A Distributed Canny Edge Detector: Algorithm and FPGA Implementation

Qian Xu, Srenivas Varadarajan, Chaitali Chakrabarti, *Fellow, IEEE*, and Lina J. Karam, *Fellow, IEEE*

***Abstract*— The Canny edge detector is one of the most widely used edge detection algorithms due to its superior performance. Unfortunately, not only is it computationally more intensive as compared with other edge detection algorithms, but it also has a higher latency because it is based on frame-level statistics. In this paper, we propose a mechanism to implement the Canny algorithm at the block level without any loss in edge detection performance compared with the original frame-level Canny algorithm. Directly applying the original Canny algorithm at the block-level leads to excessive edges in smooth regions and to loss of significant edges in high-detailed regions since the original Canny computes the high and low thresholds based on the frame-level statistics. To solve this problem, we present a distributed Canny edge detection algorithm that adaptively computes the edge detection thresholds based on the block type and the local distribution of the gradients in the image block. In addition, the new algorithm uses a nonuniform gradient magnitude histogram to compute block-based hysteresis thresholds. The resulting block-based algorithm has a significantly reduced latency and can be easily integrated with other block-based image codecs. It is capable of supporting fast edge detection of images and videos with high resolutions, including full-HD since the latency is now a function of the block size instead of the frame size. In addition, quantitative conformance evaluations and subjective tests show that the edge detection performance of the proposed algorithm is better than the original frame-based algorithm, especially when noise is present in the images. Finally, this algorithm is implemented using a 32 computing engine architecture and is synthesized on the Xilinx Virtex-5 FPGA. The synthesized architecture takes only 0.721 ms (including the SRAM READ/WRITE time and the computation time) to detect edges of 512 × 512 images in the USC SIPI database when clocked at 100 MHz and is faster than existing FPGA and GPU implementations.**

***Index Terms*— Distributed image processing, Canny edge detector, high throughput, parallel processing, FPGA.**

I. Introduction

EDGE detection is the most common preprocessing step in many image processing algorithms such as image enhancement, image segmentation, tracking and image/video coding. Among the existing edge detection algorithms, the Canny edge detector has remained a standard for many years and has best performance [10]. Its superior performance is due to the fact that the Canny algorithm performs hysteresis thresholding which requires computing high and low thresholds based on the entire image statistics. Unfortunately, this feature makes the Canny edge detection algorithm not only more computationally complex as compared to other edge detection algorithms, such as the Roberts and Sobel algorithms, but also necessitates additional pre-processing computations to be done on the entire image. As a result, a direct implementation of the Canny algorithm has high latency and cannot be employed in real-time applications.

Many implementations of the Canny algorithm have been proposed on a wide list of hardware platforms. There is a set of work [1]–[3] on Deriche filters that have been derived using Canny's criteria and implemented on ASIC-based platforms. The Canny-Deriche filter [1] is a network with four transputers that detect edges in a 256 × 256 image in 6s, far from the requirement for real-time applications. Although the design in [2] improved the Canny-Deriche filter implementation of [1] and was able to process 25 frames/s at 33 MHz, the used off-chip SRAM memories consist of Last-In First-Out (LIFO) stacks, which increased the area overhead compared to [1]. Demigny proposed a new organization of the Canny-Deriche filter in [3], which reduces the memory size and the computation cost by a factor of two. However, the number of clock cycles per pixel of the implementation [3] varies with the size of the processed image, resulting in variable clock-cycles/pixel from one image size to another with increasing processing time as the image size increases.

There is another set of work [4]–[6] on mapping the Canny edge detection algorithm onto FPGA-based platforms. The two FPGA implementations in [4] and [5] translate the software design directly into VHDL or Verilog using system-level hardware design tools, which results in a decreased timing performance as shown later in Section 6 of this paper. The parallel implementation in [6] operates on only 4 pixels in parallel, resulting in an increase in the number of memory accesses and in the processing time as compared to the proposed algorithm. Furthermore, in order to reduce the computational complexity, all of these implementations compute the high and low thresholds off-line and use the same fixed threshold values for all the images, which result in a decreased edge detection performance. In [7], a self-adapt threshold Canny algorithm is proposed for a mobile robot system where a low and high threshold values are computed for each input image and an implementation for an Altera Cyclone FPGA is presented.

Manuscript received December 10, 2012; revised June 20, 2013 and November 7, 2013; accepted November 17, 2013. Date of publication March 18, 2014; date of current version June 3, 2014. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Marios S. Pattichis.

The authors are with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ 85281 USA (e-mail: qianxu@asu.edu; svarada2@asu.edu; chaitali@asu.edu; karam@asu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIP.2014.2311656

Recently, the General Purpose Graphic Processing Unit (GPGPU) has emerged as a powerful and accessible parallel computing platform for image processing applications [8], [9]. Studies of GPGPU accelerated Canny edge detection have been presented [10]–[12]. All of these implementations are frame-based and do not have good edge detection performance since they use the same fixed pair of high and low threshold values for all images. Furthermore, as shown later in the paper, their timing performance is inferior compared to the proposed algorithm in spite of being operated at a very high clock frequency.

In the original Canny method, the computation of the high and low threshold values depends on the statistics of the whole input image. However, most of the above existing implementations (e.g., [4]–[6], [10]–[12]) use the same fixed pair of high and low threshold values for all input images. This results in a decreased edge detection performance as discussed later in this paper. The non-parallel implementation [7] computes the low and high threshold values for each input image. This results in increased latency as compared to the existing implementations (e.g., [4]–[6], [10]–[12]). Furthermore, the non-parallel implementations ([4]–[7]) result in a decreased throughput as compared to the parallel implementations ([6], [10]–[12]). The issue of increased latency and decreased throughput is becoming more significant with the increasing demand for large-size high-spatial resolution visual content (e.g., High-Definition and Ultra High-Definition).

Our focus is on reducing the latency and increasing the throughput of the Canny edge detection algorithm so that it can be used in real-time processing applications. As a first step, the image can be partitioned into blocks and the Canny algorithm can be applied to each of the blocks in parallel. Unfortunately, directly applying the original Canny at a block-level would fail since it leads to excessive edges in smooth regions and loss of significant edges in high-detailed regions. In this paper, we propose an adaptive threshold selection algorithm which computes the high and low threshold for each block based on the type of block and the local distribution of pixel gradients in the block. Each block can be processed simultaneously, thus reducing the latency significantly. Furthermore, this allows the block-based Canny edge detector to be pipelined very easily with existing block-based codecs, thereby improving the timing performance of image/video processing systems. Most importantly, conducted conformance evaluations and subjective tests show that, compared with the frame-based Canny edge detector, the proposed algorithm yields better edge detection results for both clean and noisy images.

The block-based Canny edge detection algorithm is mapped onto an FPGA-based hardware architecture. The architecture is flexible enough to handle different image sizes, block sizes and gradient mask sizes. It consists of 32 computing engines configured into 8 groups with 4 engines per group. All 32 computing engines work in parallel lending to a 32-fold decrease in running time without any change in performance when compared with the frame-based algorithm. The architecture has been synthesized on the Xilinx Virtex-5 FPGA. It occupies 64% of the total number of slices and 87% of the local memory, and takes 0.721ms (including the SRAM read/write time and the computation time) to detect edges of 512×512 images in the USC SIPI database when clocked at 100 MHz.

A preliminary version of this work was presented in [13]. The work presented herein does not only provide more elaborate discussions and performance results but also provides an improved distributed Canny edge detector both at the algorithm and architecture levels. The algorithm is improved by proposing a novel block-adaptive threshold selection procedure that exploits the local image characteristics and is more robust to changes in block size as compared to [13]. While the architecture presented in [13] was limited to a fixed image size of 512×512, a fixed image block of 64×64, this paper presents a general FPGA-based pipelined architecture that can support any image size and block size. Furthermore, no FPGA synthesis results and no comparison results with existing techniques were presented in [13]. In this paper, FPGA synthesis results, including the resource utilization, execution time, and comparison with existing FPGA implementations are presented.

The rest of the paper is organized as follows. Section 2 gives a brief overview of the original Canny algorithm. Section 3 presents the proposed distributed Canny edge detection algorithm which includes the adaptive threshold selection algorithm and a non-uniform quantization method to compute the gradient magnitude histogram. Quantitative conformance as well as subjective testing results are presented in Section 4 in order to illustrate the edge detection performance of the proposed distributed Canny algorithm as compared to the original Canny algorithm for clean as well as noisy images. In addition, the effects of the gradient mask size and the block size on the performance of the proposed distributed Canny edge detection scheme are discussed and illustrated in Section 4. The proposed hardware architecture and the FPGA implementation of the proposed algorithm are described in Section 5. The FPGA synthesis results and comparisons with other implementations are presented in Section 6. Finally, conclusions are presented in Section 7.

II. Canny Edge Detection Algorithm

Canny developed an approach to derive an optimal edge detector to deal with step edges corrupted by a white Gaussian noise. The original Canny algorithm [14] consists of the following steps: 1) Calculating the horizontal gradient G_x and vertical gradient G_y at each pixel location by convolving with gradient masks. 2) Computing the gradient magnitude G and direction θ_G at each pixel location. 3) Applying Non-Maximal Suppression (NMS) to thin edges. This step involves computing the gradient direction at each pixel. If the pixel's gradient direction is one of 8 possible main directions (0°, 45°, 90°, 135°, 180°, 225°, 270°, 315°, the gradient magnitude of this pixel is compared with two of its immediate neighbors along the gradient direction and the gradient magnitude is set to zero if it does not correspond to a local maximum. For the gradient directions that do not coincide with one of the 8 possible main directions, an interpolation is done to compute the neighboring gradients. 4) Computing high and low thresholds based on the histogram of the gradient magnitude for the entire image.

