Electronic Preprint for Journal of Information Processing Vol.22 No.1

Regular Paper

GPGPU-Assisted Subpixel Tracking Method for Fiducial Markers

Naoki Shibata[1,a)] Shinya Yamamoto[2,b)]

Received: March 28, 2013, Accepted: October 9, 2013

Abstract: With an aim to realizing highly accurate position estimation, we propose in this paper a method for efficiently and accurately detecting the 3D positions and poses of traditional fiducial markers with black frames in high-resolution images taken by ordinary web cameras. Our tracking method can be efficiently executed utilizing GPGPU computation, and in order to realize this, we devised a connected-component labeling method suitable for GPGPU execution. In order to improve accuracy, we devised a method for detecting 2D positions of the corners of markers in subpixel accuracy. We implemented our method in Java and OpenCL, and we confirmed that the proposed method provides better detection and measurement accuracy, and recognizing from high-resolution images is beneficial for improving accuracy. We also confirmed that our method is more than two times as fast as the existing method with CPU computation.

Keywords: GPGPU, fiducial marker, connected-component labeling

1. Introduction

HD resolution cameras are becoming inexpensive and many mobile devices have built-in HD cameras with very good performance. We can expect to realize an accurate and inexpensive position estimation method utilizing these inexpensive cameras. Detecting and decoding artificial fiducial markers [5], [11], [12] from a video is robust compared to detecting other features and works well under varying lighting conditions. Although tracking methods for artificial fiducial markers are old, they are still widely used in much research [4], [14], [15]. The accuracy of position and pose estimation with fiducial markers depends on the number of pixels of the marker in a video frame, where the accuracy can be defined by the amount of differences between the estimated and real positions for each corner of tracked markers. This accuracy can be improved by recognizing markers from high resolution videos. Recently, SoCs with Graphics Processing Units (GPUs) are becoming inexpensive, and General-Purpose computing on GPU (GPGPU) is especially suited for applications like image processing. By offloading CPU tasks to the GPU, the CPU can handle other tasks, and this can improve the user experience while increasing the overall power efficiency. However, existing good algorithms for a CPU are not always suited for GPU computation, and rewriting current programs for single-threaded CPU computation to highly parallelized GPGPU programs is not an easy task.

In this paper, aiming at realizing highly accurate position estimation, we propose a method for efficiently and accurately detecting the 3D positions and poses of traditional fiducial markers with black frames in high resolution images taken by cameras on mobile devices. The accuracy is improved by detecting the 2D positions of the corners of markers in subpixel accuracy, and we propose a new method for this. In our method, we utilize GPGPU computation to improve tracking speed, and for this purpose, we devised a new method for efficiently performing connected-component labeling on GPGPU. The tracking method proposed in this paper can be utilized for tracking many kinds of commonly seen square-shaped fiducial markers with black frames [5], [12].

In the proposed method, we first use GPGPU to apply an edge detection with the Sobel operator to the input image; then apply thresholding to get a binary image. We then apply connected-component labeling (CCL) to this binary image using GPGPU; and then we detect the coordinates of the corners of the markers in one pixel. We then, analyze the four outer edges of the black frame of the marker image to get the subpixel coordinates of the corners of the markers. Finally, we find the 3D coordinates of the markers by Newton's method. Our main contributions in this paper are the fast CCL algorithm for GPGPU and the method to obtain subpixel coordinates of the corners of the markers.

We implemented our method in Java and OpenCL, and evaluated the accuracy and performance of the method. We compared measurement accuracy by our method with that by ARToolKit, and our experimental results show that our method has significantly better accuracy in both distance and angle measurements. We also compared the performance of our method with ARToolKit, and our results show that our method is more than two times as fast as ARToolKit.

The remainder of this paper is organized as follows. Section 2 introduces some related works. Section 3 presents the proposed tracking methods. Section 4 explains the proposed connected-component labeling algorithm suitable for GPGPU computation. Section 5 explains the method for detecting 2D positions of the

[1] Nara Institute of Science and Technology, Ikoma, Nara 630–0192, Japan
[2] Tokyo University of Science, Yamaguchi 756–0884, Japan
[a)] n-sibata@is.naist.jp
[b)] shiny-ya@ed.yama.tus.ac.jp

corners of markers with subpixel accuracy. Section 6 show results from evaluating the accuracy and speed of the proposed method. Section 7 presents our conclusions.

2. Related Works

In this section, we introduce some related work on fiducial marker tracking, applications of fiducial markers, planar tracking, connected component labeling and subpixel object detection.

2.1 Fiducial Marker Tracking

Several tracking methods for fiducial markers have been proposed [5], [11], [12]. ARToolKit [12] would be the best-known fiducial marker tracking library, and it can be characterized in that the user can freely design the marker by specifying the design by a 16×16 pixel image. The tracking algorithm in ARToolKit first detects the black frames of the markers from the input image and calculates the transform matrices from their coordinates. Then, the transformed image is matched with the registered payload designs. ARTag [5] is an improvement over ARToolKit in terms of false recognition rate, robustness against nonuniform lighting, jitter and recognition speed. The design of the marker includes 36 bits of cells including error-correcting codes. reacTIVision [11] is developed as a component of reacTable. They use amoeba-shaped fiducial markers generated by a genetic algorithm. The markers are managed in a tree data structure. They tolerate some noise by traversing this tree structure during recognition.

2.2 Applications of Fiducial Markers

Ferretti et al. proposed a user interface for video stream management utilizing ARToolKit in Ref. [4]. With this interface, each configuration of a video stream can be manipulated using a box with a fiducial marker. By manipulating these boxes like knobs, the user can layout or resize each video clip. In Ref. [14], fiducial markers based on ARToolKit are attached to items like container boxes and pictures to realize a system to aid managing and looking up these items. Mulloni et al. developed a conference guide system called Signpost based on a Studierstube ES (Embedded Systems) framework and describe the results of their controlled user study to compare a map-based guidance system without localization function to a guidance system with a GPS-like real-time localization function [15].

2.3 Planar Tracking

Several planar tracking methods are proposed that enable tracking textured planar objects with any 2D image [3], [9]. In Ref. [9], by modeling how the shot images degrade when the planar surface is shot from steep angles, a method is presented that enables stable tracking even when the planar surface is shot from a steep angle. In Ref. [3], a method utilizing the Maximally Stable Extremal Region detector is proposed to accurately track weakly textured planar objects. These methods for planar tracking require a higher CPU load compared to tracking simple artificial markers. In Ref. [3], the authors report that their planar tracking method has a 20 fps tracking speed when 640×480 pixel images are input. In our method, we aim at a higher tracking speed for higher-resolution images.

