposed in the image processing literature. Haralick [26] suggested the use of co-occurrence matrix or gray level co-occurrence matrix. It considers the relationship between two neighboring pixels, the first pixel is known as a reference and the second is known as a neighbor pixel. In the following, we will use $\{I\left(x,y\right),0\leq x\leq N_x-1,0\leq y\leq N_y-1\}$ to denote an image with G gray levels. The $G\times G$ gray level co-occurrence matrix P_d^θ for a displacement vector $d=(dx,dy)$ and direction θ is defined as follows. The element (i,j) of P_d^θ is the number of occurrences of the pair of gray levels i and j which the distance between i and j following direction θ is d.

$$P_d^\theta(i,j)=\#\left\{\left(\left(r,s\right),\left(t,v\right)\right):I\left(r,s\right)=i,I\left(t,v\right)=j\right\}$$

Where $\left(r,s\right),\left(t,v\right)\in N_x\times N_y;\left(t,v\right)=\left(r+dx,s+dy\right)$.

Figure 2.3 shows the co-occurrence matrix P_d^θ with distance $d=1$ and the direction is horizontal ($\theta=0$). This relationship ($d=1,\theta=0$) is nearest horizontal neighbor. There will be (N_x-1) neighboring resolution cell pairs for each row and there are N_y rows, providing $R=(N_x-1)\times N_y$ nearest horizontal pairs. The co-occurrence matrix can be normalized by dividing each of its entry by R.

In addition, there are also co-occurrence matrices for vertical direction ($\theta=90$) and both diagonal directions ($\theta=45,135$). If the direction from bottom to top and from left to right is considered, there will be eight directions (0, 45, 90, 135, 180, 225, 270, 315) (Figure 2.4). From the co-occurrence matrix, Haralick proposed a number of useful texture features.

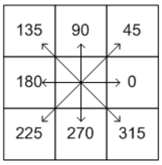


Figure 2.4: Eight directions of adjacency

2.3.2 Haralick Texture Features

Haralick extracted thirteen texture features from GLCM for an image. These features are as follows:

2.3.2.1 Angular second moment (ASM) feature

The ASM is known as uniformity or energy. It measures the uniformity of an image. When pixels are very similar, the ASM value will be large.

$$f_1 = \sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} p_{d,\theta}(i,j)^2 \tag{2.1}$$

2.3.2.2 Contrast feature

Contrast is a measure of intensity or gray-level variations between the reference pixel and its neighbor. In the visual perception of the real world, contrast is determined by the difference in the color and brightness of the object and other objects within the same field of view.

$$f_2 = \sum_{n=0}^{N_g-1} n^2 \left\{ \sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} p_{d,\theta}(i,j) \right\} , where\ n = |i-j| \tag{2.2}$$

When i and j are equal, the cell is on the diagonal and $i - j = 0$. These values represent pixels entirely similar to their neighbor, so they are given a weight of 0. If i and j differ by 1, there is a small contrast, and the weight is 1. If i and j differ by 2, the contrast is increasing and the weight is 4. The weights continue to increase exponentially as $(i - j)$ increases.

2.3.2.3 Entropy Feature

Entropy is a difficult term to define. The concept comes from thermodynamics, it refers to the quantity of energy that is permanently lost to heat every time a reaction or a physical transformation occurs. Entropy cannot be recovered to do useful work. Because of this, the term can be understood as amount of irremediable chaos or disorder. The equation of entropy is:

$$f_3 = \sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} p_{d,\theta}(i,j) log \left(p_{d,\theta}(i,j) \right) \tag{2.3}$$

2.3.2.4 Variance Feature

Variance is a measure of the dispersion of the values around the mean of combinations of reference and neighbor pixels. It is similar to entropy, answers the question 'What is the dispersion of the difference between the reference and the neighbor pixels in this window?'

$$f_4 = \sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} (i-\mu)^2 \, p_{d,\theta}(i,j) \tag{2.4}$$

2.3.2.5 Correlation Feature

Correlation feature shows the linear dependency of gray level values in the co-occurrence matrix. It presents how a reference pixel is related to its neighbor, 0 is uncorrelated, 1 is perfectly correlated.

$$f_5 = \sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} p_{d,\theta}(i,j) \frac{(i-\mu_x)(j-\mu_y)}{\sigma_x \sigma_y} \tag{2.5}$$

Where μ_x, μ_y and σ_x, σ_y are the means and standard deviations of p_x and p_y.

$$\mu_x = \sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} i.p_{d,\theta}(i,j) \quad \mu_y = \sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} j.p_{d,\theta}(i,j) \tag{2.6}$$

$$\sigma_x = \sqrt{\sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} (i-\mu)^2 \, p_{d,\theta}(i,j)} \quad \sigma_y = \sqrt{\sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} (j-\mu)^2 \, p_{d,\theta}(i,j)} \tag{2.7}$$

For the symmetrical GLCM, $\mu_x = \mu_y$ and $\sigma_x = \sigma_y$.

2.3.2.6 Inverse Difference Moment (IDM) Feature

IDM is usually called homogeneity that measures the local homogeneity of an image. IDM feature obtains the measures of the closeness of the distribution of the GLCM elements to the GLCM diagonal.

$$f_6 = \sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} \frac{1}{1+(i-j)^2} p_{d,\theta}(i,j) \tag{2.8}$$

IDM weight value is the inverse of the Contrast weight, with weights decreasing exponentially away from the diagonal.

2.3.2.7 Sum Average Feature

$$f_7 = \sum_{i=0}^{2(N_g-1)} i.p_{x+y}(i) \tag{2.9}$$

where:

$$p_{x+y}(k) = \sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} p_{d,\theta}(i,j) \quad , k = i + j = \{0, 1, 2, ..., 2(N_g - 1)\} \tag{2.10}$$

2.3.2.8 Sum Variance Feature

$$f_8 = \sum_{i=0}^{2(N_g-1)} (i - f_7)^2 p_{x+y}(i) \tag{2.11}$$

2.3.2.9 Sum Entropy Feature

$$f_9 = -\sum_{i=0}^{2(N_g-1)} p_{x+y}(i) log p_{x+y}(i) \tag{2.12}$$

2.3.2.10 Difference Variance Feature

$$f_{10} = \sum_{i=0}^{N_g-1} \left(i - f_{10}^{'}\right)^2 p_{x-y}\left(i\right) \tag{2.13}$$

where:

$$p_{x-y}(k) = \sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} p_{d,\theta}(i,j) \quad , k = |i - j| = \{0, 1, 2, ..., (N_g - 1)\} \tag{2.14}$$

$$f_{10}^{'} = \sum_{i=0}^{N_g-1} i.p_{x-y}(i) \tag{2.15}$$

2.3.2.11 Difference Entropy Feature

$$f_{11} = -\sum_{i=0}^{N_g-1} p_{(x-y)}(i) log p_{x-y}(i) \tag{2.16}$$

2.3.2.12 Information Measures of Correlation Feature 1

$$f_{12} = \frac{HXY - HXY1}{max(HX, HY)} \tag{2.17}$$

2.3.2.13 Information Measures of Correlation Feature 2

$$f_{13} = (1 - exp\left[-2(HXY2 - HXY)\right])^{1/2} \tag{2.18}$$

where:

$$p_x(i) = \sum_{j=0}^{N_g-1} p_{d,\theta}(i,j) \tag{2.19}$$

$$p_y(j) = \sum_{i=0}^{N_g-1} p_{d,\theta}(i,j) \tag{2.20}$$

$$HX = -\sum_{i=0}^{N_g-1} p_x(i) log(p_x(i)) \tag{2.21}$$

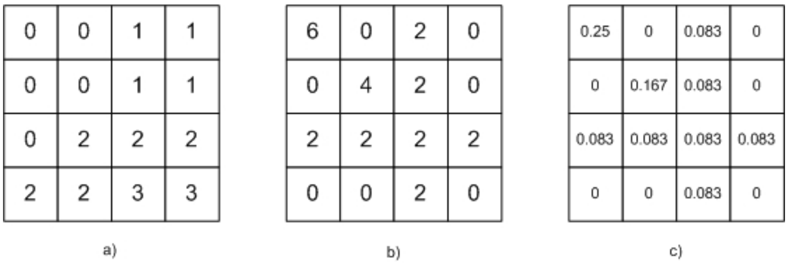
$$HY = -\sum_{i=0}^{N_g-1} p_y(i) log(p_y(i)) \tag{2.22}$$

$$HXY = -\sum_{i=0}^{N_g-1}\sum_{j=0}^{N_g-1} p_{d,\theta}(i,j) log\left(p_{d,\theta}(i,j)\right) \tag{2.23}$$

$$HXY1 = -\sum_{i=0}^{N_g-1}\sum_{j=0}^{N_g-1} p_{d,\theta}(i,j) log\left(p_x(i)p_y(j)\right) \tag{2.24}$$

$$HXY2 = -\sum_{i=0}^{N_g-1}\sum_{j=0}^{N_g-1} p_x(i)p_y(j) log\left(p_x(i)p_y(j)\right) \tag{2.25}$$

In all these notations and formulas above, the value of $p_{d,\theta}$ is the value after normalization by dividing by R. To demonstrate these features, we consider an example of a 4×4 gray-scale image [Figure 2.5].



0	0	1	1
0	0	1	1
0	2	2	2
2	2	3	3

a)

6	0	2	0
0	4	2	0
2	2	2	2
0	0	2	0

b)

0.25	0	0.083	0
0	0.167	0.083	0
0.083	0.083	0.083	0.083
0	0	0.083	0

c)

Figure 2.5: 4x4 gray-scale image(a); Co-occurrence matrix(b); Normalized Co-occurrence matrix (c)

Figure 2.5a is a 4×4 gray-scale image, Figure 2.5b is Co-occurrence matrix of the image following vertical direction($\theta = 90$ and $\theta = 270$) with distance $d = 1$. Figure 2.5c is Co-occurrence matrix after normalization (each entry is divided by the total number of possible pairs, i.e., 24). We will calculate 13 texture features following Haralick's formula.

Energy =
$0.250^2 + 0.000^2 + 0.083^2 + 0.000^2$
$+0.000^2 + 0.167^2 + 0.083^2 + 0.000^2$
$+0.083^2 + 0.083^2 + 0.083^2 + 0.083^2$
$+0.000^2 + 0.000^2 + 0.083^2 + 0.000^2$
$= 0.1386$

Contrast =

$$
\begin{aligned}
&(0-0)^2*0.250+(0-1)^2*0.000+(0-2)^2*0.083+(0-3)^2*0.000\\
&+(1-0)^2*0.000+(1-1)^2*0.167+(1-2)^2*0.083+(1-3)^2*0.000\\
&+(2-0)^2*0.083+(2-1)^2*0.083+(2-2)^2*0.083+(2-3)^2*0.083\\
&+(3-0)^2*0.000+(3-1)^2*0.000+(3-2)^2*0.083+(3-3)^2*0.000\\
&=0.9960
\end{aligned}
$$

Entropy =

$$
\begin{aligned}
&0.250*ln(0.250)+0.000+0.083*ln(0.083)+0.000\\
&+0.000+0.167*ln(0.167)+0.083*ln(0.083)+0.000\\
&+0.083*ln(0.083)+0.083*ln(0.083)+0.083*ln(0.083)+0.083*ln(0.083)\\
&+0.000+0.000+0.083*ln(0.083)+0.000\\
&=2.0915
\end{aligned}
$$

Mean(μ) =

$$
\begin{aligned}
&0*0.250+0*0.000+0*0.083+0*0.000\\
&+1*0.000+1*0.167+1*0.083+1*0.000\\
&+2*0.083+2*0.083+2*0.083+2*0.083\\
&+3*0.000+3*0.000+3*0.083+3*0.000\\
&=1.1630
\end{aligned}
$$

Variance =

$$
\begin{aligned}
&(0-1.163)^2*0.250+(0-1.163)^2*0.000+(0-1.163)^2*0.083+(0-1.163)^2*0.000\\
&+(1-1.163)^2*0.000+(1-1.163)^2*0.167+(1-1.163)^2*0.083+(1-1.163)^2*0.000\\
&+(2-1.163)^2*0.083+(2-1.163)^2*0.083+(2-1.163)^2*0.083+(2-1.163)^2*0.083\\
&+(3-1.163)^2*0.000+(3-1.163)^2*0.000+(3-1.163)^2*0.083+(3-1.163)^2*0.000\\
&=0.9697
\end{aligned}
$$

Correlation =

$$
\begin{aligned}
&(0-1.163)*(0-1.163)*0.250+(0-1.163)*(1-1.163)*0.000\\
&+(0-1.163)*(2-1.163)*0.083+(0-1.163)*(3-1.163)*0.000\\
&+(1-1.163)*(0-1.163)*0.000+(1-1.163)*(1-1.163)*0.167\\
&+(1-1.163)*(2-1.163)*0.083+(1-1.163)*(3-1.163)*0.000\\
&+(2-1.163)*(0-1.163)*0.083+(2-1.163)*(1-1.163)*0.083\\
&+(2-1.163)*(2-1.163)*0.083+(2-1.163)*(3-1.163)*0.083\\
&+(3-1.163)*(0-1.163)*0.000+(3-1.163)*(1-1.163)*0.000\\
&+(3-1.163)*(2-1.163)*0.083+(3-1.163)*(3-1.163)*0.000\\
&=0.5119
\end{aligned}
$$

Homogeneity =

$$
\begin{aligned}
&1/(1+(0-0)^2)*0.250+1/(1+(0-1)^2)*0.000\\
&+1/(1+(0-2)^2)*0.083+1/(1+(0-3)^2)*0.000\\
&+1/(1+(1-0)^2)*0.000+1/(1+(1-1)^2)*0.167\\
&+1/(1+(1-2)^2)*0.083+1/(1+(1-3)^2)*0.000\\
&+1/(1+(2-0)^2)*0.083+1/(1+(2-1)^2)*0.083\\
&+1/(1+(2-2)^2)*0.083+1/(1+(2-3)^2)*0.083
\end{aligned}
$$

$+1/(1+(3-0)^2)*0.000+(1/(1+(3-1)^2)*0.000$
$+1/(1+(3-2)^2)*0.083+1/(1+(3-3)^2)*0.000$
$=0.7213$

Sum Average =
$0*0.250+1*(0.000+0.000)+2*(0.083+0.167+0.083)$
$+3*(0.000+0.083+0.083+0.000)+4*(0.000+0.083+0.000)$
$+5*(0.083+0.083)+6*0.000$
$=2.3260$

Sum Variance =
$(0-2.326)^2*0.250+(1-2.326)^2*(0.000+0.000)+(2-2.326)^2*(0.083+0.167+0.083)$
$+(3-2.326)^2*(0.000+0.083+0.083+0.000)+(4-2.326)^2*(0.000+0.083+0.000)$
$+(5-2.326)^2*(0.083+0.083)+(6-2.326)^2*0.000$
$=2.8829$

Sum Entropy =
$0.250*ln(0.250)+(0.000+0.000)+(0.083+0.167+0.083)*ln(0.083+0.167+0.083)$
$+(0.000+0.083+0.083+0.000)*ln(0.000+0.083+0.083+0.000)$
$+(0.000+0.083+0.000)*ln(0.000+0.083+0.000)$
$+(0.083+0.083)*ln(0.083+0.083)+0.000$
$=1.5155$

Difference Average =
$0*(0.250+0.167+0.083+0.000)$
$+1*(0.000+0.000+0.083+0.083+0.000+0.000)$
$+2*(0.083+0.083+0.000+0.000)+3*(0.000+0.000)$
$=0.498$

Difference Variance =
$(0-0.498)^2*(0.250+0.167+0.083+0.000)$
$+(1-0.498)^2*(0.000+0.000+0.083+0.083+0.000+0.000)$
$+(2-0.498)^2*(0.083+0.083+0.000+0.000)+(3-0.498)^2*(0.000+0.000)$
$=0.5542$

Difference entropy =
$(0.250+0.167+0.083+0.000)*ln(0.250+0.167+0.083+0.000)$
$+(0.000+0.000+0.083+0.083+0.000)*ln(0.000+0.000+0.083+0.083+0.000)$
$+(0.083+0.083+0.000+0.000)*ln(0.083+0.083+0.000+0.000)$
$=1.0107$

$p_x(0)=p_y(0)=0.250+0.000+0.083+0.000=0.333$
$p_x(1)=p_y(1)=0.000+0.167+0.083+0.000=0.250$
$p_x(2)=p_y(2)=0.083+0.083+0.083+0.083=0.333$

$p_x(3) = p_y(3) = 0.000 + 0.000 + 0.083 + 0.000 = 0.083$

HX =
$-0.333 * ln(0.333) - 0.250 * ln(0.250) - 0.333 * ln(0.333) - 0.083 * ln(0.083)$
$= 1.2855$

HXY =
$0.250 * ln(0.250) + 0.000 + 0.083 * ln(0.083) + 0.000$
$+0.000 + 0.167 * ln(0.167) + 0.083 * ln(0.083) + 0.000$
$+0.083 * ln(0.083) + 0.083 * ln(0.083) + 0.083 * ln(0.083) + 0.083 * ln(0.083)$
$+0.000 + 0.000 + 0.083 * ln(0.083) + 0.000$
$= 2.0915$

HXY1 =
$-0.250 * ln(0.333 * 0.333) - 0.000 * ln(0.333 * 0.250)$
$-0.083 * ln(0.333 * 0.333) - 0.000 * ln(0.333 * 0.083)$
$-0.000 * ln(0.250 * 0.333) - 0.167 * ln(0.250 * 0.250)$
$-0.083 ln(0.250 * 0.333) - 0.000 * ln(0.250 * 0.083)$
$-0.083 * ln(0.333 * 0.333) - 0.083 * ln(0.333 * 0.250)$
$-0.083 * ln(0.333 * 0.333) - 0.083 * ln(0.333 * 0.083)$
$-0.000 * ln(0.083 * 0.333) - 0.000 * ln(0.083 * 0.250)$
$-0.083 * ln(0.083 * 0.333) - 0.000 * ln(0.083 * 0.083)$
$= 2.5688$

HXY2 =
$-0.333 * 0.333 * ln(0.333 * 0.333) - 0.333 * 0.250 * ln(0.333 * 0.250)$
$-0.333 * 0.333 * ln(0.333 * 0.333) - 0.333 * 0.083 * ln(0.333 * 0.083)$
$-0.250 * 0.333 * ln(0.250 * 0.333) - 0.250 * 0.250 * ln(0.250 * 0.250)$
$-0.250 * 0.333 ln(0.250 * 0.333) - 0.250 * 0.083 * ln(0.250 * 0.083)$
$-0.333 * 0.333 * ln(0.333 * 0.333) - 0.333 * 0.250 * ln(0.333 * 0.250)$
$-0.333 * 0.333 * ln(0.333 * 0.333) - 0.333 * 0.083 * ln(0.333 * 0.083)$
$-0.083 * 0.333 * ln(0.083 * 0.333) - 0.083 * 0.250 * ln(0.083 * 0.250)$
$-0.083 * 0.333 * ln(0.083 * 0.333) - 0.083 * 0.083 * ln(0.083 * 0.083)$
$= 2.5684$

Information Measures of Correlation 1 =
$= (2.0915 - 2.5688)/1.2855 = -0.3713$

Information Measures of Correlation 2 =
$= (1 - exp(-2 * (2.5684 - 2.0915)))^{1/2} = 0.7840$

Table 2.1 summarizes these 13 texture features of the image.

	Texture features	Value
1	Energy	0.1386
2	Contrast	0.9960
3	Entropy	2.0915
4	Variance	0.9697
5	Correlation	0.5119
6	IDM	0.7213
7	Sum average	2.3260
8	Sum variance	2.8829
9	Sum entropy	1.5155
10	Difference variance	0.5542
11	Difference entropy	1.0107
12	Information measures of correlation 1	-0.3713
13	Information measures of correlation	0.7840

Table 2.1: texture features of the 4 × 4 image

2.4 C++ Implementation for Calculating Co-occurrence Matrix and Texture Features

2.4.1 General Structure of the Implementation

The following C++ program deploys gray-level Co-occurrence matrix and Haralick texture feature algorithm. The program has five main steps of [Figure 2.6]:

- Step 1: read the command line from the users
- Step 2: read the content of the image from .bmp file
- Step 3: calculate the co-occurrence matrix
- Step 4: calculate Haralick texture features
- Step 5: save acquired information to a database file.

2.4.2 Data Structure

Class `CoOccurrenceMatrix` [Listing 2.1] is responsible for all calculations of Co-occurrence matrix and texture features. Variable `m_COM` is a $Ng \times Ng$ two-dimension array used to store Co-occrurrence matrix ($Ng \times Ng$ is the sized Co-occrurrence matrix, where Ng is gray level). Float variables `m_fEnergy`, `m_fEntropy`, `m_fContrast`... are used to store the value of texture features Energy, Entropy, Contrast,... respectively.

The function `CalculateCoocurrenceMatrix()`[Listing 2.2] is used for computing co-occurrence matrix. In this method, the co-occurrence matrix is computed using the function `UpdatePixel()` in eight directions with distance of one.

