sensors

MDPI

Article

Fully Parallel Implementation of Otsu Automatic Image Thresholding Algorithm on FPGA

Wysterlânya K. P. Barros [1], Leonardo A. Dias [2] and Marcelo A. C. Fernandes [1,3,*]

[1] Laboratory of Machine Learning and Intelligent Instrumentation, nPITI/IMD, Federal University of Rio Grande do Norte, Natal 59078-970, Brazil; kyurybarros@gmail.com
[2] Centre for Cyber Security and Privacy, School of Computer Science, University of Birmingham, Birmingham B15 2TT, UK; l.a.dias@bham.ac.uk
[3] Department of Computer and Automation Engineering, Federal University of Rio Grande do Norte, Natal 59078-970, Brazil
* Correspondence: mfernandes@dca.ufrn.br

Abstract: This work proposes a high-throughput implementation of the Otsu automatic image thresholding algorithm on Field Programmable Gate Array (FPGA), aiming to process high-resolution images in real-time. The Otsu method is a widely used global thresholding algorithm to define an optimal threshold between two classes. However, this technique has a high computational cost, making it difficult to use in real-time applications. Thus, this paper proposes a hardware design exploiting parallelization to optimize the system's processing time. The implementation details and an analysis of the synthesis results concerning the hardware area occupation, throughput, and dynamic power consumption, are presented. Results have shown that the proposed hardware achieved a high speedup compared to similar works in the literature.

Keywords: FPGA; image segmentation; thresholding algorithm; Otsu's method

Citation: Barros, W.; Dias, L.; Fernandes, M. Fully Parallel Implementation of Otsu Automatic Image Thresholding Algorithm on FPGA. *Sensors* **2021**, *21*, 4151. https://doi.org/10.3390/s21124151

Academic Editor: Marcin Ciecholewski

Received: 16 May 2021
Accepted: 11 June 2021
Published: 17 June 2021

1. Introduction

In recent years, there has been an increase in computational solutions employing Digital Image Processing (DIP) techniques, such as facial recognition, medical image enhancement, signature authentication, traffic control, autonomous cars, and product quality analysis [1–5]. These applications usually require real-time processing. However, meeting their processing time requirements can be complex due to the large volume of data to be processed, which is proportional to the image resolution, color depth, and, in the case of video applications, the frame rate employed. Therefore, obtaining results in real-time has become a challenge [6].

Some of the mentioned applications use segmentation algorithms to identify the image's region of interest and classify its pixels as background or object. Thresholding is one of the main image segmentation techniques in which pixels are classified based on their intensity values [7]. The Otsu algorithm, proposed in [8], is a widely used global thresholding technique, which proposes the definition of an optimal threshold by maximizing the between-class variance. However, the Otsu algorithm has a high computational cost due to the complex arithmetic operations performed interactively, hindering its use in real-time applications.

Many works in the literature proposed the Otsu algorithm developed in hardware, such as Field-Programmable Gate Arrays (FPGA), to overcome the processing time constraints. This therefore allows applications to achieve real-time or near real-time processing. The FPGA allows the exploitation of the algorithm parallelization and the development of dedicated hardware to obtain performance improvement [9–15]. However, FPGA implementations found in the literature are often developed with sequential processing schemes in some stages of the Otsu algorithm, limiting the hardware's processing speed [16–21].

Therefore, this work proposes a fully parallel FPGA implementation of the Otsu algorithm. Unlike most approaches proposed in the literature, a full-parallel implementation reduces the bottleneck for processing speed compared to sequential systems or hybrid hardware architectures, that is, architectures implemented with sequential and parallel schemes. Besides, given the continuous increase in the volume of data present in the DIP applications, a full-parallel strategy is less likely to become obsolete quickly.

The remainder of this paper is organized as follows: Section 2 presents the related works in the literature; Section 3 addresses the theoretical foundation of the Otsu method; Section 4 shows a detailed description of the architecture proposed in this paper; while Section 5 presents and analyzes the synthesis results obtained from the described implementation, including a comparison to other works. Finally, Section 7 presents the final considerations.

2. Related Works

Many proposals can be found in the literature for real-time applications of the Otsu algorithm deployed in FPGAs. In [22], an adaptive lane departure detection and alert system is presented, while in [23], a lane departure and frontal collision warning system. Meanwhile, in [24], a vision system is presented to detect obstacles and locate a robot that navigates indoors; in [25], a system for detecting moving objects is presented; [26], presents a system to assist in the diagnosis of glaucoma; and, in [27], a system for improving thermograms is presented. However, these articles provide few details about the hardware implementation.

Among the first FPGA implementations of the Ostu algorithm is the proposal of [16], synthesized for an Altera Cyclone II FPGA. The design improved the algorithm's performance through a hybrid hardware architecture and Altera MegaCores, eliminating complex divisions and multiplications of the algorithm. The architecture developed was used for the segmentation of an image with a resolution of 320×240 and pixel represented by 10 bits. Through visual segmentation results, they evaluated the implementation performance as satisfactory.

Other works have proposed a hardware implementation using logarithmic functions to eliminate the division and multiplication circuits. In [17,19], the authors implemented two versions of the Otsu algorithm, in which one version uses the logarithmic functions, to compare the results achieved between them. The architectures developed by [17] were synthesized for a Xilinx Virtex XCV800 HQ240-4 FPGA. The implementation without the logarithmic functions occupied a hardware area of 622 slices and 103 Input/Output Blocks (IOBs), obtaining a clock latency of 362.4ns. Meanwhile, the logarithmic function implementation occupied 109 slices and 49 IOBs, obtaining a clock latency of 132ns.

The architectures presented by [19] were developed on the Altera Cyclone IV EP4CE115F29C6N FPGA. The synthesis results obtained for the algorithm implemented without the logarithmic functions occupied 6525 logic elements, 4920 registers, 18,266 bits of memory, and 79 multipliers of 9 bits. Regarding the processing time, considering an image with a resolution of 1280×1024, the system achieved a maximum frequency of 130.89 MHz and latency of 589 clock cycles. In contrast, the implementation with the logarithmic function used 2440 logic elements, 1026 registers, 10,943 bits of memory, and 79 multipliers of 9 bits. Moreover, it achieved a maximum frequency of 114.62 MHz and a latency of 536 clock cycles. Therefore, the results presented in [17,19] indicate that the algorithm designed with logarithmic function reduced the FPGA area occupation and latency.

In [18,20], similar architectures of the Otsu algorithm in the Virtex-5 xc5vfx70t ffg1136-1 FPGA were deployed, available on the Xilinx ML-507 development platform. Both proposals were developed in VHDL, using fixed-point representation, operating at a clock frequency of 25.175MHz. The proposal described in [18] occupied 168 slices and 33 IOBs, while in [20], the implementation reached an area occupation of 161 slices, 21 IOBs, 72 Look-Up Tables (LUTs), 591 registers, 4 blocks of RAM (BRAMs), and 5 DSP48Es. In addition, Reference [20] presented results related to the processing time for a 640×480 image, with

pixels represented by 8 bits. This work reached a latency of 5 clock cycles and throughput of 40.28 megabits processed per second (Mbps).

Meanwhile, it was presented in [21] an implementation for binarization and thinning of fingerprint images, using the Otsu method, on a Spartan 6 LX45 FPGA. Concerning the area occupation, 1898 registers, 1859 LUTs, 735 slices, 10 IOBs and 44 BRAMs were used. Regarding processing time, a maximum clock frequency of 100MHz was achieved, with the execution time of 1489 ms for processing a 280×265 image and latency of 531 clock cycles or 5310 ns. Besides, a comparison with the same technique implemented in Matlab was also presented, showing that the FPGA was $\approx 10\times$ faster than the Matlab version.

