Development of Histogram Equalization Hardware By High-level synthesis

Akira Yamawaki^{a,*}, Makoto Miyauchi^b and Seich Serikawa^a

^aKyushu Institute of Technology, 1-1 Sensui, Tobata, Kitakyushu 804-8550, Japan

^bNational Institute of Technology, Kitakyushu College, 5-20-1 Shii, Kokuraminami, Kitakyushu 802-0985, Japan

*Corresponding Author: yama@ecs.kyutech.ac.jp

Abstract

High-level synthesis is a technology that converts a software program into hardware. High-level synthesis can greatly reduce the load of hardware design and development. An FPGA is a device that can implement any digital hardware at any time. The combination of high-level synthesis and FPGA has been attracting attention in the field of embedded devices where the life cycle of products is short and the demand for power-saving and high-performance products is high. High-level synthesis cannot convert pure software that describes the algorithm intuitively into high-performance and low-power hardware. High-level synthesis requires an input program that considers the hardware configuration in order to generate good hardware. In this paper, we develop hardware for histogram equalization to improve image contrast using high-level synthesis. In the description of the program, in addition to general methods such as fixed-point conversion, the description method we have developed is applied. Experimental results show that the histogram equalization hardware can achieve 2.3 times performance improvement compared to software implementation.

Keywords: high-level synthesis, HLS, buffer, overlap

1. Introduction

High-level synthesis is a technology that converts a software program into hardware. High-level synthesis can greatly reduce the load of hardware design and development. An FPGA is a device that can implement any digital hardware at any time. The combination of high-level synthesis and FPGA has been attracting attention in the field of embedded devices where the life cycle of products

is short and the demand for power-saving and high-performance products is high. High-level synthesis cannot convert pure software that describes the algorithm intuitively into high-performance and low-power hardware. High-level synthesis requires an input program that considers the hardware configuration in order to generate good hardware.

In this paper, we develop hardware for histogram equalization ⁽¹⁾ to improve image contrast using high-level synthesis. A variety of studies have been conducted on hardware development using high-level synthesis ⁽²⁻⁵⁾. However, there is no research that has studied in detail how to write a program to generate better histogram equalization hardware with a high-level synthesis tool.

The rest of this paper is organized as follows. Section 2

Fig.1. Flow Chart of Histogram Equalization

```

HistGen(u8 iy[], u32 hist[]){
    for(i=0;i<256;i++){hist[i]=0;
    for(i=0;i<W * H;i++){hist[iy[i]]++;
    }
}

```

Fig.2. Luminance histogram generation

outlines histogram equalization. Section 3 describes the program description methods introduced in the hardware development of histogram equalization. Section 4 evaluates the performance and power efficiency. Finally, Section 5 concludes this paper.

2. Histogram Equalization

2.1 Flow Chart

Histogram equalization is one of the methods to improve the brightness contrast of an image. Images with poor brightness contrast are blurred and difficult to see. This is because the brightness value of the image is biased to a certain local area. Histogram equalization is an algorithm that can optimally distribute biased luminance variance. That has been proved mathematically ⁽¹⁾.

Fig. 1 shows a flowchart of histogram equalization. The input and output images are gray images. A luminance histogram is generated from the input gray image. A cumulative luminance histogram is generated from the luminance histogram. The cumulative luminance histogram is normalized so that each value is between 0 and 255. The normalized cumulative luminance histogram is indexed by the pixel value of the input gray image. The value of the indexed entry is output as the converted luminance value. As a result, an output gray image in which the luminance values are better dispersed is obtained.

2.2 Luminance histogram generation

The luminance histogram shows the number of pixels with the same luminance value in the image. Figure 2 shows the concept of generating a luminance histogram from a gray image and pseudo-code based on C language.

First, the luminance histogram is initialized to zero.

Fig.3 Example of luminance histogram and its cumulative one

Then, the histogram entry indexed by the pixel value of the input gray image is incremented. This operation is performed on the entire input gray image.