2.4 Connected Component Labeling

Connected Component Labeling (CCL) is the well known technique for assigning a unique label to each of connected components in a given image. It has many applications in image processing and various methods have been proposed [1], [7], [10], [13], [17]. Suzuki et al. proposed a method based on sequential local operations in the course of a forward raster scan followed by a backward raster scan [13]. In their paper, they experimentally show that the execution time of their method is proportional to the number of the input pixels. Hawick et al. describe a method for GPGPU computation based on Suzuki's method [7]. However, since the original Suzuki method is not intended for GPGPU computation, this method could require a long execution time on GPGPU under some conditions. In this paper, we propose a CCL algorithm suitable for GPGPU computation, where the main CCL kernel does not have a loop, and the labels propagate exponentially as the kernel is applied to the pixels.

2.5 Subpixel Object Detection

There are several methods for detecting object positions in an image with subpixel accuracy. Chen et al. proposed a method to detect X-corner positions on a checkerboard with subpixel accuracy for calibration [2]. Grompone et al. proposed a subpixel line segment detector based on the Hough Transform [6]. However, since the Hough Transform requires a large amount of computations, this method is not suitable for our purpose. In this paper, we propose a method where the approximate positions of the corners of a marker are detected first, and then the edges of the marker are analyzed to detect the accurate corner positions based on the linear regression.

3. Proposed Marker Tracking Method

In this section, we first give a brief explanation of some properties required for programs to be efficiently executed on GPU. Then, we briefly describe the specifications of the fiducial marker used in this paper. After that, we describe the proposed tracking method utilizing GPGPU computation. We will explain the proposed CCL algorithm in Section 4, and the method for detecting the 2D positions of the corners of markers with subpixel accuracy in Section 5.

3.1 Properties Required for GPGPU Programs

GPUs have been primarily used for rendering 3D images. Recently, programmable shaders have been added to the rendering pipelines to enhance expressiveness. With the programmable shaders, users can specify rendering methods with shader programs. GPGPU is the way to use GPUs to perform computation utilizing the capability of shaders to execute programs [16]. A GPU has multiple thread execution blocks, and each block is capable of executing many threads in parallel. All threads in a block execute the same sequence of instructions simultaneously with different input data. If the instructions contain conditional branches, both of the branch paths are executed in turn and the unused results are discarded. Thus, the results of conditional branches in the threads in a block must be almost identical in order to make the execution efficient. If the number of itera-

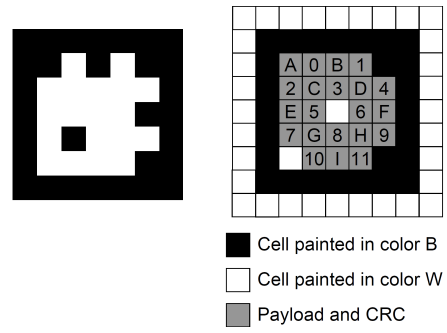
tions differs in each thread, the execution of the whole block does not finish until the last thread finishes. Thus, in order to make a GPGPU program efficient, we need to eliminate conditional branches whose results depend on the input data. Redesigning existing applications for a CPU is not a trivial matter. GPGPU programs are coded in languages like OpenCL or CUDA, and almost all discrete graphics cards and a large portion of integrated graphics processors for PCs being sold today have capability for GPGPU execution. Although a large portion of smartphones have a capability for GPGPU computation in the hardware level, GPGPU computation is not possible on smartphones due to software limitations at this time. We expect this situation to be improved within a few years.

3.2 Marker Specifications

Figure 1 (a) and (b) show an example of the marker and its structure, respectively. Markers are composed of square-shaped cells painted in two distinct colors B and W. We assigned 9 bits for CRC out of the available 21 bits so that it detects all errors up to 2 bits and thus false positive detection errors are practically eliminated.

3.3 Overview of Proposed Tracking Method

The proposed tracking method outputs all 6DOF positions and payloads of the markers in the given image captured by a camera. The method does not refer to previous images or results, and it processes each image independently. The method consists of the GPU processing part and the CPU processing part. The GPU mainly processes the image, and the CPU mainly performs transformation of coordinates.



(a) Example of Marker (b) Structure of Marker

Fig. 1 Marker used in this paper.

3.4 GPU Processing Part

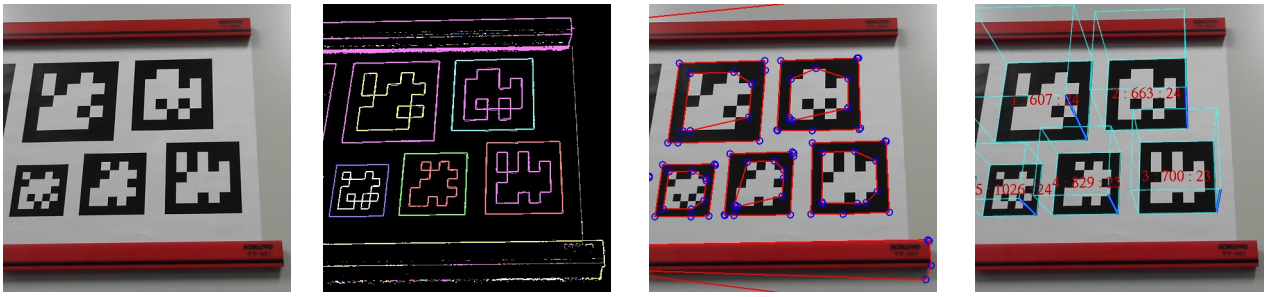
In the GPU processing part, edge detection is performed on the input image, and then candidates for the sets of pixels that represent the outer edge of markers are extracted. Then, 8 representative points for each set of the pixels are extracted. Here, 8 representative points are the pixels on the top, top right, right, bottom right, bottom, bottom left, left, and top left of the pixels. The GPU processing part consists of the following 3 steps.

Step G1 The Sobel operator and edge thinning are applied to the original image (**Fig. 2** (a)). Then, thresholding and CCL are applied (Fig. 2 (b)). The details of the CCL algorithm is described in Section 4. In Fig. 2 (b), each color represents the last few bits of the labels assigned by CCL.

