sensors

MDPI

Article

An Image Histogram Equalization Acceleration Method for Field-Programmable Gate Arrays Based on a Two-Dimensional Configurable Pipeline

Yan Wang [1], Peirui Liu [1], Dalin Li [1,2,*], Kangping Wang [1] and Rui Zhang [1]

[1] Key Laboratory of Symbol Computation and Knowledge Engineering of Ministry of Education, College of Computer Science and Technology, Jilin University, Changchun 130012, China; wy6868@jlu.edu.cn (Y.W.); liupr21@mails.jlu.edu.cn (P.L.)
[2] School of Computer Science, Zhuhai College of Science and Technology, Zhuhai 519041, China
* Correspondence: dlli16@mails.jlu.edu.cn

Abstract: New artificial intelligence scenarios, such as high-precision online industrial detection, unmanned driving, etc., are constantly emerging and have resulted in an increasing demand for real-time image processing with high frame rates and low power consumption. Histogram equalization (HE) is a very effective and commonly used image preprocessing algorithm designed to improve the quality of image processing results. However, most existing HE acceleration methods, whether run on general-purpose CPUs or dedicated embedded systems, require further improvement in their frame rate to meet the needs of more complex scenarios. In this paper, we propose an HE acceleration method for FPGAs based on a two-dimensional configurable pipeline architecture. We first optimize the parallelizability of HE with a fully configurable two-dimensional pipeline architecture according to the principle of adapting the algorithm to the hardware, where one dimension can compute the cumulative histogram in parallel and the other dimension can process multiple inputs simultaneously. This optimization also helps in the construction of a simple architecture that achieves a higher frequency when implementing HE on FPGAs, which consist of configurable input units, calculation units, and output units. Finally, we optimize the pipeline and critical path of the calculation units. In the experiments, we deploy the optimized HE on a VCU118 test board and achieve a maximum frequency of 891 MHz (which is up to 22.6 times more acceleration than CPU implementations), as well as a frame rate of 1899 frames per second for 1080p images.

Keywords: field-programmable gate arrays (FPGAs); histogram equalization; two-dimensional pipeline; hierarchical state machine

Citation: Wang, Y.; Liu, P.; Li, D.; Wang, K.; Zhang, R. An Image Histogram Equalization Acceleration Method for Field-Programmable Gate Arrays Based on a Two Dimensional Configurable Pipeline. *Sensors* **2024**, *24*, 280. https://doi.org/10.3390/s24010280

Academic Editor: Zahir M. Hussain

Received: 24 October 2023
Revised: 17 December 2023
Accepted: 19 December 2023
Published: 3 January 2024

1. Introduction

Histogram equalization (HE) is a very important image preprocessing method that is used to improve the quality of image processing results [1]. It has a wide range of applications in high-precision online industrial detection [2], unmanned driving [3], medical imaging [4], computer vision [5], video processing [6], etc. There are many different versions of HE [7], such as global histogram equalization, average brightness preserving histogram equalization, bucket modified histogram equalization, and local histogram equalization. Global histogram equalization is commonly used, and its main principle is to change the distribution of the whole image's histogram to a uniform distribution so as to improve the contrast of the image. However, the characteristics of global histogram equalization limit its maximum frame rate in high-speed scenarios, and its throughput should thus be improved.

Multi-core processors are commonly used for accelerating algorithms according to parallel computing theories. However, the accumulation part of HE limits its parallelism, and it is thus mainly implemented in the serial computing of traditional computers, which leads

to its low computing performance. Most existing HE acceleration methods, whether run on general-purpose CPUs or dedicated embedded systems, require further improvement in their frame rate to meet the needs of complex scenarios [8,9]. Even when implemented with GPUs, the performance gains are limited [10]. These methods also meet strict power consumption constraints in embedded scenarios [11,12].

In order to achieve better acceleration performance, our work explores optimization methods based on the principle of adapting algorithms to the hardware, taking field-programmable gate arrays (FPGAs) as the accelerating platform. In contrast with other typical parallel computing platforms, such as multi-core CPUs and GPUs, FPGAs have the characteristics of highly customizable parallel computation, high compatibility, and low power consumption [13].

The process of global HE can be divided into the following steps: calculating the histogram of the image, i.e., the frequency of each pixel (which is the number of pixel occurrences divided by the total number of pixels and is denoted by p_x); calculating the cumulative histogram of the image, i.e., the frequency that is less than or equal to this pixel (denoted by cdf_x); obtaining converted pixels are according to the cumulative histogram, and the cumulative histogram is multiplied by the largest pixel value (e.g., if the pixel bit width is 8, the largest pixel value is 255); and finally, the original image is converted.

Satisfactory performance cannot be reached by implementing HE on FPGAs directly [14]. In order to improve the performance of HE on FPGAs, it is necessary to optimize the method's computing architecture and make it more suitable for FPGAs.

Our work first reconstructed the logic of HE to make it suitable for parallel computation, and it was then optimized to be a 2D pipeline architecture so as to improve the FPGAs' computation speed. We then used three different methods to implement the optimized HE on FPGAs according to different resource utilization strategies. In addition, the performance bottleneck calculations and logic were optimized on the hardware level. We deployed the optimized HE on a VCU 118 test board [15] using PCIe 3.0 communication ports. Compared with a single-core CPU intel i7-8750H, the acceleration ratio of the PCIe3.0*16 interface can be up to 22.6 times higher. Compared with other FPGA-based implementations of a single-input architecture (based on Zynq-XC7Z030) [16], the performance was 43.0 times higher.

In summary, this paper makes the following contributions:

1. We reconstruct HE according to the principle of adapting algorithms to the hardware such that the customized two-dimensional configurable pipeline is able to improve the throughput.
2. We propose a fully configurable computation architecture, and the parameters of HE such as image resolution, pixel bit width, and number of input pixels can be easily customized.
3. We improve the frequencies of the HE implementation through calculation transformation and by pipelining some of the steps of the calculation, such as accumulation and comparison value updates.

2. Related Works

Many previous works have been dedicated to accelerating HE. One architecture that has been studied is the "decoder + parallel computing unit" architecture [17]. The input of the decoder is the pixel, and the output is the result corresponding to all pixel values. The number of outputs is $2^{pixel\_bitwidth}$, i.e., when a pixel is considered as an address and input into the decoder, the output bit of the corresponding address and those values less than the address are both 1, and all other bits are 0. Each computing unit receives one output bit from the decoder. If the value of the bit is 0, its own statistical value remains unchanged; if it is 1, its own statistical value increases by 1. After the statistics are finished, each parallel computing unit stores its statistical values into the BRAM sequentially after calculating the converted pixel values. The advantage of this architecture is that the cumulative histogram that corresponds to all pixels can be calculated at the same time instead of calculating

the individual histograms first and then the cumulative histogram, which saves time. However, the wiring of this architecture is very complex, and the net delay is incredibly large, so it cannot be run at a particularly high frequency. This method calculates division using the higher-order bits, and it has limited application scenarios. The results obtained from the calculation need to be sequentially stored in BRAM, which also causes greater computational latency and degrades performance.

A few other articles have applied a similar architecture [18,19]. There are also articles that improve upon this architecture or add other features. For example, the generated histogram can be displayed directly through the display module [16]. An alternative to sequential BRAM writing is to use a multiplexer, which does not require BRAM writing time, while also allowing for the random selection of the calculation results [20]. The calculation of division can also be made more flexible. One way to achieve this is to use an integer divider, but this method cannot compute rounding [21]. The other method is to multiply by a factor and then take the higher-order bits, but this method needs sufficient precision [20].

Another typical architecture is to use BRAM for histogram calculation [22]. The input pixel is used as the BRAM address, and the data stored in the BRAM correspond to the accumulated number of corresponding pixels. But because one of BRAM's ports can only be read or written in a clock cycle, different studies have used different methods. One such method is to read BRAM only when there is a change in the data. Another method is to use two-port BRAM and read on one port and write the calculated value on the other port, so that the read and write operations can be completed in one clock cycle. After the histogram statistics are completed, the accumulated circuit is used to calculate the accumulated histogram, and the converted data are calculated by the corresponding calculation and stored back in the BRAM. This architecture has the advantage of saving logical resources, but it is limited both in terms of speed and multi-input scaling.

There are many other implementations based on this architecture. Among them, the addition used to generate the histogram can be calculated by DSP, thus reducing the utilization of resources [23]. DSP can also be used in processing histograms. Not only can DSP be used to generate cumulative histograms, but it can also be programmed to achieve additional functions [24]. Based on this architecture, FPGAs and computer software can be combined to achieve heterogeneous computing with greater flexibility that can achieve more functions [25].

For the calculation of HE with multiple inputs and outputs, both of the above architectures have their corresponding variants. Multi-input and -output architectures, such as the multi-decoder architecture, are the first kind [26]. In this artchitecture, multiple pixels are input to multiple decoders, and the output of the same bit from different decoders is input to the same computing unit. Each computing unit adds the accumulated value of its own through the number of 1s obtained from multiple inputs, so as to calculate the accumulated histogram. The second type of multi-input and -output architecture, such as the multi-BRAM port architecture [27,28], can read or write in parallel, but not simultaneously. In the case of parallel writing, statistics are performed through multiple BRAM ports, and each port inputs a part of the image; then, the generated histograms are added together to generate a complete histogram. In the case of parallel reading, one pixel is input each time to multiple BRAM ports, and then multiple identical histograms will be generated and read through multiple BRAM ports, so as to realize parallel reading. The performance of these architectures is not high; the multi-decoder's wiring is too complex, resulting in a large wiring delay; the multi-BRAM ports do not achieve simultaneous input and output results; and with the increase in the use of BRAM, the performance of the circuit will be reduced.

