A case for edge video processing with FPGA SoC: reversi board detection using Hough transform

Naoki Fujieda† and Naoya Ito
Department of Electrical and Electronics Engineering, Faculty of Engineering,
Aichi Institute of Technology, Toyota, Aichi, Japan
†nfujieda@aitech.ac.jp

***Abstract*—There has been increasing needs for image and video processing at edge devices, such as low-end FPGA System-on-Chips (SoCs). In this paper, we present an accelerator of a reversi board detection algorithm for a Zynq-7000 SoC, to investigate advantages and limitations of edge video processing with FPGA SoC. The accelerator is developed using the AMD Vitis vision library and its pipeline includes twelve OpenCV functions. Thanks to high level synthesis, hardware generation itself is quite simple. However, a part of the algorithm has to be modified, because not all of the OpenCV functions are suitable for hardware implementation and thus supported by the library. In our case, contour detection was replaced with Hough transform. According to our evaluation, the board detection became 16.1 times and 9.4 times faster than the software implementations of the original and modified algorithms, respectively. The accelerator consumed 23%–54% of logic elements in the XC7Z020 device — 28,635 LUTs, 38,442 flip-flops, 50 DSP blocks, and 47 RAM blocks.**

I. Introduction

Computer vision algorithms and their applications, such as object detection [21] and anomaly detection [14], have been surprisingly grown in recent 20 years. A major turning point was marked by AlexNet [11], one of the most famous deep neural networks (DNNs), while traditional algorithms have still been widely applied. Implementing these algorithms on edge devices is challenging, where power consumption is strictly limited. The use of FPGA System-on-Chip (SoC) is one of the attractive solutions, along with an embedded GPU-based SoC, such as the Nvidia Jetson series [10]. With its reconfigurability and parallelism, an FPGA SoC-based solution has a potential for providing higher flexiblility and energy efficiency than a GPU-based one. To this end, both DNN-based [18] and traditional [5] algorithms have been investigated for FPGA implementations. Typical applications include object detection [13], pedestrian detection [19], and stereo camera [20].

Libraries and frameworks can improve the productivity of development of FPGA-based accelerators. For DNN inference, open-source frameworks such as GUINNESS [12] and Brevitas/FINN [17] support training of a quantized DNN model and generation of a streaming architecture from the model. Applications using these frameworks are, for example, an AI-based robot car [7] and a face mask detector [6]. As image processing libraries for AMD FPGAs, the Vitis vision library [2] and HiFlipVX [9] are available. They provide C/C++ functions compatible with OpenCV and OpenVX, respectively, which make it easy to offload a part of an image processing program to an FPGA. The functions are optimized for the Vitis HLS (High Level Synthesis) tool.

In this paper, we present an accelerator of a reversi board detection algorithm, to investigate advantages and limitations of edge video processing with a low-end FPGA SoC. The algorithm is originally implemented in Python with OpenCV [16]. The accelerator is developed using the Vitis vision library and implemented on a Zynq-7000 SoC. Many existing accelerators [4], [5], [13] implement algorithms that include only a few OpenCV functions. The proposed accelerator implements more complicated algorithm composed of a series of twelve OpenCV functions. We show a demonstrative system for the accelerator using a Digilent PYNQ-Z1 FPGA SoC board, an HDMI video camera, and an LCD display. The system captures a frame from the camera, detects a board and stones, overlays the detection results with the camera images, and then shows them on the display. We also conduct a performance comparison with software implementations on the PYNQ-Z1 board.

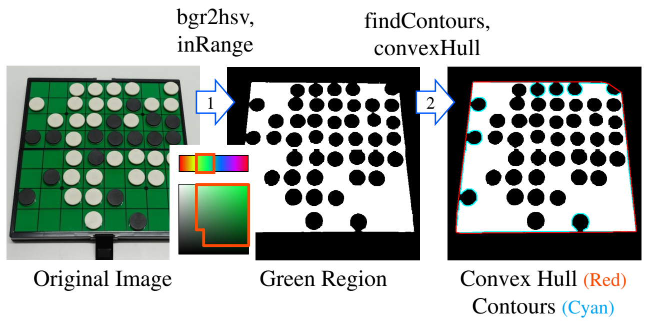


Fig. 1. Overview of the original version of board detection algorithm.