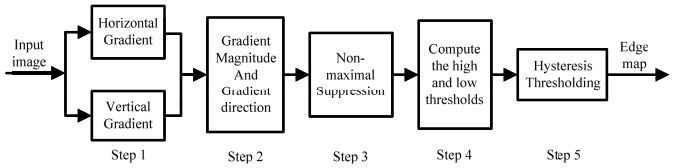


Fig. 1. Block diagram of the Canny edge detection algorithm.

The high threshold is computed such that a percentage P_1 of the total pixels in the image would be classified as strong edges. In other words, the high threshold corresponds to the point at which the value of the gradient magnitude cumulative distribution function (CDF) equals to 1-P_1. The low threshold is computed as a percentage P_2 of the high threshold. The values of P_1 and P_2 are typically set as 20% and 40%, respectively [14], [15]. 5) Performing hysteresis thresholding to determine the edge map. If the gradient magnitude of a pixel is greater than the high threshold, this pixel is considered as a strong edge. If the gradient magnitude of a pixel is between the low threshold and high threshold, the pixel is labeled as a weak edge. Strong edges are interpreted as "certain edges", and can be immediately included in the final edge images. Weak edges are included if and only if they are connected to strong edges.

A block diagram of the Canny edge detection algorithm [14] is shown in Fig. 1. In this original Canny edge detection algorithm [14], the gradient calculation (Step 1) is performed by using Finite-Inpulse Response (FIR) gradient masks designed to approximate the following 2D sampled versions of the partial derivatives of a Gaussian function:

$$F_x(x,y) = -\frac{x}{\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} = \left(-x e^{-\frac{x^2}{\sigma^2}}\right)\left(\frac{1}{\sigma^2} e^{-\frac{y^2}{\sigma^2}}\right) \quad (1)$$

$$F_y(x,y) = -\frac{y}{\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} = \left(-y e^{-\frac{y^2}{\sigma^2}}\right)\left(\frac{1}{\sigma^2} e^{-\frac{x^2}{\sigma^2}}\right) \quad (2)$$

where σ is the standard deviation of the Gaussian function. The size of the gradient masks used by the Canny edge detector is usually implemented as a function of the chosen σ, with larger values of σ yielding larger masks. However, the best choice of σ is image-dependent and can be selected by the user based on knowledge of the present noise characteristics or the size of desired objects in the image [16]. The parameter σ can also be set by a separate application that estimates the noise and/or scale of objects in the image. The effect of the gradient mask size is illustrated in Section 4.1.1.

III. Proposed Distributed Canny Edge Detection Algorithm

As discussed in Section 2, the classical Canny edge detection algorithm sets the high and low thresholds based on the distribution of the gradients at all the pixel locations of an image. Thus, directly applying the original Canny algorithm to a block of the image would fail to detect the desired edges since the statistics of a block may differ significantly from the statistics of the entire natural image. For example, Fig. 2 shows two types of blocks, namely smooth block and edge

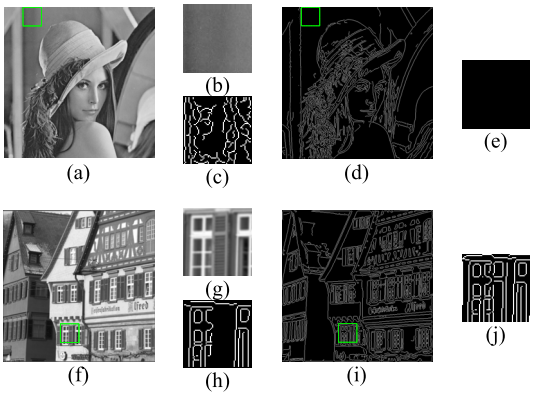


Fig. 2. (a), (f) Original Lena and Houses images; (b), (g) zoomed-in blocks of the Lena and Houses images; (c), (h) edge maps of block images in (b) and (g) obtained by applying the classical Canny edge detector to blocks separately; (d), (i) edge maps using the Canny edge detector applied to the entire Lena image and Houses image; (e), (j) zoomed-in blocks of the edge maps (d) & (i).

block, extracted from the Lena image and Houses image. Despite the fact that the smooth block [Fig. 2(b)] does not contain any significant edges, applying the original Canny algorithm to this block results in the detection of an excessive number of edges as shown in Fig. 2(c). This is due to the fact that the Canny algorithm assumes that a percentage P_1 of the total pixels in the block are strong edges. On the other hand, for an edge block [Fig. 2(g)], the majority of pixels are edges. Since P_1 is much smaller than the actual percentage of edge pixels in the edge block, it results in missing edges as shown in Fig. 2(h). For comparison, Fig. 2(d), (e), (i), and (j) show the performance of the Canny edge detector when applied to the entire image. In latter case, the Canny edge detector is able to achieve the expected edge detection performance in the smooth regions [Fig. 2(e)] and in the edge regions [Fig. 2(j)].

In order to improve the performance of the edge detection at the block level and achieve the same performance as the original frame-based Canny edge detector when this latter one is applied to the entire image, a distributed Canny edge detection algorithm is proposed. A diagram of the proposed algorithm is shown in Fig. 3. In the proposed distributed version of the Canny algorithm, the input image is divided into $m \times m$ overlapping blocks and the blocks are processed independent of each other. For an $L \times L$ gradient mask, the $m \times m$ overlapping blocks are obtained by first dividing the input image into $n \times n$ non-overlapping blocks and then extending each block by $(L+1)/2$ pixels along the left, right, top, and bottom boundaries, respectively. This results in $m \times m$ overlapping blocks, with $m = n + L + 1$. The non-overlapping $n \times n$ blocks need to be extended in order to prevent edge artifacts and loss of edges at block boundaries while computing the gradients and due to the fact that the NMS operation at boundary pixels requires the gradient values of the neighboring pixels of the considered boundary pixels in a block. Fig. 4 shows an example of non-overlapping block and its extended overlapping block version in the case when a 3×3 gradient mask. In order to perform NMS for the border pixel (i, j), the gradient information of the adjacent pixels

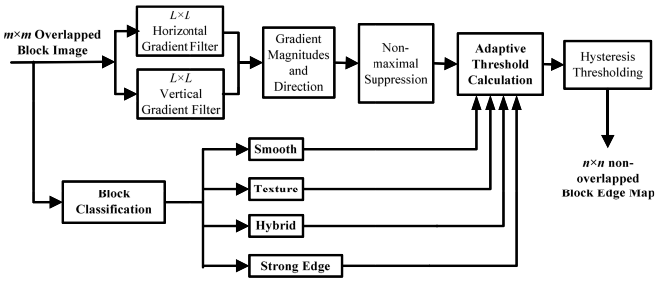


Fig. 3. Proposed distributed Canny edge detection algorithm.

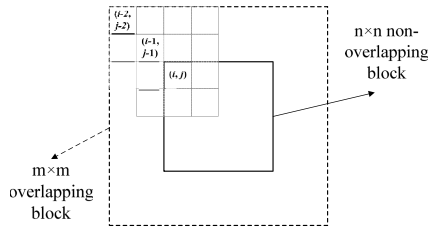


Fig. 4. An example of the structure of an $m \times m$ overlapping block, where $m = n + L + 1$ for an $L \times L (L = 3)$ gradient mask. and when the image is initially divided into $n \times n$ non-overlapping blocks.

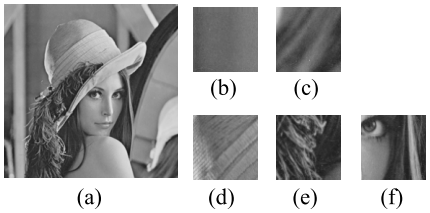


Fig. 5. (a) Original 512 × 512 Lena image; b) uniform block; (c) uniform/texture block; (d) texture block; (e) edge/texture block; (f) medium edge block of the Lena image. Shown blocks are of size 64 × 64.

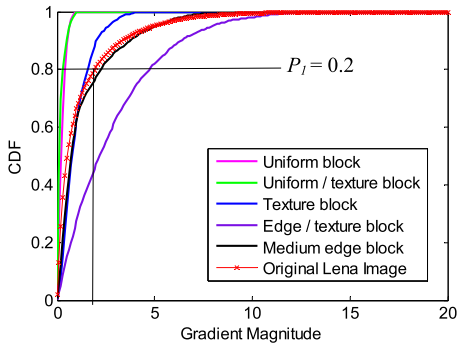


Fig. 6. Normalized gradient magnitude CDFs for the 512 × 512 Lena image and CDFs for the 64 × 64 uniform block, uniform/texture block, texture block, edge/texture block, medium/edge block shown in Fig. 5.

$(i-1, j-1)$, $(i-1, j)$, $(i-1, j+1)$, $(i, j-1)$, $(i+1, j-1)$ are needed. In order to compute the gradient of the adjacent pixels $(i-1, j-1)$, $(i-1, j)$, $(i-1, j+1)$, $(i, j-1)$, $(i+1, j-1)$ for the 3×3 gradient mask, the block has to be extended by 2 (where $(L-1)/2 + 1 = 2$) pixels on all sides in order to generate a block of size $(n+4) \times (n+4)$. Thus, m equals to $n+4$ for this example. Note that, for each block, only edges in the central $n \times n$ non-overlapping region are included in the final edge map, where $n = m - L - 1$. Steps 1 to 3 and Step 5 of the distributed Canny algorithm are the same as in the original Canny algorithm except that these are now applied at the block level. Step 4, which is the hysteresis high and low thresholds calculation, is modified to enable parallel block-level processing without degrading the edge detection performance.