Concerning the calculation method of the cumulative histogram, some studies replace division by taking the higher-order bit [17] because when the lower-order bits are all 0, directly abandoning the low position is equivalent to a right shift operation, which can replace division. However, this method can only calculate the division when the divisor

is the index of 2. Other methods do not optimize the division and use hardware DSPs or configurable logical resources to compute the division.

In addition to designing the computational architecture, the MATLAB code can be automatically converted to RTL code and run on FPGAs [29]. There are also some implementations developed using HLS [30,31] and other tools [32]. These methods may speed up development, but they are not specifically optimized for performance.

3. Parallelization of Histogram Equalization and Its Hardware Computing Architecture

3.1. Histogram Equalization Algorithm

Each image can be represented as an $m_r * m_c$ matrix of integer pixel intensities ranging from 0 to $L-1$. L is the number of pixel intensities; if the bit width of the pixel intensities is w, then L is 2^w. For example, if w is 8, L is 256.

Let p denote the normalized histogram of the image with a bin for each possible intensity, as follows:

$$p_x = \frac{n_x}{n} = \frac{\sum_{i=1}^{n}(x_i == x)}{n}, \tag{1}$$

where n_x is the number of pixels with an intensity of x; n is the total number of pixels and $n = m_r * m_c$; and x_i is the intensity value of the i'th pixel of the image, where $(x_i == x)$ is a Boolean expression that equals 1 if x_i and x are equal and 0 if they are not.

The output pixel intensity of HE for each input pixel intensity x is y_x and is defined as

$$y_x = round((L-1)\sum_{i=0}^{x} p_i), \tag{2}$$

where $round()$ rounds to the nearest integer.

3.2. Parallelizing Histogram Equalization

We parallelize HE using two approaches: (1) computing all cdf_x in parallel and (2) processing multiple input pixels simultaneously.

For processing m inputs simultaneously and calculating $\sum_{i=0}^{x} p_i$ directly, according to Formula (1),

$$n_x = \sum_{i=1}^{n/m}\sum_{j=1}^{m}(x_{ij} == x), \tag{3}$$

where x_{ij} is the j'th input in the i'th multiple input.

In addition to computing m inputs simultaneously, we can also compute the cdf_x directly in parallel. According to Formulas (1)–(3),

$$cdf_x = \frac{1}{n}\sum_{k=0}^{x}\sum_{i=1}^{n/m}\sum_{j=1}^{m}(x_{ij} == k). \tag{4}$$

We can adjust the order of accumulation as follows:

$$\frac{1}{n}\sum_{k=0}^{x}\sum_{i=1}^{n/m}\sum_{j=1}^{m}(x_{ij} == k) \Rightarrow \frac{1}{n}\sum_{i=1}^{n/m}\sum_{j=1}^{m}\sum_{k=0}^{x}(x_{ij} == k). \tag{5}$$

The innermost accumulation can be obtained directly:

$$\sum_{k=0}^{x}(x_{ij} == k) \Rightarrow (x_{ij} <= x). \tag{6}$$

Finally, the formula to calculate cdf_x in parallel is

$$cdf_x = \frac{1}{n}\sum_{i=1}^{n/m}\sum_{j=1}^{m}(x_{ij} <= x), \tag{7}$$

and according to Formulas (2) and (7), $f(x)$ is

$$f(x) = round(\frac{L-1}{n}\sum_{i=1}^{n/m}\sum_{j=1}^{m}(x_{ij} <= x)), \tag{8}$$

$\frac{L-1}{n}$ is a constant when L and n are determined.

So far, HE has been parallelized to fit m inputs simultaneously and can calculate cdf_x directly in parallel. The parallelized HE involves the computation of three dimensions. Dimension i is the time dimension, which is executed sequentially. Dimension j is used for multiple inputs, executed in parallel. The x dimension is the transformation result of each pixel's intensity, which is executed in parallel.

3.3. Principles of Hardware Computing Architecture

The typical architectures mentioned in the Related Works Section have their own disadvantages and limited performance. The architecture based on BRAM, which is treated as a lookup table, is not capable of parallel input and parallel output at the same time. In addition, the fan-out architecture has a very high net delay and clock skew.

The overall architecture should keep the "total fan out" low. The connection complexity of the architecture with a "high total fan-out" is big, which means that every cell extended in one dimension needs to connect all the cells in the other dimension. As a result, the architecture is not easy to scale, and has high net delay and clock skew.

3.4. Architecture for Parallelized Histogram Equalization

We proposed a novel approach to realize parallel computing in both dimensions, thus forming the overall architecture of a two-dimensional pipeline, as shown in Figure 1.

The overall architecture consists of input units, computing units, and output units. The input unit inputs the pixels of the image to be computed. When the input pixel (from left) is less than or equal to the pixel for which the input unit is responsible, it will increase the value of the unit above and output the result to the unit below. The input pixels will be directly output to the right.

The behavior of the input units is shown in Algorithm 1, where "pixelIn" is the input from the left, "valueIn" is the input from above, "pixelOut" is the output to the right, and "valueOut" is the output below.

Algorithm 1 The Behavior of The Input Units

Input: pixelIn,valueIn
Output: pixelOut,valueOut
  pixelOut ← pixelIn
  if (pixelIn <= compareNum) **then**
    valueOut ← valueIn + 1
  else
    valueOut ← valueIn
  end if

The computing unit performs multiplication and rounding calculations based on the values input from above, and each computing unit is responsible for the calculation of one pixel's intensity. The computing unit passes the result to the output unit below. The input from the left is the frame-synchronizing signal, which will control the output of the computing unit and is also directly output to the right.

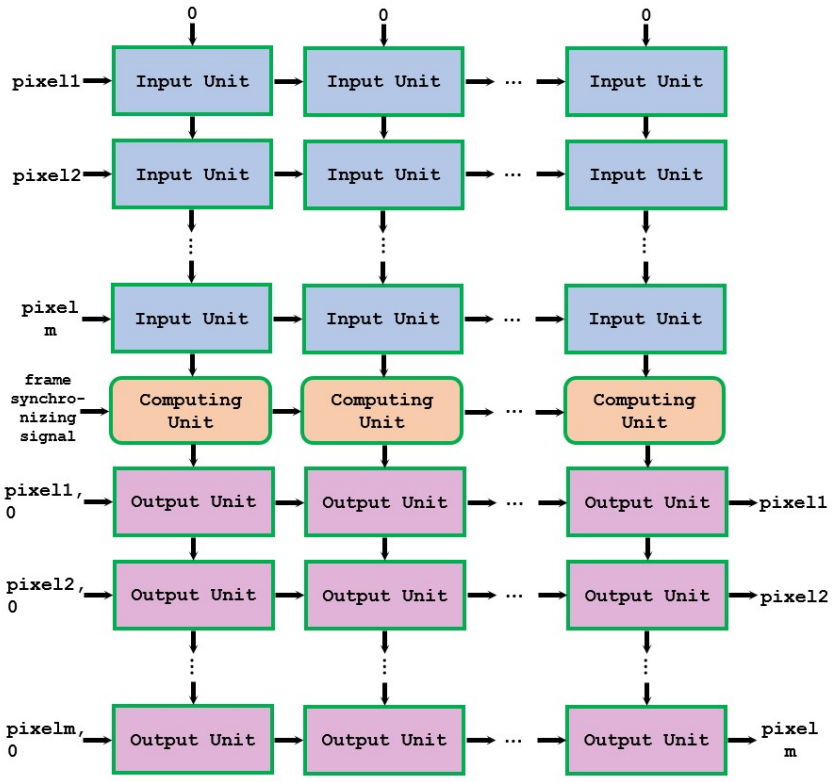


Figure 1. Two-dimensional configurable pipeline architecture.

The behavior of the computing units is shown in Algorithm 2, where "frameSynIn" is the input from the left, "valueIn" is the input from above, "frameSynOut" is the output to the right, and "valueOut" is the output below.

Algorithm 2 The Behavior of The Computing Units

Input: valueIn,frameSynIn
Output: valueOut,frameSynOut
  frameSynOut ← frameSynIn
  if (!frameSynIn) **then**
    valueOut ← valueOut
    accumulator ← accumulator + valueIn
  else
    valueOut ← $round(\frac{L-1}{n} * \text{accumulator})$
    accumulator ← 0
  end if

After the pixels of a frame are made into input units and the values are calculated by the computing units, the calculation of $f(x)$ in the histogram equalization is completed. At this point, these pixels need to be made into output units to complete their conversion. The output unit of each column is responsible for the transformation of the same pixel, that is, x in Equation (8). $f(x)$ is the input from the computing unit above and is output directly below. The pixel to be converted is input from the left, and if the input value is the unit the pixel is responsible for and has not been converted before, the value will be converted, and the result will be output to the right. A flag bit is also required to indicate whether the pixel has been converted. If the pixel has been converted, the flag bit will be set to one.

