Knowledge Engineering and Data Science (KEDS)
Vol 4, No 2, December 2021, pp. 69–84

pISSN 2597-4602
eISSN 2597-4637

Parallel Approach of Adaptive Image Thresholding Algorithm on GPU

Adhi Prahara [a, 1], Andri Pranolo [a, b, 2, *], Nuril Anwar [a, 3], Yingchi Mao [b, 4]

[a] *Informatics Department, Universitas Ahmad Dahlan*
Jl. Prof. Dr. Soepomo, S.H., Janturan, Warungboto, Umbulharjo, Yogyakarta 55164, Indonesia

[b] *College of Computer and Information, Hohai University*
1 Xikang Road, Nanjing, Jiangsu 210098, China

[1] *adhi.prahara@tif.uad.ac.id *;* [2] *andri@hhu.edu.cn*;* [3] *nuril.anwar@tif.uad.ac.id; maoyingchi@gmail.com*
** corresponding author*

ARTICLE INFO

Article history:
Submitted 25 December 2021
Revised 28 December 2021
Accepted 30 December 2021
Published online 31 December 2021

Keywords:
Adaptive image thresholding
Computational time
Graphics processing unit
Image processing
Parallel computing

ABSTRACT

Image thresholding is used to segment an image into background and foreground using a given threshold. The threshold can be generated using a specific algorithm instead of a pre-defined value obtained from observation or experiment. However, the algorithm involves per pixel operation, histogram calculation, and iterative procedure to search the optimum threshold that is costly for high-resolution images. In this research, parallel implementations on GPU for three adaptive image thresholding methods, namely Otsu, ISODATA, and minimum cross-entropy, were proposed to optimize their computational times to deal with high-resolution images. The approach involves parallel reduction and parallel prefix sum (scan) techniques to optimize the calculation. The proposed approach was tested on various sizes of grayscale images. The result shows that the parallel implementation of three adaptive image thresholding methods on GPU achieves 4-6 speeds up compared to the CPU implementation, reducing the computational time significantly and effectively dealing with high-resolution images.

I. Introduction

Segmentation is a process that partitions image into segments [1]. Segmentation is useful for changing image representation into something more meaningful and easier to analyze, e.g., finding objects and boundaries. One of the methods to perform image segmentation is image thresholding. The method partitions image into background and foreground using a given threshold. This process is also called binarization because the segmentation result is a binary image that maps "0" pixel as background and "1" pixel as foreground.

In order to perform image thresholding, the threshold value can be determined manually by observation or experiment. However, in the adaptive image thresholding method, the threshold is generated using a specific algorithm. The algorithm involves per pixel operation, histogram calculation, and iterative procedure to search the optimum threshold. Therefore, it can be costly for a high-resolution image.

Some well-known adaptive image thresholding algorithms are Otsu [2], Iterative Self-Organizing Data Analysis Technique (ISODATA) [3], and minimum cross-entropy (MCET) [4]. Otsu method iteratively searches threshold that minimizes inter-class variance. ISODATA method iteratively updates the threshold until the average inter-class distance is less than a given threshold or reaches the maximum number of iterations. MCET method searches optimal threshold by calculating the cross-entropy for all possible thresholds and selecting the one with minimum cross-entropy. The methods have been used in many image processing applications [5][6][7][8][9][10][11][12] to perform automatic image segmentation.

https://doi.org/10.17977/um018v4i22021p69-84

KEDS is Sinta 2 Journal (https://sinta.kemdikbud.go.id/journals/detail?id=6662) accredited by Indonesian Ministry of Education, Culture, Research, and Technology

In image processing, achieving real-time performance is necessary, especially when processing video streaming or image in high resolution. A high-resolution image is a common product of satellite, aerial, biometric, and medical imaging, which is also often used in the verification and segmentation process. It is crucial to analyze the algorithm's complexity to know where it should be optimized to achieve real-time performance. High-Performance Computing (HPC) advanced technology allows the algorithm to be parallelized on Graphics Processing Unit (GPU). Parallel computation can optimize the iterative and serial procedure in an algorithm.

Researchers have been proposed parallel adaptive image thresholding methods for image segmentation. Kanungo *et al.* [13] proposed a parallel genetic algorithm-based adaptive thresholding for image segmentation in uneven lighting conditions. Sandeli and Batouche [14] proposed image thresholding using multilevel thresholding based on a parallel generalized island model (GIM). Nafaji *et al.* [15] use parallel local adaptive thresholding for binarization of documents. Upadhyay *et al.* [16] proposed an adaptive thresholding approach for image segmentation on GPU. All of them gained significant speedup in computational time than serial implementation.

This research proposed a parallel implementation on GPU for three adaptive image thresholding methods: Otsu, ISODATA, and MCET. Our contribution lies in the parallel approach of the adaptive image thresholding method on GPU to optimize their computational times to deal with a high-resolution image. This paper is organized as follows: Section 2 presents the proposed approach of parallel adaptive image thresholding methods, Section 3 presents the result and discussion, and Section 4 presents the conclusion of this work.

II. Method

Adaptive image thresholding is a method to segment images using a threshold generated from a specific algorithm. The algorithm has the purpose of obtaining an optimal threshold for segmentation. In this research, some well-known adaptive image thresholding algorithms, namely Otsu, ISODATA, and MCET are parallelized to optimize high-resolution image performance.

