Intelligent Automation And Soft Computing, 2019

Vol.25,no.1,79 –90

A Distributed Heterogeneous Inspection System for High Performance In-line Surface Defect Detection

Yu-Cheng Chou[1], Wei-Chieh Liao[2], Yan-Liang Chen[2], Ming Chang[2], and Po Ting Lin[3]

[1] Institute of Undersea Technology, National Sun Yat-sen University, Kaohsiung, Taiwan
[2] Department of Mechanical Engineering, Chung Yuan Christian University, Taoyuan, Taiwan
[3] Department of Mechanical Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan

ABSTRACT
This paper presents the Distributed Heterogeneous Inspection System (DHIS), which comprises two CUDA workstations and is equipped with CPU distributed computing, CPU concurrent computing, and GPU concurrent computing functions. Thirty-two grayscale images, each with 5,000×12,288 pixels and simulated defect patterns, were created to evaluate the performances of three system configurations: (1) DHIS; (2) two CUDA workstations with CPU distributed computing and GPU concurrent computing; (3) one CUDA workstation with GPU concurrent computing. Experimental results indicated that: (1) only DHIS can satisfy the time limit, and the average turnaround time of DHIS is 37.65% of the time limit; (2) a good linear relationship exists between the processing speed ratio and the instruction sequence quantity ratio.

KEY WORDS: In-line surface defect detection, concurrent computing, distributed computing, POSIX threads, compute unified device architecture, message passing interface.

1 INTRODUCTION

EVERY manufactured product ideally needs to be tested before delivery to ensure that it satisfies the requirements of customers and has sufficient reliability. Due to high costs of testing the quality of a product, manufacturers often test only a small portion of a batch of products. However, for sensitive components, such as integrated circuits and glass panels, in-line inspection is strictly enforced in the early stages of manufacturing to enable corrective actions, leading to the improvement of overall productivity. More than two decades ago, Park and Bien (1995) designed a machine vision system using programmable hardware for industrial inspection applications, such as the liquid-crystal display (LCD) panel inspection.

Nowadays, the machine vision technology has been adopted as a non-destructive test protocol for reliable inspection and localization of surface defects in the manufacturing industry. The machine vision technology relies on image analysis to enable useful and effective functions, such as automated optical inspection, remote navigation, and dynamic visual recognition. In the past decade, numerous machine vision techniques have been proposed for surface defect inspection.

Some recently reported methods and systems are summarized as follows. Ryu et al. (2014) designed a machine vision system with a line scan camera to detect texture differences and brightness differences between well-scarfed and poorly-scarfed steel slab surfaces. Busin et al. (2013) designed a line scan color vision system to detect printing flaws that appear on the color surfaces of drinking glasses decorated in a silk-screen process. Lin et al. (2013) proposed a laser reflection point inspection method, combining fuzzy rules with an artificial neural network, for chemical stain inspection on the surface of polysilicon solar wafers. Tsai et al. (2012) proposed a regularity measure as the only discrimination feature to detect ill-defined subtle defects on non-textured and homogeneously textured surfaces. Li et al. (2012) proposed a local annular contrast based image processing algorithm to find defects on steel bar surfaces. Tsai et al. (2012) proposed a dissimilarity measure based on the optical-flow technique for defect inspection on light-emitting diode (LED) wafer die

CONTACT Yu-Cheng Chou ycchou@mail.nsysu.edu.tw

surfaces. Li and Tsai (2011) developed a machine vision based scheme to automatically detect saw-mark defects on solar wafer surfaces. Tsai and Tsai (2011) proposed an optical flow-based motion analysis scheme to detect low-contrast blemishes on LCD panels. Zhang et al. (2011) designed a multi-class support vector machine (SVM) based vision system for defect inspection of strongly reflective metal surfaces. Michaeli and Berdel (2011) proposed an in-line inspection algorithm for textured plastic surfaces produced in a high speed continuous process. Tsai et al. (2011) developed a robot vision system for ill-defined anomaly detection on surfaces of 3D objects. Tian et al. (2011) proposed a multiple classifier system, based on SVM and stacked generalization, for wheat disease diagnosis through pattern recognition on the surface of wheat leaves. Rosati et al. (2009) designed a real-time defect detection system for highly reflective curved surfaces of coated plastic components produced in the automotive industry. In addition, Chiou et al. (2011) and Tsai et al. (2010) proposed machine vision techniques to inspect micro-crack defects in the solar wafer manufacturing process.

In terms of screening products based on pre-defined criteria, the above techniques generate satisfactory performance in different applications of surface defect detection. However, the above techniques have a compromise between the speed and resolution. In other words, the above techniques avoid a situation where hundreds of mega pixels have to be processed per second continuously. Such a compromise between the speed and resolution generates a bottleneck for the applications of in-line surface inspection.

To this end, in our previous work (Chang et al., 2014), we presented the first non-destructive optical inspection system equipped with a Compute Unified Device Architecture (CUDA) workstation to achieve satisfactory performance in both the speed and resolution for the extraction and labeling of micro-sized surface defects. The performance of a single optical inspection system in the above work was tested using a back-coated mirror object with 43 mm in width and 70 mm in length. However, surface defect detection for a larger object, under the same speed and resolution requirements, was not addressed in our previous work. Therefore, this work aims to expand our previous work to deal with such a more challenging situation.

Table 1 shows the comparisons between this work, our previous work, and the above cited works. As shown in Table 1, due to the resolution requirement and the test object size, the image sizes concerned in this work and our previous work are much larger than those in the cited works. When taking the time limits into consideration, the speed requirements in this work and our previous work are calculated as 289.13 ($=80{,}000\times24{,}576\times10^{-6}/6.8$) mega pixels/sec and 144.56 ($=20{,}000\times12{,}288\times10^{-6}/1.7$) mega pixels/sec, respectively. As shown in Table 1, the results in the cited works are all obtained through single central processing unit (CPU) based algorithms. Moreover, according to Table 1, the largest speed in the cited works is calculated as 40.33 ($=512\times1{,}024\times10^{-6}/0.013$) mega pixels/sec. Thus, the single CPU-based methods in the cited works will not satisfy the time limits applied to this work and our previous work.

As previously mentioned, this work aims to expand our previous work, in order to handle surface defect detection for a larger object under the same resolution and speed requirements as those in our previous work. As shown in Table 1, in terms of hardware, two graphics processing units (GPUs) and two CPUs are employed in this work. More precisely, two computers, each of which is equipped with a CPU and a CUDA GPU, are used to simultaneously perform surface defect detection on an object eight times as large as that in our previous work. Additionally, this work and our previous work are targeted toward a production line scenario, i.e. screening out defective surfaces in real-time. Hence, this work and our previous work focus on completing defect detection within the time limits and consider defect recognition as an off-line task. Defect recognition is equivalent to image pattern recognition. Moreover, edge detection is considered a fundamental step in image pattern recognition techniques (Umbaugh, 2016). To this end, the defect detection algorithms in this work and our previous work are developed to return the defect quantity indexes, the defect size, and the image data with identified defect edges.