Thus, this work proposes an FPGA implementation of the Otsu algorithm to improve its performance. Unlike the works presented in the literature, the architecture proposed here uses a fully parallel scheme. The hardware implementation was developed in *Register-Transfer Level* (RTL), using fixed-point representation, in an Arria 10 GX 1150 FPGA. The results concerning the hardware area occupation and throughput are also presented.

3. Otsu's Algorithm

The Otsu is one of the most popular thresholding algorithms, used to find an optimal threshold that separates an image into two classes: the background and object. These classes are represented by C_0 and C_1, respectively. This method has the advantage of performing all its calculations based only on the histogram of the image [7,8].

Initially, the algorithm starts by calculating the normalized histogram of an image, $\mathbf{A}$, in grayscale, which is described as

$$\mathbf{A}(m) = \begin{bmatrix} a_{0,0}(n) & \cdots & a_{0,j}(n) & \cdots & a_{0,M-1}(n) \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{i,0}(n) & \cdots & a_{i,j}(n) & \cdots & a_{i,M-1}(n) \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{N-1,0}(n) & \cdots & a_{N-1,j}(n) & \cdots & a_{N-1,M-1}(n) \end{bmatrix} \tag{1}$$

where $a_{i,j}$ is one pixel of b bits and $M \times N$ is the image dimension. The pixels can assume L distinct integer intensity levels, represented by k and characterized as a value in a range of 0 to $L-1$, where $L = 2^b$. Each n-th pixel is processed in an instant t_s, which represents the sampling time. Thus, one complete image can be processed at every m-th moment, where

$$m = M \times N \times t_s. \tag{2}$$

This equation must be changed if more than one pixel is processed per sample time.

The histogram of each m-th image, $\mathbf{A}(m)$, is calculated and stored in the vector

$$\mathbf{p}(m) = [p_0(m), \ldots, p_k(m), \ldots, p_{L-1}(m)] \tag{3}$$

where each k-th component, $p_k(m)$, is defined as

$$p_k(m) = \frac{n_k(m)}{M \times N}, \tag{4}$$

in which $n_k(m)$ denotes the number of pixels with intensity k of the m-th image, described as

$$n_k(m) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} c_{i,j}(n), \tag{5}$$

with $c_{i,j}(n)$ expressed as

$$c_{i,j}(n) = \begin{cases} 1 & \text{if } a_{i,j}(n) = k \\ 0 & \text{otherwise} \end{cases}. \tag{6}$$

Subsequently, after obtaining the normalized histogram, stored in the vector **p**, the Otsu algorithm calculates an optimal threshold between the two classes, i.e., C_0 and C_1. The optimal threshold called here as $k^*(m)$ can be characterized as

$$k^*(m) = \underset{0 \leq k \leq L-1}{\arg\max}\, \sigma_k^2(m). \tag{7}$$

where $\sigma_k^2(m)$ is the k-th between-class variance of the m-th image, defined as

$$\sigma_k^2(m) = \frac{(\mu_{L-1}(m) \times \omega_k(m) - \mu_k(m))^2}{\omega_k(m)(1 - \omega_k(m))}, \tag{8}$$

where $\omega_k(m)$ and $\mu_k(m)$ are the probability of class occurrence given a k threshold and the mean intensity value of the pixels up to the k threshold of the m-th image, respectively, meanwhile, $\mu_{L-1}(m)$ is the average intensity of the entire m-th image, called the global mean, with a value equal to $\mu_k(m)$ when $k = L - 1$.

The variables $\omega_k(m)$ and $\mu_k(m)$ can be expressed as

$$\omega_k(m) = \sum_{i=0}^{k} p_i(m) \tag{9}$$

and

$$\mu_k(m) = \sum_{i=0}^{k} i \cdot p_i(m). \tag{10}$$

After finding the optimal threshold value, $k^*(m)$, the pixels of the input image, $\mathbf{A}(m)$, can be classified as a background or object (C_0 and C_1), generating a mask for the input image.

4. Hardware Proposal

This work proposes a fully parallel architecture of the Otsu method capable of processing images of any dimension, focused on obtaining high-speed processing. The details of the hardware implementation are described in the following subsections.

4.1. General Architecture

The general hardware architecture implemented for the Otsu algorithm is presented through a block diagram, shown in Figure 1. As can be observed, the architecture was developed based on the description presented in Section 3. Therefore, it consists of five main modules: Normalized Histogram Module (NHM), Probability of Class Occurrence Module (PCOM), Mean Intensity Module (MIM), Between-Class Variance Module (BCVM), and Optimal Threshold Module (OTM).

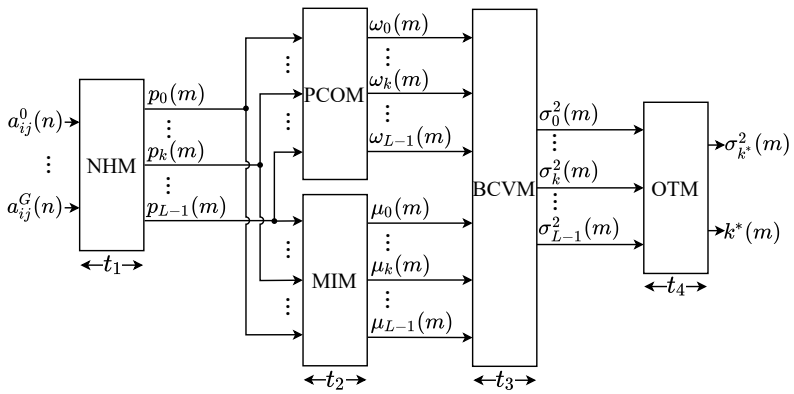

Figure 1. General architecture of the proposed Otsu Algorithm implementation.

Initially, the NHM module receives the parallel input of G image pixels, where G is the number of submodules internal to the NHM that simultaneously calculate the normalized histogram, according to Equations (3) and (4). Subsequently, the PCOM module uses the histogram components to calculate the class occurrence probabilities, according to Equation (9), while the MIM module calculates the average intensities, based on Equation (10). The PCOM and MIM modules perform their calculations simultaneously. Afterward, these two modules' outputs are supplied to BCVM, in which the values of the between-class variance are computed, according to Equation (8). Finally, the calculated between-class variances are compared in the OTM to select the optimal threshold value, as described in Equation (7).

All variables and constants shown in Figure 1 were implemented in fixed point to reduce the bit-width compared to floating-point implementations, the bits number used can be adjusted to adapt the precision of the results obtained to the desired application. Therefore, each pixel, $a_{ij}^{g}(n)$, of the input image were configured with 8 bits in the integer part (without sign), then $L = 2^8 = 256$ is defined. For the histogram components, $p_k(m)$, and the probabilities of class occurrence, $\omega_k(m)$, which has a positive value less than 1, only 24 bits are used in the decimal part. For the average intensity elements, $\mu_k(m)$, 8 bits are used in the integer part (without sign) and 24 bits in the decimal part. For the between-class variances, $\sigma_k^2(m)$, 27 bits are used in the integer part (one bit for sign) and 24 bits in the decimal part. Finally, for the optimal threshold, $k^*(m)$, only 8 bits are used in the integer part (without sign).

