Journal of Real-Time Image Processing (2024) 21:118
https://doi.org/10.1007/s11554-024-01498-6

RESEARCH

A novel pipelined architecture of entropy filter

Dat Ngo[1] · Bongsoon Kang[2]

Received: 5 March 2024 / Accepted: 13 June 2024 / Published online: 23 June 2024
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2024

Abstract

In computer vision, entropy is a measure adopted to characterize the texture information of a grayscale image, and an entropy filter is a fundamental operation used to calculate local entropy. However, this filter is computationally intensive and demands an efficient means of implementation. Additionally, with the foreseeable end of Moore's law, there is a growing trend towards hardware offloading to increase computing power. In line with this trend, we propose a novel method for the calculation of local entropy and introduce a corresponding pipelined architecture. Under the proposed method, a sliding window of pixels undergoes three steps: sorting, adjacent difference calculation, and pipelined entropy calculation. Compared with a conventional design, implementation results on a Zynq UltraScale+ XCZU7EV-2FFVC1156 MPSoC device demonstrate that our pipelined architecture can reach a maximum throughput of handling 764.526 megapixels per second while achieving 2.4× and 2.9× reductions in resource utilization and 1.1× reduction in power consumption.

Keywords Computer vision · Entropy filter · Hardware offloading · Pipelining · FPGA

1 Introduction

Chip density has begun to lose its fast-growing pace and has no longer doubled every two years. Accordingly, the growth in processing power of general purpose central processing units (CPUs) cannot cope with the continuously increasing demand for high-performance computing. Also, wire delay lengthens as more transistors are integrated into a processing core, consequently prolonging memory latency and further reducing CPU performance [3]. Several strategies have been proposed to address that issue, such as adding more processor cores or developing heterogeneous computing architectures (HeCAs). Among these two examples, the latter appears to be more promising because it encompasses an innovative idea of utilizing different kinds of processing elements.

Figure 1 illustrates a conventional homogeneous computing architecture (HoCA) and a simple HeCA. It can be observed that the HeCA comprises CPUs, graphics processing units (GPUs), and field-programmable gate arrays (FPGAs), while the HoCA only consists of CPUs. Therefore, the HeCA can offload complex functions, such as deep neural network inference and image filtering, onto GPUs and FPGAs, respectively, saving its CPUs for critical tasks. The smart network interface card (SmartNIC) [15] is an excellent example of how HeCAs can be utilized in practice. This device is a successor to the traditional NIC and is equipped with FPGA accelerators to support hardware offloading. As a result, it can free up the system's resources allocated for complex tasks such as match-action, tunneling, and load balancing, thereby assisting the CPU in handling heavy network traffic up to 40 Gbps per NIC port. Additionally, SmartNICs often come with high-level programming abstractions for their FPGA engines to provide users with great flexibility. Instead of offloading only a fixed set of functions, users can now customize SmartNIC's engines to handle their own functions, such as pattern matching operations.

To accelerate the pace of development in HeCAs and hardware offloading, the emerging demand for software-to-hardware conversion has attracted growing interest. The conversion has become more challenging for 2-D spatial image filters, which require repeating a particular operation

✉ Bongsoon Kang
bongsoon@dau.ac.kr

Dat Ngo
datngo@ut.ac.kr

[1] Department of Computer Engineering, Korea National University of Transportation, Chungju 27469, South Korea

[2] Department of Electronic Engineering, Dong-A University, Busan 49315, South Korea

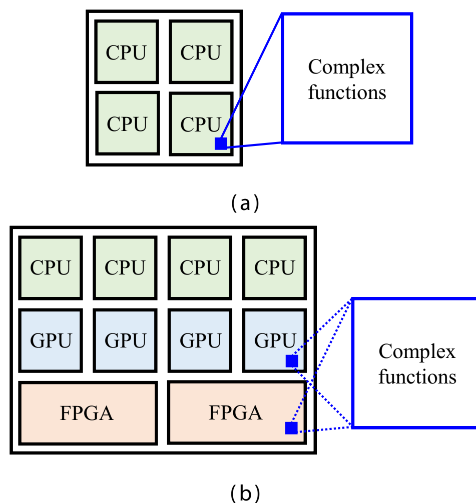

Fig. 1 Illustration of two common computing architectures. **a** Homogeneous computing architecture (HoCA). **b** Heterogeneous computing architecture (HeCA). In HoCAs, the system's processing cores are responsible for handling every task. Conversely, HeCAs enable offloading potentially complex functions onto specialized accelerators, thereby conserving the computing power of processing cores

throughout the entire image in a raster-scan order. For example, the median filter, a robust means to suppress impulse noise, is computationally intensive and has resulted in a series of recent studies [4, 14, 17, 20] on its hardware implementation.

In this sense, it is also necessary to facilitate the hardware offloading of the entropy filter, a fundamental operation in computer vision that imposes a heavy computational burden on the system's CPU. Hence, we propose a new method for local entropy calculation and present a corresponding hardware accelerator. In the remainder of this paper, we first introduce the related work on hardware offloading of entropy calculation. After that, we present the entropy concept widely adopted in computer vision and discuss a few of its applications. We then describe the proposed method and the hardware architecture. Finally, we provide and analyze implementation results, concluding the paper with our findings.

2 Related work

The adoption of hardware offloading techniques is prevalent in computer vision and network traffic monitoring, two areas demanding high-performance computing capabilities. Modern imaging systems capture high-resolution images (up to 4K or 8K), creating a challenge to process large volumes of data in a limited time frame. Similarly, network systems handle millions of packets per second, requiring real-time processing to avoid bottlenecks. Moreover, real-time applications in computer vision, such as autonomous driving and robotic-assisted surgery, necessitate immediate feedback due to their life-critical nature. Likewise, many network applications, including VoIP, video streaming, and online gaming, are latency-sensitive, as delays can significantly degrade user experience. Consequently, high-performance computing platforms like FPGAs are well-suited for these research areas.

