Real-Time Image Processing Applications on Multicore CPUs and GPGPU

R. Samet¹, O.F. Bay², S. Aydu², S. Tural¹, A. Bayram¹
¹Computer Engineering, Ankara University, Ankara, Turkey
²Education of Computer and Electronics, Gazi University, Ankara, Turkey

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. Experiments with multicore CPUs 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 methods: 1) Single-Image Transmission with Single-Pixel Processing; 2) Single-Image Transmission with Multiple-Pixel Processing; 3) Multiple-Image Transmission with Single-Pixel Processing; 4) Multiple-Image Transmission with Multiple-Pixel Processing. All methods were implemented on GeForce and Tesla. Tesla gave the best results of 23 (for the first method), 20 (for the second method), 42 (for the third method), and 58 (for the fourth method) times improvements in comparison with serial computing.

Keywords: Parallel computing, real-time image processing, image segmentation, thresholding, multicore programming, GPU programming

1 Introduction

On the one hand, image processing requires long time. On the other hand, time is usually limited in the real-time applications. So, serial image processing does not satisfy real time conditions. In order to solve this problem, parallel computing techniques, especially multicore and multiprocessing technologies, should be used.

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 whole 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”). Threads, OpenMP (Open Multi-Processing), TBB (Threading Building Blocks), and Cilk are application programming interfaces (API) to efficiently use the capacity of a multicore CPU. In this study, a general purpose and platform-independent OpenMP that supports shared-memory for multi-processing programming in C, C++, and FORTRAN will be used.

A graphic processing unit (GPU) is a single instruction and multiple data stream (SIMD) architecture where the same instruction is performed on all data elements in parallel. At the same time, 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 [1]. 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 study [2].

Multicore CPU and GPGPU technologies are widely used for non-real-time and real-time image processing applications. A short literature review related to these technologies is given below.

There are many studies reported in the literature related to non-real-time image segmentation using the threshold method [3, 4]. The performance was and still remains an urgent issue to be solved in real-time image processing applications. To this end, different algorithms and methods have been developed for serial computing [5, 6, 7]. 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 programming has been used in multithread image processing and image segmentation applications with multicore computing [8, 9]. CUDA programming has been used for parallel image segmentation by region growing, watershed, and Otsu binarization algorithms on GPU [10, 11, 12, 13]. The reduction sweep algorithm was used for image segmentation on both CPU and GPU [14]. In [15], several methods for image segmentation were implemented using CUDA and GPU and processing time was accelerated about 20 times. The authors of [16] present the results of image segmentation on a video with a frame rate of 30 Hz using CUDA and GPU.

This review shows that more efficient algorithms and methods still need to be developed to improve the performance of real-time image processing applications. One of the aims of this study was to make a contribution to this area using OpenMP and CUDA. To this end, bi-level
thresholding was implemented on the images covering the entire surface of the same metallic and cylindrical moving object in parallel with the following methods. One method 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 were analyzed. The other four methods are related to GPU programming with the CUDA platform: 1) Single-Image Transmission with Single-Pixel Processing (SISP) (Namely, 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) (Namely, images are transmitted from CPU to GPU one by one and the pixels of the images are processed multi pixels per core of GPU); 3) Multiple-Image Transmission with Single-Pixel Processing (MISP) (Namely, multiple images are transmitted from CPU to GPU together and the pixels of the images are processed one pixel per core of GPU); 4) Multiple-Image Transmission with Multiple-Pixel Processing (MIMP) (Namely, multiple images are transmitted from CPU to GPU together and the pixels of the images are processed multi pixels per core of GPU). Performance analysis related to execution time was performed by comparison of the results obtained by these methods with serial computing. The method with multicore CPU showed that, by increasing the chunk size, the execution time decreases approximately four times. All methods with GPU were implemented on GeForce and Tesla. Tesla gave best results of 23 (for SISP method), 20 (for SIMP method), 42 (for MISP method), and 58 (for MIMP method) times improvements in comparison with serial computing.

2 Proposed real-time image processing methods

Real-time applications of this study are related to the inspection of certain defects on the entire surface of metallic and cylindrical objects moving at a rate of 5 units per second or 1 unit per 200 milliseconds. Data sets or images used in this study are the images taken from the entire surface of the same metallic and cylindrical moving object. Images were used to inspect the defects in real-time. Defects are detected by image processing techniques. In order to detect certain defects of a single object, the image processing steps should be processed on K images covering its entire surface during 200 milliseconds. So, time is limited in given applications. In this study, 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 multiprocessor technologies, were used to speed up the thresholding of the metallic and cylindrical object from images covering its entire surface.

