Effect of Redundant Function Execution to Reduce Memory Access on High-level Synthesis

Moena Yamasaki, Satoshi Yonemitsu and Akira Yamawaki*

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

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

Abstract

The high-level synthesis, HLS, is a technology converting the software program to the hardware module automatically. The HLS can reduce the design burden of the hardware significantly. However, the HLS tools tend to generate a large and slow hardware generally when deep considerations of the hardware organization at software level are not taken care. This paper demonstrates a software describing method for HLS to reduce the memory access by duplicating the functions redundantly although the software usually eliminates redundant functions. The memory access may effect to the performance of the hardware module more than software. This is because the hardware modules are hard to equip the cache memory which is a rich fabrication on a chip such as the processor only can employ. The experimental result shows that our method can improve the performance than the hardware with more memory accesses, while the software degrades the performance by executing the duplicated functions compared with the software performs more memory accesses.

Keywords: high-level synthesis, memory access, redundant execution, high performance.

1. Introduction

As the market size of image processing embedded products becomes larger, many developers are entering the market. To occupy meaningful share compared with other competitions, providing high-performance and low-power products following the short life cycle is important.

To achieve the high-performance and low-power simultaneously, implementing the high computational processes on the software as the hardware modules is often required. The high-level synthesis, HLS, is a technology

converting the software program to the hardware module automatically⁽¹⁻³⁾. So, the HLS can reduce the design burden of the hardware significantly. That is, the HLS is preferable to chase the short life cycle of the high-performance and low-power image processing embedded products.

However, the HLS tools tend to generate a large and slow hardware generally when deep considerations of the hardware organization at software level are not taken care. As a result, the hardware modules would not meet the requirement of high-performance and low-power.

There are some researches to the software describing methods so that the HLS tool can generate the efficient hardware modules⁽⁴⁻⁶⁾. However, the systematic method that can be employed to all software has never shown. We have to investigate the describing method corresponding to each software.

Focusing on an image processing software, the automatic image binarization based on discriminant analysis⁽⁷⁾, this paper demonstrates a software describing method for HLS to reduce the memory access by duplicating the functions redundantly although the software usually eliminates redundant functions. The memory access may effect to the performance of the hardware module more than software. This is because the hardware modules are hard to equip the cache memory which is a rich fabrication on a chip such as the processor only can employ.

The rest of paper is organized as follows. Section 2 explains the algorithm of the automatic image binarization based on discriminant analysis, Otsu's method. Section 3 shows the describing method of software function for HLS. At first, we depicts the HLS description to the pure software following the algorithm and then shows our method with redundant execution to reduce memory access. Section 4 performs the performance evaluation and discussion. Finally, Section 5 concludes this paper and indicates the future work.

Fig. 1. Main Flow of Automatic Image Binarization

2. Automatic Image Binarization

Fig. 1 is the processing flow of the automatic image binarization using discriminant analysis, Otsu's method⁽⁷⁾.

At first, the input image in the array on the memory is converted to grayscale image. The grayscale image converted is reused by some parts. Thus, the grayscale image is stored into the array on the memory.

The brightness histogram is created by using the grayscale image. Then, the discriminant analysis is performed over the brightness histogram to find a threshold distinguishing the grayscale value into the white and black one.

Once the threshold value is found, the grayscale image previously stored into the array is loaded to perform binarization using the threshold value. Finally, the binarized image is output to the array on the memory.

3. Reconstruction for High-level Synthesis

3.1 Target Hardware Organization

Before inputting the pure software described by following the processing flow shown in Fig.1 to HLS tool, we would like to explain the hardware organization to which has been targeted.

Fig.2 shows this hardware organization. The hardware module generated by the HLS tool from binarization

Fig. 2. Hardware Organization for HLS

function in software is put to the HLS hardware shown in Fig.2. The HLS hardware has some ports to communicate some parameters with the embedded processor. In addition, the HLS hardware has AXI bus master port to access the memory directly. The image data is stored to the large memory like DDR3 SDRAM attached to the system-on-a-chip, SoC, externally. The HLS hardware can directly load and store the input and output image data on the external large memory via AXI bus master port.

3.2 Pure Software for HLS

Fig.3 is an overview of the pure software program in the pseudo code which has some pragmas to infer the input and output ports shown in Fig.2. Some loops have pragma, **PIPELINE**, to convert the loop iteration to the pipelined data path. In general, the inside of the discriminant analysis, **DiscAna**, has **PIPELINE** pragmas at the appropriate loops.

For the arguments of the top function, **AutoBin**, the pragmas, **INTERFACE**, to infer AXI bus master port are described at the top of the function.

```

1: AutoBin(*iimg,*gimg,*oimg, num){
2:   #pragma HLS INTERFACE m_axi port=iimg bundle=d
3:   #pragma HLS INTERFACE m_axi port=gimg bundle=d
4:   #pragma HLS INTERFACE m_axi port=oimg bundle=d
5:   for(i=0;i<256;i++){
6:     #pragma HLS PIPELINE II=1
7:     hist[i]=0;
8:     for(i=0;i<num;i++){
9:       #pragma HLS PIPELINE II=1
10:      gval=Gray(iimg[i]);
11:      gimg[i]=gval;
12:      hist[gval]++;
13:    }
14:    th=DiscAna(hist);
15:    for(i=0;i<num;i++){
16:      #pragma HLS PIPELINE II=1
17:      if(gimg[i]>th) oimg[i]=WHITE;
18:      else          oimg[i]=BLACK;
19:    }
20: }
  
```

Fig. 3. Pure Software for HLS

Fig. 4. Restructured Processing Flow and Number of Memory Access

3.3 Software Description to Reduce Memory Access

The memory access will significantly effect to the performance of hardware module while the software executed by the processor can hide the memory access latency by the cache memory. Thus, we attempt to reduce the memory access by restructuring the software function shown in Fig. 1 and Fig. 3. Fig. 4 depicts the restructured processing flow. Fig. 5 shows the restructured software function corresponding to Fig. 4.