The line scan camera in our previous work is set up such that the camera captures and loads an image of 20,000×12,288 pixels to a CPU in 1.7 seconds. Additionally, due to the maximum CUDA threads allowable on a CUDA GPU, an image that can be simultaneously processed by CUDA threads has a maximum size of 5,000×12,288 pixels. Hence, an image of 20,000×12,288 pixels is divided into four separate sub-images, each of which has 5,000×12,288 pixels. The time, 0.487 second shown in Table 1, is the summation of the four time periods elapsed to complete only the defect detection process on the four sub-images. In other words, the time, 0.487 second, excludes the time used to store the four output sub-images with extracted edges.

The image acquisition time and the maximum number of CUDA threads applied to our previous work are also applied to this work. As mentioned earlier, the image size handled in this work is eight times as large as that in our previous work. Meanwhile, it is necessary to store the processed sub-images for further off-line operations such as defect recognition. Therefore, as shown in Table 1, this work is targeted at a situation where a time limit of 6.8 seconds is required to perform defect detection on 32 sub-images, each of which has 5,000×12,288 pixels, and to save 32 processed sub-images as separate files.

Table 1. Comparisons between this work and cited works

	GPU	CPU	Image size (pixel)	Time limit (sec)	Time result (sec)
This work	2	2	80,000×24,576	6.8	To be presented
Chang et al. (2014)	1	1	20,000×12,288	1.7	0.487
Ryu et al. (2014)	n/a	1	4,096×1,000	n/a	n/a
Busin et al. (2013)	n/a	1	1,320×1,947	n/a	1
Lin et al. (2013)	n/a	1	640×480	n/a	n/a
D. M. Tsai, Chen, et al. (2012)	n/a	1	400×400	n/a	0.032
W. B. Li et al. (2012)	n/a	1	512×1,024	n/a	0.013
D. M. Tsai, Chiang, et al. (2012)	n/a	1	115×105	n/a	0.012
W. C. Li & Tsai (2011)	n/a	1	1,560×1,560	n/a	9.3
D. M. Tsai & Tsai (2011)	n/a	1	200×200	n/a	0.05
Zhang et al. (2011)	n/a	1	256×256	n/a	n/a
Michaeli & Berdel (2011)	n/a	1	512×512	n/a	0.102
Y. H. Tsai et al. (2011)	n/a	1	1,600×1,200	n/a	3.55
Tian et al. (2011)	n/a	1	640×480	n/a	n/a
Rosati et al. (2009)	n/a	1	782×582	n/a	n/a
Chiou et al. (2011)	n/a	1	640×480	n/a	0.18
D. M. Tsai et al. (2010)	n/a	1	1,000×1,000	n/a	0.36

To tackle the above situation, this work employs a distributed memory system consisting of two computers, each of which is equipped with a CPU and a CUDA GPU, as the hardware structure. When applying this distributed memory system to a real production line, each CUDA computer will control a line scan camera to capture images of different portions of the same object surface. Moreover, information of surface defects obtained at each CUDA computer has to be integrated to allow for decision making on whether or not the test object is a defective one. The Message Passing Interface (MPI) (Gropp et al., 1996) is a well-established industry standard for communication among processes that model a parallel program running on a distributed memory system. Therefore, as the best option, this work leverages the MPI standard to tackle the synchronization and data transmission among inspection tasks running on distributed CUDA computers.

In our previous work, a surface inspection procedure is composed of the CPU and GPU operations executed on a single CUDA computer. Each CPU operation is executed by the CPU, and each GPU operation is executed by the CUDA GPU through a large number of parallel threads. Despite that each GPU operation is carried out by parallel threads, all the CPU and GPU operations are still executed one after another. In other words, our previous work adopts a sequential computing model.

The objective of this work is to achieve in-line surface inspection on an object, which is eight times as large as that in our previous work, through a distributed system consisting of two CUDA computers. Thus, on each CUDA computer in this work, the image size to be processed is 80,000×12,288 pixels, which is four times as large as that in our previous work. Owing to the maximum number of CUDA threads allowable on the GPU, the image on each CUDA computer must be divided into 16 sub-images, each with 5,000×12,288 pixels, for GPU computing purposes. Moreover, on each CUDA computer, defect detection on the 16 sub-images and saving the 16 processed sub-images as separate files must be completed within 6.8 seconds. The sequential computing model adopted in our previous work cannot satisfy such a speed requirement. Thus, in this work, the CPU and GPU operations on each CUDA computer should be managed to handle different parts of the image data during the same time period. Hence, this work employs a concurrent computing model to allow the CPU and GPU operations to run concurrently on each CUDA computer.

The contributions of this work are summarized as follows: (1) based on the first non-destructive CUDA-enabled optical inspection system established by the authors, this paper presents the first distributed version of the system to tackle in-line surface defect detection for a large object; (2) to utilize the presented multi-CPU and multi-GPU distributed hardware structure for in-line surface inspection, this paper proposes a hybrid computing model as the software structure; the CPU and GPU operations running on a single CUDA computer are designed based on a concurrent computing model, and the CPU operations running on multiple CUDA computers are designed based on a distributed computing model; (3) to implement the proposed hybrid computing model, this paper integrates three different programming models: multithreaded programming with POSIX threads (Pthreads), parallel programming with CUDA, and parallel programming with MPI.

2 HARDWARE AND SOFTWARE SYSTEMS

2.1 Hardware System

THE hardware system proposed in our previous work is shown in Figure 1. The hardware system

consists of a single CUDA workstation equipped with a line scan camera. The resolution requirement is that each pixel represents an area of 3.5×3.5 μ m^2. The camera is a 12288-pixel line scan camera with a 12 kHz acquisition rate. Based on the maximum image size affordable by the camera's onboard memory, the size of the mirror object is chosen to be 70×43 mm^2. Additionally, the image of the test object has 20,000×12,288 pixels. It takes 1.7 seconds to capture the object image and load the image data to the computer's main memory. As mentioned earlier, the image is divided and saved as four sub-images, each of which has 5,000×12,288 pixels. In our previous work, 1.7 seconds is chosen as the time limit to complete surface inspection on the mirror object, leading to a speed requirement of 144.56 mega pixels/sec. The experimental result shows that it takes 0.487 second to complete defect detection on the four sub-images. However, 0.487 second does not include the time to save the four output sub-images with extracted edges.