A. Otsu Method

Otsu method is proposed by [2] to perform automatic thresholding on the grayscale image. Otsu method iteratively searches the threshold that maximizes inter-class variance. The steps to apply Otsu threshold is described below:

a) An image is converted into a normalized gray-level histogram using (1) and considered as the probability distribution where the number of pixels in i^{th} gray-level is n_i, the total number of pixels is N, and the probability of i^{th} gray-level is p_i.

$$p_i = n_i/N \tag{1}$$

b) Suppose the pixels are distributed into two classes (commonly as background and foreground), for all possible thresholds $i = 1 \dots k$, the probability of class occurrence ω_i, the class mean level μ_i, and the inter-class variance $\sigma_B^2(k)$ can be calculated using (2), (3), and (4), respectively. Here, $\omega(k)$ and $\mu(k)$ is the zeroth-order and first-order cumulative moments of the histogram and $\mu_T = \sum_{i=1}^{L} i \cdot p_i$ is the total mean level of an image.

$$\omega_i = \sum_{i=1}^{k} p_i = \omega(k) \tag{2}$$

$$\mu_i = \sum_{i=1}^{k} i \cdot p_i/\omega_i = \mu(k)/\omega(k) \tag{3}$$

$$\sigma_B^2(k) = \frac{[\mu_T\omega(k)-\mu(k)]^2}{\omega(k)[1-\omega(k)]} \tag{4}$$

c) The select threshold maximizes σ_B^2 using (5). This threshold is the optimal threshold.'

$$\sigma_B^2(k) = \max_{1\le k<L} \sigma_B^2(k) \tag{5}$$

If L is the number of gray levels and N is the number of pixels in the image, the computational complexity of Otsu method for grayscale image segmentation is given by the following operations:

a) Histogram initialization and histogram computation have a computational complexity of $O(L)$ and $O(N)$, respectively.

b) Search the optimum threshold by maximizing the inter-class variance has a computational complexity of $O(L)$.

c) Implementation of the Otsu threshold on the image requires computational complexity of $O(N)$.

B. ISODATA algorithm

Iterative Self-Organizing Data Analysis Technique (ISODATA) is proposed by [3] to compute the global image threshold. The method uses an iterative procedure to update the threshold. Image segmentation using the ISODATA algorithm is described as follows:

a) Compute gray-level histogram from the image.

b) Create initial segments by splitting the histogram into background and foreground segments using the initial threshold value T_0.

c) Calculate the mean of background pixels μ_B and the mean of foreground pixels μ_F.

d) Calculate a new threshold T by averaging the two means value using (6).

$$T = \frac{\mu_B + \mu_F}{2} \tag{6}$$

a) Repeat the procedures c and d until the threshold value T is less than a given threshold or the maximum iteration number is reached.

The computational complexity of ISODATA method for grayscale image segmentation, where L is the number of gray levels and N is the number of pixels in the image, is given by the following operations:

a) Histogram initialization and histogram computation have a computational complexity of $O(L)$ and $O(N)$, respectively.

b) Update the threshold until the average inter-class distance is less than a threshold or the maximum number of iterations is reached requires computational complexity of $O(Q)$, where Q is the number of iteration required by the algorithm.

c) ISODATA threshold Implementation on the image requires computational complexity of $O(N)$.

C. Minimum Cross-Entropy method

The minimum cross-entropy (MCET) method is proposed by [4] to select an optimal threshold. The method searches the optimal threshold by calculating the cross-entropy for all possible thresholds and selecting the one with minimum cross-entropy. The procedure to apply the minimum cross-entropy method for image segmentation is described below:

a) Compute normalized gray-level histogram from image using (7) where the number of pixels in i gray-level is n_i, the total number of pixels is N, and the probability of i gray-level is p_i.

$$p_i = n_i / N \tag{7}$$

b) Initialize the entropy of gray-level histogram using (8), where a and b are the minima and maximum gray-level intensity.

$$H_{CE} = \sum_{i=a}^{b} i \cdot p_i \cdot \log(i) \tag{8}$$

c) Suppose the pixel is distributed into two classes: background and foreground with a threshold T. If the mean of pixel distribution below the threshold (background) is μ_B and the mean of pixel distribution above the threshold (foreground) is μ_F, then for all possible thresholds, $T = a \ldots b$ calculate the cross-entropy of pixel distribution below and above the threshold using (9).

$$H_{CE}(T) = \sum_{i=a}^{T} n_i \mu_B(T) \log \frac{\mu_B(T)}{i} + \sum_{i=T+1}^{b} n_i \mu_F(T) \log \frac{\mu_F(T)}{i} \tag{9}$$

d) Select the optimal threshold τcorresponding to the minimum of the cross-entropy using (10).

$$\tau_{CE} = \arg\min_{a \le T \le b} H_{CE}(T) \tag{10}$$

If L is the number of gray levels and N is the number of pixels in the image, the computational complexity of MCET method for grayscale image segmentation is given by the following operations:

a) Histogram initialization and histogram computation have a computational complexity of $O(L)$ and $O(N)$, respectively.

b) Select the minimum cross-entropy from all possible thresholds has a computational complexity of $O(L^2)$.

c) Implementation of MCET threshold on the image requires computational complexity of $O(N)$.

D. Parallel Computing on GPU

GPU (Graphics Processing Unit) is a high-level parallel architecture used to do a fast operation in computer graphics, and now it can be used other than graphics, which is known as GP-GPU (General Purpose-Graphics Processing Unit) [17]. The well-known general-purpose parallel computing platform and programming model is Compute Unified Device Architecture (CUDA) from NVidia.

GPU is highly parallel, multithreaded, has many cores processors, and has very high memory bandwidth. The difference between how CPU and GPU process the data is shown in Figure 1(a) and Figure 1(b). GPU devotes more transistors to data processing than caching and flow control. GPU is built on an array of Streaming multiprocessors (SM), and it is organized into grids, blocks, and threads.

Data-parallel processing maps data elements to parallel processing threads. Figure 1(c) shows the parallel processing threads in GPU. A multithreaded program is partitioned into blocks of threads that execute independently from each other. Therefore, using GPU, the computation of adaptive image thresholding algorithms will be parallel processed, reducing computational time.

Using the advantages of GPU's parallel architecture, the adaptive image thresholding methods that involve histogram calculation, cumulative sum, search the minimum or maximum value from an array can be optimized using parallel reduction and parallel prefix sum (scan) algorithms.

1) Parallel Reduction Algorithm

A parallel reduction algorithm can optimize the computation of an array's sum, minimum and maximum value. Parallel reduction allows iteration from half of the total number of bin histograms processed parallel with a computational complexity of $O(\log(N))$ in the shared memory.