If thresholding is applied first to the input image, an inappropriate way of choosing the threshold may increase the error ratio. We apply thresholding to the detected edge of the image in order to avoid this problem.

Step G2 The components with less than 30 pixels of area size are discarded in this step. This step is composed of two phases. From the results of Step G1, a unique ID is attached to each component. We use this ID as an address in a buffer. We first prepare a buffer filled by zeroes and perform the atomic_inc operation on the value specified by the address in the buffer for each pixel. The atomic operations are operations where reading a value from memory, doing some simple calculation with the value, and writing the value back to the memory are performed without interference from other threads. The atomic_inc operation takes a pointer, and increments the pointed value. The area size for each component is assigned to the value in the buffer. In the second phase, we discard theose components having less than 30 pixels of area size.

Step G3 8 representative points are calculated for each component in this step. The unique ID attached to each component is used as the address in a buffer to find the minimum coordinates of the x axis, for each component. We first prepare a buffer filled by zeroes and we perform the atomic_min operation on the value pointed by the address with the x coordinates for each pixel. The atomic_min operation takes a pointer and a value and reads the value from the pointer, compares it with the given value, and stores the smaller value to the pointer. As a result, we obtain the minimum x coordinates for each component. In order to get the coordinates at the upper-left corner, for example, we find the point that has the minimum $(x + y)$, where x and y are the coordinates of each pixel. We obtain the minimum and maximum coordinates



(a) Original image (b) After edge detection and CCL (c) After extracting 8 representative points (d) After pose estimation

Fig. 2 Images after each step.

for 4 directions, i.e., up, upper-left, left, and lower-left.

3.5 CPU Processing Part

The 8 points of a marker image derived by the GPU part redundantly contain the 4 corners of the marker. If the marker is warped, the 8 points may contain other pixels. In the CPU part, we first choose the correct 4 corners from the 8 points. Then, we find the coordinates of the four corners in subpixel accuracy, followed by calculating three-dimensional coordinates for each corner. The CPU processing part consists of the following 5 steps.

Step C1 The coordinates of the four outer corners of the black frame shown in Fig. 1 (b) are extracted. The 8 representative points output from Step G3 should contain these points. Some of the 8 points may be redundant or irrelevant. We first choose the most distant combination of 2 points from the 8 points, and we assume that these 2 points are 2 corners of the marker. We then find the two vectors orthogonal to the line segment between these two points. The most distant points from the midpoint of the two points in the direction of the each of two vectors are the remaining two corners.

Step C2 We discard components that are unlikely to be markers in this step. First, we discard components that are too flat. Then we calculate the intersection of the two diagonals of each component. This intersection is the center point of the marker, and by utilizing the property that the center cell is painted in color W, we discard components that do not meet this condition.

Step C3 The coordinates of the four corners derived in Step C1 have a precision of only 1 pixel. In this step, we find the subpixel coordinates of the four corners. The details are described in Section 5.

Step C4 3D coordinates are calculated for each corner of the markers. As shown in **Fig. 3**, the 3D positions of the camera, the image, and the marker are determined from the field of view and the aspect ratio of the camera. We then find the three dimensional positions of the four corners with Newton's method using the property that the 3D distances between adjacent corners are 1, and the angles of the corners are 90 degrees. In order to derive the camera parameters, we used the calibration method described in Ref. [8] combined with the proposed method described in Section 5.

Step C5 From the results of Step C4, we can now calculate the 2D coordinates of each cell in each marker. The payload and CRC are read from the image and decoded.

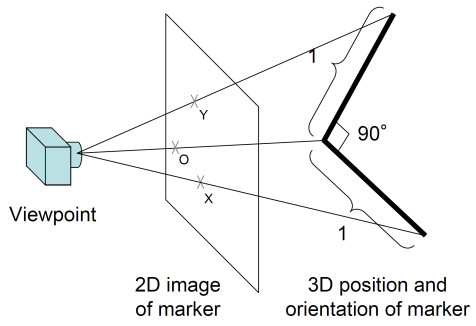


Fig. 3 Find 3D coordinates from 2D coordinates.

4. Connected Components Labeling Algorithm for GPGPU Computation

Connected Component Labeling (CCL) is the well known technique for assigning a unique label to each of connected components in a given image. It has many applications in image processing and various methods have been proposed. In Ref. [7], a CCL algorithm for GPGPU is proposed. However, in some cases, it takes a long execution time on GPGPU. More specifically, the number of iterations of the repeat~until loop in function *Mesh_Kernel_D_Analysis_Phase* in Algorithm 8 in Ref. [7] is O (width of input image).

In this paper, we propose a CCL algorithm suitable for GPGPU computation. In the proposed method, the kernel is applied to each of the pixels in the frame buffer several times. The kernel does not have a loop, and thus it finishes execution within a constant number of steps. We will experimentally show that the label propagates exponentially as the kernel is applied to the frame buffer. The kernel is designed so that it can be applied to the pixels in parallel.

The proposed CCL algorithm takes a pointer fb to the frame buffer as the input. We assume that each element (pixel) of the frame buffer can store one 32-bit integer value. Each pixel has a unique address p, and it can be accessed by $fb[p]$. We assume that the given image initially stored in the frame buffer is already thresholded, and if $fb[p]$ is 0, the pixel is in the background color. Otherwise it is in the foreground color. For the sake of simplicity, we assume that the pixels in the most perimeter including $fb[0]$ are in the background color. We will explain the proposed algorithm using the example input image shown in **Fig. 4** (a). The output of the algorithm is shown in Fig. 4 (h), and each pixel in the foreground color will be substituted with the label, which is the smallest address of the pixels in the same connected component.

