Examination of Speed Contribution of Parallelization for Several Fingerprint Pre-Processing Algorithms

Salih GÖRGÜNOĞLU[1], İlhami Muharrem ORAK[1], Abdullah ÇAVUŞOĞLU[2], Mehmet GÖK[1]
[1]*Department of Computer Engineering, Karabük University, Karabük, Turkey*
[2]*Department of Computer Engineering, Yıldırım Beyazıt University, Ankara, Turkey*
sgorgunoglu@karabuk.edu.tr

***Abstract*—In analysis of minutiae based fingerprint systems, fingerprints needs to be pre-processed. The pre-processing is carried out to enhance the quality of the fingerprint and to obtain more accurate minutiae points. Reducing the pre-processing time is important for identification and verification in real time systems and especially for databases holding large fingerprints information. Parallel processing and parallel CPU computing can be considered as distribution of processes over multi core processor. This is done by using parallel programming techniques. Reducing the execution time is the main objective in parallel processing. In this study, pre-processing of minutiae based fingerprint system is implemented by parallel processing on multi core computers using OpenMP and on graphics processor using CUDA to improve execution time. The execution times and speedup ratios are compared with the one that of single core processor. The results show that by using parallel processing, execution time is substantially improved. The improvement ratios obtained for different pre-processing algorithms allowed us to make suggestions on the more suitable approaches for parallelization.**

***Index Terms*—CUDA, fingerprint recognition, parallel processing, parallel programming, OpenMP.**

I. INTRODUCTION

Unique features of human beings may be listed as: fingerprints, iris, retina, signature, face, DNA, finger vein etc. and are used in biometric systems [1]. These characteristics are sometimes combined together to improve the security of the systems. Fingerprint recognition and verification systems show better results, when compared to other biometric systems.

For a minutiae based fingerprint recognition system, before extracting minutiae, the pre-processing steps are required and are shown in Fig.1. By applying these steps, the target is acquiring accurate features of the fingerprint image taken by a sensor.

For a fingerprint systems used in our daily life, the response time has vital importance and effects the preference of such a system. For the personnel access control systems, the quality of finger print scanned online and the number of personnel affect the response time of the system. Pre-processing consumes most of the processing time in minutiae based fingerprint recognition systems. Especially, for a real time system -requiring fast response time- pre-processing time has to be reduced as much as possible. For this reason, parallel programming methods can be used. In this study, at pre-processing stage of biometric systems, OpenMP and CUDA based parallelization techniques are used with several fingerprint images having different resolutions. OpenMP performs parallel processing on multi core processors, while CUDA based systems distribute parallel operations on GPUs.

The following section is on pre-processing algorithms and it focuses on enhancement methods, such as binarization, thinning, elimination and minutiae extraction. Section three contains a short summary of parallelization methods with OpenMP while section four does the same for CUDA. Section five presents the experimental results of our study and finally, section six concludes this paper.

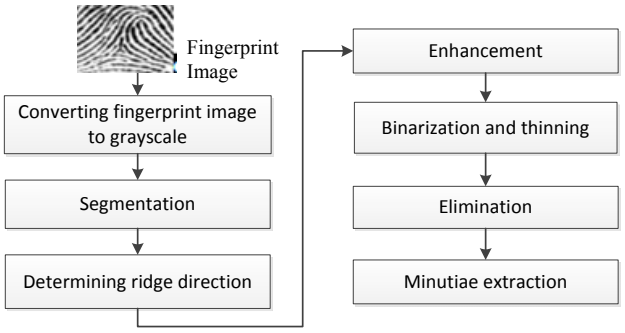

Figure 1. The pre-processing steps used in fingerprint systems

II. PRE-PROCESSING ALGORITHMS

A. Converting Color Image to Gray-Scaled Image

In order to convert the scanned image of a fingerprint, Eq.1 is used.

$$Y = 0.299\, I_R(i,j) + 0.587 I_G(i,j) + 0.144\, I_B(i,j) \quad (1)$$

Following, according to Eq. 2, this value is assigned as RGB component of the relevant pixel in the point (i, j) to obtain it's greyscale correspondence.

$$I_R(i,j) = Y,\ I_G(i,j) = Y,\ I_B(i,j) = Y \quad (2)$$

In these images often, we may get distorted areas such as valleys and ridges -which are not clear- and discontinuities at the ridge flows. To overcome these unwanted defects, fingerprint recognition systems employ enhancement algorithms to improve the performance.

