ATLANTIS PRESS

Advances in Intelligent Systems Research, volume 173

1st International Conference on Engineering and Management in Industrial System (ICOEMIS 2019)

Worker Ergonomics Surveillance in Industrial Environments Based on Parallel Computing on Face Camera Using OpenCV and CUDA

Imam Cholissodin[1, a], I Made Setia Baruna[2,b], M. Ananta Pratama[3,c], Dwi Rama Malawat[4,d] and Dewi Hardiningtyas[5,e]

[1,2,3,4]Informatics Engineering Department Brawijaya University, Indonesia

[5]Industrial Engineering Department Brawijaya University, Indonesia

[a]imamcs@ub.ac.id*, [b]made.setia@gmail.com, [c]pratama.ananta15@gmail.com, [d]dwiramamalawat10@gmail.com, [e]dewi.tyas@ub.ac.id

Keywords: worker ergonomics surveillance, industrial environment, parallel computing, face camera, opencv, cuda.

Abstract. Modern applications today have used automation contexts to adjust application behavior to the received context, especially in industrial environment. This can highly increase the user experience of ergonomic comfort, safety, and health of workers. This context can be concluded by a program using variety of sensors possessed by a device. For example, using GPS to get a location so that it can provide recommendations for places that can be visited according to user interests or using voice to give orders to a program. One context that is still rarely used is the context of the distance between the face of a worker and the computer screen he uses. This context is one of the very important contexts in the field of workers' health considering the number of device users who experience health problems due to not paying attention to the distance between the eyes and the screen of the device used. Therefore, the system we created is able to detect the distance between the face and the screen of the device by measuring the distance between the worker's eyes using the front camera installed on the device used. Thus, when the distance of the face and screen is too close, the program can give a warning to workers to keep the distance between the face and the screen of the device for the health of the workers themselves.

Introduction

Making a modern application requires a context to perform something new, besides using a mouse or a keyboard. There are a lot of contexts depending on the sensors contained in the device, for example using GPS to get a location so that it can provide places that might be of interest according to the closest location of the user or using sound to give commands to the operating system. The distance between the face and the camera or screen is a new context that is still rarely used, yet very useful. For example in the field of health, it can be used to detect eye diseases, or it can also be used to make user experience on the applications, for example by making the font size in the application adjust automatically according to the distance of the eyes from the screen. This can result in better reading comfort. On devices such as laptops or personal computers, there are already cameras or webcams that are used for video calls but have not been used for anything else. In this paper, we want to use a webcam on a laptop to measure the distance from the eyes to the camera which can later be developed to create applications that can take the advantage of this context. Google itself since Android 4.0 IceCream Sandwich has by default added an API to detect faces and eyes as well as to calculate the distance from the eyes.

In the previous studies, researchers used the distance between the two eyes of the user in front of the camera, then calculated the distance of both eyes and used the average value that had been calculated to get the distance to the camera [1]. This can be inaccurate because the eye size can make the distance of each user's eyes different. In addition, the resolution and lens of each camera can affect the eye distance measurement. In other studies, it is more focused on evaluating the distance of the screen with eye when using a computer based wireless camera without parallel computing. It's by

collecting data to find the relationship between activities such as when the user doing Setup, Internet, Word, Misc. to get average distance of eye to screen [2]. Then, the next study tried to improve the method from Rahman et al. by adding an initialization phase by measuring the average distance when the application is firstly used, by asking the user to use A4 paper as a benchmark measure during calibration [3].

In this development, they employed OpenCV and Android Face Detection API which requires an average time of 2.5 seconds to detect faces and the average CPU time usage of which is 99% to detect the eyes and face only because every pixel in the image from the video needs to be processed, depending on the image resolution used as well. These results were obtained from *LG Nexus 4 (E960) smartphone*. Taking this into the account, we are interested in optimizing the face and eye detection process into the GPU to reduce the load on the CPU. We will use *OpenCV* and *Haar Cascade* algorithm to detect faces and eyes supported by CUDA as a medium for the programming on the GPU.