II. Background

A. Board Detection Algorithm

Figure 1 abstracts the reversi board detection algorithm. It is a part of a board recognization program to locate the board and stones and recognize the color of the stones. The source code is written in Python with OpenCV and available at GitHub [16]. In this paper, we deal with the board detection part only: detection and recognization of stones are left unmodified and executed as software.

The algorithm basically consists of two steps: (1) extraction of green region and (2) finding the convex hull of the contours of the green region. We assume the input image is full-color in the RGB color space, with up to 1024 pixels on the longer side.

In the first step, the input image is blurred and color converted to the HSV color space. OpenCV defines the range of H (hue) as 0–179, and the range of S (saturation) and V (value) as 0–255. A pixel is considered green if $45 \leq H \leq 90$ and at least one of the following conditions are met:

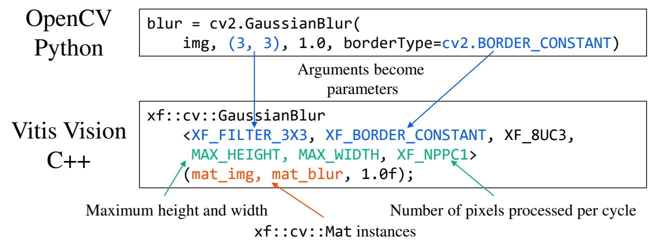
- $S \geq 89$ and $V \geq 30$, or
- $S \geq 64$ and $V \geq 89$.

These thresholds are shown in red frames in Fig. 1.

In the second step, the pair of `findContours` and `convexHull` OpenCV functions are applied to the green region, to obtain its contours. It is one of a common ways to detect objects of arbitrary shape [8]. Four sides of the board are then determined from the longest segments of lines. The second step is actually executed twice, and in the first execution, white region is determined and added to the green region. This prevents the green region from being divided by stones and improves the precision of detection.

B. Vitis Vision Library

AMD provides the Vitis vision library [2] as an open-source hardware library optimized for their FPGAs. It was called xfOpenCV before being incorporated into the Vitis library. It has a similar API to OpenCV: many functions have the same names as OpenCV in the `xf::cv` namespace. Input and output images must be an instance of the `xf::cv::Mat` class, which is similar to the `cv::Mat` class of OpenCV. Inside a function, all pixels in the instance are written and read in order. A reference of instance in the argument is converted into a simple stream interface upon high level synthesis.



```
blur = cv2.GaussianBlur(
    img, (3, 3), 1.0, borderType=cv2.BORDER_CONSTANT)
```

```
xf::cv::GaussianBlur
    <XF_FILTER_3X3, XF_BORDER_CONSTANT, XF_8UC3,
     MAX_HEIGHT, MAX_WIDTH, XF_NPPC1>
    (mat_img, mat_blur, 1.0f);
```

Fig. 2. Comparison of the Gaussian blur functions between OpenCV and Vitis vision.

Since the Vitis vision library functions are expected to be synthesized to hardware modules, there are limitations and differences from OpenCV. Major differences are as follows.

- Some arguments of OpenCV functions become template parameters, which means they cannot be modified during runtime.
- The maximum height and width of input images must be given as template parameters.
- Some functions have an option to process eight pixels per cycle, instead of one, for the highest performance. It is also determined as a template parameter.
- Writes and reads to the `xf::cv::Mat` instance must always be performed the same number of times. If an image is used by multiple functions, it must be duplicated by the `duplicateMat` function.

Figure 2 depicts an example of differnces between an OpenCV function and the corresponding Vitis vision function. Note that template parameters in C++ are enclosed in the angle brackets, while arguments are enclosed in the parentheses. In this case of the `GaussianBlur` function, the parameter σ (standard deviation) can be changed during runtime but the filter size and the type of image boundary processing cannot.