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. Four different algorithms and methods related to CUDA are proposed.

2.1 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 [17]. 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 study, bi-level segmentation is used for the segmentation of objects and the background [4]. Thresholding is often used as a preprocessing step, followed by other post-processing methods [18]. 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

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

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

\[
t_{ST} = N \times M \times \Delta t
\]

where \( t_{ST} \) is the processing time of serial thresholding and \( \Delta t \) is the processing time for thresholding on one pixel.

2.2 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. 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 and the master thread continues the program execution as shown in Fig.1.

![Fig.1. Thread organization with OpenMP](image-url)
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 where 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 chunk size for guided scheduling is approximately calculated by:

\[ \text{ChunkSize} = \frac{N_L}{N_T} \]

(3)

where \( N_L \) is a number of loop count and \( N_T \) is a number of threads. The performance or processing time of parallel thresholding with OpenMP is defined as follows:

\[ t_{MP} = \frac{t_{ST}}{N_T} + t_D = \frac{N + M\cdot N_T}{N_T} + t_D \]

(4)

where \( t_{MP} \) is a processing time of parallel thresholding with OpenMP and \( t_D \) is a processing time for fork and join of threads.

2.3 Parallel thresholding on a GPU with CUDA

In order to accelerate the thresholding process to satisfy the real-time conditions, GPU programming with CUDA is proposed. The CUDA programming model 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 in SIMD fashion independently. Threads within a warp execute the same instruction on different data elements in parallel [19].

In order to parallelize the thresholding process, the kernel should be used. To organize kernels to work in parallel, streams are used (Fig.2).

1. cudaMemcpyAsync (dst, src, size, dir, stream1);
2. cudaMemcpyAsync (dst, src, size, dir, stream2);
3. cudaMemcpyAsync (dst, src, size, dir, streamK);
4. cudaMemcpyAsync (dst, src, size, dir, stream1);
5. cudaMemcpyAsync (dst, src, size, dir, stream2);
6. cudaMemcpyAsync (dst, src, size, dir, streamK);
7. cudaMemcpyAsync (dst, src, size, dir, stream1);
8. cudaMemcpyAsync (dst, src, size, dir, stream2);
9. cudaMemcpyAsync (dst, src, size, dir, streamK).

Fig.2. Multi kernels organization by streams

Firstly, the \( K \) streams are defined (Line 1) and created (Lines 2, 3). \( K \) is a number of images. Then data (images) for created streams are transferred 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 transferred 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 multi pixels. Results can be returned from the GPU to the CPU one by one, or in a combined data array.

The algorithm 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 Fig.3.

Step 1: Send the 1st image from CPU to GPU;
Step 2: Execute the thresholding kernel on the 1st image;
Step 3: Send the 2nd image from CPU to GPU;
Step 4: Execute the thresholding kernel on the 2nd image;
Step 5: Return the 1st result from GPU to CPU;
Step 6: Repeat Step 2, 3, 4 while \( i \leq K \).

Fig.3. Algorithm for single-image transmission

The algorithm for sending \( K \) images from the CPU to the GPU in a combined data array, processing them in GPU and returning the results from the GPU to the CPU in a combined data array is given in Fig.4.

Step 1: Combine \( K \) images in a data array;
Step 2: Send the combined data array from CPU to GPU;
Step 3: Execute the thresholding kernel on \( K \) images;
Step 4: Return the combined results from GPU to CPU;
Step 5: Separate the images from combined results.

Fig.4. The algorithm for multiple-image transmission

The algorithm for distributing and processing the images as one pixel per core of the GPU is given in Fig.5.

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

Fig.5. The algorithm for single-pixel processing in the GPU

The algorithm for distributing and processing the images as \( P \) pixels per core of the GPU is given in Fig.6.

Step1: Distribute pixels as \( P \) pixels per core of GPU;
Step2: Take the \( i \)th pixel value;
Step 3: If pixel value \( \geq T \) then the result is 255;
Step 4: If pixel value \( < T \) then the result is 0;
Step 5: Repeat Step 2, 3, 4 while \( i \leq P \);
Step 6: Repeat all steps for all pixels.

Fig.6. The algorithm for multi-pixel processing in the GPU.
Four methods are proposed to execute thresholding on the GPU with CUDA: 1) SISP; 2) SIMP; 3) MISP, and 4) MIMP (Table 1).