The behavior of the output units is shown in Algorithm 3, where "pixelIn" is the input from the left, "valueIn" is the input from above, "pixelOut" is the output to the right, and "valueOut" is the output to below. Moreover, "comparedIn" is the flag bit input from the left, and "comparedOut" is the flag bit output to the right.

Each unit is only connected to up to four other units, reducing the connection complexity. The input–output of the units at the edge are special, with and input of 0 for the topmost input unit and an input of 0 for the flag bit of the leftmost output unit. In this way, we can add as many rows or columns as we need without affecting the connections between original units. This means that we can easily scale up the computing architecture without affecting the overall running frequency.

Algorithm 3 The Behavior of The Output Units

Input: pixelIn,comparedIn,valueIn
Output: pixelOut,comparedOut,valueOut
  valueOut ← valueIn
  if (pixelIn == compareNum)&&(!comparedIn) **then**
    pixelOut ← valueIn
    comparedOut ← 1
  else
    pixelOut ← pixelIn
    comparedOut ← comparedIn
  end if

The example in Figure 2 shows the data flow of input units.

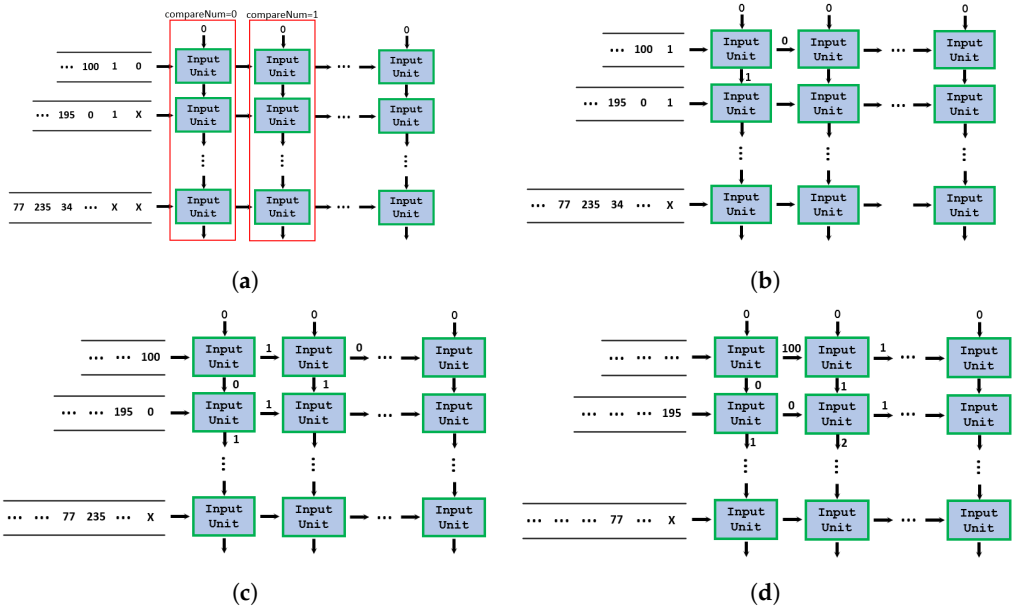


Figure 2. The data flow of input units. (**a**) t = 0. (**b**) t = 1. (**c**) t = 2. (**d**) t = 3.

The time complexity of the optimized HE is

$$O(n) = n/m, \tag{9}$$

where m is the number of inputs and n is the number of pixels in the image. The number of units N is

$$N = (2m + 1) * L. \tag{10}$$

4. Implementation of Histogram Equalization on FPGAs

We deploy the optimized HE on FPGAs. In order to study the acceleration effects and resource utilization of HE on FPGAs in depth, we implement HE through three different strategies: (1) with configurable logical (CL) resources only; (2) with CL and DSPs; and (3) with CL, DSPs, and BRAMs. Each strategy achieves a different performance and resource utilization, and can be applied in different scenarios. If we need high speed, we can use the

CL only strategy; if the resources of the FPGAs are insufficient, the strategy utilizing CL and DSPs can be used; and if the speed of the interface is limited, the strategy utilizing CL, DSPs, and BRAMs can be used.

4.1. Utilizing Only Configurable Logical Resources

This method uses configurable logical resources, except for the FIFO used as the axis interface.

4.1.1. Overall Architecture Based on FPGAs

The overall architecture based on FPGAs is shown in Figure 3. The calculation process is to input two identical frames into the *Input FIFO* successively. The first frame is used for calculation, and the second frame is used for conversion. The *Controller* controls the input of the first frame to the first set of ports (1, 2, 3, 4) and the input of the second frame to the second set of ports (5, 6, 7, 8).

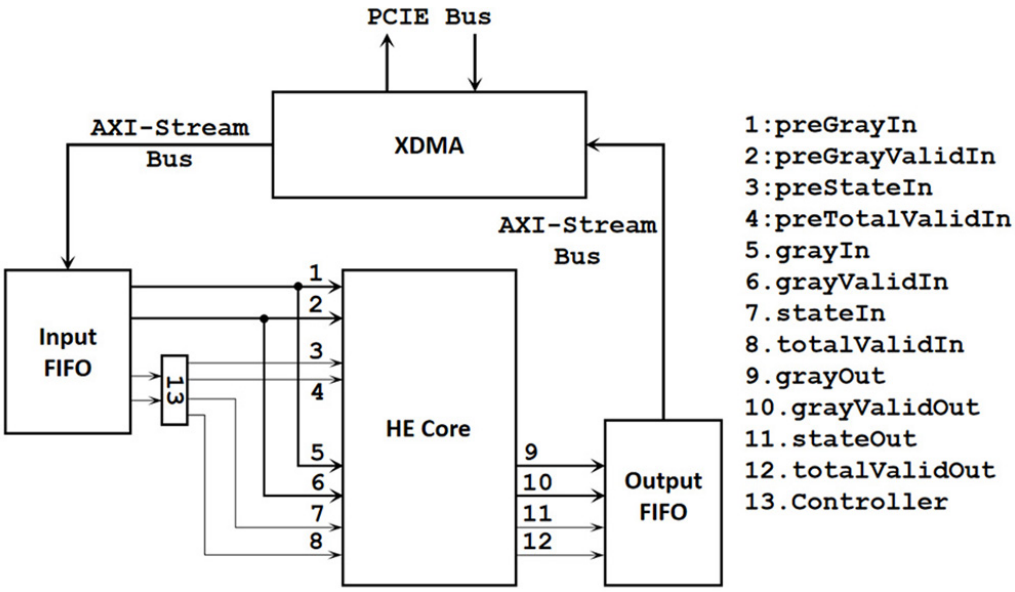


Figure 3. Overall architecture utilizing only CL.

4.1.2. Overall Architecture of HE Core

The overall architecture adopted is shown in Figure 4. In the case of increasing input and output, using a pipelined architecture to transfer multiple inputs and outputs can reduce wiring complexity and logic delay and improve performance.

A vertical row of input units constitutes a multi-input pipeline architecture. The final result of the multi-input pipeline architecture represents the number of input pixels that this computing unit is responsible for. The calculate unit takes in the input values, performs calculations, and outputs the final transformed result of the pixel it is responsible for. The calculate unit also receives the state signal, which is used to perform frame synchronization. A vertical row of output units constitutes a multi-output pipeline architecture. The "direction" in the figure indicates the direction of the data flow. The multi-input pipeline architectures, computational units, and multi-output pipeline architectures make up the two-dimensional pipeline architecture.

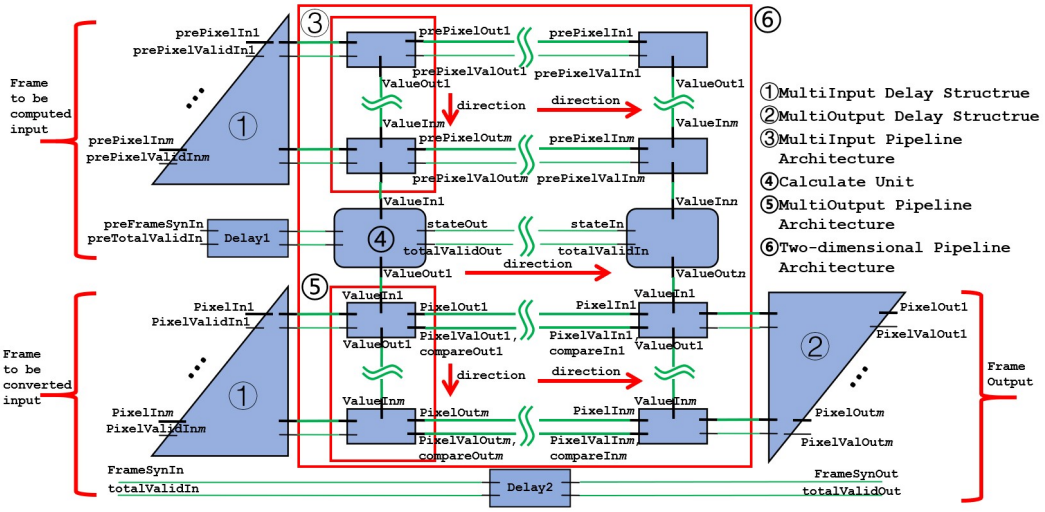


Figure 4. Overall architecture of HE core.

