IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 4 December 2023, accepted 22 December 2023, date of publication 1 January 2024,
date of current version 11 January 2024.

Digital Object Identifier 10.1109/ACCESS.2023.3348478

RESEARCH ARTICLE

Low-Power Lane Detection Unit With Sliding-Based Parallel Segment Detection Accelerator for FPGA

HEUIJEE YUN AND DAEJIN PARK, (Member, IEEE)
School of Electronic and Electrical Engineering, Kyungpook National University, Daegu 41566, South Korea

Corresponding author: Daejin Park (boltanut@knu.ac.kr)

This work was supported in part by the Brain Korea 21 (BK21) 4th Project under Grant 4199990113966; in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education under Grant NRF-2018R1A6A1A03025109 (5%) and Grant NRF-2022R1I1A3069260 (5%); in part by the Ministry of Science and Information Communication Technology (MSIT) under Grant 2020M3H2A1078119; in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) grant funded by the Korean Government (MSIT) through the Metamorphic Approach of Unstructured Validation/Verification for Analyzing Binary Code (20%) under Grant 2021-0-00944; in part by the Processing-in-Memory (PIM) Semiconductor Design Research Center (30%) under Grant 2022-0-01170; and in part by the Development of Flexible Software-Hardware (SW-HW) Conjunctive Solution for On-Edge Self-Supervised Learning (40%) under Grant RS-2023-00228970.

ABSTRACT Recently, with the development of semiconductors and VLSI (Very Large Scale Integrated Circuit), the technology required for autonomous driving is rapidly developing. One of the technologies that cannot be left out is the lane detection function. Lane recognition requires a lot of data from the camera sensor. As a result, the data size increases, making it difficult to process on a lightweight embedded board. This paper proposes a sliding-based parallel segment image processing method to solve this problem. Most boards in autonomous vehicles are lightweight, so the technique has been designed to reduce computation and power consumption. After fetching the image's pixel data, grayscale conversion, Gaussian smoothing, Sobel operator, non-maximum suppression, and hysteresis are performed in parallel. Lanes were detected by performing a Hough transform operation on an image for which edge detection was completed in parallel. Due to the nature of parallel processing, it is more effective when image input is continuous and numerous than single image processing. This algorithm is written in C language and VHDL (VHSIC Hardware Description Language) for two parts in the board, DE1-SoC, FPGA (Field Programmable Gate Array) and HPS (Hard Processor System. Due to the use of the C language and VHDL, parallel programming uses 3.1 times less time, twice as much memory and slightly more power than sequential programming. For hardware languages such as Verilog, the computation algorithms have been converted to a fixed point. When comparing HPS and FPGA, the FPGA consumed significantly fewer resources, with 18 times shorter run time, 50 times fewer clock cycles, 3 times less power, and 183 times less energy. This provides a substantial benefit.

INDEX TERMS Autonomous driving, lane detection, canny edge detection, hough transform, FPGA acceleration, low power design.

I. INTRODUCTION

Video-based advanced driver assistance system (ADAS) is essential for autonomous driving. Common ADAS include lane departure warnings, traffic signs, and pedestrian detection. ADAS has both basic and advanced video/image processing technologies. Fast processing time and low power consumption are critical requirements for driver assistance systems. However, to implement better performance, the resolution must be high and the number of pixel data will grow tremendously. If each pixel data is calculated as a matrix, a lot of computation and time will be consumed.

The associate editor coordinating the review of this manuscript and approving it for publication was Mario Donato Marino.

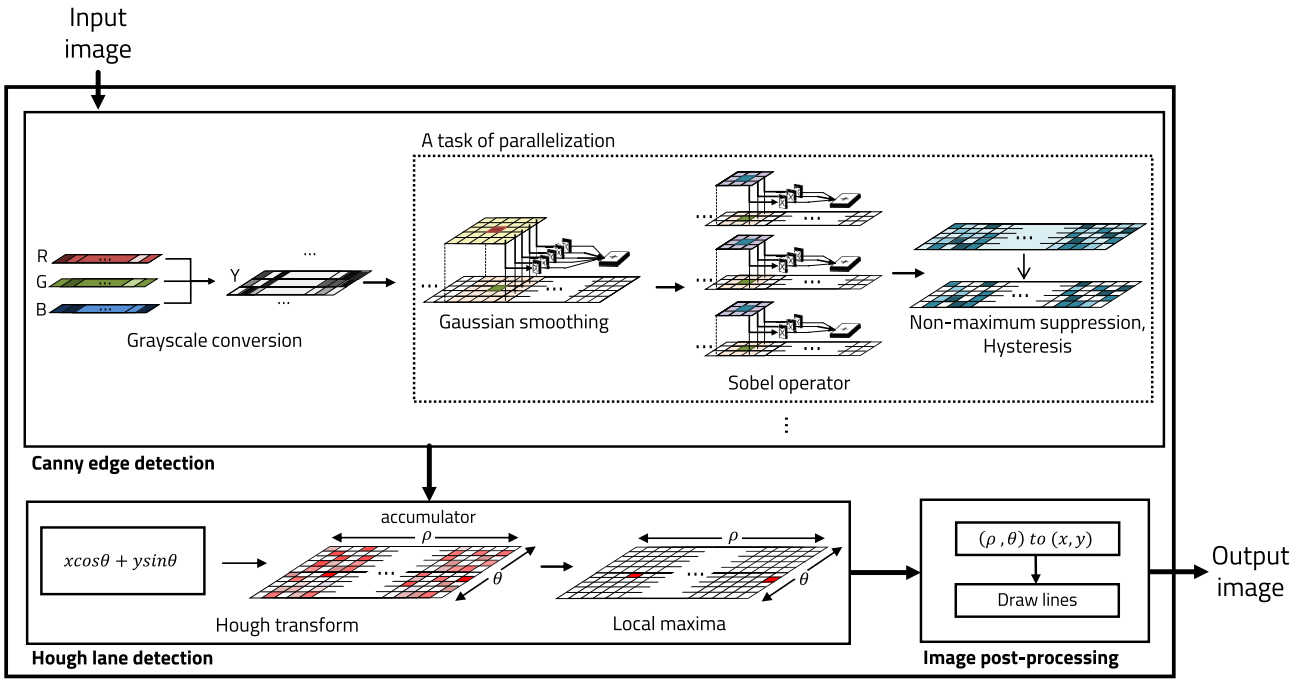


FIGURE 1. Parallelization process of canny edge detection and lane recognition algorithm with hough transform operation.

Also, to be implemented in autonomous driving technology, the processing computation must be conducted in a light weighted embedded board. Lightweighted image processing algorithms using camera sensors have always been a task to be solved, and research is being conducted in various ways [1].

This paper introduces a lane detection algorithm using parallel processing. Several deep learning models and methods can recognize lanes, however, the fundamental techniques of Canny Edge Detection and Hough transform [2], [3] are the ones employed. These two methods are utilized because they are processed as matrices and are the most basic and widely used structures in image processing. Figure 1 shows the overall operation of the algorithm. The first step is to detect all edges of the input image. Canny Edge Detection algorithm is conducted by conducting grayscale conversion, Gaussian smoothing, Sobel operator, non-maxima suppression and hysteresis operation. After collecting all the image data processed in parallel, lanes are detected using Hough transform. Due to the nature of the Hough transform operation, it cannot be processed in parallel because it scans and calculates all pixels in the image at once.

This structure can maximize efficiency by processing the lane detection algorithm in parallel using a buffer. Since this structure divides pixels according to the height of the image and proceeds by threads, the area occupied by the hardware increases as the number of threads increases. Additionally, the time and resource benefits obtained when filter matrix operations are parallelized are expected to outweigh the overhead incurred by parallelization. It is more efficient to read and process multiple small pieces of RAM (Random access memory), even if access times are longer, than sequentially reading one huge piece of RAM [4]. By parallelizing the matrix operations required for image processing, the execution time can be reduced significantly. Also, since the same calculation is performed, the lane recognition result will be the same. The peak power consumption may be higher due to the complicated process of parallelization, but the corresponding execution time will be greatly reduced and the total energy consumption will be reduced. This is a fast way to process high-resolution images using less hardware resources.

There are a number of studies that have applied edge detection algorithms or hough transform to FPGAs using HDL language [5], [6], [7]. However, for lane detection algorithms that integrate edge detection and hough transform, several studies use Linux-based high-level systems (HLS) or high-level languages [8], [9], [10]. When implemented in a high-level system, it is bound to use a lot of resources because it goes from the core to the operating system. To enable implementation on a lightweight board, proposed architecture was implemented at the RTL level using the hardware language.

There are not many papers that have studied lane detection algorithms that combine edge detection and line detection applied to FPGAs at the RTL level [11], [12]. Most researches only parallelized the hough transform or implemented it using OpenCL and GPU parallelism.

The goal was to create a lightweight lane detection algorithm that could run on a single core without a GPU. To make it even lighter, parallelization of the edge detection which requires computing every pixel in the image was used. Unlike other research, proposed algorithm was parallelized and implemented at the RTL level, eliminating the need for APIs and allowing for greater chip flexibility.