As discussed above, exploiting the local block statistics can significantly improve the edge detection performance for the distributed Canny edge detector. For this purpose, in order to learn the statistics of local blocks, we use a training database consisting of natural images. For each image in the database, we first divide the image into $n \times n$ non-overlapping blocks and classify the blocks into six types, uniform, uniform/texture, texture, edge/texture, medium edge, and strong edge block, by adopting the block classification method of [17]. This classification method utilized the local variance of each pixel using a 3×3 window that is centered around the considered pixel in order to label it as of type edge, texture, or uniform pixel. Then, each block is classified based on the total number of edge, texture, and uniform pixels in the considered block. Fig. 5 shows an example where the 512×512 Lena image is divided into blocks, and each block is classified according to this classification technique. Fig. 5(b)–(f) provide, respectively, examples of uniform, uniform/texture, texture, edge/texture, medium edge block for the 512×512 Lena image with a block size of 64×64. Note that there are no strong edge blocks for the Lena image.

In order not to degrade the performance of the original frame-based Canny algorithm when it is applied to a block, the high and low thresholds when computed at the block level should match thresholds that would have been obtained for the entire image. The gradient magnitude CDFs of each block in Fig. 5 are shown in Fig. 6,[1] along with the gradient-magnitude CDF of the entire Lena image. According to the CDF of the entire Lena image, the high threshold should be selected as 1.8 corresponding to a P_1 value (percentage of pixels that should be classified as strong edges) of 0.2. However, if P_1 is still set to 0.2 for the medium edge block, the local high threshold for that block would be about 5.2, which is significantly different from the high threshold, 1.8, that is obtained for the entire image. Such a setting will result in a loss of significant edges in the considered strong edge block. On the other hand, the local high thresholds for the uniform and uniform/texture block would be about 0.7, if P_1 is set to 0.2. Such a setting will lead to excessive edges in the uniform and uniform/texture blocks. From this analysis, it can be seen that, in order to keep a similar threshold for all block types, P_1 should be selected differently for different types of blocks by adapting its value to the local block content.

In order to determine, for each block type, the appropriate percentage value P_1 that would result in high and low threshold values similar to the ones that are obtained for the

[1] In Fig. 6, the horizontal and vertical gradients are calculated by using 3×3 gradient masks and thus m equals to 68.

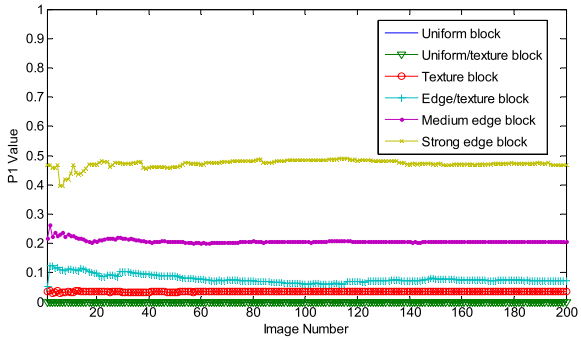


Fig. 7. P_1 values for each block type.

TABLE I
STANDARD DEVIATIONS OF P_1 VALUES FOR EACH BLOCK TYPE FOR 64 × 64 BLOCK

Block type	Uniform	Uniform/ Texture	Texture	Edge/ Texture	Medium edge	Strong edge
Var. of P_1	0	0	9.37 E-4	0.0138	0.0071	0.013

entire image, a training set[2] of 200 images is formed from the Berkeley Segmentation Image Database [18]. For each image in the training database, the high threshold of the entire image is first calculated. Then, the image is divided into blocks and the blocks are classified into six block types as discussed previously. Then, for each block type, the gradient magnitude CDF is computed and the corresponding CDF used to compute the P_1 value such that the local high threshold of the blocks in this class is the same as the one for the entire image. Fig. 7 shows the P_1 values that are obtained for different block types, each of size 64 × 64, for 200 512 × 512 images with varied content. It illustrates that the P_1 values of each block type are highly different from each other, except for the uniform block and uniform/texture block types. Also, for a given block type, the P_1 values across all 200 images are quite similar for different block sizes. The final P_1 value for a considered block type is computed as the average value of its corresponding set over all images and over all block sizes. For illustration, the standard deviations of the obtained P_1 values for each block type are shown in Table I for 64 × 64 blocks.

To evaluate the robustness of the obtained P_1 values with respect to the block size, the 512×512 images are divided into fixed-size blocks, with the block size varying from 8 × 8 to 256 × 256. Table II shows the P_1 values that are obtained for each block type and for each block size. It should be noted that the P_1 values for uniform and uniform/texture blocks are equal to 0 for all block sizes, which indicates that the uniform and uniform/texture blocks can be combined into one block type, which we refer to as smooth block type. Also, this implies that there are no pixels that should be classified as edges in a smooth block. Therefore, there is no need to perform edge detection on smooth blocks, and this results in reduced computations, a feature that is exploited in the FPGA implementation.

[2]In order to accommodate various block sizes that are integer powers of 2, the size of images in the dataset was extended from the original size of 321 × 418 to 512 × 512.

TABLE II
P_1 VALUES FOR EACH BLOCK TYPE WITH DIFFERENT BLOCK SIZES

Block Size	Block Type					
	Uniform	Uniform /texture	Texture	Edge /texture	Medium edge	Strong edge
8×8	*0*	*0*	0.0312	0.1022	0.2183	0.4820
16×16	*0*	*0*	0.0307	0.1016	0.2616	0.4830
32×32	*0*	*0*	0.0305	0.1117	0.2079	0.4852
64×64	*0*	*0*	0.0318	0.1060	0.2218	0.4670
128×128	*0*	*0*	0.0302	0.0933	0.2375	0.4842
256×256	*0*	*0*	0.0299	0.0911	0.2304	0.4893

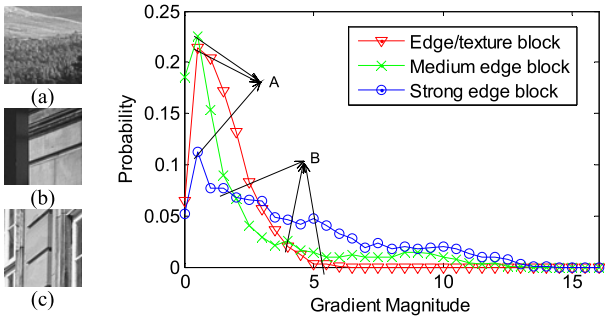


Fig. 8. (a)-(c) Different types of 64 × 64 blocks ((a) Edge/texture, (b) medium edge, (c) strong edge) and (d) corresponding gradient magnitude histograms.

In order to compute the high and low hysteresis thresholds, a finely quantized gradient magnitude histogram is usually needed. Here, we employ the non-uniform quantizer, which has been proposed by us in [13], to obtain the gradient magnitude histogram for each block such that the high threshold can be precisely calculated. As in [7] and [13], for the edge/texture, medium edge and strong edge block, it was observed that the largest peak in the gradient magnitude histograms of the Gaussian-smoothed images occurs near the origin and corresponds to low-frequency content, while edge pixels form a series of smaller peaks, where each peak corresponds to a class of edges having similar gradient magnitudes. Consequently, the high threshold should be selected between the largest peak and the second largest edge peak. Examples of gradient magnitude histograms for an edge/texture, medium edge and strong edge block are shown in Fig. 8. Based on the above observation, we proposed a non-uniform quantizer to discretize the gradient-magnitude histogram in [13]. Specifically, the quantizer needs to have more quantization levels in the region between the largest peak A and the second largest peak B and few quantization levels in other parts. Fig. 9 shows the schematic diagram of the proposed non-uniform quantizer. The first reconstruction level (R_1) is computed as the average of the maximum value and minimum value of the gradient magnitude in the considered block, and the second reconstruction level (R_2) is the average of the minimum value of the gradient magnitude and the first reconstruction level. Accordingly, n reconstruction levels can be computed as shown in [13], $R_1 = (min+max)/2$ and $R_{i+1} = (min+R_i)/2$ ($i = 2, 3, \ldots n$), where min and max represent the minimum and maximum values of the gradient magnitude, respectively, and R_i is the reconstruction level.

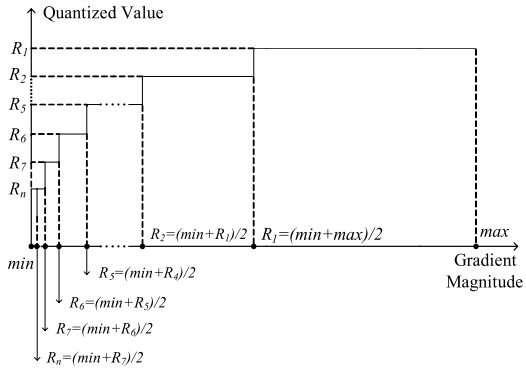


Fig. 9. Reconstruction values and quantization levels.

Step 1: Pixel classification

$$pixel\ type = \begin{cases} uniform, & \text{var}(x,y) \le T_u \\ texture, & T_u < \text{var}(x,y) \le T_e \\ edge, & T_e < \text{var}(x,y) \end{cases}$$

Step 2: Block classification

Block type	No. of pixels of pixel type	
	$N_{uniform}$	N_{edge}
Smooth	$\ge 0.3 \cdot Total\_Pixel$	0
Texture	$< 0.3 \cdot Total\_Pixel$	0
Edge/texture	$< 0.65(Total\_Pixel - N_{edge})$	(>0) & ($< 0.3 \cdot Total\_Pixel$)
Medium edge	$\ge 0.65(Total\_Pixel - N_{edge})$	(>0) & ($< 0.3 \cdot Total\_Pixel$)
Strong edge	$\le 0.7 \cdot Total\_Pixel$	$\ge 0.3 \cdot Total\_Pixel$

$var(x, y)$: the local (3×3) variance at pixel (x, y);
T_u and T_e: two thresholds as in [17];
$Total\_Pixel$: the total number of pixels in the block;
$N_{uniform}$: the total number of uniform pixels in the block;
N_{edge}: the total number of edge pixels in the block;

(a)

Let P_1 be the percentage of pixels, in a block, that would be classified as strong edges.

```
Step 1: If smooth block type
            P1 = 0;                 /* No edges*/
        else if texture block type
            P1 = 0.03;              /* Few edges*/
        else if texture/edge block type
            P1 = 0.1;               /* Some edges*/
        else if medium edge block type
            P1 = 0.2;               /* Medium edges*/
        else
            P1 = 0.4;               /* Many edges*/
Step 2: Compute the 8-bin non-uniform gradient magnitude
        histogram and the corresponding cumulative distribution
        function F(G).
Step 3: Compute High_threshold as F(High_threshold) = 1-P1
Step 4: Compute Low_threshold = 0.4*High_threshold
```

(b)

Fig. 10. Pseudo-codes for the proposed (a) block classification and (b) adaptive threshold selection scheme.