The modules of this architecture are pipelined, and the system operates on the same sample time, t_s. Nonetheless, each module has a different execution time, characterized here as t_1, t_2, t_3 and t_4. To minimize control, due to the lack of synchronism between the modules, the hardware proposed here defines $t_1 = t_2 = t_3 = t_4 = t_I$, where t_I is the time to process a complete image, being equal to the m-th moment that an image is processed. The time t_I is defined by the NHM block, since t_1 has the longest execution time. Thus, the system has an initial latency expressed as

$$D = 4 \times t_I \tag{11}$$

and a throughput characterized as

$$th = 1/t_I. \tag{12}$$

4.2. Normalized Histogram Module (NHM)

The NHM is responsible for generating the normalized histogram of the input image, by performing the Equations (3) and (4). Usually, this step of the algorithm costs more clock cycles to complete than other steps, as the entire image needs to be scanned to obtain the histogram components. We propose the parallelization of this step by calculating the components' partial values in a parallel way to optimize this process. Afterward, these values are summed to obtain their final values. The architecture of this module is shown in Figure 2.

As can be observed, the NHM module is constituted of G identical submodules, called Partial Normalized Histogram (PNH), responsible for computing the partial values of the histogram components. Each g-th input pixel, $a_{ij}^{g}(n)$, is processed by the g-th PNH_g. Likewise, the PNH modules are internally constituted of L submodules, called Partial Component of the Normalized Histogram (PCNH), as shown in Figure 3. Thus, each k-th partial component of the histogram calculated by the g-th PNH, $p_k^g(n)$, is computed in parallel by a PCNH_k^g submodule. Figure 4 shows the internal circuit of each PCNH module, consisting of a comparator ($COMP_k^g$), an adder (SUM_k^g), two registers (R) and two constants (C_1 and C_2).

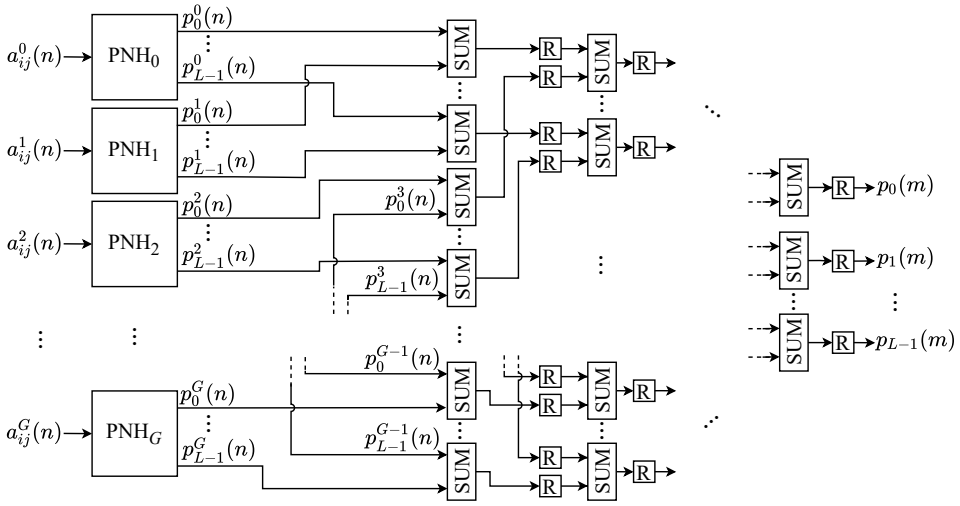

Figure 2. Architecture of the NHM.

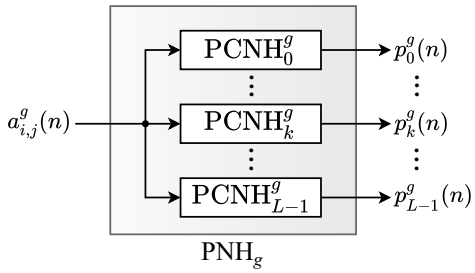

Figure 3. Architecture of the PNH$_g$.

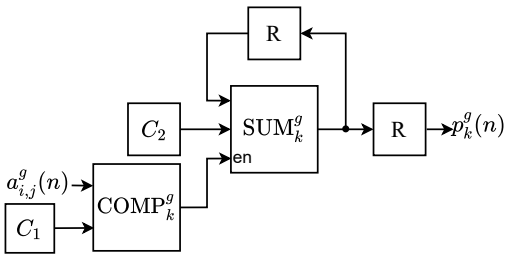

Figure 4. Architecture of the PCNH$_k^g$.

Initially, in each k-th comparator of the g-th PCNH$_k^g$, $COMP_k^g$, it is checked whether the input pixel, $a_{ij}^g(n)$, has value equal to C_1, according to Equation (6). The constant C_1 has a different value of k in each PCNH$_k^g$ submodule. Following, the $COMP_k^g$ output, $c_{i,j}(n)$, which is a Boolean value, enables the adder, SUM_k^g, when equal to 1. Therefore, when SUM_k^g is enabled, the constant C_2 is summed with its previous value, thus operating as an accumulator. The constant C_2 is defined as $\frac{1}{M \times N}$, so it determines the value of $p_k(m)$ when summed n_k times, according to Equation (4). After entering all the image pixels, each PCNH$_k^g$ outputs the k-th partial component of the normalized histogram computed in the g-th PNH, $p_k^g(n)$.

After that, the final value of each k-th component of the m-th image, $p_k(m)$, is obtained by summing all the k-th partial values provided as an output of each g-th PNH, $p_k^g(n)$. This sum is performed for each k-th component through an adder tree (SUM), as represented in Figure 2. This tree has a depth equal to $\log_2 G$. At the end, the value of the components of the normalized histogram, $p_k(m)$, is obtained, according to Equation (3).

Instead of processing 1 pixel per sample time in the histogram, the proposed architecture allows processing G pixels in parallel. Consequently, the amount of clock cycles

required in this step is reduced and, thus, the processing time of a complete image, t_I. Consequently, the latency is also reduced, and throughput increased. With this scheme, the value of t_I can be defined by

$$t_I = \left(\frac{M \times N}{G} + \lceil log_2(G) \rceil \right) \times t_s. \quad (13)$$

Through this equation, it is possible to observe that the higher the value of G, the better the performance obtained.

In each PCNH submodule, the constant C_1 assumes a grayscale value between 0 and $L-1$ and is represented with only 8 bits in the integer part (without sign). The constant C_2, which has a positive value less than 1, uses only 24 bits in the decimal part. This bit-width is also used for all the adders of the NHM module, as the normalized histogram components also assume positive values less than 1. All the k-th components of the histogram, $p_k(m)$, are transmitted in parallel to PCOM and MIM.

4.3. Mean Intensity Module (MIM)

The MIM calculates the average intensity value of the pixels up to level k, according to Equation (10). Each k-th average intensity, $\mu_k(m)$, is calculated in parallel. The architecture of this module, shown in Figure 5, consists of L gains submodules (G_k), $\frac{L}{2} \times \log_2 L$ adders (SUM) and one register (R) after each component.