Analyzing research trends, it is evident that these technologies incorporate artificial intelligence features, particularly deep learning and machine learning. The majority of research papers are programmed with high-level languages like C and Python, utilizing various APIs, with limited utilization of FPGA or gate-level implementations. The reason that it was restricted is due to a lack of APIs and the necessity for manual transformation of operations and processes. As none of the previous work has implemented all of these processes in a hardware language in parallel, it is worthwhile to pursue this approach.

II. CONSIDERATION OF THE METHOD

Currently, as AI (Artificial intelligence) technologies have advanced, researchers have developed a range of methods for lane recognition.Numerous deep learning architectures are under research and when applied to Advanced Driver Assistance Systems (ADAS), they yield remarkable progress. Notably, numerous studies have investigated precise lane recognition utilizing deep learning techniques. This section describes the rationale for applying traditional canny edge detection and hough transform instead of deep learning in this study.

For comparison purposes, the Verilog HDL (Hardware Description Language) language was used to implement both the widely-used CNN (Convolutional neural network) and FC (Fully Connected) structures in deep learning. However, the primary objective of this study is not the implementation of deep learning for lane recognition, but rather comparison. Therefore, only the structure was implemented. MNIST (Modified National Institute of Standards and Technology database), which has a relatively small dataset and input image size of 28 $\times$ 28, was utilized. The CNN architecture utilized two 2D (Two dimensional) convolution kernels, two max-pooling layers, and one fully connected layer. The optimization algorithm used was SGD (Stochastic Gradient Descent), employing negative log-likelihood and ReLU (Rectified Linear Unit) activation functions. Python was used to extract the weights. Furthermore, the implementation exclusively relied on the FC layer, which consisted of only two layers.

The hardware size of these two deep learning structures was measured through the design compiler. Figure 2 shows the result of hardware measurements of two structures. The CNN structure occupied a total of 132671.75 design areas and the FC structure occupied 2905179.77. The gate count was obtained by dividing by the aforementioned NAND gate area, 1.524. The CNN was calculated as 87054.95 and the FC structure as 1911302.48. This result occupies a very large area to be used in lightweight embedded boards. The process necessitates abundant resources leading to an increase in power consumption and chip size. Despite many developments in chip manufacturing and digital circuit design technologies, current efforts are inadequate. Thus, a more lightweight algorithm can be implemented for lane detection by utilizing Canny edge detection and Hough transform solely for calculation purposes and without any learning. Despite the difference in the artificial intelligence and algorithm employed, our approach demonstrates a significantly lighter algorithm for achieving the objective of lane recognition. Also, as a result of measuring the execution time, CNN takes 0.0135s and FC takes 0.246s. This result is also seen as resource intensive because the hardware structure has to run in fewer clock cycles using fast frequencies.

Measurement	CNN	FC
Combinational area	62875.73	1596309.20
Non-combinational area	34826.36	186758.23
Design area	132671.75	2905179.77
Gate count	87054.95	1911302.48
RAM size	78.39KB	65.53KB
Run time	0.0135s	0.246s

FIGURE 2. The result of logic utilization in the design compiler.

Also, the same structure was implemented with Python, not a hardware language. CNN trained with MNIST took 96.74 seconds and 69.58MB of memory. The structure composed of two FC layers used a total of 6.31 seconds and 607.83984 KB of memory. Memory usage of more than MB and time of more than 1 minute is not efficient to implement due to lightweight hardware. Recently, lightweight hardware deep learning has been studied a lot, but simple and traditional calculations are more efficient due to such a complex structure. To maximize hardware benefits, conventional canny edge detection and hough transform methods were utilized rather than relying on AI-based approaches.

III. BACKGROUND

In this section, information and definitions to facilitate a better understanding of the research is provided. First, a brief introduction to previous related studies of lane recognition and the characteristics of image processing will be described.

A. LANE DETECTION

Improving road safety through advanced computing and sensor technology has attracted much attention from the automotive industry. Therefore, Advanced Driver Assistance Systems (ADAS) are gaining huge popularity as they help drivers properly manage different driving conditions and

provide warnings if hazards have insight. Standard driver assistance systems include lane departure warnings, traffic sign detection [13], obstacle detection [14], and pedestrian detection [15]. Lane Departure Warning (LDW) and Forward Collision Warning (FCW) are designed to prevent accidents caused by not seeing a car pull into the next lane or being too close to the car in front. These systems are based on a lane detection algorithm.

Currently, with the development of deep learning, there are various ways to recognize lanes. After the CNN-based structure was created, a method to extract lanes using CNN was developed [16]. Additionally, a technique has been established for enhancing data processing effectiveness in the image capture and processing procedure, utilizing an edge cloud computing supplied distributed computing structure [17]. Lane recognition model that directly estimates the lane location using CNN and camera images was created [18].

Algorithms to detect lanes using image segmentation have also been created. Mask-RCNN [19] was trained for lane detection and the Kalman filter was used for lane tracking [20]. In addition, detecting traffic lanes by implementing segmentation as a filter through color is possible [21]. First, a region of interest is selected to find a threshold using a statistical method. A threshold is then used to distinguish possible lane boundaries. It uses color-based segmentation to find lane boundaries and approaches them using a quadratic function.

B. EDGE DETECTION

An edge in an image means a part in the image where the brightness value of a pixel changes rapidly. Generally, it refers to the boundary between a background and an object or between an object and an object. An edge indicates the boundary of an object in an image and contains various information such as shape and direction detection. Edge detection is the process of finding a pixel corresponding to an edge.

The detection of the rate of contrast and brightness change, or slope, is crucial. This can be accomplished using the first derivative, as the digital image operates on the difference between neighboring pixels. The Roberts operator [22] is the oldest edge detector and utilizes the two masks below to locate an edge using an approximate first derivative value. The Roberts mask has a faster computational speed than the Sobel/Prewitt mask and reliably extracts edges. However, it produces thinner edges and is more sensitive to noise than the Sobel/Prewitt mask.

Second, there is the Prewitt operator [23]. Its filter simply uses the average value of the pixel. The convolution result with the Prewit mask is like the Sobel mask, and the response time is relatively fast. However, compared to the Sobel mask, the edge is less prominent because the weight is slightly less for the change in brightness. Also, it responds more sensitively to vertical and horizontal edges than diagonal edges.

Log (Laplacian of Gaussian) [24] uses a second derivative algorithm and as it contains the formula for the derivative twice. The point at which the second derivative judges the edge is provided, and delicate edge detection is possible. All directions of an edge can also be detected. Since the Laplacian mask extracts edges using the difference value from the surrounding brightness, it is weak against noise and reacts more strongly to thin lines or isolated points in the image than to edges.

IV. IMPLEMENTATION

This section provides certain information and definitions to facilitate to better understanding of our research. First, the operation of the image process of the edge detection algorithm and lane recognition will be explained, followed by parallel processing of the algorithm.

A. CANNY EDGE DETECTION

Canny Edge Detection is currently the most basic and widely used method [25]. Canny Edge Detection has an advantage in that the position of the edge point is accurately measured with a lower error rate than other algorithms. Also, with a single edge point response, the detector returns only one point for each edge point. Figure 3 shows the algorithm of Canny Edge Detection. It performs in 5 stages, grayscale conversion, Gaussian smoothing, Sobel operator, non-maximum suppression, and hysteresis. Figure 4 each stage's original and the result images. The following sections describe each stage of the algorithm.

1) GRAYSCALE CONVERSION

To obtain a binary image of edges, it is necessary to convert RGB-based input image data to grayscale. This process transforms the input 24-bit RGB pixel data, with eight bits to each for red, green, and blue, to an eight-bit grayscale image. Each pixel's grayscale value is computed as the average of the original RGB-based image's three eight-bit color values. Figure 4 (b) illustrates the grayscale conversion of the original image (a).

2) GAUSSIAN SMOOTHING

The noise of the image can make it challenging to find the edge properly. In this stage, a 5×5 Gaussian filter is used to remove noise. As the input image is two-dimensional, the formula (1) isotropic Gaussian is used. Applying the Gaussian function using convolution can be used as a point-spread method. As the pixel values of an image are discrete, the Gaussian function must also be approximated discretely. The operation of the convolution kernel, which approximates the Gaussian function with standard deviation $\sigma = 1.0$ and mean $\mu = 0$ is shown as a formula (2). The $*$ operator indicates convolution, A is the original image data, and B is the resulting filtered image. Each dot-product result is divided by 273, which is the sum of all the values in the filter so that the intensity of the image does not change. Figure 4 (c) shows the result of the Gaussian

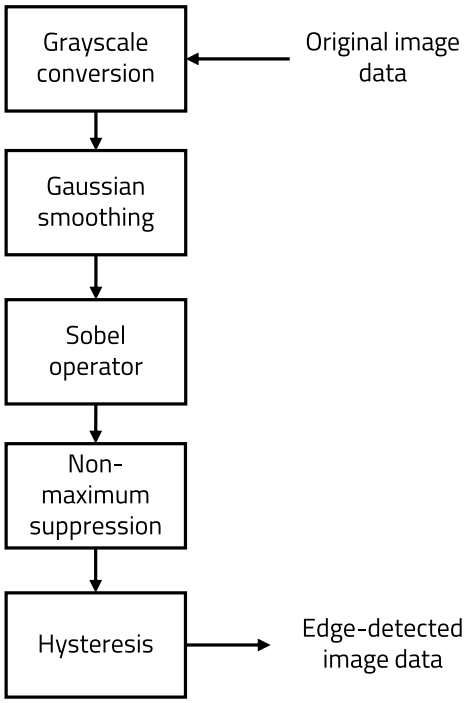


FIGURE 3. The Canny edge detection algorithm.

