Artificial Life and Robotics (2024) 29:29–36
https://doi.org/10.1007/s10015-023-00927-2

ORIGINAL ARTICLE

Light-weight color image conversion like pencil drawing for high-level synthesized hardware

Honoka Tani[1] · **Akira Yamawaki**[1]

Received: 28 August 2023 / Accepted: 24 November 2023 / Published online: 22 December 2023

Abstract
We are developing pencil-drawing-style image conversion software suitable for high-level synthesis, HLS, technology that automatically converts software into hardware. The pencil-drawing-style image conversion consists of the former and latter processes. The former generates the images expressing edge strengths and their directions. The latter process convolves the line segment corresponding to the edge strength with its direction. As hardware-oriented software description, the medium data across the former and latter processes are optimized. In addition, the former and latter processes are overlapped between the FIFO buffer passing the medium data. The obtained image is still a gray-scaled image. To make it support the color image, this paper inserts a process compositing the original color image with the grayed pencil-drawing-style image to not intervene in the pipelined data path behavior. As a result, an HLS tool used is expected to generate a hardware module with the ideal pipelined data path by one output data/one clock. The experimental results show that the colorization hardware had no significant performance degradation issues for circuit size, run time, or power efficiency compared to the pencil drawing hardware with grayscale. Compared with the software execution, our hardware supporting color image can achieve 4.2 times the performance improvement and 130 times power efficiency.

Keywords High-level synthesis · Non-photorealistic rendering · FPGA · Colorization

1 Introduction

Further development in image processing technology will accelerate the growth of virtual space systems. We aim to realize battery-driven smart glasses that can run long-term and realize virtual spaces without PCs or GPUs. One of the heavy-load image processing performed on smart glasses is non-photorealistic rendering (NPR) [1], which is the composition, processing, and transformation of authentic images in the field of view. We are developing hardware for high performance and low-power consumption for pencil drawing-style image conversion [2], which is one of the NPR methods. The development uses high-level synthesis (HLS), automatically converting software (SW) into hardware (HW). Although HLS techniques have been used to develop HW modules for several software applications, we cannot find an example adapted to pencil-drawing-style image conversion of NPRs [3–5]. Using HLS allows for quick and flexible changes and improvements to the algorithm, thus significantly reducing the burden of HW design [6–9]. However, when using HLS, large, slow HW may be generated if the software program is not HW oriented. Therefore, to use HLS effectively, it is necessary to create HW-oriented software programs. Thus, we are improving algorithms and SW programs for HLS to develop better HW.

As an initial step in developing pencil-drawing-style image conversion HW, we divided the entire process into former and latter sub-processes to reduce the target size and develop SW for each HLS [10, 11]. Then by overlapping the execution of the generated former and latter HLS modules,

Honoka Tani is the presenter of this paper.

This work was submitted and accepted for the Journal Track of the joint symposium of the 29th International Symposium on Artificial Life and Robotics, the 9th International Symposium on BioComplexity, and the 7th International Symposium on Swarm Behavior and Bio-Inspired Robotics (Beppu, Oita and Online, January 24–26, 2024).

✉ Akira Yamawaki
yama@ecs.kyutech.ac.jp

Honoka Tani
tani.honoka795@mail.kyutech.jp

[1] Kyushu Institute of Technology, Fukuoka, Japan

the data path was pipelined to achieve the ideal performance of one processing/one clock [12].

This paper attempts to insert a light-weight colorizing hardware to the data path of gray-scaled version previously we have developed, without performance degradation and huge expanding hardware resources. Thus, we propose the simple colorizing algorithm based on the alpha blending, mixing the grayed pencil like image and the original color image. Since this algorithm is relatively simple using only stream input/output data, it is expected not to impair the flow of the existing pipeline. The experiments will show the performance effects of colorization of pencil-style images in terms of circuit size, execution time, and power efficiency.

We organize this paper as follows. Section 2 briefly describes the overall process flow of the pencil-drawing-style image conversion process with gray scale, and Sect. 3 describes the proposed method for colorization. In Sect. 4, we present the verification results and a discussion. Finally, Sect. 5 describes the conclusion of this paper.

2 Pencil-drawing-style image conversion with gray scale

The pencil-drawing-style image conversion in this study is an algorithm that imitates the characteristic of pencil sketching, in which multiple line segments are overlapped along an outline in the same direction. The algorithm considers the edge of the input image as the outline and convolves the line segments along the edge direction to obtain the desired atmosphere. Figure 1 shows an overview of the conventional process.