In network traffic monitoring field, entropy is commonly used to analyze the traffic. By computing the entropy of a packet sequence over a specific time interval, traffic anomalies such as DDoS attacks can be detected. Hardware accelerators are typically employed for entropy calculation due to the high throughput requirements of high-speed links. For example, FlexSkethMon [21], a framework for traffic monitoring based on estimated network flow entropy, was implemented on a NetFPGA-SUME board, achieving a processing speed of 96 Gbps. Another entropy estimation algorithm, implemented on a NetFPGA-10 G board, processed network traffic at 30 Gbps [13]. Recently, a hardware accelerator for entropy estimation using the top-k most frequent elements demonstrated a processing speed of 181 Gbps on a Zynq UltraScale+ ZCU102 FPGA board [19]. These studies underscore the extensive use of FPGAs for the computationally intensive task of entropy estimation.

Conversely, in computer vision, there is less research on hardware accelerators for entropy calculation, partly due to the complexity of spatial relationships between image pixels. In [12], an FPGA accelerator for a small binarized neural network was presented, utilizing frame entropy to devise an adaptive filter. This filter controls the number of frames the network processes in inference mode to enhance the processing rate. The entire system was synthesized using a high-level synthesis (HLS) tool, which mapped the entropy calculation and filtering operations to the processing system (PS) rather than the high-performance programmable logic (PL). While HLS tools can synthesize FPGA accelerators from high-level programming languages (such as C/C++ and Python), the resulting accelerators are not always optimal. This trend was also observed in many algorithms for entropy calculation presented in [5].

In this paper, we leverage pipeline parallelism to design an efficient architecture for an entropy filter and implement it using Verilog HDL [1] (IEEE Standard 1364-2005). This approach fully exploits the FPGA's high-performance computing hardware available on the PL. Detailed implementation is presented in Sect. 5.

3 Preliminaries

3.1 Overview of entropy filter

Entropy is a concept adopted in diverse fields, and it is generally associated with a state of randomness or

uncertainty. In computer vision, entropy is often considered in the context of information theory and is defined as follows:

$$h(X) = -\sum_{i=1}^{n} p(x_i)\log p(x_i), \quad (1)$$

where h denotes the entropy value, X a discrete random variable accepting values in the set $\{x_1, x_2, \dots, x_n\}$, and p the probability density function (PDF) of X. Depending on whether $\log(\cdot)$ refers to a binary or natural logarithm, h can be measured in the unit of bits or nats (natural unit of information) [9]. Hereinafter, we only focus on the former case of the binary logarithm. Also, in the context of image filtering, (1) is restricted to a sliding window of pixels; therein lies the name local entropy.

Moreover, entropy is usually interpreted as a measurement of information. Thus, the larger the entropy, the more informative the image content. Figure 2 demonstrates a grayscale image and its corresponding entropy profile, calculated using a 25×25 sliding window. It can be observed that image patches with rich texture information are associated with high entropy values. On the contrary, those with fewer details possess low entropy values. As a result, entropy can be used to characterize the texture information of a grayscale image. This amazing feature renders entropy highly relevant in many computer vision applications, as discussed in the following.

3.2 Applications in computer vision

Image sharpening (or, equivalently, sharpness enhancement) is a fundamental low-level operation in digital image processing, in which an image and its scaled Laplacian are fused to accentuate the sharpness. As digital images are represented by a finite discrete set of intensities, the fusion is prone to out-of-range and edge ringing problems. Accordingly, image entropy was adopted as an attention map to control the fusion, lest the original image was over-enhanced, as demonstrated in an automatic sharpening algorithm [7] (owned by Apple Inc.).

Entropy also finds applications in the fast-expanding field of pattern recognition. For example, an image separation algorithm (owned by Qualcomm) [2] utilizes entropy to identify and remove unwanted regions (such as those associated with trees or sky) from the image, facilitating object recognition algorithms. In [6], entropy serves as one of twelve image features to form a feature space, in which Mahalanobis distance is adopted to predict the image's perceptual visibility. Similarly, in [10], entropy is one of four image features fed to a Bayes classifier for detecting salient objects.

Given the diverse applications of entropy in computer vision, it is surprising that the literature has been silent on a computationally efficient implementation of the entropy filter. We, therefore, devise a novel method for calculating local entropy and present a corresponding pipelined architecture in the upcoming sections.

4 Proposed method

4.1 Overview

Given a grayscale image $I \in \mathbb{R}^{M\times N}$ of size $M \times N$, an entropy filter with a square window of size $S \times S$ traverses the image in a raster-scan order to create a corresponding entropy profile $H \in \mathbb{R}^{M\times N}$. Under the filtering mechanism, the filter calculates the entropy value of pixel neighborhoods covered underneath the window at every pixel location. It then assigns that value to the corresponding pixel location in the profile. In the case of boundary padding adopted, the profile is the same size as the input; otherwise, the height and width are reduced by $\lfloor S/2 \rfloor$. An illustrative description of this mechanism can be found in Fig. 3.