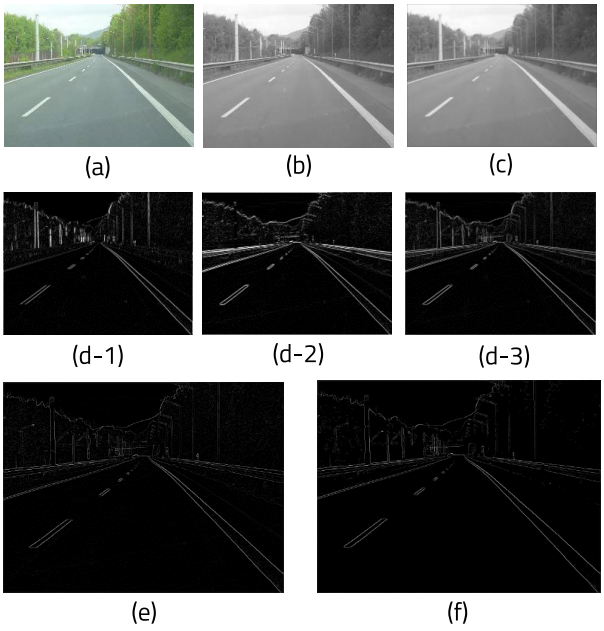


FIGURE 4. The resulting image of each stage of canny edge detection (a) Original image (b) Grayscale conversion result (c) Gaussian smoothing result (d-1) Sobel operator in x-axis (d-2) Sobel operator in y-axis (d-3) Sobel operator result (e) Non-maximum suppression result (f) Hysteresis result.

smoothing image.

$$G(x|\mu,\sigma^2) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \tag{1}$$

$$B = \frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix} * A \tag{2}$$

3) SOBEL OPERATOR

If the noise has been removed through filtering, the edge in the image could be found. An edge in an image is a part where the image intensity changes rapidly, so a differentiated operation is needed to detect it. In this stage, changes in image intensity along the horizontal and vertical directions are detected using the Sobel operator. If B, which has passed through the Gaussian filter earlier, is regarded as $f(x, y)$, the differential expression of the function should be calculated. For calculating the horizontal direction of the image, the partial derivative $\frac{\partial f}{\partial x}$ should be found. Identically, the changes in the vertical direction can be deduced with the partial derivative $\frac{\partial f}{\partial y}$.

Based on the calculation, the Sobel stage uses two 3×3 kernels for convolution. The kernels and convolution operations are shown in formula (3). These kernels represent the derivative filter used in a two-dimensional surface. The derivative depends on three rows for each pixel, where two coefficients weight pixels in the same row. Similarly, $(\partial f/\partial y)_{i,j}$ for each pixel can be found. Figure 4 (d-1) is the result of the Sobel operator in the x-axis, and (d-2) is the result of the y-axis filter. Figure 4 (d-3) is the result of the total Sobel filter image; the x and y results are combined.

$$\begin{aligned} \frac{\partial f}{\partial x} &= \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * B \\ \frac{\partial f}{\partial y} &= \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * B \end{aligned} \tag{3}$$

If this outcome is presented as a vector, the partial derivatives obtained in the x and y directions can be interpreted as the gradient of intensity. Formula (4) expresses the vector form of the partial derivatives. By calculating the norm, the magnitude of the vector can determine if the pixel is on the edge, while the angle can be obtained to determine the direction of the edge. Formula (5) calculates the norm of the intensity gradient using the Pythagorean theorem. The vector's angle can be found using the formula (6). The angle θ reflects the direction of maximum elevation at the pixel.

$$\nabla f = [\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}] \tag{4}$$

$$|G| = \sqrt{{G_x}^2 + {G_y}^2} \quad (G_x = \frac{\partial f}{\partial x}, G_y = \frac{\partial f}{\partial y}) \tag{5}$$

$$\theta = tan^{-1}(\frac{\partial f/\partial y}{\partial f/\partial x}) = tan^{-1}(\frac{|G_y|}{|G_x|}) \tag{6}$$

4) NON-MAXIMUM SUPPRESSION

It is essential to confirm that the edges extracted in the previous step are genuine edges and do not overlap. In this

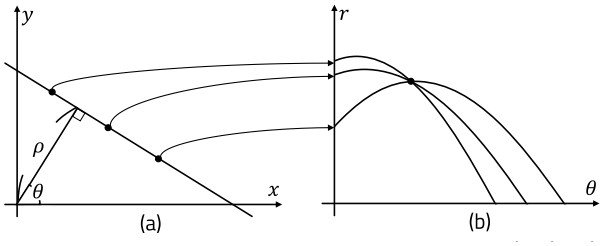


FIGURE 5. Expression of image space and hough space. (a) image space (b) hough space.

phase, only pixels with a local maximum gradient size are designated as edge pixels to remove the occurrence of several pixels signifying one edge. Hence, all values except the local maxima are eliminated to restore unclear edges. It analyzes each image pixel on the edge and contrasts the pixel with its neighboring pixels that might belong to the same edge. If a pixel is brighter than its neighboring pixels, its intensity is maintained. Otherwise, it is set to zero. Figure 4 (e) is the result of non maximum suppression of the image.

5) HYSTERESIS

Weak edges are eliminated by removing pixels that fail to meet predetermined thresholds. A hysteresis algorithm examines each pixel to determine if it surpasses the threshold. This approach enhances the present problem by exploiting the reality that edge detection conditions can vary depending on the environment. Figure 4 (f) depicts the outcome of hysteresis and complete Canny Edge Detection.

B. LANE EXTRACTION

Previously, the image edges were extracted and straight lines were extracted using the Hough transform, a mathematical shape-searching technique. Hough transform is a popular algorithm in various libraries due to its linear processing and fast real-time algorithm. Lane extraction follows a series of processes after straight line extraction.

1) HOUGH TRANSFORM

Hesse's normal form is used for expressing the lines. This form uses two parameters, (ρ, θ). The parameter ρ in the distance from the origin to the closest point on the line and θ is the angle between the x-axis and the line connecting the origin to the closest point of the line. Equation (7) is used to map pixels in the (x, y) coordinate space to the (r, θ) parameter space.

$$xcos\theta + ysin\theta = r \tag{7}$$

Figure 5 (a) explains the image space line, while (b) displays the same line in the Hough space. Each line that passes through a point is given by a single sin curve in the ρ and θ planes. The intersection of these sin curves implies the existence of the same straight line passing through the two points. Thus, if n sin curves converge at a point, n points lie on a straight line.

Algorithm 1 Local Maxima Algorithm

```
Input: Accum[theta][rho]
Output: Local maximum value of matrix Accum
Function Local_Maximum(Accum[theta][rho]):
    middle_row ← Accum.theta/2
    middle_column ← Accum.rho/2
    max_cell ← Max(Max_in_Row(middle_row), Max_in_Column(middle_column))
    next_cell ← Next_Cell(max_cell)
    if next_cell = max_cell then
        return max_cell
    else
        if next_cell.row > middle_row then
            if next_cell.column > middle_column then
                return Local_Maximum(Top − LeftSub − Matrix)
            else
                return Local_Maximum(Top − RightSub − Matrix)
        else
            if next_cell.column > middle_column then
                return Local_Maximum(Bottom − LeftSub − Matrix)
            else
                return Local_Maximum(Bottom − RightSub − Matrix)
```

The straight line expressed in this way is stored in a two-dimensional array, accumulator, consisting of *rho* and *theta*. Each element is the number of edge pixels corresponding to the straight line $(rho, theta)$ in the input image. The most prominent line in the input image can be inferred by finding the straight line with the highest value in the accumulator. Each edge pixel encountered by the algorithm contributes $+1$ to the accumulator value of every line on which that pixel lies.

2) LOCAL MAXIMA OF THE LANE

The line is extracted from the local maximum of the accumulator instead of the global maximum to eliminate false redundancy. Only the highest accumulator value is considered when several high accumulator values are nearby, and adjacent values are removed. However, there is a risk of removing lines that are close together in the input image, but for lane detection, it can be assumed that the lane markings are far enough apart that this is not a problem.

Algorithm 1 outlines this process. The input is an accum matrix derived through the Hough transform. First, the cell value of *middle_row* and *middle_column* are declared.

FIGURE 6. The result images of hough transform and lane extraction.

