J Supercomput (2018) 74:2255–2275
https://doi.org/10.1007/s11227-017-2168-6

Real-time parallel image processing applications on multicore CPUs with OpenMP and GPGPU with CUDA

Semra Aydin[1] · **Refik Samet**[2] · **Omer Faruk Bay**[1]

Published online: 12 December 2017

Abstract This paper presents real-time image processing applications using multicore and multiprocessing technologies. To this end, parallel image segmentation was performed on many images covering the entire surface of the same metallic and cylindrical moving objects. Experimental results on multicore CPU with OpenMP platform showed that by increasing the chunk size, the execution time decreases approximately four times in comparison with serial computing. The same experiments were implemented on GPGPU using four techniques: (1) Single image transmission with single pixel processing; (2) Single image transmission with multiple pixel processing; (3) Multiple image transmission with single pixel processing; and (4) Multiple image transmission with multiple pixel processing. All techniques were implemented on GeForce, Tesla K20 and Tesla K40. Experimental results of GPU with CUDA platform showed that by increasing the core number speedup is increased. Tesla K40 gave the best results of 35 and 12 (for the first technique), 36 and 13 (for the second technique), 54 and 16 (for the third technique), 71 and 17 (for the fourth technique) times improvement without and with data transmission time in comparison with serial computing. As a result, users are suggested to use Tesla K40 GPU and Multiple image transmission with multiple pixel processing to get the maximum performance.

✉ Semra Aydin
semraaydin@gazi.edu.tr; semra.avar@gmail.com

Refik Samet
samet@eng.ankara.edu.tr

Omer Faruk Bay
omerbay@gazi.edu.tr

1 Gazi University, Ankara, Turkey

2 Ankara University, Ankara, Turkey

Keywords Parallel computing · Real-time image processing · Image segmentation · Thresholding · Multicore programming · GPU programming

1 Introduction

Image processing requires long time, which is tightly limited in the real-time applications [1,2]. Processing time increases depending on both the number and resolution of the images [3]. The problem is that the serial image processing does not satisfy the real-time conditions [3]. Parallel computing techniques, especially multicore and multiprocessor technologies, should be used to solve this problem [4]. Parallel algorithms are much more complex than serial ones. Generally, the parallel algorithms are designed by modification of serial algorithms [5]. Obtained parallel algorithms can be improved and accelerated by taking into consideration the hardware that the algorithms will work with.

Segmentation is one of the steps in image processing. Thresholding is widely used for this aim. In real-time applications, multicore CPUs and GPGPU should be used to execute thresholding on many images covering the entire surface of the same metallic and cylindrical moving object to satisfy real-time conditions. A multicore CPU is a single computing component with two or more independent actual central processing units (called cores) [6,7]. Pthreads, OpenMP (Open Multiprocessing), TBB (Threading Building Blocks) and Cilk are API (Application Programming Interfaces) to efficiently use the capacity of a multicore CPU [8]. In this paper, a general-purpose and platform-independent OpenMP that supports shared memory for multiprocessing programming in C, C++ and FORTRAN will be used.

A Graphic Processing Unit (GPU) is a Single Instruction stream and Multiple Data streams (SIMD) architecture where the same instruction is performed on all data elements in parallel. On the other hand, the pixels of an image can be considered as separate data elements. So, GPU is a suitable architecture to process data elements of an image in parallel [9]. General-Purpose computing on Graphics Processing Units (GPGPU) is a tool to increase the utilization of GPU. There are many platforms to efficiently use the capacity of GPGPU, such as CUDA, DirectCompute and OpenCL. The CUDA platform, which is the most common one, will be used in this paper [10].

This paper shows that more efficient algorithms and techniques still need to be developed to improve the performance of real-time image processing applications. One of the aims of this study is to make a contribution to this area using OpenMP and CUDA. To this end, bi-level thresholding is implemented on the images covering the entire surface of the same metallic and cylindrical moving object in parallel with the five following techniques. One technique is related to CPU programming with the OpenMP platform. In this context, shared-memory multicore programming with OpenMP, scheduling threads on cores with different parameters, and performance related to the execution time are analyzed. The other four techniques are related to GPU programming with the CUDA platform:

1. Single Image Transmission with Single Pixel Processing (SISP) in which the images are transmitted from CPU to GPU one by one and the pixels of the images are processed one pixel per core of GPU;

2. Single Image Transmission with Multiple Pixel Processing (SIMP) in which the images are transmitted from CPU to GPU one by one and the pixels of the images are processed multipixels per core of GPU;
3. Multiple Image Transmission with Single Pixel Processing (MISP) in which the multiple images are combined and transmitted from CPU to GPU as a single data unit and the pixels of the images are processed one pixel per core of GPU;
4. Multiple Image Transmission with Multiple Pixel Processing (MIMP) in which the multiple images are combined and transmitted from CPU to GPU as a single data unit and the pixels of the images are processed multipixels per core of GPU.

Performance analysis related to execution time was performed by comparison of the results obtained by these techniques with serial computing. The technique with multicore CPU showed that, by increasing the chunk size, the execution time decreases approximately four times. All techniques with GPU were implemented on GeForce, Tesla K20 and Tesla K40. Tesla K40 gave best results of 35 (for SISP technique), 36 (for SIMP technique), 54 (for MISP technique) and 71 (for MIMP technique) time improvement in comparison with serial computing.

The rest part of the paper is organized as follows. In Sect. 2, some related works are presented. In Sect. 3, the real-time image processing techniques are proposed. Section 4 describes the image transmission techniques between CPU and GPU. The experimental results are given in Sect. 5. Section 6 concludes with the main findings.

2 Related works

Multicore CPU and GPGPU technologies are widely used for non-real-time and real-time image processing applications. It is well known that the multithreading, multicore and GPU architectures have advantages in comparison with serial computing [11]. A short literature review related to these technologies is given below.