C. Hough Transform

Since not all of the OpenCV functions are suitable for hardware implementation, some functions are not supported in the Vitis vision library. In our case, the `findContours` function used in the board detection algorithm is not supported. We thus modified the algorithm to use the Hough line transform instead, which will be shown in Section III. In this section, we briefly explain the Hough line transform and the corresponding `HoughLines` function of the Vitis vision library.

The Hough line transform detects lines from an edge-detected binary image. The width and height of the input image are denoted as W and H, respectively. For each pixel at (x, y), where $0 \leq x < W$ and $0 \leq y < H$, a set of lines go through it are expressed in polar coordinate as

$$\rho = x \cos\theta + y \sin\theta. \tag{1}$$

The pairs of (ρ, θ), corresponding to (x, y) in the x–y plane, form a sine curve in the ρ–θ plane. When multiple sine curves have an intersection in the ρ–θ plane, the correspoinding points in the x–y plane will form a line.

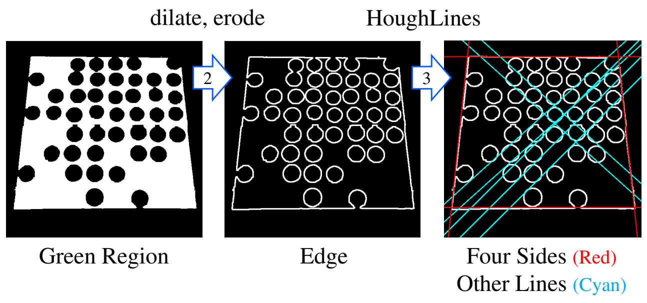


Fig. 3. Overview of the modified version of board detection algorithm.

The actual algorithm consists of two parts: voting and finding local maxima. In the voting part, each edge pixel is transformed into a sine curve in the ρ–θ plane. For every quantized point in the ρ–θ plane, the number of curves that pass through it are counted. After that, local maxima of votes that exceed a certain threshold are extracted as detected lines. They are finally output as a list of pairs of ρ and θ.

The `HoughLines` function of the Vitis vision library has two important differences from that of OpenCV. First, the origin of polar coordinate is in the center of the input image, instead of the upper-left corner. Considering this difference of the origin, Eq. (1) can be translated into

$$\rho = (x - W/2)\cos\theta + (y - H/2)\sin\theta. \qquad (2)$$

The range of ρ and θ becomes $-d/2 \leq \rho < d/2$ and $0 \leq \theta < \pi$, where $d = \sqrt{W^2 + H^2}$. Second, the detected lines are written separately to the fixed-size arrays of ρ and θ. The size of the arrays is determined by the LINESMAX template parameter. If the number of detected lines is larger than LINESMAX, the lines in excess are simply discarded. If it is smaller than LINESMAX, the last pair of ρ and θ is repeatedly written until the end of the arrays.

III. Algorithm Modification

Figure 3 describes the modified version of the reversi board detection algorithm, in order to replace contour detection with the Hough transform. Note that the step to extract green region, Step 1, is identical to the original version shown in Fig. 1. Since the Hough transform requires an edge-detected image, the Laplacian filter is applied in Step 2 as a preprocessing. The Hough transform then detects lines from the edge of the green region in Step 3.

When many stones are placed on the board, edges of stones may also form lines, as drawn by cyan lines in Fig. 3. We design the following postprocessing steps to find the four sides of the board from the detected lines:

1) clustering lines with similar θ into groups
2) making pairs of the groups that are nearly orthogonal to each other
3) finding the pair that can make the largest rectangle, and
4) extracting two lines with the maximum and the minimum ρ from each of the pair.