Algorithm 2 Part 1 - Main Function of the Algorithm

Input: Image data
Output: Edge Data
Function `Gray_scale` (*Image Pixel*) **:**
  foreach $w \leq Image\_Width$ **do**
    foreach $h < Image\_Height$ **do**
      $Pixel[gray] = (Pixel[r] + Pixel[g] + Pixel[b])/3$
  return *Pixel[gray]*
Function `Parallel_process` (*Image Pixel*) **:**
  $T \leftarrow Number\ of\ Threads$
  $H \leftarrow Image\ height/T$
  $W \leftarrow Image\ width$
  $Gray \leftarrow Gray\_scale(Pixel[r][g][b])$
  $P_{h_i} \leftarrow Gray[0 : Image\_Width][T_{i-1} * H : T_i * H]$
  On each Thread T_i with slices of pixels $P_{h_0}, \ldots, P_{h_T}$ **do in parallel**
    $Edge\_detection[P_{h_i}]$
return *Edge data*

After that, a maximum cell representing the cell with the most significant value among all the cells in the middle rows is stated. To do this, the functions *Max_in_Rows* and *Max_in_column*, which return the maximum value in a particular row or column, are used. The following cell of the maximum cell is set to find out which submatrix local maximum is located. For the *next_cell*, if the row value is greater than the *middle_row* value, it means the local maximum is either in the upper left or upper right submatrix. If the *next_cell* column is greater than the *middle_column*, it means that the top-left sub-matrix has a local maximum. Otherwise, it is displayed in the top-right sub-matrix. Finally, if the *next_cell* exists, and its row is greater than or equal to the *middle_row*, it means that the local maximum is in the bottom-left or bottom-right sub-matrix.

If the matrix *accum* has a length of N, it must iterate over 2N cells to derive *min_cell* from the first cell. In the second call, it must iterate over $2N/N$ cells. As this process is repeated, it will decrease by 1/2. Therefore the total iteration will be $\sum_{N}^{i=1} \frac{2N}{i} \approx 4N$, and the complexity is $O(N)$.

3) OPTIMIZATION

In addition, it goes through a double optimization process to achieve better results. The first optimization aims to avoid detecting lines of objects far from the horizon (e.g. objects detected from hills or clouds). When implementing the Hough transform, only the bottom half of the edge detection image will be passed when incrementing the accumulator, for removing the horizontal lines. This optimization will reduce the runtime of the Hough transform by skipping more than half of the input pixels and, as well as avoiding potentially false lines.

A second optimization eliminates horizontal lines being extracted from the accumulator. This optimization is based on the knowledge that lane boundaries tend to point outward from the camera's point of view, meaning that vertical and sloping lines are likely to correspond to lane lines. This can be implemented in the line extraction step by ignoring lines in the regenerator where θ represents a horizontal line ($80 < \theta < 100$). Figure 6 shows the result of lane detection using the Hough transform. Figure 6 shows the Hough transforms and lane extraction result images. The image's lanes overlapped in 2 red lines on the original images.

C. PARALLEL PROGRAMMING

As explained earlier, this image-processing algorithm requires many program loops. A parallel process was used in Canny Edge Detection because the Hough transform algorithm needed to be performed on the entire image data. Figure 7 describes a more detailed explanation of parallelism. First the image data was copied into memory as the height n and width m of the image. After that, converting the image to grayscale is sequentially processed with all the image data. The grayscaled image data is divided according to the number of threads, t, for a Gaussian smoothing operation. Since Gaussian smoothing uses a 5×5 filter, each thread is divided into rows by five or more pixels. Currently, if the number of threads is t, the number of divided pixels will be n/t.

After applying a Gaussian filter to remove noise, the image data from each thread is segmented into columns of three pixels, corresponding to the 3×3 Sobel filter. The sub-threads

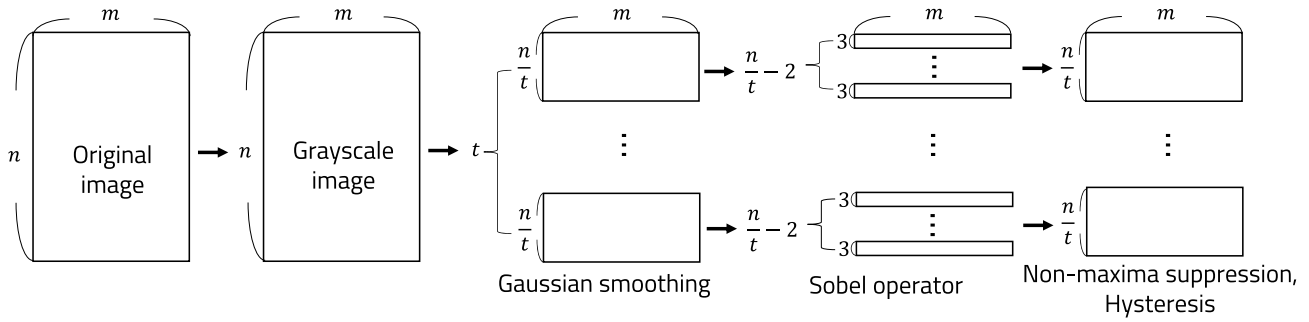


FIGURE 7. Structure of parallel processing.

operate in parallel with overlapping boundary data, resulting in a total of $n/t - 2$ sub-threads. Once sub-parallelization is complete, the image data passes through the Sobel filter again before undergoing non-maximal suppression and hysteresis. Non-maximum suppression and hysteresis are executed sequentially within a thread. The lane is extracted from the image data, which is used to extract the edges through the transform method.

Algorithm 2 and 3 show the pseudo algorithm of the throughout process. To make it easier to understand, the functions that are parallelized and the functions that are not are divided into two parts. First, part1 2, which describes the main function that is not parallelized, receives image pixels as input. The number of threads is predetermined by the user, which is T. By T, the image height can be divided to T, which is H. Parallelization proceeds through the height of the image, so the width does not change. Grayscale conversion is applied to every pixel without parallelization. A function called *gray_scale* is called, and this function derives the average of the r, g, b values of each pixel with a nested loop. Edges of pixels are detected using parallelization. At this time, a function called edge detection is processed in parallel according to the divided threads.

Algorithm 3 describes edge detection functions. First gaussian smoothing is performed by convoluting gaussian filter and pixels. After applying the Gaussian filter, the sobel filter should be applied. Since the sobel filter is 3×3, parallelization can be performed using threads at this time as well. A greater effect can be obtained by parallelizing with double threading. Partial derivatives and image gradients can be computed through sobel filter. Using these, it can calculate the angle and direction of the edges and find the strongest edge by comparing it to the surrounding pixels. At this time, the intensity of the edge can be adjusted according to the angle previously defined by the user, such as 45, 90, or 180 degrees. In addition, by using a pre-set threshold, pixel values below this threshold are set to 0 to detect clear lines.

Like this, there is no process for image segmentation by implementing it in a hardware language using Canny edge detection with parallel computation. Additionally, if it is lightweight and parallelized, it can be utilized on multiple FPGA boards.

V. EXPERIMENTAL SETUP AND EVALUATION

This section describes experiments of our parallel lane detection algorithm program and evaluates the performance with test images in C language and VHDL (VHSIC Hardware Description Language) on the FPGA(Field Programmable Gate Array) board.

A. EXPERIMENTAL ENVIRONMENT

In this study, parallel lane detection algorithms are conducted on the DE1-SoC board and simulated in Windows 10 PC. DE1-SoC board is a processor made by Intel company. DE1-SoC is divided into FPGA and HPS (Hard Processor System) blocks. The FPGA block is equipped with Cyclone V SoC 5CSEMA5F31C6 device and 64MB SDRAM (16-bit data bus). The HPS block has an 800MHz Dual-core ARM Cortex-A9 MPCore processor with 1GB DDR3 (Double Data Rate) SDRAM (Synchronous Dynamic Random Access Memory) (32-bit data bus). It can be managed by input/output and communication through Ethernet, usb ports and serial ports. Also as an Intel board, it was programmed using Quartus Prime 18.1 and simulated in Modelsim, DC (Design Compiler).

Images were manually annotated for lane detection and compared with the algorithm described above. Each of the 30 images displays the lanes from the viewpoint of the inside of the car. The results of the lane detection using HPS and FPGA are presented in Figure 6. To evaluate this algorithm, we first manually marked the lanes contained in 30 images by humans. The results of human evaluation were compared with the results output by this algorithm. To ensure accuracy, it was confirmed that the coordinates output by both HPS and FPGA were within the range set by humans.

The experiments proceeded after verifying that the coordinates of 30 images fell within the 5% error range set. Section D.Experimental Results: Time Measurement and section F.Experimental Results: Power Consumption were based on the analysis of these 30 images.