Thapliyal and Arabnia in their works [12–17] discuss a historical perspective and relevant context about how hardware and software can work in concert on scalable multiprocessor systems with a number of illustrative examples and applications in imaging science. In fact, the proposed imaging architecture presented in these works can be considered to be early designs of GPU processor architectures.

There are many studies reported in the literature related to non-real-time image segmentation using the threshold technique [18,19]. The performance was and still remains an urgent issue to be solved in real-time image processing applications. To this end, different algorithms and techniques have been developed for serial computing [20–22]. Despite some performance improvements in these works, it is very difficult to satisfy real-time conditions by serial computing. Researchers have looked into alternative solutions and found the multicore CPU and GPGPU technologies to solve this issue. At the same time, in order to efficiently use these technologies, different platforms, such as OpenMP and CUDA, have been developed and widely used. For example, OpenMP platform has been used in multithread image processing and image segmentation applications with multicore computing [23]. CUDA platform has been used for parallel image segmentation by region growing, watershed and Otsu binarization algorithms on GPU [24–27]. The reduction sweep algorithm was used

for image segmentation on both CPU and GPU [28]. In [29], several techniques for image segmentation were implemented using CUDA and GPU and processing time was accelerated about 20 times. The authors of [30] present the results of image segmentation on a video with a frame rate of 30 Hz using CUDA and GPU. Despite existing works, in order to satisfy the need for higher speed and low cost more efficient techniques and algorithms are needed. This paper tries to meet to this need.

To accelerate the image thresholding, in existing works images are transferred to the GPU one by one and each pixel is processed in separate cores. In the proposed paper, the images are combined and transmitted and multiple pixels are processed in one core. Due to these contributions, a higher acceleration rate is obtained.

3 Real-time image processing techniques

In this section, we present three techniques: (1) Serial thresholding (Sect. 3.1); (2) Parallel thresholding on a multicore CPU with OpenMP (Sect. 3.2); and (3) Parallel thresholding on a GPU with CUDA (Sect. 3.3). The final one is divided into four techniques which are SISP (Sect. 3.3.1), SIMP (Sect. 3.3.2), MISP (Sect. 3.3.3) and MIMP (Sect. 3.3.4).

Real-time applications of this study are related to the inspection of certain defects on the entire surface of metallic and cylindrical objects. Images taken from the entire surface of the same metallic and cylindrical moving object were used to inspect the defects in real time. In order to detect certain defects of a single object, the image processing steps should be processed on K images covering its entire surface. Time is limited in given applications. In this paper, only the first step of image processing related to image segmentation will be handled. Thresholding is the simplest and a fast way for image segmentation. Parallel programming techniques, such as multicore and multiprocessing technologies, were used to speed up the thresholding of the metallic and cylindrical object.

Firstly, serial thresholding is described. Then, parallel thresholding on a multicore CPU with OpenMP is presented. Finally, parallel thresholding on GPU with CUDA is discussed.

3.1 Serial thresholding

Image segmentation is the process of dividing the individual elements of an image into a set of groups so that all elements in a group have a common property. Segmentation allows visualization of the structures of interest, removing unnecessary information [31]. Thresholding is the simplest, most commonly used and the most popular technique for segmentation. Thresholding techniques can be classified into two categories: bi-level and multilevel. In this paper, bi-level segmentation is used for the segmentation of objects and the background [19]. Thresholding is often used as a preprocessing step, followed by other post-processing techniques [32]. Let us denote by $g(x, y)$ the segmented image obtained from $f(x, y)$. If we consider T as the threshold value, the resulting image will be given by following expression.

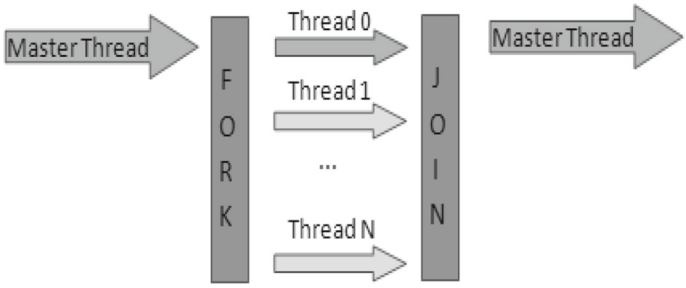


Fig. 1 Thread organization with OpenMP

$$g(x, y) = \begin{cases} 255, & \text{if } f(x, y) \geq \text{T} \\ 0, & \text{if } f(x, y) < \text{T} \end{cases} \tag{1}$$

According to serial thresholding, Eq. (1) should be calculated on each pixel of (x, y) of an original image of $f(x, y)$, where $x = 1, 2, \ldots, N$ and $y = 1, 2, \ldots, M$. The performance or processing time of serial thresholding is defined as follows:

$$t_{\text{ST}} = N * M * \Delta t, \tag{2}$$

where t_{ST} is the processing time of serial thresholding and Δt is the processing time for thresholding on one pixel.

3.2 Parallel thresholding on a multicore CPU with OpenMP

In order to accelerate the thresholding process to satisfy the real-time conditions, the shared-memory multicore programming with OpenMP is proposed. An OpenMP platform always begins with a single thread of control, called the master thread, which exists during the run time of the program (Fig. 1). The master thread may encounter parallel regions, in which the master thread will fork the new threads, each with its own stack and execution context. At the end of the parallel region, the forked threads will be terminated, intermediate results will be joined, and the master thread will continue the program execution as shown in Fig. 1.