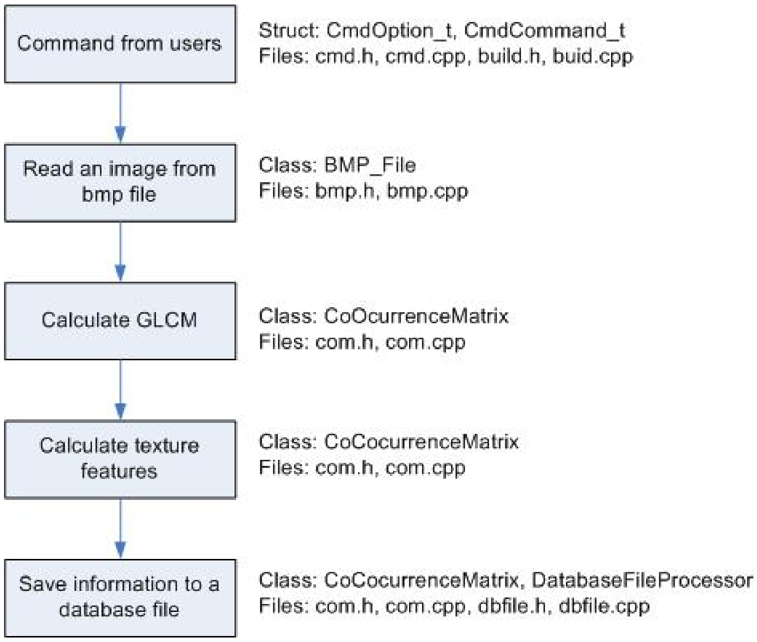

Command from users
Struct: CmdOption_t, CmdCommand_t
Files: cmd.h, cmd.cpp, build.h, buid.cpp
Read an image from bmp file
Class: BMP_File
Files: bmp.h, bmp.cpp
Calculate GLCM
Class: CoOcurrenceMatrix
Files: com.h, com.cpp
Calculate texture features
Class: CoCocurrenceMatrix
Files: com.h, com.cpp
Save information to a database file
Class: CoCocurrenceMatrix, DatabaseFileProcessor
Files: com.h, com.cpp, dbfile.h, dbfile.cpp

Figure 2.6: The structure of the program

```
class CoOccurrenceMatrix
{
  public:
    CoOccurrenceMatrix(void);   //construction
    CoOccurrenceMatrix( BMP_File* );  //construction
    ~CoOccurrenceMatrix(void);    //destroy

    bool        AddDatabaseEntry( FILE* );
    void        CalculateCOM( void );

  private:
    BMP_File*           m_BMPfile;
    float               m_COM[Ng][Ng]; //Co-Occurrence Matrix
    float               m_fEnergy;
    float               m_fEntropy;
    float               m_fContrast;
    float               m_fHomogeneity;
    float               m_fCorrelation;
    float               m_fVariance;
    float               m_fSumAver;
    float               m_fSumVari;
    float               m_fSumEntr;
    float               m_fDiffVari;
    float               m_fDiffEntr;
    float               m_fInfMeaCor1;
    float               m_fInfMeaCor2;

    void                CalculateCoocurrenceMatrix(void);
    void inline UpdatePixel( int, int, int, int );
    unsigned int inline  GetPixel(unsigned int, unsigned int);

    void                CalculateEnergy(void);
    void                CalculateEntropy(void);
    void                CalculateContrast(void);
    void                CalculateHomogeneity(void);
    void                CalculateCorrelation(void);
    void                CalculateVariance(void);
    void                CalculateSumAverage(void);
    void                CalculateSumVariance(void);
    void                CalculateSumEntropy(void);
    void                CalculateDiffVariance(void);
    void                CalculateDiffEntropy(void);
    void                CalculateInfoCorrelation(void);

};
```

Listing 2.1: The definition of class CoOccurrenceMatrix

```
 void CoOccurrenceMatrix::CalculateCoocurrenceMatrix(void)
{
    int x, y;
    int d=1; //distance
    for(y = 0; y < (int) m_BMPfile->m_iImageHeight;  y++ )
        for(x= 0; x <(int) m_BMPfile->m_iImageWidth;  x++ ){
            UpdatePixel( x, y, x-d, y-d );
            UpdatePixel( x, y, x, y-d );
            UpdatePixel( x, y, x+d, y-d );
            UpdatePixel( x, y, x-d, y );
            UpdatePixel( x, y, x+d, y );
            UpdatePixel( x, y, x-d, y+d );
            UpdatePixel( x, y, x, y+d );
            UpdatePixel( x, y, x+d, y+d );
        }
    //normalization
    for(int i=0;i<Ng;i++)
        for(int j=0;j<Ng;j++)
            m_COM[i][j] = m_COM[i][j]/normal;
 }
```

Listing 2.2: CalculateCoocurrenceMatrix() function

The function `UpdatePixel()`[Listing 2.3] reads every two pixels, which the distance between them in eight directions is `d`, from the image data and increase the value of corresponding elements of the matrix `m_COM`.

```
void inline CoOccurrenceMatrix::UpdatePixel(int x1,int y1,int x2,int
   y2)
{
//Make sure the neighbour pixel exists (can be e.g. negative):
    if( x2 < 0  ||  x2 >= (int) m_BMPfile->m_iImageWidth
        ||  y2 < 0||y2 >= (int) m_BMPfile->m_iImageHeight )
      return;
    unsigned int pixel, neighbour;
  pixel = m_BMPfile->m_ImageData[y1*m_BMPfile->m_iImageWidth + x1];
```

```
        neighbour=m_BMPfile->m_ImageData[y2*m_BMPfile->m_iImageWidth +x2];
      m_COM[pixel][neighbour] ++ ;
    }
```

Listing 2.3: UpdatePixel() and GetPixel() function

13 texture features are calculated through calling corresponding functions like `CalculateEnergy()`, `CalculateEntropy()`, `CalculateContrast()`... In these functions, based on Co-occurrence matrix `m_COM`, we update the value of texture features(`m_fEnergy`, `m_fEntropy`, `m_fContrast`...) following the Haralick's formula.

```
void CoOccurrenceMatrix::CalculateEnergy( void )
{
    unsigned int i, j;
    for( i = 0; i < Ng; i++ )
        for( j = 0; j < Ng; j++ )
            m_fEnergy +=  m_COM[i][j]* m_COM[i][j];
}

void CoOccurrenceMatrix::CalculateEntropy( void )
{
    unsigned int i, j;
    float tmp;
    for( i = 0; i < Ng; i++ ){
        for( j = 0; j < Ng; j++ ){
            tmp  =  m_COM[i][j];
            if( tmp != 0 )  //We should not take a log of 0
                tmp  =  tmp*log(tmp);
            m_fEntropy += tmp;
        }
    }
    m_fEntropy =  -m_fEntropy; // Negative
}
```

Listing 2.4: Function CalculateEnergy() and CalculateEntropy()

2.4.3 Measure Execution Time

It can be seen that calculating Co-occurrence matrix and Haralick texture features is a heavy computation. Particularly with large-size images, it requires thousands of operations; therefore it is a time-consuming process. We need to evaluate the performance by measuring the execution time to propose the necessary optimization. There are several techniques to measure execution time of a program or a part of a program in a Linux system. In [30], Stewart summarized and analyzed performance of eight measuring techniques: stop-watch, date, time, prof-gprof, clock(), software analyzers, timer/counter on-chips and logic/bus analyzers. The later technique is in the list, the more accurate it is and the more difficult it can be deployed. Among them, we choose three common measuring techniques, using processor time by clock() function, using calendar or date time and using on-chip counter. These methods can measure execution time of any piece of code.

Processor time by clock()

Processor time is the amount of time a computer program uses to process the program in

CPU. Processor time is different from actual wall clock time because it does not include any time spent waiting for I/O or when some other process is running. In Linux system, processor time is represented by the data type clock_t, and is given as a number of clock ticks relative to an arbitrary base time marking the beginning of a single program invocation. In typical usage, to measure execution, we call the clock function at the beginning and end of the interval we want to time, subtract the values, and then divide by the constant CLOCKS_PER_SEC (the number of clock ticks per second). However, this methods can only give the resolution in milliseconds, therefore it is not suitable if we want to measure the period in microseconds.

```
#include <time.h>
clock_t start, end;
double elapsed;
start = clock();
... /* Do the work. */
end = clock();
elapsed = ((double) (end - start)) / CLOCKS_PER_SEC;
```

Listing 2.5: Measure execution time using CPU time

Calendar time

Calendar time keeps track of dates and times according to the Gregorian calendar. In GNU C library ⟨*sys/time.h*⟩, the struct timeval structure represents the calendar time. It has the following members:

- `long int tv_sec`: this represents the number of seconds since the epoch. It is equivalent to a normal time_t value.
- `long int tv_usec`: this is the fractional second value, represented as the number of microseconds.

In typical usage, we call the function gettimeofday(`struct timeval *tp`, `struct timezone *tzp`) at the beginning and end of the interval we want to time, subtract the values. The `gettimeofday` function returns the current date and time in the struct timeval structure indicated by tp. Information about the time zone is returned in the structure pointed at `tzp`. If the `tzp` argument is a null pointer, time zone information is ignored.

```
#include <sys/time.h>
struct timeval  start, end ,elapsed;
gettimeofday(&start, NULL);
... /* Do the work. */
gettimeofday(&end, NULL);
if (start.tv_usec > end.tv_usec){
    end.tv_usec += 1000000;
    end.tv_sec--;
    }
elapsed.tv_usec = end.tv_usec - start.tv_usec;
elapsed.tv_sec  = end.tv_sec  - start.tv_sec;
```

Listing 2.6: : measure execution time using calendar time

This method the resolution in microseconds. However, by using the function `gettimeofday` we also add the time CPU is served for other processes by multithreading in the total elapsed time we received.

On-chip timer/counter

Most of computers have on-chip timer/counter chips can be used to obtain fine-grain, high-resolution measurements of code segments. In all x86 processors since Pentium, there is a counter called Time Stamp Counter. It is a 64-bit register which counts the number of ticks since reset. For some processor families, the time-stamp counter increments with every internal processor clock cycle. However, the CPU speed may change due to power-saving mode taken by the OS or BIOS, that makes time-stamp counter provides inaccurate results. Recently in all x86 Intel processors, the time-stamp counter increments at a constant rate. That rate may be set by the maximum core-clock to bus-clock ratio of the processor or may be set by the maximum resolved frequency at which the processor is booted.

In typical usage, to measure number of cycles, we load the current value of the processor's time-stamp counter into the EDX:EAX registers. The high-order 32 bits of the MSR are loaded into the EDX register, and the low-order 32 bits are loaded into the EAX register.

```
#if defined(__i386__)

static __inline__ unsigned long long rdtsc(void)
{
  unsigned long long int x;
     __asm__ volatile (".byte 0x0f, 0x31" : "=A" (x));
     return x;
}
#elif defined(__x86_64__)

static __inline__ unsigned long long rdtsc(void)
{
  unsigned hi, lo;
  __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
  return ( (unsigned long long)lo)|( ((unsigned long long)hi)<<32 );
}
```

Listing 2.7: Reading time stamp counter by function rdtsc()

```
#include "rdtsc.h"

unsigned long long start, end, elapsed ;
start = rdtsc();
... /* Do the work. */
end = rdtsc();
elapsed = end - start;
```

Listing 2.8: Measure execution time using time stamp counter

Return to the texture feature program, three above timing methods will be applied. To asset the performance, we need to measure only the time for calculating the Co-occurrence matrix and texture features, which means the execution time of the function `CalculateCOM()`. Therefore, we place function `clock()`, `gettimeofday` and `rtdsc()` before

Processor Frequency	2.33 GHz
L1 Cache	64 KB, 8-way cache associativity
L2 Cache	4 MB, 8-way cache associativity
RAM	8GB

Table 2.2: Specification of running environment: Intel Core 2 Duo E6550

and after we call `CalculateCOM()`. To increase the accuracy, we can repeat this part N times, and find the smallest value of execution time.

2.5 Results

In this part, we make following experiments:

- Comparing execution time of calculating the Co-occurrence matrix and texture features of the same image but with different sizes.
- Measuring the execution time for different gray-level (Ng= 32, 64, 128, 256, 512, 1024, 2048) for the same image size.
- The impact of the distance (d=1, 2, 3, 4, 5, 7, 9) on the execution time.

Experiments are performed on a desktop PC with processor Intel Core 2 Duo E6550 [Table 2.2]. All implementations are compiled by GCC compiler version 4.1.2. The tested image is the image ‘Wood’ as shown in Figure 2.7 The program is executed N= 1000 times and the shortest time is chosen to show.

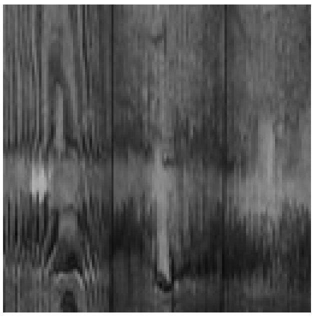


Figure 2.7: The image Wood

2.5.1 Comparing Execution Time of Different Sizes

For the first experiment, we test with the image ‘Wood’, the Table 2.3 shows the results for execution time of calculating Co-occurrence matrix and 13 texture features. The result shows that the calculating time of Co-occurrence matrix increases linearly with the size of images. When the size of the image increases 4 times, the calculating time of Co-occurrence matrix also grows up approximately 4 times. The calculating time of texture features does not vary much following the size of the image, it remains around 0.07 second. Therefore, when the size of the image is small, the calculating time of texture features is dominant. When the size of image increases, the calculating time of Co-occurrence matrix gets larger, and gradually become dominant.

It can be observed that the time measured by three methods is approximately identical. However, the resolution of CPU time is not enough to measure period of microseconds. We can choose to use either Calendar time or Time-stamp counter. In the other experiments, only Time-stamp counter is used.

Size	Time (CPU)(s)			Time (Calendar)(s)			Time (TS counter)(s)		
	GLCM	texture features	total	GLCM	texture features	total	GLCM	texture features	total
128	0.00	0.07	0.07	0.0021	0.0705	0.0726	0.0021	0.0705	0.0727
256	0.00	0.07	0.07	0.0071	0.0739	0.0811	0.0071	0.0739	0.0811
512	0.01	0.07	0.08	0.0162	0.0730	0.0892	0.0162	0.0729	0.0892
1024	0.07	0.07	0.14	0.0646	0.0710	0.1357	0.0646	0.0710	0.1357
2048	0.25	0.07	0.32	0.2545	0.0715	0.3261	0.2545	0.0715	0.3261
4096	0.99	0.07	1.06	0.9986	0.0703	1.0689	0.9986	0.0703	1.0689

Table 2.3: Execution time of the image Stone in different sizes

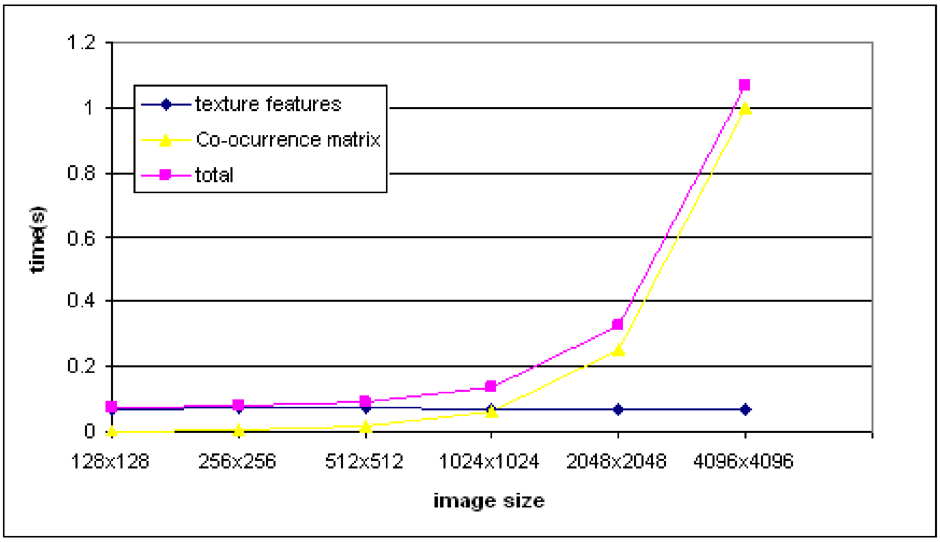


Figure 2.8: Graph of execution time of the image in different sizes with gray-level=256, d=1

2.5.2 Execution Time for Different Gray-levels

In this experiment, we test with different gray-levels (Ng= 32, 64, 128, 256, 512, 1024, 2048) of the an 2048 × 2048 image. The Table 2.4 shows the execution time.

Gray level	Time(s)		
	Co-occurrence matrix	texture features	total
32	0.213305	0.000423	0.213728
64	0.216330	0.002085	0.218415
128	0.230164	0.011255	0.241419
256	0.248919	0.072222	0.321141
512	0.294054	0.499942	0.793996
1024	1.073311	3.778518	4.851829
2048	1.500172	28.923663	30.423835

Table 2.4: Execution time with different gray-levels, image size 2048 × 2048, d=1

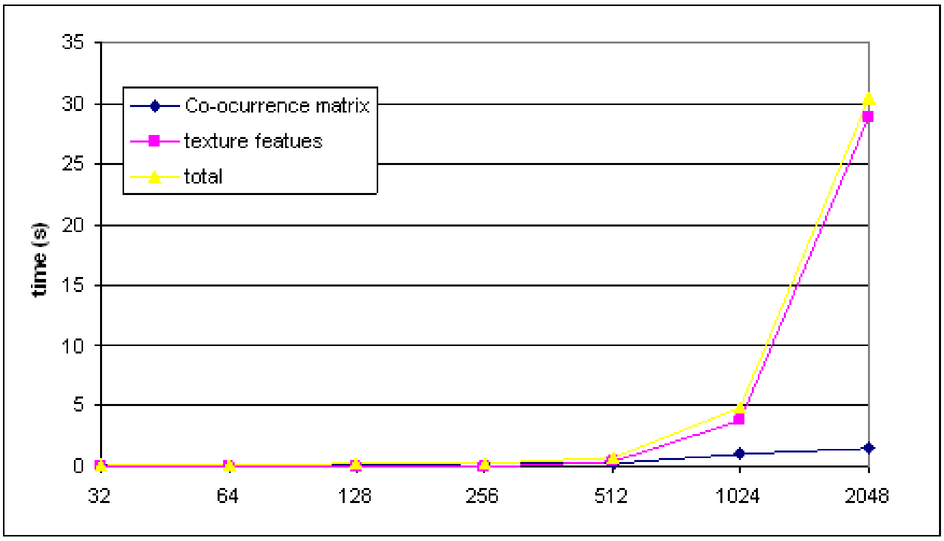


Figure 2.9: Graph of execution time with different gray-levels, image size =2048 × 2048, d=1

It can be seen that when the gray-level changes, the calculation time of Co-occurrence matrix increases marginally, meanwhile the calculating time of texture features increases exponentially (around 5-7 times). Especially, when the grey-level Ng = 2048, the calculating time of texture features become very large, 30 second.

2.5.3 The Impact of the Distance d on the Execution Time

In all previous experiments, the Co-occurrence matrix is calculated with the distance of a reference pixel to the neighbor is one (d = 1). This experiment, the dependence of execution time to this distance is evaluated. The table 2.5 shows the execution time of Co-occurrence matrix with different distances. The image is of the size 1024 × 1024. It can be seen from the Table 2.5 that the calculating time of Co-occurrence matrix still remains though the distance d changes.

Distance	Execution Time (s)		
	co-occurrence matrix	texture features	total
1	0.061896	0.073858	0.135755
2	0.060775	0.069951	0.130727
3	0.065929	0.072502	0.138432
4	0.065256	0.069930	0.135186
5	0.065143	0.069054	0.134198
6	0.060953	0.074253	0.135206
7	0.060876	0.070026	0.130903
8	0.060769	0.070782	0.131552
9	0.060589	0.069544	0.130134

Table 2.5: execution time with different distances d, image size 1024 × 1024, gray level 256

2.5.4 Conclusion

From the experiments, we have demonstrated that calculation process of co-occurrence matrix and texture features is time-consuming, especially when the size and the gray-level of the image is large. For example, for an image of size 4096 × 4096 and gray-level 256, it takes around 1 second for a Core2Duo machine to compute. Imagine that we have to process hundreds of images, the total will be considerably large. Therefore, optimization of this calculating process to reduce its execution time is necessary. The optimization is investigated in Chapter 3 and Chapter 4.

Software Optimization of Texture Features

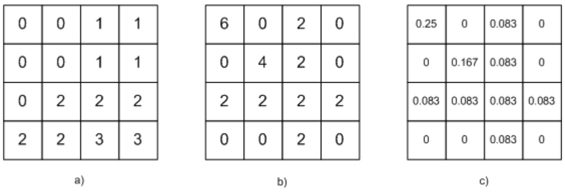
3

From results of previous chapter, it can be seen that the overall calculations for the computation of GLCM and texture features are computationally intensive and time-consuming. There are number of techniques to accelerate the computation of GLCM and texture features. The acceleration can be obtained by reducing the size of the GLCM, which means that the image data is quantized from eight bits or higher down to as few as four or five bits. However, quantization has the potential to remove pertinent information from the image. The drawback of using GLCM is that GLCM is a sparse matrix whose many elements are zero. These zero elements are unnecessary for calculating texture features. Clausi proposed to store just non-zero values of the GLCM in a linked list (Gray Level Co-occurrence Link List, GLCLL) [4] [3] or an improved structure, linked list with hash table (Gray Level Co-occurrence Hybrid Structure, GLCHS)[5]. This chapter will describe two types of optimizations: optimization in co-occurrence matrix (GLCLL, GLCHS) and optimization in texture features (feature combinations and loop unrolling).

3.1 Optimization in Co-occurrence Matrix

3.1.1 Gray Level Co-occurrence Link List

We begin this section by considering an example. Figure 3.1a is a 4x4 gray-scale image, Figure 3.1b is the co-occurrence matrix of the image following vertical direction($\theta = 90$ and $\theta = 270$) with distance $d = 1$. Figure 3.1c is the co-occurrence matrix after normalization.