The pseudo-code of the block classification technique in [17] and the proposed adaptive threshold selection algorithm is shown in Fig. 10(a) and (b), respectively.

IV. MATLAB Experimental Results

A. Parametrical Analysis

The performance of the proposed algorithm is affected by two parameters, the mask size and the block size. We discuss what is the best choice for the mask size for different types of images, and what is the smallest block size that can be used by our proposed distributed Canny algorithm without sacrificing performance.

(a) (b) (c)

Fig. 11. (a) Original "Houses" image; Edge map using the original Canny algorithm (b) with a 9 × 9 ($\sigma = 1.4$) gradient mask and (c) with a 3 × 3 ($\sigma = 0.4$) gradient mask.

1) The Effect of Mask Size: As indicated in Section 2, the size of the gradient mask is a function of the standard deviation σ of the Gaussian filter, and the best choice of σ is based on the image characteristics. Canny has shown in [14] that the optimal operator for detecting step edges in the presence of noise is the first derivative of the Gaussian operator. As stated in Section 2, for the original Canny algorithm as well as the proposed algorithm, this standard deviation is a parameter that is typically set by the user based on the knowledge of sensor noise characteristics. It can also be set by a separate application that estimates the noise and/or blur in the image. A large value of σ results in smoothing and improves the edge detector's resilience to noise, but it undermines the detector's ability to detect the location of true edges. In contrast, a smaller mask size (corresponding to a lower σ) is better for detecting detailed textures and fine edges but it decreases the edge detector's resilience to noise.

An L-point even-symmetric FIR Gaussian pulse-shaping filter design can be obtained by truncating a sampled version of the continuous-domain Gaussian filter of standard deviation σ. The size L of the FIR Gaussian filter depends on the standard deviation σ and can be determined as follows:

$$L = 2 \cdot L_{side} + 1 \tag{3}$$

$$L_{side} = \left\lfloor 2\sigma\sqrt{(-1)\log(C_T)} \right\rfloor \tag{4}$$

where C_T represents the cut-off value in the spatial domain of the continuous-domain Gaussian function and determines the cut-off error. A smaller C_T corresponds to a smaller cut-off error. In our implementation, in order to make the cut-off error small for all σ, C_T is chosen to be a small value (e.g., $C_T = 10^{-3}$). Fig. 11 shows that the relatively smaller size 3 × 3 filter (σ from 0.3 to 0.4) can detect the edges of a detailed texture image, while Fig. 12 shows that the larger 9 × 9 filter (σ from 1.2 to 1.4) suppresses noise more effectively. In addition, Fig. 13 shows that the 3 × 3 filter is exhibits a better performance ingenerating the edge maps of blurred images as compared to the 9 × 9 filter.

2) Block Size: To find out the smallest block size for which the proposed Canny algorithm can detect all the psycho-visually important edges, the perceptual visual quality of the obtained edge maps was assessed using visual quality metrics. More specifically, the quality of edge maps obtained by using different block sizes was analyzed by evaluating their corresponding Pearson's correlation coefficient (PCC), which

(a) (b) (c)

Fig. 12. (a) "Houses" image with Gaussian white noise ($\sigma_n = 0.01$); Edge map using the original Canny algorithm (b) with a 9×9 ($\sigma = 1.4$) gradient mask and (c) with a 3×3 ($\sigma = 0.4$) gradient mask.

(a) (b) (c)

Fig. 13. (a) Gaussian Blurred "Houses" image ($\sigma_{blur} = 2$); Edge map using the original Canny algorithm (b) with a 9×9 ($\sigma = 1.4$) gradient mask and (c) with a 3×3 ($\sigma = 0.4$) gradient mask.

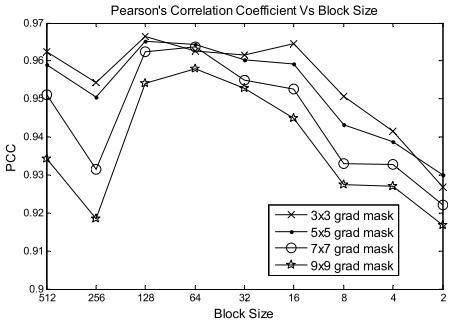


Fig. 14. Pearson's correlation coefficient reflecting the correlation between the CPBD sharpness metric [19] and MOS as a function of block size and mask size.

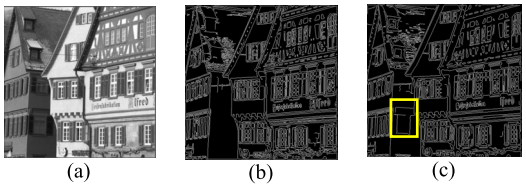
was used to measure how well state-of-the-art edge-based sharpness metrics commensurate with the actual perceived sharpness. For this purpose, the psycho-visual sharpness metric of [19] is used, which primarily depends on the edge map to estimate the cumulative probability of detecting blur (CPBD) in an image. This metric was shown in [19] to achieve the best performance among existing sharpness/blur metrics. It calculates the probability of detecting blur at each edge in an image and then obtains a final quality score over the entire image by evaluating the cumulative probability of blur detection. Four images with varying image characteristics were chosen. These were blurred using Gaussian masks with six different σ values (0.5, 1, 1.5, 2, 2.5, and 3) to generate a set of 24 images. The images were displayed one after another in a random order and subjects were asked to rate them on a 5-point scale of 1 (poor) to 5 (excellent) to form a Mean Opinion Score (MOS) by averaging the subjective scores for each image. Fig. 14 depicts the effect of replacing the frame-based Canny edge detection with the proposed distributed Canny edge detection on the sharpness metric of [19], for masks of different sizes and for different block sizes. Note that a 512×512 block corresponds to the original frame-based Canny edge detector. Fig. 14 shows that PCC has a maximal value when the block size is 64×64. It also illustrates that,

(a) (b) (c)

Fig. 15. (a) 512×512 "Salesman" image; edge-maps of (b) the original Canny edge detector, and (c) the proposed algorithm with a non-overlapping block size of 64×64, using a 3×3 gradient mask.

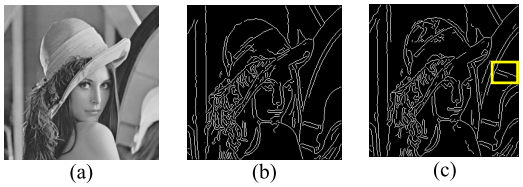
(a) (b) (c)

Fig. 16. (a) 512×512 "Fruit" image; edge-maps of (b) the original Canny edge detector, and (c) the proposed algorithm with a non-overlapping block size of 64×64, using a 3×3 gradient mask.



(a) (b) (c)

Fig. 17. (a) 512×512 "Houses" image; edge-maps of (b) the original Canny edge detector, and (c) the proposed algorithm with a non-overlapping block size of 64×64, using a 3×3 gradient mask.



(a) (b) (c)

Fig. 18. (a) 512×512 "Lena" image; edge-maps of (b) the original Canny edge detector, and (c) the proposed algorithm with a non-overlapping block size of 64×64, using a 3×3 gradient mask.

for the various block sizes, the increase or decrease in the PCC is insignificant. In other words, the PCC is robust to changes in block size. In conclusion, the proposed distributed Canny algorithm can detect all the psycho-visually important edges for images with moderate noise and blur levels, similar to the original frame-based Canny algorithm.

B. Edge Detection Performance Analysis

The edge detection performance of the proposed distributed approach is analyzed by comparing the perceptual significance of its resulting edge map with the one produced by the original frame based Canny edge detector. Figs. 15–18 show the edge maps that are obtained by the original frame-based Canny edge detector and the proposed distributed Canny edge detection algorithm with 3×3 gradient masks and a non-overlapping block size of 64×64 ($n = 64$; $m = 68$) for the 512×512 "Salesman," "Fruit," "Houses," and "Lena" images,

Fig. 19. Comparison of the edge maps of noisy images by using the original Canny edge detector and the proposed method: (a) images with Gaussian white noise ($\sigma_n = 0.01$); edge-maps of (b) the original Canny edge detector, and (c) the proposed algorithm with a non-overlapping block size of 64×64, using a 9×9 gradient mask.

respectively. As shown in Figs. 15 and 16, the proposed distributed Canny edge detection algorithm yields comparable edge detection results as compared with the original frame-based Canny edge detector. From Figs. 17 and 18, it can be seen that, in contrast to the original frame-based Canny, the proposed distributed Canny algorithm can detect edges in low contrast regions since it is able to adapt to the local statistical distribution of gradient values in a block. Furthermore, Fig. 19 shows the edge maps of images with white Gaussian noise generated by the original frame-based Canny edge detector and the proposed distributed Canny edge detection algorithm with 3×3 gradient masks and a non-overlapping block size of 64×64. Comparing the edge maps in Fig. 19(b) and (c) with the clean edge maps in Figs. 15–18, it can be seen that, the proposed Canny edge detection algorithm is more robust to noise than the original frame-based Canny.