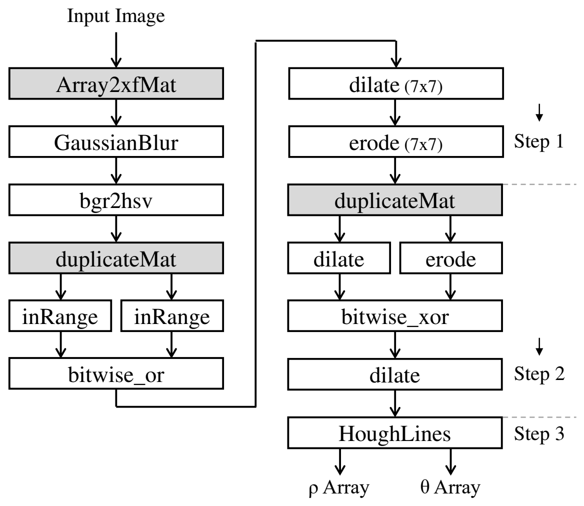


Fig. 4. The dataflow of the reversi board detection accelerator.

In Fig. 3 for example, the red lines form one pair of line groups and the cyan lines form another. The red lines are then preferred in the third step.

IV. Implementation

Figure 4 depicts the dataflow of the developed accelerator. The filter size is 3×3 unless otherwise described. It is assumed that three physically contiguous arrays are allocated on the main memory of the PS (processing system): input image, ρ values, and θ values. The accelerator reads from and writes to the PS memory when needed, using a separate AXI manager interface for each array. The accelerator also has an AXI-Lite subordinate interface to receive arguments from the PS, such as physical addresses of the arrays and size of the input image.

White boxes correspond to OpenCV functions, while gray boxes are extra functions required in the Vitis vision library. Due to an issue of difference of data types, the Laplacian filter in Step 2 is replaced with equivalent functions. As a result, the pipeline of the accelerator includes twelve OpenCV functions. In the Hough transform, the steps of ρ and θ are set to 3 pixels and 3 degrees (or $\pi/60$ radians), respectively. The maximum size of input image is 1024 $\times$ 1024 and the number of votes required to being detected as a line is 500. The maximum number of lines to detect (or LINESMAX) is set to 32.

V. Demonstrative System

A. System Design

To visualize how the developed accelerator works, a demonstrative system that includes the accelerator is also developed. Its top-level block diagram is shown in Fig. 5. The target board is a Digilent PYNQ-Z1 board, which includes a Zynq-7000 SoC (XC7Z020), and the system is based on the PYNQ platform [1]. Some existing FPGA-based video processing systems use a small CMOS camera such as OV9655 as an input device [7], [15]. We prefer the HDMI interface because the PYNQ-Z1 board has two HDMI receptacles and the related IP cores are available as a part of the PYNQ platform.

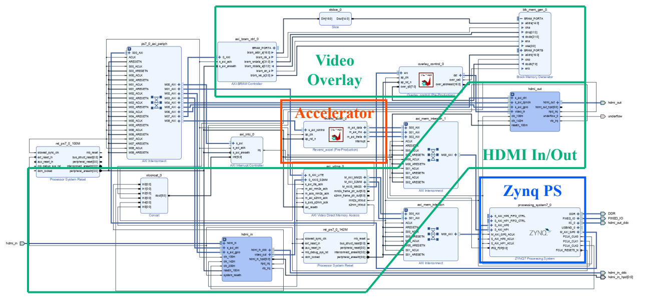


Fig. 5. The block diagram of demonstrative system for the reversi board detection accelerator.

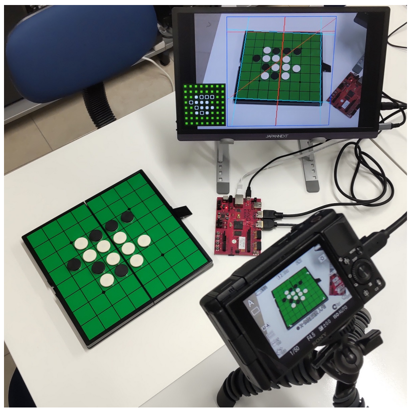


Fig. 6. A photograph where the demonstrative system is working.