0	0	1	1
0	0	1	1
0	2	2	2
2	2	3	3

a)

6	0	2	0
0	4	2	0
2	2	2	2
0	0	2	0

b)

0.25	0	0.083	0
0	0.167	0.083	0
0.083	0.083	0.083	0.083
0	0	0.083	0

c)

Figure 3.1: 4x4 gray-scale image(a); Co-occurrence matrix(b); Normalized Co-occurrence matrix (c)

We can see that the GLCM is quite sparse, it has 7 elements of zero (in total of 16 elements). With GLCM, because the loops span all elements, we have to read these zero ones, which is waste of computation time. To overcome this, instead of using a matrix to store the co-occurrence probabilities, Clausi proposed to use a linked list structure to store only non-zero co-occurrence probabilities. The linked lists are set up in the following manner. Each linked-list node is a structure containing the two co-occurring gray levels (i, j), their probability of co-occurrence, and a link to the next node on the list. In order to allow rapid searching for existing (i,j) pairs, the list must be kept sorted, based on indexes provided by the gray-level pairs. An example of such a sorted list for the image in Figure 3.1 would be $(0,0); (0,2); (1,1); (1,2); (2,0); (2,1); (2,2); (2,3); (3,2)$. To include a new gray-level pair in a linked list, a search is performed. Searching begins at the head of the list by looking for the first instance of the i^{th} gray level. If it is found, the algorithm searches for the corresponding j^{th} gray level. If the (i,j) pair is found, then the probability stored inside that node is incremented. If the (i,j) node is not found at the expected location, then a node must be entered at that location that stores the appropriate probability for that gray level pair. With this linked-list structure, to search for an existing (i,j) pair, even with a sorted linked-list, we have to start at the head of the list. This process takes time, normally requires O(N) time (N is the number of elements of co-occurrence matrix), which is the primary disadvantage of linked-list.

```
typedef struct LNode
{
    unsigned int i,j; //co-occurence gray level pairs
  float p;          //co-occurence probability
  struct LNode* next; //pointer to the next node
} ListNode;
```

Listing 3.1: The definition for node in the linked list

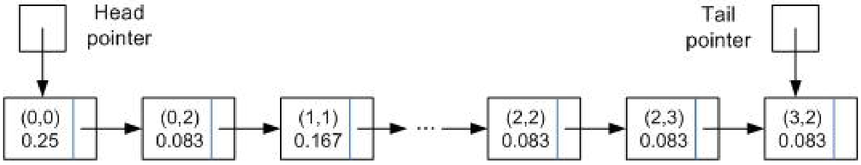


Figure 3.2: GLCLL structure for the image in Figure 3.1

To gain better searching time, the linked-list structure can be modified to array of linked-list structure as shown in Figure 3.3. This structure contains an array of N(gray level) elements, in which each element is a linked-list. Each linked-list corresponds to a row in the co-occurrence matrix. Only non-zero elements of the row are stored in the linked-list. This structure has two advantages over the normal linked-list: 1. The searching time is faster, only O(logN). 2. It uses less memory space, because in each linked-list node, only the column index of co-occurrence matrix is needed to store, while both row and column index of co-occurrence matrix are stored in normal linked-list.

```
typedef struct GList {
        GNode* pHead;
        GNode* pTail;
} GrayList;
GList list[Ng];
```

Listing 3.2: The definition for array of linked-list structure

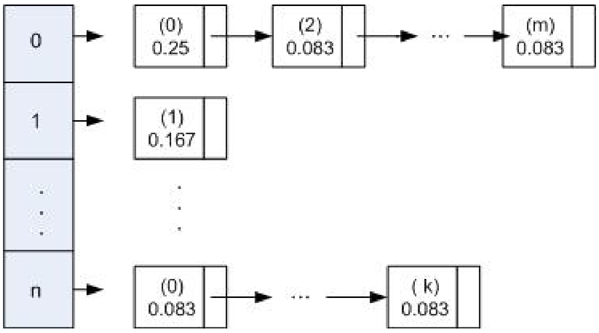


Figure 3.3: Array of linked-list structure

3.1.2 Gray Level Co-occurrence Hybrid Structure

To overcome drawback of linked-list structure, a common approach is to ”index” a linked list using a more efficient external data structure, such as hash table. Based on this combination, Clausi proposed to use the hash table and called the new structure as the gray level co-occurrence hybrid structure (GLCHS). Using the GLCHS, a two-dimensional hash table structure is created to point to the co-occurrence linked list nodes. The hash table allows for rapid access of any node in the linked list, if that node exists. The linked list allows for rapid computation of texture features by traversing the linked list from head to tail. The definition of a node in the linked list still remains, the definition for node in hash table is as in Listing 3.1.2. In the hash node structure, a char member k decides whether or not the linked node exists. $*list\_p$ is a linked list pointer, which points to the corresponding node in the linked list associated by the gray level pair.

```
typedef struct HNode  // hash node
{
  char k;
  struct ListNode* list_p; //pointer to the list node
} HashNode;

HNode HTable[Ng][Ng]; //hash table
```

Listing 3.3: The definition of has table and hash node

The creation of the hybrid data structure requires the following steps. First, a hash table is created as a two-dimension array. Each element of the array is a hash node, which is initialized (k set to zero and pointer set to NULL). Finally, the head and tail are initialized to NULL values to represent an empty doubly linked list. For a given gray level pair, if the value of k of the hash node is zero, then that particular cooccurring pair does not have a representative node on the linked list. As a result, a new ListNode is created, its gray level values are set, and it is inserted at the end of the linked list. The `listp` is then set to point to this ListNode to establish the relationship between the HashNode and its corresponding ListNode. If the hash table entry is not zero, then that HashNode already points to an existing ListNode on the linked list. Whether or not the ListNode was created, the probability associated with LinkNode is incremented by the given probability. As a result, the linked list does not have to be kept sorted, and any list node can be accessed rapidly without searching the list. This design will reduce significantly and consistently the completion times when determining co-occurrence probability in comparison with normal linked-list structure.

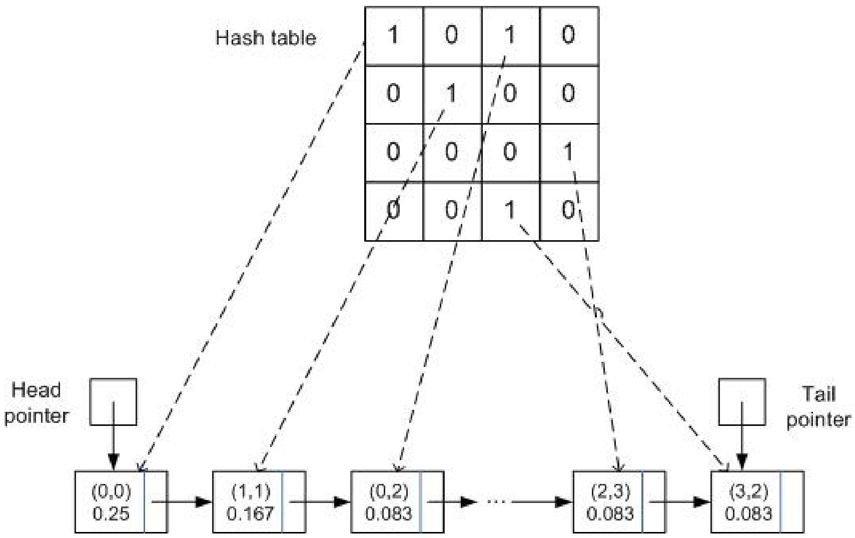


Figure 3.4: Hash table - linked list structure

This hash table - linked list structure can be improved by using array instead of linked list. Each element of the hash table points to a corresponding element of the co-occurrence array. This structure has three advantages over the hash table - linked list: 1. The building time of co-occurrence array is faster than co-occurrence linked list. 2. It does not waste the memory space to store the pointer to the next element as in the linked list. 3. The memory address of array elements are continuous, while each node of co-occurrence linked list can be in different location of memory, therefore, the array will increase cache performance, reduce the traversing time when calculating texture features.

```
typedef struct CoNode {  // element of co-occurrence matrix
    int i;
  int j;
    float number;
} CooNode;

typedef struct HNode {  // hash node
  char k;
  struct CoNode* pNode;
} HashNode;

CooNode   p[Ng*Ng];  // array of co-occurrence matrix
HashNode  HTable[Ng][Ng]; //hash table
```

Listing 3.4: The definition of components in hash table - array structure

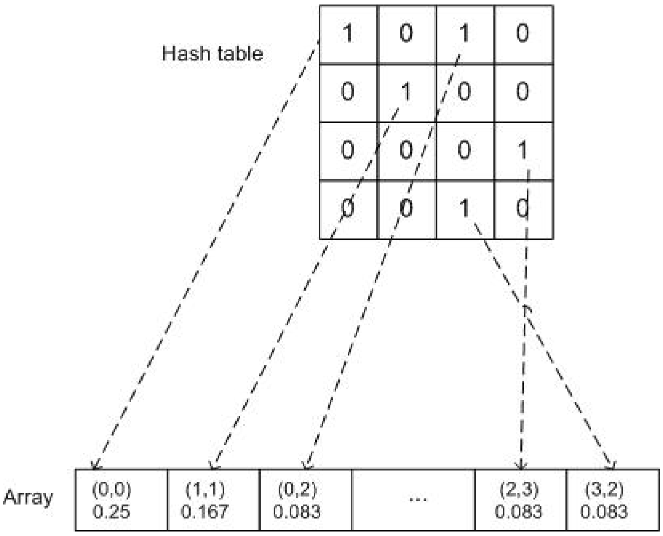


Figure 3.5: Hash table - array structure

3.2 Optimization in Texture Features

It can be seen that all the equations of texture features and statistical properties of Co-occurrence matrix contain the loop range over all elements of co-occurrence matrix $\sum_{i=0}^{N_g-1}\sum_{j=0}^{N_g-1}$ or span all elements of a row or a column $\sum_{i=0}^{N_g-1}$. Eizan Miyamoto [24] proposed that the calculation of the features that loop across the data in similar ways can be combined. We begin by combining energy f_1, contrast f_2, entropy f_3, homogeneity f_6, p_{x+y}, p_{x-y}, $p_x(i)$, $p_y(i)$, mean of $p_x(i)$, $p_y(i)$. The loops in each of these features range over all elements of co-occurrence matrix and they can be calculated directly through one double loop $\sum_{i=0}^{N_g-1}\sum_{j=0}^{N_g-1}$.

After we have above features, we can calculate variance f_4 and standard deviations of p_x, p_y through one double loop $\sum_{i=0}^{N_g-1}\sum_{j=0}^{N_g-1}$ and sum average f_7, sum entropy f_9, different entropy f_{11}, different average through one single loop $\sum_{i=0}^{N_g-1}$. Because sum average f_7 and sum entropy f_9 from 0 to $2(N_g-1)$, we have to unroll them so that they can be calculated in the loop from 0 to (N_g-1). Similarly, for the third loop, after we have $p_x(i)$, $p_y(i)$, mean and standard deviations of $p_x(i)$, $p_y(i)$, sum average, different average we can compute correlation f_5 and HXY1, HXY2 in one double loop $\sum_{i=0}^{N_g-1}\sum_{j=0}^{N_g-1}$, sum variance f_8 and different variance f_{10} in one single loop $\sum_{i=0}^{N_g-1}$. Once again, because sum variance f_8 is computed from 0 to $2(N_g-1)$, it must be unrolled to the loop from 0 to (N_g-1).

By this way, instead of one loop for each features, we have only three double loops, that will reduce computing time considerably. Listing 3.1, 3.2 and 3.3 show the implementing code of three double loops of concatenation.

```
for( i = 0; i < Ng; i++ ){
    for( j = 0; j < Ng; j++ ){
        m_fEnergy +=  m_COM[i][j]* m_COM[i][j];        //energy
        m_fContrast  += ((i-j)*(i-j))*m_COM[i][j];     //contrast
        m_fHomogeneity += m_COM[i][j]/(1+(i-j)*(i-j)); //homogeneity
        if( m_COM[i][j] != 0 )
            m_fEntropy +=  m_COM[i][j]*log(m_COM[i][j]);//entropy
        ux= ux + i*m_COM[i][j];                      //mean of p_x
        uy= uy + j*m_COM[i][j];                      //mean of p_y
        pxy[i+j] +=  m_COM[i][j];                    //p_{x+y}(k)
        if (i>=j) pdxy[i-j] +=  m_COM[i][j];         //p_{x-y}(k)
        else pdxy[j-i] +=  m_COM[i][j];
        px[i] += m_COM[i][j];                      //p_x
  }
}
```

Listing 3.5: The first double loop

```
for( i = 0; i < Ng; i++ ){
    for( j = 0; j < Ng; j++ ){
        stdDevix += ((i-ux)*(i-ux)*m_COM[i][j]);
        stdDeviy += ((j-uy)*(j-uy)*m_COM[i][j]);
    m_fVariance += (i-ux)*(i-ux)*m_COM[i][j]; // variance
    }
    if( px[i] != 0 ) hx = hx + px[i]*log(px[i]);
    DiffAver = DiffAver + i*pdxy[i];
    m_fSumAver += (2*i)*pxy[2*i] ;            //sum average
    m_fSumAver += (2*i+1)*pxy[2*i+1] ;
    if( pxy[2*i]!= 0 )
        m_fSumEntr += pxy[2*i]*log(pxy[2*i]); //sum entropy
    if( pxy[2*i+1]!= 0 )
        m_fSumEntr += pxy[2*i+1]*log(pxy[2*i+1]);
    if( pdxy[i] != 0 )
        m_fDiffEntr += pdxy[i]*log(pdxy[i]); //different entropy
}
```

Listing 3.6: The second double loop

```
for( i = 0; i < Ng; i++ ){
   for( j = 0; j < Ng; j++){
      m_fCorrelation+=((i-ux)*(j-ux)*m_COM[i][j])/(stdDevix*stdDevix);
      if((px[i]!= 0)&&(px[j]!= 0)){
          hxy1 += m_COM[i][j]*log(px[i]*px[j]);
          hxy2 += px[i]*px[j]*log(px[i]*px[j]);
      }
   }
   m_fSumVari +=(2*i-m_fSumAver)*(2*i-m_fSumAver)*pxy[2*i];
   m_fSumVari +=(2*i+1-m_fSumAver)*(2*i+1-m_fSumAver)*pxy[2*i+1];
   m_fDiffVari+=(i-DiffAver)*(i-DiffAver)*pdxy[i];//different variance
}

hxy1 = -hxy1;
hxy2 = -hxy2;
m_fInfMeaCor1 = (m_fEntropy-hxy1)/hx;               // InfMeaCor1
m_fInfMeaCor2 = sqrt(1- exp(-2*(hxy2-m_fEntropy))); // InfMeaCor2
```

Listing 3.7: The third double loop

Image sizes	Array	Linked List	Array-Linked List	Hash Table-Linked List	Hash Table-Array
128 × 128	2.20	10699.87	44.88	6.46	3.74
256 × 256	7.02	67169.19	221.04	22.24	13.34
512 × 512	26.01	375441.36	985.98	73.71	48.22
1024 × 1024	105.87	1765467.32	4065.48	277.45	191.58
2048 × 2048	382.49	6913583.91	15457.35	992.20	700.78
4096 × 4096	1457.80	21815411.62	54725.97	3549.54	2510.43

Table 3.1: Building time of co-occurrence matrix of 5 structures, gray level Ng =256 ($\times 10^6$ cycles)

3.3 Testing and Result

Tests are performed on a desktop PC with processor Intel Core 2 Duo E6550 [Table 2.2]. All implementations are compiled by GCC compiler version 4.1.2 with optimization flag $'-O3'$ and $'funroll-loops'$. The tested image is image 'Stone' as shown in Figure 3.6. Execution time is measured by time stamp counter. The program is executed N= 1000 times and the shortest time is chosen to show.

3.3.1 Optimization Co-occurrence with Different Structures

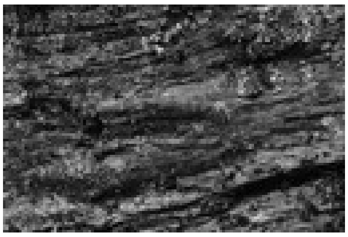


Figure 3.6: The tested image 'Stone'

Six image sizes (128×128, 256×256, 512×512, 1024×1024, 2048×2048, 4096×4096) are used as parameters. The co-occurrence probabilities are calculated with distance between pixels d=1 and in eight directions (θ= 0, 45, 90, 135, 180, 235, 270, 315). The computation time of co-occurrence probabilities and 13 texture features are compared between three scenarios, the original version and two optimization approaches.

Table 3.1 shows the building time of co-occurrence matrix of five structures, with different image sizes, gray level =256. It can be seen that for building co-occurrence matrix, using array structure is the fastest[Figure 3.7]. Building co-occurrence matrix with array structure has complex function of O(1), because we can access directly any element of the matrix to update its value. Building co-occurrence matrix with linked list has complex function of O(n), because each time we update the value of an element we have to search it from the head of the linked-list, even the linked list is sorted. This makes the building time of linked list structure much larger than of the array structure. Array of linked list structure has complex function of O(log n), therefore it is faster

Image sizes	Array	Linked List	Array-Linked List	Hash Table-Linked List	Hash Table-Array
128 × 128	168.62	16.22	17.74	16.03	15.97
256 × 256	174.16	23.92	23.97	23.72	23.77
512 × 512	178.42	31.89	32.51	31.39	30.99
1024 × 1024	182.20	35.63	36.84	35.33	35.14
2048 × 2048	179.09	35.57	36.90	35.27	35.00
4096 × 4096	180.48	30.78	32.03	30.58	30.41

Table 3.2: Calculating time of texture features of 5 structures, gray level Ng =256 ($\times 10^6$ cycles)

Image sizes	Array	Linked List	Array-Linked List	Hash Table-Linked List	Hash Table-Array
128 × 128	170.81	10716.09	62.62	22.49	19.70
256 × 256	181.18	67193.11	245.01	45.97	37.11
512 × 512	204.43	375473.25	1018.48	105.10	79.21
1024 × 1024	288.07	1765502.95	4102.32	312.78	226.72
2048 × 2048	561.57	6913619.48	15494.25	1027.47	735.79
4096 × 4096	1638.27	21815442.40	54758.00	3580.11	2540.84

Table 3.3: Total execution time of 5 structures, gray level Ng =256 ($\times 10^6$ cycles)

than the linked list structure but still much slower than the array structure. Using hash table, either combination with linked list or array, has the same complex function O(1) with array structure. However, each update, it takes time to create a new element or increment an element value, therefore the building time is larger.

Because of too large execution time of array structure, for better display of graphs, it will not be presented.

If we compare the calculation time of texture features[Figure 3.7], we see that all other structures is much faster than array structure. The speed-up achieved is between 5 and 10 times. This rate depends on the image. If the image has a sparse co-occurrence matrix, the speed-up increases. When we consider the total execution time[Figure 3.7] including both time of building co-occurrence matrix and time of calculating texture features, we can see that for small images which texture features time is dominant, other structures give the better result than array structure, for larger images which co-occurrence matrix time is dominant, array structure is better.

3.3.2 Optimization Texture Teatures

Tables 3.4 and 3.5 show calculating time of texture features and the total execution time of five structures after texture feature optimization. Comparing with previous, we see that for array structure, the calculation time of texture features reduce significantly, about 6-9 times [Figure 3.8]. For hash table structure, the calculation time of texture features decreases only a small value, about 7%-20%. And if we compare the calculation

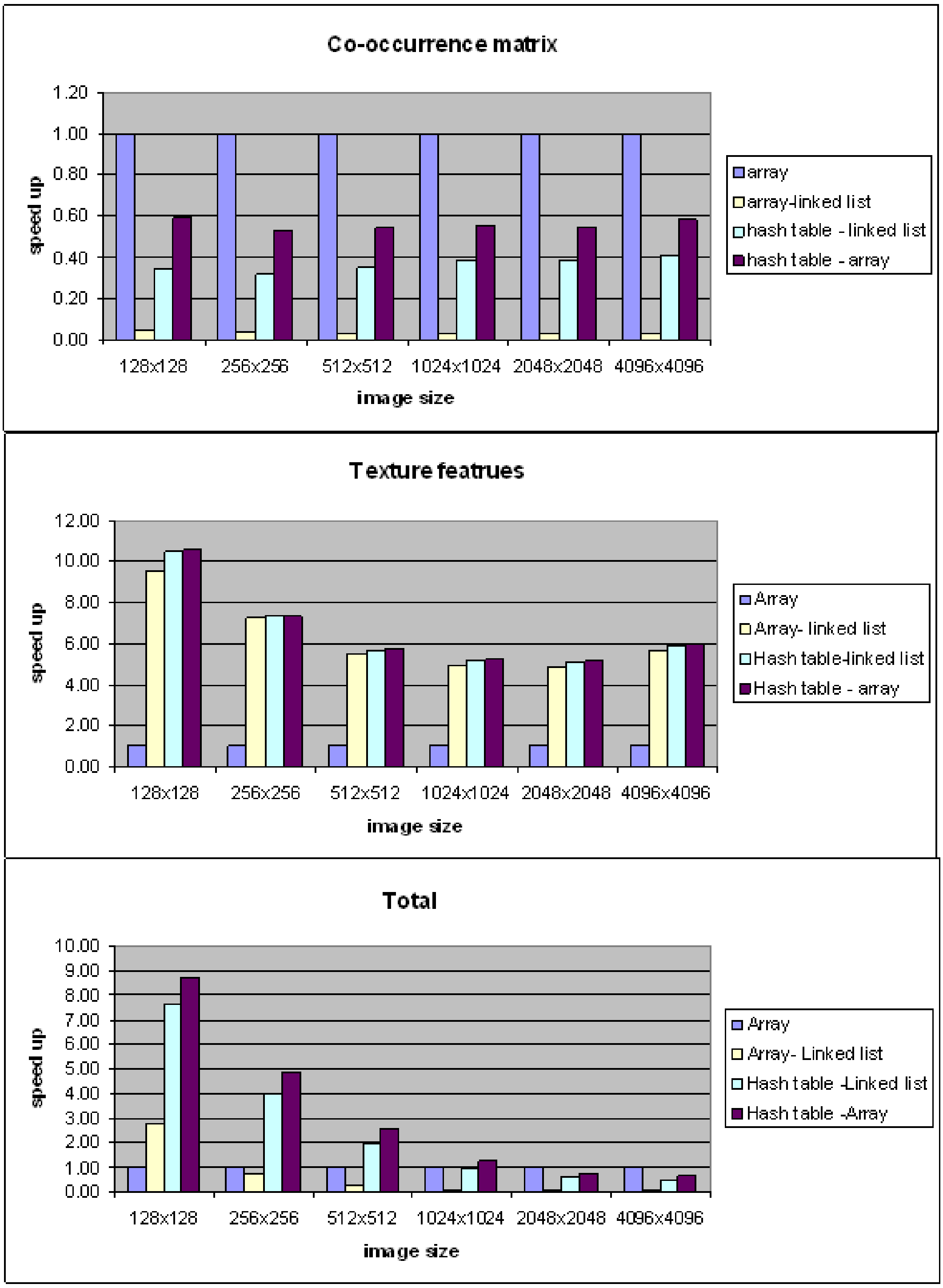


Figure 3.7: Speed up of 3 structures with different images sizes (in comparison with array structure)

Image sizes	Array	Linked List	Array-Linked List	Hash Table-Linked List	Hash table-Array
128 × 128	19.32	15.33	14.69	15.03	14.90
256 × 256	25.94	22.60	21.45	22.10	21.81
512 × 512	31.05	29.04	27.69	28.84	28.55
1024 × 1024	33.72	34.51	31.36	32.71	32.32
2048 × 2048	33.45	34.80	31.22	32.60	32.17
4096 × 4096	31.73	28.91	27.46	28.51	28.18

Table 3.4: Calculating time of texture features of 5 structures after texture feature optimization, gray level Ng =256 ($\times 10^6$ cycles)

Image sizes	Array	Linked List	Array-Linked List	Hash Table-Linked List	Hash Table-Array
128 × 128	21.51	10715.21	59.57	21.49	18.64
256 × 256	32.96	67191.78	242.49	44.34	35.15
512 × 512	57.05	375470.40	1013.67	102.55	76.77
1024 × 1024	139.59	1765501.83	4096.84	310.17	223.91
2048 × 2048	415.94	6913618.70	15488.58	1024.79	732.95
4096 × 4096	1489.52	21815440.53	54753.42	3578.05	2538.61

Table 3.5: Total execution time of 5 structures after texture feature optimization, gray level Ng =256 ($\times 10^6$ cycles)

time of texture features of two structure after optimization, the difference is small, from 6% to 21%, depending on each images.