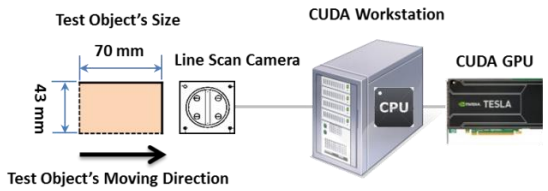


Figure 1. **Hardware system in our previous work.**

The hardware system of DHIS is expanded from the one proposed in our previous work. As shown in Figure 2, the hardware system of DHIS consists of two CUDA workstations. Each CUDA workstation is equipped with a line scan camera. A local area network (LAN) interconnects the two CUDA workstations through a high speed Ethernet switch. The size of a test object in this work is 280×86 mm^2, which is eight times as large as that in our previous work. In addition, the resolution requirement in this work is 3.5×3.5 μ m^2/pixel, the same as that in our previous work. As a result, the image size of a test object is 80,000×24,576 pixels. Each CUDA workstation of DHIS is configured to handle an area of 280×43 mm^2, corresponding to an image of 80,000×12,288 pixels. As previously mentioned, it takes 1.7 seconds to capture and load an image of 20,000×12,288 pixels to the CUDA workstation's main memory, and such an image has the maximum pixels that can be accommodated within the camera's on-board memory. Therefore, in this work, an image of 20,000×12,288 pixels will be loaded to each CUDA workstation every 1.7 seconds, meaning that a time limit of 1.7 seconds must be guaranteed on each CUDA workstation to complete defect detection on such an image and storage of the processed image. Moreover, the two CUDA workstations of DHIS are managed to operate synchronously. Hence, from the perspective of DHIS, in order to achieve in-line defect detection on an object of 280×86 mm^2, a total of 6.8 seconds must be guaranteed to complete the following tasks: (1) defect detection on an image of 80,000×12,288 pixels at each CUDA workstation; (2) integration of the defect information obtained at each CUDA workstation; (3) storage of the processed sub-images at each CUDA workstation. As a result, the speed requirement in this work is calculated as 289.13 mega pixels/sec.

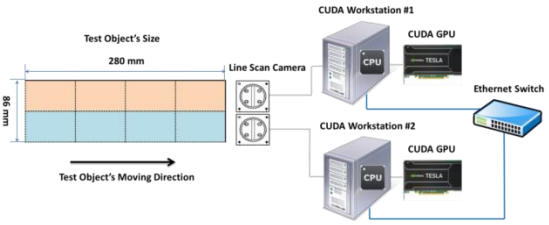


Figure 2. **Hardware system of DHIS.**

From the perspective of hardware, DHIS is a distributed system consisting of individual workstations, each of which contains two different processors, including the CPU and GPU. Therefore, DHIS has the distributed heterogeneous nature in the hardware. More importantly, the hardware system of DHIS is highly expandable to accommodate more computational resources to satisfy more rigorous requirements imposed on the speed and resolution.

2.2 *Software System*

TO utilize the presented multi-CPU and multi-GPU distributed hardware structure for in-line surface inspection, a hybrid computing model is proposed as the software structure. The hybrid computing model combines a concurrent computing model and a distributed computing model. The CPU and GPU operations running on each CUDA workstation are designed based on a concurrent computing model. On the other hand, the CPU operations running on the distributed CUDA workstations are designed based on a distributed computing model.

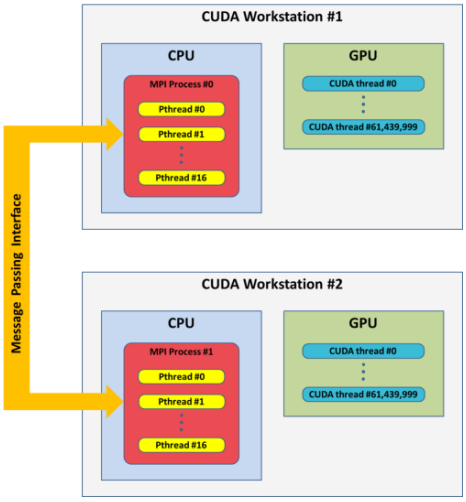


Figure 3. **Software system of DHIS.**

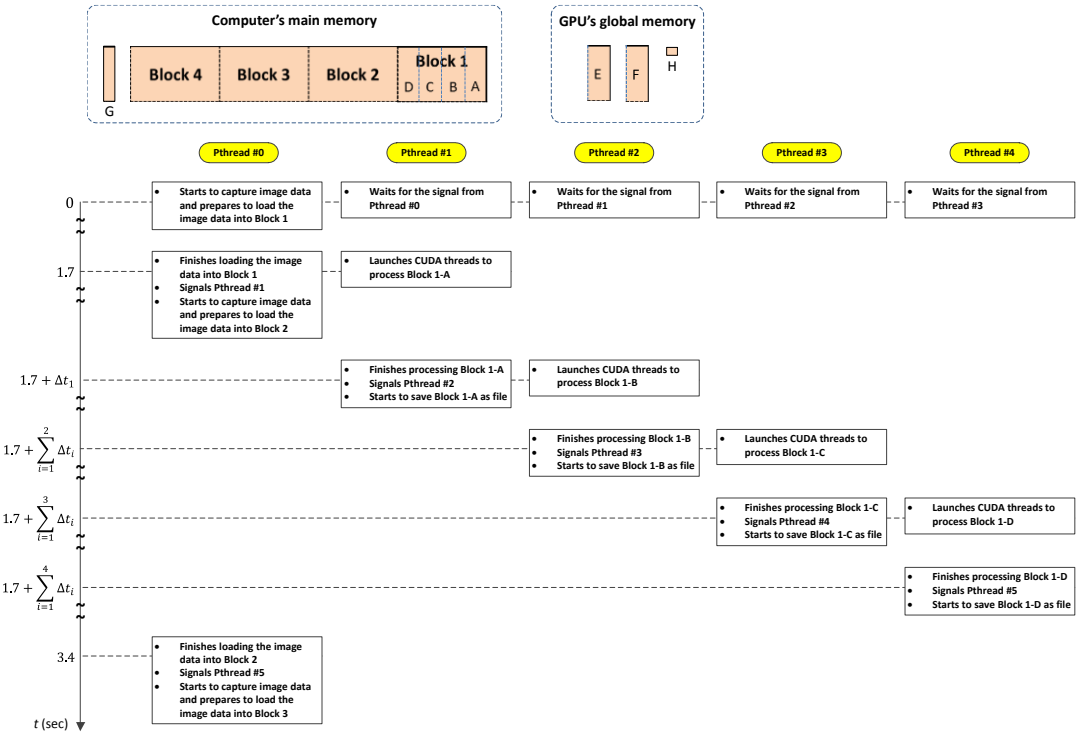


Figure 4. Concurrent computing scheme on each CUDA workstation.

To implement the proposed hybrid computing model, this paper integrates three different programming models: multithreaded programming with Pthreads, parallel programming with CUDA, and parallel programming with MPI. Therefore, as shown in Figure 3, the software system of DHIS comprises Pthreads, CUDA threads, and MPI processes. On each CUDA workstation, an MPI process and 17 Pthreads are executed by the CPU, and a total of 61,440,000 CUDA threads are executed by the GPU. Moreover, as shown in Figure 4, a concurrent computing scheme is adopted on each CUDA workstation to manage the Pthreads and CUDA threads for image data processing.