Basic Theory and Methods

1. Measuring Face Distance to the Camera

To measure the eye distance, the camera must be able to detect the two eyes on the face (Fig. 1), then the distance of the pixel obtained is calculated with Euclidean Distance (Eq. 1).

$$\text{Euclidean Distance} = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2} \tag{1}$$

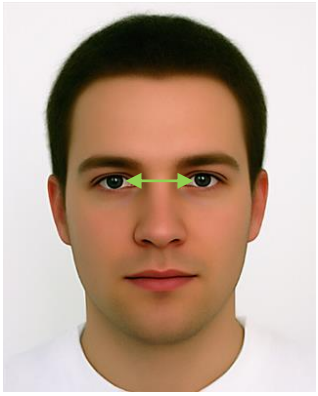

Figure 1. Distance between eyes.

At the beginning of the application run, calibration must be performed using A4 paper (29.7 cm benchmark) as a benchmark between the face with the camera. Then 50 average distances of the eyes are taken and stored. After that, the face distance can be calculated with the following Eq. 2.

$$\text{Face Distance} = d_{ref} * \left(P_{ref} \Big/ P_{sf} \right) \tag{2}$$

d_{ref} is the benchmark distance used, namely A4 paper size measuring 29.7 cm long, p_{ref} is the average distance of the eyes obtained during calibration, and p_{sf} is the eye distance when the measurement takes place.

2. OpenCV and CUDA

Based on Fig. 2a, OpenCV (*Open Source Computer Vision Library*) is an *open source library* for *computer vision* and *machine learning* that is free to use for academic and commercial use. OpenCV is designed to build and accelerate the infrastructure of *computer vision* applications. OpenCV has

several supports for programming languages such as C++, Python, Java, and Matlab and can be operated on Windows, Linux, Mac OS, and Android operating systems [4]. In OpenCV, there is also support for processing with CPU, GPU with CUDA or OpenCL. To use CUDA, you can compile from the source and enable CUDA yourself.

Figure 2a. OpenCV Logo.

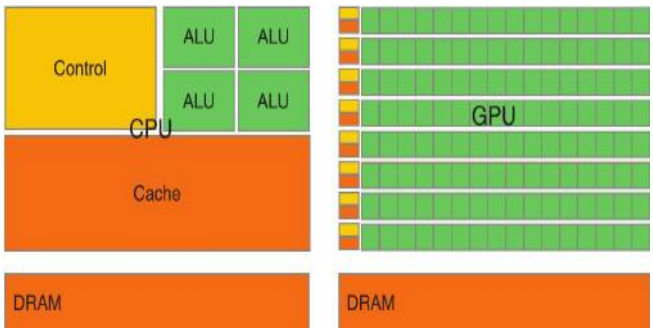

Figure 2b. CPU and GPU Comparison.

Based on Fig. 2b, CUDA is a platform for parallel computing programming made by NVIDIA for use on NVIDIA GPUs [5]. CUDA has been widely used to accelerate calculations on *computer vision* because the values in image pixels can be broken down into thousands of codes compared to CPUs the average of which only have 8 cores. We use CUDA implementations from OpenCV that we have compiled with CUDA enabled. From OpenCV, we use cuda::GpuMat to use the matrix on CPU as the image media and also cuda:: CascadeClassifier as the face and eye detection. From the coordinate data obtained from OpenCV, we calculate the eye and face distance according to the method above.

Design and Implementation

The literature study in this research uses references from a paper entitled *A New Context: Screen to Face Distance* by *Immanuel König*, *Phillip Beau*, and *Klaus David*. This paper uses an algorithm to obtain the distance between the user's face and the smartphone used by calculating the distance between the user's eyes so that the distance between the face and the smartphone can be determined. At the need analysis stage, observations are made on how facial distance is measured using a webcam and looking for the average distance for the required calculations. The system design phase is done using C++, OpenCV with CUDA and CPU to compare whether the result is in accordance with the previous calculations. After the implementation phase is completed, it will continue with the testing phase to find out whether the implemented program has been in accordance with the needs that have been obtained at the need analysis and system design phases. There is a white box testing that will compare FPS (frames per second) and CPU usage to compare whether the implementation with GPU is faster or vice versa than that with CPU.