As shown in Fig. 4, we have duplicated the grayscale functions in front of the histogram generation and image binarization. As a result, the number of memory accesses can be reduced from 4 to 3. Such duplication is not performed for the software generally. This is because this reconstruction extends the processing steps and make the execution time longer for the processor sequentially executes the instructions.

As shown in Fig. 5, the argument of the gray image, **gimg**, shown in Fig.3 is eliminated. The grayscale function, **Gray**, is also inserted into the loop at lines from 15 to 19. The inside of **Gray** is pipelined because the loop iteration including the **Gray** is specified as pipelining by pragma. Since the **Gray** is

```

1: AutoBin(*iimg,*oimg, num){
2: #pragma HLS INTERFACE m_axi port=iimg bundle=d
3: #pragma HLS INTERFACE m_axi port=oimg bundle=d
4: for(i=0;i<256;i++){
5: #pragma HLS PIPELINE II=1
6:     hist[i]=0;
7:     for(i=0;i<num;i++){
8:         #pragma HLS PIPELINE II=1
9:         gval=Gray(iimg[i]);
10:        gimg[i]=gval;
11:        hist[gval]++;
12:     }
13:     th=DiscAna(hist);
14:     for(i=0;i<num;i++){
15:         #pragma HLS PIPELINE II=1
16:         gval=Gray(iimg[i]);
17:         if(gval>th) oimg[i]=WHITE;
18:         else      oimg[i]=BLACK;
19:     }
20: }

```

Fig. 5. Restructured Software to Reduce Memory access

pipelined, this duplication may not effect to the performance of the hardware module.

4. Experiment and Discussion

4.1 Experimental Setup

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 was compiled by GCC with -O2 on the Xilinx SDK launched from Vivado 2016.4. The Vivado HLS 2016.4 was used to perform HLS. The VHDL programs generated by the Vivado HLS are compiled by the Vivado 2016.4 to generate the circuit data stream. To measure the execution time of the software and hardware, we have equipped the performance counter running at 100MHz as a memory mapped register on the FPGA.

To improve the performance, all of the floating point number are converted to the fixed point number.

4.2 Result and Discussion

Fig.6 shows the normalized execution time of the software execution, **SW**, and hardware execution, **HW**. The used software is the pure software shown in Fig. 3 and reconstructed software shown in Fig. 5. The bars are normalized to the performance of pure software set to 1.0 every software execution and hardware execution.

Fig. 6 indicates that the reconstruction duplicating the processing redundantly to reduce memory accesses degrades

Fig. 6. Normalized Execution Time

the software performance of 20% while the memory accesses are reduced. This is because the cache memory hiding the memory access latency effects well.

For the hardware execution, the performance is improved by 70%. This effect is led by reducing the memory accesses.

This fact indicates that the redundant execution of the processing to reduce memory access is important for the hardware design while the pure software engineers are hard to find such description.

5. Conclusions

The HLS can reduce the design burden of the hardware significantly. However, the HLS tools tend to generate a large and slow hardware generally when deep considerations of the hardware organization at software level are not taken care. This paper demonstrates a software describing method for HLS to reduce the memory access by duplicating the functions redundantly. The experimental result shows that our method can improve the hardware performance of 70% while the software degrades the performance of 20%. This fact indicates that the reconstruction of software for HLS is important while the pure software engineers are hard to find such reconstruction.

Currently, we are developing other reconstructing methods to improve the performance of automatic binarization more. This is because we have found that the hardware cannot overcome the software performance absolutely yet. In addition, we will develop more high-level synthesizable functions of the image processing library.

References

- (1) R. Nane, V.-M. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels, "DWARV 2.0: A CoSy-based C-to-VHDL Hardware Compiler", In FPL, pp. 619-622. 2012.
- (2) C. Pilato and F. Ferrandi, "Bambu: A Modular Framework for the High Level Synthesis of Memory-intensive Applications", In FPL, pp. 1-4, 2013.
- (3) A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J.H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-Level Synthesis for FPGA-based Processor/Accelerator Systems", In ACM FPGA, pp. 33-36. 2011.
- (4) Akira Yamawaki and Seiichi Serikawa: "A Describing Method of an Image Processing Software in C for a High-level Synthesis Considering a Function Chaining", IEICE Transactions on Information and Systems, Vol. E101-D, No. 2, pp.324-334, 2018.
- (5) Miyu Fujita, Kazunari Yosikawa, Naohiro Iwanaga, and Akira Yamawaki: "High-level Synthesis Oriented Describing Method of Template Matching", Proceedings of the 4th IIAE International Conference on Industrial Application Engineering 2016, pp.20-24, 2016.
- (6) Kazunari Yoshikawa, Naohiro Iwanaga, Tatuya Hamachi, and Akira Yamawaki: "Development of Fixed-point Simultaneous Square Root and Logarithm Operations for High-level Synthesis", Proceedings of the 2nd International Conference on Intelligent Systems and Image Processing 2014, pp.264-269, 2014.
- (7) Rafael C. Gonzalez and Richard E. Woods, "Digital Image Processing Third Edition", Pearson Prentice Hall, pp. 743-747, 2008.