The operations of five Pthreads, including Pthread #0 to Pthread #4, within 3.4 seconds from the initial time point are shown in Figure 4. Each CUDA workstation is in charge of 1/2 of the object. Block 1 to Block 4 are memory blocks allocated in the computer's main memory to store the image data at runtime. Each block stores 1/8 of the entire object image. Additionally, each block consists of four sub-blocks, such as Block 1-A to Block 1-D, and therefore each sub-block stores 1/32 of the entire object image.

Block E and Block F, each with the same size as a sub-block in the computer's main memory, are memory blocks allocated in the GPU's global memory for multithreaded operations on the CUDA GPU. Block E stores the image data that are sent from the CPU and will be processed by the CUDA threads to obtain the defect quantity indexes, the defect size, and the image data with identified edges. Block F stores the image data with identified edges that will be sent back to the corresponding sub-block in the computer's main memory.

Block H and Block G, which are not drawn to scale in Figure 4, are memory blocks to accommodate three integers in the GPU's global memory and 48 integers in the computer's main memory, respectively. In this work, the defect quantity indexes include the number of defect initial pixels and the number of defect terminal pixels, and the defect size is equivalent to the number of defect pixels. When the above three integers for the image data in Block E are obtained, they will be stored in Block H and sent back to the corresponding locations in Block G.

The concurrent computing scheme shown in Figure 4 is illustrated as follows:

(1) At $t = 0$: Pthread #0 starts to capture the image of the first 1/8 of the object and prepares to load the image data into Block 1. Among Pthread #1 to Pthread #4, each Pthread waits for the signal from the Pthread with an ID one less than its own.

(2) At $t = 1.7$: Pthread #0 finishes loading the image data into Block 1 and thereby signals Pthread #1. Pthread #0 then starts to capture the image of the next 1/8 of the object and prepares to load the image data into Block 2. Pthread #1 receives the signal from Pthread #0, and it then copies the image data from Block 1-A to Block E and launches CUDA threads to perform defect detection operations.

(3) At $t = 1.7 + \Delta t_1$: Pthread #1 obtains the defect information kept in Block G and the output image data

kept in Block 1-A, and thereby signals Pthread #2. Pthread #1 then starts to save the output image data as a file. Pthread #2 receives the signal from Pthread #1, and then it copies the image data from Block 1-B to Block E and launches CUDA threads to perform defect detection operations.

(4) At $t = 1.7 + \sum_{i=1}^{2} \Delta t_i$: Pthread #2 obtains the defect information kept in Block G and the output image data kept in Block 1-B, and thereby signals Pthread #3. Pthread #2 then starts to save the output image data as a file. Pthread #3 receives the signal from Pthread #2, and then it copies the image data from Block 1-C to Block E and launches CUDA threads to perform defect detection operations.

(5) At $t = 1.7 + \sum_{i=1}^{3} \Delta t_i$: Pthread #3 obtains the defect information kept in Block G and the output image data kept in Block 1-C, and thereby signals Pthread #4. Pthread #3 then starts to save the output image data as a file. Pthread #4 receives the signal from Pthread #3, and then it copies the image data from Block 1-D to Block E and launches CUDA threads to perform defect detection operations.

(6) At $t = 1.7 + \sum_{i=1}^{4} \Delta t_i$: Pthread #4 obtains the defect information kept in Block G and the output image data kept in Block 1-D, and thereby signals Pthread #5. Pthread #4 then starts to save the output image data as a file.

(7) At $t = 3.4$ seconds: Pthread #0 finishes loading the image data into Block 2 and thereby signals Pthread #5. Pthread #0 then starts to capture the image of the next 1/8 of the object and prepares to load the image data into Block 3.

The idea of the presented concurrent computing scheme is that the time used to obtain the defect quantity indexes and the defect sizes for 1/8 of the object, such as $\sum_{i=1}^{4} \Delta t_i$ in Figure 4, can be limited within 1.7 seconds by decoupling itself from the time used to save the output image data as four individual files. Moreover, once the last 1/8 of the object image, i.e. the image in Block 4, has been processed on each CUDA workstation, a total of 48 integers, comprising 16 sets of the defect quantity indexes and the defect size, are stored in Block G. The 48 integers on CUDA workstation #2 must be sent to CUDA workstation #1 for data integration. As shown in Fig. 3, such inter-platform operations are designed through a distributed computing model based on the Message Passing Interface (MPI) standard. The operations will be performed by two distributed MPI processes: MPI process #1 sends the integer array to MPI process #0, whereas MPI process #0 receives the integer array and combines it with the local one. Such inter-platform operations also need to be completed within the same 1.7 seconds allotted to the last 1/8 of the object image.

Based on the maximum CUDA threads allowable on the CUDA GPU adopted in this work and our previous work, the image size suitable for CUDA multithreading is chosen to be 5,000×12,288 pixels. Therefore, as shown in Figure 3, on each CUDA workstation, a total of 61,440,000 (=5,000×12,288) CUDA threads, i.e. CUDA threads with ID #0 to ID #61,439,999, are launched to perform defect detection operations concurrently. The defect detection operations are pixel-wise operations, including the binarization, defect labeling, and defect edge detection. For each 1/32 of the object image, such as the image data in Block 1-A, a corresponding Pthread, such as Pthread #1, will activate 61,440,000 CUDA threads to perform these pixel-wise operations one after another.

From the software aspect, DHIS contains a hybrid computing model: a concurrent computing model for the CPU and GPU operations running on a single computer, and a distributed computing model for the CPU operations running on multiple computers. Additionally, to implement the hybrid computing model, DHIS employs a hybrid programming model, which integrates Pthreads, MPI, and CUDA programming models. Therefore, DHIS has the heterogeneous nature in the software.

3 CUDA-BASED DEFECT DETECTION ALGORITHMS

IN this work, the purpose of performing surface defect detection on a mirror object is to obtain the defect quantity indexes, the defect size, and the image with defect edges. Moreover, in order to exploit the high performance feature of a CUDA GPU, the defect detection operations are developed to be pixel-wise operations, including the binarization, defect labeling process, and defect edge detection. The binarization aims to generate a binary image for the subsequent operations (defect labeling and defect edge detection), as well as to obtain the defect size, which is equivalent to the number of defect pixels. The defect labeling process aims to obtain the defect quantity indexes, which include the number of defect initial pixels and the number of defect terminal pixels. The defect edge detection aims to obtain an image with extracted defect edges. Each CUDA thread represents a pixel on a 5,000×12,288 pixel image (1/32 of the object image) and performs the above pixel-wise operations on the corresponding pixel.