The final stage of this research is to draw the conclusion from the research that has been done after all the previous stages have been completed. The conclusion contains the answer to the formulation of the problem whether the performance of the program using GPU is better than the performance of the program using CPU. The program code used in this study was made from the beginning using OpenCV that can be downloaded from the website. Then, the specification of the hardware and software requirements when implementing and testing are as follows:

- Intel i3 7100
- RAM 8 GB
- NVIDIA GeForce GTX 1060
- Windows 10
- Visual Studio Community 2015
- CUDA Toolkit 8.0
- OpenCV compiled with CUDA enabled

The following is the function for detecting faces and eyes with OpenCV and CUDA.

```
52  input_gpu.upload(input);
53  cv::cuda::cvtColor(input_gpu, output_gpu, CV_BGR2GRAY);
54  cascade_gpu->setFindLargestObject(true);
55  cascade_gpu->setScaleFactor(1.1);
56  cascade_gpu->setMinNeighbors(4);
57  cascade_gpu->setMinObjectSize(cv::Size(50, 50));
58  cascade_gpu->detectMultiScale(output_gpu, faces_gpu);
59  cascade_gpu->convert(faces_gpu, faces);
```

Source Code 1. Face Detection Feature

In the face detection code above, the image matrix, which is the input, will be uploaded from the host to the GPU device. Then, the GPU will perform detectMultiScale that will detect where the face is. After obtaining it on the GPU, it will be converted to take the obtained vector value back to the host.

```
64  cv::cuda::GpuMat roi = output_gpu(faces[0]);
65  std::vector<cv::Point> eyesCenter;
66  cascade_eye_gpu->detectMultiScale(roi, eyes_gpu);
67  cascade_gpu->setFindLargestObject(true);
68  cascade_gpu->setScaleFactor(1.1);
69  cascade_gpu->setMinNeighbors(100);
70  cascade_gpu->setMinObjectSize(cv::Size(50, 50));
71  cascade_gpu->convert(eyes_gpu, eyes);
72  for (size_t j = 0; j < eyes.size(); j++)
73  {
74  eyesCenter.push_back(cv::Point(faces[0].x + eyes[j].x + eyes[j].width / 2, faces[0].y +
75  eyes[j].y + eyes[j].height / 2));
76  }
```

Source Code 2. Eye Detection Feature

In the eye detection code above, the face portion will only be taken from the image from the camera and stored on the ROI variable. Then, detectMultiScale will be carried out to get all the detection results. After that, the midpoint of all the detection results is searched for and stored in the eyesCenter vector.

```
84  double eyeDistance = cv::norm(cv::Point(eyesCenter[0].x, eyesCenter[0].y) -
    cv::Point(eyesCenter[1].x, eyesCenter[1].y));
85  faceDistance = AVERAGE_DISTANCE * (averageDistance / eyeDistance);
```

Source Code 3. Face Distance Calculation Feature

After obtaining the 2 eyes, the distance between the 2 eyes is taken. Then, face distance is calculated by a formula that has been explained on the basic theory above, namely in Eq. 1 and Eq. 2. Next, the results of the program are shown in Fig. 3a and Fig. 3b, each of which automatically obtains the distance of workers with the monitor screen with the condition of a close face and a distant face.

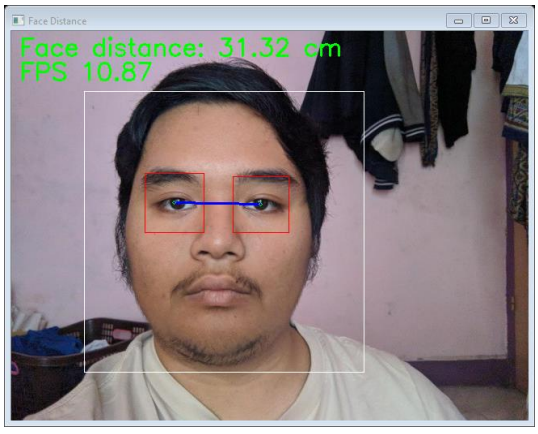

Figure 3a. The program is run with a close face.