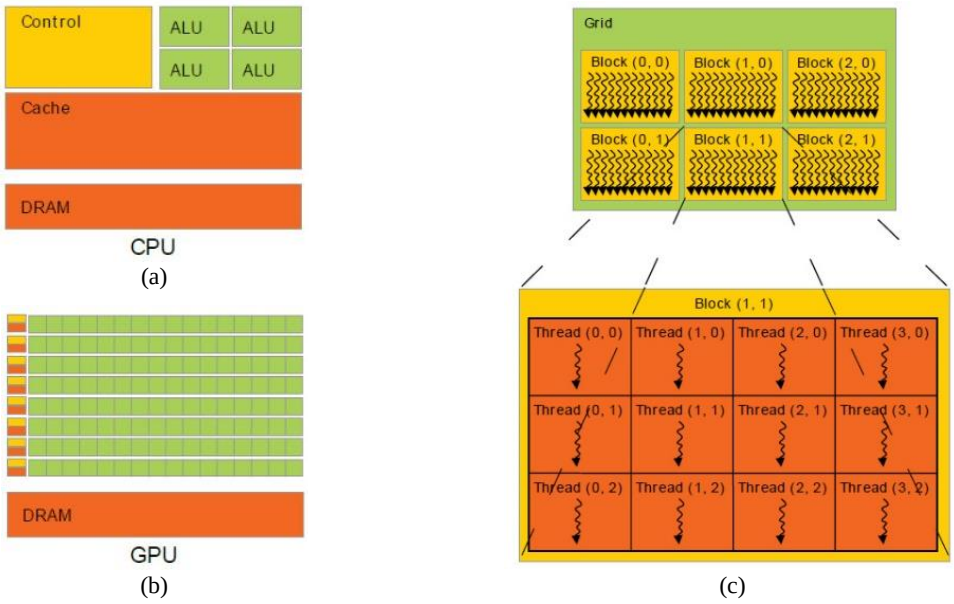


Fig. 1. GPU devotes more transistors to data processing [17]; (a) CPU data process; (b) GPU data process; and (c) GPU parallel processing

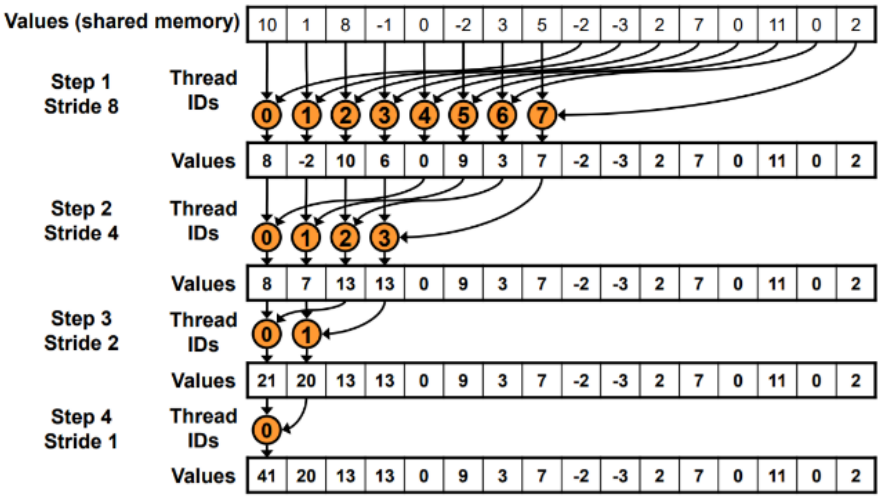


Fig. 2. Illustration of parallel sum reduction algorithm [18]

Every half of the total number of bin histograms is summed (sum reduction) or compared (min or max reduction) to the other half. The process is reduced to half every iteration until all of the element is processed. Loop unrolling can optimize the thread when the processed data is within the thread warp. The illustration of the parallel sum reduction algorithm is shown in Figure 2.

2) Parallel Prefix Sum (Scan) Algorithm

A parallel prefix sum (scan) algorithm can be used to calculate the cumulative sum of the histogram on shared memory. The procedure of parallel prefix sum (scan) algorithm is described as follow:

a) Up-sweep (reduction) phase, sum every bin in the histogram with the bin on its right according to its stride. This step has a computational complexity of $O(\log(N))$. The illustration of the up-sweep (reduction) phase is shown in Figure 3.

b) Set the last bin in the histogram to zero.

c) Down-sweep phase, sum every bin in the histogram with the bin on its right according to its stride. This step also has a computational complexity of $O(\log(N))$. The illustration of the down-sweep phase is shown in Figure 4.

The parallel prefix sum (scan) algorithm has a computational complexity of $O(2\log(N))$ where the $O(\log(N))$ is in the up-sweep phase and the down-sweep phase.

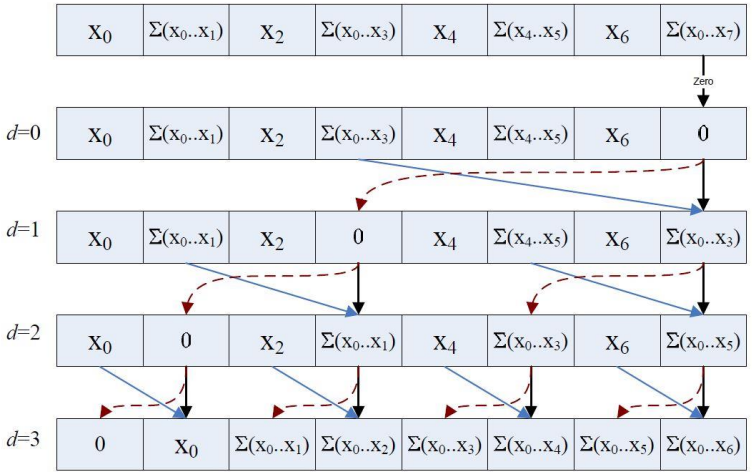


Fig. 3. Illustration of up-sweep (reduction) phase [19]