3.1 Binarization

THE binarization algorithm is straightforward and based on a predefined threshold. In this work, a pixel with a value larger than the threshold is considered as a defect pixel. Therefore, as shown in Figure 5, if a pixel value is larger than the threshold, the pixel value is set to 255, and the first element of Block H is increased by 1. Otherwise, the pixel value is set to zero. Block H is a memory block allocated in the GPU's global memory to store three integers including the defect quantity indexes and the defect size. The first element of Block H is used to store the number of defect pixels, which represents the defect size. To avoid the race condition among CUDA threads,

increasing the first element of Block H is implemented as an atomic operation. On each CUDA workstation, every time when the binarization is completed by all the CUDA threads, the number of defect pixels on a 5,000×12,288 pixel image (1/32 of the object image) is acquired, and the resultant binary image is ready for the subsequent pixel-wise operations.

```
BEGIN
IF  pixel value > threshold  THEN
  pixel value ← 255
  Block_H[0] ← Block_H[0] + 1
ELSE
  pixel value ← 0
END
```

Figure 5. Pseudocode for the binarization.

3.2 Defect Labeling

THE defect labeling process aims to acquire two defect quantity indexes, including the number of defect initial pixels and the number of defect terminal pixels. Therefore, the defect labeling process contains two pixel-wise operations: the defect initial pixel labeling and the defect terminal pixel labeling.

The defect initial pixel labeling algorithm is illustrated in Figure 6. A pixel is considered as a defect initial pixel, if it satisfies the following requirements: its value is 255, and the values of the upper-surrounding pixels (left, upper-left, upper, and upper-right pixels) are all zero. If a pixel is determined to be a defect initial pixel, the second element of Block H is increased by 1. To avoid the race condition among CUDA threads, increasing the second element of Block H is implemented as an atomic operation. On each CUDA workstation, every time when the defect initial labeling is completed by all the CUDA threads, the number of defect initial pixels on a 5,000×12,288 pixel image (1/32 of the object image) is acquired.

```
BEGIN
IF  pixel value = 255  THEN
  IF  left pixel value = 0  AND
      upper-left pixel value = 0  AND
      upper pixel value = 0  AND
      upper-right pixel value = 0  THEN
    Block_H[1] ← Block_H[1] + 1
END
```

Figure 6. Pseudocode for the defect initial pixel labeling.

The defect terminal pixel labeling algorithm is illustrated in Figure 7. A pixel is considered as a defect terminal pixel, if it satisfies the following requirements: its value is 255, and the values of the lower-surrounding pixels (right, lower-right, lower, and lower-left pixels) are all zero. If a pixel is determined to be a defect terminal pixel, the third element of Block H is increased by 1. To avoid the race condition among CUDA threads, increasing the third element of Block H is also implemented as an atomic operation. On each CUDA workstation, every time when the defect terminal labeling is completed by all the CUDA threads, the number of defect terminal pixels on a 5,000×12,288 pixel image (1/32 of the object image) is acquired.

```
BEGIN
IF  pixel value = 255  THEN
  IF  right pixel value = 0  AND
      lower-right pixel value = 0  AND
      lower pixel value = 0  AND
      Lower-left pixel value = 0  THEN
    Block_H[2] ← Block_H[2] + 1
END
```

Figure 7. Pseudocode for the defect terminal pixel labeling.

3.3 Defect Edge Detection

THE defect edge detection aims to generate an image with defect edges that are composed of defect edge pixels.

```
BEGIN
IF  pixel value = 255  THEN
  IF  left pixel value = 0  AND
      right pixel value = 255  THEN
    Block_F[thread_id] ← 255
  ELSE IF  upper-left pixel value = 0  AND
           lower-right pixel value = 255  THEN
    Block_F[thread_id] ← 255
  ELSE IF  upper pixel value = 0  AND
           lower pixel value = 255  THEN
    Block_F[thread_id] ← 255
  ELSE IF  upper-right pixel value = 0  AND
           lower-left pixel value = 255  THEN
    Block_F[thread_id] ← 255
  ELSE IF  right pixel value = 0  AND
           left pixel value = 255  THEN
    Block_F[thread_id] ← 255
  ELSE IF  lower-right pixel value = 0  AND
           upper-left pixel value = 255  THEN
    Block_F[thread_id] ← 255
  ELSE IF  lower pixel value = 0  AND
           upper pixel value = 255  THEN
    Block_F[thread_id] ← 255
  ELSE IF  lower-left pixel value = 0  AND
           upper-right pixel value = 255  THEN
    Block_F[thread_id] ← 255
END
```

Figure 8. Pseudocode for the defect edge detection.

As shown in Figure 8, if a pixel complies with one of the following eight conditions, it is considered as the corresponding defect edge pixel:

(1) defect left edge pixel: its left and right pixel values are zero and 255, respectively;

(2) defect upper-left edge pixel: its upper-left and lower-right pixel values are zero and 255, respectively;

(3) defect upper edge pixel: its upper and lower pixel values are zero and 255, respectively;

(4) defect upper-right edge pixel: its upper-right and lower-left pixel values are zero and 255, respectively;

(5) defect right edge pixel: its right and left pixel values are zero and 255, respectively;

(6) defect lower-right edge pixel: its lower-right and upper-left pixel values are zero and 255, respectively;

(7) defect lower edge pixel: its lower and upper pixel values are zero and 255, respectively;

(8) defect lower-left edge pixel: its lower-left and upper-right pixel values are zero and 255, respectively.

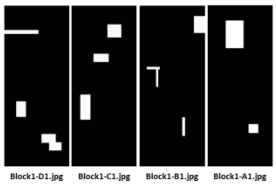
Block F is a memory block allocated in the GPU's global memory to store the image with identified defect edges. Moreover, Block F is initialized as an array of zeros. As shown in Figure 8, if a pixel is determined to be a defect edge pixel, a value of 255 is assigned to the corresponding element of Block F, i.e. Block_F[thread_id], where thread_id is the ID of the CUDA thread in charge of the operations on that pixel. On each CUDA workstation, every time when the defect edge detection is completed by all the CUDA threads, a 5,000×12,288 pixel image (1/32 of the object image) with defect edges is acquired.

4 EXPERIMENTAL RESULTS

IN this work, the size of a test object is 280×86 mm^2 and the resolution requirement is 3.5×3.5 μm^2/pixel. As a result, the image of a test object has 80,000×24,576 pixels. Each CUDA workstation of DHIS is configured to handle one-half of a test object, corresponding to an image of 80,000×12,288 pixels. Due to the maximum CUDA threads allowable on the CUDA GPU adopted in this work, the above image of 80,000×12,288 pixels needs to be separated into 16 sub-images (each with 5,000×12,288 pixels), which will be processed sequentially on each CUDA workstation.