The overall processing flow consists of former processing and latter processing. In the former processing, an edge strength image of the input image is obtained. The color input image is gray scaled, and a Sobel filter is applied to extract edges. When the Sobel filter is applied, the edge strength is given in the x- and y-directions and the vector direction is the edge. This is separated into eight directions of 22.5 degrees each to obtain an edge strength image. The line segments corresponding to the edge strength images obtained from the former processing are convolved in the later processing. The output images are combined, and the brightness is inverted to produce a pencil-drawing-style image.

3 Proposal method

This paper proposes a method to add colorization processing to the existing grayscale pencil-drawing-style image conversion algorithm without performance loss. In 3.1, we show the improved processing flow for colorization and describe the specific algorithm in 3.2. We then propose a method of inserting the data path without disturbing the flow of the existing pipelined data path in 3.3. Finally, we describe an overview of the operation of the proposed HW based on an overview of the existing HW operation in 3.4.

3.1 Restructured processing flow for color image

To realize a colorized pencil-drawing-style image conversion, we propose a processing flow shown in Fig. 2. Therefore, we develop a relatively simple algorithm that can tailor the degree of transparency based on alpha blending. This idea is motivated by seeing the color source image through its grayscale pencil drawing image. This see-through method is explained in 3.2.

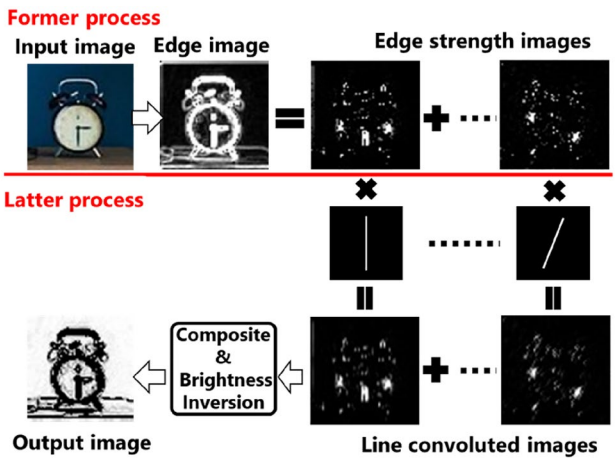


Fig. 1 Conventional pencil drawing style conversion

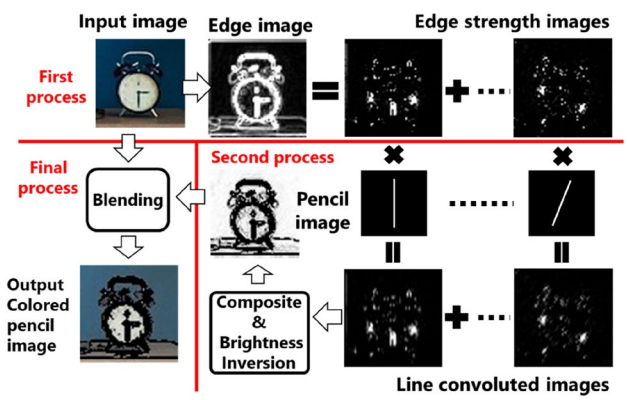


Fig. 2 Colored pencil drawing style image conversion

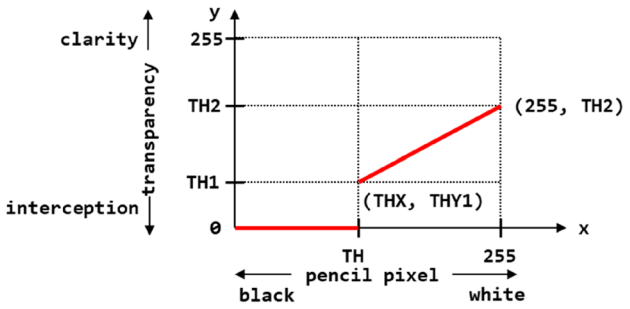


Fig. 3 From gray to transparency

3.2 Colorization algorithm

By overlapping the pencil image onto the input color image, colorization is realized. The transparency in overlapping is calculated from the pixel values of the pencil image, and the overlap of the input color image is adjusted according to the transparency. Figure 3 shows the method of calculating transparency.