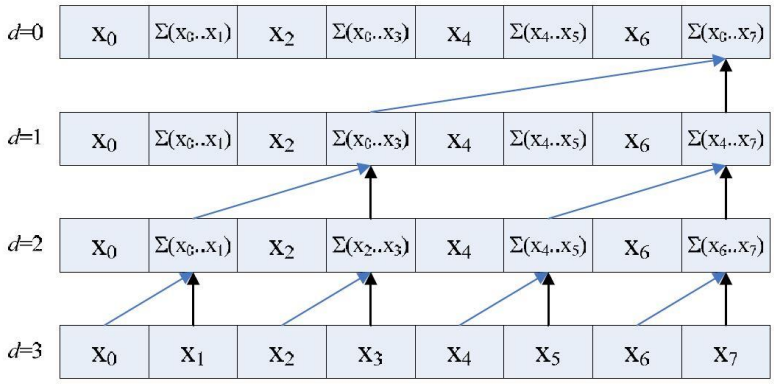


Fig. 4. Illustration of down-sweep phase [19]

III. Result and Discussion

The computational time of adaptive image thresholding algorithms on GPU has been tested on FVC2004 (Fingerprint Verification Competition) dataset [20]. The dataset consists of several fingerprint images. Selected images in the dataset are resized into various sizes using the bi-cubic interpolation method. The proposed approach is built using C++ with an additional CUDA library and runs on Intel Core i7-7700HQ 2.8GHz processor, 16 GB of RAM, and NVidia GeForce GTX 1050. The GPU has Pascal architecture with five streaming multiprocessors and computes capability 6.1.

A. Adaptive Image Thresholding Implementation

In this research, three adaptive image thresholding algorithms are implemented on GPU: Otsu, ISODATA, and MCET. The parallel approach of the three methods is similar except finding the optimum threshold to perform binarization. First, image data must be copied from host to the device memory. Several kernels to compute histogram, probability histogram, and cumulative histogram to find the optimal threshold and apply the threshold in the image are used. Finally, the binary image result is copied back to the host from device memory. The implementation of Otsu, ISODATA, and MCET methods on GPU is shown in Algorithm 1.

As shown in Algorithm 1, the parallel approach of the adaptive image thresholding method uses several kernels to perform a specific operation, will keep short computation runs on streaming multiprocessors and increase its availability. The number of threads per block and the block per grid can be configured to run the kernel effectively. It is also suitable for error handling because it can be monitored on each kernel execution.

Algorithm 1. Implementation of adaptive image thresholding method on GPU.

```
ENUM method ← OTSU = 1, ISODATA = 2, MCET = 3

READ image data and method

COPY image data from host (CPU) to device (GPU)

SET threshold ← 0

histogram ← compute histogram from image data
probability histogram ← compute probability histogram from a histogram
cumulative histogram ← compute cumulative sum histogram from probability histogram

SWITCH (method)
   CASE OTSU
      threshold ← find threshold that maximizes inter-class variance from cumulative sum histogram
         CASE ISODATA
      threshold ← update the threshold until the average inter-class distance is less than a given
         threshold or the maximum number of iteration is reached
   CASE MCET
      above-threshold and below-threshold means ← compute above-threshold and below-threshold means
         from cumulative sum histogram
```

```
      cross-entropy histogram ← compute cross-entropy histogram from above-threshold and below-
         threshold means
      threshold ← compute the index of minimum cross-entropy from cross-entropy histogram
END SWITCH

binary image ← apply threshold to image data

COPY binary image from device (GPU) to host (CPU)
```

The highest computational complexity is $O(N)$ which lies in the histogram computation and image thresholding step. The parallel implementation of these steps will reduce the computational complexity because the work is computed at once and distributed to the total number of threads used for computation. The parallel approach of histogram computation on GPU is shown in Algorithm 2.

Histogram computation uses the atomic addition function from CUDA and utilizes shared memory to store the partial histogram, which will reduce the queue at the addition instruction level to the number of threads block. The partial histogram in shared memory is then merged parallel to the histogram in global memory. This operation also uses atomic addition, which will reduce the queue at the addition instruction level to the number of blocks in a grid.

Without partial histogram computation in shared memory, the histogram computation is likely to have long queues and be forced to perform serial computation. All operations that equal the number of data need to access and performed in addition to one specific bin in the histogram. For the gray-level histogram, the number of histogram bins is fixed to 256. The queue is proportional to the data and their distribution in the image. With partial histogram computation, the queue is reduced to the number of threads and blocks used.

Algorithm 2. The computation of histogram on GPU.

```
GPU CONFIGURATION
   block ← 256// block size
   grid ← 256 // grid size

FUNCTION compute the histogram

READ image data and image size

t ← threadIdx.x
n ← the number of histogram bin

// histogram initialization with zeros
ALLOCATE shared memory (smem) to store the histogram
   IF t < n THEN
      the t^th index of smem histogram ← 0
   END IF
SYNCHRONIZE the threads

p ← threadIdx.x + blockIdx.x * blockDim.x
q ← blockDim.x * gridDim.x

// compute partial histogram in shared memory
WHILE p < image size DO
   r ← the p^th index of image data
   atomic addition of the r^th index of smem histogram with 1
   p ← p + q
END WHILE
SYNCHRONIZE the threads

// merge the partial histogram in shared memory to histogram in global memory
IF t < n THEN
   atomic addition of the t^th index of histogram with the t^th index of smem histogram
END IF

END FUNCTION
```

After the histogram is obtained, the probability histogram is computed by simple division. Otsu method uses the probability of 0^{th} order histogram, computed by dividing the value in every bin with the total number of data and the probability of 1^{st} order histogram computed by multiplying 0^{th} order histogram with the corresponding gray level. MCET method uses the probability of an entropy histogram computed by multiplying first-order histogram with the gray level log. ISODATA method uses the 0^{th} order histogram and 1^{st} order histogram. The kernel configuration is a block with 256 threads to calculate the 256-bins histogram.

The computation of the cumulative sum of histogram uses a parallel prefix sum (scan) algorithm. The computational complexity can be reduced to $O(2\log(N))$ from $O(N)$. To avoid bank conflict, it utilizes half of the histogram bin's total number as thread block and some offsets. Bank conflict occurs when two or more threads want to access the same bank memory address, forcing serial access to memory. With proper offsets, bank conflict can be avoided. The computation of the cumulative sum of a histogram is shown in Algorithm 3.

