CLAHE Implementation and Evaluation on a Low-End FPGA Board by High-Level Synthesis

Koki HONDA††, Kaijie WEI†, Masatoshi ARAI††, Nonmembers, and Hideharu AMANO†, Fellow

SUMMARY Automobile companies have been trying to replace side mirrors of cars with small cameras for reducing air resistance. It enables us to apply some image processing to improve the quality of the image. Contrast Limited Adaptive Histogram Equalization (CLAHE) is one of such techniques to improve the quality of the image for the side mirror camera, which requires a large computation performance. Here, an implementation method of CLAHE on a low-end FPGA board by high-level synthesis is proposed. CLAHE has two main processing parts: cumulative distribution function (CDF) generation, and bilinear interpolation. During the CDF generation, the effect of increasing loop initiation interval can be greatly reduced by placing multiple Processing Elements (PEs), and during the interpolation, latency and BRAM usage were reduced by reusing how to hold CDF and calculation method. Finally, by connecting each module with streaming interfaces, using data flow pragmas, overlapping processing, and hiding data transfer, our HLS implementation achieved a comparable result to that of HDL. We parameterized the components of the algorithm so that the number of tiles and the size of the image can be easily changed. The source code for this research can be downloaded from https://github.com/kokihonda/fpga_clahe.

key words: FPGA, image processing, CLAHE, side camera mirror, high-level synthesis

1. Introduction

Recently, automobile companies have tried to replace the side mirrors with cameras to reduce air resistance [1], and it provides us with the opportunity to improve the image quality of the side mirror using image processing techniques.

Image processing in a car requires strict requirements such as low latency and low power consumption. But if these restrictions can be met, embedded CPUs are the best choice because software developing effort is usually less costly than that of hardware design. However, there are many processes that cannot be met by embedded CPUs. In such cases, FPGAs are a promising option because FPGAs are energy efficient and are good at low-latency processing. Especially in recent years, the development of self-driving cars has led to the embedding of complicated processing in vehicles, and the demand for hardware accelerator using FPGAs is increasing [2], [3]. Nowadays, some cars embed Advanced Driver-Assistance Systems (ADAS) on FPGAs [4].

In our paper, we selected Contrast Limited Histogram Equalization (CLAHE) from many image processing techniques, which stresses the contrast of the image value. Applying CLAHE to the side mirror image is expected to reduce the glare of other cars’ lights reflected in the side mirror. CLAHE is suitable for FPGA because it can be processed by the data flow manner. Although there is a wide variety in the size of FPGAs, considering the cost perspective, a low-end FPGA is especially preferred. As our design methodology, we adopted Vivado HLS [5], which is a commonly used high-level synthesis tool considering its maintainability, flexibility, and easiness of design. Shortly, in this paper, we focus on the implementation of CLAHE on a low-end FPGA board by Vivado HLS. In our previous short paper [6], we briefly presented an early implementation of our work. This paper gets deeper into the design.

The rest of the paper is organized as follows. We describe the brief history of histogram equalization in Sect. 2.1 and then introduce ZYNQ and PYNQ in Sect. 2.2, 2.3 respectively. Section 2.4 mentions some related work. Then, we show the hardware implementation in Sect. 3, and analyze the result in Sect. 4. Finally, we conclude the paper and mention further work in Sect. 5.

2. Background

2.1 CLAHE

Histogram equalization, a basic technique to improve image quality, adjusts contrast using the image’s histogram. It is used in various situations such as X-ray image enhancement [7] and pre-processing of object tracking [8]. In histogram equalization, a pixel is converted by using Eq. (3), where Eq. (1) is the frequency of occurrence of the pixel value \( i \), Eq. (2) is the Cumulative Distribution Function (CDF), \( n \) is the total number of pixels in the image, and \( n_i \) represents the number of pixels whose pixel value is \( i \), \( L \) is the number of pixel value levels (usually 256), \( I_{\text{new}} \) is the converted pixel value. Problems occur when an image contains both bright parts and dark parts because the histogram is not biased to a particular range. To address such flaws, Adaptive Histogram Equalization (AHE) [9] was proposed. In the AHE, an image is partitioned into sub-tiles, and for each tile, CDF is generated. Then, a pixel is interpolated using the histograms by Eq. (4) called bilinear interpolation. (See Fig. 2, as well.) In Eq. (4), \( n \) is the value of the pixel of interest. (In Fig. 2, the green pixel.) Most pixels are interpolated using the cumulative histogram of four adjacent tiles.
tile, while the pixels in the corner are interpolated using one adjacent tile and the pixels on the edge are interpolated using two adjacent tiles shown in Fig. 1. The noise, however, often increases when the histogram slope is steep.

