High-level Synthesis Oriented Describing Method of Template Matching

Miyu Fujita^{a,*}, Kazunari Yosikawa^a, Naohiro Iwanaga^a, Akira Yamawaki^{a,*}

^aKyushu Institute of Technology, 1-1 Sensui-cho Tobata-ku Kitakyushu-shi Fukuoka-ken, 804-8550, Japan

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

Abstract

High-level synthesis (HLS) is the technique that automatically convert the software to the hardware. However, hardware is not produced with excellent performance and circuit scale if software is not described in consideration of the hardware. In this paper, we create HLS oriented template matching function that generates optimal hardware. In the experiment, we perform HLS pure software program and the newly created program, and compare those circuit scale and processing speed. Further, we compare processing speed of hardware processing and software processing. Thereby, we confirm the usefulness of HLS oriented template matching function

1. Introduction

Recently, Implementation hardware of systems is advancing along with performance improvement of electronic devices that perform moving image processing. By implementation hardware, some merit are estimated, such as speeding up by pipelined processing, low power consumption and reduction of CPU load. We explain the general flow of hardware into the following. First, description by high-level language and operation verification. Next, description by hardware description language based on them and operation verification. Finally, logic synthesis and implementation. In this way, it is necessary to do twice inconsistent description and operation verification by hardware description language and high-level language. So it takes a lot of effort and time. Therefore, HLS is used. HLS is the technique that creates programs described by hardware description language from programs described by high-level language. By using HLS, description by hardware description language is eliminated and consistent operation verification becomes possible.

In software development, useful specific moving image

processing functions such as template matching exist in library such as OpenCV. However, those are not described in consideration of hardware. For example, memory accesses is not a simple streaming access, so that optimal pipelining is not performed. And, those includes a large number of processing that is not suitable for hardware, such as floating-point arithmetic so that many resources are needed. From the above, practical hardware is not generated, even if HLS is processed.

On the other hand, the hardware implementation of template matching has been variously studied, such as the custom hardware on the ASIC and hardware module described in a hardware description language⁽¹⁻⁵⁾. But those are not possible to use on high-level language level in a unified manner.

In this study, we develop the template matching function that can be used on software and can generate optimal hardware by processing HLS. In the experiment, we perform HLS pure software program and the newly created program, and compare those circuit scale and processing speed. Further, we compare processing speed of hardware processing and software processing. Thereby, we confirm the usefulness of HLS oriented template matching function

2. Template Matching

In this section, we describe about the algorithm of the template matching that we used in this study. This function has two inputs of input image and template image. First, two input images are converted to the gray scale images. Then the template matching is performed by using gray scaled images. According to matching result, the binary image is output that the matched pixel is '1' and unmatched pixel is '0'.

2.1 Gray Scaling

To make the gray scaling, we used the NTSC weighted average method. This method performs a predetermined

weighting to RGB elements and takes the average as Equation (2.1).

$$I = 0.298912 \times R + 0.58611 \times G + 0.114478 \times B \quad (2.1)$$

The I which represents a luminance value is used as the gray scaled value.

2.2 Template Matching

As shown in Figure 2.1, the template image slides over the input image. At each slide, the similarity degree is calculated every a pixel on the template image and input image covered by the template image.

Fig.2.1 template matching

We used the sum of absolute difference (SAD) to calculate the similarity degree as shown in Eq. 2.2.

$$SAD = \sum_{x=0}^{X-1} \sum_{y=0}^{Y-1} |I(x, y) - T(x, y)| \quad (2.2)$$

The $I(x, y)$ represents a luminance value of each coordinate of the input image when the most upper left of partial overlapping the template image is $(0, 0)$. The $T(x, y)$ represents a luminance value of each coordinate of the template image. The X is the width of template image, and the Y is the height of that. When the SAD between the template image and input image is little, the input image and template image has much higher similarity. If the similarity degree is below the threshold previously set, '1' representing the match is outputted. If the SAD exceeds the threshold, '0' representing the unmatched is outputted.

3. To HLS Oriented Description

3.1 Targeted Hardware Overview

In this section, we describe the hardware structure of template matching to be generated by the HLS. As shown in Figure 3.1, the grayscale processing and the template matching process are divided into two hardware modules that run in parallel. Each hardware module is pipelined. Two functions are connected with FIFO. The former process outputs the gray scaled pixel to FIFO pixel-by-pixel. The latter process acquires the gray scaled pixel from FIFO to perform the template matching and outputs the binary pixel every clock cycle. Thus, the processing of the entire image will be completed by the number of clocks that is same as the number of the total pixels.