The overlap of images is adjusted by varying the thresholds (TH, TH1, TH2). The TH is a primary threshold to emphasize the pencil drawing. The larger TH, the stronger pencil strokes are left. The TH1 and TH2 represent the strength of the transparency about the part with less pencil stroke. The looser the slope of the line, the blurrier see-though image is. The *x*-axis direction indicates the pixel value of the pencil drawing, where 0 is black and 255 is white. When the pixel value is less than TH, the transparency is always set to 0, and the pencil image is output as it appears. The *y*-axis direction indicates the degree of transparency. Transparency '0' indicates that the input color image is blocked, while transparency '255' indicates that the input color image is output as it is.

$$a = \frac{\mathrm{TH2}-\mathrm{TH1}}{255-\mathrm{TH}}, \tag{1}$$

$$y = \left(\mathrm{pen}_{\mathrm{gray}_{\mathrm{img}}} - \mathrm{TH}\right) * a + \mathrm{TH1}. \tag{2}$$

Therefore, the output pixel value "final_img", overlapping the pencil image "pen_gray_img" and the input color image "org_col_img", can be given by the following equation. The transparency, y, is still normalized from 0 to 255. However, since the transparency is inherently rate [%], Eq. (3) divides the parts about y with 256 to make their ranges to be [0:1.0].

$$\mathrm{final}_{\mathrm{img}} = \frac{\mathrm{org}_{\mathrm{col}_{\mathrm{img}}} * y + \mathrm{pen}_{\mathrm{gray}_{\mathrm{img}}} * (256-y)}{256}. \tag{3}$$

A pseudo code representing these equations is shown in Fig. 4.

```
u16 Transmittance( u8 src, u8 TH, u8 TH1, u8 TH2 ){
    int32 a, b, y;

    if( src <  th ){
        y = 0;
    } else{
        a = ((TH2 - TH1) << 8) / ( 255 - TH );
        y = ((a * (src - TH)) >> 8) + TH1;
    }
    return y;
}
void Coloring(u32 *org_col_img, u8 *pen_gray_img,
              u32 *final_img  , u16 w, u16 h){
  int i, j;
  u32 org_pix, final_pix;
  u16 y;
  u8 fr, fg, fb, pen_pix;

  for( i = 0; i < height; i++ ){
    for( j = 0; j < width; j++){
      org_pix = org_col_img [i * w + j];
      pen_pix = pen_gray_img[i * w + j];
      y = Transmittance(org_pix, TH, TH1, TH2);

      fr=(y* (org_pix      & 0xff)+(256 - y)*pen_pix)>>8;
      fg=(y*((org_pix>> 8) & 0xff)+(256 - y)*pen_pix)>>8;
      fb=(y*((org_pix>>16) & 0xff)+(256 - y)*pen_pix)>>8;

      final_img[i * w + j]= nb << 16 | ng << 8 | nr;
    }
  }
}
```

Fig. 4 Transmitting function and coloring

The Transmittance function in Fig. 4 corresponds to Eqs. (1) to (2). As explained above, the transparency is calculated from the pixel values of the pencil drawing. The inside of Transmittance has been converted from the floating-point calculation to the fixed-point calculation. This is because HLS is not good at handling floating-point numbers. When floating-point numbers are included, the calculators used makes the amount of HW very large. To prevent this, when multiplication is performed in a function, the value is once shifted to the left to make it larger. This way, the calculation result becomes an integer and does not adversely affect the HLS. When calculating the transparency "*y*," the desired value can be obtained by right shifting the value again.

The coloring function shown in the bottom of Fig. 4 realizes Eq. (3). It is assumed that the color pixel consists of B, G, and R each of them is 8-bit data. Therefore, the final RGB blending the original RGB by the transparency are calculated individually. To calculate the final RGB (fb, fg, and fr), original RGB is extracted from the color pixel by shifting and masking. The dividing with 256 is realized by right shifting of 8-bit to eliminate the divider, making the amount of HW large. Finally, the obtained RGB are concatenated to new single pixel, and this pixel is stored to the output image.

3.3 Restructure of existing data path for coloring

As shown in Fig. 2, the blending process using the pencil drawing image and the original color image must be added. Intuitively, the execution time of the whole process becomes larger by the added processes. To avoid this performance degradation, we propose the HW organization shown in Fig. 5. Figure 5a is the conventional HW organization [13] and Fig. 5b is that of the proposed one.

The existing data path is shown in Fig. 5a. The input image is read from external memory, and the edge strength image obtained through former processing is written to a FIFO, first-in-first-out, buffer. The FIFO buffer allows the received data to be passed directly to subsequent processing. The PF, pixel feeder, is a HW module compensating for the lack of pixels. The intermediate image across each stage is shaved by the inherent algorithm, making the memory access stream style for efficient HW generation. The role of PF expands the image shaved back to the original size of the image. In former processing and later processing, the position of the output image after processing is shifted to the lower right corner, so the PF must restore it. The role of MemStore is to output the image compensated continuously into the external memory. The SW description following this HW organization has been converted to the ideal pipelined HW module with one output per one clock.