Algorithm 3. The computation of the cumulative sum of the histogram on GPU.

```
GPU CONFIGURATION
   block ← 256 / 2
   grid ← 1

FUNCTION compute the cumulative sum of histogram

READ probability histogram

t ← threadIdx.x
n ← the number of histogram bin
offset ← 1
p ← t
q ← t + n / 2
offset1 ← p >> 4
offset2 ← q >> 4

// load data to shared memory
ALLOCATE shared memory (smem) to store the cumulative sum of histogram
   the (p + offset1)th index of smem cumulative sum of histogram ← the pth index of
      probability histogram
   the (q + offset2)th index of smem cumulative sum of histogram ← the qth index of
      probability histogram
SYNCHRONIZE the threads

// up-sweep (reduction) phase
FOR d = n >> 1 TO d > 0 DO
   SYNCHRONIZE the threads
   IF t < d THEN
      p ← offset * (2 * t + 1) – 1
      q ← offset * (2 * t + 2) – 1
      p ← p + p >> 4
      q ← q + q >> 4
      the qth index of smem cumulative sum of histogram ← the qth index of smem cumulative
         sum of histogram + the pth index of smem cumulative sum of histogram
   END IF
   offset ← offset * 2
   d ← d >> 1
END FOR

// set the last element to zero
IF t = 0 THEN
   the (n – 1)th index of cumulative sum of histogram ← the (n – 1 + (n – 1) >> 4)th index
      of smem cumulative sum of histogram
   the (n – 1 + (n – 1) >> 4)th index of smem cumulative sum of histogram ← 0
END IF

// down-sweep phase
FOR d = 1 TO d < n DO
```

```
   offset ← offset >> 1
   SYNCHRONIZE the threads
   IF t < d THEN
      p ← offset * (2 * t + 1) – 1
      q ← offset * (2 * t + 2) – 1
      p ← p + p >> 4
      q ← q + q >> 4
      temp value ← the p^th index of smem cumulative sum of histogram
      the p^th index of smem cumulative sum of histogram ← the q^th index of smem cumulative
         sum of histogram
      the q^th index of smem cumulative sum of histogram ← the q^th index of smem cumulative
         sum of histogram + temp value
   END IF
   d ← d * 2
END FOR

// copy data from shared memory to global memory
the p^th index of cumulative sum of histogram ← the (p + 1 + (p + 1) >> 4)^th index of smem
cumulative sum of histogram
IF q < n – 1 THEN
   the q^th index of cumulative sum of histogram ← the (q + 1 + (q + 1) >> 4)^th index of
      smem cumulative sum of histogram
END IF

END FUNCTION
```

Computation to find the optimal threshold from the cumulative sum of the histogram is different for each method. However, a block with 256 threads is used to match the number of histogram bins because all methods are based on a histogram. Otsu method finds a threshold that maximizes inter-class variance can be achieved using a parallel reduction algorithm to find the index of maximum inter-class variance. Algorithm 4 shows the computation of inter-class variance on GPU.

Algorithm 4. The computation of inter-class variances on GPU

```
GPU CONFIGURATION
   block ← 256
   grid ← 1

FUNCTION compute the inter-class variances

READ cumulative sum of 0^th order and 1^st order probability histogram

t ← threadIdx.x
n ← the number of histogram bin

// load data to shared memory
ALLOCATE shared memory (smem) to store the cumulative sum of 0^th and 1^st order probability
histogram
   smem cumulative sum of 0^th order histogram ← cumulative sum of 0^th order probability
      histogram
   smem cumulative sum of 1^st order histogram ← cumulative sum of 1^st order probability
      histogram
   smem value ← 0
   smem index ← t
SYNCHRONIZE the threads

// compute inter-class variances
numerator ← power of two of (the (n – 1)^th index of smem cumulative sum of 1^st order
   histogram * the t^th index of smem cumulative sum of 0^th order histogram – the t^th index of
   smem cumulative sum of 1^st order histogram)
denominator ← the t^th index of smem cumulative sum of 0^th order histogram * (1 – the t^th
   index of smem cumulative sum of 0^th order histogram) + EPSILON)
the t^th index of smem value ← numerator / denominator
SYNCHRONIZE the threads
```

```
// find the index of maximum value of inter-class variance using parallel reduction algorithm
FOR s = blockDim.x / 2 TO s > 0 DO
   IF t < s AND the (t + s)th index of smem value > the tth index of smem value THEN
      the tth index of smem index ← the (t + s)th index of smem index
      the tth index of smem value ← the (t + s)th index of smem value
   END IF
   SYNCHRONIZE the threads
   s ← s >> 1
END FOR

// get the index of maximum value and copy to global memory
IF t = 0 THEN
   threshold ← the 0th index of smem index
END IF

END FUNCTION
```

At each iteration in the ISODATA method, the threads compute the average data below and above the threshold, compute the new threshold, and compare the new threshold with the previous threshold. If the difference of the thresholds is less than a given threshold or the iteration is reached the maximum number of iterations, the optimum threshold is obtained. Algorithm 5 shows the ISODATA computation on GPU.