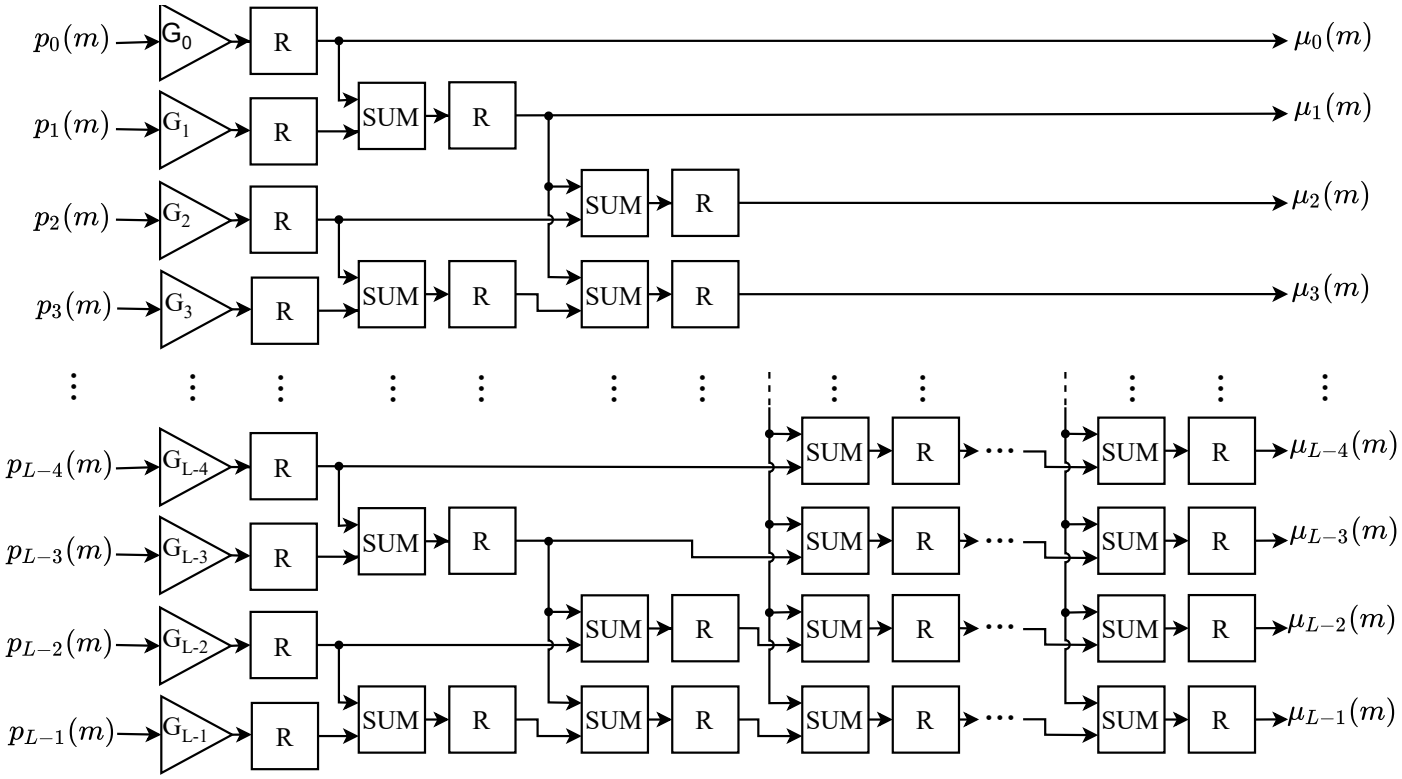

Figure 5. Architecture of the MIM.

Based on the Equation (10), each k-th $p_k(m)$ component of the normalized histogram is first multiplied by a gain with the value of k. These gains are represented in the architecture block diagram by $G_0, ..., G_{L-1}$, and the index indicates the value of the applied gains. Thereafter, each k-th average intensity, $\mu_k(m)$, is obtained by summing the outputs of all gains with an index from 0 to k. The sum of these values is carried out in parallel based on the adder proposed by [28], with a maximum of $\log_2 L$ cascading adders using this technique.

All k-th gains of this module, G_k, have their output represented with 8 bit in the integer part (without sign) and 24 bits in the decimal part. Similarly, this bit resolution is also used for the adders, SUM, as the average intensity values of the pixels are at most equal to $L = 256$, for input images with pixels represented by 8 bits. All k-th average intensities, $\mu_k(m)$, are provided to BCVM in parallel.

4.4. Probability of Class Occurrence Module (PCOM)

The probability of class occurrence for a given threshold k, $\omega_k(m)$, is performed in PCOM based on Equation (9). This module has an architecture similar to MIM, but it does not have the gain submodule to weight the input. Therefore, the inputs are directly linked to the adders (SUM). Thus, this architecture is composed only of adders and registers, as shown in Figure 5.

According to Equation (9), the $\omega_k(m)$ values are obtained by adding all the components of the histogram from index 0 to k. Thus, using the parallel adder proposed by [28], all $\omega_k(m)$ values are computed simultaneously through the sum of the k-th entries $p_k(m)$.

The adders in this module were implemented for the same bit resolution of the inputs, $p_k(m)$, since the probability of class occurrence also assumes positive values less than 1. All k-th probabilities $\omega_k(m)$ calculated are propagated to the BCVM in parallel.

4.5. Between-Class Variance Module (BCVM)

The k-th between-class variance of the m-th image, $\sigma_k^2(m)$, are calculated by BCVM based on the Equation (8). The BCVM module is internally composed of L equal submodules, named Between-Class Variance of k (BCV_k), with the same architecture shown in Figure 6. Each k-th $\sigma_k^2(m)$ is computed in parallel by the k-th submodule BCV_k. This submodule consists of four multipliers ($MULT_{i,k}$), two subtractors ($SUB_{i,k}$), a point shift (BPC_K), a Look-Up Table (LUT_k) and eleven registers (R).

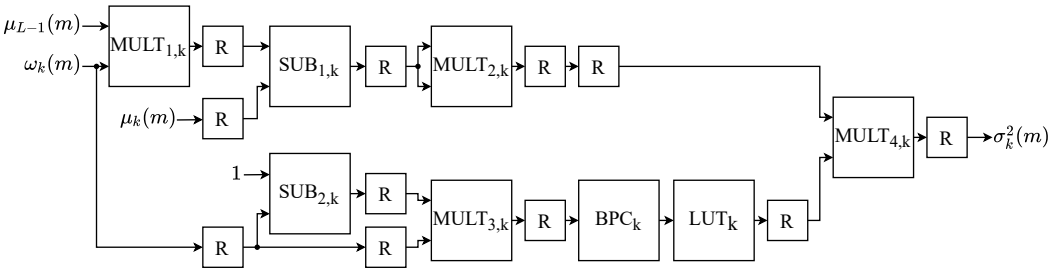

Figure 6. Architecture of the BCV_k submodule.

Equation (8) is performed in parallel in this architecture. Therefore, the numerator and denominator are performed simultaneously. Concerning the numerator, it is obtained by first multiplying the k-th probability, $\omega_k(m)$, by the global mean, $\mu_{L-1}(m)$, on $MULT_{1,k}$. Subsequently, the k-th $SUB_{1,k}$ submodule performs the subtraction between $\mu_k(m)$ and the $MULT_{1,k}$ output. Finally, the result of this subtraction is multiplied by itself in the k-th $MULT_{2,k}$. Regarding the denominator, it is calculated by initially subtracting the k-th probability, $\omega_k(m)$, from the value 1 in the k-th $SUB_{2,K}$. Lastly, the result is multiplied by the same $\omega_k(m)$ in the k-th $MULT_{3,k}$.

The division arithmetic operation is highly costly to the hardware in terms of processing speed, being the architecture's bottleneck due to the highest critical time. One way to avoid using the division is to multiply the numerator by the reciprocal of the denominator. By definition, the reciprocal of a number is its inverse. Thereupon, the denominator's reciprocal can be approximated for a range of predefined values and stored in a LUT. Thus, the division can be performed using only one LUT and a multiplier, LUT_k and $MULT_{4,k}$, consequently increasing the throughput of the implementation.

Thus, each k-th value in the output of $MULT_{3,k}$ has a reciprocal approximated value in the k-th LUT_k. This LUT was configured with a depth of L, storing words of 33 bits, where

9 bits represent the integer part (one bit for sign) and 24 bits the fractional. The mapping of the output value of each k-th $MULT_{3,k}$ to an address of the LUT_k is performed by shifting the binary point eight bits to the right by the k-th BPC_k. The approximate value of the reciprocal shown at the output of each k-th LUT_k is multiplied by the calculated value of the numerator in $MULT_{4,k}$. The result of this multiplication is the k-th between-class variance of the m-th image, $\sigma_k^2(m)$.