Algorithm 1 Preparation for Connected Components Labeling

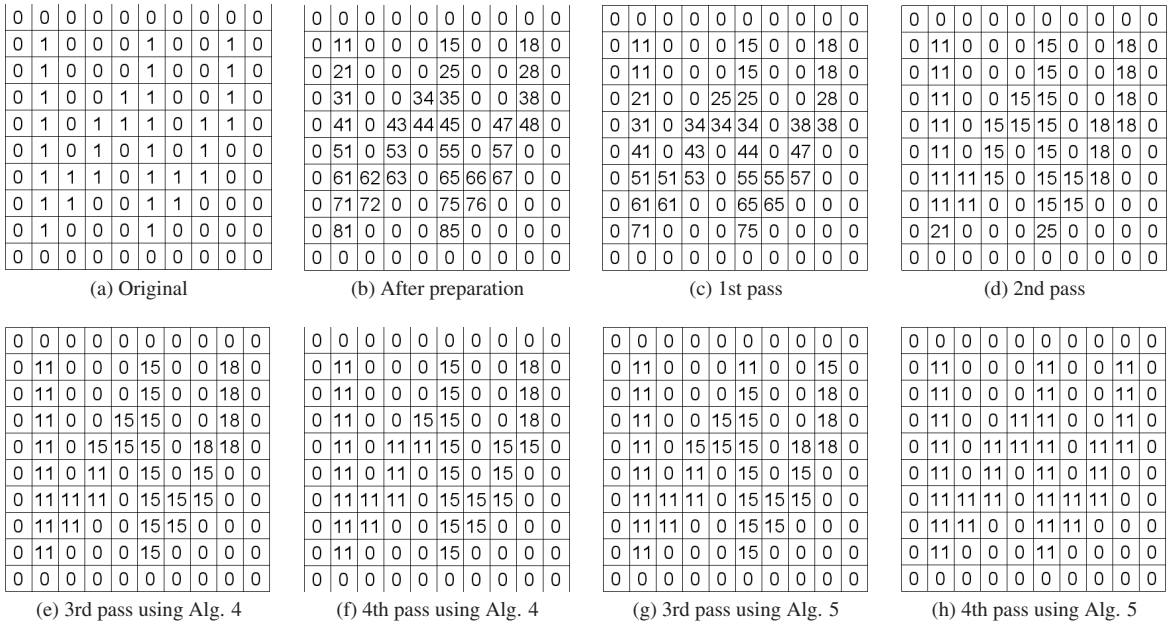
Input: Pointer to the frame buffer fb, pixel number p.

```
1: if fb[p] = 0 then
2:     fb[p] = 0;
3: else
4:     fb[p] = p;
5: end if
```

The proposed algorithm consists of the preparation part and the propagation part. The preparation kernel is first applied to all pixels once at the initial stage. Then, the propagation kernel will be applied to all pixels several times. The preparation part is shown in Algorithm 1, and it is applied to all pixels in parallel. As a result, the pixels in the foreground color are substituted with the address of each one (Fig. 4 (b)).

Algorithm 3 is a naive algorithm for the propagation part. Function CCLSub shown in Algorithm 2 called by Algorithm 3 returns the smallest label stored in the connected adjacent pixels. By applying Algorithm 3 once to the all pixels, the labels can be propagated by one pixel as shown in Fig. 4 (c).

Now, we try to improve the propagation speed of Algorithm 3.



(a) Original (b) After preparation (c) 1st pass (d) 2nd pass

(e) 3rd pass using Alg. 4 (f) 4th pass using Alg. 4 (g) 3rd pass using Alg. 5 (h) 4th pass using Alg. 5

Fig. 4 Transition of pixel value in frame buffer.

Algorithm 2 CCLSub(fb, p)

Input: Pointer to the frame buffer fb, pixel number p.

```
1: g = fb[p];
2: if g ≠ 0 then
3:     s = fb[up(p)];
4:     if s ≠ 0 ∧ s < g then
5:         g = s;
6:     end if
7:     s = fb[down(p)];
8:     if s ≠ 0 ∧ s < g then
9:         g = s;
10:    end if
11:    s = fb[left(p)];
12:    if s ≠ 0 ∧ s < g then
13:        g = s;
14:    end if
15:    s = fb[right(p)];
16:    if s ≠ 0 ∧ s < g then
17:        g = s;
18:    end if
19: end if
20: return g;
```

Algorithm 3 Naive CCL

Input: Pointer to the frame buffer fb, pixel number p.

```
1: g = CCLSub(fb, p);
2: if g ≠ 0 then
3:     fb[p] = g;
4: end if
```

Algorithm 4 shows an improved version. Applying it to Fig. 4 (b) results in Fig. 4 (c), and there is no difference. However, we apply the algorithm again and we get Fig. 4 (d). By the effect of line 3 in Algorithm 4, labels are propagated further than 1 pixel. However, if we apply the algorithm once or twice more, we get the results shown in Fig. 4 (e) and (f), respectively, and we see that after the labels are once propagated vertically to the ends, the algorithm

Algorithm 4 Still Slow CCL

Input: Pointer to the frame buffer fb, pixel number p.

```
1: g = CCLSub(fb, p);
2: if g ≠ 0 then
3:     g = fb[fb[fb[fb[g]]]];
4:     fb[p] = g;
5: end if
```

Algorithm 5 Proposed CCL Algorithm

Input: Pointer to the frame buffer fb, pixel number p.

```
1: h = fb[p];
2: g = CCLSub(fb, p);
3: if g ≠ 0 then
4:     g = fb[fb[fb[fb[g]]]];
5:     atomic{ fb[h] = min(fb[h], g); }
6:     atomic{ fb[p] = min(fb[p], g); }
7: end if
```

propagates labels by only constant pixels.

The proposed CCL algorithm is shown in Algorithm 5, and it propagates labels fast after labels are propagated vertically to the ends. Applying this algorithm to Fig. 4 (d) gives (g), we see that label 11 is propagated to pixel 15, and label 15 is propagated to pixel 18 by the effect of line 5. In the next application of the kernel, label 11 will propagate to the pixels that have labels 15 or 18 by the effect of line 4 (Fig. 4 (h)). We will experimentally show that this algorithm propagates labels exponentially in Section 6.3.

5. Subpixel Corner Detection

In this section, we will explain the method for detecting coordinates of the 4 corners of a marker with subpixel accuracy. The method takes approximate coordinates of the corners as input, and it outputs coordinates with improved accuracy. In this method, we measure the distance between the calculated edges from the input coordinates and the real borders of the marker in

Algorithm 6 Algorithm to Find Subpixel Positions of Edges

Input: Ends A and B of line segment, the number m of distance measurement.