Blocks with the blue frame and the red frame are the Zynq PS and the developed accelerator, respectively. Two green frames correspond to video subsystems: an HDMI input/output subsystem and a video overlay subsystem.

The HDMI subsystem is extracted from the base design of the PYNQ platform, while some unnecessary blocks are omitted from the video pipeline. The input pipeline is usually tied to the output pipeline, which means basically the same image as the input is output after a slight delay. Frame buffers are allocated on the PS main memory. This enables a user to receive the most recently captured frame.

The video overlay subsystem is composed of an overlay controller, a block memory, and an AXI interface for the block memory. The controller is inserted into the HDMI output pipeline in order to overlay a user-drawn image on the origial output image. The block memory works as the frame buffer for overlay, which is written from the PS via the AXI interface and read from the controller. This design makes the processing of HDMI signals asynchronous to the other processing such as detection and recognization. In other words, even during a long processing, the input video does not look to be stuck on the output device. The resolution of the overlay is 480 $\times$ 270 pixels and each pixel has 8 bits, one of which is to determine if the corresponding pixel is transparent.

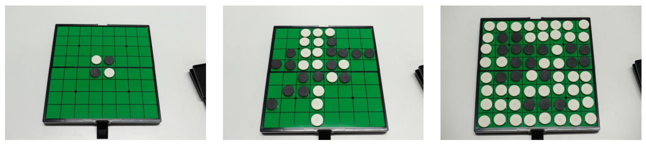


Fig. 7. Examples of pictures used in the performance evaluation.

B. Working Example

Figure 6 depicts a working example of the demonstrative system. The system uses an HDMI video camera and an LCD display as HDMI input and output devices, respectively. Full HD (1920 $\times$ 1080 pixels) images are captured from the camera. The central area of 1024 pixels square, shown in a blue box, is cropped and used for the further processing. The board detection using the accelerator is conducted at first and, if succeeded, detection and recognition of stones are performed by software. Their results are graphically shown by writing an image to the overlay frame buffer. When processing for one frame is completed, processing for another frame starts, which is the most recently captured at that point.

With this system, we demonstrated that moderately complicated video processing became almost real-time. As we will evaluate quantitatively in Section VI.B, it only takes tens of milliseconds to complete the board detection. While the system fails to detect a board, lines detected by the Hough transform quickly follow the input video. Once a board is successfully detected, it takes longer time (about a second) to recognize the stones on the board.

VI. Evaluation

A. Methodology

In this section, we evaluate the developed accelerator in two aspects: the performance of board detection and the amount of required hardware. In the performance evaluation, we measure the time to complete the whole process of board detection and stone recognization, as done in the original software [16]. We also counts the number of boards detected and stones correctly recognized. They are compared with the *original* and the *modified* versions of software implementation. In the evaluation of the amount of hardware, we count the numbers of look-up tables (LUTs), flip-flops (FFs), digital system processing (DSP) blocks, and RAM blocks required. It is important to assess the possibility of further extension, because the number of logic elements available in a Zynq-7000 SoC is not so large.

For comparison, we prepared another version of the accelerator and the system where only the `HoughLines` function is hardware implemented. We call this version *Hough-only*, to distinguish from the *full* version as shown in Fig. 4. In this case, the green region extraction and the edge detection, or Step 1 and Step 2 in Fig. 4, are processed in software.

As a dataset, we prepared 100 pictures of reversi boards taken by a digital camera. The number of stones on the board is gradually increased as the picture ID increases. Figure 7 gives

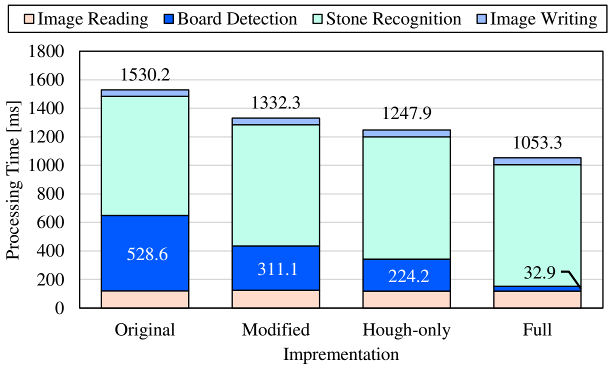


Fig. 8. Comparison of the average time for processing.