Let $\mathbf{u} = \left[u_0, u_1, \dots, u_{S^2-1}\right]$ be a vector containing all pixels covered by the window. Each pixel can take on an intensity value in the discrete set $\{x_0, x_1, \dots, x_{n-1}\}$, with n depending on the number of bits used to represent the image. For

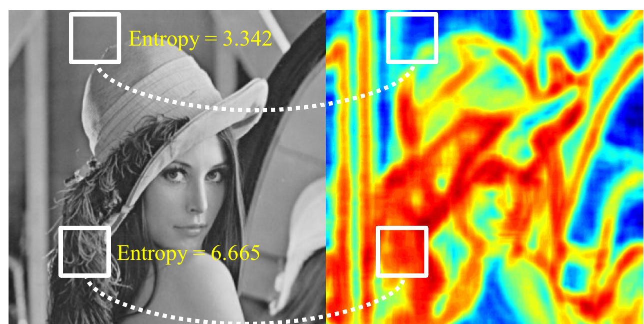

Fig. 2 An example image and its corresponding entropy profile (calculated using a 25×25 sliding window). An upper patch with low texture information has an entropy value of 3.342, whereas a lower patch with high texture information has an entropy value of 6.665

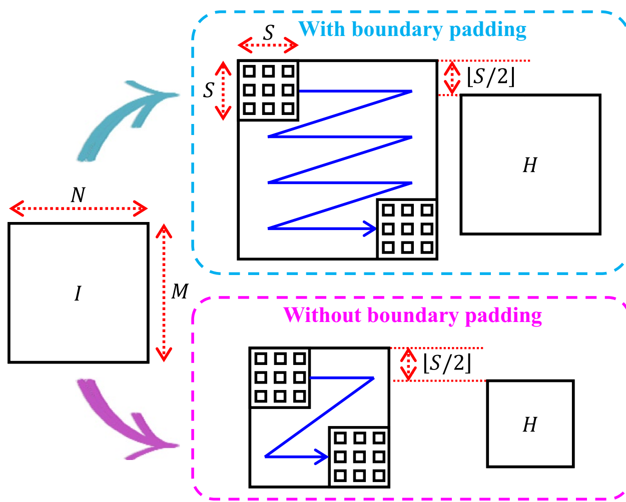

Fig. 3 Illustration of the filtering mechanism

example, $n = 256$ and $x_0 = 0, x_1 = 1, \ldots, x_{n-1} = 255$ in a common 8-bit representation. The problem herein is to design an efficient means to calculate the local entropy while ensuring the compactness and efficacy of the corresponding hardware accelerator. In the following subsections, we first introduce a conventional method and discuss its limitations. Thereafter, we present our solution to remold the conventional way of calculating local entropy.

4.2 Conventional method

Given $\mathbf{u}$, its entropy value h can be calculated by (i) constructing a histogram to obtain the PDF and then (ii) applying (1), as shown in Algorithm 1. Let $\mathbf{p} = [p_0, p_1, \ldots, p_{n-1}]$ denote the histogram of $\mathbf{u}$. We first initialize $\mathbf{p}$ with zeros and then iterate through all pixels in $\mathbf{u}$. At each iteration, we utilize the pixel intensity u_i as an index ($j \leftarrow u_i$) to increment the corresponding bin p_j. When the for loop terminates, we obtain the histogram $\mathbf{p}$, which represents the number of occurrences of each pixel intensity. Therefore, in the second loop, we divide p_j by S^2 (the total number of pixels in $\mathbf{u}$) to get the density. To calculate the entropy value h, we also initialize it with zero and iterate through all bins in $\mathbf{p}$.

The above-mentioned algorithm is deceptively simple, but arriving at an efficient hardware implementation is not straightforward. Firstly, constructing the histogram for each vector $\mathbf{u}$ returned by the filter as it traverses the image is highly inefficient. Secondly, the histogram size increases exponentially with the pixel's word length (the number of bits utilized to represent pixel intensities), leading to a significant consumption of hardware resources. For example, an 8-bit representation requires 256 registers (or memory locations) to hold the histogram bins, and this number doubles on the increment of the word length. Consequently, in this case, the cost for filtering an image equals $\mathcal{O}(M \cdot N \cdot S^2)$ (histogram construction) plus $\mathcal{O}(M \cdot N \cdot n)$ (entropy calculation).

Regarding these two issues, we can resolve the former by exploiting the sliding mechanism. More specifically, we can update the histogram by simply decrementing/incrementing bins corresponding to pixel intensities that leave/enter the kernel. As a result, the cost of histogram construction can be reduced to $\mathcal{O}(M \cdot N \cdot 2S)$. The sole hardware accelerator [18] we found in the literature was designed using this approach.

However, resolving the latter issue without remolding the entropy calculation to eliminate the histogram construction remains challenging. We will present our efforts in addressing this in the following subsection.

Algorithm 1 Conventional entropy calculation

DATA
  Input $\mathbf{u} = [u_0, u_1, ..., u_{S^2-1}]$
  Output h

AUXILIARY FUNCTION
  init($\mathbf{p}$) Initialize $\mathbf{p}$ with zeros

CALCULATION
(i) Constructing a histogram
  declare $\mathbf{p} = [p_0, p_1, ..., p_{n-1}]$
  init($\mathbf{p}$)
  for $i \leftarrow 0$ to $S^2 - 1$
    $j \leftarrow u_i$
    $p_j \leftarrow p_j + 1$
  end for
(ii) Applying (1)
  initialize $h \leftarrow 0$
  for $i \leftarrow 0$ to $n - 1$
    $h \leftarrow h - (p_j/S^2)\log(p_j/S^2)$
  end for

4.3 Our proposed solution

We observe that it is possible to obtain the number of occurrences of pixel intensities without constructing a histogram. By sorting the pixels and then taking adjacent differences, we can determine how many pixel intensities there are and how many times each of them occurs. Let us consider a 3×3 window, and assume that $\mathbf{u} = [0, 51, 62, 33, 100, 0, 51, 100, 51]$ is the vector of pixels covered underneath. The result obtained after sorting and

taking adjacent differences is $\mathbf{d} = [0, 33, 18, 0, 0, 11, 38, 0]$. We can then easily determine the number of pixel intensities by counting the total transitions from "zero → non-zero" and "non-zero → non-zero." In our example, there are four of those transitions (0 → 33, 33 → 18, 0 → 11, and 11 → 38), signifying that there are five different pixel intensities (0, 32, 51, 62, and 100).