B. C LANGUAGE PERFORMED IN HPS BLOCK

The program was programmed using C language in both floating-point and fixed-point data, utilizing the HPS segment of the DE1-SoC board. For an exact comparison between C

Algorithm 3 Part 2 - Parallel Canny Edge Detection

Input: Gray pixel data
Output: Edge Data

```
Function Edge_detection(gray):
    /* Gaussian smoothing */
    foreach y < 5 do
        foreach x < 5 do
            Pix_gau[x][y] =
              Pix[x][y] * gaussian_filter[x][y]
    /* Sobel filter */
    S_T ← H − 2
    On each Thread S_T do in parallel
        foreach y < 3 do
            foreach x < 3 do
                Pix_sob_X[x][y] =
                  Pix_gau[x][y] * sobel_filter_X[x][y]
                Pix_sob_Y[x][y] =
                  Pix_gau[x][y] * sobel_filter_Y[x][y]
    /*Non-maxima suppression*/
    foreach y < P_W do
        foreach x < P_H do
            theta = atan((Pix_sob_X[x][y] +
              Pix_sob_Y[x][y])/2)
            if theta == −45, 45 then
                if Pix_sob[x][y] < Pix_sob[x][y − 1]
                  then
                    Pix_nonMax[x][y] = 0
                if Pix_sob[x][y] < Pix_sob[x][y + 1]
                  then
                    Pix_nonMax[x][y] = 0
            Continue iterating at a pre-determined
              theta value
    /* Hysteresis */
    foreach y < P_W do
        foreach x < P_H do
            if Pix_nonMax[x][y] < Th then
                Pix_Hys[x][y] = 0
```

and hardware languages, the conversion of data to floating point was also implemented in C. Figure 8 illustrates the HPS development design flow, which begins by writing the aforementioned program in C and compiling it. Then, the executable file is launched on a bootable Linux card on the board. Because global memory access is slow, it is necessary to cache pixels in local memory. We implemented two forms of parallel programming: POSIX (Portable Operating System Interface for Unix) threads [27] and OpenMP (Open Multi-Processing) [28].

In addition, the resulting image obtained using HPS is monitored using FPGA bridges. Figure 9 shows output

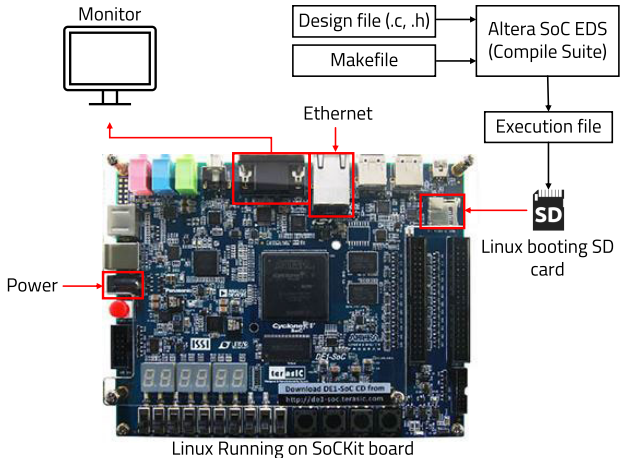


FIGURE 8. Design flow of HPS block [26].

display of the images. To look at the resulting images of each step, the original image, edge detection image, and lane image were displayed. After each stage of operation is completed, the image data is stored in the SDRAM buffer. The data stored in this buffer is transferred to the video DMA (Direct Memory Access) using a DMA controller. By using the board, each result can be immediately shown at runtime. Real-time lane recognition is also possible.

The display screen can be viewed by directly connecting with DMA but using a standard library, it can output the image using a file pointer since it is written in C code. Comparison becomes easy as the coordinates used when drawing the line can be output. The resulting image files can be shown in figure 6.

The POSIX library known as PThread (POSIX Thread) is a low-level implementation and OpenMP is a high-level implementation. Both PThread and OpenMP can be seen as standards for shared memory techniques, but PThread focuses on task parallelism, and execution code must be written in detail to fit the PThread type. OpenMP considers both task parallelism and data parallelism, and it has the difference that it can be easily converted into code using parallel programming without making significant modifications to the program.

C. VHDL PERFORMED IN FPGA BLOCK

To implement the algorithm in the FPGA block of the board, VHDL language is used. As the hardware language and FPGA of the board cannot embrace the jpg format in one place, the input image must be converted to a text-based file. Figure 10 illustrates the design flow of the algorithm implemented in the FPGA block. As it must be executed in the intel-based board, the input image data is converted into MIF (Memory initialization file) file format. MIF is an ASCII (American Standard Code for Information Interchange) text file that specifies the content of the memory block such as RAM and ROM (Read Only Memory). MATLAB is used to convert jpg and bmp files to mif file format.

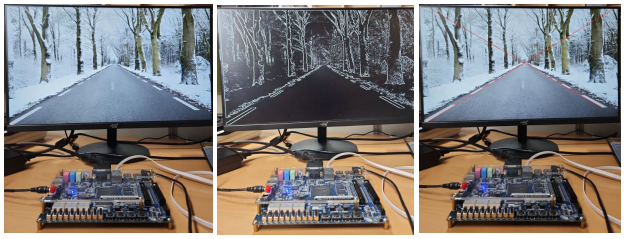


FIGURE 9. Result of HPS output.

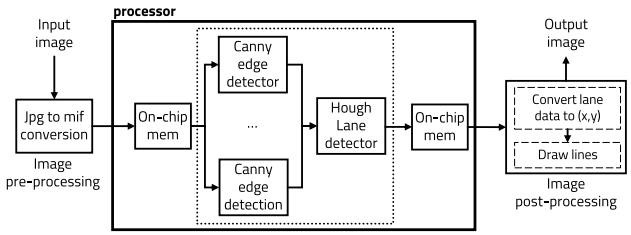


FIGURE 10. Algorithm of lane detection in FPGA block.

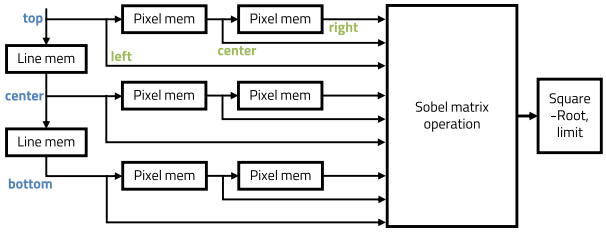


FIGURE 11. The block diagram of sobel operation in FPGA block.

The above mathematical operations require transformation because they can only be implemented as a hardware language in fixed-points. First, the floating-point operation of grayscale conversion can be expressed as $Y = 0.299 \times R + 0.587 \times G + 0.114 \times B$. The equation can be approximated for a fixed-point design using equation 8. A factor of 16 to the floating-point argument gives the factor it takes for a fixed-point implementation. Therefore, the fixed-point equation can be approximated as $Y = 5R + 9G + 2B$. One pixel of an RGB-based image consists of 24 bits, 8 bits each of RGB. After converting 24 bits per pixel to grayscale, it was reduced to 12 bits.

$$\begin{aligned} 0.299 \times 16 &= 4.78 \approx 5 \\ 0.587 \times 16 &= 9.39 \approx 9 \\ 0.114 \times 16 &= 4.78 \approx 2 \end{aligned} \tag{8}$$

Figure 11 shows the block diagram of the Sobel filter operation in the FPGA block. To operate a 3×3 filter, 3 line memories for top, middle, and bottom are required. Each line of memory stores the row pixel data of the image data divided by threads. Then the pixel memories store the left, center, and right pixels of each line memory. In this way, nine memories are created, and the Sobel operation is performed on these

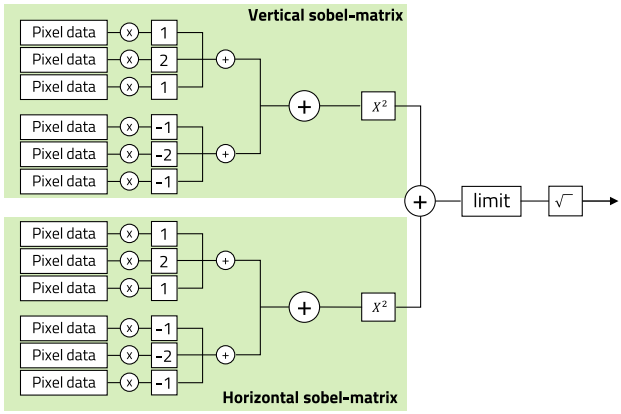


FIGURE 12. The block diagram of sobel filter operation arithmetic calculation.

memories. With nine data stores, two arithmetic units are used, horizontal and vertical filtering. After the operation, the results are combined and square-rooted.

The Sobel filter mentioned above is composed of 1,2,1,−1,−2 and −1. The vertical Sobel matrix calculation process and overall calculation are expressed in figure 12. The median value was multiplied by 2 to express the elements of 2 and −2. The adder adds the six values; the top three are positive values, and the bottom three are negative. Then the result of the filter matrix (G_x) is squared. The results of vertical and horizontal filter operations are summed and square rooted. As the image data has a limit of 0 to 255, the result value must be divided by two and subtracted from 255.

After each thread's edge detection operation, the resultant pixels are collected, and the Hough transform is performed. However, trigonometric operations and multipliers are required for the Hough transform, which is problematic from a hardware point of view. The CORDIC (COordinate Rotation DIgital Computer) algorithm [29] is used to apply for hardware operation.