1: $sx = 0$; $sxx = 0$; $sxy = 0$; $sy = 0$;
2: $\vec{r}$ = the unit vector orthogonal to AB;
3: **for** ($j = 1$; $j <= m$; $j++$) (in parallel) **do**
4: $\vec{p_0} = A + \frac{j}{m+1}\vec{AB}$;
5: $sum = 0$; $summax = -HUGE$; $argmax = 0$;
6: **for** ($i = -n$; $i <= n$; $i++$) **do**
7: $\vec{p} = \vec{p_0} + u \cdot i \cdot \vec{r}$;
8: $sum = sum + (c(\vec{p}) - t)$;
9: **if** $sum > summax$ **then**
10: $summax = sum$; $argmax = i$;
11: **end if**
12: **end for**
13: $d = u \cdot argmax$;
14: **atomic**{ $sx = sx + \frac{j}{m+1}$; }
15: **atomic**{ $sxx = sxx + (\frac{j}{m+1})^2$; }
16: **atomic**{ $sxy = sxy + d \cdot \frac{j}{m+1}$; }
17: **atomic**{ $sy = sy + d$; }
18: **end for**
19: $det = sxx \cdot m - sx^2$;
20: $a = \frac{-sx \cdot sxy + sxx \cdot sy}{det}$;
21: $b = \frac{m \cdot sxy - sx \cdot sy}{det}$;
22: $A' = A + a \cdot \vec{r}$;
23: $B' = B + (a + b) \cdot \vec{r}$;
24: **return** A', B';

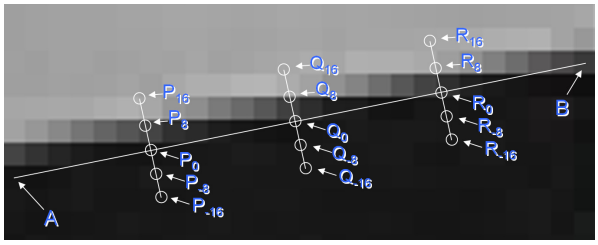


Fig. 5 Example of marker edge in captured image.

the image. We measure distances at multiple points on the edge, and apply linear regression to the measured distances to get the distance between the input corner coordinates and the real border in the image. We then correct the positions of the 4 edges, and, by taking the intersections of neighboring edges, we calculate the accurate corner positions.

5.1 Measuring Distance Between Input Edge and Real Border

We now explain how to measure the distance between an input edge and the real border in the image using the example shown in **Fig. 5**. Suppose that A and B are the two ends of an edge derived by Step C1 in Section 3.5. We calculate the distance at multiple points on the edge with subpixel accuracy. We now explain how to make measurement at P_0. $\vec{r}$ is defined as the unit vector orthogonal to AB. P_i is defined as $P_i = P_0 + u \cdot i \cdot \vec{r}$. The pixel value at p interpolated by the bicubic method is denoted by $c(p)$. The threshold of the pixel is denoted by t, and the number of sampled points is denoted by n. The value of u and n should be adjusted by the user, but we use $u = 1/8, n = 32$ here. We define the distance d between the edge and the real border by the following equation, and the measurement can be made by calculating the value d by this equation.

$$d = u \cdot \arg\max_x \left\{ \sum_{i=-n}^{x} (t - c(P_i)) + \sum_{i=x+1}^{n} (c(P_i) - t) \right\} \quad (1)$$

We make the same measurements at Q_0 and R_0.

5.2 Adjusting Edges and Corners by Linear Regression

We now adjust the edges and the corners based on the measured distances. We denote the distance measured at p_k by d_k. In this case, we apply linear regression to the pairs $\left(\frac{|p_k A|}{|BA|}, d_k\right)$, and we get a and b such that $d_k \approx a + b\frac{|p_k A|}{|BA|}$. Then, the adjusted ends A' and B' of the edge can be calculated as follows.

$$A' = A + a \cdot \vec{r}$$
$$B' = B + (a + b) \cdot \vec{r}$$

We apply the above operation to all 4 edges. Then, the adjusted corners are calculated by extending the adjusted edges and finding the intersections of the adjacent edges.

5.3 Parallel Algorithm

We now describe how to efficiently execute the above method in parallel. First, Eq. (1) can be transformed as follows.

$$\begin{aligned} d &= u \cdot \arg\max_x \left\{ -\sum_{i=-n}^{x} (c(P_i) - t) + \sum_{i=x+1}^{n} (c(P_i) - t) \right\} \\ &= u \cdot \arg\max_x \left\{ -\sum_{i=-n}^{n} (c(P_i) - t) + 2\sum_{i=x+1}^{n} (c(P_i) - t) \right\} \\ &= u \cdot \arg\max_x \sum_{i=x+1}^{n} (c(P_i) - t) \end{aligned} \quad (2)$$

Algorithm 6 shows the proposed parallel algorithm, where the loop from line 3 to 18 can be executed in parallel. From line 6 to 12, Eq. (2) is calculated and the result is stored in d. From line 14 to line 17, and from line 19 to line 21, the calculations for the linear regression are performed. This algorithm adjusts only one edge, and this can be applied to the all edges in parallel.

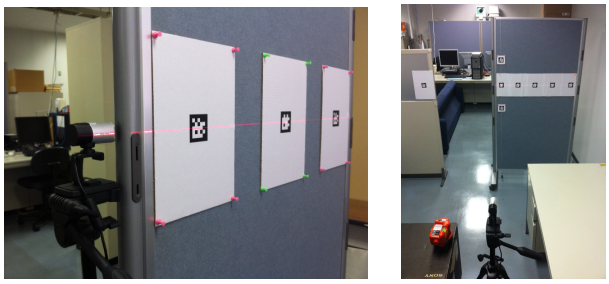
Although the algorithm can be executed in parallel, we implemented the subpixel corner detection algorithm in a nonparallel fashion in this paper.

6. Evaluation

In this section, we show the results from evaluating of the proposed method. We first compare the tracking accuracy of the proposed method with ARToolKit. Then, we compare the execution speed of the two methods. Then, we show some evaluation results of the proposed connected component labeling algorithm. Finally, we show the evaluation of detection accuracy by the proposed subpixel corner detection algorithm.

6.1 Tracking Accuracy

In order to measure the accuracy for the two methods, we used a Microsoft LifeCam Studio Q2F-00008 web camera. Our preliminary experiments revealed that the angle of view of the camera is changed by the autofocus function, and thus we turned off



(a) Adjusting heights of markers (b) Measurements

Fig. 6 Setting up markers and camera.

the autofocus function and set the focus to infinity.