To further assess the performance of the proposed distributed Canny algorithm, quantitative conformance evaluations and subjective tests are performed. The conformance evaluations aim to evaluate the similarity between edges detected by the original frame-based Canny algorithm and the proposed distributed Canny edge detection algorithm, while the subjective tests aim to validate whether the edge detection performance of the proposed distributed Canny is better, worse, or similar to the original frame-based Canny as perceived by subjects.

1) Conformance Evaluation: In order to quantify the similarity of two edge maps, three metrics, P_{co} (percentage of edge pixels detected by both implementations) P_{nd} (percentage of edge pixels detected by the original Canny edge detection algorithm that were not detected by the proposed distributed Canny edge detection algorithm also referred to as false negatives) and P_{fa} (percentage of edge pixels detected by the proposed Canny edge detection algorithm that were not detected by the original Canny edge detection algorithm, also referred to as false positives), as proposed in [12], are used for the conformance test. Here, the edge map generated by the original frame-based Canny for the clean image is considered as the reference for both the clean and noisy cases. This is compared with the edge map generated by the proposed algorithm for the same clean image, as well as the edge maps generated by these two algorithms (original Canny and proposed) for the noisy version of the same image. Experiments were conducted using the USC SIPI Image Database [20] and the Standard Test Image Database [21]. The results are shown in Table III.

TABLE III
CONFORMANCE EVALUATION

	Clean Image	Noisy Image	
	Proposed Algorithm	Original Canny	Proposed Algorithm
P_{co}	94.8%,	56.2%	64.9%
P_{nd}	1.2%	13.7%	16.6%
P_{fa}	4%	30.1%	18.5%

For clean images, the detected edge pixel percentage (P_{co}) was higher than 94.8%. This further validates our claim that the proposed distributed algorithm detects almost all edges that are detected by the original Canny edge detection algorithm.

For noisy images, as shown in Table III, the P_{co} of the proposed distributed Canny algorithm is higher than that of the original Canny algorithm, and the error (the sum of P_{nd} and P_{fa}) of the proposed distributed Canny algorithm is much lower than the original Canny detector. This validates our earlier observation that, for noisy image, the edge detection performance of the proposed distributed Canny algorithm is better than the original frame-based Canny algorithm.

2) Subjective Testing: Subjective tests were conducted by having human subjects evaluate the quality of the detected edge maps that are generated by the proposed algorithm and the original Canny for both clean and noisy images, without the subjects knowing which algorithm produced which edge maps, using images from the SIPI Database [20] and the Standard Test Image Database [21].

The edge maps of clean images obtained by the original Canny algorithm and the proposed algorithm were displayed simultaneously for comparison together with the original image, which was displayed in the middle. Fig. 20 shows a snapshot of the subjective test interface. This was done for both clean and noisy images. Experiments were conducted using a 19-inch DELL LCD monitor having a resolution of 1024×1280. The average room luminance is measured to be 30 cd/m^2. The subjects were asked to rate the quality of the displayed edge maps using a five-point scale with scores 1 to 5 corresponding, respectively, to the rightmost edge map is "worse," "slightly worse," "same," "slightly better," and "better" compared to the edge map shown on the left. These scores were then automatically converted internally

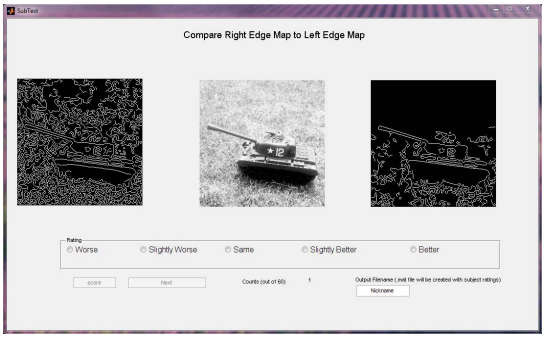


Fig. 20. Snapshot of the performed subjective test comparing the edge maps generated by the original Canny algorithm and the proposed algorithm.

Fig. 21. Subjective scale where scores 1, 2, 3, 4, & 5 correspond to "worse," "slightly worse," "same," "slightly better," and "better," with respect to the original Canny algorithm.

by the interface into five-point scale scores from 1 to 5 corresponding, respectively, to the edge map produced by the proposed distributed Canny algorithm is "worse," "slightly worse," "same," "slightly better," and "better" compared to the original Canny algorithm (see Fig. 21). Thus, a score higher than 3 indicates that the generated edge map by the proposed algorithm is better than the one obtained using the original Canny algorithm and vice versa. The images are randomly displayed. Each case is randomly repeated four times, with the left and right images swapped in a random fashion, to obtain better subjective response statistics [22]. Ten subjects with normal to corrected-to-normal vision took the test. Two Mean Opinion Scores (MOS) were computed by averaging the subjective scores over the set of clean images and over the set of noisy images, respectively. The obtained MOS values were 3.69 (slightly better than the original Canny) for the clean image set and 4.6 (better than the original Canny) for the noisy image set. Results from the subjective tests show clear preference of the subjects for the edge maps that are generated by the proposed algorithm for both clean and noisy images. These results indicate that the proposed algorithm results in a superior performance as compared to the original frame-based Canny algorithm. This is partly due to the capability of the proposed algorithm to adapt to the local image characteristics since the P_1 value is selected differently for different types of blocks by adapting its value to the local block content.

V. FPGA Implementation of the Proposed Distributed Canny Edge Detection Algorithm

In order to demonstrate the parallel efficiency of the proposed distributed Canny edge detection algorithm, we describe an FPGA-based hardware implementation of the proposed algorithm in this section. Fig. 22 gives a bird's eye view of the embedded system for implementing the distributed Canny edge detection algorithm based on an FPGA platform. It is composed of several components, including an embedded micro-controller, a system bus, peripherals & peripheral controllers,

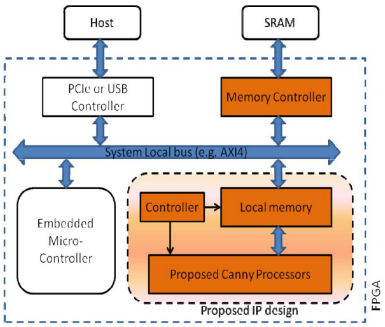


Fig. 22. Block diagram of the embedded system for the proposed algorithm.

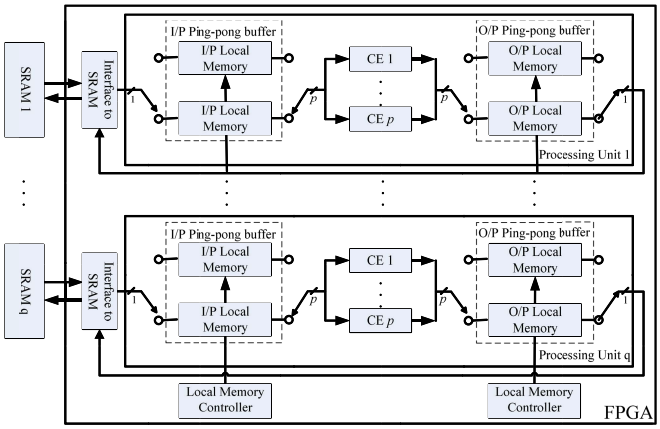


Fig. 23. The architecture of the proposed distributed Canny algorithm.

external Static RAMs (SRAM) & memory controllers, and an intellectual property (IP) design for the proposed distributed Canny detection algorithm. The embedded micro-controller coordinates the transfer of the image data from the host computer (through the PCIe (or USB) controller, system local bus, and memory controller) to the SRAM; then from the SRAM to the local memory in the FGPA for processing and finally storing back to the SRAM. Xilinx and Altera offer extensive libraries of intellectual property (IP) in the form of embedded micro-controllers and peripherals controller [23], [24]. Therefore, in our design, we focused only on the implementation of the proposed algorithm on the Xilinx Virtex-5 FPGA and the data communication with external SRAMs. These components have been highlighted in Fig. 22. In the rest of this section, we provide a high-level overview of the proposed IP design (Section 5.1) followed by details of each of the units in the computing engine (Sections 5.2). The FPGA synthesis results are presented in Section 6.

A. Architecture Overview

The proposed architecture, shown in Fig. 23, consists of q processing units (PU) and external dual-port Static RAMs (SRAMs). In Fig. 23, each PU consists of p computing engines (CE), where each CE processes an $m \times m$ overlapping image block and generates the edges of an $n \times n$ block, where $m = n + L + 1$ for an $L \times L$ gradient mask. The dataflow through this architecture is as follows. For each PU, the SRAM controller fetches the input data from SRAM and stores them

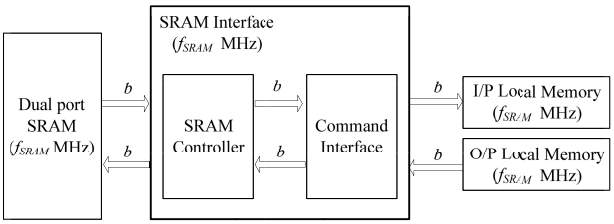


Fig. 24. High throughput memory interface.

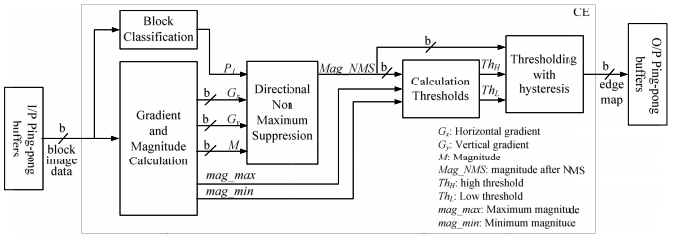


Fig. 25. Block diagram of CE (computing engine for edge detection).

into the input local memory in the PU. The CEs read this data, process them and store the edges into the output local memory. Finally, the edges are written back to the SRAM one output value at a time from the output local memory.