In this study, we propose a method of insertion into the data path that does not affect the flow of the existing pipeline. The proposed data path is shown in Fig. 5b. Figure 5b is represented in the program as shown in Fig. 6. In the HW configuration, the input image for pencil drawing and the input image for colorization are read from different physical ports [14]. Pencil pixels " pen_gray_img " obtained from src1 and " org_col_img " (scr2) are overlapped by the Coloring function shown in Fig. 4. On the scr1 side, the processes are executed starting with the first process. There is a FIFO buffer between processes, and this buffer is set by pragma. Comparing with Fig. 5a, we can see that similar data path is realized.

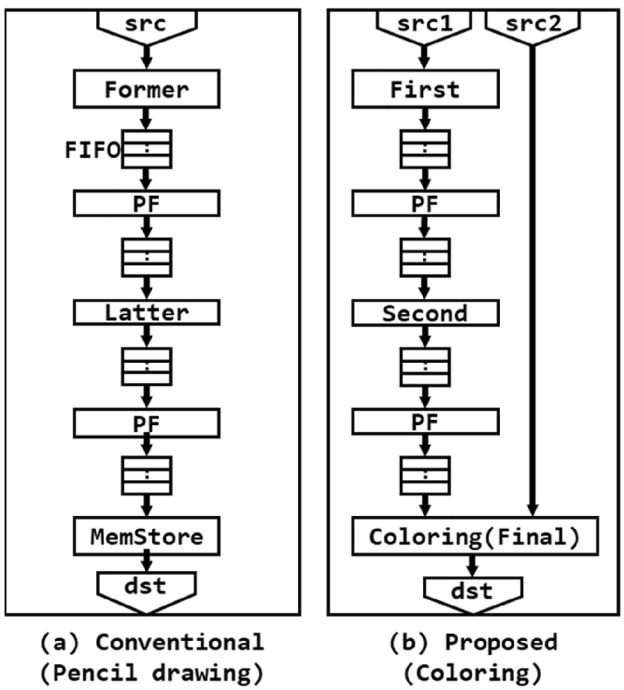


Fig. 5 Conventional data path and proposed data path

3.4 Execution snapshot of pipelined hardware

Figure 7 shows a snapshot of the execution of existing gray-scale HW. As explained in Sect. 2 using Fig. 1, the former process with the Sobel filter and the latter process with the line convolution perform the window processing. Since the output of window-level processing is transferred to memory non-continuously, the HLS tool cannot infer burst transfers, causing a significant performance loss. To make this transfer continuous, a memory access streaming technique is applied [10]. Figure 7a shows this continuous processing. A virtual window is assumed and slides over the input image using the raster scan method. Here, the window size is assumed to be 3×3. Although the virtual window contains an invalid pixel placing at the outside of the input image, the memory streamer is generating the output pixel. In this area, the output image is invalid, that is to be partially shaved. The memory streamer starts to output correct output pixels after the whole of the virtual window enters the input image. This technique can make HLS generate a pipelined HW module with 1 output / 1 clock, but the image shaving remains as a side effect.

```
void Pencil(u32 *src1, u32 *src2, u32 *dst, u16 w, u16 h)
{
#pragma HLS INTERFACE mode=m_axi bundle=d0 port=src1
#pragma HLS INTERFACE mode=m_axi bundle=d1 port=src2
#pragma HLS INTERFACE mode=m_axi bundle=d0 port=dst

   static u64 md1[IMG_SIZE], md2[IMG_SIZE];
   static u8  md3[IMG_SIZE], md4[IMG_SIZE];

#pragma HLS DATAFLOW
#pragma HLS STREAM depth=16 type=fifo variable=md1
#pragma HLS STREAM depth=16 type=fifo variable=md2
#pragma HLS STREAM depth=16 type=fifo variable=md3
#pragma HLS STREAM depth=16 type=fifo variable=md4

   PencilFirst (src1, md1,      w, h);
   PF64        (md1 , md2,      w, h);
   PencilSecond(md2 , md3,      w, h);
   PF8         (md3 , md4,      w, h);
   Coloring    (src2, md4, dst, w, h);
}
```