4.1.3. State Machine Design

Since this architecture is composed of many computing units, sharing a global state increases the total fan-out degree of the state machine. Different from the global state machine, we adopt a pipelined distributed hierarchical state machine link, as shown in Figure 5.

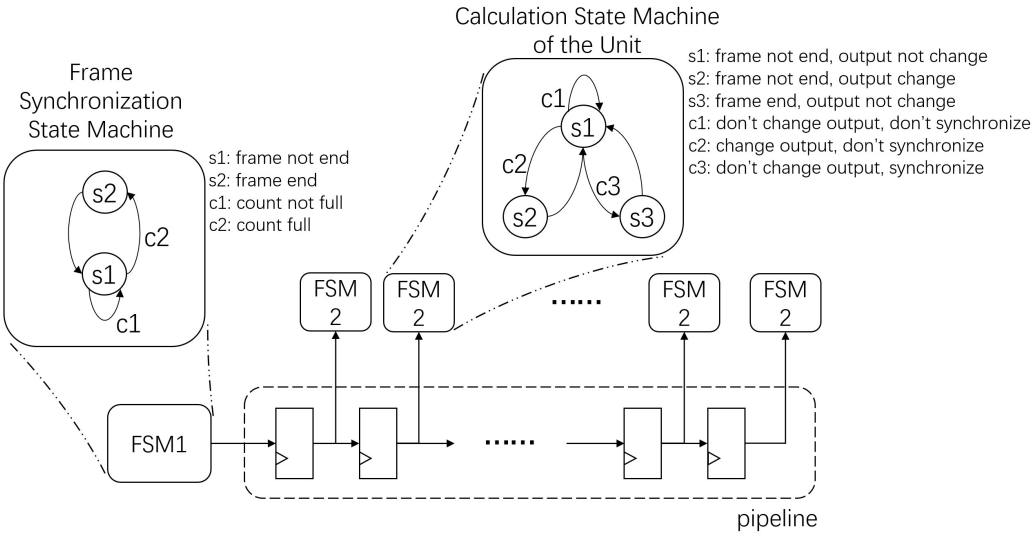


Figure 5. State machine link.

The first layer's state machine is used to perform frame synchronization. Here, the state machine is at the head of the pipeline, and the output state is transmitted downstream along the pipeline. The second layer's state machine is used to indicate the state of the computed histogram, and each cell has its own second-layer state machine.

In the frame synchronization state machine, the state of $s1$ is in the frame, at which point the counter counts the number of pixels received. The state of $s2$ is the end of the frame, at which point the counter is cleared. $c1$ is the transition condition from $s1$ to $s1$ when the counter has not reached the end of a frame. $c2$ is the transition condition from $s1$ to $s2$ when the counter reaches the end of a frame.

In the calculation state machine, $s1$ is the state that counts and keeps the output value, $s2$ counts and changes the output value, and $s3$ outputs the output value. $c1$ represents the condition when the output value should not be changed, $c2$ is when the output value should be changed but the frame does not end, and $c3$ is when the frame ends.

Benefiting from the Xilinx's distributed clock, the latency between blocks is very low, so we use a distributed state machine architecture, where each state machine is measured at the nearest computation unit and takes the clock signal from the block clock, so that the different state machines can maintain a high degree of synchronization. When the number of processing units increases, the maximum frequency caused by the extension of the state machine line will not decrease as the single state machine does.

4.1.4. Details of Some Units

Since the input and output pipeline architecture needs to output the result of the input in the same clock cycle, there will be a delay structure at the input and output to ensure that the final output of the pipeline architecture is the result from the same clock cycle input. The multi-input and multi-output delay structure is shown in Figure 6.

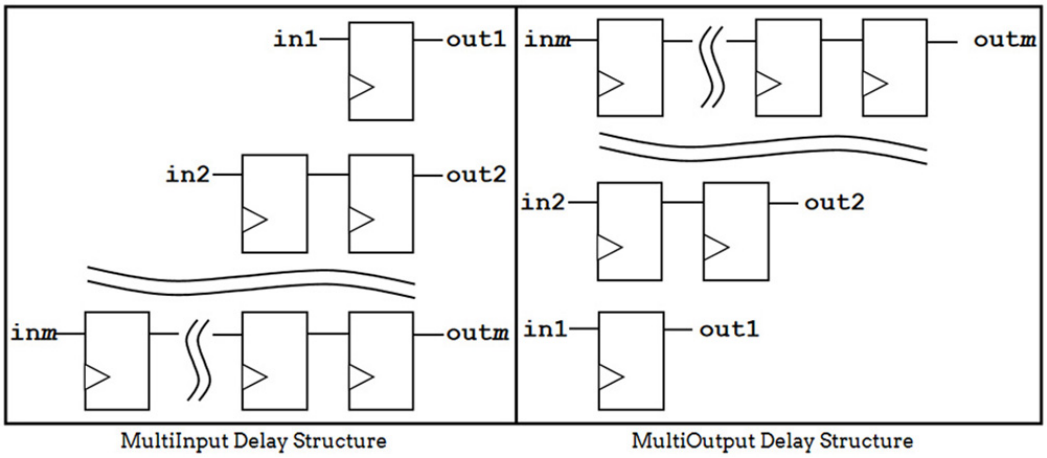


Figure 6. Multi-input/output delay structure.

4.1.5. Calculation Method Optimization

After we calculate $\sum_{i=1}^{n/m} \sum_{j=1}^{m} (x_{ij} <= x)$ in Equation (7), we need to calculate $f(x)$ in Equation (8). This requires the use of multiplication and division, but we can optimize these functions to addition and subtraction.

For multiplication by $L-1$, L is the exponent of 2, and multiplication by the exponent of 2 can be solved by simply shifting bits to the left, as follows:

$$x * (L-1) \Rightarrow (x << w) - x, \tag{11}$$

where w is the pixel bit width. In this way, we optimize multiplication to subtraction.

In the case of dividing by n in Equation (8), it can be compared with n when accumulating using an accumulation register. If the result is greater than n, the result register is incremented by 1, and the accumulated result is reduced by n and stored back in the accumulation register. Thus, Algorithm 4 can be optimized to Algorithm 5.

Algorithm 4 Dividing by n

Input: valueIn
Output: result
1: accumulator ← 0
2: **loop**
3: accumulator ← accumulator + valueIn
4: **end loop**
5: result ← accumulator/n

Algorithm 5 Optimized Dividing by n

Input: valueIn
Output: result
accumulator ← 0
result ← 0
loop
  accumulator ← accumulator + valueIn
  if (accumulator > n) **then**
    result ← result + 1
    accumulator ← accumulator − n
  end if
end loop

In this way, we optimize division to addition and subtraction.

By integrating the above two methods, we can obtain an optimized calculation method. The optimized HE is shown in Algorithm 6.

Algorithm 6 The Behavior of The Computing Unit

Input: valueIn,frameSyn
Output: valueOut
if (!frameSyn) **then**
  valueOut ← valueOut
  if (accumulator + (valueIn<<w) − valueIn ≥ n) **then**
    accumulator ← accumulator + (valueIn<<w) − valueIn − n
    result ← result + 1
  else
    accumulator ← accumulator + (valueIn<<w) − valueIn
    result ← result
  end if
else
  valueOut ← result
  accumulator ← n/2
  result ← 0
end if

The computing unit receives the number from the input module and multiplies the received number by $L - 1$. The value obtained after multiplication is added to the previous value in the accumulation register and is compared with n. Then, the accumulator and the result are updated according to the method above.

4.1.6. Critical Path Optimization

The optimized calculation method does not run at very high frequencies, so the critical path needs to be optimized. Here are some optimizations for the critical path:

1. Increase the number of stages in the pipeline: As Algorithm 7 shows, updating a in one clock cycle does not result in strong performance for the following computation.

Algorithm 7 Updating a in one clock cycle

if (a + b > c) **then**
  a ← a + b − c
else
  a ← a + b
end if

We can split this algorithm into three stages. The first stage computes $a + b$, the second stage compares the result to c and calculates the difference from c, and the third stage selects a result and returns it to a.

2. Replace comparison with signed subtraction: Since the comparison operation circuit will degrade the performance degradation of Algorithm 7, the subtraction operation needed at the same time can be changed into a signed subtraction operation, and the sign of the subtraction operation can be used instead of the output of the comparison operation, so that the comparison and subtraction can be performed at the same time, thus improving performance.
3. Replace signed subtraction with addition: The binary subtraction $x * L - x$ is calculated as

$$x * L - x = x * L + \sim x + 1, \tag{12}$$

and when we put 0s in the lower-order bits of x to obtain $x * L$ and then replace the lowest bit of $x * L$ with 1, we obtain $x * L + 1$ directly:

$$x * L - x = \{x, (\$size(L) - 1)'b0, 1'b1\} + \sim x. \tag{13}$$

The above equations refer to Verilog syntax.