In order to increase the throughput, the SRAM external memory is organized into q memory banks, one bank per PU. Since only one b-bit data, corresponding to one pixel value, can be read from a SRAM at a time, such an organization helps multiple PUs to fetch data at the same time and facilitates parallel processing by the PUs. For an image of size $N \times N$, each SRAM bank stores a tile of size N^2/q image data, where the term "tile" refers to an image partition containing several non-overlapping blocks. The SRAM bank is dual-port so that the PUs can read and write at the same time.

In order to maximize the overlap between data read/write and data processing, the local memory in each PU is implemented using dual port block RAM (BRAM) based ping-pong buffers. Furthermore, in order that all the p CEs can access data at the same time, the local memory is organized into p banks. In this way, a total of $p{\cdot}q$ overlapping blocks can be processed by q groups of p CEs at the same time. The processing time for an $N \times N$ image is thus reduced approximately by a factor of $p{\cdot}q$. If there are enough hardware resources to support more CEs and more PUs, the throughput would increase proportionally.

However, FPGAs are constrained by the size of on-chip BRAM memory, number of slices, number of I/O pins; and the maximum throughput achievable is a function of these parameters. Since each CE processes an $m \times m$ overlapping image block, for a b-bit implementation, this would require $3 \times m \times m \times b$ bits to store the vertical and horizontal gradient components and the gradient magnitude, as discussed in Section 5.2. To enable parallel access of these blocks, there are three BRAMs of size $m \times m \times b$ in each CE. In addition, $2p \times m \times m \times b$ bits are needed for each of the ping-pong buffers. Therefore, for each PU, $3p \times m \times m \times b$ bits are need for p CEs and $4p \times m \times m \times b$ bits are needed for the input and output ping-pong buffers. This results in a total of $7p \times m \times m \times b \times q = 7pqm^2b$ bits for the FPGA memory. Thus, if there are more CEs and/or larger sized block, more FPGA memory is required. Similarly, if there are more PUs, more I/O pins are required to communicate with the external SRAM memory banks. Thus, the choice of p and q depends on the FPGA memory resources and numbers of I/O pins. We do not consider the numbers of slices as a constraint since the number of available slices is much larger than required by the proposed algorithm.

The customized memory interface, shown in Fig. 24, has a $2b$-bit wide internal data-bus. In our application, the dual-port SRAM, the memory interface and the local memories, which connect with the SRAM interface, operate at the same frequency, which is f_{SRAM} MHz.

B. Computing Engine (CE)

As described before, each CE processes an $m \times m$ overlapping image block and generates the edges of an $n \times n$ non-overlapping block. The computations that take place in CE can be broken down into the following five units: 1) block classification, 2) vertical and horizontal gradient calculation as well as magnitude calculation, 3) directional non-maximum suppression, 4) high and low threshold calculation, and 5) thresholding with hysteresis. Each of these units is mapped onto a hardware unit as shown in Fig. 25 and described in the following subsections. The communication between each component is also illustrated in Fig. 25 and will be described in detail in the following subsections.

Suppose the input image data has been stored in the external memory (SRAMs). For each PU, once the ping-pong buffers have loaded p $m \times m$ overlapping image blocks, which we refer to as a Group of Blocks (GOB), from the SRAM, all the p CEs can access block data from ping-pong buffers at the same time. For each CE, the edge detection computation can start after n $m \times m$ overlapping block is stored in CE's local memories. In addition, in order to compute the block type, vertical gradient and horizontal gradient in parallel, the $m \times m$ overlapping block is stored in three local memories, marked as local memory 1, 2 and 3 as shown in Figs. 26 and 27.

1) Block Classification: As stated before, the $m \times m$ overlapping block is stored in the CE's local memory 1 and is used for determining the block type. The architecture for the block classification unit consists of two stages as shown in Fig. 26. Stage 1 performs pixel classification while stage 2 performs block classification. For pixel classification, the local variance of each pixel is utilized and the variance is calculated as follows:

$$\text{var} = \frac{1}{8}\sum_{i=1}^{9}(x_i - \bar{x})^2 \tag{5}$$

where x_i is the pixel intensity and $\bar{\boldsymbol{x}}$ is the mean value of the 3×3 local neighborhood. Thus, the pixels in the 3×3 windows are fetched from the local memory and stored in one FIFO buffer to compute the local variance. The computation

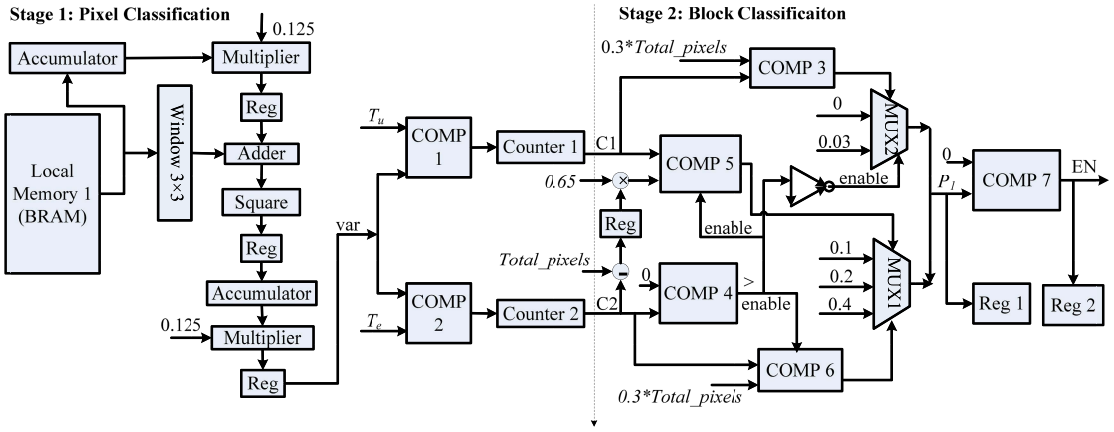


Fig. 26. The architecture of the block classification unit.

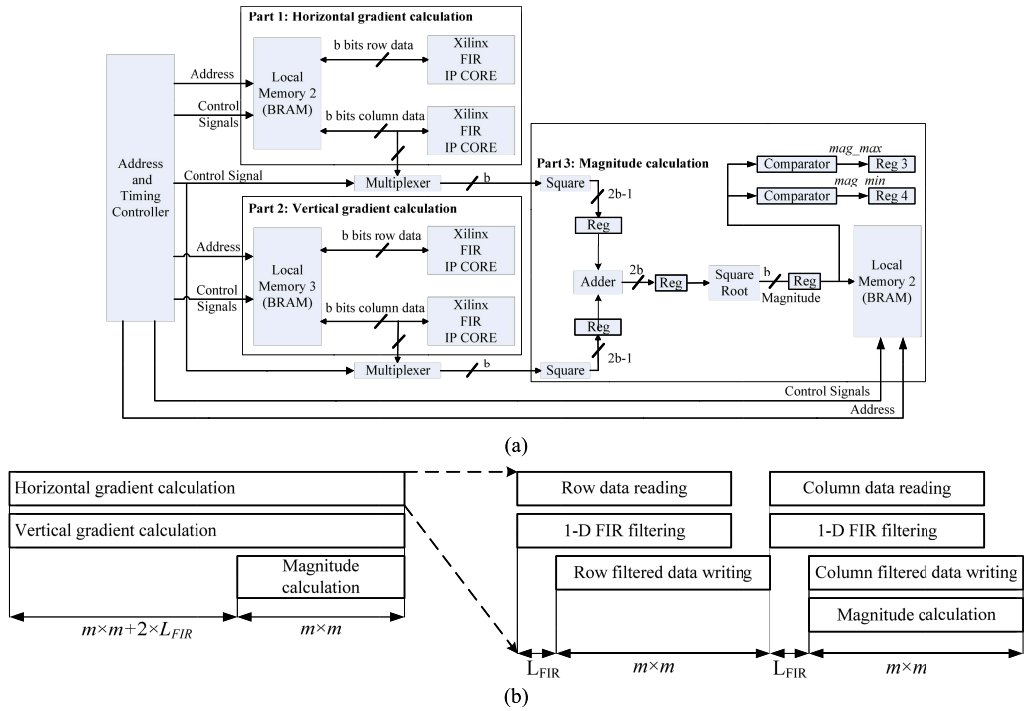


Fig. 27. Gradient and Magnitude Calculation Unit: (a) architecture; (b) executing time.

is done using one adder, two accumulators, two multipliers and one square (right shift 3 bits to achieve multiplication by 1/8). Next, the local variance is compared with T_u and T_e [17] in order to determine pixel type. Then two counters are used to get the total number of pixels for each pixel type. The output of counter 1 gives C1, the number of uniform pixels, while the output of counter 2 gives C2, the number of edge pixels. The block classification stage is initialized once the C1 and C2 values are available. C2 is compared with 0 and the result is used as the enable signal of COMP 5, COMP 6 and MUX 2. C1 and C2 are compared with different values as shown in Fig. 10(a), and the outputs are used as the control signals of MUX 1 and MUX 2 to determine the value of P_1. Finally, the P_1 value is compared with 0 to produce the enable signal, marked as EN. If the P_1 value is larger then 0, then EN signal enables gradient calculation, magnitude calculation, directional non-maximum suppression, high and low threshold calculation and thresholding with hysteresis units. Otherwise, these units do not need to be activated and the edge map with all zero value pixels is stored back into the SRAM. The P_1 and EN are the outputs for the block classification unit and are stored in the registers for thresholding calculation. The latency between the first input and the output P_1 is $m \times m$ +12 clock cycles and the total execution time for the block classification component is $m \times m$ +13.