B. Segmentation

Segmentation process is used to separate the fingerprint image from the background image [2]. In the fingerprint

Digital Object Identifier 10.4316/AECE.2014.02001

image, as mentioned earlier, there may be some distorted areas or the regions where the ridges and valleys of the fingerprint are not clear. These unwanted regions are segmented by the algorithm. The segmentation is based on variance value, and here the image is divided into $w \times w$ sized blocks and then the variation of each block is calculated. If a variance of a block is less than a specified threshold, this block is assigned as background according to Eq. 5 which does not contain fingerprint information [3].

These steps are carried out by using following equations where $m(i,j)$ is the mean gray scale value of $w \times w$ sized block and $v(i,j)$ is the variance of the block.

$$m(i,j) = \frac{1}{w.w} \sum_{u=i-w/2}^{i+w/2} \sum_{v=j-w/2}^{j+w/2} I(u,v) \quad (3)$$

$$v(i,j) = \frac{1}{w.w} \sum_{u=i-w/2}^{i+w/2} \sum_{v=j-w/2}^{j+w/2} (I(u,v) - m(i,j))^2 \quad (4)$$

$$C = \begin{cases} 1 & if \quad v(i,j) > Threshold \\ 0 & otherwise \end{cases} \quad (5)$$

C. Determining ridge direction

The directions of the ridges on fingerprints are used for several purposes such as to enhance the quality of the fingerprint image and to classify the fingerprint. The following steps are applied to determine the ridge direction [4]. Again, the fingerprint is divided into $w \times w$ sized blocks. The gradient values in x and y directions of each pixel in the block are calculated by sobel operator of $\partial_x(i,j)$ and $\partial_y(i,j)$ as given below.

$$\partial_x(i,j) = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \partial_y(i,j) = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad (6)$$

The direction of $w \times w$ sized block centred in (i,j) can be calculated with:

$$V_y(i,j) = \sum_{u=i-W/2}^{i+W/2} \sum_{v=j-W/2}^{j+W/2} 2\partial_x(u,v)\partial_y(u,v) \quad (7)$$

$$V_x(i,j) = \sum_{u=i-W/2}^{i+W/2} \sum_{v=j-W/2}^{j+W/2} \partial^2{}_x u,v) - \partial^2{}_y(u,v) \quad (8)$$

$$\theta(i,j) = \frac{1}{2}\tan^{-1}\frac{V_y(i,j)}{V_x(i,j)} \quad (9)$$

where, $\theta(i,j)$ gives the angular direction of the block centered in.

D. Enhancement algorithms.

Enhancement is the process of elimination of noises from the fingerprint. One way of getting rid of the noise is (i.e. after determination of the ridge direction), a mask of (3x9) shown in Fig. 2. is applied to fingerprint image taking the direction of ridge into consideration, by this process, tiny disconnections are repaired and also the ridge is smoothed [5].

Enhancement algorithm consists of following steps:

Transformed coordinates of $I(i,j)$ pixel can be calculated with:

$$j' = j + (u.\cos\theta + v.\sin\theta) \quad (10)$$

$$i' = i + (-u.\sin\theta + v.\cos\theta) \quad (11)$$

where, u and v take (-4,-3,…,3,4) and (-1,0,1) values accordingly. The value of the $I(i,j)$ pixel centered in the mask is obtained as the following.

$$K = \sum_{u=-1}^{1} \sum_{v=-4}^{4} W(u,v) \quad (12)$$

$$I(i,j) = \frac{1}{K} \sum_{u=-1}^{1} \sum_{v=-4}^{4} W(u,v) I(i',j') \quad (13)$$

	-4	-3	-2	-1	0	1	2	3	4
-1	0	1	4	9	16	9	4	1	0
0	25	16	9	4	1	4	9	16	25
1	0	1	4	9	16	9	4	1	0

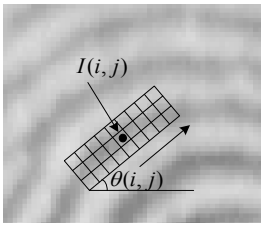

Figure 2. Enhancement mask and its application

E. Binarization

Binarization is the transformation process of image into gray scale. The image is transformed into gray scale after the enhancement process. In this stage, the average value of the local block is taken as threshold for binarization. This process is done by applying following steps.