some examples. The resolution of all of the pictures is 1024 × 683 pixels. The board detection is considered success if a sufficiently large rectangle (> 200 × 200 pixels) is found in the input image. For each board detected, the stone recognition algorithm guesses the color of the stone in each cell as either white, black, blank, or unknown. A blank stone means that a stone is not detected in that cell, while an unknown stone indicates that a stone is detected but the algorithm fails to recognize its color. The accuracy of the stone recognition is determined by comparing the guessed color with the pre-determined correct answer (in white, black, or blank). The results of the stone recognition are also saved as small images. Note that the case where the board detection gives a wrong region will surface as a drop of the stone recognition accuracy, instead of the board detection accuracy.

The latency is measured separately in four parts: (1) image reading, (2) board detection, (3) stone recognition, and (4) image writing. When the board is not detected, the latter two parts are skipped and omitted from the calculation of average time.

The accelerator is high level synthesized using the Vitis vision library 2021.1 and Vitis HLS 2021.1. The demonstrative system is synthesized and implemented with Vivado 2020.2, which is the version recommended for PYNQ 2.7. However, due to a memory management issue of the Python library, PYNQ was downgraded to 2.6. This version of PYNQ includes Python 3.6.5 and OpenCV 3.4.3.

B. Latency and Accuracy

Figure 8 compares the average time taken to process one image by each implementation. The board detection became 16.1 times faster and the time taken by the whole process became 31% shorter. In the *original* software implementation, the board detection had 35% of the time of the whole process. The percentage now reduced to 3.1% in the *full* version of hardware implementation. By comparing with the *modified* version, it can be seen that the modification of algorithm itself also contributed to the speedup. The *modified* version took 41% shorter time than the *original*. The hardware implementation was even 9.5 times faster.

TABLE I
COMPARISON OF THE NUMBER OF BOARDS DETECTED AND STONES CORRECTLY RECOGNIZED.

Implementation	# boards	% stones
Original	100	99.98 (6,399 / 6,400)
Modified	100	99.80 (6,387 / 6,400)
Hough-only	64	97.07 (3,976 / 4,096)
Full	98	98.04 (6,149 / 6,272)

TABLE II
THE NUMBER OF LOGIC ELEMENTS IN USE.

Impl. / Module	LUT	FF	DSP	RAM
w/o Accelerator	10,354	15,508	0	41.5
Hough-only	33,375	48,293	11	73.0
Full	41,831	58,268	50	88.5
	(79%)	(55%)	(23%)	(63%)
- accelerator	28,635	38,442	50	47.0
	(54%)	(36%)	(23%)	(34%)
- Array2xfMat	1,048	998	4	0.0
- GaussianBlur	3,231	3,538	33	3.0
- bgr2hsv	363	438	6	1.0
- duplicateMat (1)	49	68	0	0.0
- inRange (1)	81	143	0	0.0
- inRange (2)	73	125	0	0.0
- bitwise_or	57	71	0	0.0
- dilate (7x7)	946	1,585	0	3.5
- erode (7x7)	1,024	1,729	0	3.5
- duplicateMat (2)	51	68	0	0.0
- dilate (1)	307	382	0	1.5
- erode	290	334	0	1.5
- bitwise_xor	50	78	0	0.0
- dilate (2)	276	358	0	1.5
- HoughLines	17,388	24,336	7	30.0

Table I compares the number of boards detected (# boards) and the percentage of stones correctly recognized (% stones) for each implementation. It was confirmed that the effect of the modification of algorithm on accuracy was only 0.2%, by detecting slightly larger or smaller region than the actual board. However, we encountered a serious drop of accuracy with hardware implementations. It was more apparent when only the Hough transform was implemented in hardware, where more than a third of the boards were not or wrongly detected. We noticed that some of the lines were omitted randomly, even though we gave exactly the same input image. We think the most likely possibility is that there is a bug in the library, regarding uninitialized variables. According to the revision history of the file that implements the Hough transform [3], it looks that a modification to fix the very bug has been made in version 2024.1, the latest version at the time of writing this paper. It is left as future work to adopt the latest version of environment or backport that modification to the version currently using.