To achieve the optimal performance in multithread applications, different scheduling types and chunk sizes should be tested. With OpenMP, static, dynamic and guided scheduling mechanisms can be specified. Static scheduling divides the loop into equal-sized chunks or as equal as possible in the case when the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size. Dynamic scheduling uses the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue. By default, the chunk size for dynamic scheduling is 1. Guided is similar to dynamic scheduling, but the chunk size starts off large and decreases to better handle the load imbalance between iterations. The optional chunk parameter specifies the minimum chunk size to use. By default, the optimal (S) chunk size for guided scheduling is defined as follows:

$$S = \frac{N_L}{N_T}, \tag{3}$$

where N_L is a number of operations in the loop and N_T is a number of threads.

The processing time of parallel thresholding with OpenMP (t_{MP}) is defined as follows:

$$t_{MP} = \frac{t_{ST}}{N_T} + t_0 = \frac{N * M * \Delta t}{N_T} + t_0, \tag{4}$$

where t_0 is a processing time for fork and join of threads. One of factors that effects t_0 is chunk size of S.

3.3 Parallel thresholding on a GPU with CUDA

The CUDA platform consists of functions, called kernels, which can be executed simultaneously by a large number of threads on the GPU. Threads are grouped into warps. A warp consists of 32 threads which are executed as SIMD architecture independently. Threads within a warp execute the same instruction on different data elements in parallel [33].

In order to parallelize the thresholding process, the kernels should be used. To organize kernels to work in parallel, streams are used (Table 1).

As shown in Table 1, firstly, the K streams are defined (Line 1) and created (Lines 2, 3). Then, data (images) for created streams are transmitted asynchronously from the CPU to the GPU (Lines 4, 6). After that, kernels execute the same instructions on K images asynchronously (Lines 5, 7). Finally, the results are transmitted from the GPU to the CPU (Lines 8, 9).

Images can be sent from the CPU to the GPU one by one or in a combined data array. Images can be processed in the cores of the GPU as one pixel by one pixel or in multipixels. Results can be returned from the GPU to the CPU one by one, or in a combined data array. Algorithm 1 for sending K images from the CPU to the GPU one by one, processing them in GPU and returning the results from the GPU to the CPU one by one is given in Table 2. Algorithm 2 for sending K images from the CPU to the GPU in a combined data array, processing them in the GPU and returning the results from the GPU to the CPU in a combined data array is given in Table 3. Algorithm 3 for distributing and processing the images as one pixel per core of the GPU is given in Table 4. Algorithm 4 for distributing and processing the images as P pixels per core of the GPU is given in Table 5.

Four techniques are proposed to execute thresholding on the GPU with CUDA: (1) SISP; (2) SIMP; (3) MISP, and (4) MIMP (Table 6).

3.3.1 SISP technique

In this technique, the images are transmitted from the CPU to the GPU one by one and results are returned from the GPU to the CPU one by one using the proposed Algorithm 1 (Table 2). Also, the pixels of the images are distributed and processed one pixel per core of the GPU using the proposed Algorithm 3 (Table 4).

Table 1 Multiple kernel organization by streams

Line	Code	Explanation
1	cudaStream_t stream1, stream2, …, stream K	The K streams are defined. K is the number of images
2	cudaStreamCreate (& stream1)	Stream 1 is created
…		
3	cudaStreamCreate (& stream K)	Stream K is created
4	cudaMemcpyAsync (dst, src, size, dir, stream1)	Image 1 is transmitted asynchronously from the CPU to the GPU
5	kernel<<< grid, block, 0, stream1 >>> ()	Kernel works asynchronously
…		
6	cudaMemcpyAsync (dst, src, size, dir, stream K)	Image K is transmitted asynchronously from the CPU to the GPU
7	kernel<<< grid, block, 0, stream K >>> ()	Kernel works asynchronously
8	cudaMemcpyAsync (dst, src, size, dir, stream1)	The result of stream 1 is transmitted from the GPU to the CPU
…		
9	cudaMemcpyAsync (dst, src, size, dir, stream K)	The result of stream K is transmitted from the GPU to the CPU

Table 2 Algorithm 1: Single image transmission

Step	Description
1	Send the 1st image from CPU to GPU
2	Execute the thresholding kernel on the 1st image
3	Send the 2nd image from CPU to GPU
4	Execute the thresholding kernel on the 2nd image
…	
$2K-1$	Send the Kth image from CPU to GPU
$2K$	Execute the thresholding kernel on the Kth image
$2K+1$	Return the processing result of the 1st image from GPU to CPU
…	
$3K$	Return the processing result of the Kth image from GPU to CPU

3.3.2 SIMP technique

In this technique, the images are transmitted from the CPU to the GPU one by one and the results are returned from the GPU to the CPU one by one using the proposed Algorithm 1 (Table 2). Also, the pixels of the images are distributed and processed as multi pixels per GPU core using the proposed Algorithm 4 (Table 5). Pixels of images are distributed among cores of GPU as P pixels per core. The number of pixels per core depends on GPU hardware.

Table 3 Algorithm 2: Multiple image transmission

Step	Description
1	Combine K images in a data array
2	Send the combined data array from CPU to GPU
3	Execute the thresholding kernel on K images
4	Return the combined processing results of K images from GPU to CPU
5	Separate the images from combined results

Table 4 Algorithm 3: Single pixel processing in the GPU

Step	Description
1	Distribute pixels as one pixel per core of GPU
2	If pixel value $\geq T$ then the result is 255
3	If pixel value $< T$ then the result is 0
4	Repeat Step 2 and Step 3 for all pixels

Table 5 Algorithm 4: Multi pixel processing in the GPU

Step	Description
1	Distribute pixels as P pixels per core of GPU
2	Take the ith pixel value
3	If pixel value $\geq T$ then the result is 255
4	If pixel value $< T$ then the result is 0
5	Repeat Steps 2, 3, 4 while $i \leq P$
6	Repeat all steps for all pixels

3.3.3 MISP technique

In this technique, the images are transmitted from the CPU to the GPU in a combined data array [34] and the results are returned from the GPU to the CPU in a combined data array using the proposed Algorithm 2 (Table 3). After transferring the combined results to CPU, they are separated according to size of the images. Also, the pixels of the images are distributed and processed one pixel per core of the GPU using the proposed Algorithm 3 (Table 4).