If a vector V in xy coordinates has elements of x and y and has an angle of ϕ, it can be expressed as: $V' = \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} xcos(\phi) - ysin(\phi) \\ xsin(\phi) + ycos(\phi) \end{bmatrix}$. These equations can be rearranged by using trigonometric functions: $x' = cos(\phi)(x - ytan(\phi))$, $y' = cos(\phi)(xtan(\phi) + y)$. The multiplication term of tangent can be approximated using $\phi_0 = atan2^{-i} \approx 2^{-i}$, $(i > 3)$, where i is the iteration index. This iteration of the operation can be expressed as equation 9. $K_i = cos(\phi)$ is the gain and d_i is the direction of the angle.

$$\begin{aligned} x_{i+1} &= K_i[x_i - y_i \cdot d_i \cdot 2^{-i}] \\ y_{i+1} &= K_i[x_i + y_i \cdot d_i \cdot 2^{-i}] \end{aligned} \tag{9}$$

This equation can be applied to the Hough transformation equation 7. It can be expressed as: $R_{1x} = (x + ytan(\phi))cos(\phi)$, $0 < \phi < \frac{\pi}{2}$, $\pi < \phi < \frac{3\pi}{2})R_{1y} = (y - xtan(\phi))cos(\phi)$, $\frac{\pi}{2} < \phi + \frac{\pi}{2} < \frac{3\pi}{2}$, $\frac{3\pi}{2} < \phi + \frac{\pi}{2} < 2\pi)$.

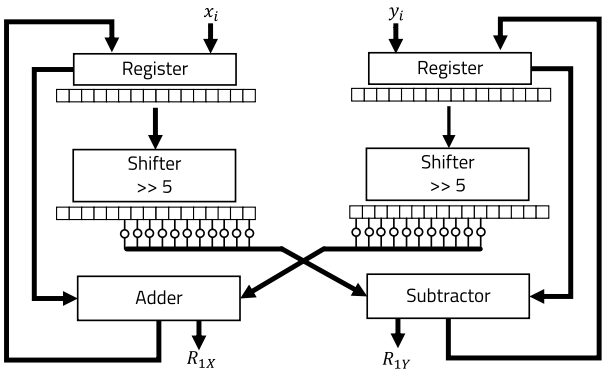


FIGURE 13. The block diagram of hough cordic operation register calculation.

By iterating ϕ by units and changing it from 0 to π, all lines can be extracted from the (x, y) coordinates. Since R at a given pixel location at a constant angle is needed to be calculated, the direction is constant and a constant angle increments the angle.

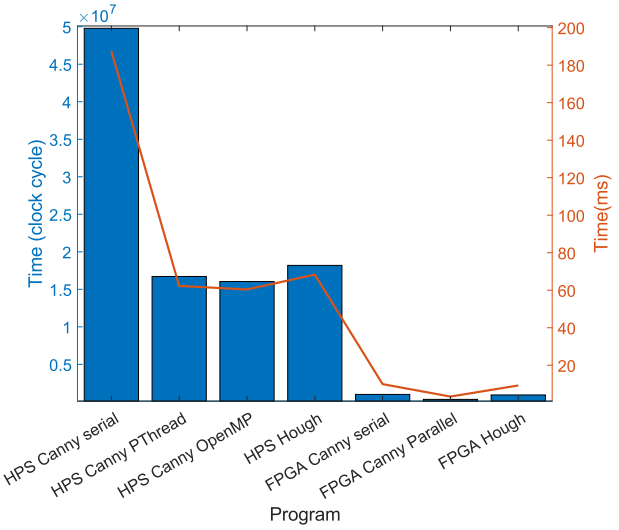
Figure 13 shows the block diagram of the CORDIC Hough transform. When step iteration of $\phi = 1.79°$, $tan(\phi) = 0.031 = 2^{-5}$, therefore 5 bits should be shifted. The output will be R_{1x} and R_{1y}. Starting to $\phi = 0$, ϕ will iterate in a unit of 1.79 each time to π. By calculating each ϕ, R will be calculated and fetched to the accumulator array. A counter and single-state machine were added with the Hough transform block to make it repeatable.

As described above, the data of the two lanes can be extracted from the converted values through the local maxima. In the local maxima algorithm, the accumulator array used single RAM same as canny edge detection. Image post-processing in VHDL is complex and hardware disadvantageous, so this task was executed with MATLAB. After converting the two extracted lane data from the Hough plane to the image plane, two lines were drawn on the original image.

This process was also tested using 30 image files. In the process of implementation in the FPGA block, the pre-processing process of converting the image file to mif and the post-processing process of drawing the resulting coordinates on the image are parts that must be done manually by humans. Because there are parts that must be done manually, it is impossible to compare with massive datasets such as nuScene [30] and CULane [31], which are widely used datasets in ADAS. When the image processing is done manually in this way, the resulting image is output with a line drawn on the image, as shown in Figure 6.

D. EXPERIMENTAL RESULTS: TIME MEASUREMENT

Figure 14 (a) displays the time-measured algorithm results for serial and parallel processing on the board. The program utilized VHDL and C languages with three threads and



(a)

VHDL program (FPGA)		Canny edge detection	Hough transform
Worst-case slack	Setup slack	-2.813	-2.573
	Hold slack	0.151	0.665

(b)

FIGURE 14. (a) Time measurement of lane detection algorithm (b) Time slack measurement of FPGA block.

applied a 1028 × 720 size test image. For the identical experimental conditions, both the HPS block and FPGA block were calculated using fixed-point arithmetic. When executed on the HPS block of the board, the C language program recorded 255.83 ms, 67920671 cycles for the entire process, and 187.53 ms, 49786813 cycles for edge detection. In contrast, edge detection using POSIX thread processing required only 131.52 ms, 34916820 cycles and 62.95 ms, 16712392 cycles. For OpenMP, the detection of edges took 60.41 ms and 16,038,055 cycles. Parallel processing can significantly decrease processing time by two-thirds.

Programmed in FPGA block of the board using VHDL, cost 993618 clock cycle in serial processing of edge detection and 922599 clock cycle in Hough transform. The cycle is 10ns per clock, so it can be converted to 9.93 ms and 9.22 ms. The exact structure of the Hough transform has been used. Therefore the clock cycle took the same in serial and parallel processing. In parallel programming, edge detection took 331217 cycles, which is 3.31 ms. The clock cycle has decreased by one-third. However, the proportion of the time required for the Hough transform designed in VHDL is quite large, so the total time differs by 1 ms. Also the worst-case time slacks were measured and expressed in figure 14 (b). In Canny Edge Detection, the worst-case setup slack took −2.813, and hold slack took 0.151. Hough transform took worst-case setup slack as −2.573, and hold slack took 0.665.

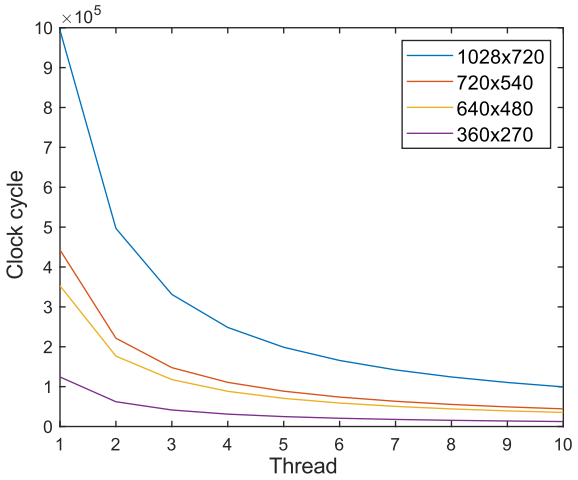


FIGURE 15. Time measurement of lane detection algorithm.

The result of the performance between HPS and FPGA implementation of the board can be different as the algorithms perform the same operation on the same board. When using the HPS block, about 13 times more time and 36 times more clock cycles are used than when using the FPGA block. Using an HPS block can take a lot of time and cycle because the clock speed of the board is 265.4MHz, which is slow. Also the C code of the algorithm consists of a lot of for loops, and if-conditional code, which requires more cycles. For easy comparison, comparing two blocks at the same clock frequency, 100MHz: Canny Edge Detection in HPS block with serial process will take 497.86ms, PThread will take 167.12ms, OpenMP will take 160.38ms. Hough transform will take 181.32ms. These are more than 50 times larger than the FPGA clock cycle.

The correlation between the number of threads and the clock cycle according to the resolution of the image is graphed in figure 15. At a fixed resolution, clock cycles according to threads progressed in inverse proportion. However, the change in clock cycle according to the resolution was not proportional. Therefore if the clock cycle is c_1 for thread 1, which is serial processing, the expression is $y = c_1\frac{1}{x}$, y is clock cycle and x is a number of threads. If the image resolution cannot be reduced, it is derived that it is better to use threads five or higher. However, if the resolution of the image is low and the clock cycle is low enough, it can be optimized with a small number of threads considering the trade-off relationship.

E. EXPERIMENTAL RESULTS: AREA, MEMORY

Figure 16 shows the serial lane parallel processing shows memory measurements. The total amount of virtual memory on the board allocated to the C program in HPS was measured. The sequential program used 3712 KB of memory and each parallelized program used 12276 KB and 10835 KB of memory. Parallelized programs use twice as much memory as serial programs. We concluded that OpenMP takes less time than the POSIX thread. Because in the lightweight process model, repetitive tasks are not complex, and do not require massive memory.

	Program	Serial processing	Parallel processing	
			Thread	OpenMP
C program (HPS)	Edge detection	3216 KB	11784 KB	10336 KB
	Hough transform	496 KB	492 KB	499 KB
	Total time	3712 KB	12276 KB	10835 KB

FIGURE 16. Memory measurement of lane detection algorithm in HPS block.