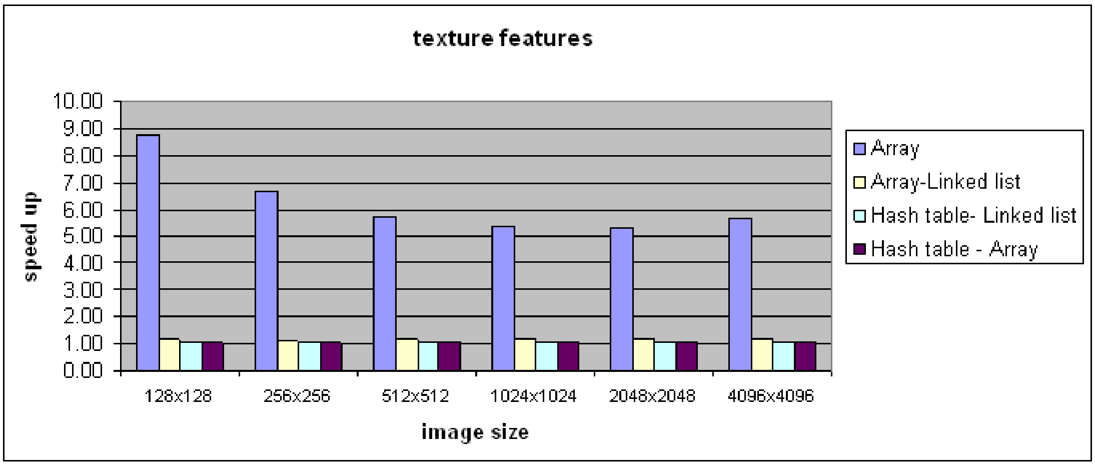


Figure 3.8: Speed up in calculating texture features of 4 structures after optimization in texture features (in comparison with normal implementation)

If we compare the calculation time of texture features between four structures after optimization, we can realize that other structures are faster than array structure, however

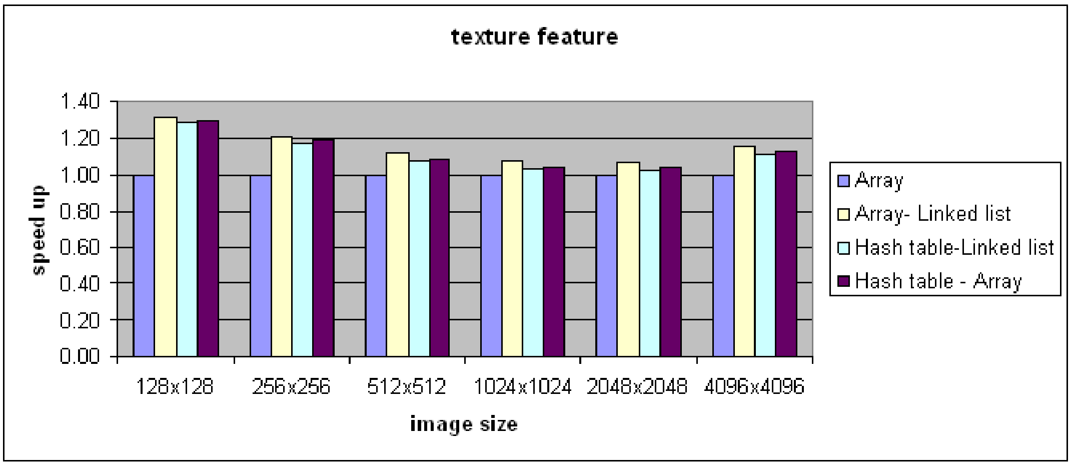


Figure 3.9: Speed up in calculating texture features of 3 structures in comparison with array structure after optimization

the speed-up is not as high as before optimization [Figure 3.9]. This speed-up value depends on how parsed the co-occurrence matrix is. If the matrix is parsed (contains many zero elements), the speed-up value is high. We check these properties by comparing calculation time of texture features of images with different number of non-zero elements in their co-occurrence matrix [Table 3.6]. It can be seen from the Figure 3.10 that the speed-up value reduces when the non-zero elements increase.

non-zero elements	array (10^6 cycles)	hash table-linked list (10^6 cycles)	speed up (times)
574	3.33	0.70	4.77
10083	15.75	9.19	1.71
18538	19.29	15.00	1.29
33411	28.19	24.45	1.15

Table 3.6: Calculation time of texture features of 2 structures with different non-zero elements, image size =128x128 gray level Ng =256

3.4 Conclusions

After we concatenate the features, we can reduce the computation time significantly. To compare between different structure approaches of co-occurrence matrix, array structure gives fastest building time of co-occurrence matrix, while other structures have smaller calculation time of texture features. Structures like hash table, array of linked list should be used when the computing time of texture features is dominant and when building of co-occurrence matrix is dominant(the image with large size), array structure should be used.

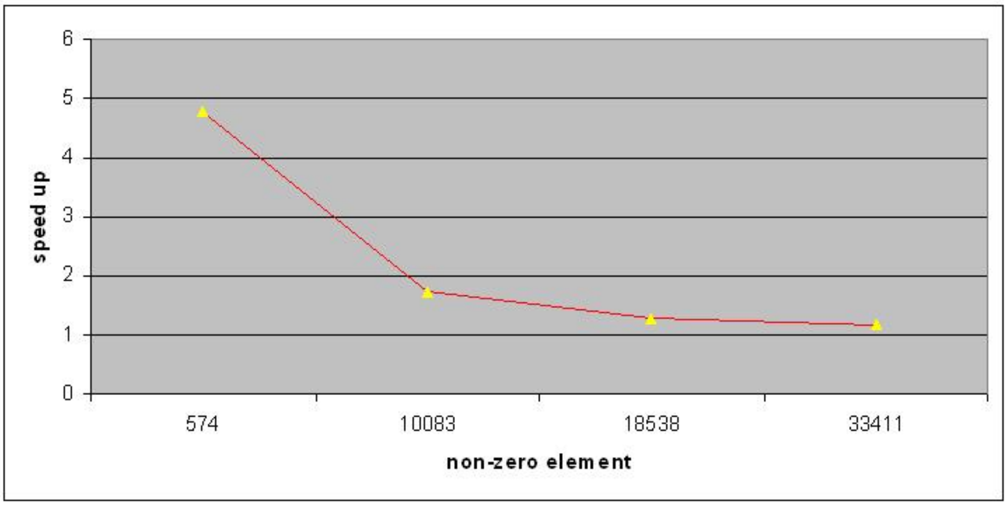


Figure 3.10: Speed up with different non-zero elements, hash table - linked list structure in comparison with array structure, image size 128x128, gray level =256

4

Parallel Implementation of Texture Feature Extraction

With the emerging of parallel processing, there are several researches about computing GLCM and texture features in parallel. Khalaf et al [27] proposed a hardware architecture following odd-even network topology to parallelize computing GLCM. Markus Gipp et al [19] accelerated the computation of Haralicks texture features using Graphics Processing Units (GPUs). Tahir [18] presented an FPGA based coprocessor for GLCM and texture features and their application in prostate cancer classification. Results from these researches demonstrated that parallel processing could provide significant increase in speed for GLCM and texture feature computations. In this chapter, the parallel implementations of GLCM and texture features will be discussed using Cell Broadband Engine Processor.

4.1 Overview of the Cell Broadband Engine

The Cell Broadband Engine (Cell BE) Architecture [15] is a heterogeneous multi-core architecture that extends the 64-bit PowerPC Architecture. The Cell BE processor is the result of the collaboration between Sony, Toshiba and IBM known as STI, formally begun in early 2001. Although the Cell BE processor is initially intended for applications in media-rich consumer-electronics devices such as game consoles and high-definition televisions, the architecture has been designed to enable fundamental advances in processor performance. These advances are expected to support a broad range of applications in both commercial and scientific fields.

The Cell Broadband Engine is a single-chip multiprocessor 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. It is fully compliant with the 64-bit PowerPC architecture and can run 32-bit and 64-bit operating systems and applications. The second type of processor element, the SPE, is optimized for running computation-intensive SIMD applications, and it is not optimized for running an operating system. 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. Figure 4.1 shows a block diagram of the Cell Broadband Engine. In this diagram, SXU is Synergistic Execution Unit, LS is 256KB of Local Store, DMA is Direct Memory Access controller that supports DMA transfers.

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,

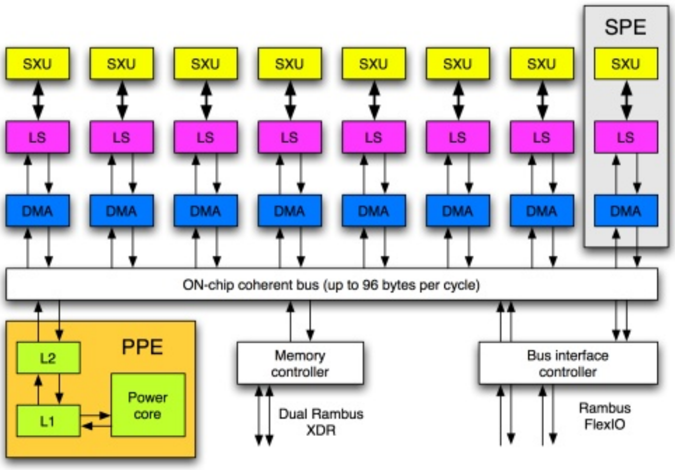


Figure 4.1: Block diagram of the Cell Broadband Engine architecture.

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 between main storage and their local stores. A DMA operation can transfer either a single block area of size up to 16KB, 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 EIBs internal bandwidth is 96 bytes per cycle with a peak bandwidth of 204.8 GB/s) [35], 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 provides peak bandwidths of 25 Gbytes/s inbound and 35 Gbytes/s outbound can deliver a sustained bandwidth of 25.6 GB/s.

What makes the Cell BE such a high-performance processor is that it overcomes three performance-limiting walls: power use, memory use and processor frequency [15].

Scaling the power-limitation wall

One aspect in improving the performance of microprocessors is to improve power efficiency at about the same rate as the performance increase. One way to increase power efficiency is to differentiate between: processors optimized to run an operating system and control-intensive code, and processors optimized to run computation-intensive applications. The Cell Broadband Engine does this by providing a general-purpose PPE to run the operating system and other control-plane code, and eight SPEs specialized for computing data-rich (data-plane) applications.

Scaling the memory-limitation wall

In traditional homogeneous single- and multi-core processors, large data is normally stored in DRAM memory if it is not fit in cache memory. As a result, the program performance is declined due to the activity of moving data between main storage and the processor. The Cell BEs SPEs use two mechanisms to deal with long main-memory latencies:a 3-level memory structure (main storage, local stores in each SPE, and large register files in each SPE), asynchronous DMA transfers between main storage and local stores. The Cell BE can support 128 simultaneous transfers between the eight SPE local stores and main storage.

Scaling the frequency-limitation wall

Conventional processors require increasingly deeper instruction pipelines to achieve higher operating frequencies. This technique has reached a point of diminishing returns. By specializing the PPE and the SPEs for control and compute-intensive tasks, respectively, the Cell Broadband Engine Architecture, on which the Cell BE is based, allows both the PPE and the SPEs to be designed for high frequency without excessive overhead. The PPE achieves efficiency primarily by executing two threads simultaneously rather than by optimizing single-thread performance. Each SPE achieves efficiency by using a large register file, which supports many simultaneous in-process instructions without the overhead of register-renaming or out-of-order processing. Each SPE also achieves efficiency by using asynchronous DMA transfers, which support many concurrent memory operations without the overhead of speculation.

The Cell BE is designed to be programmed in high-level languages, such as (but certainly not limited to) C/C++. The instruction set for the PPE is an extended version of the PowerPC Architecture instruction set. The extensions consist of the vector/SIMD multimedia extensions, a few additions and changes to PowerPC Architecture instructions, and C/C++ intrinsics for the vector/SIMD multimedia extensions. The instruction set for the SPEs is a new SIMD instruction set, the Synergistic Processor Unit Instruction Set Architecture, with accompanying C/C++ intrinsics, and a unique set of commands for managing DMA transfer, external events, interprocessor messaging, and other functions. The instruction set for the SPEs is similar to that of the PPEs vector/SIMD multimedia extensions, in the sense that they operate on SIMD vectors. However, the two vector instruction sets are distinct, and programs for the PPE and SPEs are often compiled by different compilers.

4.2 The Cell Instruction Set Architecture

The Cell has two types of processor element: the PPE and the SPE, which use two different instruction sets.

4.2.1 The PPE Instruction Set

The PPE uses both the PowerPC instruction set and Vector/SIMD Multimedia Extension instruction set.The PowerPC instruction set uses instructions that are 4 bytes long and word-aligned. It supports byte, halfword, word, and doubleword operand accesses between storage and its 32 general-purpose registers (GPRs). The instruction set also supports word and doubleword operand accesses between storage and a set of 32 floating-point registers (FPRs). Signed integers are represented in two-complement form.

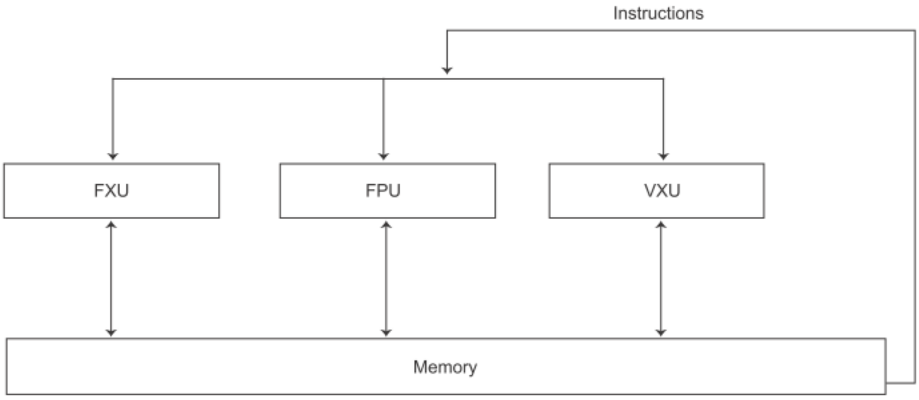


Figure 4.2: Concurrent execution of integer, floating-point, and vector units

The Vector/SIMD Multimedia Extension instruction set uses instructions that, like PowerPC instructions, are 4 bytes long and word-aligned. However, all of its operands are 128 bits wide. Most of the Vector/SIMD Multimedia Extension operands are vectors, including single-precision floating-point, integer, scalar, and fixed-point of vector-element sizes of 8,16, and 32 bits. Vector/SIMD Multimedia Extension instructions are executed in 128-bit Vector/SIMD Multimedia Extension unit (VXU), which operates concurrently with the PPUs fixed-point integer unit (FXU) and floating-point execution unit (FPU) [Figure 4.2].

The Vector/SIMD Multimedia Extension model uses a set of fundamental data types, called vector types. An vector variable which is stored in a 128-bit register can be:

- Sixteen 8-bit values, signed or unsigned.
- Eight 16-bit values, signed or unsigned.
- Four 32-bit values, signed or unsigned.
- Four single-precision IEEE-754 floating-point values.

The vector types use the prefix `vector` in front of one of standard C data typesfor example `vector signed int` and `vector char`. Through a set of of C-language extensions are available for Vector/SIMD Multimedia Extension programming, it is easily to manipulate with vector types. To illustrate SIMD programming, we consider an example of array-summing. The code in Listing 4.1 contains two versions of a function that sums an input array of N unsigned integer values. The first version performs the scalar operations with N iterations of the main loop. The second one uses vector commands, which reduces the number of iterations 4 times. Figure 4.3 explains SIMD operations to get the sum. Since the data type of the input array is `unsigned int`, SIMD operations can handle four values at a time. Accordingly, divide the array into four segments, calculate the partial sums for these segments and add the four partial sums together at the end. To calculate partial sums, it makes use of the `vec_add()` function that sequentially adds

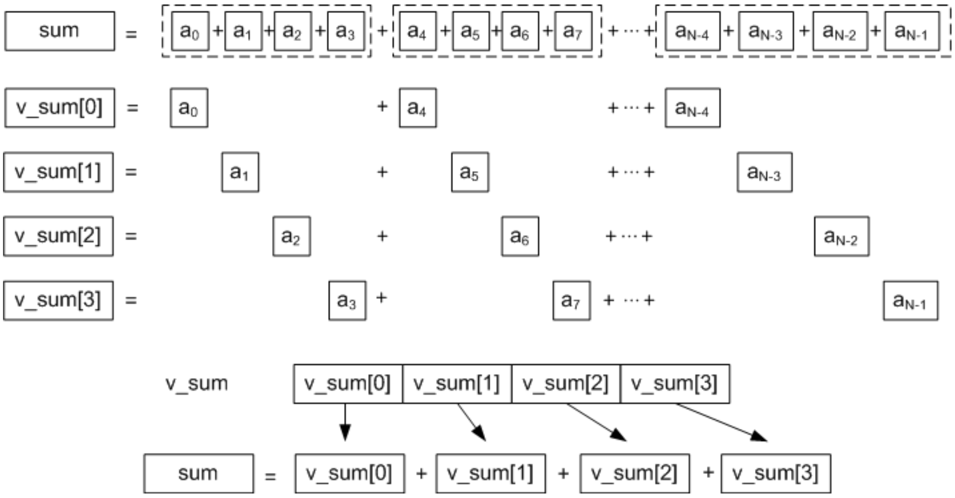


Figure 4.3: Array-summing using SIMD instructions

the vector variable `v_a` to the vector variable `v_sum`. Upon completion of partial sum calculations, the inclusive sum is obtain by totaling four element of vector `v_sum`.

```
// scalar array-summing
unsigned int scalar_sum(unsigned int a[N])
{
  int i;
  unsigned int sum = 0;
  for (i=0;i<N;i++)
    sum += a[i];

  return sum;
}

//Vectorized for Vector/SIMD Multimedia Extension
unsigned int vectorized_sum(unsigned int a[N])
{
  int i;
  unsigned int sum = 0;
  vector unsigned int v_a = (vector unsigned int *)a;
  vector unsigned int v_sum = {0,0,0,0};

  for(i=0;i<N/4;i++)
    v_sum = vec_add(v_a,v_sum);

  sum = v_sum[0] + v_sum[1] + v_sum[2] + v_sum[3];
  return sum;
}
```

Listing 4.1: Example of Array-summing

4.2.2 The SPE Instruction Set

The SPE instruction set is a new SIMD instruction set called the Synergistic Processor Unit Instruction Set Architecture [13]. It is similar to that of the PPEs vector/SIMD multimedia extensions, in the sense that they operate on SIMD vectors. However, the two vector instruction sets are distinct, and programs for the PPE and SPEs are compiled by different compilers. A large set of SPU C/C++ language extensions (intrinsics) [14] make the underlying SPU Instruction Set Architecture and hardware features conveniently available to C programmers. Listing 4.2 is the SPU version of array-summing example. The difference with PPE version is the SIMD function `spu_add(a,b)`. The list of all other intrinsics can be found in the C/C++ Language Extensions for Cell Broadband Engine Architecture specification [14].

```
unsigned int vectorized_sum(unsigned int a[N])
{
  int i;
  unsigned int sum = 0;
  vector unsigned int v_a = (vector unsigned int *)a;
  vector unsigned int v_sum = {0,0,0,0};

  for(i=0;i<N/4;i++)
    v_sum = spu_add(v_a,v_sum);

  sum = v_sum[0] + v_sum[1] + v_sum[2] + v_sum[3];
  return sum;
}
```

Listing 4.2: Array-summing using SIMD instructions in SPU

4.3 Data Transfer and Communication

In parallel programming, the data transfer and communication from core to core or core to memory is substantial. A better understanding of them is necessary to choose the right communication mechanisms. In this section the data transfer and communication performance of the Cell BE is analyzed. While the more detail analysis has been made [35], this analysis focuses only to three common mechanisms, which are used in most of The Cell programs: sequential DMA transfers, list DMA transfers and mailbox. The benchmarks run on the Cell Blade with two Cell processors [see Section 4.9.1 for specifications].