Fig. 6 Top function realizing whole processing flow

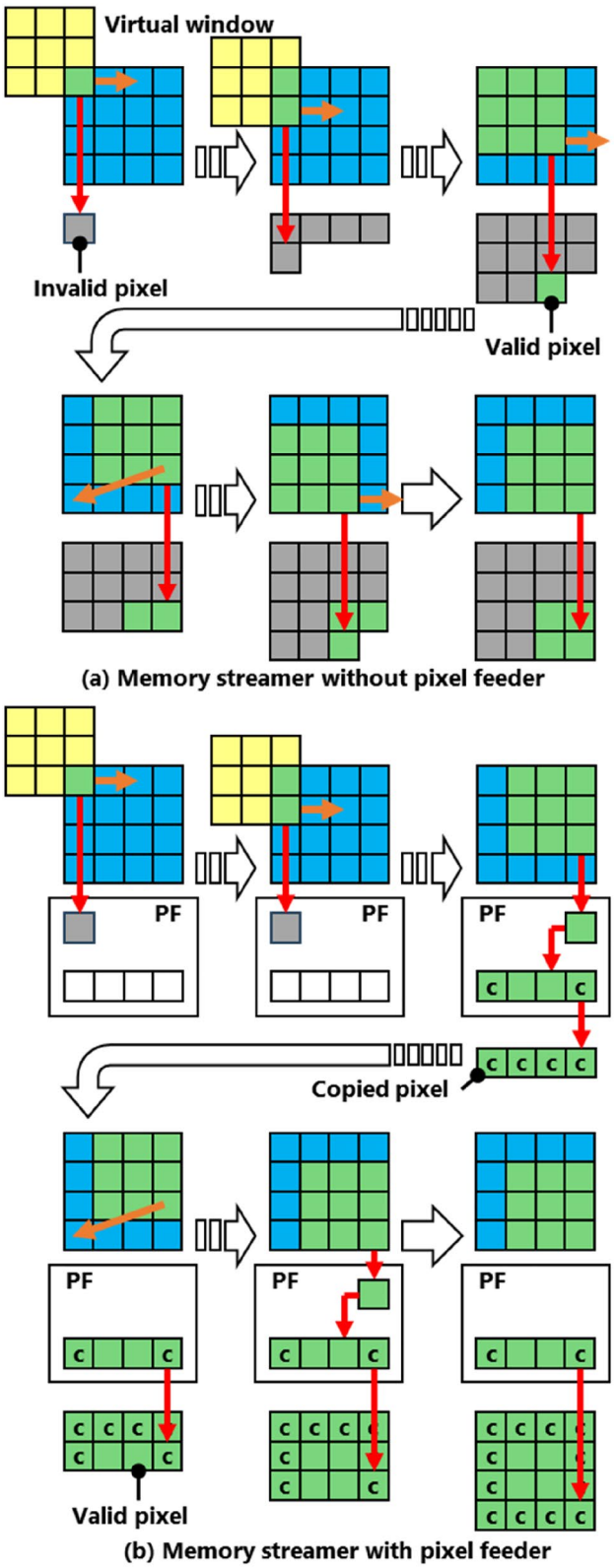


Fig. 7 Execution snapshot of conventional hardware

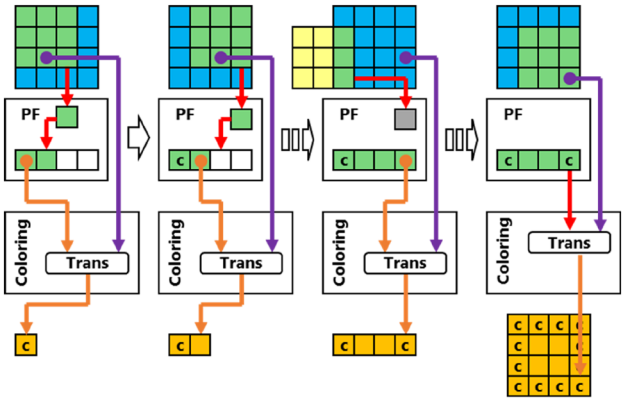


Fig. 8 Execution snapshot of proposed hardware

To prevent such image shaving, we have proposed the insertion of the pixel feeder between each process as shown in Fig. 5. The PF briefly copies the valid pixels to neighbor invalid pixels on its line buffer. The memory output is performed continuously by storing the compensated pixels on the buffer sequentially into the memory. This is shown in Fig. 7b. So, the HLS can generate the straight pipeline data path from the memory input to the output without any pipeline stall.

Figure 8 shows an execution snapshot of the proposed HW with coloring. The coloring HW gets the sequential pencil pixels from the PF continuously. In parallel, the coloring HW can get the original color pixel from the individual physical port accessing the input image on the memory. The transmitted final pixel goes to the memory continuously. This pipelined operation indicates that although our HW expansion inserts the coloring process, the HLS may realize an ideal pipelined data path with 1 output/1 clock.