We used 5 cm × 5 cm markers placed at a height of 1.5 m. We used a BOSCH DLE40 laser range finder with accuracy of ±1.5 mm to measure the distance during setting up. We set up the markers and the camera as follows.

(1) We used a string with weight to place the markers and the camera at the same height. We confirmed that they are placed exactly by a laser beam (**Fig. 6** (a)).

(2) We attached a marker on a cubicle partition, and then we put marks on the left, right, top and bottom of the marker at the same distance from the marker. Then, we measure the distance between the camera and the markers with a laser range finder, and we adjusted the camera position so that the distances are equalized.

(3) We took a string with weight and a horizontal laser beam with the camera, and adjusted the camera so that they become perfectly vertical and horizontal in the captured image.

(4) We attached markers on the partition at the same height, and the adjusted their positions based on the distance between the markers.

We used three different resolutions: 640 × 480 (480 p), 1280 × 720 (720 p), and 1920 × 1080 (1080 p). We obtained the results by setting up the camera and markers, starting the applications, waiting for about 10 seconds until the video stabilizes, and then taking and processing 1000 frames of video.

We measured the error in distance measurement for each method. We placed the camera in front of a marker as shown in Fig. 6 (b), and changed the distance between the marker and the camera from 150, 250 and 350 cm. We used a marker with 50 × 50 mm of size. **Figure 7** shows the component ratios for distance measurement errors at video frames. Since only one marker is captured with the camera, only one marker should be detected for each video frame. If no marker is detected from a video frame, this frame is classified into the "false negative" category. If more than one markers are detected, that frame is classified into the "false positive" category. For video frames with only one marker detected, we calculated the difference between the measured distance and the real distance from the camera to the center of the marker, and calculated the ratio of the frames in each category according to the measurement errors to make the graph.

We can see that the proposed method shows significantly better results compared to ARToolKit above 720 p. We can clearly see the subpixel detection method is beneficial to the accuracy of the method, and that detecting from higher resolution videos are beneficial in recognizing and measuring distances of markers.

We evaluated the accuracy in measurement of the direction of

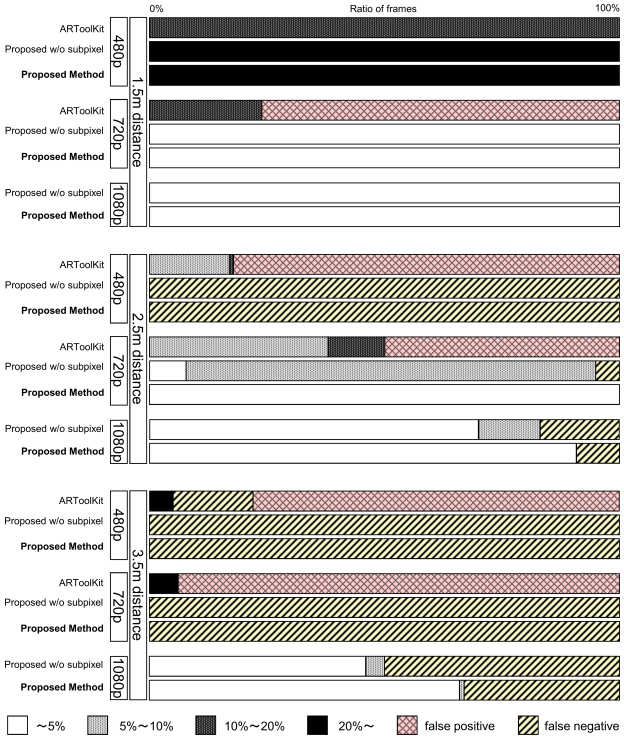


Fig. 7 Error in distance measurement.

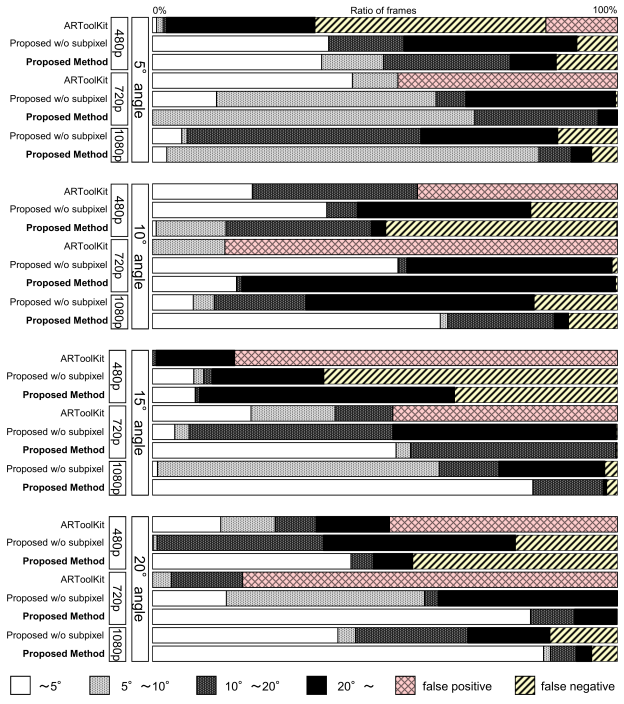


Fig. 8 Error in angle measurement.

normal vector of a marker. We attached a marker in the same size as above on a cubicle partition. We placed the camera at 2 m of distance from the partition, and we directed the camera perpendicularly to the partition. Thus, the normal vectors of the all markers are parallel to the direction of the camera. We changed the angle between the direction of the camera and the direction of a marker from 5° to 20° at 5° interval. The results are shown in **Fig. 8**. This graph shows the component ratios for angular measurement errors at video frames. Video frames from which no marker is detected and more than one markers are detected are

(a) COMPLEX input image (b) SIMPLE input image

Fig. 9 Input images for measuring execution time.

classified into "false negative" and "false positive" categories, respectively. For video frames with only one marker detected, we calculated the angle between the measured normal vector of the detected marker and the direction of the camera, and calculated the ratio of the frames in each category according to the angles to make the graph. We can see that our method provides better results above 720 p. We can see the effectiveness of the sub-pixel corner detection method and detecting from higher resolution videos.

6.2 Tracking Speed

We compared the time for the proposed method and ARToolKit to process one video frame. We implemented the proposed method using Java SE 7 (64 bit) and OpenCL via JOCL (Java bindings for OpenCL). We also compared the cases where the OpenCL kernels and other parts in the proposed method are all executed on the CPU, and the case where the OpenCL kernels are executed on the GPU. We used PCs with the following two configurations for the speed evaluation. We used a Logitech HD Pro Webcam C920 web camera for capturing video, and AMD Accelerated Parallel Processing SDK for executing OpenCL kernels on the CPU devices.