4.3.1 Sequential DMA Transfer

In this section, the behavior of the sequential DMA transfers is revealed through some benchmarks. The benchmarks measure the latency and bandwidth of simple blocking puts and gets, when the target is in main memory or in another SPEs local store. However put benchmark is not presented due to its similarity with the get performance. A sequential DMA transfer is restricted to continuous pieces of memory. The maximum size of a sequential DMA is 16 KB. For the transfer to both the global memory and local store, variables are aligned to to 128-byte boundary, which will give the maximum

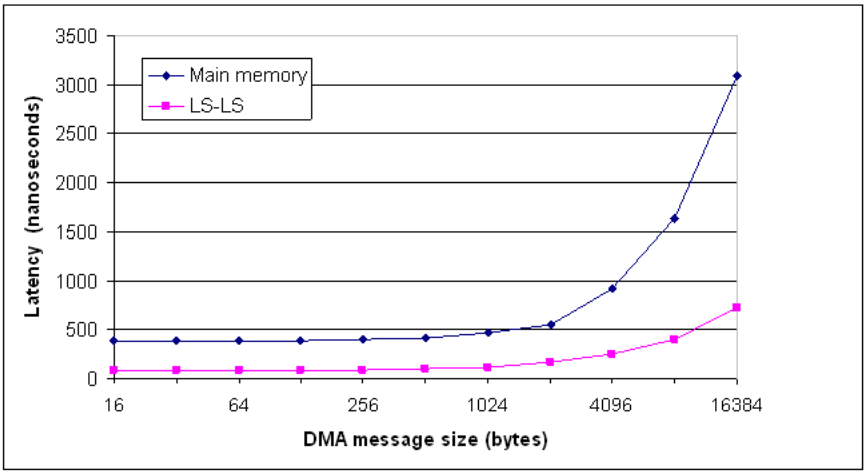


Figure 4.4: Latency of DMA get transfer (LS-Main memory and LS-LS) with different DMA message sizes

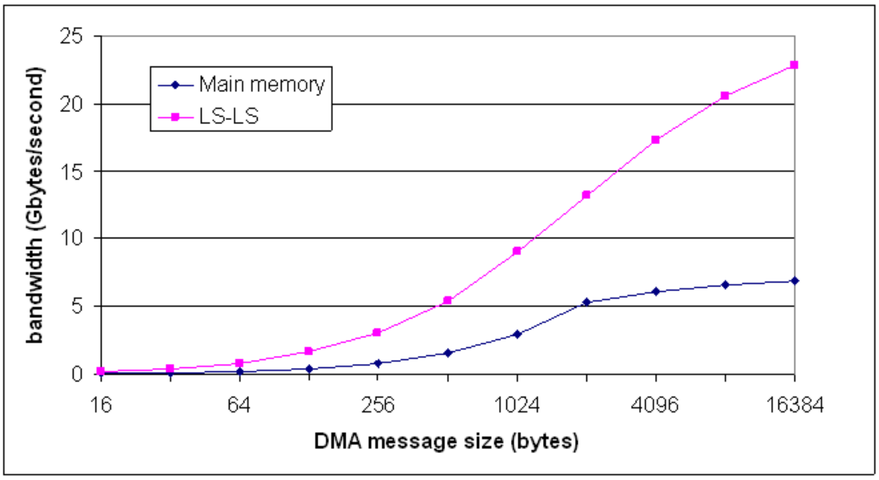


Figure 4.5: Bandwidth of DMA get transfer (LS-Main memory and LS-LS) with different DMA message sizes

throughput [6]. In each benchmark the results are deduced from timing 10000 iterations of the DMA operation to increase the timing accuracy. The transfer size ranges from 16 to 16384 bytes. The number of SPEs ranges from 1 to 16.

Figure 4.4 and 4.5 shows the latency and the bandwidth of DMA read operations from LS to main memory and from LS to LS with different DMA sizes. In LS to LS DMA transfer, the source and the target SPE are neighbors. The graph shows that LS to LS transfer has smaller latency and larger bandwidth than LS to main memory transfer. Furthermore, we can see that up to a DMA size of 1024 bytes the throughput scales approximately linearly with the DMA sizes. The latency difference between a 16 and 1024 byte transfer is therefore quite small.

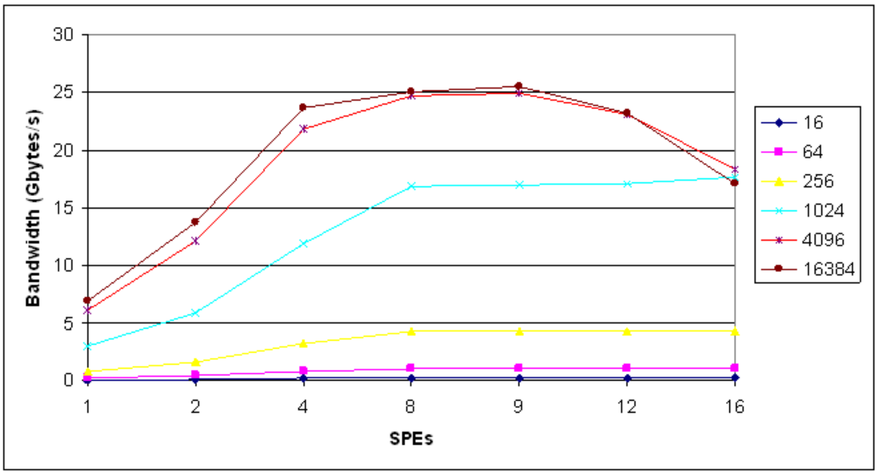


Figure 4.6: Bandwidth of LS- main memory DMA get transfer with different SPEs and DMA message sizes

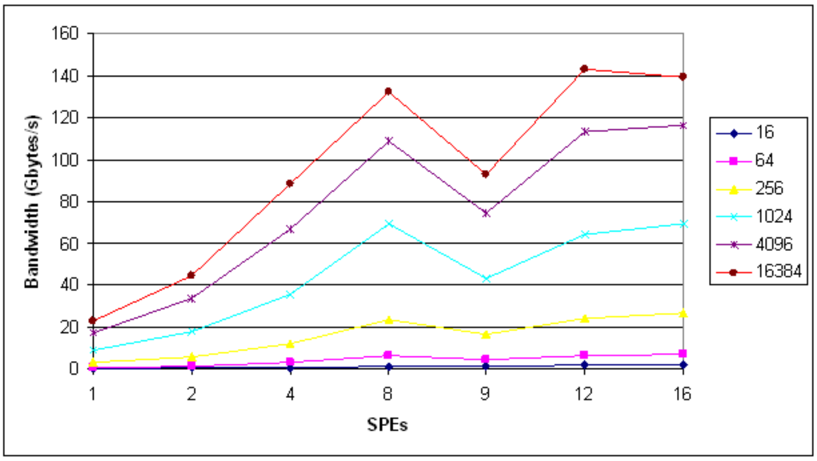


Figure 4.7: Bandwidth of LS-LS DMA get transfer with different SPEs and DMA message sizes

Figure 4.6 and 4.7 show the bandwidth of DMA read operations from LS to main memory and from LS to LS with different number of working SPEs. The graph shows that LS to main memory transfer reach the peak bandwidth of 25G bytes/s with 8-9 SPEs. For LS to LS transfer, with number of SPEs equal or smaller than 8, as expected, the aggregated throughput scales with the number of SPEs. There is a dearease when the number of SPEs is 9 because the SPEs are scheduled on physically different Cell processors. The FLEXIO link has lower bandwidth than EIB and forms the bottleneck when multiple SPEs of different Cell processors communicate.

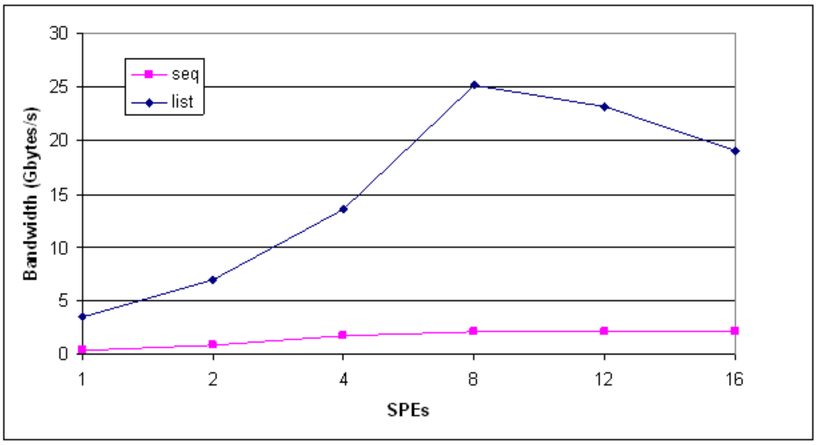


Figure 4.8: Bandwidth of list and sequential LS- main memory DMA get transfer with different SPEs

4.3.2 List DMA Transfer

In the previous section the characteristics of the sequential DMA transfers were investigated. In this section we continue with the list DMA transfers. DMA list is a sequence of transfer elements (or list elements) that, together with an initiating DMA-list command, specifies a sequence of DMA transfers between a single continuous area of LS and possibly discontinuous areas in main storage. DMA lists can therefore be used to implement scatter-gather functions between main storage and the LS. For this list DMA benchmark, the list transfer element size is 128 bytes, the number of list elements is 16. To compare the performance difference of sequential DMA versus a single list DMA, the sequential DMA transfers are also benchmarked with the same settings. The results of LS-to-main memory and LS-to-LS benchmarks are presented in Figure 4.8 and 4.9 respectively. The first observation to make is that the throughput in both LS-main memory and LS-LS of list transfer is much higher than of sequential DMA. List DMA transfer should be use when we need to transfer to discontinuous locations. The second remark is that LS-LS transfer has higher throughput than LS-main memory. There is a drop in LS-main memory list transfer when the number of SPE is larger than 8, perhaps because there is a contention of transfer to main memory.

4.3.3 Mailbox Communication Mechanisms

Mailbox is a simply and widely used communication mechanism of the Cell [2]. Mailboxes are designed to transfer 32-bit messages between the local SPU and the PPE or local SPU and other SPEs. The mailboxes are accessed from the local SPU using the channel interface and from the PPE or other SPEs using the MMIO interface. The mailbox has two variants, one that is context safe and one that is not. The difference lies in whether it is checked if the context is currently running on the SPE. When there are more SPE contexts than physical SPEs, context scheduling is performed. The context unsafe mailbox assumes that the SPE context will always be on the same SPE. In the

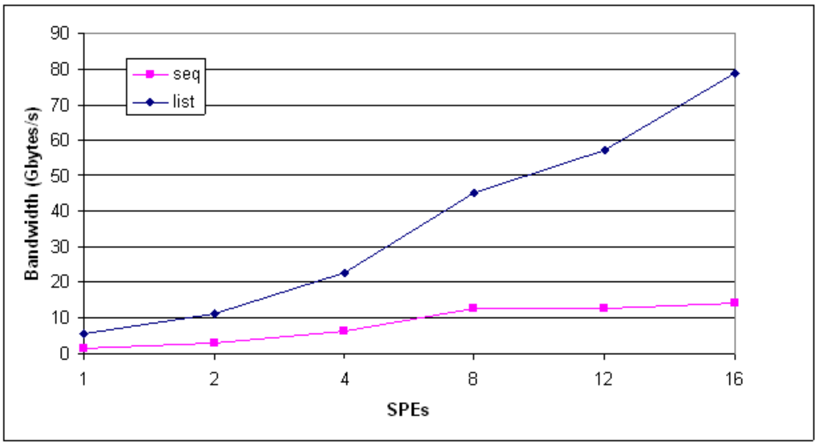


Figure 4.9: Bandwidth of list and sequential LS- LS memory DMA get transfer with different SPEs

benchmark, the latency of two types of mailbox is compared. Table 4.1 shows the latency of two mailbox mechanisms. It can be seen that the latency of fast mailbox mechanism is smaller than the normal one.

SPEs	Mailbox normal	Mailbox fast
1	1.76	0.44
2	1.76	0.43
4	1.77	0.43
8	1.78	0.42
16	2.36	0.88

Table 4.1: Average latency of mailbox communication in us

4.4 Parallel Implementation

In three previous sections, we already investigated the general information and the instruction set of the Cell BE. We also did some small communication benchmarks to test the performance of the Cell. In this section, we parallelize the calculating process of GLCM. In parallel programming on the Cell BE, we have to make use of two following point to maximize the performance of the Cell:

- Operate multiple SPEs in parallel to maximize operations that can be executed in a certain time unit.
- Perform SIMD parallelization on each SPE to maximize operations that can be executed per instruction.

For parallelization the work over SPEs, there are three ways in which the SPEs can be used: the multistage pipeline model, the parallel stages model and the services model

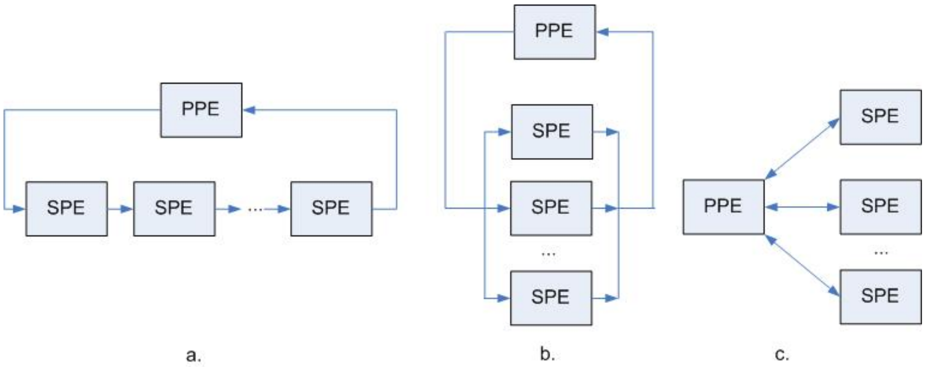


Figure 4.10: Parallel programming model: a.Multistage pipeline model b.Parallel stages model c.Services model

[15]. Figure 4.10 displays 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 feed to 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 being processed. 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 stages model can be applied when the data is partitionable or tasks can be process concurrently. Each SPE will process different parts of data or different tasks in parallel. This is the basic and most popular parallel model.

In the services model, the PPE assigns different services to different SPEs, and the PPEs main process calls upon the appropriate SPE when a particular service is needed. This model is suitable for service-driven applications.

For parallelizing the work on each SPE, the SIMD instructions can accelerate significantly the computing process. With SIMD processing, data are presented in vector type with the length of 128 bits. A single SIMD instruction will be applied to multiple elements of the vector to exploit data-level parallelism. However, not every parallelizable application benefits from SIMD processing because of not only data dependency issues but also non-aligned and irregular data access problems. Unfortunately, the co-occurrence matrix belongs to these problems.

In following parts, parallelism of co-occurrence matrix is described in detail.

4.5 Parallelism of Co-occurence Matrix on Multi-SPEs

In this section the data-level parallelism of co-occurrence matrix on multi-SPEs is implemented.

4.5.1 Parallel Strategy

In the sequential implementation using array structure, to calculate co-occurrence matrix, each pixel and its neighbors of the image are read sequentially. To parallelize this process, among three models: the multistage pipeline model, the parallel stages model and the services model, the second one is the fittest because we can easily partition the image by the number of working SPEs. Each divided part of the image is processed independently in the corresponding SPE. Figure 4.11 describes the model using 4 SPEs. We process a sub-part of the image in each SPE, creating a sub-cooccurrence matrix. After that, these matrices will be summed up to produce final co-occurrence matrix. At this stage, there are several ways to calculate the sum of all sub-cooccurrence matrices.

The first method is tree structure as in Figure 4.11. 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 method utilizes SPEs inefficiently, because in first step, only SPE0 and SPE2 work, and in second step, only SPE0 works.

The second method is described in Figure 4.12. Sub-matrix in each SPEs is partitioned into N parts (N is the number of SPEs). Each of these parts is added with corresponding parts from all other SPEs. Following this method, all SPEs work equally and the final co-occurrence matrix is calculated in one step.

As described before, in calculation of co-occurrence matrix, we have to read the value of a pixel and its neighbors in eight directions (0, 45, 90, 135, 180, 225, 270, 315 degree) to update the matrix. Therefore, another way to parallelize this process is in each SPE, we read a pixel and its neighbor in one direction as displayed in Figure 4.13. After we have sub-cooccurrence matrix in each SPE, these matrices will be summed up to produce final co-occurrence matrix.

When choosing the model or dividing data and tasks for SPEs, we must notice that the memory (Local Store or LS) of SPE is limited in only 256 KBytes. Storing data in LS can accelerate the computing process, however 256 KB is a considerably small capacity, which can not be used to store data of large size. Therefore we have to consider carefully when making use of local store. An image is stored in the main memory, at the beginning of calculation process, small parts of the image is transfered to Local Store of SPEs through DMA mechanism. While the size of an image can vary from KBytes to MBytes, it may not fit with LS. Therefore we have to consider how many parts of the image will be split to process in each SPE. Another thing must be considered here is the size of co-occurrence matrix. If the co-occurrence matrix is defined as an array of 4-byte integer type and the dimension of the matrix is 32×32, 64×64, 128×128, 256×256 and 512×512, its size will be 4KB, 16KB, 64KB, 256KB and 1024 KB respectively. We can easily realize that co-occurrence matrix with gray level equal or larger than 256 can not be store in LS of SPE. These large-size matrices must be stored in main memory, which slows down its building significantly due to DMA command to transfer its elements from main memory to LS and vice versa.

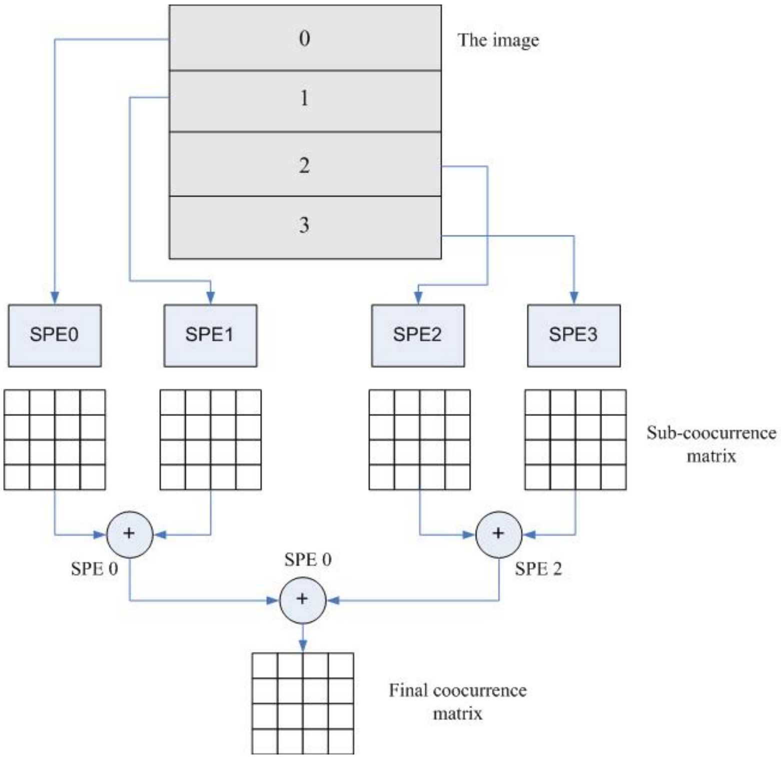


Figure 4.11: Method 1: Parallel implementation of co-occurrence matrix by splitting an image with 4 SPEs

4.5.2 Implementation on the Cell BE

This section describes details about parallel implementation of co-occurrence matrix in PPE and SPEs. As analyzed in section 4.5.1, the better parallel strategy which uses SPEs more effectively is that splitting an image to calculate sub-cooccurrence matrix in each SPEs, then splitting these sub-matrices to calculate the final matrix.

4.5.2.1 Initialization environment for SPEs in PPE and SPEs

In general, a typical execution sequence of a Cell BE program comprises following steps:

- (In PPE) Load the SPE program to the LS.
- (In PPE) Instruct the SPE to execute the SPE program.
- (In SPE) Transfer required data from the main memory to the LS.
- (In SPE) Process the data in the LS in accordance with the requirements.
- (In SPE) Transfer the processed result from the LS to the main memory.
- (In SPE) Notify the PPE program of the termination of processing.

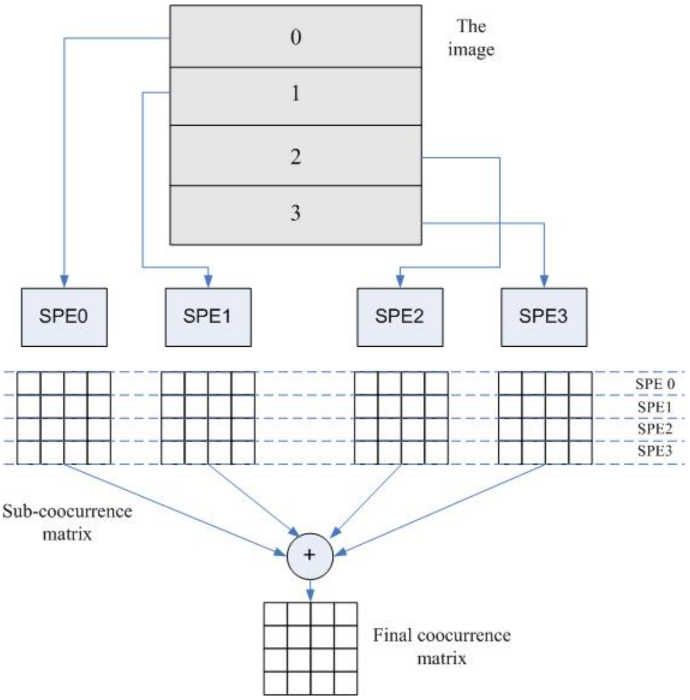


Figure 4.12: Method 2: Parallel implementation of co-occurrence matrix by splitting an image with 4 SPEs

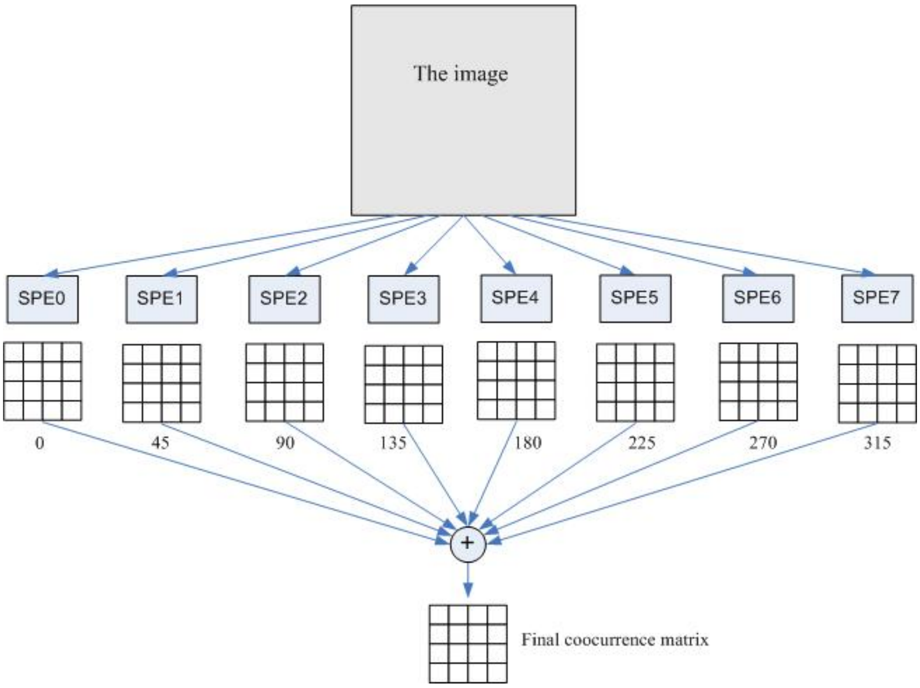


Figure 4.13: Parallel implementation of co-occurrence matrix in eight directions with 8 SPEs