\[ p_x(i) = \frac{n_i}{n} \]  

(1)

\[ cdf_x(i) = \sum_{j=0}^{i} p_x(x = j) \]  

(2)

\[ I_{\text{new}} = \text{round}((L - 1) \times cdf_x(i)) \]  

(3)

To prevent the problem, Contrast Limited Adaptive Histogram Equalization (CLAHE) [10] was proposed. In CLAHE, a histogram generation process is devised like Fig. 3. In the AHE, a histogram is converted to a CDF once a histogram is generated, while in CLAHE, after generating a histogram, it is clipped by a predefined threshold value and is further redistributed to the pixel on the histogram. Then, pixels are converted to a CDF. This change reduces the noise because clipping prevents a CDF from being steep.

\[ I_{\text{new}} = \frac{s}{s + w} (t \cdash f_{UL}(n) + \frac{z}{z + t} \cdash f_{LR}(n)) + \frac{w}{s + w} (t \cdash f_{BL}(n) + \frac{z}{z + t} \cdash f_{BR}(n)) \]  

(4)

2.2 ZYNQ

ZYNQ is an SoC that is equipped with Processing System (ARM-based CPUs, DRAM, Communication interfaces, etc.), and a Programmable Logic (FPGA, PCIe, etc.). It enables us to easily coordinate PS and PL. For example, if you want to transfer data to the FPGA from PS DRAM, all you have to do is calling the memory allocating function and the Direct Memory Access (DMA) transfer function.

2.3 PYNQ

PYNQ is a framework that aims to make it easy to handle
FPGAs using Python. However, it does not mean that you can design FPGAs with Python at present. It is just a way to easily handle a pre-designed design using Python.

Python is one of the most popular programming languages for scientific computing and machine learning. The fact that we can use Python for FPGAs means that we can use the vast resources of Python and easily connect them.

In this study, the evaluation of the actual system was done in Jupyter Notebook, and the data transfer was done using the PYNQ DMA library. Jupyter Notebook is a platform that provides an interactive environment for various programming languages. When using Python, IPython, a modern interactive environment, is running as the kernel.

2.4 Related Work

Since CLAHE is an efficient image processing algorithm, there has been some implementation conducted on FPGAs.

Ferguson et al. [11]’s work is one of the earliest papers that implemented CLAHE on an FPGA. In their work, they used an 8-bin histogram due to the insufficient resources of FPGA at that time, so it was not practical. Furthermore, the execution time is not shown.

Kokufuta et al. [12] implemented CLAHE on Xilinx’s Virtex-4 FPGA board, a high-end FPGA board at the time. They achieved 537.9 fps on a 640*480 image. However, this does not take into account the bandwidth of off-chip memory, and the actual performance is limited to the bandwidth. They implemented sliding window-based CLAHE while we focus on interpolation-based CLAHE. In the sliding window-based CLAHE, one histogram is generated per pixel using adjacent pixels. Although this is a computationally-intensive process, it is possible to efficiently generate a histogram by taking advantage of the fact that adjacent pixels are almost the same. They designed efficient hardware by making good use of buffers and FIFOs. Both interpolation-based CLAHE and sliding window-based CLAHE are powerful calculation methods. Compared to sliding window-based CLAHE, which requires elaborate implementation at the HDL level, interpolation-based CLAHE can convert pixels using a single expression, bilinear interpolation, after generating CDFs. Since it is difficult to describe the hardware details in HLS, we decided to implement an interpolation based CLAHE.

Unal et al. [13] implemented interpolation-based CLAHE on a Zynq 7000 board by HDL. They achieved 450.1 fps on a 512*512 image and 33.3 fps on a 1920*1080 image. Their design is based on a carefully designed HDL description, and its performance becomes a good indicator for evaluating the level of our HLS based design.

Schatz [8] implemented CLAHE for continuous images on an FPGA. The latency was reduced by interpolating the image using the CDF of the previous image. This technique can be used only when there is almost no difference between consecutive images. At the present stage, we generate a CDF from an individual image, since the target application must be more robust to camera shake or sudden image change. However, as Schatz’s approach is useful for reducing resource usage, we might introduce it into our implementation in the future.

To the best of our knowledge, there is no CLAHE implementation on a low-end FPGA by HLS. HLS implementation is essential for practical usage because of its better maintainability and easy to be modified. We will later show that our high-level synthesis designs achieve performance that is comparable to elaborate HDL implementations.