2) Gradient and Magnitude Calculation: Since the gradient and magnitude calculation unit is independent of the block classification unit, these two components can work in parallel. In addition, the horizontal gradient and vertical gradient can also be computed in parallel. The $m \times m$ overlapping block is also stored in the CE's local memory 2 and 3 and is used as input to compute the horizontal and vertical gradient, respectively. The architecture for the gradient and magnitude

calculation unit is shown in Fig. 27(a). It consists of three computation parts (horizontal gradient calculation, vertical gradient calculation and magnitude calculation) and one address and time controller, which provides the address and control signals to coordinate the computation. The execution timing of this unit is shown in Fig. 27(b).

For the horizontal and vertical gradient calculation, the input block image is convolved with the 2D horizontal and vertical gradient masks, respectively; the size of these masks can be chosen based on the features of the image. These 2D gradient masks are separable, so the implementation of 2D convolution is achieved by two 1D convolutions. To implement the 1D convolution, we adopt Xilinx's FIR IP core, which provides a highly parameterizable, area-efficient implementation that utilizes the symmetry characteristics of the coefficient set [25]. Thus, each 2D convolution operation is implemented with two 1D FIR cores, resulting in a total of four 1-D FIR cores for both the horizontal and vertical gradient computations. The results of the rowwise 1-D convolutions for the horizontal and vertical gradient computations are stored back into local memory 2 and 3, respectively, and are used as input for the columnwise 1D convolutions. Since the row and column access times are the same for BRAM, the row-filtered block image data does not need to be transposed before applying the column filter. The FIR filter outputs are the input to the magnitude calculation unit which computes the magnitude of the gradient from the horizontal and vertical gradients for each pixel. The magnitude computation consists of two multipliers and one square-root computation module. The square-root function is implemented using the CORDIC 4.0 IP core of the Xilinx core generator with the optimized pipeline mode. In addition, the unsigned fraction data format was used and the round mode was chosen as truncate. Finally, the gradient magnitude is stored back into local memory 1 for further computation. The maximum and minimum values of the magnitude, marked as $mag\_max\ and\ mag\_min$, are the outputs of this unit and used as the input of the threshold calculation unit. The execution timing for this unit is illustrated in Fig. 27(b). Here, L_{FIR} is the latency of Xilinx FIR IP core. A Mealy Finite State Machine (FSM) is used in the timing controller module since it requires less number of states and is faster than a Moore FSM.

3) Directional Non Maximum Suppression (NMS): The horizontal and vertical gradient and the gradient magnitude are fetched from local memory 2, 3 and 1, respectively; and used as input to the directional NMS unit. Fig. 28(a) and (b) show the mechanism and the architecture of the directional NMS, respectively. As described in Section 2, this step involves computing the gradient direction at each pixel. According to the sign and value of the horizontal gradient G_x and the vertical gradient G_y, the direction can be determined; and thus the gradient magnitudes of four nearest neighbors along the direction are selected to compute two intermediate gradients M_1 and M_2 as shown in Fig. 28(a). The corresponding architecture is shown in Fig. 28(b). The horizontal gradient component G_x and the vertical gradient component G_y control the selector which forwards the magnitude [marked as $M(x, y)$ in Fig. 28(b)] of neighbors

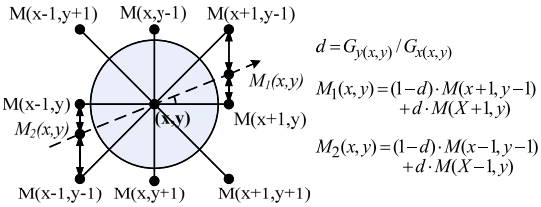


Fig. 28. Directional non maximum suppression unit: (a) mechanism; (b) architecture.

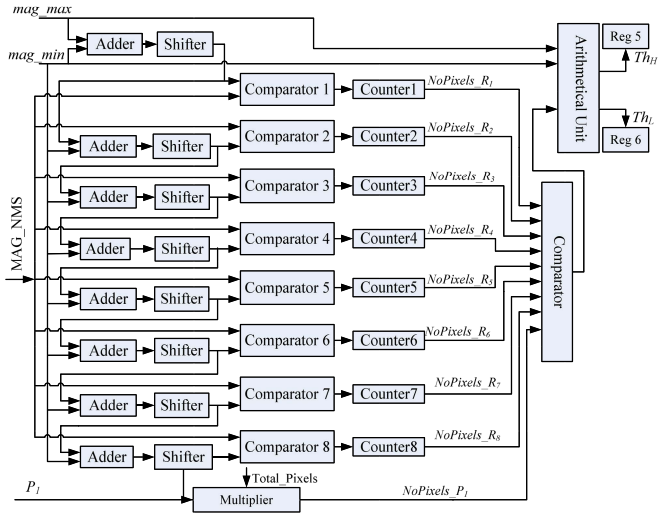


Fig. 29. The architecture of thresholds calculation unit.

along the direction of the gradient into the arithmetic unit. This arithmetic unit consists of one divider, two multipliers and one adder, which are also implemented by the Xilinx Math Functions IP cores. Finally, the output of the arithmetic unit is compared with the gradient magnitude of the center pixel. The gradient of the pixel that does not correspond to a local maximum gradient magnitude is set to zero. The final gradient magnitude after directional NMS [marked as $Mag\_NMS(x, y)$ in Fig. 28(b)] is stored back into local memory 1 as the input for the hysteresis thresholding unit. The latency between the first input and the first output is 20 clock cycles and the total execution time is $m \times m + 20$.

4) Calculation of Thresholds: Since the gradient magnitude of each pixel after directional NMS is used as the input for the calculation of thresholds, this unit can be pipelined with the directional NMS unit. Beside, the P_I value, which is determined by the block classification unit,

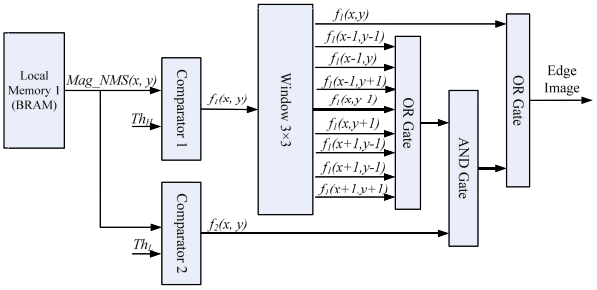


Fig. 30. Pipelined architecture of thresholding with hysteresis.

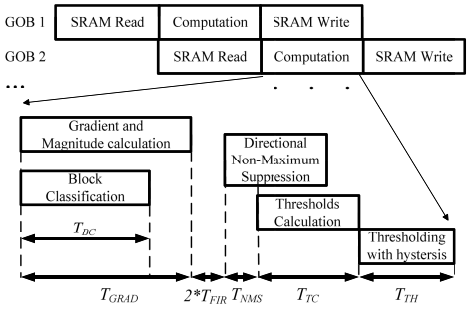


Fig. 31. Execution time of the 512 × 512 image.

Fig. 32. Comparison between the fixed-point Matlab software simulation result and FPGA hardware simulation result for detecting edges of the 512 × 512 Houses image using a block size of 64 × 64 and a 3 × 3 gradient mask: (a) edge map generated by the fixed-point MATLAB implementation; (b) edge map generated by the FPGA implementation.

the mag_max, *and* mag_min, which are determined by the gradient and magnitude calculation unit, are the inputs for this unit. As discussed in Section 3, the 8-step non-uniform quantizer is employed to obtain the discrete CDF; and the high and low thresholds are calculated based on this discrete CDF. The corresponding architecture is shown in Fig. 29. The adders and shifters are used to obtain the reconstruction levels R_i, and the comparators (1 to 8) and counters (1 to 8) are used to count the corresponding numbers of pixels, noted by $NoPixels\_R_i$, which have a gradient magnitude equal to or less than each reconstruction level R_i. The P_I value, which is determined by the block classification unit, is multiplied with the total number of pixels in the block. The obtained pixel number, noted by $NoPixels\_P_I$, is compared with each $NoPixels\_R_i$ in order to select the level i, where $NoPixels\_R_i$ is closest to the $NoPixels\_P_I$. According to the selected level i, mag_max, and mag_min, the arithmetical unit can compute the corresponding R_i, which is the high threshold Th_H. Finally, the low threshold Th_L is computed as 40% of the high threshold. The latency of the critical path is $m \times m + 6$ clock cycles.

5) Thresholding With Hysteresis: The gradient magnitude of each pixel after directional NMS is fetched from local memory 1 and used as input to the thresholding unit. Meanwhile, the Th_H and Th_L, which are determined by the threshold calculation unit, are also the inputs for this unit. Fig. 30 illustrates the pipelined design of this stage, where f_1 represents a strong edge pixel while f_2 represents a weak edge pixel. If any of the neighbors of the current pixel is a strong edge pixel, the center weak edge pixel is then considered as a strong edge pixel; otherwise, it is considered as a background non-edge pixel. The latency between the first input and the first output is 10 cycles and the total execution time for the hysteresis thresholding unit is $m \times m$ +10 cycles.

VI. Synthesis Results

The proposed FPGA-based architecture can support multiple image sizes and block sizes. To demonstrate the performance of the proposed system, a Xilinx Virtex-5 FPGA [26] was used to process grayscale images with a block size of 64 × 64. The data width is 16 bits (Q8.7) with 8 bits to represent the integer part since the maximum gray value of the image data is 255, and 7 bits to represent the fractional part since the Gaussian filter parameters are decimals. Our analysis shows that 7 bits are sufficient to meet the accuracy requirement of the Gaussian filter parameters, which is typically in the order of 0.001.