The subtractor $SUB_{1,k}$ was configured with 9 bits in the integer part (one bit for sign) and 24 bits in the decimal part, while $SUB_{2,k}$, uses only 24 bits in the decimal part. The multiplier $MULT_{1,k}$ uses 8 bits in the integer part (without sign) and 24 bits in the decimal part, while $MULT_{2,k}$ uses 18 bits in the integer part (one bit for sign) and 24 bits in the decimal part. Meanwhile, $MULT_{3,k}$ was configured with only 24 bits in the decimal part, and $MULT_{4,k}$ uses 27 bits in the integer part (one bit for sign) and 24 bits in the decimal part. Each k-th between-class variance calculated is propagated in parallel to the OTM.

4.6. Optimal Threshold Module (OTM)

The OTM module performs the last step of the Otsu algorithm, responsible for comparing all k-th values of the between-class variance, $\sigma_k^2(m)$, to determine the optimal threshold of the m-th image, $k^*(m)$, based on Equation (7). The architecture of this module is shown in Figure 7. As can be observed, it consists of $L-1$ comparators ($COMP_k$), $2 \times (L-1)$ multiplexers (MUX) and a register (R) after each component.

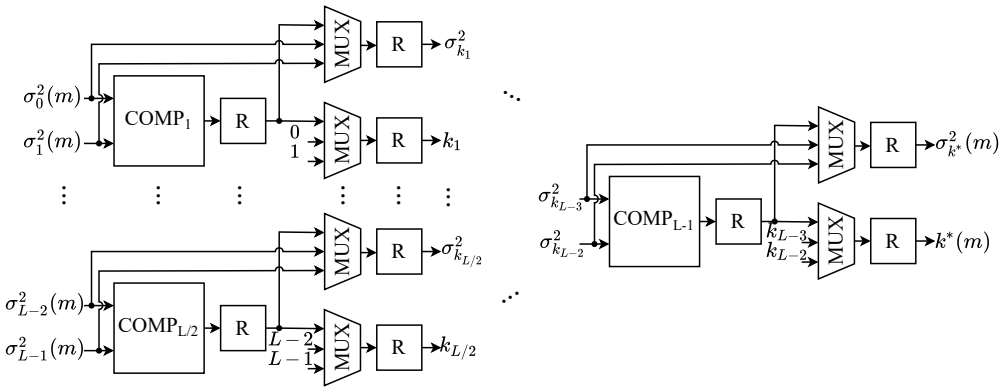

Figure 7. Architecture of the OTM.

According to Equation (7), the optimal threshold of the m-th image, $k^*(m)$, is the threshold value for which the highest value of between-class variance is obtained. For this purpose, the between-class variances of the m-th image, $\sigma_k^2(m)$, are compared through a comparator tree. Each k-th $COMP_k$ compares whether a given variance $\sigma_k^2(m)$ is greater than the other, $\sigma_{k+1}^2(m)$. This comparator is used as a key selector of two multiplexers, defining on the outputs the largest variance value that was compared and its respective threshold k. All outputs of the multiplexers are passed to the next branch of the tree until the last comparator, $COMP_{L-1}$, in which the optimal threshold value of the m-th image, $k^*(m)$, should be selected as the output of the multiplexers, as well as its between-class variance, $\sigma_{k^*}^2(m)$.

Therefore, the optimal threshold of the m-th input image is determined by the proposed architecture of a fully parallel design. A new value of $k^*(m)$ is computed for every m-th instant.

5. Results

The architecture presented in the previous section was developed on an FPGA Arria 10 GX 1150, and the analysis of the synthesis results was carried out concerning hardware area occupation, throughput, and power consumption.

5.1. Hardware Area Occupation Analysis

Initially, the hardware area occupation analysis was performed for the architecture with one PNH module only. The results are shown in Table 1. The first to the third columns indicate the number of logical cells occupied (n_{LC}), the number of multipliers implemented using DSP blocks (n_{Mult}), and the number of block memory bits (n_{BitsM}), respectively. It also presents the resources used in percentage.

Table 1. Hardware area occupation for $n_{PNH} = 1$.

n_{LC}	n_{Mult}	n_{BitsM}
591,946 (69.3%)	1222 (80.5%)	2162 K (3.9%)

As can be observed, only 4% of the block memory bits available have been used, while 69% of the logic cells were occupied. The most-used resource was the multipliers, occupying 80% of the total DSPs available. Therefore, the data presented in Table 1 demonstrate the feasibility of implementing the proposed architecture in the target FPGA. Besides, the Arria-10 FPGA still has resources available that can be used to implement additional logic, thus allowing an increase in the number of PNH modules.

PNH modules require only logical cells for their implementation since they are not designed with multipliers and memories. Therefore, the 30% of unused logic cells in the Arria-10 allow for the increasing of the number of PNH modules. Hence, we also analyzed the area occupation for the Arria-10 FPGA by increasing the number of PNH modules (n_{PNH}). The number of occupied logical cells is presented in Table 2, as there is no change in the use of other resources.

Table 2. Number of logic cells required per PNH module.

n_{PNH}	n_{LC}
1	591,946 (69.3%)
2	629,096 (73.6%)
4	680,884 (79.7%)
6	740,368 (86.6%)
8	788,092 (92.2%)
10	848,630 (99.3%)

Figure 8 shows the curve obtained by linear regression using the set of values presented in Table 2. The equation associated with the regression analysis is expressed by

$$n_{LC} = \left\lceil (2.8 \times n_{PNH} + 56.9) \times 10^4 \right\rceil. \quad (14)$$

This equation can obtain the number of logical cells occupied by the architecture without the NHM module when defining $n_{PNH} = 0$.

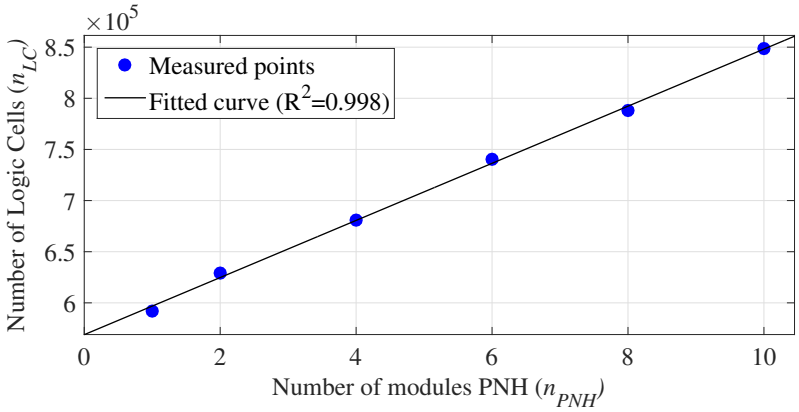

Figure 8. Linear regression curve for the hardware area occupation (logic cells) per n_{PNH}.

Therefore, Equation (14) allows for the estimation of the maximum number of PNH modules an FPGA can support. Table 3 presents the maximum number of PNH modules supported on different commercial FPGAs [29,30]. The first and second columns present the label and FPGA, respectively, while the third and fourth columns present the number of logical cells available (n_{LC}^{max}) and the maximum number of PNH modules that can be implemented with these resources (n_{PNH}^{max}). Hence, a high degree of parallelism can be achieved through our proposed architecture, limited only by the FPGA resources available.

Table 3. Estimated number of PNH modules, n_{PNH}^{max}, for some commercial FPGAs.