Fingerprint is divided into $w \times w$ sized blocks where w is taken as 9 (e.g. empirical value). The mean value of gray scale value for block centered in (i,j) can be calculated with:

$$I_{Local\ mean}(i,j) = \frac{1}{w.w} \sum_{u=i-w/2}^{i+w/2} \sum_{v=j-w/2}^{j+w/2} I(u,v) \quad (14)$$

If, $I(i,j) > I_{Local\ mean}(i,j)$ then pixel is signed as white and otherwise as black, i.e.

$$C = \begin{cases} 255(white) & if \quad I(i,j) > I_{Localmean}(i,j) \\ 0\,(black) & otherwise \end{cases} \quad (15)$$

F. Thinning Fingerprint

Thinning process is to express the binarized fingerprint in terms of a single pixel thickness. Thinning process consists of four stages [6, 7]. Firstly, the targeted pixel has to be black. If the pixel is white this step is skipped. Secondly, vertical and horizontal neighbours of the targeted pixel have to be white. Thirdly, at least one neighbour of the pixel has to be black. Finally, while the targeted pixel is to be removed and operation breaks down the continuity then this pixel is kept.

G. Elimination Algorithm

If an enhanced fingerprint image is studied carefully, on fingerprint ridges, disconnection, tiny bifurcation points and small lines in few pixels length can be found. These have to be removed by an elimination algorithm [8,9].

H. Minutiae Extraction

Minutiae extraction includes finding ridge endings and ridge bifurcation points of a fingerprint, and determining

angles and their positions. Black and white crossing number nearby ridge endings is found by using Eq. 16. For example, for the point *P* given in Fig. 3, if *CN=1,* ridge is labelled as end point and if *CN=3* the ridge is bifurcation point [10-13].

$$CN = 0.5 \sum_{i=1}^{8} \left| P_i - P_{i+1} \right| \quad (16)$$

P1	P2	P3
P8	P	P4
P7	P6	P5

Figure 3. Mask used in minutiae extraction

For each minutiae point, the values of x, y, θ are stored in order to be used in matching process. In Fig. 4, outputs of the pre-processing operations in minutiae extraction are shown.

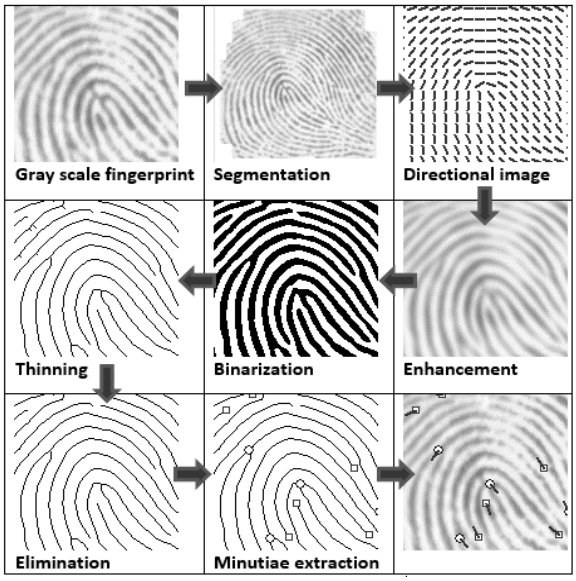

Figure 4. Outputs of the pre-processing operations in minutiae extraction

III. Parallel Processing with OpenMP

Processor manufacturers introduced multi core processors to meet ever increasing demand of computer applications. Nowadays, to improve the performance by using multi core processors become an important research subject. In a multi core system, a task has to be divided into subtasks and each one has to be assigned to different cores. However, since the memory used by these subtasks is shared by all cores, timing has to be properly adjusted. OpenMP is a tool used for parallel processing in shared memory. It is developed in C/C++. It is an application programming library making serial processes to run as parallel [14-16].

By programming OpenMP, especially distributing the target data and processing of each part by a subtask are the main focus. For that reason, loop based data processing is heavily used. In image processing, large number of numerical calculations are carried out. Parallelizing these numerical calculations improve overall processing time [15].

An OpenMP application starts with a task called master thread. Then this task is divided into parallel subtasks. Once the parallel subtasks are completed, the control is returned to master task (Fig. 5). This event is called fork-join operation [16,17].