3.3.4 MIMP technique

In this technique, the images are transmitted from the CPU to the GPU in a combined data array and the results are returned from the GPU to the CPU in a combined data array using the proposed Algorithm 2 (Table 3). Also, the pixels of the images are distributed and processed as multipixels per GPU core using the proposed Algorithm 4 (Table 5).

Table 6 Proposed techniques

techniques	Algorithms for transmission of the images and results	Algorithms for distributing and processing the images
SISP	Algorithm 1 (Table 2)	Algorithm 3 (Table 4)
SIMP	Algorithm 1 (Table 2)	Algorithm 4 (Table 5)
MISP	Algorithm 2 (Table 3)	Algorithm 3 (Table 4)
MIMP	Algorithm 2 (Table 3)	Algorithm 4 (Table 5)

4 Image transmission between CPU and GPU

In real-time applications, data transmission time is also very important factor. In systems with GPU, data transmission time consists of two components. These components are defined as the time spent for transmission of the data from CPU to GPU and from GPU to CPU, accordingly. Before executing a kernel on the GPU, all of the data used by kernel need to be transmitted from the CPU memory to the GPU memory. After execution, the results produced by the kernel most likely need to be transmitted back to the CPU memory. $cudaMemcpy$ function is used to transmit data in both directions.

Transmission time in both directions consists of two components. First component is latency which includes preparation overhead. This overhead may occur due to instruction decoding, memory latency, waiting for bus access and other causes. Second component is the propagation time which depends on bandwidth (the number of bits propagated per second). This property has great impact on the performance of a graphic processor since all data which shall be used in the computation must be copied to the graphics processor.

The Hockney model describes in its simplest form how the bandwidth and latency affect the transmission time (t) which is necessary to transmission a given set of data [35].

$$t = L + m/B, \tag{5}$$

where L is the latency, B is bandwidth and m is size of transmitted data.

Latency and bandwidth depend on graphic card, memory allocation, memory architecture, memory speed, CPU architecture, CPU speed, chipsets and bus clock frequency. Calculation of the transmission time accepting into account all of above-listed parameters is not so easy task. In practice, measured transmission time is used.

5 Experimental results

Experiments were related to the real-time detection of standard defects such as scratches, dents, wrinkles and crimps on the surface of the military cases [36,37] (Fig. 2). Eight images covering the entire 360-degree (8×45 degree) surface of the same moving military cases were used to detect the defects (Fig. 3). A multicore CPU with OpenMP and GPGPU with CUDA was used to perform the parallel segmentation

Fig. 2 Military cases

Fig. 3 Images covering the entire 360-degree (8x45 degree) surface of the same military case

of the military cases and background using the thresholding. Speedup rate (r) was used to evaluate segmentation techniques:

$$r = t_{\mathrm{ST}}/t_{\mathrm{PT}}, \tag{6}$$

where t_{PT} is the processing time of parallel thresholding.

The following platform was used: Intel Core i7-3630QM CPU with 4 cores and hyper threading technologies; 8 GB RAM; Windows 7. The codes were written in C++ using the Visual Studio 2012. Images with different resolutions (320×240, 640×480, and 1280×960) were used.

Table 7 Experiment results on a multicore CPU with OpenMP

Chunk size	Speedup with		
	Static scheduling	Dynamic scheduling	Guided scheduling
1	3.42	4.03	4.08
2	3.30	4.00	4.1
4	3.39	4.12	4.02
6	3.44	3.95	4.01
10	3.47	3.86	4.06
15	3.56	3.93	3.64
30	3.41	3.81	3.66
60	3.42	3.50	3.58
120	3.44	3.30	3.39
240	2.86	2.79	3.04
480	1.77	1.80	1.75

5.1 Parallel thresholding on a multicore CPU with OpenMP

Static, dynamic and guided scheduling types with different chunk sizes were implemented to speed up the segmentation process (Table 7).

Table 7 presents the experimental results of the speedup of different scheduling types with different chunk sizes. As seen, the dynamic and guided scheduling types gave the best results. By increasing the chunk size, the speedup is decreased for all scheduling types. In summary, in order to obtain the best results by OpenMP, chunk sizes should be as small as possible and dynamic or guided scheduling types should be used. There is one important point to be underlined. Namely, as shown in Table 7, the values of speed up with dynamic and guided scheduling exceed 4. The reason is that the CPU with four cores has hyperthreading technology.

5.2 Parallel thresholding on a GPU with CUDA

NVIDIA GeForce GT 635M with 96 cores, Tesla K20 with 2496 cores and Tesla K40 with 2880 cores were used. The number of thread size was set to 1024. Four techniques were implemented: (1) SISP; (2) SIMP; (3) MISP; and (4) MIMP.

5.2.1 SISP

In this technique, eight images were sent and executed one by one. The pixels of the images were distributed as one pixel (or 8 bits) per GPU core (Table 8).