3. Design and Implementation

3.1 Setting

We implemented the design based on the CLAHE code [14] of OpenCV [15]. It is an open-source library for image processing developed and published by Intel, and its functions are highly optimized, so it is suitable for the start point of development. However, since the design of the CPU and the design of the FPGA are quite different, a lot of changes are necessary. Our design is created by Vivado 2019.1 and Vivado HLS 2019.1.

Our target board is PYNQ Z1 [16], a low-end Zynq 7020 board. It has dual-core ARM Cortex-A9@650MHz and Artix 7 family FPGA and can utilize the PYNQ framework.

3.2 Implementation Challenge

On resource-poor low-end FPGA, streaming calculation is required to achieve high performance. We reduce the usage of BRAM because it is not necessary to hold the entire image. Besides, the overhead for data transfer can be hidden. In OpenCV CLAHE code, the CDF calculation function and the interpolation function were connected through a two-dimensional array holding the CDF. This connection hinders streaming because it must be connected via streaming interfaces (i.e. FIFO) to achieve streaming. In the multi-core CPU, high performance can be achieved by processing each CDF generation and interpolation in parallel. On the other hand, comparing with data flow processing, in low-end FPGAs, this design is unsuitable because it requires more space to hold the CDF and pixel until the start of processing, and the latency is increased eventually. so, overcoming this obstacle is the main design challenge.

3.3 Typical pragma of Vivado HLS

Here, we describe the pragmas used in our implementation briefly.

**pragma HLS dataflow:** If you give a data flow pragma, each process will be executed in an overlapping manner. However, each process must be connected by streaming interfaces and there are some additional restrictions.

**pragma HLS array_partition:** divides the array into
smaller arrays. Since BRAM has a limited number of elements that can be accessed in one clock cycle, this pragma may improve operational throughput.

**pragma HLS pipeline:** makes a pipeline of the loop. This pragma shortens the loop initiation interval. If option allows you to specify the target loop initiation interval.

**pragma HLS stream:** When it is used under a dataflow pragma, the arguments specified by the stream pragma are implemented in the FIFO. You can specify the FIFO depth with the depth option.

### 3.4 Overall Design

Our overall design is as shown in Fig. 4; it consists of four parts: pixel input module, CDF generation module, interpolation module, and pixel output module. In our design, each module is connected with streaming interfaces (hls::stream in Vivado HLS). Connecting by the streaming interfaces and adding a dataflow directive, the HLS tool can synthesize the design that can be executed in an overlapping manner. There is no need to synchronize between modules because what each module needs to do is only receiving the data, processing it, and outputting the data.

### 3.5 Pixel Input Module and Pixel Output Module

In the pixel input module, pixels are distributed by tiles and sending via multiple streaming interfaces, and the pixel output module sends converted pixels from the top module. I/O is processed at the speed of 1 pixel / 1 cycle.

### 3.6 CDF Generation Module

First, the image is divided into sub-tiles and each tile’s CDF is generated. The following two architectures, shown in Fig. 6 and Fig. 7 are possible. In Plan Single, there is one CDF processing unit. The unit generates the CDF of proper tile depending on the pixel index, while there provide more CDF processing units in Plan Multiple considering the horizontal tile divisions. The previous Pixel input module supplies the appropriate pixels for each unit.

We adopted Plan Multiple because of the following two reasons. One is that the HLS description of the CDF processing unit becomes simple. Simple HLS description is valuable because they are less prone to bugs and easier to change. We no longer have to think about the ownership of the pixel concerning tiles. Although Plan Single is resource-efficient as it can share the logic of histogram generation, it is necessary to identify the buffer holding the cumulative histogram from the pixel index when perform processing. In Plan Multiple, on the other hand, each unit simply creates a cumulative histogram from the input pixels.

Figure 5 shows the percentage of resource usage for the three (CDF generation, interpolation, and inter-module FIFO) that account for most of the resources of the overall system. Pixel input/output module is omitted from Fig. 5 because their resource usage is very small. From Fig. 5, CDF generation is dominant in DSP, FF, and LUT, but since the resource usage of DSP, FF, and LUT is about 50% of the whole board (explained in Evaluation Section), there is no problem to prepare multiple CDF generation units. Another reason is that the overall execution time is expected to be shorter. Using the method of [17], it is possible to set the histogram create loop initiation interval 1 of pipeline pragma, but when we synthesized the unit with the loop initiation interval of 1 on our target board, the output could not satisfy the target operating frequency. Therefore, we decided to synthesize with loop initiation interval 2. Then, in Plan Single, the execution clock cycle is roughly...
double and the total clock cycle will also be about twice even if the processing is completely overlapped. On the other hand, in Plan Multiple, since each unit can work in parallel, it is expected that the total clock cycle will be much smaller than twice the number of pixels even if the loop initiation interval is 2.