Algorithm 5. The computation of ISODATA on GPU

```
GPU CONFIGURATION
   block ← 256
   grid ← 1

FUNCTION compute the ISODATA

READ cumulative sum of 0th order and 1st order histogram and maximum number of iteration

t ← threadIdx.x
n ← the number of histogram bin

// load data to shared memory
ALLOCATE shared memory (smem) to store the cumulative sum of 0th order and 1st order histogram
   smem cumulative sum of 0th order histogram ← cumulative sum of 0th order histogram
   smem cumulative sum of 1st order histogram ← cumulative sum of 1st order histogram
   smem means below threshold ← 0
   smem means above threshold ← 0
   smem value ← 0
SYNCHRONIZE the threads

// compute all possible means below-threshold and above-threshold
IF t < n – 1 THEN
   the tth index of smem means below-threshold ← floor ((the tth index of smem cumulative
      sum of 1st order histogram / (the tth index of smem cumulative sum of 0th order histogram
      + EPSILON)) + 0.5)
   numerator ← the (n – 1)th index of smem cumulative sum of 1st order histogram – the (t +
      1)th index of smem cumulative sum of 1st order histogram
   denominator ← the (n – 1)th index of smem cumulative sum of 0th order histogram – the (t
      + 1)th index of smem cumulative sum of 0th order histogram + EPSILON
   the tth index of smem means above-threshold ← floor ((numerator / denominator) + 0.5)
END IF
SYNCHRONIZE the threads

// compute the average inter-class means
the tth index of smem value ← floor (((the tth index of smem means below-threshold + the tth
   index of smem means above-threshold) / 2) + 0.5)
SYNCHRONIZE the threads
```

```
// compute the difference between the current threshold and the previous threshold
IF t = 0 THEN
   iteration ← 0
   difference ← 1
   T ← floor ((the (n – 1)th index of cumulative sum of 1st order histogram / (the (n – 1)th
      index of cumulative sum of 0th order histogram + EPSILON)) + 0.5)
   WHILE difference > 0 AND iteration < maximum number of iteration DO
      threshold ← the Tth index of smem value
      difference ← absolute of (the Tth index of smem value – threshold)
      T ← the Tth index of smem value
      iteration ← iteration + 1
   END WHILE
END IF
SYNCHRONIZE the threads

END FUNCTION
```

The cross-entropy computation uses a parallel sum reduction algorithm to compute the sum above-threshold and below-threshold entropy from the histogram. The sum is used to compute the entropy histogram. To compute all possible thresholds in parallel (iterates through all possible thresholds while performing parallel sum reduction algorithm to compute the sum above-threshold and below-threshold), the configuration is set to use a block with 256 threads and a grid with 256 blocks. Algorithm 6 shows the cross-entropy computation on GPU.

Algorithm 6. The computation of cross-entropy on GPU.

```
GPU CONFIGURATION
   block ← 256
   grid ← 256

FUNCTION compute the cross-entropy

READ 0th order probability histogram, cumulative sum of 0th order and 1st order probability
histogram

b ← blockIdx.x
t ← threadIdx.x
n ← the number of histogram bin

// load data to shared memory
ALLOCATE shared memory (smem) to store the sum and entropy below-threshold and above-
threshold
   data below-threshold ← the bth index of cumulative sum of 1st order probability histogram
      / (the bth index of cumulative sum of 0th order probability histogram + EPSILON)
   data above-threshold ← (the (n-1)th index of cumulative sum of 1st order probability
      histogram – the bth index of cumulative sum of 1st order probability histogram) / (the
      (n-1)th index of cumulative sum of 0th order probability histogram – the bth index of
      cumulative sum of 0th order probability histogram + EPSILON)
   the tth index of smem below-threshold entropy ← 0
   the tth index of smem above-threshold entropy ← 0
SYNCHRONIZE the threads

// compute entropy above-threshold and below-threshold
IF t > b AND data above-threshold > 0 THEN
   the tth index of smem above-threshold entropy ← (t + 1) * the tth index of 0th order
      probability histogram * log of (data above-threshold)
END IF

IF t <= b AND data below-threshold > 0 THEN
   the tth index of smem below-threshold entropy ← (t + 1) * the tth index of 0th order
      probability histogram * log of (data below-threshold)
END IF
SYNCHRONIZE the threads
```

```
// perform parallel sum reduction
FOR s = b / 2 TO s > 0 DO
   IF t < s THEN
      the t^th index of smem above-threshold entropy ← the t^th index of smem above-threshold
         entropy + the (t + s)^th index of smem above-threshold entropy
      the t^th index of smem below-threshold entropy ← the t^th index of smem below-threshold
         entropy + the (t + s)^th index of smem below-threshold entropy
   END IF
   SYNCHRONIZE the threads
END FOR

// compute cross-entropy
IF t = 0 THEN
   the b^th index of cross-entropy histogram ← global entropy – the 0^th index of smem above-
      threshold entropy – the 0^th index of smem below-threshold entropy
END IF

END FUNCTION
```

Finding the index of minimum cross-entropy can be done using a parallel reduction algorithm that compares half of the histogram bins with the other half of the histogram bins. The number of histogram bins is reduced for every iteration. Algorithm 7 shows the computation to find the index of minimum cross-entropy on GPU.

Algorithm 7. The computation to find the index of minimum cross entropy on GPU.

```
GPU CONFIGURATION
   block ← 256
   grid ← 1

FUNCTION find the index of minimum cross-entropy

READ   cross-entropy histogram

t ← threadIdx.x

// load data to shared memory
ALLOCATE shared memory (smem) to store the cross-entropy histogram
   smem cross-entropy histogram ← cross-entropy histogram
   smem index ← t
SYNCHRONIZE the threads

// find index of minimum value using reduction
FOR s = blockDim.x / 2 TO s > 0 DO
   IF t < s AND the (t + s)^th index of smem cross-entropy histogram < the t^th index of smem
      cross-entropy histogram THEN
      the t^th index of smem cross-entropy histogram ← the (t + s)^th index of smem cross-
         entropy histogram
      the t^th index of smem index ← the (t + s)^th index of smem index
   END IF
   SYNCHRONIZE the threads
   s ← s >> 1
END FOR

// copy the result to global memory
IF t = 0 THEN
   threshold ← the 0^th index of smem index
END IF

END FUNCTION
```

The implementation of image thresholding is parallelized using thread-level parallelism on GPU. The approach is practical because the operation is independent for each pixel. The result of image

thresholding is a binary image "1" for pixels above the threshold and "0" for pixels below the threshold. Algorithm 8 shows the implementation of image thresholding on GPU.