Also, the first zero in $\mathbf{d}$ means that a particular pixel intensity occurs twice. The following non-zero (33) signifies the occurrence of a different intensity which appears once because of the next non-zero (18). This non-zero is followed by two zeros, signifying a three-time occurrence of another intensity. Similarly, the remaining patterns of $\mathbf{d}$ (11 → 38 → 0) signify a sole occurrence followed by a two-time occurrence of two different intensities, respectively. Hence, we obtain the histogram $\mathbf{p} = [2, 1, 3, 1, 2]$ without explicitly constructing it. With this information, we can easily calculate the entropy value.

In Algorithm 2, we present a new method for entropy calculation that facilitates the implementation of a pipelined hardware accelerator, which will be discussed later in Sect. 5. For ease of description, we adopt two auxiliary functions: Sort($\mathbf{u}$), which sorts the elements of $\mathbf{u}$ in ascending order, and the Kronecker delta $\delta(a, b)$, which returns one if $a = b$ and zero otherwise.

In the first step, we declare a vector $\mathbf{v}$ to hold the sorted elements of $\mathbf{u}$. After that, we apply the Kronecker delta function to every adjacent pair in $\mathbf{v}$ and store the $(S^2 - 1)$-element result in $\mathbf{d}$. It is noteworthy that each element of this vector is either zero or one and thus only requires a single bit for representation. Consequently, the vector $\mathbf{d}$ can be expressed compactly using $(S^2 - 1)$ bits, which is highly beneficial for the hardware implementation phase.

In the last step of pipelined entropy calculation, we first prepare a look-up table (LUT) holding the value of each term of the summation in (1). Theoretically, an S^2-entry LUT is sufficient for an $S \times S$ entropy filter. However, we utilize an $(S^2 + 1)$-entry LUT with a zero-indexed entry equal to zero to retain the simplicity of our method. As shown in Algorithm 2, we use a variable called addr to retrieve values from the LUT. Since the logarithm of zero is undefined, special handling is required when addr becomes zero. To address this, instead of using multiplexers to resolve the case where addr = 0 when calculating the entropy, we add an entry $l_{\text{addr}} = 0$ for addr = 0 in the LUT. This approach simplifies the design and saves hardware resources (Table 1).

Algorithm 2 Our proposed entropy calculation

```
DATA
    Input    u = [u_0, u_1, ..., u_{S^2-1}]
    Output   h

AUXILIARY FUNCTIONS
    Sort(u)   Sorting function (in ascending order)
    δ(a, b)   Kronecker delta function
    δ(a, b) = 1 if a = b, and δ(a, b) = 0 if a ≠ b

CALCULATION
(i) Sorting
    declare v = [v_0, v_1, ..., v_{S^2-1}]
    v ← Sort(u)
(ii) Calculating adjacent differences
    declare d = [d_0, d_1, ..., d_{S^2-2}]
    for i ← 0 to S^2 − 2
        d_i ← δ(v_i, v_{i+1})
    end for
(iii) Pipelined entropy calculation
  (iii.a) Initialize look-up table
    declare LUT = [l_0, l_1, ..., l_{S^2}]
    for i ← 0 to S^2
        l_i ← 0 if i = 0 or i = S^2
        l_i ← −(i/S^2) log(i/S^2) otherwise
    end for
  (iii.b) Calculate entropy
    for i ← 0 to S^2 − 1
        if i = 0 .......................... first iteration
            h ← l_1 · d_i
            addr ← 2(1 − d_i)
            cnt ← 2 − d_i
        else if i = S^2 − 1 ................ last iteration
            h ← h + l_addr − (cnt − S^2) · l_1
        else .................................... others
            h ← h + (l_addr + l_1) · d_i
            if addr = 0
                addr ← 2(1 − d_i)
                cnt ← cnt + (2 − d_i)
            else
                addr ← (1 − d_i)(addr + 1)
                cnt ← cnt + 1
            end if
        end if
    end for
```

After initializing the LUT, we iterate through $\mathbf{d}$ to calculate the entropy. We adopt two variables: addr to access the LUT's entries and cnt to track the number of pixels that have been processed. In the first iteration ($i = 0$), d_0 is used to initialize three variables: h, addr, and cnt. The case of d_0 being zero means that a particular pixel intensity occurs twice; thus, addr ← 2 and cnt ← 2, whereas h remains zero. Otherwise, that pixel intensity only occurs once, and addr ← 1, cnt ← 1, and $h \leftarrow l_1$ correspondingly. In subsequent iterations ($1 \leq i \leq S^2 - 2$), if d_i is zero, h remains unchanged, and addr is updated to track the number of occurrences of the previous pixel intensity. If d_i is one, it means a new pixel intensity occurs (hence, the term l_1), and addr currently points to an LUT entry corresponding to the density value of

Table 1 Step-by-step procedure for calculating entropy using the proposed method

	Normal case			Extreme case		
Input	[0, 51, 62, 33, 100, 0, 51, 100, 51]			[10, 10, 10, 10, 10, 10, 10, 10, 10]		
Sorting	[0, 0, 33, 51, 51, 51, 62, 100, 100]			[10, 10, 10, 10, 10, 10, 10, 10, 10]		
Adjacent differences	[0, 33, 18, 0, 0, 11, 38, 0]			[0, 0, 0, 0, 0, 0, 0, 0]		
Kronecker delta	[0, 1, 1, 0, 0, 1, 1, 0]			[0, 0, 0, 0, 0, 0, 0, 0]		
Entropy calculation	h	addr	cnt	h	addr	cnt
0 (first iteration)	0	2	2	0	2	2
1	0.8344	0	3	0	3	3
2	1.1866	0	4	0	4	4
3	1.1866	2	6	0	5	5
4	1.1866	3	7	0	6	6
5	2.0672	0	8	0	7	7
6	2.4194	0	9	0	8	8
7	2.4194	2	11	0	9	9
8 (last iteration)	2.1972			0		