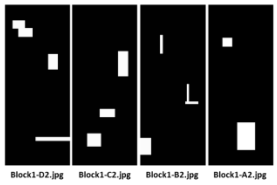
The objective of this work is to validate that the proposed DHIS can acquire the correct defect quantity indexes and defect sizes for all the 32 sub-images (each with 5,000×12,288 pixels) within the time limit of 6.8 seconds. Thus, for system validation purposes, a total of 32 grayscale JPEG images, each with 5,000×12,288 pixels and simulated defect patterns, are created to represent the entire image of an object. The downscaled versions, defect quantity indexes, and defect sizes for the 32 grayscale images are shown in Figure 9 to Figure 12.

CUDA Workstation #1



	No. of defect initial pixels	No. of defect terminal pixels	No. of defect pixels
Block1-A1.jpg	2	2	34,22,394
Block1-B1.jpg	3	4	18,36,505
Block1-C1.jpg	3	3	33,41,963
Block1-D1.jpg	3	3	30,36,988

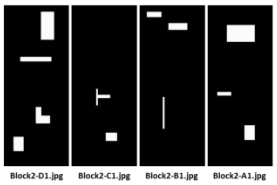
CUDA Workstation #2



	No. of defect initial pixels	No. of defect terminal pixels	No. of defect pixels
Block1-A2.jpg	2	2	34,22,394
Block1-B2.jpg	4	3	18,36,505
Block1-C2.jpg	3	3	33,41,963
Block1-D2.jpg	3	3	30,36,988

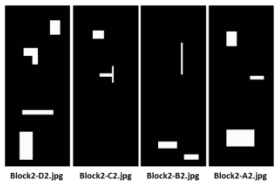
Figure 9. Defect information and downscaled versions for Block 1 images on each CUDA workstation.

CUDA Workstation #1



	No. of defect initial pixels	No. of defect terminal pixels	No. of defect pixels
Block2-A1.jpg	3	3	3,981,059
Block2-B1.jpg	3	3	1,546,739
Block2-C1.jpg	2	3	954,370
Block2-D1.jpg	4	4	4,836,652

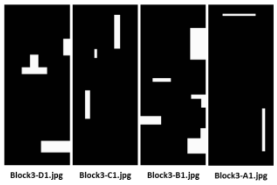
CUDA Workstation #2



	No. of defect initial pixels	No. of defect terminal pixels	No. of defect pixels
Block2-A2.jpg	3	3	3,981,059
Block2-B2.jpg	3	3	1,546,739
Block2-C2.jpg	3	2	954,370
Block2-D2.jpg	4	4	4,836,652

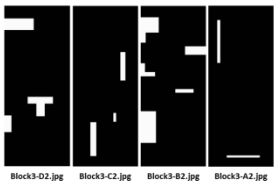
Figure 10. Defect information and downscaled versions for Block 2 images on each CUDA workstation.

CUDA Workstation #1



	No. of defect initial pixels	No. of defect terminal pixels	No. of defect pixels
Block3-A1.jpg	2	2	1,171,066
Block3-B1.jpg	6	5	6,862,872
Block3-C1.jpg	3	3	2,165,475
Block3-D1.jpg	4	3	4,242,111

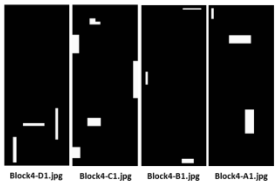
CUDA Workstation #2



	No. of defect initial pixels	No. of defect terminal pixels	No. of defect pixels
Block3-A2.jpg	2	2	1,171,066
Block3-B2.jpg	5	6	6,862,872
Block3-C2.jpg	3	3	2,165,475
Block3-D2.jpg	3	4	4,242,111

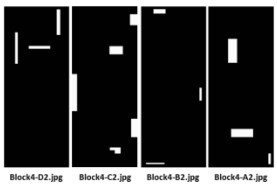
Figure 11. Defect information and downscaled versions for Block 3 images on each CUDA workstation.

CUDA Workstation #1



	No. of defect initial pixels	No. of defect terminal pixels	No. of defect pixels
Block4-A1.jpg	3	3	2,464,707
Block4-B1.jpg	3	3	637,043
Block4-C1.jpg	5	5	3,178,557
Block4-D1.jpg	3	3	1,394,547

CUDA Workstation #2



	No. of defect initial pixels	No. of defect terminal pixels	No. of defect pixels
Block4-A2.jpg	3	3	2,464,707
Block4-B2.jpg	3	3	637,043
Block4-C2.jpg	5	5	3,178,557
Block4-D2.jpg	3	3	1,394,547

Figure 12. Defect information and downscaled versions for Block 4 images on each CUDA workstation.

Each image for CUDA workstation #1 is rotated by 180° to create its counterpart for CUDA workstation #2. Therefore, such paired images have the same number of defect pixels. Due to an angle of 180° between each pair of images, the numbers of defect initial and defect terminal pixels for an image on CUDA workstation #1 are the numbers of defect terminal and defect initial pixels, respectively, for the corresponding image on CUDA workstation #2. For

example, as shown in Figure 13, Block2-C1.jpg on CUDA workstation #1 has two defect initial pixels and three defect initial pixels, whereas Block2-C2.jpg on CUDA workstation #2 has three defect initial pixels and two defect terminal pixels.

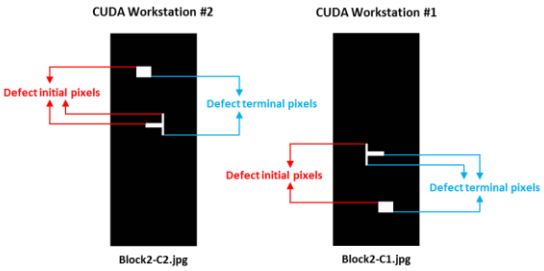


Figure 13. Defect initial and defect terminal pixels for Block 2-C images on each CUDA workstation.

In this work, the above 32 images are used to conduct the defect detection experiment on the following three hardware and software configurations: (1) Configuration #1: distributed system with two CUDA workstations; CPU distributed computing with MPI processes, CPU concurrent computing with Pthreads, and GPU concurrent computing with CUDA threads; (2) Configuration #2: distributed system with two CUDA workstations; CPU distributed computing with MPI processes and GPU concurrent computing with CUDA threads; (3) Configuration #3: standalone system with one CUDA workstation; GPU concurrent computing with CUDA threads.

Configuration #1 is indeed the proposed DHIS, Configuration #2 is DHIS without CPU concurrent computing, and Configuration #3 is the system presented in our previous work. Each of the two CUDA workstations adopted in this work has an NVIDIA Tesla C2075 GPU, an Intel Xeon E5-2620 CPU, and 32 gigabytes of RAM. Meanwhile, a 64-bit Windows 7 operating system runs on each CUDA workstation.