C. Amount of Hardware

Table II summarizes the numbers of LUTs, FFs, DSP blocks, and RAM blocks required by the developed systems. For the *full* version of the accelerator, a per-function breakdown is also shown in the table. For comparison, the case without any accelerator (*w/o Accelerator*) is also measured. The number of RAM blocks is counted as 1 for a 36-kib RAM and 0.5 for a 18-kib RAM. The accelerator used 54% of LUTs, 36% of FFs, 23% of DSPs, and 34% of RAMs compared to the number of elements available in the XC7Z020 device. As for the whole system, 79% of LUTs were in use. We will have to consider downsizing of some of the modules for a further extension.

When looking at the breakdown carefully, the following observations can be made.

- Selection of the step of θ in the Hough transform heavily affects the numbers of LUTs, FFs, and RAM blocks. To process one pixel in every cycle, the voting process must be done in parallel for each possible value of θ. In our case, as the step was set to 3 degree, we needed 60 parallel voting. For each voting process, a 18-kib RAM block is used to store the number of votes. A few hundred LUTs and FFs are also used to increment the votes by one and the ρ value by $\cos\theta$.
- The number of DSP blocks for the Hough transform is considered to be a constant. DSPs are used for initilization of the ρ value and conversion of results to floating-point numbers. From Eq. (2), the ρ value at the upper-left corner pixel becomes

$$\rho = -(W/2)\cos\theta - (H/2)\sin\theta. \tag{3}$$

  It is also required for each possible value of θ. However, this calculation is only required during initialization and it don't have to be done in parallel. Conversion of results can also be done sequentially.
- Gaussian blur might be optimized by replacing with a custom filter, or `Filter2D`. It was the second largest module in the developed accelerator. As we have described in Fig. 2, the parameter σ is still an argument. Even though σ is a constant, it looked that the filter coefficients were always calculated, or not removed by constant propagation.
- Filters that requires neighboring pixel values use a small number of RAM blocks as line buffers. When an accelerator includes a large number of such filters, RAM capacity might become a problem.
- The number of logic elements for simple operation, such as duplication, comparison, and bitwise logic operation, is negligible.

VII. Conclusion

In this paper, we presented an edge video processing accelerator and its applied system, taking a reversi board detection as a theme. Through the development and evaluation of them, we found the following pros and cons of low-end FPGA SoC-based edge video processing: (P1) generation of accelerator from a series of OpenCV functions became quite simple using the Vitis video library, (P2) moderately complicated video processing became almost real-time; (C1) developers might have to modify their algorithm to be suitable for hardware implementation, and (C2) a limited number of logic elements, especially LUTs, might limit the expandability of the accelerator.

An immediate future work is to apply hardware implementation to the parts that have not been accelerated in this paper — stone recognition. However, we will have to optimize some of the resource-eating functions such as `HoughLines` and `GaussianBlur` before doing so. For the long-term, we are planning to use the findings of this research to accelerate image and video processing in various fields. We will then compare these results with embedded GPU-based solutions.

Acknowledgment

A part of this study was supported by the Nitto Foundation. The first author personally thanks Mr. Daikan Harada and Mr. Kotaro Hikosaka for their conrtibution in initial evaluations.