LUT	
l_0	0
l_1	0.3522
l_2	0.4822
l_3	0.5283
l_4	0.5200
l_5	0.4711
l_6	0.3900
l_7	0.2820
l_8	0.1510
l_9	0

the last pixel intensity (l_{addr}). Therefore, the term ($l_{\text{addr}} + l_1$) is added to h to update the entropy value. During that course of calculation, cnt is also updated correspondingly. Finally, in the last iteration ($i = S^2 - 1$), cnt is used to compensate for the entropy value.

To facilitate the understanding of our proposed method, we present a step-by-step procedure for calculating entropy using Algorithm 2 in Table 1. The normal case corresponds to the example discussed at the beginning of this subsection, while the extreme case addresses the scenario where all input values are equal.

As our proposed method is hardware-design-oriented, it is not easy to compare with the conventional method from the software perspective. For example, if we utilize Batcher's parallel sorting algorithm [11] as the Sort(·) function, the total cost will be $\mathcal{O}\Big(M \cdot N \cdot \big(\log S^2\big)^2\Big)$ plus $\mathcal{O}(M \cdot N \cdot S^2)$. Compared with the conventional method, ours does not depend on n; however, it is still challenging to determine which one possesses a lower cost. Therefore, we present a pipelined hardware architecture in the next section and compare it with a conventional design [18] whose details are available in Appendix 8.

5 Pipelined architecture

Figure 4 illustrates the overall block diagram of the proposed hardware accelerator, in which the first two blocks (line memories and register banks) are common in 2-D spatial image filters. They help handle the input stream to obtain pixels covered underneath the window. Hence, an $S \times S$ window requires $(S - 1)$ line memories (each can store an image line, i.e., N pixels) and $(S - 1)$ register banks (each consists of S registers).

The following three blocks (sorting network, adjacent difference calculation, and pipelined entropy calculation) realize our proposed method. Concerning the first, we adopt the optimized merging-sorting network [16] for a fast and compact implementation. After that, we can easily compare each consecutive pair of $\mathbf{v}$ utilizing an exclusive OR gate followed by a reduction OR gate. The one-bit result is identical to that of the Kronecker delta function. Given the vector $\mathbf{d}$, we can calculate the entropy with the pipelined architecture in Fig. 5.

This architecture implements the for loop in section *(iii.b)* of Algorithm 2, where results after each iteration are stored in pipelined registers. For example, in the first iteration, we utilize the first bit d_0 of $\mathbf{d}$ to initialize three variables addr, cnt, and h. Accordingly, in the first stage of the pipelined architecture, d_0 controls three multiplexers to initialize the three corresponding registers addr_0, cnt_0, and ht_0. Meanwhile, we use another register ($\mathbf{d}_1^{S^2-2}$) to store the remaining bits of $\mathbf{d}$, i.e., $\mathbf{d}_1^{S^2-2} = \big[d_1, d_2, \dots, d_{S^2-2}\big]$. In the following stages, which correspond to iterations from the second to the $(S^2 - 1)$-th, we extract every bit of $\mathbf{d}$ to update the intermediate results stored in addr_i, cnt_i, and ht_i registers, with $1 \le i \le S^2 - 2$. Finally, we carry out the calculation in the last iteration to obtain the entropy value in the h register.

As mentioned earlier in Sect. 2, we do not utilize the HLS tool due to its suboptimal performance. Instead, we design the proposed hardware accelerator using Verilog HDL [1] (IEEE Standard 1364-2005) and use Xilinx Vivado v2019.1 to obtain the implementation results, which are presented in the next section. We also perform a quantitative comparison with a conventional design in [18]. The target device for both designs is an XCZU7EV-2FFVC1156 MPSoC [22] on a Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit,