Under Configuration #1, a total of 32 Pthreads will simultaneously load the 32 images on the two CUDA workstations. In addition, on each CUDA workstation, when Pthread #i starts to process its image through CUDA threads, the preceding Pthread, Pthread #(i-1), will begin to save its processed image as a file. Moreover, when the last Pthread, i.e. Pthread #16, begins to save its processed image as a file on each CUDA workstation, MPI process #1 on CUDA workstation #2 and MPI process #0 on CUDA workstation #1 will start to send and receive the defect data, respectively. The defect data include the defect quantity indexes and the defect sizes for the 16 images on CUDA workstation #2.

Under Configuration #2, MPI process #0 and MPI process #1 will simultaneously start the defect detection operations on CUDA workstation #1 and CUDA workstation #2, respectively. Moreover, on each CUDA workstation, a total of 16 images will be processed one after another through CUDA threads. Due to such a sequential processing mode on each CUDA workstation, an image will not be loaded for processing until the preceding processed image has been saved as a file. After the last processed image has been saved as a file, MPI process #1 on CUDA workstation #2 and MPI process #0 on CUDA workstation #1 will start to send and receive the defect data, respectively. The defect data include the defect quantity indexes and the defect sizes for the 16 images on CUDA workstation #2.

Under Configuration #3, all the 32 images will be processed one after another through CUDA threads on a single CUDA workstation. Thus, an image will not be loaded for processing until the preceding processed image has been saved as a file. Once the last processed image has been saved as a file, the defect quantity indexes and the defect sizes for all the 32 images are acquired.

Moreover, for each configuration, in addition to the defect quantity indexes and the defect sizes for all the 32 images, the defect detection experiment also aims to measure the turnaround time between two moments on CUDA workstation #1: (1) the moment when the first image, Block1-A1.jpg, starts to be loaded for processing; and (2) the moment when the defect quantity indexes and the defect sizes for all the 32 images have been acquired. The defect detection experiment is conducted 30 times under each configuration to obtain the results.

The experimental results show that, under each configuration, the defect quantity indexes and the defect sizes obtained for all the 32 images at each run of the experiment are the same as those shown in Figure 9 to Figure 12. However, the performances in terms of the turnaround times under the three configurations are significantly different.

Figure 14 shows the turnaround times of 30 experimental runs under each configuration, and Table 2 lists the statistics of the turnaround time results. As shown in Table 2, the average turnaround time under Configuration #1 is 2.56 seconds, which is 37.65% of the time limit, 6.8 seconds. By contrast, the average turnaround times under Configurations #2 and #3 are 39.09 and 79.59 seconds, respectively, which do not fall within the time limit. On the other hand, the average processing speed under Configuration #1 is 15.27 (= 39.09/2.56) and 31.09 (=79.59/2.56) times of those under Configurations #2 and #3, respectively. Also, the numbers of independent instruction sequences used to process the 32 images under Configurations #1, #2, and #3 are 32, 2, and 1, respectively. Thus, the number of independent instruction sequences under Configuration #1 is 16 (= 32/2) and 32 (=32/1) times of those under Configurations #2 and #3, respectively. Hence, the experimental results also indicate a good linear relationship between the processing speed ratio and the instruction sequence quantity ratio.

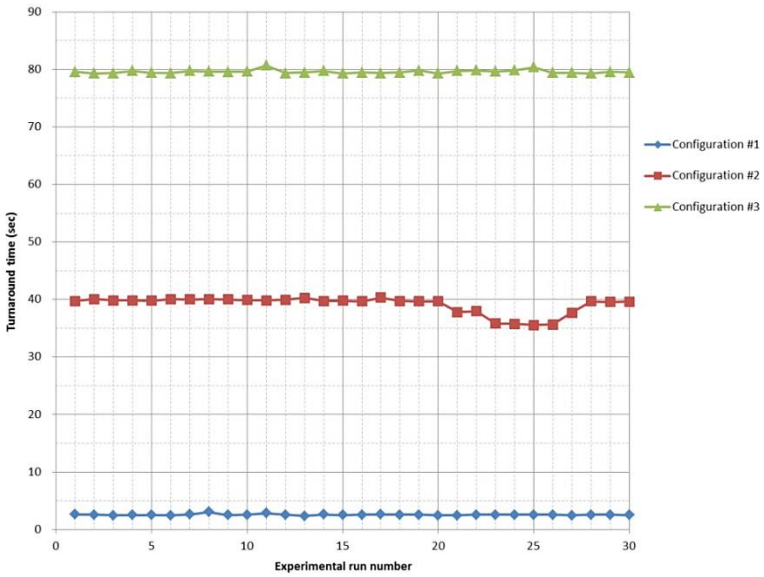


Figure 14. **Turnaround times of 30 experimental runs under each configuration.**

Table 2. **Statistics of the turnaround time results**

Configuration / Turnaround time (sec)	Configuration #1: • Distributed system • CPU distributed computing • CPU concurrent computing • GPU concurrent computing	Configuration #2: • Distributed system • CPU distributed computing • GPU concurrent computing	Configuration #3: • Standalone system • GPU concurrent computing
Average	2.56	39.09	79.59
Standard deviation	0.12	1.51	0.31
Maximum	3.06	40.33	80.70
Minimum	2.34	35.54	79.29

5 CONCLUSION

AN in-line surface defect detection task, with high resolution and high speed requirements, essentially demands that the underlying processing infrastructure handle a large volume of image data within a short image acquisition cycle. To this end, the Distributed Heterogeneous Inspection System (DHIS) was proposed in this study. DHIS has a distributed multi-CPU and multi-GPU hardware architecture and is equipped with functions for CPU distributed computing, CPU concurrent computing, and GPU concurrent computing. The resolution and speed requirements in this study were 3.5×3.5 μm^2/pixel and 289.13 mega pixels/sec, respectively. A total of 32 grayscale JPEG images, each with 5,000×12,288 pixels and simulated defect patterns, were created to conduct the performance experiment under three different system configurations, including: (1) DHIS; (2) DHIS without CPU concurrent computing function; (3) a non-distributed, standalone system with only GPU concurrent computing function. The experimental results showed that: (1) the defect quantity indexes and the defect sizes obtained under the above three system configurations were all correct; (2) only DHIS completed the required tasks within the time limit of 6.8 seconds; (3) the average turnaround time of DHIS was 2.56 seconds, which is 37.65% of the time limit; (4) a good linear relationship was found to exist between the processing speed ratio and the instruction sequence quantity ratio. Therefore, the proposed DHIS can satisfy the high resolution and high speed requirements specified in this study. Furthermore, due to its distributed and expandable hardware structure and software algorithm, DHIS has the potential to handle in-line surface defect detection tasks with even more challenging resolution and speed requirements.

6 ACKNOWLEDGMENT

This research is supported by the Ministry of Science and Technology in Taiwan under grants MOST 106-2221-E-110-042, MOST 105-2221-E-110-060, MOST 104-2221-E-110-062, MOST 103-2221-E-110-087, and NSC 102-2218-E-033-002-MY2.

7 REFERENCES

L. Busin, N. Vandenbroucke, and L. Macaire, (2013). Contribution of a color space selection to a flaw detection vision system. *Journal of Electronic Imaging*, *22*(3), 17.