4. Use clock enable: The clock enable port can be utilized to simplify the conditional branch circuit and thus improve performance. Clock enable is conducted by establishing the condition that the output value does not change in the outermost layer. This was also on the critical path before optimization.
5. Condition separation: Some state machines have too many states, in which case the condition can be separated into two registers to reduce the signal delay.
6. Loop structure optimization: The feedback circuit with only finite instances of feedback can be optimized into multiple non-feedback circuits with the same structure in series to improve the performance.

The pseudocode of the computing unit with the critical path optimization applied is shown in Algorithm 8, where "q1" is the first digit of the shift register, "q2" is the second digit of the shift register, and "qk" is the k'th digit of the shift register. The input to the shift register is "~frameSyn". Algorithm 8 refers to Verilog syntax.

Algorithm 8 The Behavior of The Computing Unit

Input: valueIn,frameSyn
Output: valueOut

```
valueIn255 ← {valueIn,8'h01} + {8'hFF,~value}
if (q3&q4&q5) then
    stage1 ← valueIn255 + remainder
else
    stage1 ← valueIn255
end if
stage2 ← stage1
stage2diff ← stage1 − pixelNum
if (stage2diff[sign_bit]) then
    remainder ← stage2
else
    remainder ← stage2diff
end if
if (stage2diff[sign_bit]&q4) then
    quotient ← quotient
else
    if (stage2diff[sign_bit]) then
        quotient ← 0
```

Algorithm 8 *Cont.*

```
    else
        quotient ← quotient + 1
    end if
end if
if (q4) then
    quotientReg ← quotient
else
    quotientReg ← quotientReg
end if
if (q1)|(q2&q3&q4) then
    if (q1) then
        accumulator ← accumulator + remainder
    else
        accumulator ← pixelNum/2
    end if
else
    accumulator ← accumulator
end if
accudiff1 ← accumulator − pixelNum
accudiff2 ← accudiff1 − pixelNum
accudiff3 ← accudiff2 − pixelNum
if (q8) then
    valueOut ← ~accudiff1 + ~accudiff2 + ~accudiff3
else
    valueOut ← valueOut
end if
```

4.1.7. The Architecture of the Computing Unit

The architecture of the computing unit is shown in Figure 7.

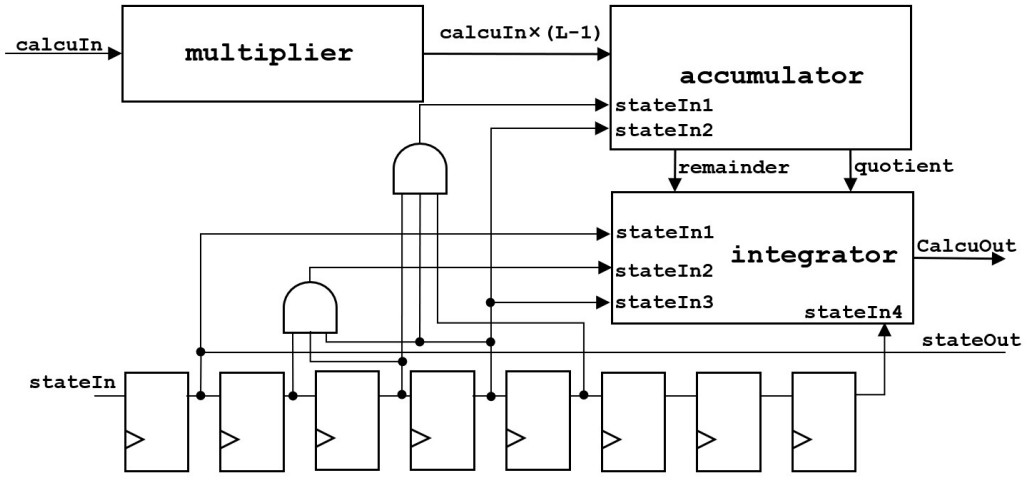


Figure 7. The architecture of the computing unit.

The statistical value of pixels *calcuIn* will be input to *multiplier* for multiplication and then input to *accumulator* for accumulation. The accumulative value will be compared with the total number of pixels. If the value is greater than the total number of pixels, the total number of pixels will be subtracted. Due to the fact that the *accumulator* accumulative value is updated in three cycles, it will output one quotient and three remainders. Thus, in the end, it will input into *integrator*, add the remainder of the three groups together,

subtract and compare with the total number of pixels three times, and add the result to the quotient to obtain the final result. Moreover, *stateIn* is the frame-synchronization signal.

The architecture of the *accumulator* is shown in Figure 8.

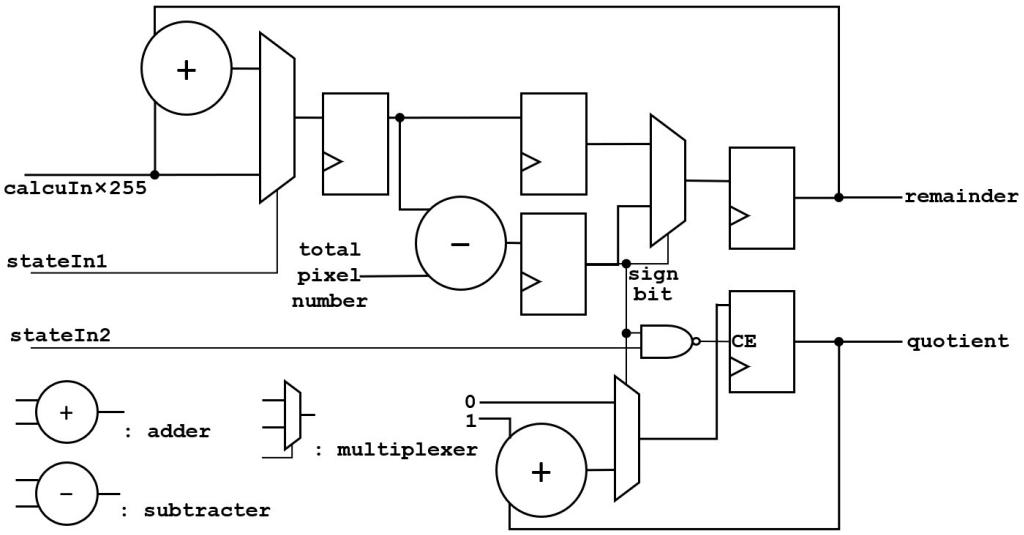


Figure 8. The architecture of the accumulator.

The value after multiplication is input into the *accumulator*, and the *quotient* and *remainder* are calculated. The calculation takes three clock cycles, and each cycle has an input, so the accumulator outputs three remainders. Since all quotients are added to one register, only one quotient is output. *stateIn*1 is an AND logic with *stateIn*'s clock delay of three, four, and five cycles, which is an indication signal for the accumulator to reaccumulate at the beginning of a frame. *stateIn*2 is an indicator signal that the accumulator can clear the quotient.

The architecture of the *integrator* is shown in Figure 9.

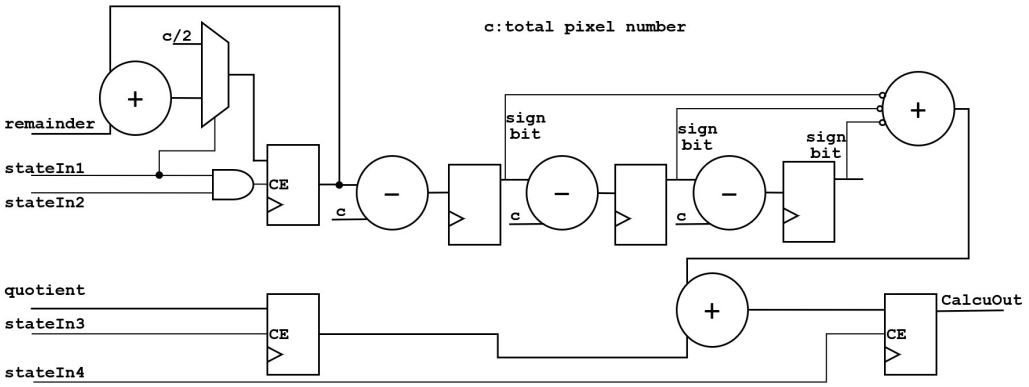


Figure 9. The architecture of the integrator.

The *integrator* adds the three remainders of the input together, compares them with the total number of pixels three times, and adds the three results into the quotient to obtain the final result. *stateIn*1 is an indication signal for the initialization of the integrator, *stateIn*2 is the indication signal of the three accumulations, *stateIn*3 is the indication signal to add the result of the three subtractions and the quotient, and *stateIn*4 is an indication signal that the integrator outputs the result.

4.2. Utilizing CL and DSP

The overall architecture of this method is the same as the previous method, but the configurable logic resources responsible for the computation in the computing unit are replaced with DSP. The accumulation and division functions of the unit are completed by DSP. To calculate the rounding value of a number multiplied by a fraction, we can use DSP's integer multiplication and accumulation:

$$round(x * \frac{a}{b}) = x * \lfloor \frac{a * 2^{\$size(b)+accuracy-1-\$size(a)}}{b} \rfloor, \quad (14)$$

where b is 2 to the n and dividing by b takes the higher-order bits. We take 9 higher-order bits and round them to the 8-bit result. Thus, the approximate division operation is realized. When the accuracy of the calculation is sufficient, there is no error.