OLDPC CPU : Intel Core 2 Quad 6600 (2.4 GHz), GPU : nVidia GeForce GTX 560 Ti, OS : Windows 7 Enterprise (64 bit)

NEWPC CPU : Intel Core i7 3770T (2.5 GHz), GPU : nVidia GeForce GTX 670, OS : Windows 7 Enterprise (64 bit)

We used the images shown in **Fig. 9** (a) and (b) (COMPLEX and SIMPLE images, hereafter) as input images captured with the web camera. We placed one corresponding marker in the SIMPLE images and three corresponding markers of the same size in the COMPLEX images for the methods. We measured the time for detection, excluding the time for capturing images from the camera and displaying the resulting images. The execution time was measured using the profiling functions built in the OpenCL framework, and the System.currentTimeMillis function in Java. The shown values are averages over 100 frames. The results are shown in **Fig. 10**. We can see that the proposed method runs more than two times as fast as ARToolKit when GPU is used. We measured the execution time for each part in the proposed method on the NEWPC. The results are shown in **Fig. 11**. We can see that the parts executed on the CPU are not taking excessive execution time, and the bottleneck is the CCL.

6.3 Connected Component Labeling

We first investigated how many kernel applications are needed to complete labeling by the proposed method. We used square images with simple text, spiral patterns and zigzag spiral patterns as shown in **Fig. 12** (a), (b) and (c), respectively. The black and white pixels in the figures represent the pixels in the background and foreground colors, respectively. We varied the pattern sizes and counted the number of kernel applications until the correct labeling was derived. When we changed the image size, we magnified or reduced the size of the font according to the image size for the text pattern, and we changed the number of rotations of the spiral and zigzag spiral patterns while the width of the patterns were maintained to 1 pixel. The results are shown in **Table 1**. The image sizes in the table show the lengths in pixels of one side of the square image. We see only a linear increase in the number of kernel applications when the image size is exponentially increased.

We investigated the execution speed of the CCL algorithms. We specified the three patterns used above with

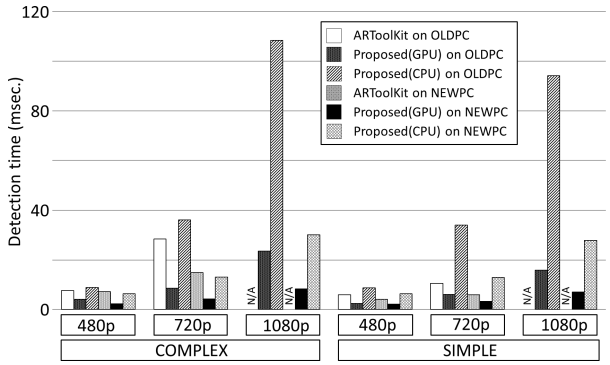


Fig. 10 Comparison of detection time.

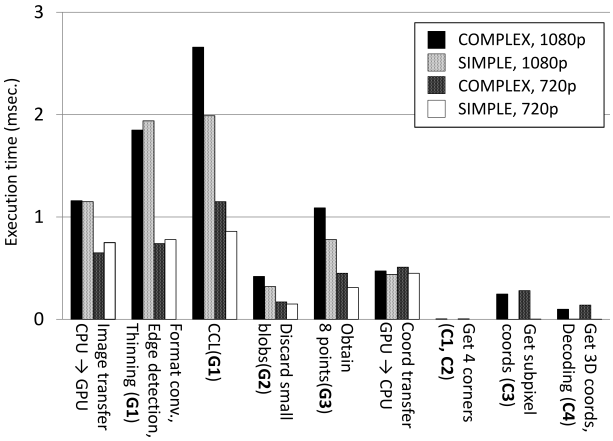
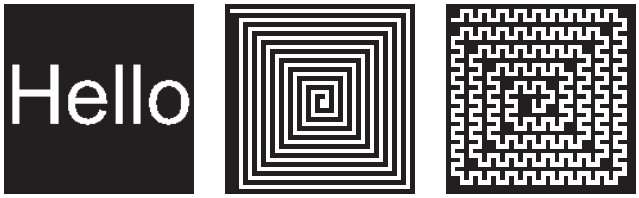


Fig. 11 Execution time of each part.



(a) TEXT (160 × 160 pixels) (b) SPIRAL (40 × 40 pixels) (c) ZSPIRAL (40×40 pixels)

Fig. 12 Patterns for evaluating CCL algorithm.

Table 1 Number of kernel applications to finish labeling.

Image size	40	80	160	320	640	1280
TEXT	5	5	5	6	6	7
SPIRAL	6	7	8	9	10	11
ZSPIRAL	7	8	9	10	10	11

Table 2 Time to execute CCL (msec.).

Pattern	TEXT	SPIRAL	ZSPIRAL
Proposed on OLDPC	2.4	9.8	11.9
OpenCV on OLDPC	6.6	94.2	78.4
Proposed on NEWPC	1.5	3.4	3.2
OpenCV on NEWPC	3.6	54.3	42.1

Table 3 Corner detection accuracy (pixel).

		Noise intensity				
		0	2	4	8	16
Blur radius	–	0.077	0.077	0.079	0.092	1.2
	2	0.069	0.069	0.073	0.26	0.38
	4	0.065	0.082	0.13	0.93	1.1
	6	0.29	0.30	0.37	0.76	2.1
	8	1.1	1.1	1.2	1.5	2.9

1280×1280 image size as input, and measured the execution times on GPU excluding the time for transferring data between CPU and GPU. We executed Algorithm 1 once and Algorithm 5 eleven times. As a comparison, we executed the implementation of the method in Ref. [1] obtained from http://opencv.willowgarage.com/wiki/cvBlobsLib. We used g++ version 4.6.3 x86_64 with an optimization option –O3 in order to compile the implementation, and we executed the binary on Ubuntu 12.04 OS (64 bit) with VirtualBox 4.1.18. **Table 2** shows the result.

We can see that the proposed method is significantly faster on all patterns. Although the number of kernel applications are same, we see differences in the execution time for the proposed method. One explanation for this is the difference in cache hit ratios.