In the first step, a PPE module initializes an SPE module running by creating a thread on the SPE, using the `spe_context_create`, loading the program `com_spu` to SPEs using `spe_program_load` [Listing 4.3]. In Listing 4.3, `NUM_SPE` indicates the number of SPEs used.

```
for(i=0;i<NUM_SPE; i++){
  spe[i] = spe_context_create(0, NULL);
  if (!spe){
    perror("spe_context_create");
    exit(1);
  }

  ret =spe_program_load(spe[i], &com_spu);
  if (ret ){
    perror("spe_program_load");
    exit(1);
  }
}
```

Listing 4.3: Create context on SPEs

In second step, an SPE context is executed on a physical SPE by calling the `spe_context_run` function. This subroutine causes the current PPE thread to transit to a SPE thread by passing its execution control from the PPE to the SPE whose context is scheduled to run on [Listing 4.4]. In the listing, `com_params` retains the pointer to the parameter set that should be passed to the SPE program. This parameter set contains necessary information a SPE must know to load the data from main memory or other SPEs' LS, such as Effective Address (EA) of the image, co-occurrence matrix, texture features, EA of other SPEs' LS, etc [Listing 4.5]. In the SPE side, this parameter set is loaded from PPE through DMA transfer command [Listing 4.6]. `com_params` is 128-byte aligned through attribute `__attribute__((aligned(128)))`. Finally, initialization process ends by sending a mailbox message [Listing 4.7] from PPE to SPEs, SPEs stall until receiving this message, after that SPEs start calculating process (from step 3).

```
void *run_com_spe(void *thread_arg)
{
  int ret;
  thread_arg_t *arg = (thread_arg_t *) thread_arg;
  unsigned int entry = SPE_DEFAULT_ENTRY;
  spe_stop_info_t stop_info;
  ret = spe_context_run(arg->spe, &entry,0, arg->com_params,NULL, &
      stop_info);
  if (ret<0){
    perror("spe_context_run");
    exit(1);
  }
  return NULL;
}

for(i=0;i<NUM_SPE; i++){
  ret = pthread_create(&thread[i], NULL, run_com_spe, &arg[i]);
  if (ret) {
    perror(" prthead_create");
    exit(1);
  }
}
```

Listing 4.4: Running SPE program

```
typedef struct {
  unsigned long long ea_image; //effective address of the image
  unsigned long long ea_com;  // EA of co-occurrence matrix
  unsigned long long ea_features; // EA of features
  unsigned long long ea_normal; // EA of normalizaion
  unsigned long long ea_done;   // EA of Done
  unsigned long      lsa_params[NUM_SPE];//LS address of SPE variables
  void*          ea_ls[NUM_SPE]; // EA of SPEs' Local Store
  int          size_w;  // width size of the image
  int          size_h;  // height size of the image
  int                pad[4];  //padding
} com_params_t;

com_params_t com_params[NUM_SPE] __attribute__((aligned(128)));
```

Listing 4.5: `com_params` structure

```
mfc_get(&com_params, argp ,sizeof(com_params_t),tag, 0, 0);
mfc_write_tag_mask (1 << tag);
mfc_read_tag_status_all();
```

Listing 4.6: Loading parameter set in SPE side

```
//send message from PPE to SPU
for ( i=0; i<NUM_SPE; i++)
  _spe_in_mbox_write(spe[i], (unsigned int *)&i, 1,
      SPE_MBOX_ANY_NONBLOCKING);

//receiving message in SPU
myId = _spu_read_in_mbox();
```

Listing 4.7: Sending and receiving mailbox message

4.5.2.2 Calculating Sub-cooccurrence Matrix in SPEs

As described in parallel tragedy, the image will be divided into `NUM_SPE` parts. Each part is processed in one SPE, creating sub-cooccurrence matrix. In SPE, image data is stored in an array of unsigned char `m_ImageData[MAX_SIZE]`, the sub-cooccurrence matrix is defined as a two-dimension array of unsigned integer `matrix[Ng][Ng]`, where `Ng` is the number of gray level. With large-size images, the size of the part processed in a SPE, `size_per_spe`, can be larger than Local Store capability, moreover, each DMA command can only transfer up to 16KB, therefore, each part will be divided into blocks which can be transfered from main memory to LS through DMA. These blocks have the height in pixels `block_h`, the size in bytes `size_per_block = image_w * image_h / NUM_SPE` and the number of block `num_block = image_h /(block_h*NUM_SPE)`.

After all parameters of image blocks are defined, through the loop, blocks are transfered sequentially to SPE. Each pixel of blocks is read from `m_ImageData` to update sub-cooccurrence matrix `matrix[Ng][Ng]`, as in Listing 4.8. The normalization variable `normal` is also updated each loop.

```
unsigned char  m_ImageData[MAX_SIZE] __attribute__((aligned(128)));
unsigned int matrix[Ng][Ng] __attribute__((aligned(128)));

int block_h=16;
int size_per_spe = image_w * image_h / NUM_SPE;
int size_per_block = image_w * block_h;
int num_block = image_h/ (block_h*NUM_SPE);

for (i =0; i < num_block; i++){
  //load blocks
  mfc_get (m_ImageData, com_params.ea_image + (i*size_per_block-
      image_w+ myId*size_per_spe)*sizeof(char),sizeof(char)*(
      size_per_block+ 2*image_w),tag, 0, 0);
  mfc_write_tag_mask (1 << tag);
  mfc_read_tag_status_all ();

for( y = 1;  y <= block_h;  y++ ){
  for( x = 1;  x < image_w-1;  x++ ){
    y_w = y*image_w;
    matrix[m_ImageData[y_w+x]][m_ImageData[y_w-image_w +x-1]]++ ;
    matrix[m_ImageData[y_w+x]][m_ImageData[y_w-image_w  +x]]++ ;
    matrix[m_ImageData[y_w+x]][m_ImageData[y_w-image_w  +x+1]]++ ;
    matrix[m_ImageData[y_w+x]][m_ImageData[y_w +x-1]]++ ;
    matrix[m_ImageData[y_w+x]][m_ImageData[y_w +x+1]]++ ;
    matrix[m_ImageData[y_w+x]][m_ImageData[y_w+image_w  +x-1]]++ ;
    matrix[m_ImageData[y_w+x]][m_ImageData[y_w+image_w  +x]]++ ;
    matrix[m_ImageData[y_w+x]][m_ImageData[y_w+image_w  +x+1]]++ ;
    normal = normal + 8;
  }
}
}
```

Listing 4.8: Calculating sub-cooccurrence matrix in SPE

4.5.2.3 Calculating Total Co-occurrence Matrix in SPEs

After SPEs finish calculating sub-cooccurrence matrices, the next step is to generate the total co-occurrence matrix by summing all the sub-ones. At this point, SPEs need to be synchronized with others so that all SPEs know that the calculation process of sub-matrices is already finished and the data is available. SPEs do this job by sending the normalization value to PPE and wait until PPE receives all of them, calculates their sum and sends back the total value to SPE. The synchronization is implemented through mailbox mechanism as in Listing[4.9]

```
//in SPE
_spu_write_out_box(normal);
normal= _spu_read_in_mbox();

//in PPE
for (i=0;i<NUM_SPE;i++){
  while (!spe_out_mbox_status(spe[i])) {} //check mailbox status
  unsigned int temp;
  spe_out_mbox_read(spe[i], &temp, 1); //read the mailbox
  normal +=temp;
}

for ( i=0; i<NUM_SPE; i++)
  spe_in_mbox_write(spe[i], (unsigned int *)&normal, 1,
      SPE_MBOX_ANY_NONBLOCKING); //write the normalization value to SPE
```

Listing 4.9: Computing the normalization value and synchronization between SPEs

As discussed in parallel strategy, each SPE will calculate a part of total co-occurrence matrix. It loads corresponding part of sub-cooccurrence matrix from others SPE, sum all these parts and then divides by normalization value. Parts from other SPEs is allocated in `temp_matrix[i]` with size determined by `ROW_SZ`. Their total sum after normalization is allocated in `temp_t_matrix`,which is a part of total co-occurrence matrix and is needed to transfer back to main memory. Listing 4.10 shows the process. By the end of this computation stage, we have a co-occurrence matrix in the main memory. The next step, calculating texture features from co-occurrence matrix will be discussed in section 4.7.

```
for(k=0;k<PER_SPE/ROW_SZ;k++){
  for (i=0;i<NUM_SPE;i++){
    if (i!=myId){
      mfc_get(temp_matrix[i], com_params.ea_ls[i] + lsa_params_spu[i].
          lsa_com + (myId*Ng*PER_SPE+k*ROW_SZ*Ng)*sizeof(int), ROW_SZ*
          Ng*sizeof(int),tag,0,0);
      mfc_write_tag_mask(1<<tag);
      mfc_read_tag_status_all();
    }
  temp_matrix[myId] = matrix[(myId*PER_SPE + k*ROW_SZ)*Ng];
  for(i=0;i<ROW_SZ;i++){
    for(j=0;j<Ng;j++){
      for(h=0;h<NUM_SPE;h++)
        temp_t_matrix[i][j] += temp_matrix[h][i][j];
      temp_t_matrix[i][j] = temp_t_matrix[i][j]/normal;
    }
  }

  mfc_put(temp_t_matrix, com_params.ea_com + (myId*Ng*PER_SPE+k*ROW_SZ
      *Ng)*sizeof(float), ROW_SZ*Ng*sizeof(float), tag, 0, 0);
  mfc_write_tag_mask (1 << tag);
  mfc_read_tag_status_all
}
```

Listing 4.10: Summing sub-cooccurrence matrices and normalization

4.6 Implementation of the Co-occurrence Matrix in the SPE using SIMD Instructions

SIMD instruction is very effective in accelerating programs containing large data-level parallelism such as multimedia applications since multiple items of data can be processed at once. However, 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 these problems. To build a co-occurrence matrix, based on neighboring pair of pixels, elements

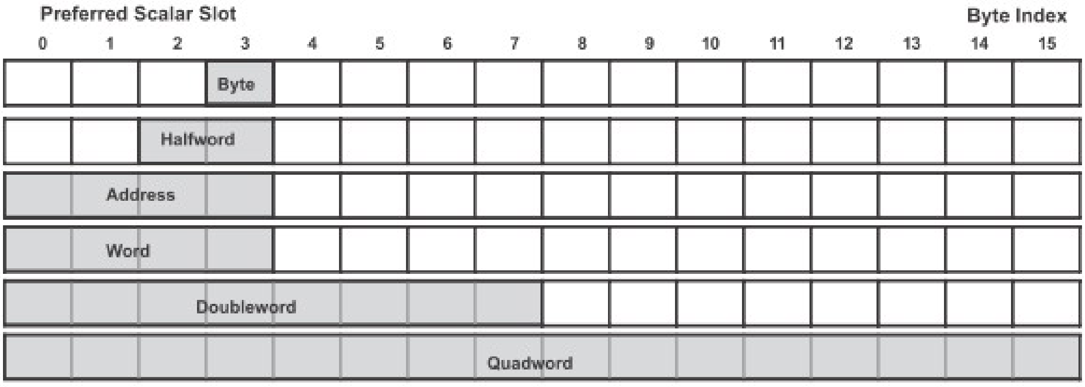


Figure 4.14: Register layout of data types and preferred scalar slot

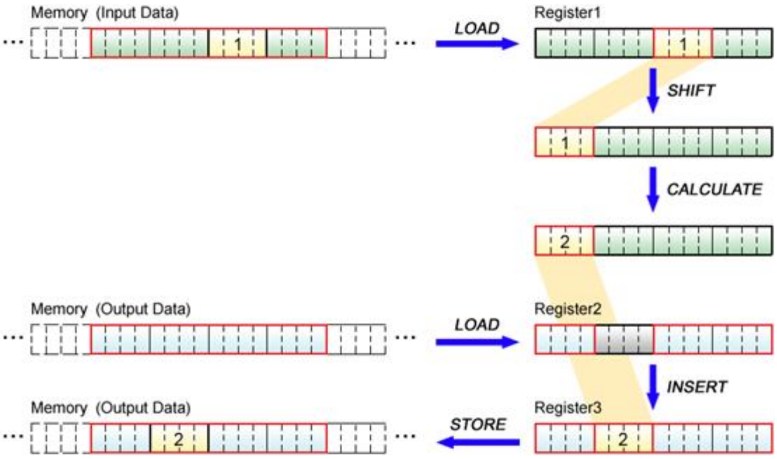


Figure 4.15: SPE scalar operations

of the matrix are accessed irregularly to be updated 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 displayed in Figure 4.14. Scalar data only uses a predetermined part of the register called the preferred slot. `char` type is located to Byte 3, `short` type to Bytes 2 and 3, `int` 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 requires that they must be aligned in preferred slot by shuffling through `rotqby` and `shufb` instructions [1]. Figure 4.15 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 [1], 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. Listing 4.11 is the implementation of LDT method in sub-cooccurrence matrix.

```
vector unsigned int matrix[Ng][Ng] __attribute__((aligned(128)));
vector unsigned int one ={1,1,1,1};

for( y = 1;  y <= block_h;  y++ ){
  for( x = 1;  x < image_w-1;  x++ ){
    y_w = y*image_w;
    matrix[m_ImageData[y_w+x]][m_ImageData[y_w-image_w +x-1]]+=one ;
    matrix[m_ImageData[y_w+x]][m_ImageData[y_w-image_w  +x]]+=one ;
    matrix[m_ImageData[y_w+x]][m_ImageData[y_w-image_w  +x+1]]+=one ;
    matrix[m_ImageData[y_w+x]][m_ImageData[y_w +x-1]]+=one;
    matrix[m_ImageData[y_w+x]][m_ImageData[y_w +x+1]]+=one ;
    matrix[m_ImageData[y_w+x]][m_ImageData[y_w+image_w  +x-1]]+=one ;
    matrix[m_ImageData[y_w+x]][m_ImageData[y_w+image_w  +x]]+=one ;
    matrix[m_ImageData[y_w+x]][m_ImageData[y_w+image_w  +x+1]]+=one ;
    normal += eight;
  }
}
```

Listing 4.11: Calculating sub-cooccurrence matrix using LDT method

If we still use the scalar computation in calculating sub-cooccurrence matrix, we can use SIMD instructions in summing all these matrices. Elements of sub-matrices are aligned and accessed consecutively, which is perfect for SIMD operations. Listing 4.12 shows the SIMD code.

```
vector unsigned int *v_temp_matrix[NUM_SPE];
vector float *v_tempt_t_matrix = (vector float *) temp_t_matrix;

for (i=0;i<NUM_SPE;i++)
  v_temp_matrix[i] = (vector unsigned int*) temp_matrix[i];

for(k=0;k<PER_SPE/ROW_SZ;k++){
  //...
  for(i=0;i<ROW_SZ;i++){
    for(j=0;j<Ng/4;j++){
      for(h=0;h<NUM_SPE;h++)
        v_temp_t_matrix[i*Ng/4+j] += v_temp_matrix[h][i*Ng/4+j];
      v_temp_t_matrix[i*Ng/4+j] = divf4(spu_convtf(v_temp_t_matrix[i][
          j],0),normal);
    }
  }
  //...
}
```

Listing 4.12: Summing sub-cooccurrence matrices and normalization using SIMD instruction

4.7 Parallel Implementations of Texture Feature Extraction

In previous section, parallel implementation in calculating co-occurrence matrix is investigated. Based on the matrix, 13 Haralick texture features can be extracted. In this section, parallelization of feature extraction process is considered. Section 4.7.1 is the parallel strategy and section 4.7.2 is the details of applying the strategy on PPE and SPEs.

4.7.1 Parallel Strategy

As described in chapter 2, using gray level co-occurrence matrix, Haralick defined 13 texture features. The easiest way to parallelize the computing process of these features is that calculating one or a group of features in each processor. However this approach has a bottleneck since the computing intensiveness of features is different from each others, which makes the load unbalanced between SPEs. Another approach which can give better balance is dividing the co-occurrence matrix into smaller ones, each of them is processed in a SPE to generate features. The sum of these features in all SPEs is the final value. Figure 4.16 depicts this strategy with 4 SPEs.

Because of dependency of features with each others, we cannot compute all the features in parallel. Features are classified into three groups following the order of execution step 1 to step 3. The dependency of features are desribed by dash lines in the Figure 4.16. Three groups as divided as follow:

- Group 1: Energy f1, Contrast f2, Entropy f3, Homogeneity f6, Mean μ, P_{x+y}, P_{x-y}, P_x.
- Group 2: Variance f4, Sum average f7, Sum entropy f9, Different entropy f11, Deviation σ, Different average, HX, HXY HXY1, HXY2.
- Group 3: Correlation f5, Sum variance f8, Different variance f10, Information measure of Correlation f12, f13.

In a SPE, elements of sub-cooccurrence matrix are aligned and accessed consecutively to calculate features. Therefore, it is appropriated to apply SIMD instruction to vectorize the computing process. Details about vectorization is described in the next section.

4.7.2 Implementation on the Cell BE

In this section, parallel strategy discussed in section 4.7.1 is implemented on the Cell. Figure 4.16 shows three steps in the computing process. In SPE, texture features are

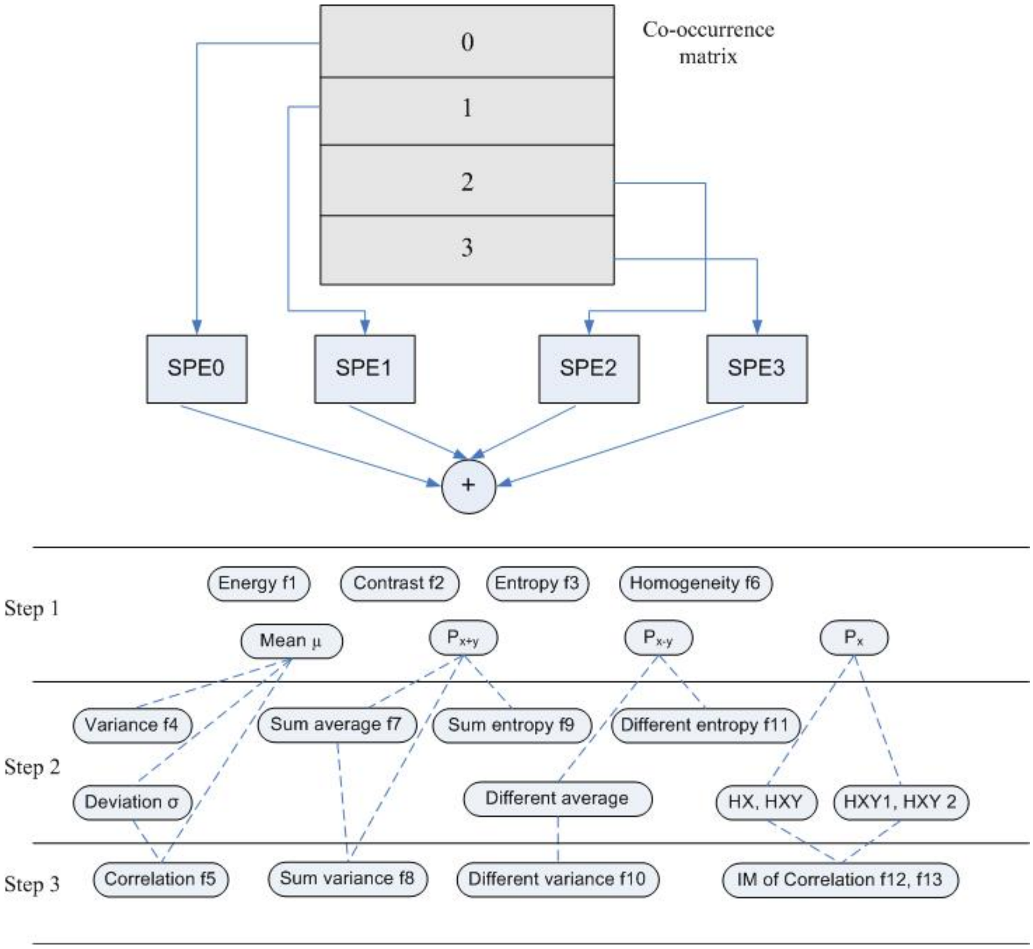


Figure 4.16: parallel model in computing texture features

defined as members in the structure `features`[listing 4.13]. `m_features` is the shared data between SPEs. In each step, `m_features` is calculated based on sub-cooccurrence matrix and then sent back to PPE to synchronize with data from other SPEs.