As seen, Tesla K40 gave the best result 35 times of improvement without count of transmission time and 12 times of improvement with count of transmission time in comparison with serial computing. Another finding is that, in general, by increasing

Table 8 Data transmit and process performance evaluation for SISP technique

GPU	Single image resolution	Serial time, t_s (ms)	Kernel time, t_k (ms)	Speedup without data transmission, t_s/t_k	Data transmission from CPU to GPU (1 image), t^1_{CPU} (ms)	Data transmission from GPU to CPU (1 image), t^1_{GPU} (ms)	Data transmission from GPU to CPU (8 Image), t^8_{GPU} (ms)	Parallel execution time, t_p (ms)	Speedup with data transmission, t_s/t_p
GeForce GT 635M	320 × 240	4.88	0.54	8.9	0.23	0.09	0.63	1.94	2.51
	640 × 480	9.30	0.92	10.07	0.34	0.17	1.24	4.79	1.94
	1280 × 960	26.59	2.91	9.10	0.71	0.40	3.24	16.18	1.64
Tesla K20	320 × 240	8.25	0.35	23.02	0.06	0.07	0.54	0.96	8.52
	640 × 480	16.51	0.90	18.27	0.17	0.28	2.05	3.13	5.26
	1280 × 960	44.13	5.09	8.65	0.48	0.87	7.18	12.76	3.45
Tesla K40	320 × 240	5.79	0.16	35.19	0.05	0.03	0.24	0.45	12.85
	640 × 480	10.68	0.36	28.98	0.09	0.08	0.64	1.06	10.01
	1280 × 960	30.09	1.65	18.23	0.26	0.25	1.91	3.68	8.17

Table 9 Data transmit and process performance evaluation for SIMP technique

GPU	Single image resolution	Serial time, t_s (ms)	Kernel time, t_k (ms)	Speedup without data transmission, t_s/t_k	Data transmission from CPU to GPU (1 image), t^1_{CPU} (ms)	Data transmission from GPU to CPU (1 image), t^1_{GPU} (ms)	Data transmission from GPU to CPU (8 Image), t^8_{GPU} (ms)	Parallel execution time, t_p (ms)	Speedup with data transmission, t_s/t_p
GeForce GT 635M	320×240	4.61	0.59	7.73	0.33	0.13	0.98	1.81	2.55
	640×480	8.95	0.90	9.90	0.37	0.18	1.42	3.89	2.30
	1280×960	26.67	3.02	8.80	0.63	0.37	2.95	13.12	2.03
Tesla K20	320×240	9.01	0.44	20.14	0.07	0.08	0.65	1.17	7.67
	640×480	16.87	0.82	20.41	0.13	0.23	1.79	2.75	6.12
	1280×960	54.53	5.17	10.53	0.51	0.87	7.34	13.04	4.17
Tesla K40	320×240	5.75	0.15	36.17	0.05	0.03	0.24	0.43	13.14
	640×480	10.55	0.30	34.65	0.09	0.08	0.62	1.00	10.46
	1280×960	29.95	1.60	18.71	0.28	0.25	1.87	3.66	8.18

Table 10 Data transmit and process performance evaluation for MISP technique

GPU	Single image resolution	Serial time, t_s (ms)	Kernel time, t_k (ms)	Speedup without data transmission, t_s/t_k	Data transmission from CPU to GPU, t^1_{CPU} (ms)	Data transmission from GPU to CPU, t^1_{GPU} (ms)	Parallel execution time, t_p (ms)	Speedup with data transmission, t_s/t_p
GeForce GT 635M	320×240	4.68	0.80	5.85	0.10	0.23	1.13	4.13
	640×480	9.16	2.62	3.49	0.60	0.69	3.92	2.33
	1280×960	26.07	10.10	2.58	2.26	2.50	14.86	1.75
Tesla K20	320×240	8.45	0.44	18.92	0.26	0.52	1.23	6.83
	640×480	15.23	0.54	27.78	0.97	1.33	2.86	5.31
	1280×960	43.99	1.04	42.01	3.86	4.73	9.64	4.56
Tesla K40	320×240	5.76	0.10	54.58	0.10	0.13	0.34	16.76
	640×480	10.61	0.21	48.42	0.38	0.41	1.01	10.46
	1280×960	30.41	0.63	47.77	1.45	1.12	3.21	9.46

Table 11 Data transmit and process performance evaluation for MIMP technique

GPU	Single image resolution	Serial time, t_s (ms)	Kernel time, t_k (ms)	Speedup without data transmission, t_s/t_k	Data transmission from CPU to GPU, t_{CPU} (ms)	Data transmission from GPU to CPU, t_{GPU} (ms)	Parallel execution time, t_p (ms)	Speedup with data transmission, t_s/t_p
GeForce GT 635M	320×240	5.15	0.59	8.61	0.10	0.26	0.96	5.32
	640×480	9.51	1.80	5.27	0.53	0.61	2.95	3.21
	1280×960	27.06	6.84	3.95	2.21	2.23	11.19	2.41
Tesla K20	320×240	8.28	0.43	19.09	0.22	0.78	1.14	7.22
	640×480	15.28	0.45	33.71	1.06	1.46	2.98	5.12
	1280×960	44.32	0.75	58.96	3.36	4.34	8.45	5.24
Tesla K40	320×240	5.77	0.09	61.50	0.10	0.13	0.33	17.36
	640×480	10.47	0.16	63.52	0.44	0.44	1.05	9.89
	1280×960	30.61	0.43	71.02	1.41	1.11	2.96	10.32

Table 12 Comparison results of the proposed techniques without transmit time

GPU	Image resolution	Speedup without transmission			
		SISP	SIMP	MISP	MIMP
GeForce GT 635M	320 × 240	8.03	8.25	5.98	7.26
	640 × 480	10.19	9.25	3.36	5.04
	1280 × 960	8.92	9.00	2.54	3.80
Teska K20	320 × 240	23.02	20.14	18.92	19.09
	640 × 480	18.27	20.41	27.78	33.71
	1280 × 960	8.65	10.00	42.02	58.96
Teska K40	320 × 240	**35.19**	**36.17**	**54.58**	61.50
	640 × 480	28.98	34.65	48.42	63.52
	1280 × 960	18.23	18.71	47.77	**71.02**

The best results are highlighted in bold

Table 13 Comparison results of the proposed techniques with transmit time

GPU	Image resolution	Speedup with transmission			
		SISP	SIMP	MISP	MIMP
GeForce GT 635M	320 × 240	2.51	2.55	4.13	5.32
	640 × 480	1.94	2.30	2.33	3.21
	1280 × 960	1.64	2.03	1.75	2.41
Teska K20	320 × 240	8.52	7.67	6.83	7.22
	640 × 480	5.26	6.12	5.31	5.12
	1280 × 960	3.45	4.17	4.56	5.24
Teska K40	320 × 240	**12.85**	**13.14**	**16.76**	**17.36**
	640 × 480	10.01	10.46	10.46	9.89
	1280 × 960	8.17	8.18	9.46	10.32

The best results are highlighted in bold

the image resolution the speedup rate decreases for all kinds of GPU for both cases (without and with transmission time).