D. R. Butenhof, (1997). *Programming with POSIX Threads* (1 ed.): Addison-Wesley Professional.

D. Buttlar, J. Farrell, and B. Nichols, (1996). *PThreads Programming: A POSIX Standard for Better Multiprocessing* (1 ed.): O'Reilly Media.

M. Chang, Y.-C. Chou, P. T. Lin, and J. L. Gabayno, (2014). Fast and High-Resolution Optical Inspection System for In-Line Detection and Labeling of Surface Defects. *CMC: Computers, Materials & Continua*, *42*(2), 125-140.

Y. C. Chiou, J. Z. Liu, and Y. T. Liang, (2011). Micro crack detection of multi-crystalline silicon solar wafer using machine vision techniques. *Sensor Review*, *31*(2), 154-165.

S. Cook, (2012). *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs* (1 ed.): Morgan Kaufmann.

W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, (2014). *Using Advanced MPI: Modern Features of the Message-Passing Interface* (1 ed.). Cambridge, MA, USA: MIT Press.

W. Gropp, E. Lusk, N. Doss, and A. Skjellum, (1996). A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, *22*(6), 789-828.

W. Gropp, E. Lusk, and A. Skjellum, (2014). *Using MPI: Portable Parallel Programming with the Message-Passing Interface* (3 ed.). Cambridge, MA, USA: MIT Press.

W. B. Li, C. H. Lu, and J. C. Zhang, (2012). A local annular contrast based real-time inspection algorithm for steel bar surface defects. *Applied Surface Science*, *258*(16), 6080-6086.

W. C. Li and D. M. Tsai, (2011). Automatic saw-mark detection in multicrystalline solar wafer images. *Solar Energy Materials and Solar Cells*, *95*(8), 2206-2220.

C.-S. Lin, C.-W. Lin, S.-W. Yang, S.-K. Lin, and C.-C. Chiu, (2013). The Chemical Stain Inspection of Polysilicon Solar Cell Wafer by the Fuzzy Theory Method. *Intelligent Automation & Soft Computing*, *19*(3), 391-406.

E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, (2008). NVIDIA Tesla: A unified graphics and computing architecture. *IEEE micro*, *28*(2), 39-55.

Message Passing Interface Forum. (2015). *MPI: A Message-Passing Interface Standard, Version 3.1*. Knoxville,Tennessee, USA.

W. Michaeli and K. Berdel, (2011). Inline inspection of textured plastics surfaces. *Optical Engineering*, *50*(2), 6.

NVIDIA. (2014). *CUDA C Programming Guide*. Retrieved from http://docs.nvidia.com/cuda/cuda-c-programming-guide/

J. Park and Z. Bien, (1995). Design of an Advanced Machine Vision System for Industrial Inspection. *Intelligent Automation & Soft Computing*, *1*(2), 209-219.

G. Rosati, G. Boschetti, A. Biondi, and A. Rossi, (2009). Real-time defect detection on highly reflective curved surfaces. *Optics and Lasers in Engineering*, *47*(3-4), 379-384.

S. G. Ryu, D. C. Choi, Y. J. Jeon, S. J. Lee, J. P. Yun, and S. W. Kim, (2014). Detection of Scarfing Faults on the Edges of Slabs. *ISJI International*, *54*(1), 112-118.

Y. Tian, C. Zhao, S. Lu, and X. Guo, (2011). Multiple Classifier Combination for Recognition of Wheat Leaf Diseases. *Intelligent Automation & Soft Computing*, *17*(5), 519-529.

D. M. Tsai, C. C. Chang, and S. M. Chao, (2010). Micro-crack inspection in heterogeneously textured solar wafers using anisotropic diffusion. *Image and Vision Computing*, *28*(3), 491-501.

D. M. Tsai, M. C. Chen, W. C. Li, and W. Y. Chiu, (2012). A fast regularity measure for surface defect detection. *Machine Vision and Applications*, *23*(5), 869-886.

D. M. Tsai, I. Y. Chiang, and Y. H. Tsai, (2012). A Shift-Tolerant Dissimilarity Measure for Surface Defect Detection. *Ieee Transactions on Industrial Informatics*, *8*(1), 128-137.

D. M. Tsai, and H. Y. Tsai, (2011). Low-contrast surface inspection of mura defects in liquid crystal displays using optical flow-based motion analysis. *Machine Vision and Applications*, *22*(4), 629-649.

Y. H. Tsai, D. M. Tsai, W. C. Li, W. Y. Chiu, and M. C. Lin, (2011). Surface defect detection of 3D objects using robot vision. *Industrial Robot-an International Journal*, *38*(4), 381-398.

S. E. Umbaugh, (2016). *Digital image processing and analysis: human and computer vision applications with CVIPtools*: CRC press.

X. W. Zhang, Y. Q. Ding, Y. Y. Lv, A. Y. Shi, and R. Y. Liang, (2011). A vision inspection system for the surface defects of strongly reflected metal based on multi-class SVM. *Expert Systems with Applications*, *38*(5), 5930-5939.

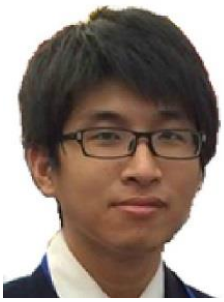
8 DISCLOSURE STATEMENT

NO potential conflict of interest was reported by the authors.

9 NOTES ON CONTRIBUTORS

Yu-Cheng Chou received his Ph.D. Degree in Mechanical and Aeronautical Engineering from University of California, Davis. He is an Assistant Professor in the Institute of Undersea Technology at National Sun Yat-sen University, Taiwan. His research focuses on intelligent control and design optimization for underwater vehicles and underwater instrumentation systems.

Wei-Chieh Liao received his Master of Science Degree in Mechanical Engineering from Chung Yuan Christian University, Taiwan. His master thesis relates to real-time surface defect detection through a single CUDA (Compute Unified Device Architecture) enabled GPU platform.



Yan-Liang Chen received his Master of Science Degree in Mechanical Engineering from Chung Yuan Christian University, Taiwan. His master thesis relates to high performance surface defect detection through multiple CUDA (Compute Unified Device Architecture) enabled GPU platforms.

Ming Chang received his Ph.D. Degree in Mechanical Engineering from National Taiwan University. He is the chairman of the Chinese Metrology Society and a Professor in the Department of Mechanical Engineering at Chung Yuan Christian University, Taiwan. His research interests include precision metrology, photomechanics, automated optical inspection, and nanotechnology.

Po Ting Lin received his Ph.D. Degree in Mechanical and Aerospace Engineering from Rutgers University. He is an Associate Professor in the Department of Mechanical Engineering at National Taiwan University of Science and Technology. His research interests include stochastic parametric modeling and multidisciplinary design optimization of complex engineering systems.