2.3 Normalized cumulative luminance histogram

The normalized cumulative luminance histogram (NCHG) created from the luminance histogram is used to implement a transformation function to improve the variance of luminance values. Fig. 3 shows an example of a cumulative luminance histogram. Fig. 3 shows that the luminance histogram has a bias around the center. The cumulative luminance histogram is a graph in which the values of the luminance histogram are cumulatively added in the horizontal axis direction. Due to its characteristics, the cumulative luminance histogram gradually increases monotonically, and rises sharply in the part where the luminance histogram is biased. Eventually, the cumulative luminance histogram reaches the total number of pixels.

The shape of the cumulative histogram is suitable as a luminance conversion function. Fig. 4 shows the concept of

Fig.4 Luminance transform using Normalized cumulative luminance histogram

$$NChist[j] = \frac{255}{W \times H} \sum_{i=0}^j hist[i]$$

```

NChistGen(float NChist[], u32 hist[])
{
    NChist[0] = hist[0] * 255 / (W * H);
    for( i = 1; i < 256; i++ )
        NChist[i] = NChist[i-1]
            + hist[i] * 255 / (W * H);
}

```

Fig. 5 Normalized cumulative luminance histogram generation

this reason. The NCHG with values from 0 to 255 can be obtained by dividing the cumulative luminance histogram by the total number of pixels and multiplying by the maximum luminance value. NCHG becomes the conversion function. The brightness value of the input image is mapped to the brightness value of the output image by NCHG. Then, the part where the luminance value of the input image is biased is dispersed by the steep part of the conversion function. As a result, an output image with optimally distributed luminance values is obtained.

2.4 NCHG generation

Figure 5 shows pseudo code for creating a normalized cumulative luminance histogram (**NChist** []) from the luminance histogram (**hist** []). *W* represents the width of the input image and *H* represents its height. Each element of the luminance histogram is divided by the image size $W \times H$ each time it is cumulatively added. As a result, the range of the value of the cumulative luminance histogram is 0 to 1.0. Furthermore, the value of the cumulative luminance histogram is multiplied by 255. Therefore, the range of the value of the normalized cumulative luminance histogram is 0 to 255.

2.5 Luminance transformation

Using the NCHG obtained in the previous section, the biased luminance value of the input gray image is converted

$$s = T(r) = NChist[r]$$

```

LumTrans(float NChist[], u8 iy[], u8 oy[])
{
    for(i=0; i < W*H; i++)
        oy[i] = NChist[iy[i]];
}

```

Fig. 6 Luminance Transformation

```

PureHistEq(u8 iy[], u8 oy[], u32 W, u32 H)
{
    u32 hist [256];
    float NChist[256];

    HistGen ( iy, hist, W, H );
    NChistGen( hist, NChist );
    LumTrans ( iy, oy, NChist );
}

```

Fig. 7 Pure histogram equalization

to a more dispersed new luminance value. Fig. 6 shows the pseudo code for luminance conversion.

The array of normalized cumulative luminance histogram (**NChist** []) is indexed by the luminance value of the input image. The value output from the indexed entry is the converted luminance value. After the above operations are performed on the entire image, a clear output gray image is obtained from the blurred input gray image.

2.6 Pure histogram equalization

By arranging the above processes, histogram flattening that is intuitive to the algorithm can be obtained. Fig. 7 shows the program list of the pure histogram equalization. We need to reconstruct this pure program list, considering the hardware configuration.

3. Restructure for high-level synthesis

3.1 Fixed point formatting

If your program contains floating-point arithmetic, high-level synthesis may convert it to large-scale hardware. In addition, floating-point operations are implemented as device-specific IP cores. Therefore, the hardware after conversion is device-dependent and lacks versatility. Therefore, the function containing floating-point operations, **NChistGen**, is reconstructed into a fixed-point program

```

#define K 255 // 255/(256*256) * 216
NChistGenFix(u64 NChist[], u32 hist[])
{
    NChist[0] = hist[0] * K;
    for( i = 1; i < 256; i++ )
        NChist[i] = NChist[i-1]
            + hist[i] * K;
}

LumTransFix(u64 NChist[], u8 iy[], u8 oy[])
{
    for(i=0; i < W*H; i++)
        oy[i] = NChist[iy[i]] >> 16;
}

```

Fig. 8 Fixed point formatting

as shown in the upper part of Fig. 8. The image size is 256×256 .