Area	Canny edge detection	Hough transform
Combinational area	295.569	453.392
Non-combinational area	257.193	333.436
Design area	630.338	898.308
Gate count	413.607	589.440
RAM size	7.68KB	45KB

FIGURE 17. The result of logic utilization in design compiler.

The programs were simulated using the Design compiler and Quartus. Figure 17 shows the result of Design Compiler simulation. The design compiler was compiled using the SAED32 library to check the approximate area and gate count before uploading to the board. RAM area was excluded because the area could be different for each process, therefore only the logic gate was measured. Canny Edge Detection took a total of 630.338 design area and Hough transform took up 898.308 design areas. To count the number of gates used in the program, the size of the NAND gate was measured, which is the smallest gate used. The gate counts can be measured by dividing the total design area into NAND gate area, which is 1.524. Therefore total gate count of Canny Edge Detection was 413.607, and Hough Lane Detection was 589.440. RAM size was measured by the input bytes and the depth of the memory. In canny edge detection, two single port RAMs have been used and they have 3bytes by 1028 depth, which can be calculated as 7.68KB. Hough Lane Detection algorithm used a single port RAM that has 1 byte by 45000 depth, 45KB.

By employing the parallelization method outlined above, the RAM utilized in each thread can be reduced, resulting in shorter processing times. However, parallelization can increase the access time and pins required for the divided RAM, potentially leading to a larger interface footprint with the controller. Although serial interfaces are inherently simpler and smaller than parallel interfaces, they are significantly slower in terms of throughput.

It can be seen that the time gain obtained by parallelizing many matrix operations executed during image processing far exceeds these overheads. Also as the size of RAM can grow unexpectedly depending on the resolution of the image,

Program			Canny edge detection	Hough transform	Total
C program (HPS)	Serial processing		8.8 mW / 1.650mJ	2.4 mW / 0.163mJ	11.2 mW / 1.813 mJ
	Parallel processing	Thread	9.6 mW / 0.604mJ	2.3 mW / 0.157mJ	11.9 mW / 0.761mJ
		OpenMP	9.8 mW / 0.592mJ	2.3 mW / 0.159mJ	12.1 mW / 0.751mJ
VHDL program (FPGA)			2.8 mW /0.009mJ	3.02 mW / 0.027mJ	5.82 mW / 0.036mJ

FIGURE 18. Power consumption of lane detection algorithm.

parallelization can be the most efficient method for low execution time. Therefore, it can be demonstrated that this algorithm uses a small area of the board and puts a small load on the board even with deep parallelization.

F. EXPERIMENTAL RESULTS: POWER CONSUMPTION

Figure 18 illustrates the power consumption of each algorithm. The algorithm's power in the HPS block using C language has been measured by DMM(Digital Multimeter). To measure current or voltage, the DMM was connected directly to the power line of the DE1-SoC by removing its insulating layer. The stripped power line's positive and negative voltage lines were connected to measure the voltage and current flow. Additionally, 30 lane images were utilized and averaged to determine power.

The initial power consumption during booting was measured and subtracted from the total power consumed, isolating the power utilized solely by the program. Additionally, energy consumption was measured for further comparison with the FPGA and HPS. Formula 10 can be utilized to calculate power, while formula 11 can be employed for energy calculations.

$$P = V \times I \tag{10}$$

$$J = W \times t \tag{11}$$

When programming the lane detection algorithm, Canny Edge Detection consumed 8.8mW and 1.65mJ when executed sequentially. When using Hough transform, 2.4mW and 0.163mJ were consumed. The total power consumed was 11.2mW and the energy was 1.813mJ. During the experiment, the number of parallel program threads for both HPS and FPGA was set to 3. A slight increase in power consumption was observed when executing the algorithm in parallel compared to series. When using PThread, the consumption was 9.6mW and 0.604mJ, and when using OpenMP, 9.8mW and 0.592mJ were consumed.

Furthermore, the HPS block consumes approximately 50 times more energy than the FPGA block. The HPS block incurs significant energy usage during the booting process, as well as the compilation and execution of the C program. Conversely, the FPGA block operates with lower energy consumption since its hardware and logic blocks are compiled and executed as intended. This is particularly advantageous in terms of time since the hardware can be tailored and employed based on the algorithm's requirements.

Also, the relationship between the area obtained from the board utilization experiment and the power consumption in FPGA block can be found. If the power consumption according to the area is calculated, it can be seen that canny edge detection uses 0.006mW per gate and hough transform uses 0.005mW. This means that a very small amount of power is used for each gate, and operation is possible with low power.

TABLE 1. Comparison with other studies.

	This work	Khongprason. [32]	Hajjouji [33]	Malmir [34]
Architecture	Parallelized edge detection	Overlapping pipeline	Inverse Hough transform	Dual stage landmark detection
Data format	Floating-point	Floating-point	Floating-point	Floating-point
Speed (s)	0.0033	0.0076	0.0147	0.04
Hardware Resources (LUT)	233	91,050	1996	38,220
Frame resolution	1028X720	480X270	640X480	1280x720
Platform	DE1-SoC	Zynq-7000 APSoC	Virtex-5	KC705 evaluation board

VI. COMPARISON WITH OTHER STUDIES

Although it is difficult to make an accurate comparison of the hardware resources of lane recognition algorithms, Table 1 shows comparison data with other studies. Three studies that did not use the AI method in lane recognition were compared with our study. Khongprasongsiri et al. [32] work uses hough transform and it used overlapping pipelines in GPU for better speed performance. Hajjouji et al. [33] uses inverse hough transform to reduce the dimension of the accumulator. Malmir and Shalchian [34] uses dual stage, ROI (Region of Interest) for hough transform and stripe detection stages.

By referring to the table, it can be said that our work has the fastest speed, frame per second, for lane detection. Also our work can cover large images with 1028 X 720 resolution. In addition, it can be seen that the use of hardware resources is overwhelmingly low. Of course, it is not accurate to compare using different hardware boards for each study, but it is about 390 times different from the largest used numerically.

VII. CONCLUSION

This paper introduces a lane detection algorithm using parallel processing on a lightweight embedded board using C language and VHDL. The HPS block of the board has been used for C language processing and the FPGA block has been used for VHDL. In the HPS block, Canny Edge Detection algorithm has been performed parallel using PThread and OpenMP. The algorithm converted to a fixed-point because hardware language is not appropriate

for operating in floating-points. CORDIC Hough transform is used for Hough transform in fixed point. Also, as pre-processing and post-processing of the image cannot be expressed in VHDL, MATLAB is used for result image processing.

Using parallel programming improves time efficiency significantly. In terms of run time, HPS takes 187.53 ms for serial, 62.95 ms for parallel, 9.93 ms for FPGA, and 3.31 for 3 threads, which takes 3 times less time. The clock cycle also took three times less. However, when using parallel processing on HPS, the memory increased from 3216KB to 11784KB. On the other hand, when used in FPGA, 52.68 KB of RAM was used, and it can be seen that less than 1% utilization of the total ALM logic of the board is used. By using parallelization to reduce the amount of RAM used by one thread, it was able to see the effect of reducing processing time. Power decreased from 8.8mW to 0.604mW in HPS and 2.8mW in FPGA. In terms of energy, HPS took 0.76mJ and FPGA took 0.036mJ.

When comparing the performance of HPS and FPGA, it took about 18 times less time at runtime from 187.53ms to 9.93ms. In terms of clock cycle, HPS uses 49786813 cycles and FPGA uses 993618 cycles, about 50 times less. Also, in power, HPS uses 8.8mW and FPGA uses 3 times less, and HPS uses 1.65mJ and FPGA uses 183 times less energy with 0.009mJ. There are tremendous benefits when running on an FPGA as the blocks on the board are organized more efficiently by processing them in parallel using a hardware language.

Throughout this study, it was found that the efficiency of lane detection can be increased by using parallel processing and the FPGA block of the board. However, if the FPGA block is used, there are advantages in time, memory, area, and energy. However it does have restrictions because it uses hardware language. Therefore, optimizing according to the situation is necessary by considering various trade-offs.

Implementing operations for parallel processing of Canny edge detection and Hough transform using a hardware language has resulted in several efficiencies. The objective of this study is to implement lightweight algorithms that can run on FPGAs. The research reveals the potential for executing this algorithm more efficiently via hardware language and even without an API.

In the future, we plan to perform real-time lane recognition based on parallel processing using curved Hough transform to improve the versatility and flexibility of different lane environments. In addition, we plan to approach this improved process using various APIs to design on a lighter board. Also automation of the image processing, which is currently done manually, is planned and it could be tested on a larger dataset.

ACKNOWLEDGMENT

The EDA tool was supported by the IC Design Education Center (IDEC), South Korea.

REFERENCES

[1] Y. Huang, Y. Li, X. Hu, and W. Ci, "Lane detection based on inverse perspective transformation and Kalman filter," *KSII Trans. Internet Inf. Syst.*, vol. 12, no. 2, pp. 643–661, 2018.

[2] E. A. Sekehravani, E. Babulak, and M. Masoodi, "Implementing Canny edge detection algorithm for noisy image," *Bull. Electr. Eng. Informat.*, vol. 9, no. 4, pp. 1404–1410, Aug. 2020.