Label	FPGA	n_{LC}^{max}	n_{PNH}^{max}
$FPGA_1$	Arria 10 GX 1150	854,400	10
$FPGA_2$	Agilex AGF 027	1,825,600	44
$FPGA_3$	Stratix 10 GX 10M	6,932,160	227

5.2. Time Processing Analysis

The data related to the system's processing time was obtained considering a clock cycle of 13.3ns, which is defined by its critical path. Moreover, as the circuit operates with the same clock, the sampling time is also defined as $t_s = 13.3$ ns.

The system's processing time, for different amounts of n_{PNH}, is presented in Table 4. The first column indicates the number of n_{PNH}. Meanwhile, from the second to fourth columns are shown, respectively, the image processing time, t_I, defined according to Equation (13), the initial system latency, D, according to Equation (11) and the throughput, th, which in this work consists of the number of images processed per second (IPS), determined through Equation (12). According to Equation (13), the processing time of an image depends on its size. Thus, the data presented in Table 4 concern the processing of a 3840×2160 image with 4K resolution.

Table 4. System's processing time.

n_{PNH}	t_I (ms)	D (ms)	th (IPS)
1	110.32	441.28	9
2	55.16	220.64	18
4	27.58	110.32	36
6	18.39	73.56	54
8	13.79	55.16	72
10	11.03	44.12	90

As can be observed in Table 4, the value of n_{PNH} is directly proportional to th and inversely proportional to t_I and D. Thus, the more PNH modules employed in the implementation, the better the system performance. Figure 9 shows the curve obtained by linear

regression that relates the values of n_{PNH} and th. The equation associated with this curve is expressed by

$$th = th(n_{PNH} = 1) \times n_{PNH}. \tag{15}$$

According to (12), and (13), Equation (15) can be rewritten as

$$th = \frac{1}{\left(\frac{M \times N}{1} + \lceil log_2(1) \rceil\right) \times t_s} \times n_{PNH} = \frac{n_{PNH}}{M \times N \times t_s}. \tag{16}$$

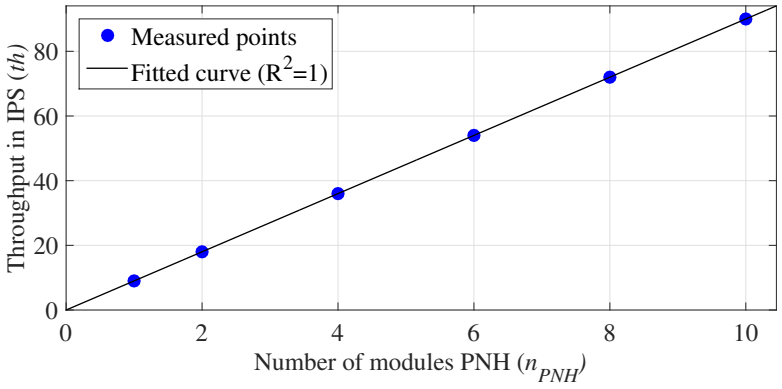

Figure 9. Throughput in IPS, th, per n_{PNH}.

Equation (16), allows one to define t_I values as

$$t_I = \frac{1}{th} = \frac{M \times N \times ts}{n_{PNH}}. \tag{17}$$

When comparing this equation to Equation (13), it is observed that the only difference between them is the absence of the sum of the logarithm. The reason for that is

$$\lceil log_2(n_{PNH}) \rceil \ll \frac{M \times N}{n_{PNH}}. \tag{18}$$

Thus, this component can be disregarded in the calculation of t_I.

Afterward, the proposed architecture's processing time was analyzed for other commercial FPGAs, previously presented in Table 3, adopting for each FPGA an $n_{PNH} = n_{PNH}^{max}$. For this purpose, the values t_I and th were defined according to Equations (16) and (17), respectively, and different image resolutions were considered. The results are shown in Table 5.

Table 5. Values of t_I and th for some commercial FPGAs for different image sizes.

Image Size	$FPGA_1$		$FPGA_2$		$FPGA_3$	
	t_I (ms)	th (IPS)	t_I (ms)	th (IPS)	t_I (ms)	th (IPS)
4 K (3840 × 2160)	11.03	90	2.51	398	0.49	2057
8 K (7680 × 4320)	44.13	22	10.03	99	1.94	514
10 K (10,240×4320)	58.80	16	13.40	74	2.60	385
16 K (15,360×8640)	176.50	5	40.11	24	7.78	128

All FPGA models analyzed achieved a high throughput in processing images with 4 K resolution, allowing real-time processing of 4 K videos in all models. For images with 8 K and 10 K resolutions, real-time processing proved to be more viable in the $FPGA_2$ and $FPGA_3$. Finally, a high throughput was also achieved by the $FPGA_3$ in processing 16 K images, allowing real-time processing of videos with this resolution. Therefore, the

$FPGA_3$ offers better performance due to the high number of PNH modules that can be implemented, i.e., the increased architecture's parallelism.

6. Comparison with State-of-the-Art Works

Comparisons with the state-of-the-art works were performed for the three commercial FPGAs previously mentioned. The architectures were implemented using the maximum number of PNH modules, presented in Table 3. The results were compared only with works that presented data about hardware area occupation and processing time.

6.1. Hardware Occupation Comparison

Table 6 presents the comparison of the hardware area occupation. The first column indicates the reference analyzed. The second to fifth columns show the FPGA, the number of logical cells, multipliers, and the number of bits of memory blocks, respectively, of the reference. Meanwhile, the last four columns present the commercial FPGA to be compared and the ratio between the hardware components used in our proposed implementation, n_{work}, and those used in the literature, n_{ref}. The ratio of the hardware occupation can be expressed as

$$R_{occupation} = \frac{n_{work}}{n_{ref}}, \tag{19}$$

where n_{work} and n_{ref} can be replaced by n_{LC}, n_{MULT}, or n_{BitsM}. This ratio was calculated using the hardware occupancy data presented in Tables 1 and 3.

Table 6. Hardware occupation comparison with other works.

Ref.	$FPGA_{ref}$	n_{LC}	n_{Mult}	n_{BitsM}	$FPGA_{work}$	$R_{occupation}$ n_{LC}	n_{Mult}	n_{BitsM}
[21]	Spartan 6	2975	-	792 K	$FPGA_1$	≈287.19×	-	≈2.73×
					$FPGA_2$	≈613.65×	-	≈2.73×
					$FPGA_3$	≈2330.13×	-	≈2.73×
[19]	Cyclone IV	2440	46	10.94 K	$FPGA_1$	≈350.16×	≈26.56×	≈197.62×
					$FPGA_2$	≈748.20×	≈26.56×	≈197.62×
					$FPGA_3$	≈2841.05×	≈26.56×	≈197.62×
[20]	Virtex 5	1031	5	144 K	$FPGA_1$	≈828.71×	≈244.40×	≈15.01×
					$FPGA_2$	≈1770.71×	≈244.40×	≈15.01×
					$FPGA_3$	≈6723.72×	≈244.40×	≈15.01×

The proposal presented by [21] was deployed on a Spartan-6 LX45 FPGA and occupied an area of 1859 LUTs and 44 RAM blocks. This FPGA uses about 1.6 logic cells (LC) per LUT, having used about 2,975 LC, and each block of RAM has 18 Kbits, so 792 Kbits of memory is used [31]. The number of multipliers used was not available.

In [19], results were presented for two different implementations of the Otsu method, deployed on the Altera Cyclone IV EP4CE115F29C6N FPGA. The comparison with this work was performed considering the best results presented by its authors, i.e., the method using logarithmic functions. The implemented hardware used 2440 LC, 10.94 Kbits of memory and 46 multipliers.

The proposal presented in [20] uses the Virtex-5 xc5vfx70tffg1-136-1 FPGA and occupied a total of 161 slices, 4 BRAMs and 5 multipliers. Each slice of this FPGA has 4 LUTs of 6-input, hence, 6.4 LC per slice was used, and each memory block has 36 Kbits [32]. Thus, about 1030 LC and 144 Kbits of memory were used.