<table>
<thead>
<tr>
<th>Methods</th>
<th>Images and Results</th>
<th>Image Distributing and Processing Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>SISP</td>
<td>Fig.3</td>
<td>Fig.5</td>
</tr>
<tr>
<td>SIMP</td>
<td>Fig.3</td>
<td>Fig.6</td>
</tr>
<tr>
<td>MISP</td>
<td>Fig.4</td>
<td>Fig.5</td>
</tr>
<tr>
<td>MIMP</td>
<td>Fig.4</td>
<td>Fig.6</td>
</tr>
</tbody>
</table>

2.3.1 SISP method

In this method, 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 in Fig.3. Also, the pixels of the images are distributed and processed one pixel per core of the GPU using the proposed algorithm in Fig.5.

2.3.2 SIMP method

In this method, 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 in Fig.3. Also, the pixels of the images are distributed and processed as multi pixels per GPU core using the proposed algorithm in Fig.6.

2.3.3 MISP method

In this method, 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 in Fig.4. Also, the pixels of the images are distributed and processed one pixel per core of the GPU using the proposed algorithm in Fig.5.

2.3.4 MIMP method

In this method, 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 in Fig.4. Also, the pixels of the images are distributed and processed as multi pixels per GPU core using the proposed algorithm in Fig.6.

3 Experiment results

Experiments were related to the real-time detection of standard defects on the surface of the military cases, such as scratches, dents, wrinkles, and crimps (Fig.7).

Eight images covering the entire 360-degree (8x45 degrees) surface of the same moving military cases were used to detect the defects (Fig.8).

A multicore CPU with OpenMP and GPGPU with CUDA were used to implement the parallel segmentation through the thresholding of the military cases and background.

Speed up value was used to evaluate segmentation techniques:

\[ \text{SpeedUp} = \frac{t_{ST}}{t_{PT}} \]  

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

Fig.7. 7.62 mm Military cases

Fig.8. Images covering the entire 360-degree (8x45 degrees) surface of the same military case
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 (320x240, 640x480, and 1280x960) were used.

3.1 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 2).

Table 2. Experiment results on a multicore CPU with OpenMP

<table>
<thead>
<tr>
<th>Chunk Size</th>
<th>Speed up with</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Static</td>
<td>Dynamic</td>
<td>Guided</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scheduling</td>
<td>Scheduling</td>
<td>Scheduling</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3.42</td>
<td>4.03</td>
<td>4.08</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3.30</td>
<td>4.00</td>
<td>4.1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3.39</td>
<td>4.12</td>
<td>4.02</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3.44</td>
<td>3.95</td>
<td>4.01</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>3.47</td>
<td>3.86</td>
<td>4.06</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>3.56</td>
<td>3.93</td>
<td>3.64</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>3.41</td>
<td>3.81</td>
<td>3.66</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>3.42</td>
<td>3.50</td>
<td>3.58</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>3.44</td>
<td>3.30</td>
<td>3.39</td>
<td></td>
</tr>
<tr>
<td>240</td>
<td>2.86</td>
<td>2.79</td>
<td>3.04</td>
<td></td>
</tr>
<tr>
<td>480</td>
<td>1.77</td>
<td>1.80</td>
<td>1.75</td>
<td></td>
</tr>
</tbody>
</table>

Table 2 presents the experiment results of the speed up 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 speed up 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.

3.2 Parallel thresholding on a GPU with CUDA

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

3.2.1 SISP

In this method, 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 3).

As seen, Tesla gave the best result of 20 times improvement in comparison with serial computing. Another point with Tesla was that, for high image resolution, the speed up rate was decreased. GeForce gave 9 times improvement, which was less than Tesla. This was due to the fewer number cores (96) in comparison with Tesla, which has 2496 cores. Also, the speed up rate for GeForce was approximately the same for different image resolutions. In summary, in order to obtain the best results by the SISP method, Tesla should be used and image resolution should be as small as possible.

3.2.2 SIMP

In this method, 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 4).

As seen, Tesla gave the best result of 20 times improvement in comparison with serial computing. Another point with Tesla was that, for high image resolution, the speed up rate was decreased. GeForce gave 9 times improvement, which was less than Tesla. This was due to the fewer number cores (96) in comparison with Tesla, which has 2496 cores. Also, the speed up rate for GeForce was approximately the same for different image resolutions. In summary, in order to obtain the best results by the SIMP method, Tesla should be used and image resolution should be as small as possible.

3.2.3 MISP

In this method 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 5).