Algorithm 8. The implementation of image thresholding on GPU.

```
GPU CONFIGURATION
   block ← 256
   grid ← 256

FUNCTION apply the threshold on image

READ image data, image size and threshold

t ← threadIdx.x + blockIdx.x * blockDim.x
s ← blockDim.x * gridDim.x

// create binary image using image thresholding
WHILE t < image size DO
   IF the t^th index of image data < threshold THEN
      the t^th index of binary image ← 0
   ELSE
      the t^th index of binary image ← 1
   END IF
   t ← t + s
END WHILE

END FUNCTION
```

B. Adaptive Image Thresholding Result

The parallel adaptive image thresholding method is tested on selected images from the FVC2004 (Fingerprint Verification Competition) dataset [20]. The result of adaptive image thresholding implementation is the binary image as shown in Figure 5 where (a) is the fingerprint image, (b) is the binary image generated by the Otsu method with threshold = 154, (c) is the binary image generated by ISODATA method with threshold = 156 and (d) is the binary image generated by MCET method with threshold = 123. As shown in Figure 5, the methods produce a different optimal threshold because the algorithm to search the optimum threshold is also different.

C. Computational Time Evaluation

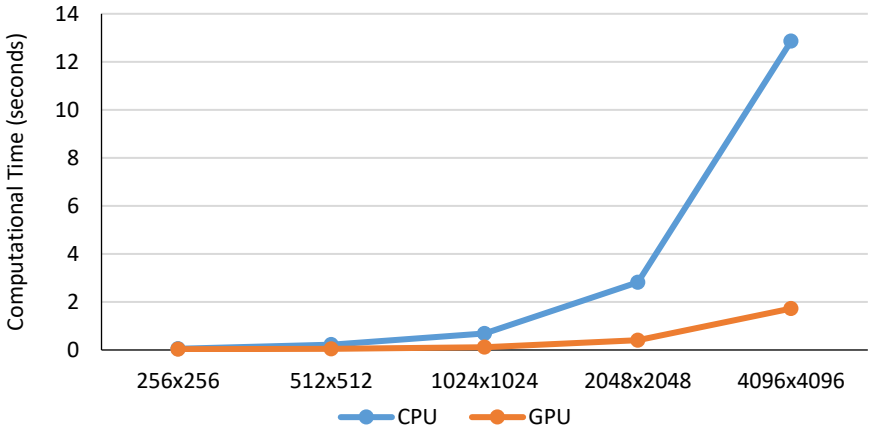
The test was conducted on selected images from FVC2004 (Fingerprint Verification Competition) dataset [20]. The images are resized to generate various image sizes, namely 256×256, 512×512, 1024×1024, 2048×2048, and 4096×4096. The purpose of this experiment is to measure the computational time of the proposed parallel approach of adaptive image thresholding methods when dealing with a large number of data (pixels).

The computational time evaluation on CPU and GPU is shown in Figure 6 where (a) Otsu method, (b) ISODATA method, and (c) MCET method. The proposed parallel approach gains speedup 4-6 times than CPU implementation from implementing adaptive image thresholding methods on GPU.

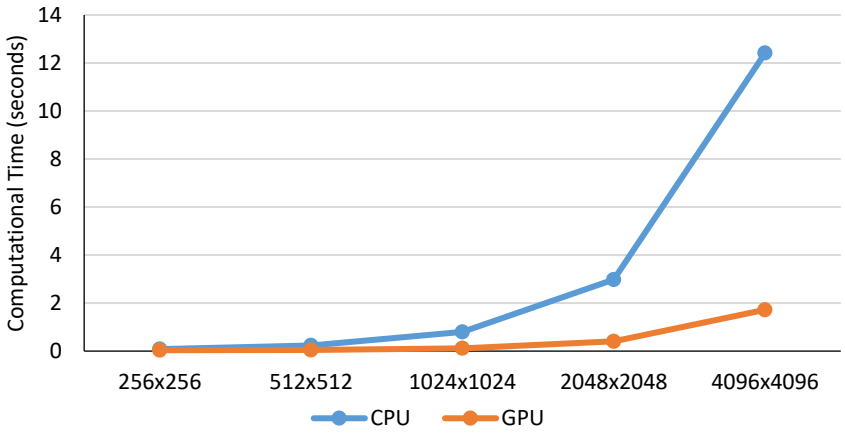
(a) (b) (c) (d)

Fig. 5. The result of adaptive image thresholding implementation; (a) fingerprint image; (b) Otsu method with threshold = 154; (c) ISODATA method with threshold = 156; and (d) MCET method with threshold = 123

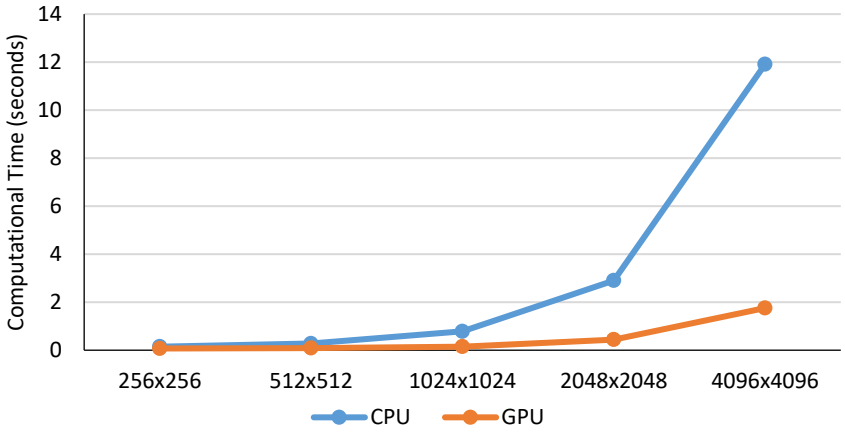
The performance significantly increases when dealing with larger data. The result shows that the parallel approach of the adaptive image thresholding method on GPU allows image segmentation to be processed in real-time, even when dealing with a large resolution of the image.