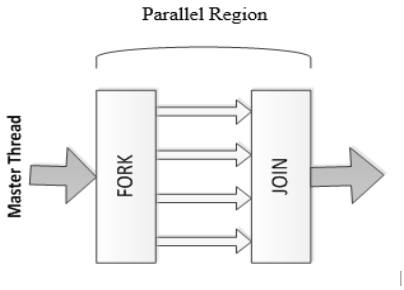

Figure 5. OpenMP programming model

Parallel processing is introduced into a source code as follows:

#pragma omp <instruction> [statement(s)]

Parallel processing flow with OpenMP can be achieved by several instructions. In this study, "parallel for" instruction which utilizes loop level parallel processing is used. Since loop operations are widely used in signal and image processing algorithms, by using parallel processing in loop level reduces the processing time. Selectable statements in instruction can be used in order to organize the loop operation. OpenMP is mostly used in order to make loops run in parallel. With OpenMP, since the targeted loop is divided into subtasks, the data on which loop operation is applied need to be divided appropriately. To achieve this in a loop, the variables which are shared among subtasks and the ones specific to subtasks have to be identified [15-17].

In Algorithm 1, the source code of an algorithm which converts an image data possessing 24-bit color depth into gray scale is shown. In this code, data and iRed2Gray, iGreen2Gray and iBlue2Gray variables are divided among sub tasks. The Z variable, used as loop indices, takes different values for each subtask.

Algorithm 1: Converting image into gray scale

```
1. #pragma omp parallel for private(z)\
2. shared(data,iRed2Gray,iGreen2Gray,iBlue2Gray)
3. for(z = 0; z < h*w*3; z+=3){
4.     data[z] = data[z+1] = data[z+2] =
       ( data[z+0]*iRed2Gray +data[z+1]*iGreen2Gray +
         data[z+2]*iBlue2Gray ) >> 15;
5. }
```

In this example, image data is divided into equal parts to be processed. In the program given in Algorithm 1, image data is accessed by an array. Loop is repeated 786.432 times serially for an image having 512 pixels width and height for 24-bit depth as shown in Fig. 6.

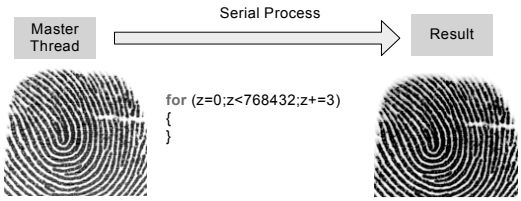

Figure 6. Converting an image into gray scale serially.

For a dual core system, this loop is divided into two sub-parts with fork mechanism of OpenMP. Then each sub-part is computed as a subtask. At the end of the parallel region, the subtasks are terminated and the control passes back to master thread with the join mechanism (Fig. 7). Generating and terminating subtasks and concurrency of them are

managed by OpenMP library. It has to be noticed that these management process brings additional processing time to the processor.

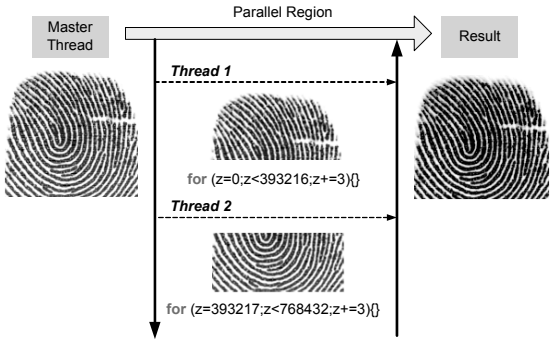

Figure 7. Converting image into gray scale with parallel processing

The relationship among data blocks processed in a loop can make parallelization difficult [17]. In each step, since an independent single pixel is processed, algorithm converting an image to gray scale is suitable for parallelization

IV. Parallel Processing with GPUs

GPUs has the capability of parallel processing along with its ordinary features of graphical processing. For example, NVIDIA GeForce GTS 450 graphics card, has 192 core processors each one having 1566 MHz communication speed with 1 GB DDR3 memory and 128-bits bus. Having this features, GPU can be considered as the most cost effective and powerful parallel processing unit. By taking this capacity of GPUs into consideration, running general purpose applications over them is becoming common. The producers of GPU (such as NVIDIA, AMD, ATI) provide programmable GPUs with high level programming languages in order to fulfil this request [18, 19].