Listing 4.14 presents step 1. Here, the computing is deployed by SIMD instructions, therefore we have to define pointers of vector type (`v_COM`,`v_fEnergy`, `v_fContrast`, `v_fEntropy` ...) point to sub-cooccurrence matrix and feature variables(`temp_matrix`, `m_features.m_fEnergy`, `m_features.m_fContrast`, `m_features.m_fEntropy` ...). Function `FeatureCalculation1(myId*PER_SPE)` [listing 4.15] calculates vector varibales `v_fEnergy`, `v_fContrast`, `v_fEntropy`, `v_fHomogeneity`, `v_ux`, `v_px[]` and two scalars `m_features.pxy[]`, `m_features.pdxy[]`. After calculation, the scalars are formed by summing all elements of the vector variables. And finally, structure `m_features` is loaded from PPE to update and then transfered back by DMA get & put command for synchronization. Because `m_features` is shared data, therefore to avoid the simultaneous update, the pieces of code manipulating `m_features` must be critical sections. The sync library provides `mutex` mechanism to handle this synchronization. Before accessing `m_features`, the mutex is locked, after finishing update, the mutex is unlocked.

```
typedef struct {
  float   m_fEnergy ;
  float   m_fEntropy ;
  float   m_fContrast ;
  float   m_fHomogeneity ;
  float   m_fCorrelation ;
  float   m_fVariance ;
  float   m_fSumAver ;
  float   m_fSumVari ;
  float   m_fSumEntr ;
  float   m_fDiffVari;
  float   m_fDiffEntr;
  float   m_fInfMeaCor1;
  float   m_fInfMeaCor2
  float   ux ;
  float   stdDevix ;
  float   DiffAver;
  float   pdxy[Ng];
  float   px[Ng];
  float   pxy[2*Ng];
  float   hx;
  float   hxy1;
  float   hxy2;
  char    pad[4];
} features;

features m_features __attribute__((aligned(128)));
```

Listing 4.13: `features` structure

```
vector float * v_COM = (vector float *)temp_matrix;
FeatureCalculation1(myId*PER_SPE);

m_features.m_fEnergy = v_fEnergy[0] + v_fEnergy[1] + v_fEnergy[2] +
    v_fEnergy[3];
m_features.m_fContrast = v_fContrast[0] + v_fContrast[1] + v_fContrast
    [2] + v_fContrast[3];
m_features.m_fEntropy = v_fEntropy[0] + v_fEntropy[1] + v_fEntropy[2]
    + v_fEntropy[3];
m_features.m_fHomogeneity = v_fHomogeneity[0] + v_fHomogeneity[1] +
    v_fHomogeneity[2] + v_fHomogeneity[3];
m_features.ux = v_ux[0] + v_ux[1] + v_ux[2] + v_ux[3];

mutex_lock(mutex);

temp_features.m_fEnergy += m_features.m_fEnergy;
temp_features.m_fContrast += m_features.m_fContrast;
temp_features.m_fEntropy += m_features.m_fEntropy;
temp_features.m_fHomogeneity += m_features.m_fHomogeneity;
temp_features.ux += m_features.ux;
for(k =0; k <(PER_SPE+Ng)/4;k++)
  v_temp_pxy[myId*PER_SPE/4+k] = spu_add(v_pxy[myId*PER_SPE/4+k],
      v_temp_pxy[myId*PER_SPE/4+k] );
for (k =0; k <Ng/4;k++)
  v_temp_pdxy[k]  = spu_add(v_pdxy[k],v_temp_pdxy[k]);
for (k =0; k <Ng/4;k++)
```

```
    v_temp_px[k]  = spu_add(v_px[k],v_temp_px[k]);

  mutex_lock(mutex);
  spu_write_out_mbox((unsigned int)&myId);
```

Listing 4.14: Step 1 in computing features

```
  void FeatureCalculation1(int m){
    int j,k;
    vector float index;
    vector float temp;
    vector float temp_px;
    for (j = 0; j< PER_SPE; j++) {
      for (k = 0; k< Ng/4; k++){
        v_fEnergy = spu_madd( v_COM[j*Ng/4+k], v_COM[j*Ng/4+k],
            v_fEnergy); //energy

        v_fContrast = spu_madd(v_index_ct[j*Ng/4+k], v_COM[j*Ng/4+k],
            v_fContrast); //contrast

        comp = spu_cmpabseq(v_COM[j*Ng/4+k],0);
        temp = spu_sel(one,v_COM[j*Ng/4+k],comp);
        v_fEntropy = spu_madd(temp,logf4(temp),v_fEntropy); //entropy

        index = spu_add(v_index_ct[j*Ng/+k],one);
        v_fHomogeneity  = spu_add(v_COM[j*Ng/4+k]/index,v_fHomogeneity);

        v_ux = spu_madd(v_COM[j*Ng/4+k], v_index3[j][k], v_ux);

        temp_px = spu_add(v_temp_matrix [j*Ng/4+k],temp_px);

        m_features.pxy[m+j+4*k]   +=  m_COM[j][4*k];
        m_features.pxy[m+j+4*k+1] +=  m_COM[j][4*k+1];
        m_features.pxy[m+j+4*k+2] +=  m_COM[j][4*k+2];
        m_features.pxy[m+j+4*k+3] +=  m_COM[j][4*k+3];

        if (j+m>=4*k) m_features.pdxy[j+m-4*k] +=  m_COM[j][4*k];
          else m_features.pdxy[4*k-j-m] +=  m_COM[j][4*k];
        if (j+m>=4*k+1) m_features.pdxy[j+m-4*k+1] +=  m_COM[j][4*k+1];
          else m_features.pdxy[4*k+1-j-m] +=  m_COM[j][4*k+1];
        if (j+m>=4*k+2) m_features.pdxy[j+m-4*k+2] +=  m_COM[j][4*k+2];
          else m_features.pdxy[4*k+2-j-m] +=  m_COM[j][4*k+2];
        if (j+m>=4*k+3) m_features.pdxy[j+m-4*k+3] +=  m_COM[j][4*k+3];
          else m_features.pdxy[4*k+3-j-m] +=  m_COM[j][4*k+3];
      }
      m_features.px[j+m] = temp_px[0] + temp_px[1] + temp_px[2] +
          temp_px[3];
      temp_px = zero;
    }
  }
```

Listing 4.15: Function `FeatureCalculation1(myId*PER_SPE)`

We explain some instructions in the function `FeatureCalculation1()`. Energy feature, in scalar implementation, is calculated through the formula:

`m_fEnergy += m_COM[j][k]*m_COM[j][k]`

In SIMD implementation, this formula is converted to:

```
v_fEnergy = spu_madd(v_COM [j*Ng /4+ k],v_COM [j*Ng /4+ k], v_fEnergy)
```

`spu_add(a,b,c)` is a function of vector multiply and add, where each element of vector `a` is multiplied by vector `b` and added to the corresponding element of vector `c`.

Contrast feature, in scalar implementation, is calculated by the formula:

```
m_fContrast += (i-j)*(i-j)*m_COM[i][j]
```

In SIMD implementation, it is transformed to:

```
v_fContrast = spu_madd(v_index_ct[j*Ng/4+k],v_COM[j*Ng/4+ k],v_fContrast)
```

Vector `v_index_ct[j*Ng/4+k]` stores values of `(i-j)*(i-j)`.

Entropy feature, in scalar implementation, is calculated by the formula:

```
if(m_COM[j][k] != 0)
    m_fEntropy += m_COM[j][k]*log(m_COM[j][k]);
```

In SIMD implementation, it is transformed to:

```
comp = spu_cmpabseq( v_COM [j*Ng/4+k],0);
 temp = spu_sel(one , v_COM [j*Ng/4+k],comp);
 v_fEntropy = spu_madd(temp,logf4(temp),v_fEntropy);
```

`spu_cmpabseq(a, b)` is a function of vector compare absolute equal. The absolute value of each element of vector `a` is compared with the absolute value of the corresponding element of vector `b` . If the absolute values are equal, all bits of the corresponding element of the result vector are set to one; otherwise, all bits of the corresponding element of the result vector are set to zero. `spu_sel(a,b,c)` is a function of select bits. For each bit in the 128-bit vector `c`, the corresponding bit from either vector `a` or vector `b is selected`. If the bit is 0, the bit from `a` is selected; otherwise, the bit from `b` is selected. We need two functions `spu_cmpabseq()` and `spu_sel()` because in calculation of entropy, we use logarithm function and we have to guarantee that the parameter of logarithm function is non-zero. While the value of the elements of vector `v_COM[j*Ng/4+k]` can be zero, through two functions `spu_cmpabseq()` and `spu_sel()`, these zero elements will be replaced by one. Because logarithm of one is zero, it will not change the result of the entropy.

The other features are calculated following the same method with three above features.

4.8 Implementation of Co-occurrence Matrix and Texture Feature Extraction with Non-zero Elements

As discussed in chapter 3, co-occurrence matrix is usually a parse matrix with a lot of zero elements. To avoid waste time for these elements, several structures have been proposed e.g., linked list, hash table, etc, which stores only non-zero elements of the co-occurrence matrix. Using these structures can reduce the computation time of texture feature extractions, however increases the calculation time of co-occurrence matrix. Therefore, it is suitable for such small-size images whose computation time of texture features is dominant. In section 4.5, 4.6 and 4.7 we use the normal array structure which contains both zero and non-zero elements for the co-occurrence matrix. In this section, we consider the implementation on the Cell of the approach that using only non-zero elements for the co-occurrence matrix. Figure 4.17 describes this implementation on the Cell with 2 SPEs.

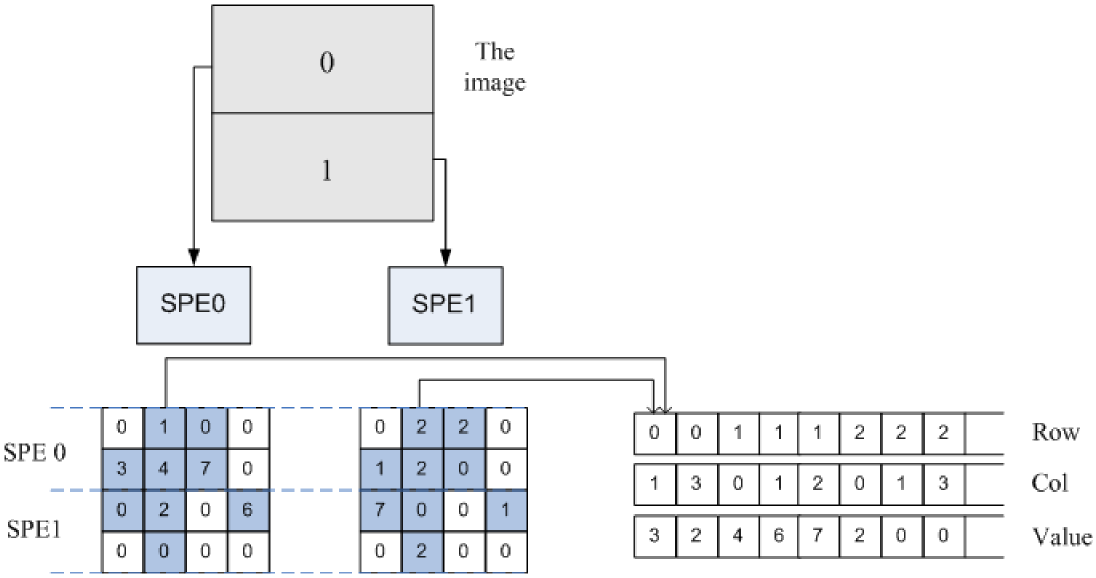


Figure 4.17: Implementation of Co-occurrence Matrix with Non-zero Elements

In the first step, in each SPE, sub-cooccurrence matrix is calculated as normal array approach discussed in section 4.5. In the second step, each corresponding element of these sub-matrices is added. If the sum of elements is zero, it will be skipped. If it is non-zero, the sum, the row and column index of elements will be store in three arrays. These three array have the same number of elements and can be accessed by an index to read the row, the column and the corresponding value of the co-occurrence matrix. In comparison with normal array approach, this approach use more memory since both row and column index must be stored in two arrays. Its execution time is also longer since it needs and additional comparison step of the sum of sub-matrices elements with zero. The result of comparison between two approaches will be showed in the next section.

4.9 Experimental Results and Analysis

4.9.1 Experiment Environment

In previous sections, the parallel strategies as well as the implementation on the Cell of GLCM and texture feature extraction have been discussed in detail. In this section the practical performance of these algorithms is investigated. Firstly, we present and analyze the experimental results of the two parts of the normal array approach: calculating co-occurrence matrix and computing texture features in section 4.9.2 4.9.3. Secondly, in section 4.9.4, the results of non-zero elements approach are showed and compared with the normal approach. Finally, in section 4.9.5, also the performance and efficiency of the Cell processor is compared with modern state-of-the-art x86-based consumer processors from Intel.

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	2MB	4 MB
RAM	2GB	8GB
Operating System	Suse 10	Red Hat 4.1.2
32/64-bit	32-bit	64-bit
Kernel version	2.6.27	2.6.18

Table 4.2: specification of running environment

The input images have follow sizes: 128 × 128, 256 × 256, 512 × 512, 1024 × 1024 and gray levels: 16, 32, 64, 128. The experiments are implemented in PPE, 1 SPE, 2 SPEs, 4 SPEs, 8 SPEs and 16 SPEs.

The development tool is Cell BE SDK 3.1 including GCC compiler 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.

Time is measured using the SPU decrementer. 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 will be read at the beginning and the end of the measured interval. To increase to accuracy, in each experiments, the results are deduced from smallest time of 1000 iterations. In all the measurements, time is displayed in microsecond.

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 second Cell processor shares the memory controller of via the FLEXIO bus. 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.

To compare the Cell processor to modern x86 processors, two x86 based machines are used. The tests are run on an Intel Pentium 4 and Core2 Duo. The machine specifications are listed in Table 4.2.

4.9.2 Implementation of Co-occurrence Matrix

In this section we explain two different discussed techniques, namely scalar and LDT for implementation of co-occurence matrix (normal array approach) on the Cell architecture.

4.9.2.1 Scalar Implementation

Table 4.3 and Figures 4.18 describes the building time of co-concurrence matrix of 4 different 128-gray level images using PPE or 1, 2, 4, 8 and 16 SPEs. The results show that with the image size of 512 × 512 or larger, the building time of co-occurrence decreases when more SPEs are used. Meanwhile with the image size smaller than 512 × 512, time is at the bottom with 4-8 SPEs and grow up after that. This can be explained when we consider components of building time in Figure 4.19. As described in section 4.5.1, there are two steps in building co-concurrence matrix: calculation sub-matrix in each SPEs and adding them. Time of sub-matrix depends on size of the image and declines when more SPEs are used. Time of summing is independent with size of the image and increases following the number of SPEs due to the rising of synchronization time. With small-size images, when using large number of SPEs, the summing time becomes dominant over the time of sub-matrix, that makes the total time increased.

Images Sizes	PPE	1 SPE	2 SPE	4 SPE	8 SPE	16 SPE
128 × 128	1890	986	716	649	785	1010
256 × 256	5740	3495	1957	1295	1034	1363
512 × 512	21497	13520	6977	3733	2338	2024
1024 × 1024	87366	53642	27036	13800	7257	4660

Table 4.3: Scalar implementation of building co-concurrence matrix for different image sizes and the number of gray level is 128

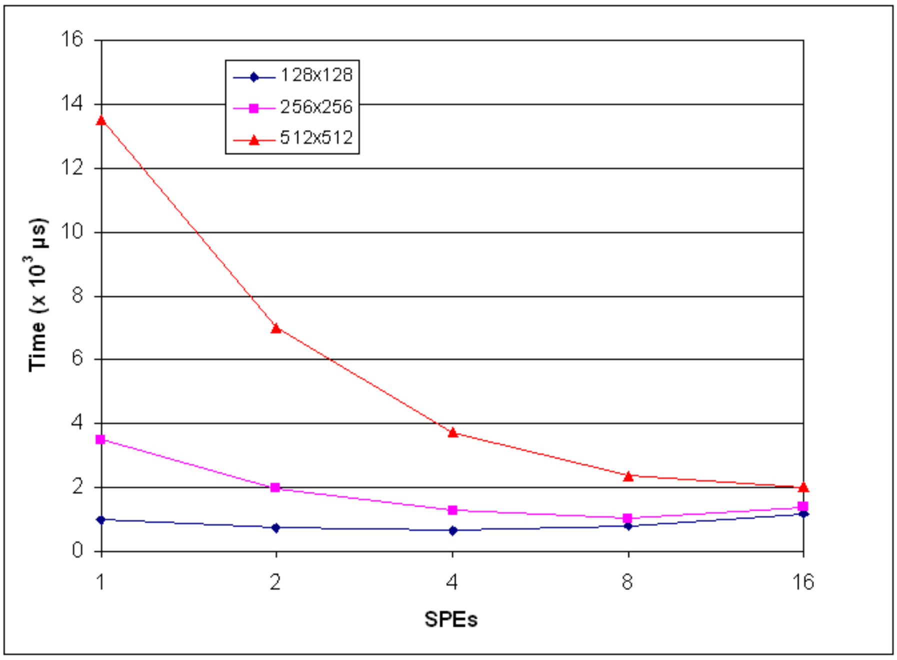


Figure 4.18: Scalar implementation of building co-concurrence matrix for three image sizes: 128 × 128, 256 × 256, 512 × 512 and the number of gray level is 128

Table 4.4 presents the result for the 512 × 512 image with variations of gray-level from 16 to 128. The total time reduces following the deduction of gray-level though the differences are marginal. The decrease of gray-level will make the time for summing sub-matrices drop, however with the image size 512 × 512, the time for computing sub-matrices is dominant and nearly unchanged with variants of gray-level, which is the

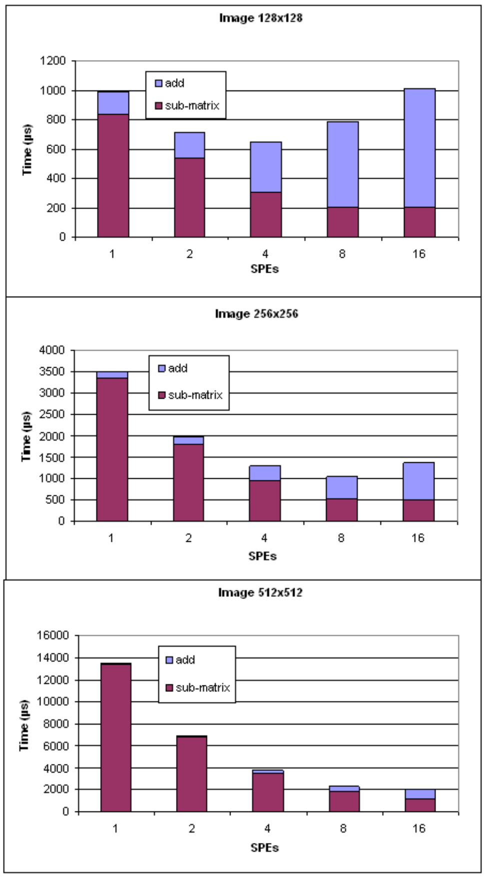


Figure 4.19: Components of building time of co-concurrence matrix for three image sizes: 128 × 128, 256 × 256, 512 × 512 and the number of gray level is 128

reason why the total time only declines small values.

Gray Levels	1 SPE	2 SPE	4 SPE	8 SPE	16 SPE
128	13520	6977	3733	2338	2024
64	13425	6859	3558	2143	1810
32	13395	6748	3525	2050	1791
16	13371	6708	3510	2046	1780

Table 4.4: Building time of co-concurrence matrix with different gray level, image size 512x512

4.9.2.2 Large Data Type Implementation

As explained in section 4.6, in large data type implementation, co-occurrence matrix is defined as vector type. The matrix is updated using SIMD command, therefore the smaller computing time is expected. Table 4.5 shows the result for the image of size 512x512, gray-level 64, using 1, 2, 4, 8 and 16 SPEs. Figure 4.20 compares two ap-

Image	1 SPE	2 SPE	4 SPE	8 SPE	16 SPE
512 × 512	8271	4335	2347	1543	1663

Table 4.5: Large data type implementation of building co-concurrence matrix of the image of size 512 × 512, gray level Ng =64

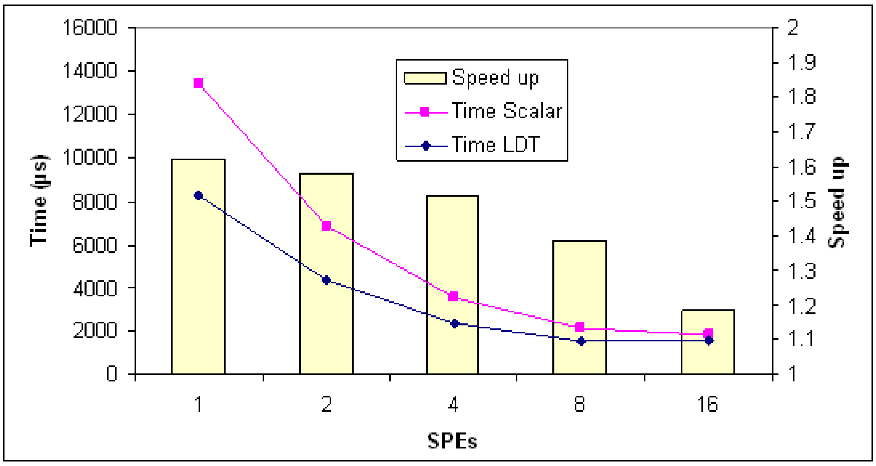


Figure 4.20: Comparison of the implementation of the co-occurrence matrix using scalar and LDT techniques, image size 512 × 512, gray level Ng =64

proaches: scalar and LDT. The speed-up is from 1.2x to 1.6x, reduces when more SPEs are used. In LDT approach, elements of sub-cooccurrence matrix in SPE are defined as vector type, sub-matrix is updated using SIMD instructions, which makes its building time smaller than scalar approach. The next stage, summing of sub-matrices, both approaches implement SIMD instructions, which consumes similar time. When using more cores, time of sub-matrix decreases while time of summing rises, causing the reduction of total speed up.

4.9.3 Implementation of Texture Features

Table 4.6 describes the computation time of 13 texture features of 4 different 128-gray level images using PPE or 1, 2, 4, 8 and 16 SPEs. There are two significant points can be observed from the result. First point is that with the same image size, the computing time of texture features reduces following the number of SPEs N, however, it grows with large N (16 SPEs). This can be explained that when the number of cores is large, the time for synchronization data between them becomes larger than the computing time, which leads to the increase of execution time. Second point is that with different image sizes, the execution time when using the same number of SPEs is almost unchanged, while it varies when using PPE. The reason is that the program for the PPE uses scalar instructions and the computation time depends on the number of non-zero elements of co-occurrence matrix. In the calculation of texture features, we have to use logarithm function and with zero elements, we do not need to calculate their logarithm. With this four image of size 128 × 128, 256 × 256, 512 × 512, 1024 × 1024, the number of non-zero elements is 6214, 9344, 12360 and 13675 respectively (of total 16384 elements of 128×128 co-occurrence matrix). This number of the 1024 × 1024 image is largest, which makes its computation time slowest. The story with SPE is different, using SIMD instructions makes the computation time independent with zero elements of co-occurrence matrix. That is the cause for the similar execution time of different image when using the same number of SPEs.

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

Table 4.6: Execution time of the calculation 13 texture features for different image sizes when the number of the gray level is 128

The calculation time of texture features depends on gray level. It decreases following the decrement of gray level. Table 4.7 and Figure 4.21 show this dependence. The growth of time when using 16 SPEs in comparison with 8 SPEs is caused by rising of synchronization time as explained before.

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	118	211
32	1510	122	98	84	88	175
16	415	82	76	72	88	174

Table 4.7: Execution time of the calculation 13 texture features for different gray levels when the image size is 512 × 512

4.9.4 Implementation of Co-occurrence Matrix and Texture Features with Non-Zero Elements

In all previous sections, we showed the results of normal array approach. In this section, we present the result of non-zero element approach and compare the performance of them. Table 4.8 is the execution time of building co-occurrence matrix of two approaches. It can be seen that time of the non-zero element approach is higher than the normal approach due to it has to compare with zero in searching non-zero elements. Table 4.9 is the execution time of calculating texture features. This time of the non-zero element approach is smaller than the normal approach because it has to calculate for only non-zero elements instead of all ones. However, when we consider the total execution time (Table 4.10 and Figure 4.22), the different between two approaches is negligible. There is a trade-off between the execution time of co-occurrence matrix and texture features, the larger time for co-occurrence matrix accompany the smaller time of texture features. In practice, we can choose one of two approaches depending on applications.