In order to satisfy the accuracy, we use two DSPs to calculate the multiplication with a bit width of 34 bits. One DSP is used to compute the multiplication and accumulation operations of the lower-order bits, and the other DSP is used to compute the multiplication and accumulation operations of the higher-order bits, and then add the output value from the low-order bits.

4.3. Utilizing CL, DSP, and BRAM

Based on utilizing DSP resources, this method adds an image cache implemented by BRAM. This architecture does not require the same image to be input twice. The overall architecture is shown in Figure 10. The frame to be converted will be stored in the image cache and be output according to the current state.

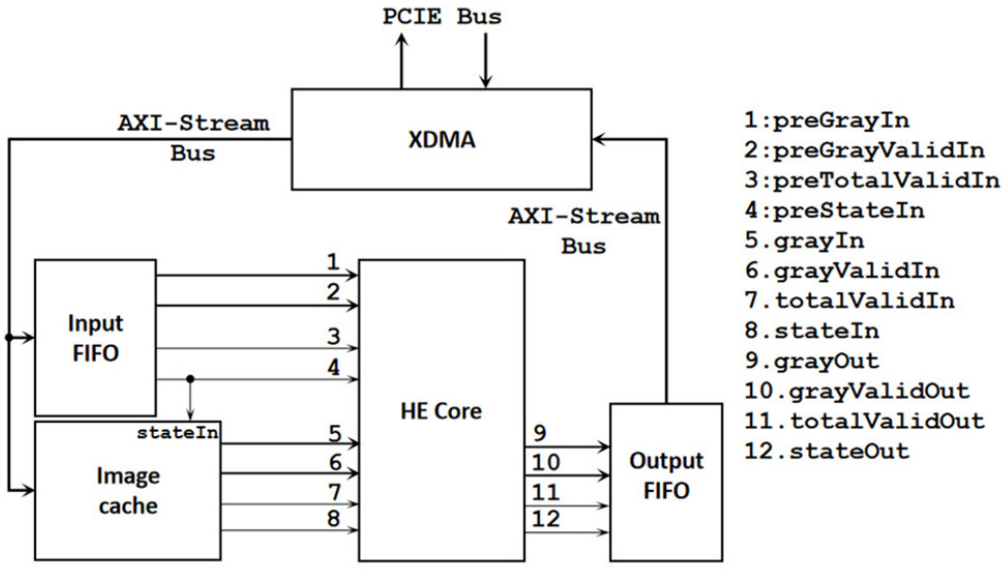


Figure 10. Overall architecture utilizing CL, DSP, and BRAM.

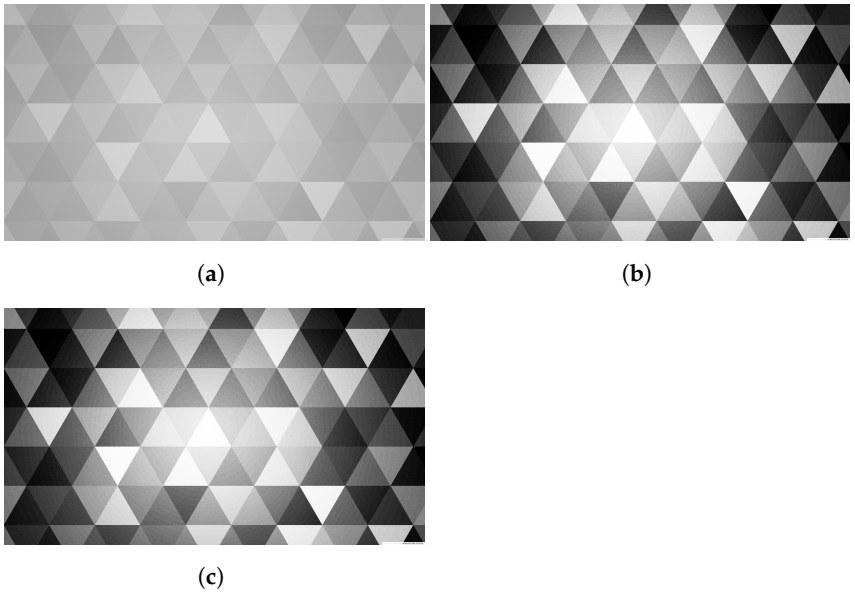
5. Experiments

5.1. Experimental Environment

The VCU 118 evaluation board platform was used in this experiment. The PCIe interface of the evaluation board is PCIe3.0*16. The FPGA on the VCU118 contains 2.6 million logical units, 6840 DSPs, and 2160 Block RAM, with a total capacity of 75.9 MB. The chip has a 16 nm fabrication process, the maximum frequency of the clock buffer and DSP is 775 MHz, and the maximum frequency of the Block RAM is 737 MHz. The XDMA AXI side operates at 250 MHz.

5.2. Experimental Results

Figure 11 shows the picture before conversion and the picture after conversion with FPGAs. The converted image is consistent with the openCV's histogram equalization output image, which proves that the running result is correct.



(**a**) (**b**)

(**c**)

Figure 11. (**a**) Figure before histogram equalization. (**b**) Figure after histogram equalization with FPGAs. (**c**) OpenCV histogram equalization output image.

5.3. Resource Utilization of the Implementations with Different Parameters

This experiment tests the resource utilization of the three architectures with different resolutions and different numbers of PCIe lanes.

Figure 12 shows the resource utilization of LUT.

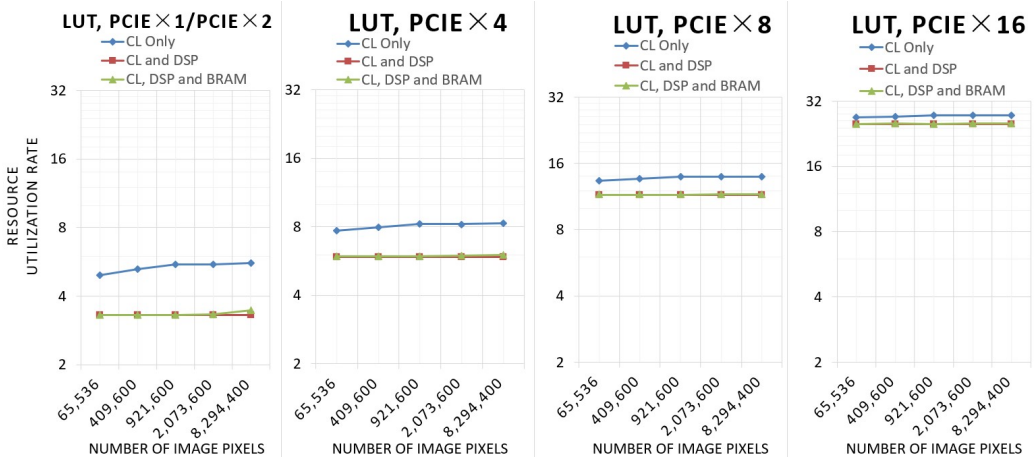


Figure 12. Resource utilization of LUT.

As the number of pixels rises, the LUT utilization of the architecture using only configurable logic (CL) increases, while the LUT utilization of the other two architectures utilizing DSP remains basically the same. Due to the fact that a larger number of image pixels requires a wider bit width calculation when utilizing only CL, the number of DSPs and the computing bit width utilizing CL and DSP and utilizing CL, DSP, and BRAM remain the same. In the architecture utilizing CL, DSP, and BRAM, the utilization of LUTs will be slightly higher at 4 k resolution because a portion of the LUTs will be used to route FIFOs with high BRAM utilization.

Figure 13 shows the resource utilization of flipflop.

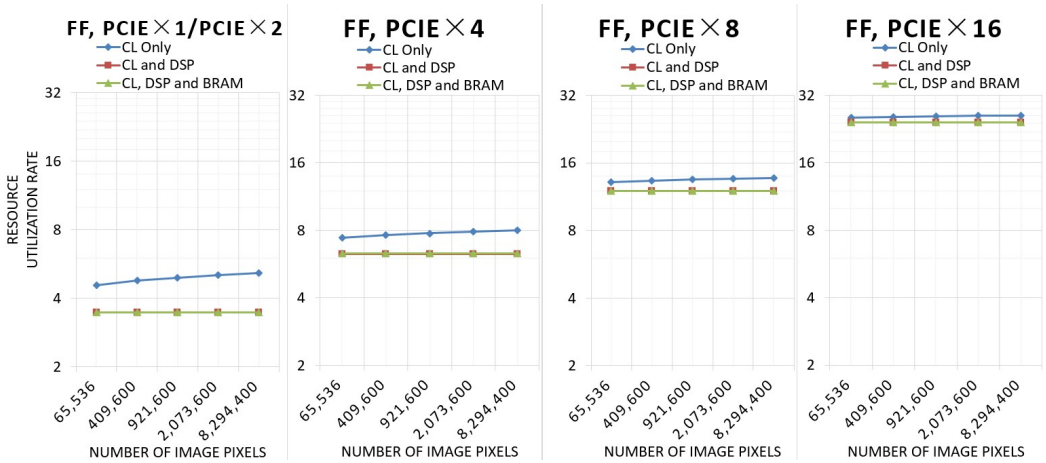


Figure 13. Resource utilization of flipflop.