6.4 Subpixel Corner Detection

We describe the result of applications of the proposed subpixel corner detection method to synthetic images with blurs and noise. We specified $u = 1/8$ and $n = 32$, and we made measurements at 8 points between A and B.

We used the BufferedImage and Graphics2D classes in Java SE 6 with the anti-alias option turned on. We generated the input images as follows. We first made a BufferedImage object with 512×512 pixels, and filled it with the color of (205, 205, 205). We then drew a rotated square 300 pixels on a side at the center of the buffer with the color of (51, 51, 51). We confirmed that the corners of the square can be specified by subpixel precision, and the fractions are reflected on the drawn square. We then applied a Gaussian Blur filter to the buffer, and then we added noise with uniform random numbers. We specified this image as the input of the proposed method, and observed the distances between the detected corners and the specified coordinates of the square. **Table 3** shows the average distances of 1,000 trials with different angles of rotation for the square. Noise intensity in the table represents the intensity of the added noise, and adding noise of intensity at in means that we added uniform random numbers within $[-in, in]$ to each pixel. Blur radius r in the table represents the radii of the Gaussian Blur filters, and it means that the standard deviation of the Gaussian distribution is $r/3$.

The results show that the proposed method can detect corner coordinates with less than 0.15 pixels of accuracy with up to 4 pixels of the blur radius and up to 4 pixels of noise intensity.

7. Conclusion

We proposed a method for detecting the 3D positions and poses of traditional fiducial markers in high-resolution images utilizing GPGPU computation. Our main contributions in this paper are a fast CCL algorithm for GPGPU and a method to obtain subpixel coordinates of the corners of the markers. Our experiments showed that our method is faster and more accurate than ARToolKit. We are distributing our implementation at our web site https://freecode.com/projects/gpumarkertracker.

References

[1] Chang, F., Chen, C. and Lu, C.: A linear-time component-labeling algorithm using contour tracing technique, *Comput. Vis. Image Underst.*, Vol.93, No.2, pp.206–220 (2004).

[2] Chen, D. and Zhang, G.: A new sub-pixel detector for x-corners in camera calibration targets, *Proc. 13th Intl Conf. Central Europe on Computer Graphics Visualization and Computer Vision*, pp.97–100 (2005).

[3] Donoser, M., Kontschieder, P. and Bischof, H.: Robust planar target tracking and pose estimation from a single concavity, *Proc. 2011 10th IEEE Intl. Symposium on Mixed and Augmented Reality, ISMAR '11*, pp.9–15 (2011).

[4] Ferretti, S., Roccetti, M. and Strozzi, F.: On developing tangible interfaces for video streaming control: A real case study, *Proc. 18th Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video, NOSSDAV '08*, pp.51–56 (2008).

[5] Fiala, M.: Artag, a fiducial marker system using digital techniques, *Conf. Computer Vision and Pattern Recognition, CVPR 2005*, Vol.2, pp.590–596 (2005).

[6] Gioi, R., Jakubowicz, J., Morel, J. and Randall, G.: On straight line segment detection, *J. Math. Imaging Vis.*, Vol.32, No.3, pp.313–347 (2008).

[7] Hawick, K., Leist, A. and Playne, D.: Parallel graph component labelling with GPUs and CUDA, *Parallel Comput.*, Vol.36, No.12, pp.655–678 (2010).

[8] Heikkila, J. and Silven, O.: A four-step camera calibration procedure with implicit image correction, *Proc. Conf. Computer Vision and Pattern Recognition*, pp.1106–1112 (1997).

[9] Ito, E., Okatani, T. and Deguchi, K.: Accurate and robust planar tracking based on a model of image sampling and reconstruction process, *Proc. 2011 10th IEEE Intl. Symposium on Mixed and Augmented Reality, ISMAR '11*, pp.1–8 (2011).

[10] Jung, I. and Jeong, C.: Parallel connected-component labeling algorithm for GPGPU applications, *Intl. Symposium on Communications and Information Technologies* (*ISCIT*), pp.1149–1153 (2010).

[11] Kaltenbrunner, M. and Bencina, R.: Reactivision: A computer-vision framework for table-based tangible interaction, *Proc. 1st Intl. Conf. Tangible and Embedded Interaction, TEI '07*, pp.69–74 (2007).

[12] Kato, H. and Billinghurst, M.: Marker tracking and hmd calibration for a video-based augmented reality conferencing system, *Proc. Augmented Reality* (*IWAR '99*), pp.85–94 (1999).

[13] Suzuki, K., Horiba, I. and Sugie, N.: Fast connected-component labeling through sequential local operations in the course of forward raster scan followed by backward raster scan, *IPSJ Journal*, Vol.41, No.11, pp.3070–3081 (2000).

[14] Komatsuzaki, M., Tsukada, K. and Siio, I.: Drawerfinder: Finding items in storage boxes using pictures and visual markers, *Proc. 16th Intl. Conf. Intelligent user Interfaces*, *IUI'11*, pp.363–366 (2011).

[15] Mulloni, A., Wagner, D., Barakonyi, I. and Schmalstieg, D.: Indoor positioning and navigation with camera phones, *IEEE Pervasive Computing*, Vol.8, No.2, pp.22–31 (2009).

[16] Owens, J., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. and Purcell, T.: A survey of general-purpose computation on graphics hardware, *Computer Graphics Forum*, Vol.26, No.1, pp.80–113 (2007).

[17] Park, J., Looney, G. and Chen, H.: Fast connected component labeling algorithm using a divide and conquer technique, *Computers and Their Applications*, pp.373–376 (2000).

Naoki Shibata is an Associate Professor at Nara Institute of Science and Technology. He received his Ph.D. degree in Computer Science from Osaka University, Japan, in 2001. He was an Assistant Professor at Nara Institute of Science and Technology 2001–2003 and an Associate Professor at Shiga University 2004–2012. His research areas include mobile computing, inter-vehicle communication, multimedia communication, and parallel algorithms. He is a member of IPSJ, ACM and IEEE/CS.

Shinya Yamamoto received his Ph.D. (Engineering) degree from the Nara Institute of Science and Technology, Japan, in 2009. He is currently an Assistant Professor in the Department of Electrical Engineering at The Tokyo University of Science, Yamaguchi. His research interests include networks, distributed virtual environments, and ubiquitous systems. He is a member of IPSJ and ACM.