References

[1] Advanced Micro Devices. PYNQ — Python Productivity to AMD Adaptive Compute platforms. [Online]. Available: https://www.pynq.io/

[2] ——. Vitis Vision Library. [Online]. Available: https://github.com/Xilinx/Vitis_Libraries/tree/main/vision

[3] ——. xf_houghlines.hpp (Vitis Vision Library). [Online]. Available: https://github.com/Xilinx/Vitis_Libraries/blob/main/vision/L1/include/imgproc/xf_houghlines.hpp

[4] P. Babu and E. Parthasarathy, "Hardware Acceleration of Image and Video Processing on Xilinx Zynq Platform," *Intelligent Automation and Soft Computing*, vol. 30, no. 3, pp. 1063–1071, 2021.

[5] A. Cortés, I. Vélez, and A. Irizar, "High level synthesis using Vivado HLS for Zynq SoC: Image processing case studies," in *31st Conference on Design of Circuits and Integrated Systems*, 2016, pp. 1–6.

[6] N. Fasfous *et al.*, "BinaryCoP: Binary Neural Network-based COVID-19 Face-Mask Wear and Positioning Predictor on Edge Devices," in *35th International Parallel and Distributed Processing Symposium Workshops*, 2021, pp. 108–115.

[7] F. Hamanaka, T. Kanamori, and K. Kise, "A low cost and portable mini motor car system with a BNN accelerator on FPGA," in *14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, 2021, pp. 84–89.

[8] A. Huamán. Convex Hull, OpenCV Tutorials. [Online]. Available: https://docs.opencv.org/4.4.0/d7/d1d/tutorial_hull.html

[9] L. Kalms, A. Podlubne, and D. Göhringer, "HiFlipVX: an Open Source High-Level Synthesis FPGA Library for Image Processing," in *15th International Symposium on Applied Reconfigurable Computing*, 2019, pp. 149–164.

[10] L. S. Karumbunathan. Solving Entry-Level Edge AI Challenges with NVIDIA Jetson Orin Nano. NVIDIA Corporation. [Online]. Available: https://developer.nvidia.com/blog/solving-entry-level-edge-ai-challenges-with-nvidia-jetson-orin-nano/

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *25th International Conference on Neural Information Processing Systems*, 2012, pp. 1097–1105.

[12] H. Nakahara *et al.*, "GUINNESS: A GUI based binarized deep neural network framework for software programmers," *IEICE Transcations on Information and Systems*, vol. E102.D, no. 5, pp. 1003–1011, 2019.

[13] M. P. R. Prasad and A. K. Singh, "FPGA based object parameter detection for Embedded Vision Application," *International Journal of Computing and Digital Systems*, vol. 14, no. 1, pp. 1091–1099, 2023.

[14] L. Ruff *et al.*, "A Unifying Review of Deep and Shallow Anomaly Detection," *Proceedings of the IEEE*, vol. 109, no. 5, pp. 756–795, 2021.

[15] S. Sarkar, S. S. Bhairannawar, and K. B. Raja, "FPGACam: A FPGA based efficient camera interfacing architecture for real time video processing," *IET Circuits, Devices and Systems*, vol. 15, no. 8, pp. 814–829, 2021.

[16] A. Tanaka. reversi_recognition. [Online]. Available: https://github.com/lavox/reversi_recognition

[17] Y. Umuroglu *et al.*, "FINN: A framework for fast, scalable binarized neural network inference," in *25th International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 65–74.

[18] M. P. Véstias, "A Survey of Convolutional Neural Networks on Edge with Reconfigurable Computing," *Algorithms*, vol. 12, no. 8, pp. 154:1–154:24, 2019.

[19] M. Wasala and T. Kryjak, “Real-time HOG+SVM based object detection using SoC FPGA for a UHD video stream,” in *11th Mediterranean Conference on Embedded Computing*, 2022, pp. 1–6.

[20] C. Wu and K. Weng, “The Development and Implementation of a Real-Time Depth Image Capturing System Using SoC FPGA,” in *30th International Conference on Advanced Information Networking and Applications Workshops*, 2016, pp. 934–938.

[21] Z. Zou *et al.*, “Object Detection in 20 Years: A Survey,” *Proceedings of the IEEE*, vol. 111, no. 3, pp. 257–276, 2023.