Through Table 6, we found that in all comparisons performed our design presented a more significant hardware area occupation, due to the high degree of parallelism adopted in our proposed implementation, which results in more hardware resources.

6.2. Time Processing Comparison

Table 7 presents a throughput comparison. The analysis was carried out for all the commercial FPGAs previously presented. As can be seen, the first column presents the analyzed reference, while the second to fourth columns show the image resolution (IR), the clock (Clk), and the throughput achieved (th_{ref}), respectively, by the reference. The last three columns indicate, respectively, the FPGAs analyzed, the throughput (th_{work}), and speedup reached with our architecture. The values of th_{work} are calculated according to Equation (16), employing the same clock and image size adopted by the compared reference. The speedup calculation is defined as

$$Speedup = \frac{th_{work}}{th_{ref}}. \tag{20}$$

Table 7. Throughput comparison with other works.

Ref.	RI	Clk (ns)	th_{ref} (IPS)	FPGA	th_{work} (IPS)	Speedup
[21]	280 × 265	10.00	671.59	$FPGA_1$	13,469.83	≈20.06×
				$FPGA_2$	59,088.95	≈87.98×
				$FPGA_3$	298,621.34	≈444.65×
[19]	1280 × 1024	8.72	43.72	$FPGA_1$	874.90	≈20.01×
				$FPGA_2$	3848.92	≈88.03×
				$FPGA_3$	19,833.44	≈453.65×
[20]	640 × 480	39.72	16.39	$FPGA_1$	819.43	≈49.99×
				$FPGA_2$	3602.87	≈219.82×
				$FPGA_3$	18,494.20	≈1128.38×

In [21], the runtime results are presented using a clock of 10 ns and a 280 × 265 input image, with pixels represented by 8 bits. The processing time is 1.489 ms, achieving a throughput of 671.59 IPS.

In [19], time and latency data were presented considering a clock of 8.72 ns and the processing of a 1280 × 1024 image, also with pixel representation for 8 bits. As the processing time of an image was not presented by the authors, it was estimated as the clock value times the number of cycles required to enter an image and obtain its respective output. The processed image's input and output are performed serially, requiring 1280 × 1024 clocks cycles to scan the entire image. The latency indicated for the implementation of the Otsu method using logarithmic optimization is equal to 536. Thus, the calculated time for processing an image is equal to $(1280 \times 1024 \times 2 + 536) \times 8.72 \times 10^{-9} \approx 22.87$ ms per image. Hence, achieving a throughput of 43.72 IPS.

The proposal presented by [20] displays results from throughput for processing a 640 × 480 image, with pixels represented by 8 bits, and adopting a clock of 39.72 ns. The throughput shown indicates the number of megabits processed per second (Mbps), which in turn is 40.28 Mbps. As each pixel has 8 bits, the number of images processed per second can be obtained by $\frac{40.28 \times 10^6}{8 \times 640 \times 480} \approx 16.39$ IPS.

According to the results presented in Table 7, our proposed architecture obtained better throughput values than other works in the literature in all comparisons performed. High speedup values were obtained, varying between 20× and 1128×. The performance achieved by the developed architecture is mainly due to the high degree of parallelism explored in this proposal, with a focus on parallelizing the calculation of the histogram. In contrast, works in the literature present implementations with a low degree of parallelism and are focused on optimizing the calculation of between-class variance.

6.3. Power Consumption Comparison

Table 8 presents a comparison of our design's dynamic power consumption compared to other proposals in the literature. The dynamic power consumption can be expressed as

$$P_d \propto N_g \times F_{clk} \times V_{DD}^2 \tag{21}$$

where P_d is the dynamic power consumption, N_g is the number of hardware components, F_{clk} is the frequency and V_{DD} is the supply voltage. The frequency at which a circuit can operate is proportional to the voltage [12,33]. Thus, the Equation (21) can be rewritten as

$$P_d \propto N_g \times F_{clk}^3. \tag{22}$$

For all comparisons, the value of N_g was calculated as

$$N_g = n_{LC} + n_{Mult}. \tag{23}$$

Based on the Equation (22), the saved dynamic energy, S_d, can be expressed by

$$S_d = \frac{N_g^{ref} \times (F_{clk}^{ref})^3}{N_g^{work} \times (F_{clk}^{work})^3} \tag{24}$$

where N_g^{ref} and F_{clk}^{ref} are the number of hardware components and clock of literature works, respectively, and N_g^{work} and F_{clk}^{work} are the number of components and the clock adopted in this work to obtain the same throughput of the reference to which it is being compared.

Table 8. Dynamic power comparison with other works.

Ref.	N_g^{ref}	F_{clk}^{ref} (MHz)	N_g^{work}	FPGA	F_{clk}^{work} (MHz)	S_d
[21]	2975	100.00	593,168	$FPGA_1$	4.98	$\approx 2.82 \cdot 10^1\times$
				$FPGA_2$	1.13	$\approx 1.13 \cdot 10^3\times$
				$FPGA_3$	0.22	$\approx 4.03 \cdot 10^4\times$
[19]	2486	114.62	593,168	$FPGA_1$	5.73	$\approx 2.33 \cdot 10^1\times$
				$FPGA_2$	1.30	$\approx 9.33 \cdot 10^2\times$
				$FPGA_3$	0.25	$\approx 3.46 \cdot 10^4\times$
[20]	1036	25.175	593,168	$FPGA_1$	0.50	$\approx 1.55 \cdot 10^2\times$
				$FPGA_2$	0.11	$\approx 6.80 \cdot 10^3\times$
				$FPGA_3$	0.02	$\approx 9.98 \cdot 10^5\times$

Table 8 shows that our implementation presents a significant reduction in energy consumption. The results presented indicate an energy saving, regarding works in the literature, between $2.33 \cdot 10^1$ and $9.98 \cdot 10^5\times$. This reduction is obtained through the high degree of parallelism of the technique, which allows one to obtain high throughput with the circuit operating at a low clock frequency. Therefore, although the circuit uses many hardware components, the energy consumed is reduced due to the low-frequency operation.

7. Conclusions

This work presented a parallel implementation proposal of the Otsu method in FPGA. The hardware architecture was developed using RTL design and fixed-point representation. All the implementation details were presented, and the synthesis results were related to the hardware area occupation and processing time. The results showed that the proposed architecture achieved high throughput, enabling real-time processing of high-resolution videos. Comparisons with state-of-the-art works were also performed regarding the hardware area occupancy, throughput and dynamic power consumption. Our proposed architecture out-

performed the compared ones in terms of throughput and power consumption, achieving a speedup from 20× until 10^3× and reducing the power from 23× until 10^5×. However, the amount of hardware resources required is higher than in other architectures due to the high degree of parallelism adopted in our method. Finally, as future work, a study will be carried out to analyze the impact on the result's accuracy by reducing the number of bits used. Besides, tests will be conducted with the processing of videos in real-time.

Author Contributions: All the authors have contributed in various degrees to ensure the quality of this work. (e.g., W.K.P.B. and M.A.C.F. conceived the idea and experiments; W.K.P.B. and M.A.C.F. designed and performed the experiments; W.K.P.B., L.A.D. and M.A.C.F. analyzed the data; W.K.P.B., L.A.D. and M.A.C.F. wrote the paper. M.A.C.F. coordinated the project.). All authors have read and agreed to the published version of the manuscript.

Funding: This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES)—Finance Code 001.

Acknowledgments: The authors wish to acknowledge the financial support of the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) for their financial support.