4 Experiments and discussion

We used a high-level synthesis tool of Xilinx Vitis HLS 2022.2. The SW program was converted into a HW behavior in VHLD of HW description language (HDL). The generated HW behavior was converted into circuit data for writing to an FPGA by Xilinx Vivado 2022.2. The FPGA was a Xilinx Zynq-7000, and the FPGA board ZYBO Z7 from DIGILENT was used to perform demonstration experiments on an actual machine. The CPU of the PC is Intel Core i5. A display was connected to the HDMI port on the FPGA board for visual confirmation.

The images used in the experiment are shown in Fig. 9. The image size is 1280 width × 720 height. In this study, the window size of the Sobel filter and the line segment is 3 × 3.

Fig. 9 Input image (W1280 × H720)

4.1 Output image

In this paper, we propose an additional method of colorization processing without performance loss, but the output image should also be examined to confirm the effectiveness of this processing. The colorization process in this study can change the atmosphere of the output image by changing the threshold shown in Fig. 3 in the range of 0 to 255. The base threshold value is TH = 100, TH1 = 100, and TH2 = 150, and each threshold value is changed. The output images obtained by varying thresholds are shown in Fig. 10.

The more significant the TH is, the more clearly the outline of the pencil image is drawn. However, if TH is too large, even the near-white areas of the pencil pixels are output, making the output image appear rough. If TH1 is close to 0, the output image becomes paler; if TH2 is close to 255, the output image becomes more distinct.

4.2 Circuit size

The circuit size was measured using reports generated by Vitis HLS. Figure 11 shows this experimental result. Here, the numbers of LUTs, D flip-flops, and the FPGA embedded memories, BRAMs are shown.

From Fig. 11, the number of LUTs and FFs increases for the proposed HW compared to the conventional HW. This is because the Color function contains many multiplications. The number of BRAMs is almost equal. This is because there is no difference in the number of FIFO buffers used in the data path among the conventional HW and the proposed one, as shown in Fig. 5.

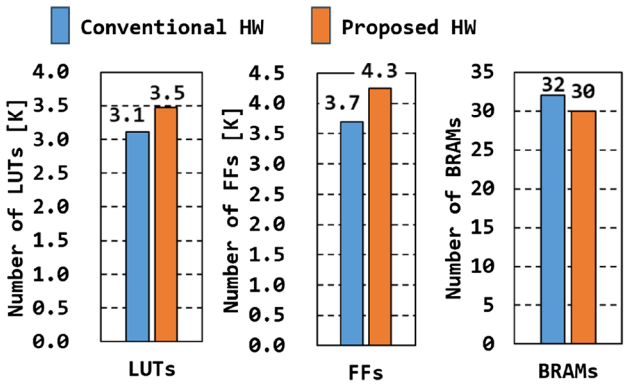
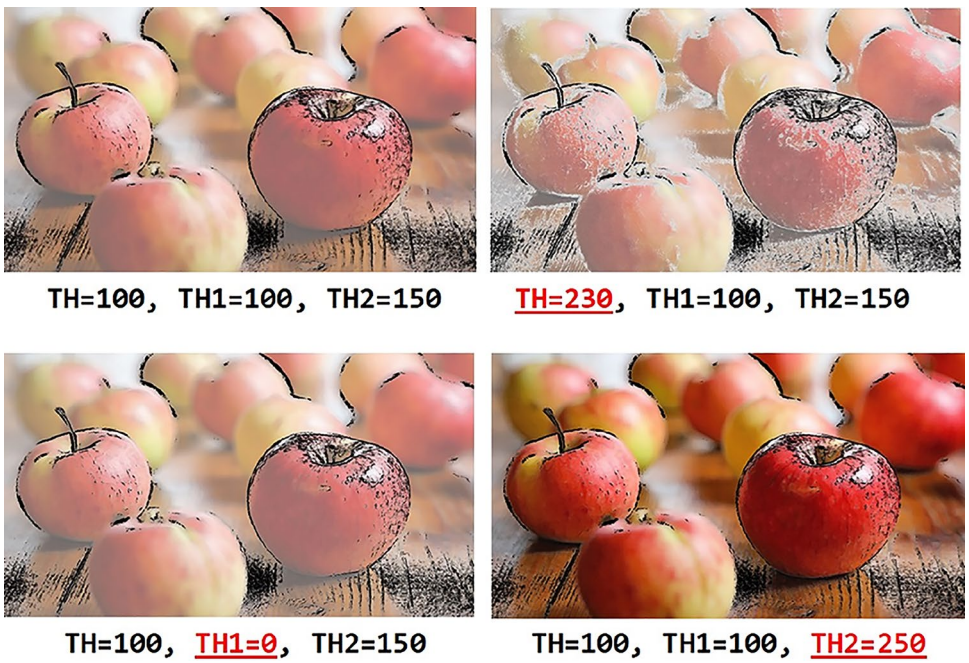