Fig.3.1 Hardware structure

3.2 Optimization of memory access

In the template matching, in order to compare the template and the original image, it is necessary to access some region of the memory corresponding to the size of the template. However, there is only one memory port in the hardware. By the straight forward description accessing several elements on the array in the pure software, the pipelining is inhibited by the resource conflict to the memory port. To achieve pipelining as described in Section 3.1, the memory access must always be the one by one pixel.

Figure 3.2 shows the flow of processing for each one pixel, when size of input image assumed 5×5 and size of template image assumed 3×3 . We prepared the buffer which has the width same as input image and height same as template image, and window which has the size same as template image. The $a \sim j$ in buffer are undefined value that initially reside the buffer. First, as ①, target pixels are input to the bottom line of the same column as the target pixel of the buffer (the target column). Then, as shown in ②, target

Fig.3.2 processing for memory access serialization

column in the buffer are input to the right-most column of the virtual window. At that moment, all of value in the virtual window move to left, as of ③. When the line of the target pixel moves to the next line, all of value in the buffer move upward, as of ④. Repeating this operation, when the virtual window was filled all as ⑤, performing similarity calculation and matching decision as ⑥. And, If matching '1', if unmatching '0' are mapped to the target coordinates of

the output as ⑦.

3.3 Programming

Figure 3.3 shows the overview of the program list for the HLS realizing the processing mentioned above.

Some parameters are statically declared as follows.

Inheight: Input image's height

Inwid : Input image's width

tmphei : template image's height

tmpwid : template image's width

Size : inheight * inwid

thre : threshold of matching

Function (1) is the main function of the template matching functions. The input arguments (intmp) is the array holding continuously the template image and the input image. The output argument (outg) is the binary map of '0' or '1' as the matching information. The process (1.1) performs the function (2) to perform a gray-scaling. The process (1.2) performs the function (3) acquiring the template image and performing the template matching.

Function (2) is a function that performs gray-scaling. It extracts each of the RGB values from the input data, and calculates the brightness value for each pixel. In this time, we multiplied by 6553.7 (2^{16}) to each coefficient of Equation (2.1) and performed 16bit right shift calculated value. By doing so, operations become fixed-point arithmetic, and number of processing in hardware implementation is reduced.

Function (3) is the function that acquires the template image and performs template matching. First, the template image data are put into the tp at processing (3.1). Brightness value of the target pixel is stored in the variable pix at process (3.2). The contents of the buffer is pushed onto one line as of ④ in Figure 3.3 at process (3.3). Also, the reference column on the buffer is stored in the intermediate variable pixel between buffer and windows. Target pixel is placed in the buffer and pixel as ① at process (3.4). The contents of the window is shifted to one column left as ③ at process (3.5). Pixel that contains the reference column of the buffer is placed in the right-most column of the window as ② at process (3.6). Then, function (4) is performed, and similarity calculation is performed at process (3.7).

The similarity 'simi' and the threshold 'thre' are compared and output is determined to '1' or '0' at process (3.8). Target memory is advancing one at process (3.9).

Function (4) is a function of the similarity calculation. The difference of the luminance values of corresponding pixels in the template image and the window is calculated at process

Function(1)

```
void top( int intmp[innum], int outg[size]) {
    int ingray[ innum ];
    (1.1) gray ( intmp , ingray );
    (1.2) match( ingray, outg );
}
```

Function(2)

```
gray(int in[innum], int out[innum]){
    int i, pix,simi;
    unsigned char r, g, b;
    for(i=0 ; i < innum ; i++){
        pix = in[i];
        r = pix;
        g = pix >> 8;
        b = pix >> 16;
        y = 19595*r + 39470*g+ 7471*b;
        out[i] = y >> 16;
    }
}
```

Function(3)

```
match(int in[size], int out[size]){
    int tp[tmphei][tmpwid]; //template image
    int i, j, k, l, ptr, num = 0,pix;
    static int buf[tmphei][inwid ]; //buffer
    static int win[tmphei][tmpwid]; //window
    static int pixel[tmphei];

    num = 0;
    (3.1) for(i=0 ; i < tmphei; i++ )
        for(j=0; j < tmpwid; j++ )
            tp[i][j]=in[num++];

    for(i=0 ; i < inheight ; i++){
        for(j=0 ; j < inwid ; j++){
            (3.2) pix = in[num];
            (3.3) for(k=0; k<tmphei-1; k++){
                buf[k][j] = buf[k+1][j];
                pixel[k] = buf[k][j];
            }
            (3.4) pixel[tmpwid - 1]=buf[tmpwid - 1][j]=pix;
            (3.5) for(k = 0; k<tmphei ; k++)
                for(l=0; l<tmpwid-1; l++)
                    win[k][l]=win[k][l+1];
            (3.6) for(k=0; k<tmphei; k++)
                win[k][tmpwid-1] = pixel[k];
            (3.7) similar (win,tmpgray,simi);
            (3.8) if(simi <= thre){out[num]=1;}
            (3.9) else{out[num]=0;}
            num++;
        }
    }
}
```