(a)



(b)



(c)

Fig. 6. Performance evaluation of adaptive image thresholding implementation; (a) performance comparison of Otsu method implementation on CPU and GPU; (b) performance comparison of ISODATA method implementation on CPU and GPU; and (c) performance comparison of MCET method implementation on CPU and GPU

IV. Conclusion

Image processing applications, for example, perform segmentation, usually requiring high-resolution images such as satellite, aerial, biometric, or medical images as the input. The segmentation method, which involves per pixel operation and iterative procedure, can be costly in handling many data/pixels in the high-resolution image. Therefore, this research proposed a parallel approach of adaptive image thresholding algorithms, namely Otsu, ISODATA, and minimum cross-entropy on GPU to deal with high-resolution images. The experiment was conducted on selected fingerprint images taken from FVC2004 (Fingerprint Verification Competition) dataset. From the experiment with the various scale of image resolutions, GPU implementation's computational time shows 4-6 times more speed up than CPU implementation. The performance is significantly increased when dealing with larger image resolution. This result shows that the parallel approach allows image segmentation to be processed in real-time, even when dealing with large image resolution. The contributions are shown in the analysis result of the adaptive image thresholding algorithms that can be optimized using the parallel approach to produce a significant speedup in a computational time when dealing with a high-resolution image. In future work, the proposed parallel approaches will be further optimized using multi-GPUs and implemented in more complex cases such as the segmentation of aerial or medical images.

Acknowledgment

This research is supported by LPPM Universitas Ahmad Dahlan research grant no. PF-062/SP3/LPPM-UAD/VI/2018.

Declarations

Author contribution

All authors contributed equally as the main contributor of this paper. All authors read and approved the final paper.

Funding statement

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Conflict of interest

The authors declare no known conflict of financial interest or personal relationships that could have appeared to influence the work reported in this paper.

Additional information

Reprints and permission information are available at http://journal2.um.ac.id/index.php/keds.

Publisher's Note: Department of Electrical Engineering - Universitas Negeri Malang remains neutral with regard to jurisdictional claims and institutional affiliations.

References

[1] R. C. Gonzalez and R. E. Woods, *Digital image processing*. Prentice Hall, 2008.

[2] N. Otsu, "A Threshold Selection Method from Gray-Level Histograms," *IEEE Trans. Syst. Man. Cybern.*, vol. 9, no. 1, pp. 62–66, Jan. 1979.

[3] G. H. Ball, D. J. Hall, and S. R. Institute, *Isodata: A Method of Data Analysis and Pattern Classification*. Stanford Research Institute, 1965.

[4] C. H. Li and C. K. Lee, "Minimum cross entropy thresholding," *Pattern Recognit.*, vol. 26, no. 4, pp. 617–625, Apr. 1993.

[5] A. M. A. Talab, Z. Huang, F. Xi, and L. HaiMing, "Detection crack in image using Otsu method and multiple filtering in image processing techniques," *Optik (Stuttg).*, vol. 127, no. 3, pp. 1030–1033, Feb. 2016.

[6] Z. He and L. Sun, "Surface defect detection method for glass substrate using improved Otsu segmentation," *Appl. Opt.*, vol. 54, no. 33, p. 9823, Nov. 2015.

[7] Y. Feng, H. Zhao, X. Li, X. Zhang, and H. Li, "A multi-scale 3D Otsu thresholding algorithm for medical image segmentation," *Digit. Signal Process.*, vol. 60, pp. 186–199, Jan. 2017.

[8] P. Zhang *et al.*, "Multi-component segmentation of X-ray computed tomography (CT) image using multi-Otsu thresholding algorithm and scanning electron microscopy," *Energy Explor. Exploit.*, vol. 35, no. 3, pp. 281–294, May 2017.

[9] S. Sarkar, S. Das, and S. S. Chaudhuri, "A multilevel color image thresholding scheme based on minimum cross entropy and differential evolution," *Pattern Recognit. Lett.*, vol. 54, pp. 27–35, Mar. 2015.

[10] D. Oliva, S. Hinojosa, V. Osuna-Enciso, E. Cuevas, M. Pérez-Cisneros, and G. Sanchez-Ante, “Image segmentation by minimum cross entropy using evolutionary methods,” *Soft Comput.*, pp. 1–20, Aug. 2017.

[11] T. Kaur, B. S. Saini, and S. Gupta, "Optimized Multi Threshold Brain Tumor Image Segmentation Using Two Dimensional Minimum Cross Entropy Based on Co-occurrence Matrix," Springer, Cham, 2016, pp. 461–486.

[12] S. Hemalatha and S. M. Anouncia, "Unsupervised segmentation of remote sensing images using FD based texture analysis model and ISODATA," *Int. J. Ambient Comput. Intell.*, vol. 8, no. 3, pp. 58–75, 2017.

[13] P. Kanungo, P. K. Nanda, and A. Ghosh, "Parallel genetic algorithm based adaptive thresholding for image segmentation under uneven lighting conditions," in *2010 IEEE International Conference on Systems, Man and Cybernetics*, 2010, pp. 1904–1911.

[14] M. Sandeli and M. Batouche, "Multilevel thresholding for image segmentation based on parallel distributed optimization," in *2014 6th International Conference of Soft Computing and Pattern Recognition (SoCPaR)*, 2014, pp. 134–139.

[15] M. H. Najafi, A. Murali, D. J. Lilja, and J. Sartori, "GPU-Accelerated Nick Local Image Thresholding Algorithm," in *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, 2015, pp. 576–584.

[16] P. K. Upadhyay, S. Chandra, and A. Sharma, "A novel approach of adaptive thresholding for image segmentation on GPU," in *2016 Fourth International Conference on Parallel, Distributed and Grid Computing (PDGC)*, 2016, pp. 652–655.

[17] J. Fung and S. Mann, "Using graphics devices in reverse: GPU-based Image Processing and Computer Vision," in *2008 IEEE International Conference on Multimedia and Expo*, 2008, pp. 9–12.

[18] M. Harris, "Optimizing cuda," *SC07 High Perform. Comput. With CUDA*, p. 18, 2007.

[19] Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," *GPU gems*, vol. 3, no. 39, pp. 851–876, 2007.

[20] D. Maltoni, D. Maio, A. K. Jain, and S. Prabhakar, *Handbook of fingerprint recognition*. Springer Science & Business Media, 2009.