5.2.2 SIMP

In this technique, eight images were sent and executed one by one. The pixels of the images were distributed as four pixels (or 32 bits) per GPU core (Table 9).

As seen, Tesla K40 gave the best result 36 times of improvement without count of transmission time and 13 times of improvement with count of transmission time in comparison with serial computing. Another finding is that by increasing the image resolution the speedup rate decreases for all kinds of GPU for both cases (without and with count of transmission time).

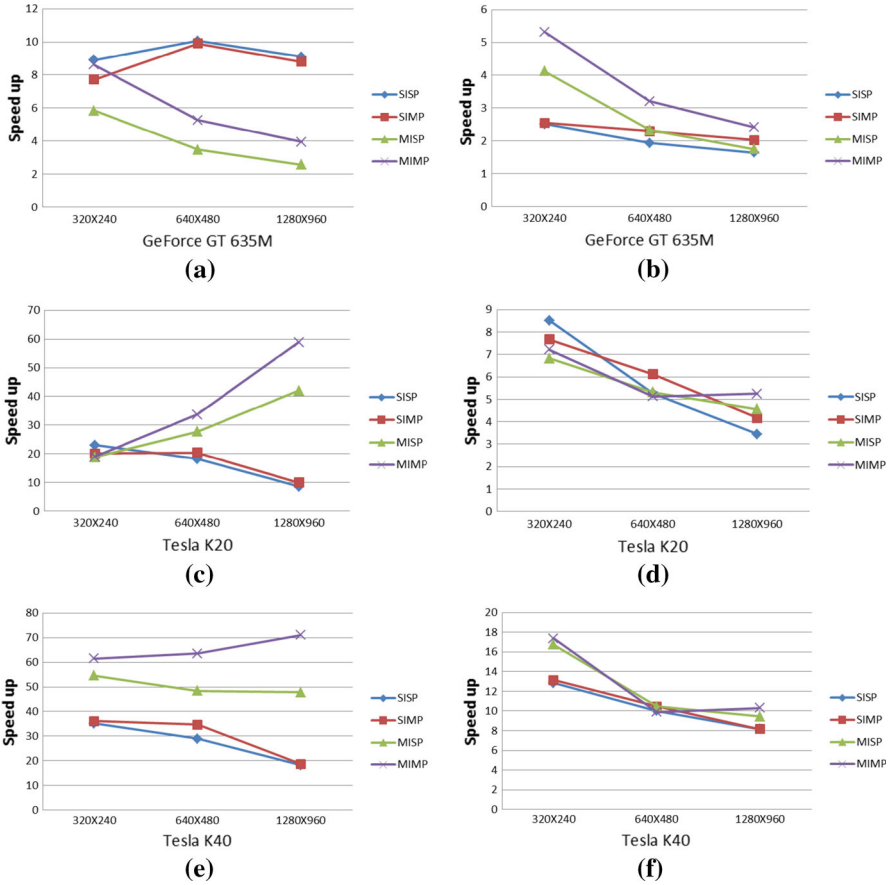
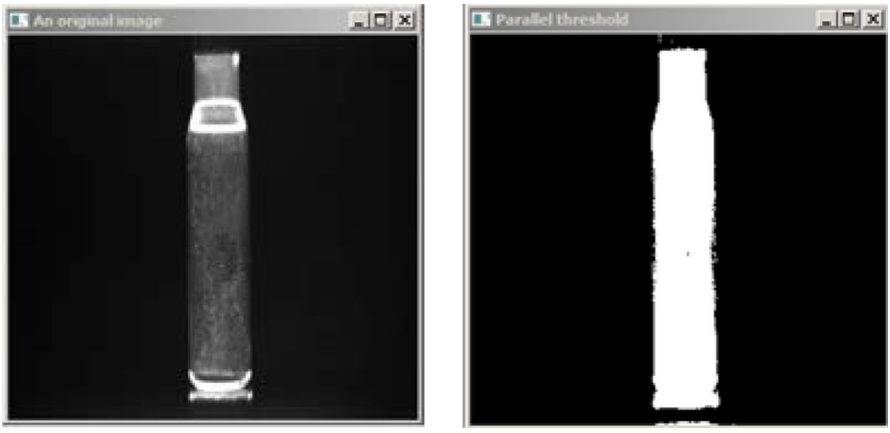


Fig. 4 Comparison results of the proposed techniques using: **a** GeForce GT 635M without transmission time; **b** GeForce GT 635M with transmission time; **c** Tesla K20 without transmission time; **d** Tesla K20 with transmission time; **e** Tesla K40 without transmission time; **f** Tesla K40 with transmission time

5.2.3 MISP

In this technique, eight images were combined in a data array. This data array was sent and executed in a kernel. The pixels of the images were distributed as one pixel (or 8 bits) per GPU core (Table 10).

As seen, Tesla K40 gave the best result 54 times of improvement without count of transmission time and 16 times of improvement with count of transmission time in comparison with serial computing. Another finding is that by increasing the image resolution, the speedup rate decreases for Geforce and Tesla K40 and increases for Tesla K20 without transmission time. Also by increasing the image resolution the speedup rate decreases for Geforce, Tesla K20 and Tesla K40 with transmission time.