The CDF generation unit, shown in Fig. 8 consists of the following loops.

- A loop for histogram generation and clipping,
- two loops to redistribute clipped value, and
- a loop for applying cumulative sum and creating the CDF.

Most of the execution clock cycles are occupied by the histogram generation and clipping loop.

3.7 Interpolation Module

In OpenCV, the cumulative histogram of all tiles was assigned to an array at the CDF generation function and passed to the interpolation function. However, in this manner, interpolation cannot start until the cumulative histograms of all tiles are completed. In other words, the processing cannot be overlapped. We, therefore, changed the implementation to overlap the processing like Fig. 9. First, since interpolation requires a cumulative histogram of four tiles near a pixel, two buffers for holding one row of tiles’ CDF are required. However, since the module wants to receive the cumulative histogram from the CDF generation module while the processing is in progress, we prepare one more buffer to hold three rows of tiles. Then, the cumulative histogram is rotated around the buffers and substituted by the upcoming line to proceed with the processes. The array partition pragma is added to retrieve the required data in one clock cycle. As a result, the interpolation unit can start to proceed when it receives a cumulative histogram of one row of tiles. This is because the top edge can be interpolated if there is a cumulative histogram for one row of tiles.

This change significantly reduces the latency and BRAM usage because the number of pixels that must be retained before processing is reduced. Specifically, for a full HD image, the number of pixels that must be kept is now one-eighth (from 1920*1080 to 1920*135). The amount of CDF that must be retained is also relieved: the amount is cut to three of the vertical divisions of a tile.

The interpolation module, shown in Fig. 10, consists of
a loop that performs bilinear interpolation shown in Eq. (4). Because the data required for the calculation can be available in one clock cycle, this loop can be executed at interval 1.

3.8 HLS Implementation Detail

First, since the data flow pragma is given in the top function, each module can be executed in an overlapping manner as shown in Fig. 11. As a general rule, the innermost loop in each function is given a pipeline pragma with a loop initiation interval of 1. Also, to achieve loop initiation interval 1, the array is properly given an array partition pragma. A stream pragma is given to the argument that passes the data.

Table 1 shows loops in CDF generation unit. Most of the execution time is used to create histogram and clipping loop. Redistribution is done in two loops. This is the same as the OpenCV implementation.

Table 2 shows loops in Interpolation. By building a deep pipeline and applying array partition pragma, interpolation achieves loop initiation interval 1 in target frequency. This is one of the HLS implementation merits.

Table 3 shows the FIFO depth of hls::stream between modules. Proper FIFO depth is important because of FIFOs of insufficient depth cause deadlocks or poor throughput. The value was determined experimentally.

4. Evaluation

4.1 Evaluation Setup

In our implementation, for a 512*512 image, and the image is divided by 16*16 tiles, whose size is 32*32. For a 1920*1080 image, it is divided into 16*12 tiles, whose size is 120*90. The range of the histogram is 0 to 255. The interface of the top module is assumed to be 1 pixel per 1 cycle. The design is assumed to be an IP core that will be implemented in other FPGAs used for car electronics with other cores in the future. PYNQ Z1 was adopted as an example small FPGA board. So, the current implementation is done that the image data are stores in the DRAM. When we build a real back-mirror system with a PYNQ Z1 board, it is feasible that the image data from the camera are transferred from PS part to the DRAM through the interface at 1 pixel/clock at 142 MHz according to the implementation example [18].

4.2 Performance Target

With considering the further implementation on an in-vehicle camera, the performance target is 30 fps for the full HD image.

Algorithmically, histogram generation is $O(n)$ and interpolation is $O(n)$, where $n$ is the number of pixels, so if each unit can be implemented efficiently, and processing can be sufficiently overlapped, the execution time is close to the clock period $\times$ the number of pixels + data transfer time. In case of the full HD image, assuming an operating frequency of 100 Mhz, execution time becomes $1920 \times 1080 / 10 ns / 10^9 = 20.76ms +$ data transfer time(from camera to display). Since 30 fps is corresponding to about $33.3 ms$, the performance target can be achieved if it is implemented efficiently.