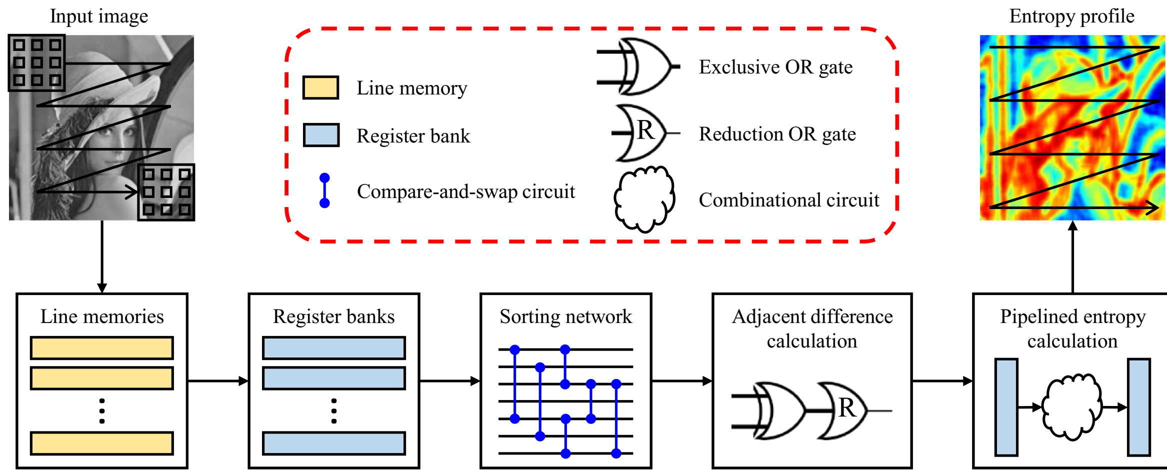

Fig. 4 Block diagram of the proposed hardware architecture

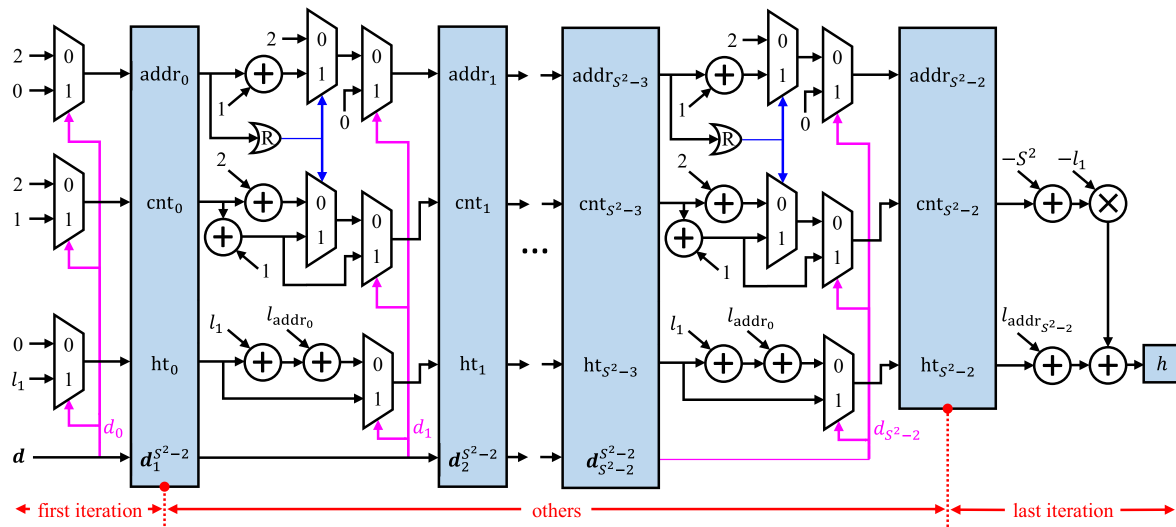

Fig. 5 Block diagram of the pipelined entropy calculation

equipped with 460,800 registers, 230,400 LUTs, and 312 block RAMs.

6 Implementation results

Table 2 summarizes the implementation results of two hardware accelerators (the conventional in [18] and ours) of a 5×5 entropy filter. Concerning memory usage, the two designs consume the same amount of block RAMs because they both need the first two blocks (line memories and register banks) to obtain $S^2 = 25$ image pixels. However, it can be observed that our design is significantly smaller than the conventional. It only requires 7849 registers and 5748 LUTs, which occupy about 1.70% and 2.49% of the related hardware resources. Compared with those of the conventional design, we achieve approximately 2.4× and 2.9× reductions in register and LUT consumption. This reduction in resource utilization consequently leads to a corresponding drop (1.1×) in power consumption.

Finally, considering the maximum frequency, our design is slightly faster than the conventional. It is noteworthy that the capability of handling 764.526 megapixels per second is far beyond the requirement of computer vision edge devices.

Table 2 Implementation results of the entropy filter's hardware accelerators

Vivado 2019.1						
Device		XCZU7EV-2FFVC1156				
Design		Conventional [18]		Ours		Gain
Slice logic utilization	Available	Used	Utilization	Used	Utilization	
Slice registers (#)	460,800	18,869	4.09%	7849	1.70%	× 2.4
Slice LUTs (#)	230,400	16,512	7.17%	5748	2.49%	× 2.9
Block RAMs (#)	312	4	1.28%	4	1.28%	–
Power consumption		1.808 W		1.686 W		× 1.1
Minimum period		1.332 ns		1.308 ns		–
Maximum frequency		750.751 MHz		764.526 MHz		–

7 Conclusions

In this paper, we introduced a growing trend toward hardware offloading to increase computing power and then pointed out that the literature lacked an efficient implementation of the entropy filter. We also argued that the conventional entropy calculation is not hardware-friendly and hereby proposed a hardware-oriented alternative. We remolded the conventional into a novel three-step method of sorting, adjacent difference calculation, and pipelined entropy calculation. After that, we presented a corresponding hardware accelerator and demonstrated its superiority over the conventional design. Implementation results on a Zynq UltraScale+ XCZU7EV-2FFVC1156 MPSoC device showed 2.4× and 2.9× reductions in resource utilization and 1.1× reduction in power consumption. With this astonishing compactness and efficacy, our proposed architecture can still provide a high throughput of handling 764.526 megapixels per second, rendering it highly beneficial for computer vision edge devices.

Appendix: conventional hardware architecture

Figure 6 illustrates the block diagram of the conventional hardware architecture for the entropy filter. The design is based on the sliding mechanism (described in Sect. 4.2) and the cumulative histogram [8].

Given n, representing a set of discrete intensities assigned to a pixel, this conventional design requires n registers, corresponding to Bin 0, Bin 1, …, Bin $(n-1)$ in Fig. 6, to store the number of occurrences of each pixel intensity. Each bin can increase or decrease by a specific amount based on whether the associated pixel intensity is about to enter or leave the sliding window. As shown in Fig. 6, when the window moves to an adjacent location, it will exclude pixels in the red column from its coverage while including pixels in the blue column. To handle these two groups of pixels, the conventional design utilizes register banks, as shown Fig. 6.