During fixed-point conversion, we converted the floating-point array of the normalized cumulative luminance histogram to a 64-bit integer. In the fixed-point format, the lower 16 bits are the fraction part and the upper 48 bits are the integer part. The normalization coefficient **K** was fixed-point as $255 / (256 \times 256) \times 216 = 255$. In addition, the transformation function, **LumTrans**, which uses the fixed-point normalized cumulative luminance histogram, was also reconstructed as shown in the lower part of Fig. 8. Finally, the lower 16 bits of the fixed-point number are removed to extract only the integer part.

3.2 Port duplication to same argument

Looking at the list in Fig. 7, two sub functions use the arguments of the input gray image. More specifically, **HistGen** and **LumTrans** use **iy**. The high-level synthesis tool converts the arguments of the array into physical ports for memory access. When multiple functions access the same argument, the high-level synthesis tool inserts a synchronization mechanism that waits for access to the physical port to avoid contention for a single memory access port. As a result, memory access becomes sequential and the performance drops significantly⁽⁵⁾.

Fig. 9 is the pseudo code that duplicates the arguments. High-level synthesis can generate hardware that can access memory continuously without arbitration mechanism of port access. As a result, high-performance hardware is generated⁽⁵⁾.

4. Experiment and Discussion

To perform the experiment on the real machine, we have developed the hardware platform. The used FPGA is Xilinx ZYNQ FPGA. The used FPGA board is Digilent ZYBO. The embedded processor of ARM Cortex-A9 runs at 650MHz. The hardware modules run at 100MHz.

The software program of the proposed description method shown in Section 3 was converted to hardware (HDL program) by Vivado HLS 2018.3 of Xilinx HLS tool. The generated HDL program was implemented on the FPGA board Digilent ZYBO (Zynq-7000 development board), and the hardware was evaluated. The size of the input grayscale image is 256×256 pixels.

We inserted the pragmas for memory access ports and pipelines into Figure 9 as appropriate.

The software execution time of CortexA9 was 3.0 [ms],

and the execution time of high-level synthesis hardware was 1.3 [ms]. The performance improvement was $3.0 / 1.3 = 2.3$ times. The power efficiency reached $650 \text{ [Mhz]} / 100 \text{ [MHz]} \times 2.3 = 15$ times.

5. Conclusions

High-level synthesis is a technology that automatically converts software programs into hardware. High-level synthesis cannot convert pure software that describes the algorithm intuitively into high-performance and low-power hardware. In this paper, we developed hardware for histogram flattening to improve image contrast using high-level synthesis. At that time, the description method of the program in consideration of the hardware was applied. The high-level synthesis hardware achieved a performance improvement of 2.3 times compared with software execution. The power efficiency was 15 times.

References

- (1) Rafael C. Gonzalez and Richard E. Woods, "Digital Image Processing Third Edition," Pearson Prentice Hall, pp. 743-747, 2008.
- (2) B. Vaidya, M. Surti, P. Vaghasiya, J. Bordiya, and J. Jain, "Hardware acceleration of image processing algorithms using Vivado high level synthesis tool", In Proc. of 2017 International Conference on Intelligent Computing and Control Systems (ICICCS), pp. 29-34. 2017.
- (3) M. Weinhardt, "Comparing Register-Transfer-, C-, and System-Level Implementations of an Image Enhancement Algorithm", Applied Reconfigurable Computing, Springer International Publishing, pp. 245--257, 2016.
- (4) Moena Yamasaki and Akira Yamawaki, "Description Method for High-level Synthesis of Histogram Generation and Their Evaluation," IEIE Transactions on Smart Processing and Computing, vol. 8, no. 3, pp.178-185, June 2019.
- (5) Moena Yamasaki and Akira Yamawaki: "Duplicating same argument of function to realize efficient hardware for high-level synthesis", Artificial Life and Robotics, DOI:10.1007/s10015-019-00576-4 (2019.11)