The situation of flipflop utilization is similar to that of LUT utilization. The architecture utilizing only CL uses more flipflops as the computational bit width increases, while the other two architectures are unchanged. However, the architecture utilizing CL, DSP, and BRAM does not need to use more flipflops at 4k resolution, so the utilization of flipflops will not increase.

Figure 14 shows the resource utilization of BRAM.

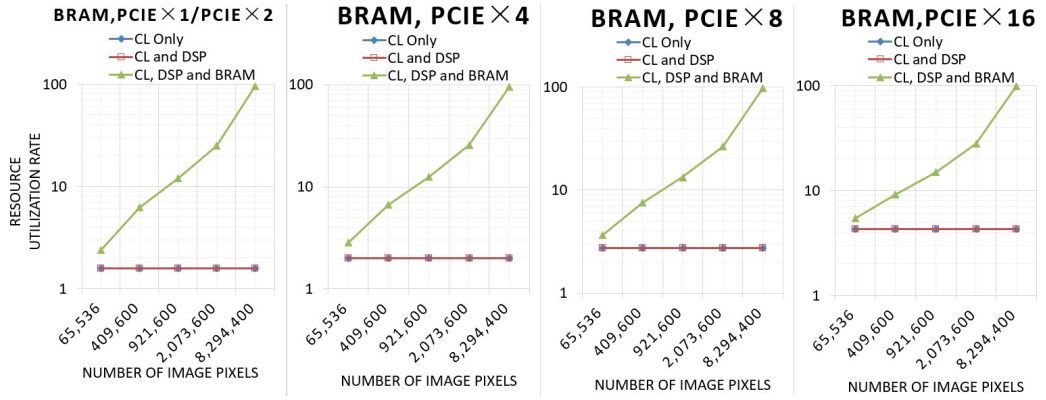


Figure 14. Resource utilization of BRAM.

The architecture using only CL and the architecture using CL and DSP hardly utilize any BRAM resources, while the BRAM utilization of architectures using CL, DSP, and BRAM will increase significantly with the increase in the number of pixels.

As the number of PCIe channels increases, the LUT, FlipFlop, and BRAM utilization of the three architectures increases. However, the utilization of DSP for the architecture utilizing CL and DSP and the architecture utilizing CL, DSP, and BRAM is 7.49%, independent of the number of inputs and the total number of pixels.

5.4. Performance Comparison with CPU and Other FPGA-Based Implementations under Different Configurations

5.4.1. Comparison with Single-Core CPU

The CPU model is an i7-8750H, and the HE is realized using the OpenCV framework. The OpenCV framework uses only one core of the CPU. This performance test method measures the difference between the running time of 11 images and 1 image, divides by 10 to obtain the running result of each image, and takes the average result of 10 tests.

Figure 15 shows the acceleration rate of this implementation for a single-core CPU.

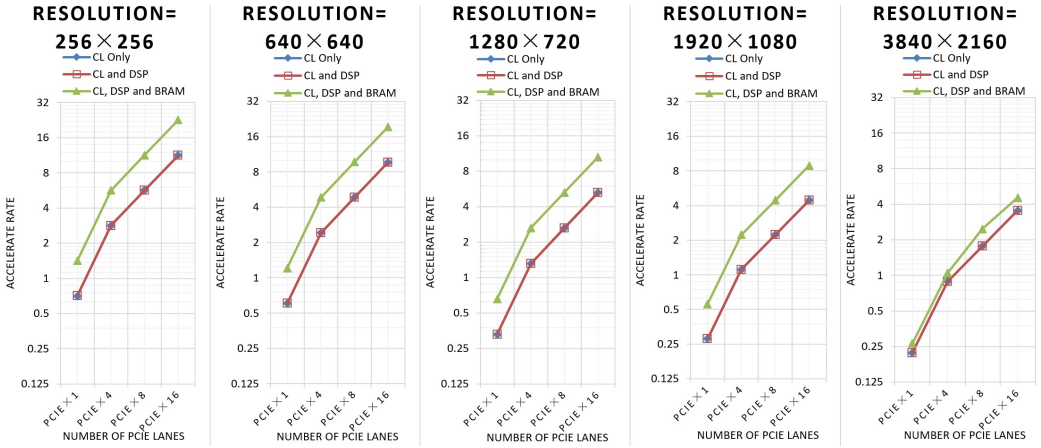


Figure 15. Acceleration rate for a single-core CPU.

When the resolution of the image is 3840 × 2160, in the architecture using CL, DSP, and BRAM, the BRAM utilization rate is close to 100%, so the running frequency will be decreased, and the performance will not be doubled compared with the other two architectures.

5.4.2. Comparison with Other FPGA-Based Implementations

A comparison with other FPGA-based implementations is shown in Table 1. The acceleration rate is the performance of proposed implementation divided by the performance of other FPGA-based implementations. Acceleration ratios take into account the impact of different models on performance.

Table 1. Acceleration rate of the proposed method (PCIe*16) and other FPGA-based implementations (image size: 256 × 256).

Other FPGA-Based Implementations	Model of FPGA	Frequency (MHz)	Acceleration Rate (Utilizing CL, CL, and DSP)	Acceleration Rate (Utilizing CL, DSP, and BRAM)
Hazra et al. [16]	XC7V2000T	368	21.4×	43.0×
Alsuwailem et al. [17]	EP2S15F484C3	250	31.5×	63.3×
Sachdeva et al. [18]	unknown	250	31.5×	63.3×
Li et al. [20]	xc4vsx35-10ff668	200	39.4×	79.1×
Sadad et al. [21]	XC7A100T	100	78.7×	158.2×
Li et al. [22]	XC4010	50	1573.8×	3163.0×
Lianfa et al. [23]	Flex10k10	3.33	2360.7×	4744.5×
Salcic et al. [25]	Flex8000	30.0	262.3×	527.2×

Compared to the fastest single-input FPGA-based implementation [16], the performance of the architecture utilizing CL and the architecture using CL and DSP is about 21.4 times better, and the performance of the architecture utilizing CL, DSP, and BRAM is about 43.0 times better (65,536 pixels under PCIe×16).

Other architectures with multiple inputs do not mention a specific frequency, so we consider them architecturally. The architecture in [26] is difficult to run at very high frequency due to the need to connect modules at a long distance, the high fan-out degree, and the complex wiring. The architecture in [27,28] does not really support multiple inputs and outputs at the same time.

The previous comparisons are based on works accelerating the same algorithm, but on different hardware platforms, as they are finished at different times. There is also work using the ultrascale+ architecture of FPGAs [33], which implements the Contrast-Limited Adaptive Histogram Equalization (CLAHE) algorithm. In this work, the authors

redesigned a vector stream format (4 ppc) to implement the CLAHE algorithm, which enables the processing of a 4K video stream at 60 fps. Our work can reach 964 fps for 4K images. This difference is mainly caused by the difference in complexity between the two algorithms. We both designed optimized FPGAs architectures for accelerating CLAHE and HE. Therefore, users can select which one to use according to their requirements. In addition, the innovative methods in [33] can also be references for our future research.

6. Discussion and Conclusions

In order to improve the calculation performance of HE, we optimized HE according to the characteristics of FPGAs and proposed a two-dimensional pipeline architecture which parallelizes HE in two dimensions. At the same time, the complexity of the connection is not high, so the FPGAs can run at a high frequency. The optimized HE is implemented on FPGAs, and the number of inputs, input bit width, image size, and resource usage strategy can be configured. Pipelining and critical path optimization have been performed for the computing unit so that it can run at higher frequencies.

Our proposed methods can improve the computational speed of HE. Compared with the single-core CPU i7-8750H, the acceleration ratio of the PCIe3.0*16 interface can be up to 22.6 times higher. Compared with other FPGA-based implementations of single-input architectures (based on Zynq-XC7Z030) [16], the performance of the proposed architecture is 43.0 times higher.

Our work mainly aimed to achieve better algorithm acceleration performance rather than optimized resource utilization and power consumption. Therefore, in the performance comparison section, we only compared the acceleration ratio and frequency, without taking into account resource utilization and power consumption.

The current acceleration ratio is mainly limited by the speed of the IO interface. Secondary factors affecting the performance are the maximum BRAM operating frequency and the BRAM capacity. A single HE core can run at the maximum frequency of the FPGA (4k, 32 inputs, based on xcvu13p@891MHz), except for when BRAM effects are present.

Future improvements can use the latest PCIe interface. If we switch to a later version of PCIe, the acceleration ratio will be greater, ideally up to 90× based on PCIe5.0. Faster BRAM with a larger capacity can also be used to ensure that the whole circuit runs at a very high frequency.

Author Contributions: Conceptualization, P.L. and D.L.; methodology, P.L. and D.L.; software, P.L. and D.L.; validation, P.L. and D.L.; formal analysis, Y.W.; investigation, Y.W.; resources, K.W. and R.Z.; data curation, K.W. and R.Z.; writing—original draft preparation, P.L.; writing—review and editing, D.L. and P.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Natural Science Foundation of China (No. 62072212), the Development Project of Jilin Province of China (Nos. 20220508125RC, 20200201290JC), the National Key R&D Program (No. 2018YFC2001302), the Jilin Provincial Key Laboratory of Big Data Intelligent Cognition (No. 20210504003GH), the Universities' Key Scientific Research Platforms and Projects (grant number 2021ZDZX1083), and the Guangdong Key Disciplines Project (grant number 2022ZDJS139).