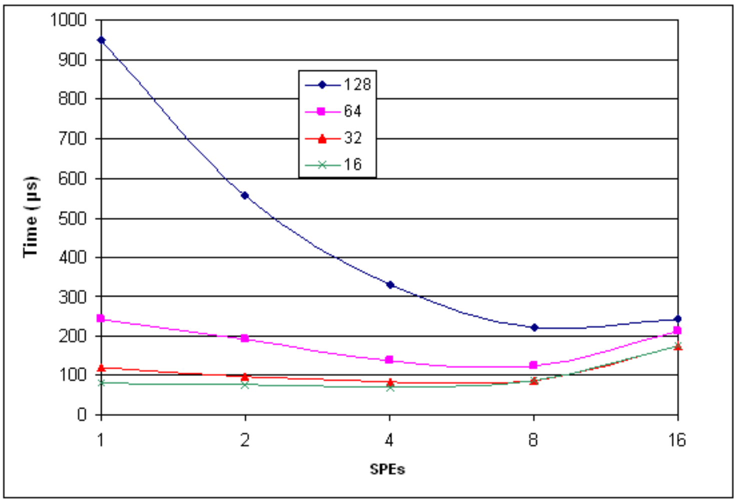


Figure 4.21: Execution time of the calculation 13 texture features for different gray levels when the image size is 512 × 512

Image	1 SPE	2 SPE	4 SPE	8 SPE	16 SPE
128 × 128	**945** 851	**643** 544	**436** 351	**595** 550	**1031** 950
256 × 256	**3448** 3380	**1946** 1835	**1133** 1050	**926** 877	**1217** 1071
512 × 512	**13480** 13425	**7003** 6859	**3643** 3558	**2194** 2143	**1912** 1810
1024 × 1024	**53585** 53530	**27139** 27002	**13637** 13572	**7266** 7229	**4527** 4440

Table 4.8: Execution time of building co-concurrence matrix of non-zero element (in bold text) and normal array approach, the number of gray level is 64

4.9.5 Comparison

In the previous sections the performance of parallel implementation of co-occurrence matrix and texture features have been revealed. In this section the performance results of the Cell processor are compared to the performance results of several x86 processors: a Pentium 4 and Core 2 Duo E6550. The exact specifications of the test platform can be found in Section 4.9.1. For the Cell, 8 SPEs and 16 SPEs are used. 4 images of gray level 64 and 128 are tested. Co-occurrence matrix is calculated using LDT. Table 4.11, 4.12 and 4.13 show the execution time and speed-up (in comparison with Core2 Duo E 6550) of each platforms in calculating co-occurrence matrix and texture features. Figure 4.23 displays graphs of speed-up.

For calculating co-occurrence matrix, it can be seen from Table 4.11 that with small image size (128×128), Core2 Duo is the most efficient. Actually, there is few computation in building co-occurrence matrix, just loads the image data then updates the matrix,

Image	1 SPE	2 SPE	4 SPE	8 SPE	16 SPE
128 × 128	**137**	**88**	**81**	**94**	152
	241	191	136	115	211
256 × 256	**162**	**111**	**84**	**96**	**160**
	242	193	138	116	212
512 × 512	**181**	**141**	**89**	**95**	**164**
	242	193	137	116	211
1024 × 1024	**190**	**143**	**90**	**100**	**165**
	243	192	137	115	213

Table 4.9: Execution time of calculating texture features of non-zero element (in bold text) and normal array approach, the number of gray level is 64

Image	1 SPE	2 SPE	4 SPE	8 SPE	16 SPE
128 × 128	**1082**	**731**	**517**	**689**	**1183**
	1092	735	487	665	1161
256 × 256	**3610**	**2057**	**1217**	**1022**	**1377**
	3622	2028	1188	993	1283
512 × 512	**13661**	**7144**	**3732**	**2289**	**2076**
	13667	7052	3695	2259	2021
1024 × 1024	**53775**	**27282**	**13727**	**7366**	**4692**
	53773	27194	13709	7344	4653

Table 4.10: Total execution time of calculating co-concurrence matrix and texture features of non-zero element (in bold text) and normal array approach, the number of gray level is 64

therefore it is hard for the Cell to demonstrate its superiority. When the workload (image size) increases, the Cell performs efficiently. For example, with the image size of 1024 × 1024, when using the Cell with 16 SPEs, the speed-up is approximately 10x. This speed-up can be more impressive if the workload continues increasing.

For calculating texture features, the speed-up can be seen clearly for all image sizes. Calculating texture features is a computation-intensive process, therefore it can benefit from the advantages of the Cell. With gray level of 64, the speed up of a 1024 × 1024 image using 8 SPEs is 8x [Table 4.12], this rate is 16x when doubling the gray level [Table 4.13]. This performance improvement is really impressive.

When we consider the execution time of texture feature extraction algorithm, which includes both the execution time of co-occurrence matrix and texture features, the speed-up is displayed in Figure 4.23. The speed-up increases following the size of images, it is larger than 9x when using 16 SPEs for the 1024 × 1024 image.

4.9.6 Conclusion

In this chapter, the parallel implementation of the texture feature extraction algorithm on the Cell Architecture is investigated. At first, we introduced specifications of the Cell Architecture and did some performance benchmarks. After that, we studied the parallel

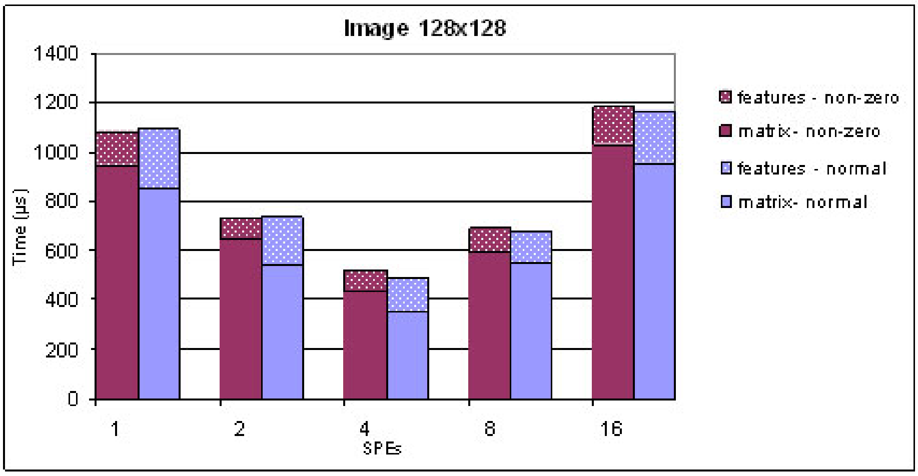


Figure 4.22: Execution time of normal array and non-zero element approach, the image size is 128 × 128, the gray level is 64

Image	Pentium 4	Core2 Duo	Cell 8 SPEs	Cell 16 SPEs
128 × 128	1270 0.48x	612 1x	678 0.90x	1276 0.48x
256 × 256	4894 0.47x	2318 1x	877 2.64x	1402 1.65x
512 × 512	19257 0.46x	8872 1x	1543 5.75x	1527 5.81x
1024 × 1024	67782 0.52	35091 1x	4627 7.58x	3640 9.64x

Table 4.11: Performance(time in μs and speed-up in comparison with Core2 Duo) of co-occurrence matrix with different platforms when gray level is 64

strategies for parallelization the calculation of co-occurrence matrix and texture features, then we implemented them on the Cell. The results showed that the algorithm run much faster in SPE than in PPE, especially when more SPEs are used. The execution time of the algorithm was also compared between different hardware platforms, and the Cell showed as an impressive speed-up in comparison with Core2 Duo as we expected.

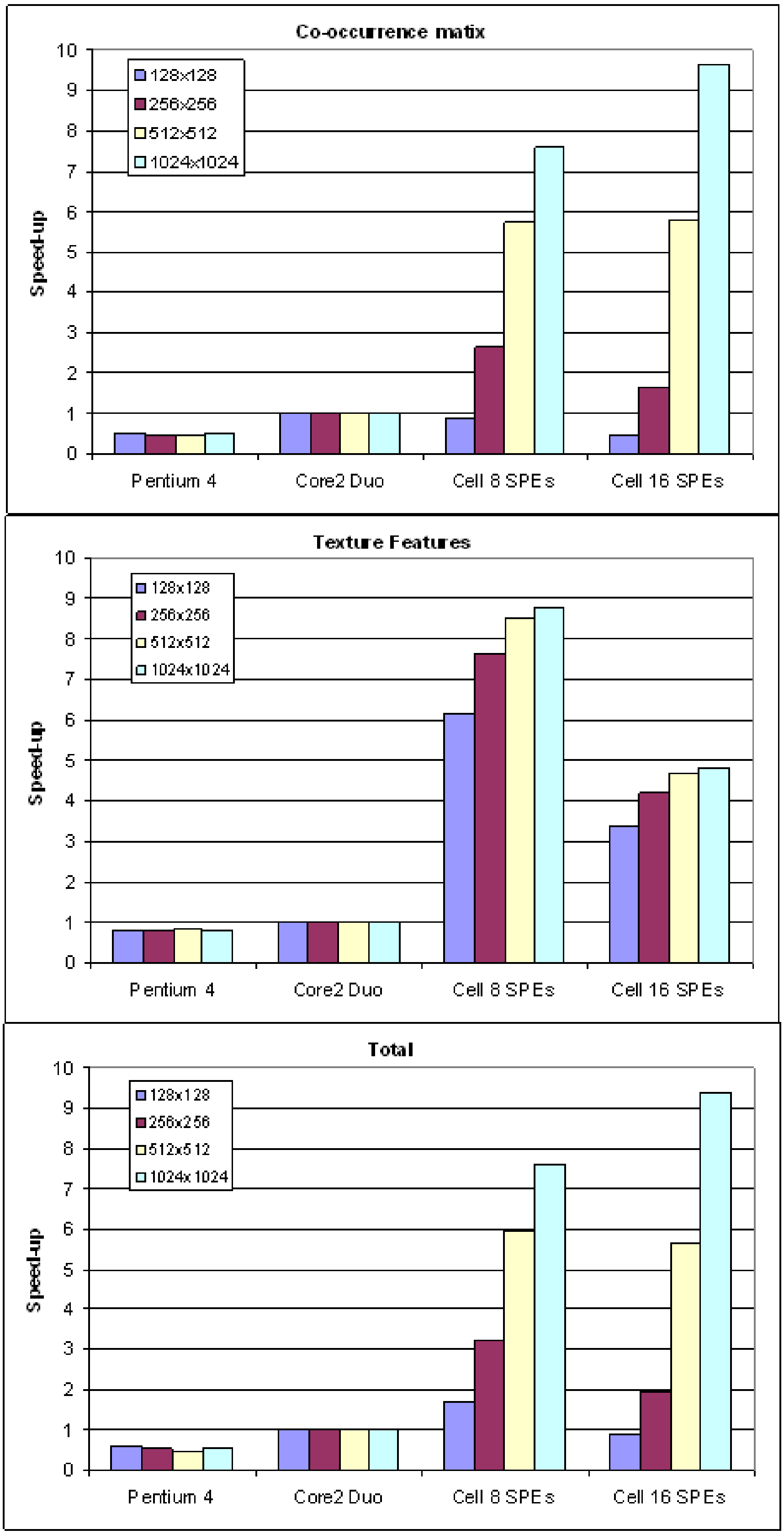


Figure 4.23: Performance speed-up (in comparison with Core2 Duo) of texture feature extraction algorithm (including the calculation of co-occurrence matrix and texture features) when gray level is 64

Image	Pentium 4	Core2 Duo	Cell 8 SPEs	Cell 16 SPEs
128 × 128	885 0.80x	712 1x	116 6.14x	211 3.37x
256 × 256	1082 0.82x	887 1x	116 7.65x	211 4.20x
512 × 512	1167 0.84x	984 1x	116 8.48x	211 4.66x
1024 × 1024	1242 0.82x	1020 1x	116 8.79x	211 4.83x

Table 4.12: Performance(time in μs and speed-up in comparison with Core2 Duo) of texture features with different platforms when gray level is 64

Image	Pentium 4	Core2 Duo	Cell 8 SPEs	Cell 16 SPEs
128 × 128	3090 0.76x	2337 1x	227 10.30x	250 9.35x
256 × 256	3920 0.79x	3082 1x	234 13.17x	250 12.33x
512 × 512	4552 0.79x	3576 1x	223 16.04x	242 14.78x
1024 × 1024	4756 0.80x	3777 1x	227 16.64x	253 14.93x

Table 4.13: Performance(time in μs and speed-up in comparison with Core2 Duo) of texture features with different platforms when gray level is 128

Conclusion and Future Work

5

This chapter presents the conclusions from this thesis. In section 5.1, we provide a summary of the thesis. Future work are proposed in section 5.2.

5.1 Conclusions

In this thesis, we have studied, analyzed and implemented the texture feature extraction algorithm, both sequential and parallel. At first, we studied texture analysis and texture feature extraction algorithms in theoretical, then we chose gray-level co-occurrence matrix algorithm and Haralick's texture features to implement. Experiments showed that GLCM algorithm is very time consuming, especially with images of large sizes and gray-levels. Several methods to optimize co-occurrence matrix by omitting its zero elements have been considered and improved, such as using linked list, hash table or hybrid structures like array - linked list, hast table - linked list, hash table - array. These structures can make the computing process of texture features faster, however, increases the building time of the matrix, for example 60% with hash table-array structure. Therefore, it may not be effective for image of large size. For calculating texture features, we can optimize by combinning features which have the similar computing way in one loop, using loop-unrolling...

With the desire to obtain better results in accelerating GLCM algorithm, we have parallelized the algorithm on the Cell architecture. The Cell is a single-chip multiprocessor with nine processors: one PowerPC Processor Element (PPE) and eight Synergistic Processor Element (SPE) which are optimized for running compute-intensive SIMD applications. For GLCM algorithm, co-occurrence matrix is calculated in parallel by dividing an input image into sub-parts, each parts will be processed simultaneously in one SPE to generate sub matrices. And then these matrices are summed up to create final co-occurrence matrix. In each SPE, due to data dependency issues, non-aligned and irregular data access problems, we can not implement data-parallelism by SIMD instructions. However, by using large data type approach, which is defined each element of co-occurrence matrix as a vector type, we can make use of SIMD instructions. This approach can reduce execution time from 20% to 60% depeding on number of used SPEs. In total, building time of co-occurrence matrix decreases when increasing number of working SPEs. The reduction is impressive in compared to modern X86 architectures such as Pentium 4 or Core2 Duo, especially when the workload(size of input images) is heavy. For example, with an image of size 1024x1024, gray-level of 64, using 16 SPEs, the speed-up is 9.6x.

In computing 13 Haralick's texture features, we have taken advantage of both multicores and SIMD instructions. The parallel strategy is dividing the co-occurrence matrix into smaller ones, each of them is processed in a SPE to generate features. The sum of these features in all SPEs is final values. In each SPE, the features is calculated by SIMD instructions. Features which is stored in the main memory are shared data between cores, therefore we had to make the synchronization process between a SPE and the PPE atomic. Time for this atomic synchronization is considerable when larger number of cores are used and can make the execution time increase. Compare results of the Cell with a Core2 Duo processor, we can see an impressive speed-up, 16x for an image of size 1024x1024, gray-level of 128. If we continue increase the gray-level, the speed-up will be even more impressive.

5.2 Future Work

Following the investigations described in this thesis, the main lines of the research remains open and a number of projects could be taken up:

- Continue to optimization the parallel implementation of the texture features algorithm. We observed that when the workload (image size, number of gray level) is small, the communication time between cores can become dominant, reduce the performance the implementation. Therefore, the optimization should focus on reducing the communication and synchronization time between SPEs.

- Implement the texture features algorithm on other parallel platforms or on FPGAs. In the thesis, we focus on the parallel implementation of the algorithm on the Cell, compare the result with single-core implementation. The comparison will be more appropriated if we can compare the performance between the Cell implementation with other parallel implementations.

- Apply the texture features algorithm on real applications for classifications. In this thesis, we just investigated in the algorithm and did not apply it for any particular application. Therefore, an implementation of the algorithm on a real application such as medical image processing or remote sensing will make the research more applicable.

Bibliography

[1] A. Azevedo and B. Juurlink, *Scalar processing overhead on simd-only architectures*, IEEE International Conference on Application-Specific Systems, Architectures and Processors (2009), 183–190.

[2] C. C. Chi, *Parallel h.264 decoding strategies for cell broadband engine*, Master's thesis, Computer Engineering group, Delft University of Technology, the Netherlands, 2009.

[3] D. A. Clausi and M. Jernigan, *A fast method to determine cooccurrence texture features using a linked list implementation*, Remote Sensing of Environment **36** (1996), 506–509.

[4] ———, *A fast method to determine co-occurrence texture features*, IEEE Trans. on Geoscience and Remote Sensing, vol. 36, 1998, pp. 298–300.

[5] D. A. Clausi and Y. Zhao, *Rapid determination of co-occurrence texture features*, Geoscience and Remote Sensing Symposium **4** (2001), 1880–1882.

[6] A. Arevalo et al, *Programming the cell broadband engine examples and best practices*, IBM, December 2007.

[7] C. S. Carson et al, *Region-based image querying*, IProceedings of IEEE Workshop on Content-Based Access of Image and Video Libraries (1997), 42–49.

[8] E. Niblack et al, *The qbic project: querying images by color, texture and shape*, IBM Research Report (1993), 298–300.

[9] H. Tamura et al, *Textural features corresponding to visual perception*, IEEE Transactions on Systems, Man and Cybernetics **8(6)** (1978), 460–472.

[10] L. M. Kaplan et al, *Fast texture database retrieval using extended fractal features*, 1998, pp. 162–173.

[11] R. M. Haralick, *Statistical and structural approaches to texture*, Proceedings of the IEEE **67** (1979), 786–804.

[12] J. K Hawkins, *Textural properties for pattern recognition*, Picture Processing and Psychopictorics (1969).

[13] IBM, *Synergistic processor unit instruction set architecture, version 1.2*, January 2007.

[14] IBM, *C/c++ language extensions for cell broadband engine architecture, version 2.5*, February 2008.

[15] ———, *Software development kit for multicore acceleration version 3.1: Programming tutorial*, 2008.

[16] D. G. Kendall, *Shape manifolds, procrustean metrics, and complex projective spaces*, Bulletin of the London Mathematical Society **16** (1984), 81–121.

[17] F. Liu and R. W. Picard, *Periodicity, directionality and randomness: Wold features for image modelling and retrieval*, IEEE Transactions on Pattern Analysis and Machine Intelligence (1996), 722–733.

[18] A. Bouridane M. A. Tahir and F. Kurugollu, *An fpga based coprocessor for glcm and haralick texture features and their application in prostate cancer classification*, Analog Integr. Circuits Signal Process. **43** (2005), no. 2, 205–215.

[19] N. Harder M. Gippa, G. Marcus and R. Mnner, *Haralicks texture features using graphics processing units (gpus)*, Proceedings of the World Congress on Engineering, vol. 1, 2008.

[20] M. Schrder M. Schroder and A. Dimai, *Texture information in remote sensing images: A case study*, Workshop on Texture Analysis (1998).

[21] R. Manmatha and S. Ravela, *A syntactic characterization of appearance and its application to image retrieval*, Human Vision and Electronic Imaging II (1997), 484–495.

[22] A. Materka and M. Strzelecki, *Texture analysis methods - a review*, Tech. report, Institute of Electronics, Technical University of Lodz, 1998.

[23] R. Mehrotra and J. E. Gary, *Similar-shape retrieval in shape data management*, IEEE Computer **28(9)** (1995), 57–62.

[24] E. Miyamoto and T. Merryman, *Fast calculation of haralick texture features*, Technical Report, Carnegie Mellon University (2008).

[25] R. S. Poulsen N. G. Nguyen and C. Louis, *Some new color features and their application to cervical cell classifacation*, Pattern Recognition 16 **4** (1983), 401–411.

[26] K. Shanmugam R. M. Haralick and I. H. Dinstein, *Textural features for image classification*, IEEE Transactions on Systems, Man and Cybernetics **3** (1973), 610–621.

[27] M. El-Gabali S. Khalaf and N. Abdelguerfi, *A parallel architecture for co-occurrence matrix computation*, Proceedings of the 36th Midwest Symposium Circuits and Systems **2** (1993), 945 – 948.

[28] J. Sklansky, *Image segmentation and feature extraction*, IEEE Transactions on Systems,Man, and Cybernetics **SMC-8** (1978), 237–247.

[29] J. R. Smith and F. S. Chang, *Automated binary texture feature sets for image retrieval*, In Proc. Int. Conf. on Acoustics, Speech, and Signal Processing (1996), 2241–2246.

[30] D. B. Stewart, *Measuring execution time and real-time performance*, Embedded Systems Conference, Boston (2006).

[31] M. Stricker and A. Dimai, *Color indexing with weak spatial constraints*, Storage and Retrieval for Image and Video Databases IV (1996), 29–40.

[32] M. Stricker and M. Orengo, *Similarity of color images*, Storage and Retrieval for Image and Video Databases III (1995), 381–392.

[33] R. Sutton and E. L. Hall, *Texture measures for automatic classification of pulmonary disease*, IEEE Transactions on Computers **C-21** (1972), 667–676.

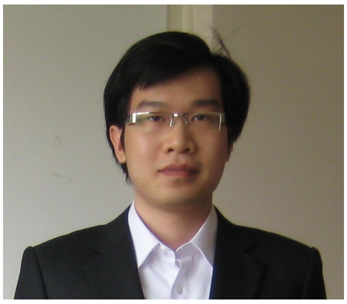
[34] M. J. Swain and D. H. Ballard, *Color indexing*, International Journal of Computer Vision **7(1)** (1991), 11–32.

[35] J. N. Dale T. Chen, R. Raghavan and E. Iwata, *Cell broadband engine architecture and its first implementation: a performance view*, IBM J. Res. Dev. **51** (2007), no. 5, 559–572.

[36] M. Tuceryan and A. K. Jain, *Texture analysis*, In The Handbook of Pattern Recognition and Computer Vision (1998), 207–248.

[37] S. E. Umbaugh, *Computer imaging: Digital image analysis and processing*, Taylor and Francic Group, 2005.

Curriculum Vitae



Tuan Anh Pham

Nationality: Vietnamese

Date of birth : 29, May, 1984

Bachelor of Science in Electrical Engineering at Hanoi University of Technology, 2007