[3] M. Marzougui, A. Alasiry, Y. Kortli, and J. Baili, "A lane tracking method based on progressive probabilistic Hough transform," *IEEE Access*, vol. 8, pp. 84893–84905, 2020.

[4] T. Alexoudi, G. T. Kanellos, and N. Pleros, "Optical RAM and integrated optical memories: A survey," *Light, Sci. Appl.*, vol. 9, no. 1, p. 91, May 2020.

[5] X. Zhou, N. Tomagou, Y. Ito, and K. Nakano, "Efficient Hough transform on the FPGA using DSP slices and block RAMs," in *Proc. IEEE Int. Symp. Parallel Distrib. Process., Workshops Phd Forum*, May 2013, pp. 771–778.

[6] W. He and K. Yuan, "An improved Hough transform and its realization on FPGA," in *Proc. 9th World Congr. Intell. Control Autom.*, Jun. 2011, pp. 13–17.

[7] R. Jeyakumar, M. Prakash, S. Sivanantham, and K. Sivasankaran, "FPGA implementation of edge detection using Canny algorithm," in *Proc. Online Int. Conf. Green Eng. Technol. (IC-GET)*, Nov. 2015, pp. 1–4.

[8] P. Promrit and W. Suntiamorntut, "Design and development of lane detection based on FPGA," in *Proc. 14th Int. Joint Conf. Comput. Sci. Softw. Eng. (JCSSE)*, Jul. 2017, pp. 1–4.

[9] M. Gopinathan, R. Soundarrakumar, A. Kalaiselvi, and A. Mohideen, "Implementation of lane detection in autonomous vehicle using FPGA," in *Proc. Int. Conf. Emerg. Trends Eng. Med. Sci. (ICETEMS)*, Nov. 2022, pp. 141–147.

[10] K. Kamimae, S. Matsui, Y. Araki, T. Miura, K. Motoyoshi, K. Yamashita, H. Ikehara, T. Kawazu, H. Yuwei, M. Nishimura, S. Abe, K. Okino, Y. Hashiguchi, K. Fukuda, K. Yanagihara, T. Manabe, and Y. Shibata, "A lane detection hardware algorithm based on Helmholtz principle and its application to unmanned mobile vehicles," in *Proc. Int. Conf. Field-Program. Technol. (ICFPT)*, Dec. 2022, pp. 1–4.

[11] H.-K. Jeong and Y.-J. Jeong, "Design of Hough transform hardware accelerator for lane detection," in *Proc. IEEE Int. Conf. IEEE Region (TENCON)*, Oct. 2013, pp. 1–4.

[12] X. Wang, C. Kiwus, C. Wu, B. Hu, K. Huang, and A. Knoll, "Implementing and parallelizing real-time lane detection on heterogeneous platforms," in *Proc. IEEE 29th Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, Jul. 2018, pp. 1–8.

[13] A. Mulyanto, R. I. Borman, P. Prasetyawan, W. Jatmiko, P. Mursanto, and A. Sinaga, "Indonesian traffic sign recognition for advanced driver assistent (ADAS) using YOLOv4," in *Proc. 3rd Int. Seminar Res. Inf. Technol. Intell. Syst. (ISRITI)*, Jakarta, Indonesia, Dec. 2020, pp. 520–524.

[14] A. Mulyanto, W. Jatmiko, P. Mursanto, P. Prasetyawan, and R. I. Borman, "A new Indonesian traffic obstacle dataset and performance evaluation of YOLOv4 for ADAS," *J. ICT Res. Appl.*, vol. 14, no. 3, pp. 286–298, Mar. 2021.

[15] R. Ayachi, Y. Said, and A. Ben Abdelaali, "Pedestrian detection based on light-weighted separable convolution for advanced driver assistance systems," *Neural Process. Lett.*, vol. 52, no. 3, pp. 2655–2668, Dec. 2020.

[16] J. Tang, S. Li, and P. Liu, "A review of lane detection methods based on deep learning," *Pattern Recognit.*, vol. 111, Mar. 2021, Art. no. 107623.

[17] W. Wang, H. Lin, and J. Wang, "CNN based lane detection with instance segmentation in edge-cloud computing," *J. Cloud Comput.*, vol. 9, no. 1, pp. 1–10, Dec. 2020.

[18] Q. Zou, H. Jiang, Q. Dai, Y. Yue, L. Chen, and Q. Wang, "Robust lane detection from continuous driving scenes using deep neural networks," *IEEE Trans. Veh. Technol.*, vol. 69, no. 1, pp. 41–54, Jan. 2020.

[19] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Oct. 2017, pp. 2980–2988.

[20] L. Riera, K. Ozcan, J. Merickel, M. Rizzo, S. Sarkar, and A. Sharma, "Driver behavior analysis using lane departure detection under challenging conditions," 2019, *arXiv:1906.00093*.

[21] R. Muthalagu, A. Bolimera, and V. Kalaichelvi, "Lane detection technique based on perspective transformation and histogram analysis for self-driving cars," *Comput. Electr. Eng.*, vol. 85, Jul. 2020, Art. no. 106653.

[22] B. K. Shah, V. Kedia, R. Raut, S. Ansari, and A. Shroff, "Evaluation and comparative study of edge detection techniques," *IOSR J. Comput. Eng.*, vol. 22, no. 5, pp. 6–15, 2020.
[23] G. N. Chaple, R. D. Daruwala, and M. S. Gofane, "Comparisions of Robert, Prewitt, Sobel operator based edge detection methods for real time uses on FPGA," in *Proc. Int. Conf. Technol. Sustain. Develop. (ICTSD)*, Feb. 2015, pp. 1–4.
[24] V. Torre and T. A. Poggio, "On edge detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vols. PAMI-8, no. 2, pp. 147–163, 1986, doi: 10.1109/TPAMI.1986.4767769.
[25] J. Canny, "A computational approach to edge detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-8, no. 6, pp. 679–698, Nov. 1986.
[26] Intel. *Overview of HPS Design Guidelines for SoC FPGA Design*. Accessed: Jan. 10, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683192/18-1/overview-of-hps-design-guidelines-for.html
[27] M. Greenberg and A. J. Blatt, "Executable formal semantics for the POSIX shell," *Proc. ACM Program. Lang.*, vol. 4, pp. 1–30, Jan. 2020.
[28] T. G. Mattson, Y. H. He, and A. E. Koniges, *The OpenMP Common Core: Making OpenMP Simple Again*. Cambridge, MA, USA: MIT Press, 2019.
[29] P. A. Kumar, "FPGA implementation of the trigonometric functions using the CORDIC algorithm," in *Proc. 5th Int. Conf. Adv. Comput. Commun. Syst. (ICACCS)*, Mar. 2019, pp. 894–900.
[30] H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom, "NuScenes: A multimodal dataset for autonomous driving," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2020, pp. 11618–11628.
[31] X. Pan, J. Shi, P. Luo, X. Wang, and X. Tang, "Spatial as deep: Spatial CNN for traffic scene understanding," in *Proc. AAAI Conf. Artif. Intell.*, 2018, vol. 32, no. 1, pp. 7276–7283.
[32] C. Khongprasongsiri, P. Kumhom, W. Suwansantisuk, T. Chotikawanid, S. Chumpol, and M. Ikura, "A hardware implementation for real-time lane detection using high-level synthesis," in *Proc. Int. Workshop Adv. Image Technol. (IWAIT)*, Jan. 2018, pp. 1–4.
[33] I. El Hajjouji, S. Mars, Z. Asrih, and A. El Mourabit, "A novel FPGA implementation of Hough transform for straight lane detection," *Eng. Sci. Technol., Int. J.*, vol. 23, no. 2, pp. 274–280, Apr. 2020.
[34] S. Malmir and M. Shalchian, "Design and FPGA implementation of dual-stage lane detection, based on Hough transform and localized stripe features," *Microprocessors Microsyst.*, vol. 64, pp. 12–22, Feb. 2019.

HEUIJEE YUN received the bachelor's degree in electronic engineering from Kyungpook National University, Daegu, South Korea, in 2022, where she is currently pursuing the M.S. degree with the School of Electronic and Electrical Engineering. She has a lot of experience in the algorithm of real-time object detection on lightweight embedded boards. She has published several journal/conference papers. She is researching technologies to optimize deep learning-based object detection algorithms to be applied to low-power embedded systems. Her main research interests include lightweight object recognition programs and simulation of autonomous driving.

DAEJIN PARK (Member, IEEE) received the B.S. degree in electronics engineering from Kyungpook National University, Daegu, South Korea, in 2001, and the M.S. and Ph.D. degrees in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 2003 and 2014, respectively. He was a Research Engineer with SK Hynix Semiconductor and Samsung Electronics for more than 12 years, from 2003 to 2014, and has worked on designing low-power embedded processors architecture and implementing fully AI-integrated system-on-chip with intelligent embedded software on the custom-designed hardware accelerator, especially for hardware/software tightly coupled applications, such as smart mobile devices and industrial electronics. He has been a full-time Processor with the School of Electronic and Electrical Engineering and the School of Electronics Engineering, Kyungpook National University, since 2014. He has published more than 240 technical articles and 50 patents. He was nominated as one of the Presidential Research Fellows 21, Republic of Korea, in 2014.