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Kaur, M.; Sarkar, R.K.; Dutta, M.K. Investigation on quality enhancement of old and fragile artworks using non-linear filter and histogram equalization techniques. *Optik* **2021**, *244*, 167564. [CrossRef]
2. Fridman, Y.; Rusanovsky, M.; Oren, G. ChangeChip: A reference-based unsupervised change detection for PCB defect detection. In Proceedings of the 2021 IEEE Physical Assurance and Inspection of Electronics (PAINE), Washington, DC, USA, 30 November–2 December 2021; pp. 1–8.

3. Bi, Z.; Yu, L.; Gao, H.; Zhou, P.; Yao, H. Improved VGG model-based efficient traffic sign recognition for safe driving in 5G scenarios. *Int. J. Mach. Learn. Cybern.* **2021**, *12*, 3069–3080. [CrossRef]
4. Roy, S.; Bhalla, K.; Patel, R. Mathematical analysis of histogram equalization techniques for medical image enhancement: A tutorial from the perspective of data loss. *Multimed. Tools Appl.* **2023**, 1–30. [CrossRef]
5. Saba, T. Computer vision for microscopic skin cancer diagnosis using handcrafted and non-handcrafted features. *Microsc. Res. Tech.* **2021**, *84*, 1272–1283. [CrossRef] [PubMed]
6. Pan, H.; Lan, J.; Wang, H.; Li, Y.; Zhang, M.; Ma, M.; Zhang, D.; Zhao, X. UWV-Yolox: A Deep Learning Model for Underwater Video Object Detection. *Sensors* **2023**, *23*, 4859. [CrossRef] [PubMed]
7. Dhal, K.G.; Das, A.; Ray, S.; Gálvez, J.; Das, S. Histogram equalization variants as optimization problems: A review. *Arch. Comput. Methods Eng.* **2021**, *28*, 1471–1496. [CrossRef]
8. Ashiba, H.; Sadic, N.; Hassan, E.S.; El-Dolil, S.; Abd El-Samie, F.E. New Proposed Algorithms for Infrared Video Sequences Non-uniformity Correction. *Wirel. Pers. Commun.* **2022**, *126*, 1051–1073. [CrossRef]
9. Biswas, T.; Bhattacharya, D.; Mandal, G. Dynamic strategy to use optimum memory space in real-time video surveillance. *J. Ambient. Intell. Humaniz. Comput.* **2023**, *14*, 2771–2784. [CrossRef]
10. Wang, L.; Jia, H.; Zhang, Y.; Li, K.; Wei, C. EgpuIP: An Embedded GPU Accelerated Library for Image Processing. In Proceedings of the 2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys), Chengdu, China, 18–20 December 2022; pp. 914–921.
11. Aguirre-Castro, O.; García-Guerrero, E.; López-Bonilla, O.; Tlelo-Cuautle, E.; López-Mancilla, D.; Cárdenas-Valdez, J.; Olguín-Tiznado, J.; Inzunza-González, E. Evaluation of underwater image enhancement algorithms based on Retinex and its implementation on embedded systems. *Neurocomputing* **2022**, *494*, 148–159. [CrossRef]
12. Tang, C.; Xia, S.; Qian, M.; Wang, B. Deep learning-based vein localization on embedded system. *IEEE Access* **2021**, *9*, 27916–27927. [CrossRef]
13. Boutros, A.; Betz, V. FPGA Architecture: Principles and Progression. *IEEE Circuits Syst. Mag.* **2021**, *21*, 4–29. [CrossRef]
14. Gao, M.; Li, S.; Zhu, L.; Bai, Y.; Wang, P.; Guan, N.; Wang, K.; Yin, H. An FPGA-based real-time infrared target detection system with visual image positioning. In Proceedings of the AOPC 2022: Optical Sensing, Imaging, and Display Technology, Online, 18–19 December 2023; Volume 12557, pp. 284–289.
15. AMD. AMD Virtex UltraScale+ FPGA VCU118 Evaluation Kit. Available online: https://www.xilinx.com/products/boards-and-kits/vcu118.html (accessed on 17 August 2023).
16. Hazra, S.; Ghosh, S.; Maity, S.P.; Rahaman, H. A new FPGA and programmable soc based VLSI architecture for histogram generation of grayscale images for image processing applications. *Procedia Comput. Sci.* **2016**, *93*, 139–145. [CrossRef]
17. Alsuwailem, A.M.; Alshebeili, S.A. A new approach for real-time histogram equalization using FPGA. In Proceedings of the 2005 International Symposium on Intelligent Signal Processing and Communication Systems, Hong Kong, China, 13–16 December 2005; pp. 397–400.
18. Sachdeva, N.; Sachdeva, T. An FPGA Based Real-time Histogram Equalization Circuit for Image Enhancement. *Int. J. Electron. Commun. Technol.* **2010**, *1*, 63–67.
19. Alsuwailem, A. A Novel FPGA Based Real-time Histogram Equalization Circuit for Infrared Image Enhancement. *J. Act. Passiv. Electron. Devices* **2008**, *3*, 311–321.
20. Li, Y.; Zhong, S.; Wang, B.; Yan, L.; Liu, T. An FPGA-based histogram equalization implementation. In Proceedings of the MIPPR 2009: Medical Imaging, Parallel Processing of Images, and Optimization Techniques, Yichang, China, 30 October–1 November 2009; Volume 7497, pp. 198–201.
21. Sadad, N.U.; Afrin, A.; Mondal, M.N.I. FPGA based Histogram Equalization for Image Processing. In Proceedings of the 2021 3rd International Conference on Electrical & Electronic Engineering (ICEEE), London, UK, 7–9 July 2021; pp. 89–92.
22. Li, X.; Ni, G.; Cui, Y.; Pu, T.; Zhong, Y. Real-time image histogram equalization using FPGA. In Proceedings of the Electronic Imaging and Multimedia Systems II. International Society for Optics and Photonics, Beijing, China, 18–19 September 1998; Volume 3561, pp. 293–299.
23. Lianfa, B.; Xing, L.; Qian, C.; Baomin, Z. The hardware design of real-time infrared image enhancement system. In Proceedings of the International Conference on Neural Networks and Signal Processing, Nanjing, China, 14–17 December 2003; Volume 2, pp. 1009–1012.
24. Gu, D.; Yang, N.; Pi, D.; Hua, M.; Shen, X.; Zhang, R. DSP+ FPGA-based real-time histogram equalization system of infrared image. In Proceedings of the Semiconductor Optoelectronic Device Manufacturing and Applications, Nanjing, China, 7–9 November 2001; Volume 4602, pp. 160–165.
25. Salcic, Z.; Sivaswamy, J. IMECO: A reconfigurable FPGA-based image enhancement co-processor framework. *Real-Time Imaging* **1999**, *5*, 385–395. [CrossRef]
26. Gautam, K.S. Parallel histogram calculation for FPGA: Histogram calculation. In Proceedings of the 2016 IEEE 6th International Conference on Advanced Computing (IACC), Bhimavaram, India, 27–28 February 2016; pp. 774–777.
27. Jamro, E.; Wielgosz, M.; Wiatr, K. FPGA implementaton of strongly parallel histogram equalization. In Proceedings of the 2007 IEEE Design and Diagnostics of Electronic Circuits and Systems, Krakow, Poland, 11–13 April 2007; pp. 1–6.

28. Shahbahrami, A.; Hur, J.Y.; Juurlink, B.; Wong, S. FPGA implementation of parallel histogram computation. In Proceedings of the 2nd HiPEAC Workshop on Reconfigurable Computing, Gothenborg, Sweden, 27–29 January 2008; pp. 63–72.
29. Memon, F.; Jameel, F.; Arif, M.; Memon, F. Model based FPGA design of histogram equalization. *Sindh Univ. Res. J.* **2016**, *48*, 435–440.
30. Soma, P.; Sravanthi, C.; Srilakshmi, P.; Jatoth, R.K. Implementation of Single Image Histogram Equalization and Contrast Enhancement on Zynq FPGA. In *Advances in Communications, Signal Processing, and VLSI*; Springer: Singapore, 2021; pp. 75–82.
31. Bhaumik, V.; Mustafa, S.; Parth, V.; Jay, B.; Jenish, J. Hardware acceleration of image processing algorithms using Vivado high level synthesis tool. In Proceedings of the 2017 International Conference on Intelligent Computing and Control Systems (ICICCS), Melur, Madurai, 15–16 June 2017; pp. 29–34.
32. Zunin V.V. Intel OpenVINO Toolkit for Computer Vision: Object Detection and Semantic Segmentation. In Proceedings of the 2021 International Russian Automation Conference (RusAutoCon), Sochi, Russia, 5–11 September 2021; pp. 847–851.
33. Kryjak, T.; Blachut, K.; Szolc, H.; Wasala, M. Real-Time CLAHE Algorithm Implementation in SoC FPGA Device for 4K UHD Video Stream. *Electronics* **2022**, *11*, 2248. [CrossRef]