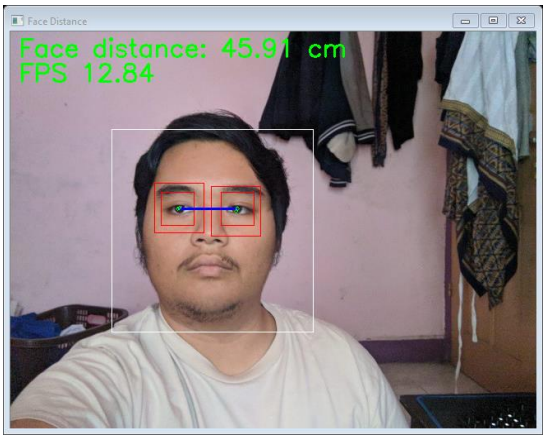

Figure 3b. The program is run with a far face.

Testing and Analysis

In first test, we recorded FPS when running with CPU and GPU 100 times. The following Fig. 4a of the obtained data, the average FPS when running using CPU is 14.01 FPS whereas if it is run with GPU, the result is 42.95 FPS. Then, in second test, we record CPU usage with Performance Monitor provided by Windows, every 1 second for 30 seconds when running with CPU and GPU. The following Fig. 4b of the data obtained, in terms of the average FPS, the result shows 68.17% when it is run using CPU while 34.19% is the result if it is run with GPU.

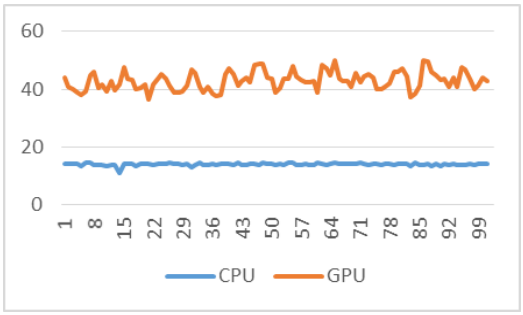

Figure 4a. FPS graph result.

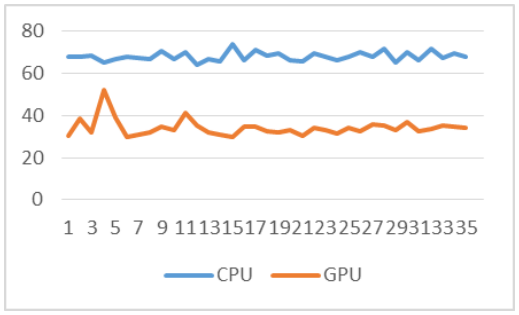

Figure 4b. CPU usage graph result.

Conclusions and Suggestions

From the experiments, the optimization of facial distance computing process by utilizing OpenCV and CUDA as the media to process data previously processed on the CPU and on the GPU in parallel is successful. The testing is carried out by changing the processing of image conversion to grayscale and doing haar cascade detection from CPU to GPU. The results obtained are higher FPS and lower resource usage than the implementations with CPU. From the experimental results, it can be concluded that the implementation of GPU can increase the FPS obtained by 3 times as much as that of CPU on the system we use. In addition, the automatic CPU usage has decreased because some processes are transferred to GPU so that the system using CPU usage decreases by twice as much as that using CPU usage. From the results obtained, it can be concluded that the optimization is successful with a fairly large value reaching twice as much. The suggestion for further research is to be able to increase the performance again by directly taking the coordinates from the face without moving the value to the host first to reduce the overhead. Besides that, it can also implement the detection of more than one face and can be connected with an adaptive reduction system to control the contrast of the screen, so that the eyes of workers are not easily tired.

References

[1] Rahman, K.A.,Hossain, M.S., Bhuiyan, M.A.,Tao Zhang; Hasanuzzaman, M.,Ueno, H., in: Person to Camera Distance Measurement Based on Eye-Distance. Multimedia and Ubiquitous Engineering. MUE '09. Third International Conference on, pp.137, 141 (2009).

[2] Eastwood-Sutherland, C., Gale, T.J., in: Eye-screen distance monitoring for computer use, Engineering in Medicine and Biology Society, EMBC. Annual International Conference of the IEEE. Boston (2011).

[3] König, I., Beau P., David K., in: A new context: Screen to face distance. 8th International Symposium on Medical Information and Communication Technology (ISMICT), Firenze (2014).

[4] OpenCV, in: About OpenCV, accessed from https://opencv.org/about.html on 1 May 2018.

[5] NVIDIA, in: About CUDA, accessed from https://developer.nvidia.com/about-cuda on 1 May 2018.