A. NVIDIA CUDA Programming model

NVIDIA developed CUDA programming in 2007 to make parallel programming on GPU easier. CUDA works as a co-processor to increase the capability of main central processing unit (CPU). In this model computer is called as host and graphical interface is called as device. Similarly main memory is called as host memory while memory of GPU is called as device memory. The data to be processed is transferred to device memory by means of the defined functions and after processed results are returned backed to host memory. Hence the some part of the program is computed by GPU and the rest is by CPU itself. This model is known as heterogeneous model [20].

Application code running on GPU is named as kernel process. A single kernel can be processed by several threads which the simple units are running process on GPU. CUDA organize several threads as logical blocks each of which is run by multi-processor on GPU. Since CUDA blocks can consist of limited number of threads, in order to increase the number of threads running concurrently, CUDA blocks are organized in grid structure [18-20].

CUDA is an extension of C and within the access of C/C++ programmer. It supports memory pointers, therefore such a program can randomly access to the memory for write and read operations. Additionally CUDA framework provides memory management system which enable applications to access to GPU core, cache memory and global memory. This structure is shown in Fig. 8.

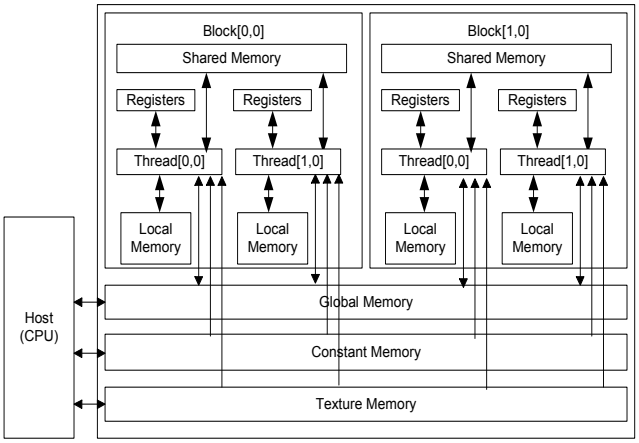

Figure 8. Memory access structure of CUDA [21].

The fingerprint image to be processed by GPU is located into grid form and then it is processed as block by block. In this processing structure each pixel is handled by a thread.

B. CUDA Kernel

CUDA kernel calls are done as

CUDACall <<< GridSize, BlockSize>>>(Parametre1, Parametre2, .., ParametreN)

Where the BlockSize specify the number of threads in a block. GridSize defines the number of the blocks consisted of by the data to be processed. These parameters are chosen according to application requirements and the capacity of GPU. For example, GTS 450 GPU can run maximum 256 threads for a block. Parameters in the call defines the pointers of the data in the global memory and some required by application.

CUDA kernel calls are also named as device code since they are run by GPU. These codes are executed by host code running on the CPU. In Algorithm 2, a device code for converting an image having 24-bits color depth in the memory into gray scale is shown.

Algorithm 2: Kernel code doing gray scale conversion

```
1.  global__ void togray kernel(int w, int h, unsigned char *g_data,
unsigned char *g_result)
2.  {
3.      int row = blockIdx.y * blockDim.y + threadIdx.y;
4.      int col = blockIdx.x * blockDim.x + threadIdx.x;
        // Test if index out of image boundaries
5.  if (row >= h || col >= w) return;
6.      int cIdx = (row * w + col)*3;
7.  g_result[cIdx/3] =          g_data[cIdx]*0.299+
    g_data[cIdx+1]*0.587 + g_data[cIdx+2]*0.114;
8.  }
```

Kernel code in C program is specified by using __global__ extension. This pre-position indicates that related function will be run by GPU. Here, w and h parameters specify the width and height of the data to be processed accordingly. While, g_data and g_result parameters are the pointers for source and target image accordingly and blockIdx, blockDim and threadIdx used in the code are built-in variables defined by CUDA API. By means of these variables, it is possible to identify which pixel is processed [20]. These operations are done on GPU memory (global memory). In Algorithm 3, a part from host

code doing gray scale conversion is shown.

In this study, firstly we have executed a serial program for minutiae based fingerprint processing. Then the loops inside image processing algorithms used prior to minutiae based stage were coded parallel and executed by OpenMP and CUDA based platforms.

This application is developed in Visual C++ 2010 SP1 platform. wxWidgets 2.8.12 is used for visual component library. For parallelization, OpenMP 2.0 library is used. The application is developed on Windows 8 Professional 32bit OS and the PC had an Intel Core i3 550 3.2 GHz, 4 Gb 1333Mhz RAM, intel H55 mainboard, Nvidia Geforce GTS 450 783 MHz and 500 Gb 7200 rpm hard disk.