4.3 Real Machine Evaluation

This section compares the OpenCV code running on the dual-core ARM Cortex-A9 on PYNQ Z1, the OpenCV code on the qual-core ARM Cortex-A57 on Jetson Nano, OpenCV(CUDA) code on 128 CUDA-core Maxwell GPU on Jetson Nano, and our design running on the Artix 7 FPGA on PYNQ Z1. OpenCV 3.2.0 is used on PYNQ Z1’s CPU, and OpenCV 4.1.1 is used on Jetson Nano’s CPU and GPU. Jetson Nano is an embedded computer with a quad-core ARM Cortex-A57@1.43Ghz and a 128-core Maxwell GPU@921Mhz. In our setting, Jetson Nano runs on 10W
mode. The Jetson Nano adopts more advanced process technology comparing with the PYNQ-Z1 board. We’re comparing these two platforms mainly because they’re both edge computing hardware.

The real machine evaluation is presented in Fig. 13 and power consumption is shown in Table 4. Figure 12 shows a schematic diagram of the block design used in the real machine evaluation. An image is sent to the FPGA by DMA transfer. The FPGA execution time includes the image transfer time between PS DRAM and FPGA. All execution times are measured by %timeit, which is IPython’s magic commands.

For a 512*512 image, it took 17.0 ms for Cortex-A9, 2.07 ms for Cortex-A57, 2.45 ms for GPU, and 2.59 ms for FPGA. For a 1920*1080 image, it took 132 ms for Cortex-A9, 14.7 ms for Cortex-A57, 19.4 ms for GPU, and 21.6 ms for FPGA. We achieved a speedup of about 6.56 times for a 512*512 image and a speedup of about 6.1 times for a 1920*1080 image compared to Cortex-A9. We couldn’t achieve comparable performance to Cortex-A57 and Maxwell GPUs provided in Jetson Nano. The Jetson Nano is much more powerful than the PYNQ Z1 as you can see from the performance difference of the CPUs on board. Therefore, this is incomparable for these two completely different level platforms.

We measured the power consumption of the entire boards using ordinary power watt meter (System Artware SHW3A). As shown in Table 4, the power consumption of PYNO Z1 is 3.1W (FPGA itself is much less than 1W) while that of Jetson Nano is 10.1W and 6.5W respectively, so our design has an advantage in terms of power consumption. Figure 14 shows energy efficiency comparison. Our design achieves about 2 times the power efficiency of the CPU on the Jetson Nano and about 1.69 times the power efficiency of the GPU on the Jetson Nano in a full HD image.

4.4 Comparison with Related Research

Table 5 compares our results with a related study [13]. We used clock cycles estimated by Vivado HLS C/RTL co-simulation. From Table 5 our high-level synthesis implementation achieved comparable performance to the HDL implementation of the related work. The operating frequency is also higher in our implementation, probably because we are building a deep pipeline, especially with interpolation calculations (the depth = 51). Perhaps it is difficult to build a pipeline with dozens of stages in HDL. Furthermore, FPS can be calculated by \( \frac{1000}{21.6 \approx 46.7} \) for a 1920*1080 image, which is more likely to meet performance goals.

Table 5 also shows the lower bound of the execution time. Both our work and related study [13] assume 1 pixel / 1 clock cycle I/O. Under that premise, the number of pixels in the image \( \times \) the clock period is the lower bound. From this, we can see that our design can be implemented efficiently. If we want to achieve higher performance, we need to increase the I/O bandwidth, but it is difficult for the low-end FPGA board used in this study to handle the data volume.
4.5 Resource Utilization

The resource utilization rate is shown in Fig. 15. Resources other than BRAM make little difference as the image size changes. The amount of BRAM used increases as the image size increases. From this, we can see that scalability is limited by the BRAM. This is because as the image size increases, the BRAM usage increases due to the increase in the depth of the FIFO that stores the pixels between the pixel input module and the interpolation module. Note the change in depth between the pixel input module and the interpolation module in Table 3.

5. Conclusions and Future Work

This paper presented an implementation of CLAHE on the low-end FPGA board. In the CDF generation, by placing multiple PEs, the effect of increasing the loop initiation interval was reduced. In interpolation, latency and BRAM usage were reduced by revising how to hold CDF and calculation method. Then, by connecting each module with streaming interfaces and adding a data flow pragma, it is possible to overlap processing and hide data transfer. As a result of these techniques, we were able to achieve performance comparable to the implementation in HDL by HLS implementation. In future work, we will investigate the application of the designed CLAHE to the side mirror image.
Masatohi Arai is currently a professor in the Graduate School of Science and Engineering, Saitama University. His research interests include the area of image-processing and model-based development.

Hideharu Amano received Ph.D degree from the Department of Electronic Engineering, Keio University, Japan in 1986. He is currently a professor in the Department of Information and Computer Science, Keio University. His research interests include the area of parallel architectures and reconfigurable systems.