The next block determines the quantity to increase or decrease each histogram bin according to input pixel intensities. To implement this block, [18] proposed using barrel shifters, which were more memory-efficient than ROMs in [8]. Specifically, a barrel shifter decodes a $\lceil \log n \rceil$-bit input

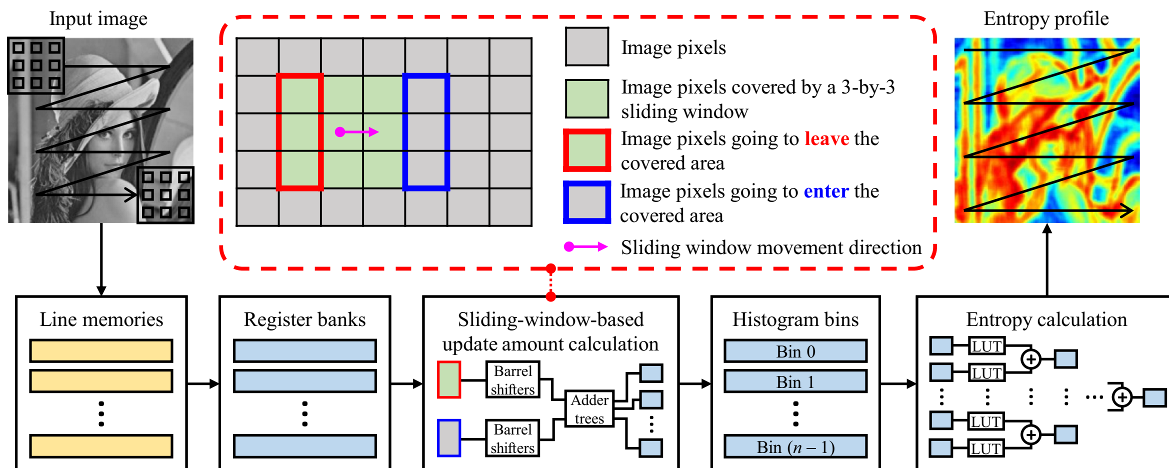

Fig. 6 Block diagram of the conventional hardware architecture. We illustrate the sliding mechanism using a 3 × 3 window

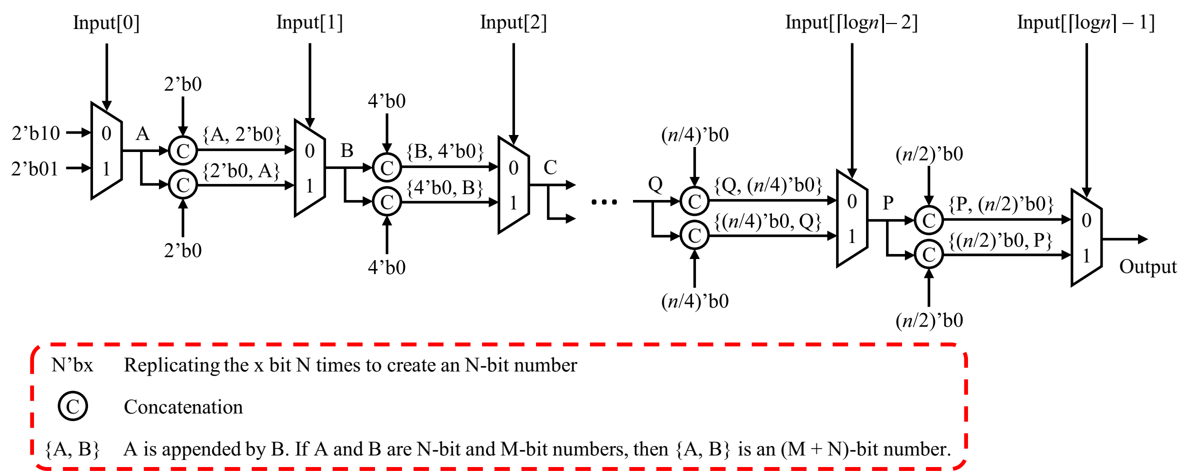

Fig. 7 Block diagram of the barrel shifter. Input and output are $\lceil \log n \rceil$-bit and n-bit data, respectively. We adopt the square bracket to access individual bits of the input

into an n-bit signal, with each bit indicating an increment or decrement of a bin associated with the input intensity. For example, an 8-bit input of 11111110_2 ($= 254_{10}$) causes the barrel shifter to produce a 256-bit output of $4000\ldots00_{16}$, signifying an increment or decrement of Bin 254. Table 3 demonstrates the input–output relationship of the barrel shifter.

Given an $S \times S$ window, S pixels are about to leave, and another S pixels are about to enter the window's coverage every time the window slides. Accordingly, the conventional design requires $2S$ barrel shifters (details of the barrel shifter are available in Fig. 7). Half of these shifters signify *bin decrease events* (BDEs), and the other half signifies *bin increase events* (BIEs). The conventional design then converts every bit of BDE outputs to two's complement representation and performs negation. Meanwhile, it solely converts every bit of BIE outputs to two's complement representation. Afterward, it adds these negative and positive two's complement numbers together using n adder trees to determine the quantity for increasing or decreasing each histogram bin.

Table 3 Input–output relationship of the barrel shifter

Input ($\lceil \log n \rceil$ bits) (in decimal)	Output (n bits) (in hexadecimal)
0	000 … 01
1	000 … 02
2	000 … 04
…	…
$n - 2$	400 … 00
$n - 1$	800 … 00

After updating histogram bins, the final block is to calculate the summation in (1). In this block, [18] utilized n LUTs to calculate summation terms corresponding to n bins. The content of each LUT is identical to that of the LUT described in Sect. 4.3. The outputs of these LUTs then undergo a large adder tree to compute the entropy value.