In Table I, serial and parallel execution times of the programs are given. For the test purposes fingerprints images are taken from the FVC2004 DB4A fingerprint database. Time measurement is done by QueryPerformanceCounter API of Windows in terms of µs. For measuring each processing time, program is stopped and started again.

As it may be observed from Table I, parallel processing done with CUDA provides better performance. In all different sized images, performance improvement in enhancement stage is remarkable and improvement in overall processing time can be considered as noticeable. The contribution of CUDA and OpenMP to performance increase can also be seen from Table I.

To calculate the speedup ratio for the stages, Eq. 16 can be used [22].

$$\text{speedup} = \frac{Tserial}{Tparallel} \tag{16}$$

TABLE I. SERIAL AND PARALLEL EXECUTION TIME OF DEVELOPED PROGRAM FOR DIFFERENT IMAGE RESOLUTIONS

Pre-processing algorithms	**Image size(288x384) Local block size w=24**					**Image size(300x480) Local block size w=24**				
	Execution Time (µs)			**Speed up**		**Execution Time (µs)**			**Speed up**	
	Serial	**openMP**	**CUDA**	**openMP**	**CUDA**	**Serial**	**openMP**	**CUDA**	**openMP**	**CUDA**
Converting grayscale	2895	1299	2815	2.229	1.028	1036	264	1210	3.924	0.856
Segmentation	5169	2554	10925	2.024	0.473	3530	596	4489	5.923	0.786
Ridge direction	154541	85371	24847	1.810	6.220	130019	126825	11122	1.025	11.690
Enhancement	109927	48230	7081	2.279	15.524	43378	24030	2796	1.805	15.514
Binarization	146499	47257	14497	3.100	10.105	77392	23644	8123	3.273	9.528
Thinning	24073	15986	40435	1.506	0.595	11908	8042	17936	1.481	0.664
Elimination	159704	78960	133679	2.023	1.195	105229	83583	52288	1.259	2.012
Total	**602808**	**279657**	**234279**	**2.156**	**2.573**	**372492**	**266984**	**97964**	**1.395**	**3.802**
Pre-processing algorithms	**Image size(328x364) Local block size w=24**					**Image size(640x480) Local block size w=24**				
	Execution Time (µs)			**Speed up**		**Execution Time (µs)**			**Speed up**	
	Serial	**openMP**	**CUDA**	**openMP**	**CUDA**	**Serial**	**openMP**	**CUDA**	**openMP**	**CUDA**
Converting grayscale	1300	857	1120	1.517	1.161	959	275	1028	3.487	0.933
Segmentation	3888	3684	5155	1.055	0.754	2723	479	4022	5.685	0.677
Ridge direction	140897	55327	13819	2.547	10.196	98189	87814	12646	1.118	7.764
Enhancement	53985	24867	3584	2.171	15.063	41661	18577	2768	2.243	15.051
Binarization	109278	31650	9479	3.453	11.528	74958	22354	8145	3.353	9.203
Thinning	48135	15117	24882	3.184	1.935	12192	5516	19944	2.210	0.611
Elimination	119065	67648	77303	1.760	1.540	109066	79779	62048	1.367	1.758
Total	476548	199150	135342	2.393	3.521	339748	214794	110601	1.582	3.072

In Eq. 16, *Tserial* is the execution time of the serial algorithm and *Tparalel* is the execution time of the parallel algorithm. By this formula, speedup ratio of enhancement stage for CUDA and OpenMP in the 640x480 sized image can be calculated as 41661/2768=15.051 and 41661/18577=2.243 respectively.

This value for CUDA is the highest speedup ratio among the pre-processing stages. The average speedup ratio for CUDA is about 3.42. For the stages namely converting grayscale, segmentation and thinning, the speedup ratios for CUDA are less than 1. The reason for this is total execution time for serial algorithm is very short. Besides that, the transfer time of the data between host and device are higher comparing to serial running time. For example, gray scale conversion; serial execution time is 959 µs while Host-to-Device transfer time 273 µs and Device-to-Host transfer time 271 µs. Therefore, the cost of parallel processing on CUDA obviously become higher. Same reason can be concluded for other algorithms having similar speedup ratio. On the other hand, for the ridge direction, enhancement, binarization and elimination algorithms, serial execution time is higher than the transfer time. This leads better performance on CUDA application.