(a) **(b)**

Fig. 5 **a** The original image; **b** Segmentation result by parallel thresholding

5.2.4 MIMP

In this technique, eight images were combined in a data array. These data were sent and executed in a kernel. The pixels of the images were distributed as four pixels (or 32 bits) per GPU core (Table 11).

As seen, Tesla K40 gave the best result 71 times of improvement without count of transmission time and 17 times of improvement with count of transmission time in comparison with serial computing. Another finding is that by increasing the image resolution the speedup rate decreases for Geforce and increases for Tesla K20 and Tesla K40 without transmission time. Also, in general, by increasing the image resolution the speedup rate decreases for Geforce, Tesla K20 and Tesla K40 with transmission time.

The comparison results of the proposed techniques with CUDA in terms of speedup are given in Tables 12, 13 and Fig. 4.

Different computers were used to implement GeForce, Tesla K20 and Tesla K40. Due to the differences of CPUs of these computers, the different serial times to process the image with same resolution were measured. For example, the serial time to process the image with resolution of (320×240) by different CPUs was measured as 4.88, 8.25 and 5.79 ms (see the column of serial time in Table 8). Speedup rate for each GPU was affected by the capacity of CPUs.

In general, GeForce gave less improvement than Tesla K20 and K40. This is due to the fewer number cores (96) in comparison with Tesla K20 and K40, which have 2496 and 2880 cores. As shown in Tables 12 and 13, the best results of speedup rate without and with transmission times were obtained by using Tesla K40 for all techniques and image resolutions. Among all techniques, MIMP gave the maximum speedup 71 times without transmission time and 17 times with transmission time. From Tables 8, 9, 10, 11, 12, 13, it can be summarized that Tesla K40 GPU and MIMP technique should be used to get the maximum performance. As seen, there is a big difference between

speedup rates without and with transmission time. The reason is the transmission time between CPU and GPU.

As seen, Tesla K40 gave the best results for all techniques. With Tesla K40, the speedup rates for MISP and MIMP techniques were higher than those of the SISP and SIMP ones. Another point with Tesla was that, by increasing the image resolution, the speedup rate increased. In summary, in order to obtain the best results with CUDA, MISP and MIMP techniques should be used.

An example for segmentation results with parallel thresholding is given in Fig. 5.

6 Conclusion

This paper has presented the image processing applications using multicore and multiprocessing technologies to satisfy real-time conditions. To this end, the algorithms and techniques for the parallel image segmentation through thresholding on K images covering the entire surface of the same metallic and cylindrical moving objects were proposed. A multicore CPU with OpenMP and GPGPU with CUDA was used to implement the thresholding of military cases using eight real images covering their entire surface. Obtained implementation results were compared with the results of serial computing in terms of speedup metric. Experiment results have showed that a GPU with CUDA has a huge capacity to increase the performance of real-time applications.

The best results of speedup rate without and with transmission times were obtained by using Tesla K40 for all techniques and image resolutions. Four techniques have been proposed to process the real-time thresholding such as SISP, SIMP, MISP and MIMP. Among all proposed techniques, MIMP gave the maximum speedup 71 times without transmission time and 17 times with transmission time in comparison with serial computing. As seen, there is a big difference between speedup rates without and with transmission time. The reason is the transmission time between CPU and GPU. As summary, Tesla K40 GPU and MIMP technique should be used to get the maximum performance.

As future work, the time to transmit images from the CPU to the GPU and results from the GPU to the CPU will be analyzed and optimized. More studies can be made on the chained-cubic tree and optical chained-cubic tree topologies. It would be interesting to apply our implementation on these topological properties [38,39].

References

1. Hu J, Zhang T, Jiang H (2006) New multi-DSP parallel computing architecture for real-time image processing. J Syst Eng Electron 17(4):883
2. Mondal P, Biswal PK, Banerjee S (2016) FPGA based accelerated 3D affine transform for real-time image processing applications. Comput Electr Eng 49(1):69
3. Mertes JG, Marranghello N, Pereira AS (2013) Real-time module for digital image processing developed on a FPGA. In: 12th IFAC Conference on Programmable Devices and Embedded Systems. IFAC Proceedings Volumes 46(28), p 405
4. Daz-Pernil D, Berciano A, Pea-Cantillana F, Gutirrez-Naranjo MA (2013) Segmenting images with gradient-based edge detection using membrane computing. Pattern Recognit Lett 34(8):846
5. Huqqani AA, Schikuta E, Ye S, Chen P (2013) Multicore and GPU parallelization of neural networks for face recognition. Procedia Comput Sci 18:349