From the above description, it becomes evident that the conventional design is resource-consuming because it requires n registers for histogram bins and substantial quantities of resources for the resultant adder trees. Moreover, its space complexity is dependent on n and thus increases exponentially with n. Hence, it is inefficient compared to our proposed hardware architecture.

Acknowledgements This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2023R1A2C1004592).

Author Contributions BK conceptualized and supervised this study. DN wrote the software and the original manuscript. All authors reviewed the manuscript.

Data Availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interest The authors declare that they have no Conflict of interest.

References

1. (2006) IEEE standard for verilog hardware description language. IEEE Std 1364-2005 (Revision of IEEE Std 1374-2001), pp. 1–590. https://doi.org/10.1109/IEEESTD.2006.99495

2. Ahuja, D., Fang, I.T., Jiang, B., Sharma, A.: Inventors; Qualcomm Inc, assignee. Entropy based image separation. US patent US20120075440A1 (2012)
3. Beckmann, B.M., Wood, D.A. Managing Wire Delay in Large Chip-Multiprocessor Caches. In: Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37'04), pp. 319–330 (2004). https://doi.org/10.1109/MICRO.2004.21
4. Chen, W.T., Chen, P.Y., Hsiao, Y.C., Lin, S.H.: A low-cost design of 2D median filter. IEEE Access **7**, 150623–150629 (2019). https://doi.org/10.1109/ACCESS.2019.2948020
5. Chen, C., da Silva, B., Chen, R., Li, S., Li, J., Liu, C.: Evaluation of fast sample entropy algorithms on fpgas: from performance to energy efficiency. Entropy **24**, 1177 (2022). https://doi.org/10.3390/e24091177
6. Choi, L.K., You, J., Bovik, A.C.: Referenceless prediction of perceptual fog density and perceptual image defogging. IEEE Trans. Image Process. **24**(11), 3888–3901 (2015). https://doi.org/10.1109/TIP.2015.2456502
7. Crandall, R.E., Klivington, J.A., Merwe, R.vd. inventors; Apple Inc, assignee. Automatic Image Sharpening. US patent US20130084019A1 (2013)
8. Fahmy, S.A., Cheung, P.Y.K., Luk, W.: Novel FPGA-based implementation of median and weighted median filters for image processing. In: Proceedings of the International Conference on Field Programmable Logic and Applications, pp. 142–147 (2005). https://doi.org/10.1109/FPL.2005.1515713
9. ISO (2008) Quantities and units—Part 13: Information science and technology (IEC 80000-13:2008). https://www.iso.org/standard/31898.html. Accessed 31 July 2023
10. Kang, U.J., Kang, B.: Salient Object Detection (SOD) Algorithm based on Bayes Classifier. J. Korean Inst. Inf. Technol. **20**(8), 91–97 (2022). https://doi.org/10.14801/jkiit.2022.20.8.91
11. Knuth, D.E.: Networks for sorting. In: The Art of Computer Programming, pp. 219–247. Addison-Wesley, Boston (1998)
12. Kwan, E., Nunez-Yanez, J.: Entropy-driven adaptive filtering for high-accuracy and resource-efficient fpga-based neural network systems. Electronics **9**, 1765 (2020). https://doi.org/10.3390/electronics9111765
13. Lai, Y.K., Wellem, T., You, H.P.: Hardware-assisted estimation of entropy norm for high-speed network traffic. Electron. Lett. **50**(24), 1845–1847 (2014). https://doi.org/10.1049/el.2014.2377
14. Lakshmi, K.R., Padmaja, M.: Efficient and enhanced high throughput image denoising using chronical fuzzy set. In: Proceedings of the 4th International Conference on Electronics, Communication and Aerospace Technology (ICECA), pp. 924–930 (2020). https://doi.org/10.1109/ICECA49313.2020.9297429
15. Netronome: Agilio CX 2x40Gbe (2017). https://www.netronome.com/products/agilio-cx/. Accessed 31 July 2023
16. Ngo, D., Lee, S., Lee, G., Kang, B.: Single-image visibility restoration: a machine learning approach and its 4k-capable hardware accelerator. Sensors **20**, 5795 (2020). https://doi.org/10.3390/s20205795
17. Pooya, S.A., Farhad, R.: A novel median based image impulse noise suppression system using spiking neurons on FPGA. Comput. Methods Biomech. Biomed. Eng. Imaging Vis. **8**(6), 631–640 (2020). https://doi.org/10.1080/21681163.2020.1777464
18. Sim, H., Kang, B.: Hardware architecture for entropy filter implementation. J. IKEEE **26**(2), 226–231 (2022). https://doi.org/10.7471/ikeee.2022.26.2.226
19. Soto, J.E., Ubisse, P., Hernández, C., Figueroa, M.: A hardware accelerator for entropy estimation using the top-k most frequent elements. In: Proceedings of the 2020 23rd Euromicro Conference on Digital System Design (DSD), pp. 141–148 (2020). https://doi.org/10.1109/DSD51259.2020.00032
20. Subramaniam, J., Kannan, R.J., Ebenezer, D.: Parallel and pipelined 2-D median filter architecture. IEEE Embed. Syst. Lett. **10**(3), 69–72 (2018). https://doi.org/10.1109/LES.2017.2771453
21. Wellem, T., Lai, Y.K., Huang, C.Y., Chung, W.Y.: A flexible sketch-based network traffic monitoring infrastructure. IEEE Access **7**, 92476–92498 (2019). https://doi.org/10.1109/ACCESS.2019.2927863
22. Xilinx: Zynq UltraScale+ MPSoC data sheet: overview (2022). https://docs.xilinx.com/v/u/en-US/ds891-zynq-ultrascale-plus-overview. Accessed 1 Aug 2023

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.