On the other hand average speedup ratio for OpenMP is 1.88. By using Eq. 16 and considering all speedup ratios calculated, the lowest and highest speedup ratio for OpenMP is achieved for the determining ridge direction and segmentation stage with the value of 1.118 and 5.685 respectively.

Fig. 9 shows the execution times in bar chart and Fig.10 shows the speedup ratio of pre-processing algorithms in the 640x480 sized image. It can be seen from Fig. 10 that speedup ratio of CUDA is higher than that of Open MP.

The fingerprints used in the algorithms are taken from FVC2004 fingerprint database. Images stored in the database are taken from different sensors such as optic, thermal or capacitive. Hence, each image has different quality. Different images having same resolution can have different speed up values in parallel computation due to the image quality. The number of ridge bifurcation and ridge ending can be higher in low quality images. This can result in poor performance in parallel algorithms.

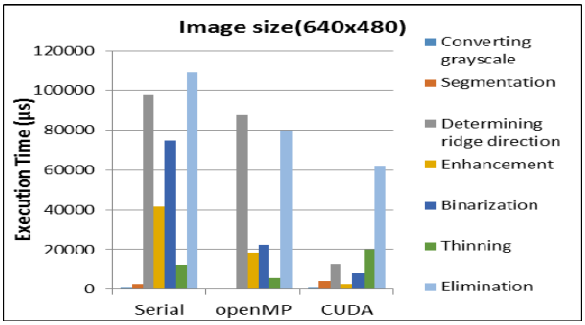

Figure 9. Pre-processing time for serial and parallel processing

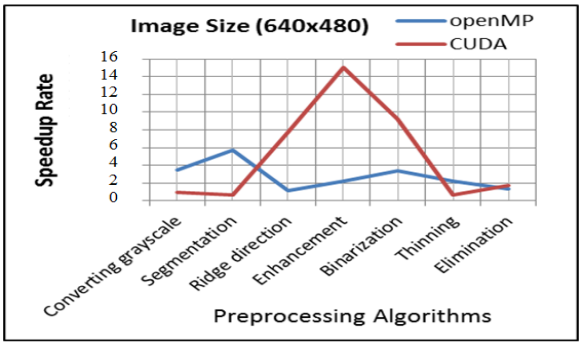

Figure 10. The speedup ratio of pre-processing algorithms.

V. Conclusions

Parallel programming techniques such as CUDA and OpenMP provides various methods to improve the performance of algorithms. In this study, we have examined the effect of parallelization in the pre-processing stages used in minutiae based fingerprint analysis. The execution times of the serial and parallel versions (i.e. two approaches) of these pre-processing algorithms are analyzed and the best contribution in terms of parallelization towards the stages of pre-processing has been shown. Experimental results show that parallel processing can be affectively used for processing fingerprint images especially when dealing with large databases.

The number of the cores in current multi-core CPUs is 2, 4 and 6 and increasing. With the increase of cores, OpenMP can yield better performance. On the other hand, CUDA shows higher speedup rates compared to OpenMP. However, transferring data between CPU and GPU has degrading affect on CUDA performance especially for low serial execution time. Hence, for the application having high the data size enables CUDA for better speedup ratios. However, the performance of the parallel algorithms depends on possibility of parallelization of the algorithm as well as data size. Therefore, if branching, transferring data between host and device and using results in another part are done more often, it will lead worse performance in CUDA. These effects can be observed in some of the pre-processing algorithms. Other reason affecting the performance is the quality of fingerprint image.

Based on the results two open areas will be focused on in future studies. Since only global memory access is used in CUDA programming in this study. Texture Memory and Shared Memory having fast access rate by GPU can be studied and the results can be compared with the global memory approach. On the other hand experimental results prove that most of the pre-processing algorithms can be effectively run on GPU while few of them do not bring so much benefit due to their non-parallel structure. Therefore, for pre-processing of fingerprint images, hybrid solution can be suggested for using both CPU and GPU together by distributing algorithms accordingly. The implementation of hybrid system based on the knowledge gathered in this study will be considered in future studies.

References

[1] A. Venckauskas, N. Morkevicius, K. Kulikauskas, "Study of finger vein authentication algorithms for physical access control", Elektronika Ir Elektrotechnika (Electronics and Electrical Engineering), vol. 121, no. 5, pp. 101-104, 2012.