6. Mahafzah BA (2011) Parallel multithreaded IDA heuristic search: algorithm design and performance evaluation. Int J Parallel Emerg Distrib Syst 26(1):61
7. Mahafzah BA (2013) Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture. J Supercomput 66(1):339
8. Szgyi Z, Trk M, Pataki N (2011) Multicore C++ standard template library in a generative way. In: Proceedings of the Third Workshop on Generative Technologies (WGT) 2011. Electronic Notes in Theoretical Computer Science, vol 279(3), p 63
9. Smistad E, Elster AC, Lindseth F (2014) GPU accelerated segmentation and centerline extraction of tubular structures from medical images. Int J Comput Assist Radiol Surg 9(4):561. https://doi.org/10.1007/s11548-013-0956-x
10. Brodtkorb AR, Hagen TR, SeTra ML (2013) Graphics processing unit GPU programming strategies and trends in GPU computing. J Parallel Distrib Comput 73(1):4
11. Patil S, Junnarka A (2015) Color image segmentation using median cut and contourlet transform: a parallel segmentation approach. Int J Comput Sci Inf Technol (IJCSIT) 5(6):7353
12. Thapliyal H, Arabnia H (2006) Reversible programmable logic array (RPLA) using Fredkin and Feynman gates for industrial electronics and applications. In: Proceedings of 2006 International Conference on Computer Design and Conference on Computing in Nanotechnology, Las Vegas, pp 70–74
13. Thapliyal H, Arabnia H, Bajpai R, Sharma K (2007) Combined integer and variable precision (CIVP) floating point multiplication architecture for FPGAs. In: Proceedings of 2007 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, pp 449–450
14. Arabnia HR, Oliver MA (1986) Fast operations on raster images with SIMD machine architectures. Comput Graph Forum **5**(3):179–188. https://doi.org/10.1111/j.1467-8659.1986.tb00296.x
15. Gopineedi PD, Thapliyal H, Srinivas MB, Arabnia HR (2006) Novel and efficient 4:2 and 5:2 compressors with minimum number of transistors designed for low-power operations, pp 160–168
16. Balasubramanian P, Arisaka R, Arabnia H (2012) RB DSOP a rule based disjoint sum of products synthesis method. In: Proceedings of 2012 International Conference on Computer Design, Las Vegas, pp 39–43
17. Thapliyal H, Srinivas M, Arabnia H (2005) Reversible logic synthesis of half, full and parallel subtractors. In: Proceedings of 2005 International Conference on Embedded Systems and Applications, Las Vegas, pp 165–172
18. Al-amri SS, Kalyankar NV, D KS (2010) Image segmentation by using threshold techniques. CoRR abs/1005.4020
19. Osuna-Enciso V, Cuevas E, Sossa H (2013) A comparison of nature inspired algorithms for multi-threshold image segmentation. Expert Syst Appl 40(4):1213
20. Wei S, Hong Q, Hou M (2011) Automatic image segmentation based on PCNN with adaptive threshold time constant. Neurocomputing 74(9):1485
21. Han S, Tao W, Wu X, cheng Tai X, Wang T (2010) Fast image segmentation based on multilevel banded closed-form method. Pattern Recognit Lett 31(3):216
22. Ayala HVH, dos Santos FM, Mariani VC, dos Santos Coelho L (2015) Image thresholding segmentation based on a novel beta differential evolution approach. Expert Syst Appl 42(4):2136
23. Wang R, Li C, Wang J, Wei X, Li Y, Zhu Y, Zhang S (2015) Threshold segmentation algorithm for automatic extraction of cerebral vessels from brain magnetic resonance angiography images. J Neurosci Methods 241:30
24. Happ P, Feitosa R, Bentes C, Farias R (2012) A parallel image segmentation algorithm on GPUs. In: Proceedings of the 4th GEOBIA, Rio de Janeiro, 2012, pp 580–586
25. Smistad E, Elster AC, Lindseth F (2014) GPU accelerated segmentation and centerline extraction of tubular structures from medical images. Int J Comput Assist Radiol Surg 9(4):561
26. Korbes A, Vitor GB, de Alencar Loyufoi R, Ferreira JV (2010) Analysis of a step-based watershed algorithm using CUDA. Int J Curr Res Rev 1(1):6
27. Singh BM, Sharma R, Mittal A, Ghosh D (2011) Parallel implementation of Otsus binarization approach on GPU. Int J Comput Appl 32(2):16
28. Farias R, Farias R, Marroquim R, Clua E (2013) Parallel image segmentation using reduction-sweeps on multicore processors and GPUs. In: Proceedings of the 2013 XXVI Conference on Graphics, Patterns and Images, SIBGRAPI '13. IEEE Computer Society, Washington, DC, pp 139–146
29. Prosser N (2010) Medical image segmentation using gpu accelerated variational level set methods. Master's thesis, Rochester Institute of Technology

30. Abramov A, Kulvicius T, Wörgötter F, Dellen B (2010) Real-time image segmentation on a GPU. In: Keller R, Kramer D, Weiss JP (eds) Facing the multicore-challenge. Lecture notes in computer science, vol 6310. Springer, Berlin, Heidelberg
31. Smistad E, Falch TL, Bozorgi M, Elster AC, Lindseth F (2015) Medical image segmentation on GPUs a comprehensive review. Med Image Anal 20(1):1
32. Li Y, Jiao L, Shang R, Stolkin R (2015) Dynamic-context cooperative quantum-behaved particle swarm optimization based on multilevel thresholding applied to medical image segmentation. Inf Sci 294:408
33. Chen Z, Meng X, Guo L, Liu G (2012) GICUDA: a parallel program for 3D correlation imaging of large scale gravity and gravity gradiometry data on graphics processing units with CUDA. Comput Geosci 46:119
34. Bay OF, Samet R, Aydn S, Tural S, Bayram A (2015) Performance analysis of GPU-based parallel image segmentation using CUDA. In: Proceedings of the 2th International Conference on Advanced Technology and Sciences (Antalya-Turkey, 2015), ICAT'15, pp 426–429
35. Hovland RJ Latency and bandwidth impact on gpu-systems. Tech. rep., Norwegian University of Science and Technology
36. Samet R, Aydin S, Bay OF, Tural S, Bayram A (2015) Real time image processing applications on multicore CPU and GPGPU. In: The 21st International Conference on Parallel and Distributed Processing, WORLDCOMP'15, Las Vegas-Nevada, 27–30 July 2015
37. Samet R, Aydin S, Tural S, Bayram A (2016) Primer defects detection on military cartridge cases. In: The 15th annual International Conference, NICOGRAPH'15, Hangzhou, 6–8 July 2016
38. Abdullah M, Abuelrub E, Mahafzah B (2011) The chained-cubic tree interconnection network. Int Arab J Inf Technol 8(3):334
39. Mahafzah BA, Alshraideh M, Abu-Kabeer TM, Ahmad EF, Hamad NA (2012) The optical chained-cubic tree interconnection network: topological structure and properties. Comput Electr Eng 38(2):330. https://doi.org/10.1016/j.compeleceng.2011.11.023