Fig. 11 Circuit size

Fig. 10 Output image



4.3 Execution time

HW execution time is measured by running on an FPGA. The SW execution time is also measured on a PC to be compared with the HW execution time. The following equation gives the execution time. The clock frequency of the CPU on the PC is 3.7 GHz, and that of the FPGA is 100 MHz

$$\text{Exec. time (ms)} = \frac{\text{Total number of clocks (clks)}}{\text{clock frequency (Hz)}}. \tag{4}$$

The measured execution time is shown in Fig. 12. As shown in Fig. 9, the total number of pixels in the image used in this study is 921,600. Therefore, the ideal HW execution time with 1 output data/1 clock is 9.216 ms.

The execution times for the proposed HW and the conventional HW were equal. The proposed HW can achieve the same performance of the conventional HW although the proposed HW is expanded by embedding the coloring process compared with the conventional one. This fact indicates that our strategy shown in Fig. 5 not to intervene the pipeline execution has been successfully accomplished.

4.4 Power efficiency

The following equation defines the power efficiency of HW compared to SW on a PC.

$$\text{Power efficiency}_{\text{without HW resource}} = \frac{\text{SW exec. time (s)}\times F_{\text{CPU}}(\text{Hz})}{\text{HW exec. time (s)}\times F_{\text{FPGA}}(\text{Hz})}. \tag{5}$$

Power efficiency is also calculated considering the circuit size.

$$= \frac{\text{Power efficiency}_{\text{with HW resource}}}{\text{SW exec. time (s)}\times F_{\text{CPU}}(\text{Hz})} \cdot \frac{\text{Amount(HW)}}{\text{Amount(HWref)}}. \tag{6}$$

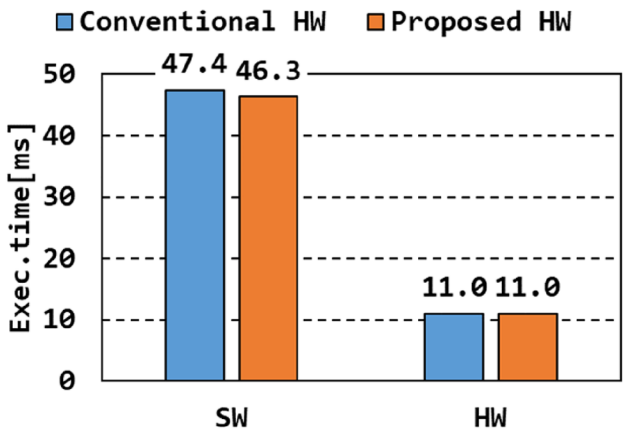


Fig. 12 Execution time

The amount of HW is calculated using the number of truth tables, LUTs, D flip-flops, FFs, and embedded RAMs, BRAMs. In this paper, conventional HW is used as the reference HW.

$$= \frac{\text{LUT(HW)}}{\text{LUT(HWref)}} \times \frac{\frac{\text{Amount(HW)}}{\text{Amount(HWref)}}}{\frac{\text{FF(HW)}}{\text{FF(HWref)}}} \times \frac{\text{BRAM(HW)}}{\text{BRAM(HWref)}}. \tag{7}$$

The power efficiency of the HW compared to the SW on the PC is shown in Fig. 13.

The colorization power efficiency was about 20% less efficient than the conventional power efficiency. This is due to the increase in circuit size, although the performance is the same. However, even with the colorized HW, the performance improvement of 4.6 times and the power efficiency of 130 times compared to SW are considered enough.

5 Conclusion

In this paper, we developed colorization HW for pencil-drawing-style image conversion using HLS and compared its performance with the existing pencil drawing HW. As a result, although the overall circuit size increased, the power efficiency considering the circuit size also showed enough performance to indicate no problem. The proposed HW execution time was close to the ideal value calculated from the input images. From the above, we were able to develop efficient colorization HW using HLS.