[2] S. Görgünoğlu, A. Çavuşoğlu, "A Fast and Simple Algorithm for Fingerprint Segmentation", Engineering Science and Technology an International Journal (JESTECH) (formerly TEKNOLOJI), vol. 11, no. 2, pp. 87-92, 2008.

[3] B. M. Mehtre, B. Chatterjee, "Segmentation of fingerprint images–A composite method", Pattern Recognition, vol. 22, no.4, pp. 381–385, 1989.

[4] L. Hong, Y. Wan, A. Jain, "Fingerprint image enhancement: algorithm and performance evaluation", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 20, no. 8, pp. 777-789, 1998.

[5] A. Çavuşoğlu, S. Görgünoğlu, "A fast fingerprint image enhancement algorithm using a parabolic mask", Computers & Electrical Engineering, vol. 34, no. 3, pp. 250-256, 2008.

[6] S. W. Smith, The Scientist and Engineer's Guide to Digital Signal Processing, San Diego: California Technical Publishing, pp. 436-442, 1999.

[7] V. Espinosa-Duro, "Fingerprints thinning algorithm", Aerospace and Electronic Systems Magazine IEEE, vol. 18, no. 9, pp. 28–30, 2003.

[8] M. Tico, P. Kuosmanen, "An algorithm for fingerprint image postprocessing", Thirty-Fourth Asilomar Conference on Signals, Systems and Computers, vol. 2, pp. 1735 – 1739, 2000.

[9] Q. Xiao, H. Raafat, "A combined statistical and structural approach for fingerprint", IEEE International Conference on image postprocessing, Systems, Man and Cybernetics, 331-335, 1990.

[10] J. C. Amengual, A. Juan, J. C. Perez, F. Prat, S. Saez, J. M. Vilar, "Real-time minutiae extraction in fingerprint images", Sixth International Conference on Image Processing and Its Applications, vol. 2, pp. 871 – 875, 1997.

[11] S. Kasei, M. Deriche, B. Boashash, "Fingerprint Minutiae Exraction using block-direction on reconstructed images", TENCON '97. IEEE Region 10 Annual Conference. Speech and Image Technologies for Computing and Telecommunications, vol. 1, pp. 303-306, 1997.

[12] Z. Shi, V. Govindaraju, "A chaincode based scheme for fingerprint minutiae exraction", Pattern Recognition Letters, vol. 27, no. 5, pp. 462-468, 2006.

[13] M. Gamassi, V. Piuri, F. Scotti, "Fingerprint local analysis for high-performance minutiae extraction", IEEE International Conference on Image Processing ICIP 2005, vol. 3, pp. III - 265-8, 2005.

[14] T. Xionggang, C. Jun, "Paralel Image Processing with OpenMP", The 2nd IEEE International Conference on Information Management and Engineering (ICIME), pp. 20-23, 2010.

[15] Y. Liu, F. Gao, "Parallel implementations of image processing algorithms on multi core", Fourth Internatinal Conference on Genetic and Evolutionary Computing, pp.71-74, 2010.

[16] E. Ramaraj, A. S. Rajan, "Median filter using open multiprocessing in agriculture", IEEE 10th International Conference on signal processing ICSP2010, pp. 42-45, 2010.

[17] H. Cao, X. Gu, "OpenMP Parallelization of Jacquin Fractal Image Encoding", 2010 International Conference on E-Product E-Service and E-Entertainment (ICEEE), pp. 1-4, 2010.

[18] S. Park, S. Ponce, J. Huang, Y. Cao, F. Quek, "Low-Cost, High Speed Computer Vision Using NVIDIA's CUDA Architecture", 37th IEEE Applied Imagery Pattern Recognition Workshop, 1-4 (2008)

[19] L. Pan, L. Gu, J. Xu, "Implementation of Medical Image Segmentation in CUDA", Proceedings of the 5th International Conference on Information Technology and Application in Biomedicine, 1-2 (2008)

[20] CUDA Programming Guide Version 4.2. NVIDIA Corporation Santa Clara, California, 4-12 (2012)

[21] Y. Li, K. Zhao, X. Chu, J. Liu, "Speeding up k-Means algorithm by GPUs", Journal of Computer and System Sciences, vol. 79, pp. 216-229, 2013.

[22] S. Akhter, J. Roberts, Multi-Core Programming, Intel Press, Hillsboro, pp. 13-14, 2006.