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

References

1. Barros, W.K.; Morais, D.S.; Lopes, F.F.; Torquato, M.F.; Barbosa, R.d.M.; Fernandes, M.A. Proposal of the CAD system for melanoma detection using reconfigurable computing. *Sensors* **2020**, *20*, 3168. [CrossRef] [PubMed]
2. Menaka, R.; Janarthanan, R.; Deeba, K. FPGA implementation of low power and high speed image edge detection algorithm. *Microprocess. Microsystems* **2020**, *75*, 103053. [CrossRef]
3. Younis, D.; Younis, B.M. Low Cost Histogram Implementation for Image Processing using FPGA. In *IOP Conference Series: Materials Science and Engineering*; IOP Publishing: Bristol, UK, 2020; Volume 745, p. 012044.
4. Sreenivasulu, M.; Meenpal, T. Efficient hardware implementation of 2d convolution on FPGA for image processing application. In Proceedings of the 2019 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT), Coimbatore, India, 20–22 Febuary 2019; pp. 1–5.
5. Altuncu, M.A.; Kösten, M.M.; Çavuşlu, M.A.; Şahın, S. FPGA-based implementation of basic image processing applications as low-cost IP core. In Proceedings of the 2018 26th Signal Processing and Communications Applications Conference (SIU), Izmir, Turkey, 2–5 May 2018; pp. 1–4.
6. Bailey, D. *Design for Embedded Image Processing on FPGAs*; Wiley-IEEE, Wiley: Atlanta, GA, USA, 2011.
7. Gonzalez, R.C.; Woods, R.E. *Digital Image Processing*, 4th ed.; Pearson: New York, NY, USA, 2018.
8. Otsu, N. A Threshold Selection Method from Gray-Level Histograms. *IEEE Trans. Syst. Man Cybern.* **1979**, *9*, 62–66. [CrossRef]
9. Gokhale, M.; Graham, P. *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*; Springer: Dordrecht, The Netherland, 2005.
10. Vahid, F. *Digital Design with RTL Design, Verilog and VHDL*, 2nd ed.; John Wiley & Sons: Hoboken, NJ, USA, 2010.
11. Dias, L.A.; Damasceno, A.M.; Gaura, E.; Fernandes, M.A. A full-parallel implementation of Self-Organizing Maps on hardware. *Neural Netw.* **2021**. [CrossRef] [PubMed]
12. Silva, S.N.; Lopes, F.F.; Valderrama, C.; Fernandes, M.A. Proposal of Takagi–Sugeno Fuzzy-PI Controller Hardware. *Sensors* **2020**, *20*, 1996. [CrossRef] [PubMed]
13. Torquato, M.F.; Fernandes, M.A. High-performance parallel implementation of genetic algorithm on fpga. *Circuits Syst. Signal Process.* **2019**, *38*, 4014–4039. [CrossRef]
14. Da Costa, A.L.; Silva, C.A.; Torquato, M.F.; Fernandes, M.A. Parallel implementation of particle swarm optimization on fpga. *IEEE Trans. Circuits Syst. Ii Express Briefs* **2019**, *66*, 1875–1879. [CrossRef]
15. Coutinho, M.G.; Torquato, M.F.; Fernandes, M.A. Deep neural network hardware implementation based on stacked sparse autoencoder. *IEEE Access* **2019**, *7*, 40674–40694. [CrossRef]
16. Jianlai, W.; Chunling, Y.; Min, Z.; Changhui, W. Implementation of Otsu's thresholding process based on FPGA. In Proceedings of the 2009 4th IEEE Conference on Industrial Electronics and Applications, Xi'an, China, 25–27 May 2009; pp. 479–483. [CrossRef]
17. Tian, H.; Lam, S.K.; Srikanthan, T. Implementing Otsu's thresholding process using area-time efficient logarithmic approximation unit. In Proceedings of the 2003 International Symposium on Circuits and Systems, Bangkok, Thailand, 25–28 May 2003; Volume 4, p. IV. [CrossRef]
18. Pandey, J.G.; Karmakar, A.; Shekhar, C.; Gurunarayanan, S. A Novel Architecture for FPGA Implementation of Otsu's Global Automatic Image Thresholding Algorithm. In Proceedings of the 2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems, Mumbai, India, 5–9 January 2014; pp. 300–305. [CrossRef]

19. Torres-Monsalve, A.F.; Velasco-Medina, J. Hardware implementation of ISODATA and Otsu thresholding algorithms. In Proceedings of the 2016 XXI Symposium on Signal Processing, Images and Artificial Vision (STSIVA), Bucaramanga, Colombia, 31 August–2 September 2016; pp. 1–5. [CrossRef]
20. Pandey, J.G.; Karmakar, A. Unsupervised image thresholding: hardware architecture and its usage for FPGA-SoC platform. *Int. J. Electron.* **2019**, *106*, 455–476. [CrossRef]
21. Das, R.K.; De, A.; Pal, C.; Chakrabarti, A. DSP hardware design for fingerprint binarization and thinning on FPGA. In Proceedings of The 2014 International Conference on Control, Instrumentation, Energy and Communication (CIEC), Calcutta, India, 31 January–2 February 2014, pp. 544–549. [CrossRef]
22. Wang, W.; Huang, X. An FPGA co-processor for adaptive lane departure warning system. In Proceedings of the 2013 IEEE International Symposium on Circuits and Systems (ISCAS2013), Beijing, China, 19–23 May 2013; pp. 1380–1383. [CrossRef]
23. Zhao, J.; Bingqian Xie.; Huang, X. Real-time lane departure and front collision warning system on an FPGA. In Proceedings of the 2014 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 9–11 September 2014; pp. 1–5. [CrossRef]
24. Alhamwi, A.; Vandeportaele, B.; Piat, J. Real Time Vision System for Obstacle Detection and Localization on FPGA. In *Computer Vision Systems*; Nalpantidis, L., Krüger, V., Eklundh, J.O., Gasteratos, A., Eds.; Springer International Publishing: Cham, Switzerland, 2015; pp. 80–90.
25. Ren, X.; Wang, Y. Design of a FPGA hardware architecture to detect real-time moving objects using the background subtraction algorithm. In Proceedings of the 2016 5th International Conference on Computer Science and Network Technology (ICCSNT), Changchun, China, 10–11 December 2016; pp. 428–433. [CrossRef]
26. Tulasigeri, C.; Irulappan, M. An advanced thresholding algorithm for diagnosis of glaucoma in fundus images. In Proceedings of the 2016 IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT), Bangalore, India, 20–21 May 2016; pp. 1676–1680. [CrossRef]
27. Kim, H.S. FPGA-based of thermogram enhancement algorithm for non-destructive thermal characterization. *Int. J. Eng.* **2018**, *31*, 1675–1681.
28. Ladner, R.E.; Fischer, M.J. Parallel Prefix Computation. *J. ACM* **1980**, *27*, 831–838. [CrossRef]
29. Intel. Intel® Stratix® 10 GX/SX Device Overview. Available online: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/s10-overview.pdf (accessed on 2 January 2021).
30. Intel. Intel® Agilex™ FPGAs and SoCs Advanced Information Brief (Device Overview). Available online: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/agilex/ag-overview.pdf(accessed on 2 January 2021).
31. Xilinx. Spartan-6 FPGA Configurable Logic Block. Available online: https://www.xilinx.com/support/documentation/user_guides/ug384.pdf (accessed on 13 June 2021).
32. Xilinx. Virtex-5 Special Edition. Available online: https://www.xilinx.com/publications/archives/xcell/Xcell59.pdf (accessed on 13 June 2021).
33. McCool, M.; Robison, A.D.; Reinders, J. Chapter 2—Background. In *Structured Parallel Programming*; Morgan Kaufmann: Boston, MA, USA, 2012; pp. 39–75.