As seen, Tesla gave the best result of 20 times improvement in comparison with serial computing. Another point with Tesla was that, for high image resolution, the speed up rate was decreased. GeForce gave 9 times improvement, which was less than Tesla. This was due to the fewer number cores (96) in comparison with Tesla, which has 2496 cores. Also, the speed up rate for GeForce was approximately the same for different image resolutions. In summary, in order to obtain the best results by the MISP method, Tesla should be used and image resolution should be as small as possible.
As seen, Tesla gave the best result of 42 times improvement in comparison with serial computing. Another point with Tesla was that, as the image resolution increased, speed up rate increased. GeForce gave 6 times improvement, which was less than Tesla. This is due to the fewer number cores (96) in comparison with Tesla, which has 2496 cores. Also, the speed up rate of GeForce was decreased by increasing the image resolutions. In summary, in order to obtain the best results by the MISP method, Tesla should be used and image resolution should be as large as possible.

### 3.2.4 MIMP

In this method, eight images were combined in a data array. This 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 6).

<table>
<thead>
<tr>
<th>GPU type</th>
<th>Image Resolution</th>
<th>Serial Computing Time (ms)</th>
<th>Kernel Time (ms)</th>
<th>Speed up Rate on Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce GT 635M</td>
<td>320x240</td>
<td>5.01</td>
<td>0.69</td>
<td>7.26</td>
</tr>
<tr>
<td></td>
<td>640x480</td>
<td>9.34</td>
<td>1.85</td>
<td>5.04</td>
</tr>
<tr>
<td></td>
<td>1280x960</td>
<td>26.92</td>
<td>6.91</td>
<td>3.8</td>
</tr>
<tr>
<td>Tesla K20</td>
<td>320x240</td>
<td>8.38</td>
<td>0.43</td>
<td>19.09</td>
</tr>
<tr>
<td></td>
<td>640x480</td>
<td>15.28</td>
<td>0.45</td>
<td>33.71</td>
</tr>
<tr>
<td></td>
<td>1280x960</td>
<td>44.32</td>
<td>0.75</td>
<td>58.96</td>
</tr>
</tbody>
</table>

As seen, Tesla gave the best result of 58 times improvement in comparison with serial computing. Another point with Tesla was that, as the image resolution increased, the speed up rate increased. GeForce gave 7 times improvement, which is less than Tesla. This is due to the fewer number cores (96) in comparison with Tesla, which has 2496 cores. Another point with GeForce was that the speed up rate decreased by increasing image resolutions. In summary, in order to obtain the best results by the MIMP method, Tesla should be used and image resolution should be as large as possible.

The comparison results of the proposed methods with CUDA in terms of speed up are given in Table 7 and Fig. 9.

<table>
<thead>
<tr>
<th>GPU type</th>
<th>Image Resolution</th>
<th>Speed up on Kernel by</th>
<th>SISP</th>
<th>SIMP</th>
<th>MISP</th>
<th>MIMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce GT 635M</td>
<td>320x240</td>
<td>8.03</td>
<td>8.25</td>
<td>5.98</td>
<td>7.26</td>
<td></td>
</tr>
<tr>
<td></td>
<td>640x480</td>
<td>10.19</td>
<td>9.25</td>
<td>3.36</td>
<td>5.04</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1280x960</td>
<td>8.92</td>
<td>9</td>
<td>2.54</td>
<td>3.8</td>
<td></td>
</tr>
<tr>
<td>Tesla K20</td>
<td>320x240</td>
<td>23.02</td>
<td>20.14</td>
<td>18.92</td>
<td>19.09</td>
<td></td>
</tr>
<tr>
<td></td>
<td>640x480</td>
<td>18.27</td>
<td>20.41</td>
<td>27.78</td>
<td>33.71</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1280x960</td>
<td>8.65</td>
<td>10</td>
<td>42.02</td>
<td>58.96</td>
<td></td>
</tr>
</tbody>
</table>

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

This paper presented the image processing applications using multicore and multiprocessor technologies to satisfy real-time conditions. To this end, algorithms and methods 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 were 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 speed-up values. Experiments showed that a GPU with CUDA has a huge capacity to increase the performance of real-time applications. For example, CUDA speeded up the real-time thresholding process 58 times in comparison with serial computing.

As future work, the time to transfer images from the CPU to the GPU and results from the GPU to the CPU will be analyzed and optimized. Another future work is that the proposed algorithms and methods will be implemented on a different multicore CPU and GPU.

Acknowledgement

This work was funded by the Ministry of Science, Industry and Technology of Turkey under San-Tez grant #0018.STZ.2013-1.

5 References