In this paper, colorization was performed using a relatively simple algorithm, and the atmosphere of the output image was arbitrarily changed. In the future, we would like to develop HLS HW for other colorization methods and

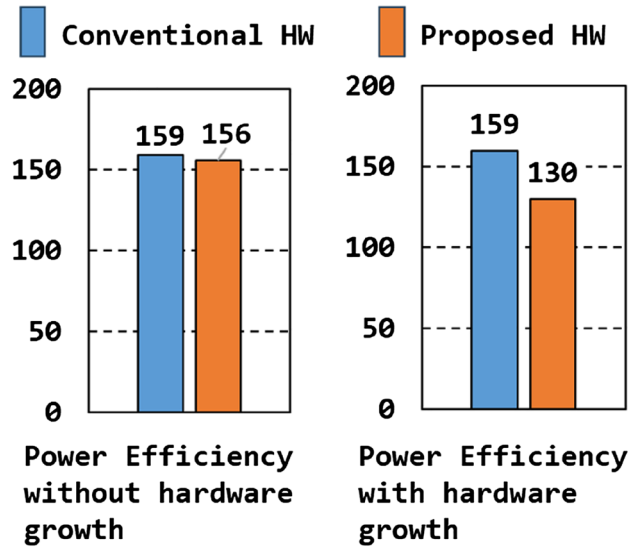


Fig. 13 Power efficiency compared to software execution on PC

compare their performance. Finally, we plan to perform real-time processing using a camera.

Data availability The data that support the findings of this study are available from the corresponding author upon reasonable request.

References

1. Kumar MPP, Poornima B, Nagendraswamy HS et al (2019) A comprehensive survey on non-photorealistic rendering and benchmark developments for image abstraction and stylization. Iran J Comput Sci 2:131–165
2. Cewu L, Li X, Jiaya J (2012) Combining sketch and tone for pencil drawing production. In: Proceedings of international symposium on non-photorealistic animation and rendering 2012, pp 65–73
3. Younes H, Ibrahim A, Rizk M, Valle M (2021) Algorithmic-level approximate tensorial SVM using high-level synthesis on FPGA. Electronics 10(2):205. https://doi.org/10.3390/electronics10020205
4. Sjövall P, Lemmetti A, Vanne J, Lahti S, Hämä-läinen TD (2022) High-level synthesis implementation of an embedded real-time HEVC intra encoder on FPGA for media applications. ACM Trans Des Autom Electron Syst 27(4):1–34 https://doi.org/10.1145/3491215 **(Article No.: 35)**
5. Akgün G, Khan H, Hebaish M, Elshimy M, Ghany MAAE, Göhringer D (2020) SysIDLib: a high-level synthesis FPGA library for online system identification. In: Applied reconfigurable computing. Architectures, tools, and applications. ARC 2020. Lecture notes in computer science, vol 12083. https://doi.org/10.1007/978-3-030-44534-8_8
6. Nane R, Sima V-M, Olivier B, Meeuws R, Yankova Y, Bertels K (2012) DWARV 2.0: a CoSy-based C-to-VHDL hardware compiler. In: 22nd international conference on field programmable logic and applications (FPL), pp 619–622. https://doi.org/10.1109/FPL.2012.6339221
7. Ferrandi F et al (2021) Invited: Bambu: an open-source research framework for the high-level synthesis of complex applications. In: 2021 58th ACM/IEEE design automation conference (DAC), pp 1327–1330
8. Özkan MA et al (2020) AnyHLS: high-level synthesis with partial evaluation. IEEE Trans Comput Aided Des Integr Circuits Syst 39(11):3202–3214
9. Mousouliotis PG, Petrou LP (2020) CNN-grinder: from algorithmic to high-level synthesis descriptions of CNNs for low-end-low-cost FPGA SoCs. Microprocess Microsyst https://doi.org/10.1016/j.micpro.2020.102990
10. Tani H, Yamawaki A (2023) Memory access optimization for former process of pencil drawing style image conversion in high-level synthesis. In: Parallel and distributed computing, applications and technologies (PDCAT 2022), lecture notes in computer science, vol 13798, pp 57–68
11. Tani H, Yamawaki A (2023) Effect of line segment size on pencil drawing-like image conversion hardware developed by high-level synthesis. In: Proceedings of the 28th international symposium on artificial life and robotics 2023 (AROB 2023), pp 740–743
12. Tani H, Yamawaki A (2023) Process integration to realize full pipelined pencil drawing style image conversion hardware in high-level synthesis. In: 2023 5th international conference on computer communication and the internet (ICCCI), Fujisawa, Japan, 2023, pp 251–255. https://doi.org/10.1109/ICCCI59363.2023.10210175
13. Tani H, Yamawaki A (2023) Process chaining without image and performance loss for pencil drawing style image conversion in high-level synthesis. In: The 10th IIAE international conference on intelligent systems and image processing 2023. To appear
14. Yamasaki M, Yamawaki A (2020) Duplicating same argument of function to realize efficient hardware for high-level synthesis. Artif Life Robot 25(2):248–252. https://doi.org/10.1007/s10015-019-00576-4