Function(4)

```
similar(int win[tmphei][tmpwid],
int tmpgray[tmphei][tmpwid], int out){
    int i, j,absl, sad=0;
    for(i=0 ; i < tmphei ; i++){
        (4.1) for(j=0 ; j < tmpwid ; j++){
            (4.2) absl = abs(win[i][j] - tmpgray[i][j]);
            sad = sad + absl;
        }
    }
    out = sad;
}
```

Fig.3.3 simplified source code

(4.1). And similarity is calculated by gradually adding them

at process (4.2).

4. Experiment

4.1. Environment method

We performed HLS of existing template matching function and created programs by using the Vivado HLS.2015 of Xilinx, Inc. We evaluated the circuit scale of the hardware module converted from the C program listed in Figure 3.3. In addition, we compared the processing time hardware produced by HLS and software. Hardware processing time calculated by multiplying the number of clocks and clock minimum period derived by vivad HLS. Software processing time was measured using time command of cygwin on windows7. We took the average by measuring 20 times each 3 pattern of image size $64 * 64$, $256 * 256$, $512 * 512$, and calculated the time that is unrelated processing, estimated the actual processing time.

4.2 Experimental result

Table4.1 shows the experimental results for the number of clock and the circuit scales. The conventional program is the case that the pure software without optimization shown in 3.3 is converted by the Vivado HLS. The HLS oriented program is the case that the optimized software as shown in 3.3 is converted. In this experiment, we set the input image size to 64×64 and the template image size to 8×8 .

Table4.1 Comparison of circuit scale and number of clocks

		Conventional program	
circuit scale	LUT	2239	94
	FF	2117	87
number of clocks		810602	4165

For the circuit scale, we succeeded to reducing LUT about 96%, and FF about 95%. For the number of clocks, the conventional program took 810602 clocks which is huge number of clocks. On the other hand, HLS oriented program took 4165 clocks. So number of clocks also can be decreased significantly. The number of pixels of input image is 4096 and the number of pixels of template image is 64, so the number of total input pixel is 4160. In addition, the number of clocks which take for one pixel is 5 clocks. Therefore, it

was an ideal result that number of input pixels $\div$ the number of clocks by pipelining.

Table.4.2 shows processing times of each image size of software (ms). From these results, we calculated that the time that is unrelated processing is 17.72 (ms) and the actual processing time per pixel is about 17.5 (ns). And the minimum clock period of hardware is 4.806 (ns)

Table.4.2 Each processing time of software(ms)

	64 × 64	256×256	512×512
processing time	17.8	19.0	22.1

Table.4.3 Comparison of processing time (μs)

	software	hardware
processing time	77.824	20.017

Table.4.3 shows the comparison of processing time (μs). As shown in Table 4.3, the processing time decrease about 74.28% compared to software. In other words, hardware processing speed is about 1658.3 times of software.

5. Conclusion

This paper shows the description method of the HLS oriented template matching function, which is optimized for the memory access to realize completely pipelining hardware. The experimental results shows that our proposed method can achieve the reduction of the circuit scale about 95%, and the number of clocks about 91% compared with the conventional program without our optimization. By this improvement, our proposal can achieve the speedup of 1658.3 times compared to the software execution.

As future works, we will apply the proposed description method to more sophisticated template matching process like the variable size and angle of the template image.

6. References

- (1) Kaoru Hashimoto, Yasuaki Ito, and Koji Nakano : “Template Matching using DSP slices on the FPGA”, Computing and Networking (CANDAR), 2013 First International Symposium on, pp 338 – 344, 2013
- (2) Hezel, S Kugel, A, Manner, R., and Gavrilu, D.M. : “FPGA-based template matching using distance transforms”, pp 89 – 97, 2002
- (3) Xuejun Liang, Jack Jean, and Karen Tomko : “Data Buffering and Allocation in Mapping Generalized Template Matching on Reconfigurable Systems”, Journal of Supercomputing, Vol.19, (1), pp 77-91, 2001
- (4) Ehsan Norouznezhad, Abbas Bigdeli, Adam Postula, and Brian C. Lovell : “Object tracking on FPGA-based smart cameras using local oriented energy and phase features”, ICDSC '10 Proceedings of the Fourth ACM/IEEE International Conference on Distributed Smart Cameras, pp 33-40, 2010
- (5) Gause, J., Cheung, P.Y.K., and Luk, W. : “Reconfigurable shape-adaptive template matching architectures”, Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on, pp 98 – 107, 2002