To store grayscale images, we used the SRAM (CY7C0832BV) [27]. This is a dual ported SRAM with 110 pins. The Xilinx Virtex-5 FPGA (XC5VSX240T) has 960 I/O pins and so, to satisfy the I/O pin constraint, the maximum number of PUs is 8 ($q = 8$). The local memory on the FPGA for a block size of 64 × 64, which is needed to support 8 PUs, is equal to $7pqm^2b = 4046p$ Kbits (see Section 5.1), for $q = 8$, $m = 68$ (for a 64 × 64 block size and a 3 × 3 gradient mask size), and $b = 16$. Since the available memory resource on the FPGA is 18,576 Kbits, the p value using the memory constraint is determined to be 4. The p value could have also been constrained by the number of available slices. Since the number of slices for the considered FPGA is very large (37 440) and since each CE only utilizes a small slice percentage (as shown later in Section 5.3.2), the local memory resource in each PU constrains p, the number of CEs in each PU, and not the numbers of slices. Taking all this into consideration, our design has $q = 8$ PUs and each PU has $p = 4$ CEs. This design is coded in Verilog and synthesized on a Xilinx Virtex-5 device (XC5VSX240T) using the Xilinx's ISE software and verified using Modelsim. According to the 'Place and Route' synthesis report, our implementation can achieve an operating frequency of 250 MHz. But we choose 100 MHz to support a pipelined implementation of SRAM read/write and CE processing as described later in Section 6.2.

A. Resource Utilization

The FPGA resource utilization in each CE is listed in Table IV, followed by the resource utilization for each PU in Table V. The FIR IP core and the arithmetic functions are

TABLE IV
RESOURCE UTILIZATION ON XC5VSX240T FOR 1 CE

Block Size	Number of CE	Occupied slices	Slice Reg.	Slice LUTs	DSP48Es	Total used Memory
64×64	1	747 (2%)	1,270 (1%)	2,578 (2%)	7 (1%)	217(KB) (1%)

TABLE V
RESOURCE UTILIZATION ON XC5VSX240T FOR 1 PU

Block Size	Number of CE	Occupied slices	Slice Reg.	Slice LUTs	DSP48Es	Total used Memory
64×64	4	2,988 (8%)	5,080 (4%)	10,312 (8%)	28 (3%)	2023(KB) (10%)

TABLE VI
RESOURCE UTILIZATION ON XC5VSX240T FOR AN 8-PU ARCHITECTURE

Block Size	Number of CE	Occupied slices	Slice Reg.	Slice LUTs	DSP48 Es	Total used Memory
64×64	32	23,904 (64%)	40,640 (32%)	82,496 (65%)	224 (25%)	16,184 (KB) (87%)

TABLE VII
CLOCK CYCLES FOR EACH UNIT

	T_{GRAD}	T_{FIR}	T_{NMS}	T_{TC}	T_{TH}
Clock Cycles	9248	16	20	4630	4634

implemented using DSP48Es. The on-chip memory is implemented using BRAMs. Table VI summarizes the resource utilization of the 8-PU architecture. It shows that the 8-PU architecture occupies 64% of the slices and 87% of the BRAM memory.

B. Execution Time

Fig. 31 shows the pipeline implementation of SRAM read/write with the CE computation, where each SRAM bank stores a tile of size N^2/q image data and each ping or pong buffer stores a group of blocks (GOB) of size $p \times m \times m$ image data. Since our design has $q = 8$ PUs, one SRAM can hold a tile of size 32,768 ($64 \times 64 \times 8$) image data for a 512×512 image. In addition, for $p = 4$ CEs and for a 64×64 block size ($n = 64$; $m = 68$), the image data stored in the SRAM result in two GOBs. These GOBs can be pipelined. As shown in Fig. 31, while GOB 2 is loaded to the ping-pong buffers, the CEs process GOB 1. Also, while GOB 1 is written back into SRAM, the CEs process GOB 2 at the same time. Such a pipelined design can increase throughput.

Fig. 31 also shows the computation time of each stage in a CE during the processing of an $m \times m$ overlapping block ($m = 68$ for a 64×64 block and a 3×3 gradient mask). As shown in Fig. 31, T_{BC}, the time to classify the block type, is less than T_{GRAD}, the time for Gradient and Magnitude calculation, which equals to 9248 clock cycles. T_{FIR}, the FIR filter computation latency equals to 8 clock cycles for a 3×3 FIR separable filter. The high and low thresholds calculation unit is pipelined with the directional NMS unit and the latency of the NMS unit is 20 clock cycles. This is referred to as T_{NMS} in Fig. 31. T_{TC} represents the latency of the thresholds calculation stage and is equal to 4630 cycles, while T_{TH} represents the latency of the thresholding with hysteresis stage and is equal to 4634 cycles. Table VII shows the latency for each unit. Therefore, one CE takes $T_{CE} = T_{GRAD} + T_{FIR} + T_{NMS} + T_{TC} + T_{TH} = 18548$ cycles.

Each PU takes 18,496 cycles to load 4 68×68 overlapping blocks from the SRAM into the local memory. It also takes 16,384 (4 64×64 non-overlapping blocks) cycles to write final edge maps into SRAM. If SRAM operates at f_{SRAM}, the SRAM read time is $18{,}496/f_{SRAM}$. The CE processing time equals to $18{,}548/f_{CE}$ when CE is clocked at f_{CE}. In order to completely overlap communication with computation and avoid any loss of performance due to communication with SRAM, given a fixed f_{SRAM}, f_{CE} should be selected such that the processing time is approximately equal to the SRAM read time (since the SRAM write time is less than the read time). Thus, $18{,}496/f_{SRAM} = 18{,}548/f_{CE}$, and the f_{CE} can be set to be 1.003 times higher than f_{SRAM}.

The maximum speed of the employed SRAM device (CY7C0832BV) is 133MHz. However, we choose the SRAM clock rate as $f_{SRAM} = 100$ MHz to allow for sufficient design margin. Thus, $f_{CE} \approx 100$ *MHz*, which is lower than the maximum operating frequency (250 MHz) of the used FPGA according to the synthesis report. The total computation period for one CE is $T_{CE} = 18,\ 548/10^5 \approx 0.186$ ms when clocked at 100 MHz. Thus, for a 512×512 image, the total computation period is $T_{com} = 0.372$ms; while the total execution time, including the SRAM read/write time and the computation time, is $T_{total} = (18,496 + 16,384)/1E^5 + 0.186 \times 2 = 0.721$ ms. The simulation results also show that, at a clock frequency of 100MHz, the execution time for processing 512×512 images is 0.721ms for the images in the USC SIPI database.

C. FPGA Experiments and Results

1) Conformance Tests: In order to validate the FPGA generated results, two conformance tests are performed. One aims to evaluate the similarity between edges detected by the fixed-point FPGA and Matlab implementation of the distributed Canny edge detection algorithm. The other is to measure the similarity between edges detected by the fixed-point FPGA and the 64-bit floating-point Matlab implementation of the distributed Canny edge detection algorithm. Our results for both tests were performed on the USC SIPI image database.

For the first test, the difference between the edges detected by the fixed-point FPGA and Matlab implementations is calculated. Fig. 32 shows an example of the obtained fixed-point Matlab and FPGA results for the Houses image using the proposed algorithm with a 64×64 block size and 3×3 gradient masks. The FPGA simulation result is obtained using Modelsim and assumes the original image data has been stored in SRAMs. It can be seen that the two edge maps are the same. Furthermore, the quantitative difference between the FPGA simulation result and the fixed-point Matlab simulation result

TABLE VIII
RUNTIME COMPARISON ON VARIOUS PLATFORMS FOR "LENA" IMAGE

Image Size	Frame-based Canny		Proposed distributed Canny (64 ×64 block)
	OpenCV (ms) @ 3.33 GHz	FPGA (ms) @ 100 MHz	FPGA (ms) @ 100 MHz
256×256	2.206	0.67	0.535
512×512	5.97	2.74	0.721
1024×1024	20.852	12.02	1.837
2048×2048	76.102	56.30	6.301

is zero. The same results were obtained for all 26 images in dataset.

In order to verify similarity of the edges generated by the fixed-point FPGA and 64-bit floating-point Matlab implementation of our proposed algorithm, the P_{co}, P_{nd} and P_{fa}, metrics as described in Section 4.2.1, were used. For all images, the correctly detected edge pixel percentage (P_{co}) was higher than 98.2%. Furthermore, less than 1.8% of all edge pixels were detected by only one of the detectors (P_{nd} and P_{fa}). This leads to the conclusion that the hardware implementation of our proposed algorithm can successfully detect significant edges.

2) Performance Test: Furthermore, to show the efficiency of the FPGA-based system, we compared the FPGA implementation of the original frame-based Canny edge detector with an assembly optimized CPU implementation of the original Canny edge detector available in Intel's OpenCV library, which makes good use of thread parallelization and SSE instructions. Also, to show the efficiency of the FPGA implementation of our proposed distributed Canny algorithm, the proposed FPGA distributed Canny implementation is compared with the FPGA and CPU implementations of the original frame-based Canny edge detector. The two candidate hardware platforms were:

- CPU: Intel Core i7-975, four 3.33G cores, 8192KB cache and 12GB RAM;
- FPGA: Xilinx SXT Virtex-5, 37,440 slices, 1,056 DSP48E slices and 18,576 Kbits BRAM.

Runtime results are shown in Table VIII for the standard "Lena" image. The absolute runtimes of the two algorithms were recorded and averaged over 100 times. Since the FPGA runtime does not include the image loading time from the host to the external SRAMs, for fair comparison, the loading time from the HDD to the RAM for the CPU implementation is also not included in the reported runtime. In general, the CPU loading time is shorter than the FPGA loading time since the speed of the internal CPU bus is faster than when using an external bus such as PCI or PCI external.

From Table VIII, it can be seen that the speed of the FPGA implementation for the original frame-based Canny edge detection is faster than the frame-based CPU implementation even though, for the CPU implementation, the CPU is clocked 33 times faster. Furthermore, the FPGA implementation for the proposed distributed Canny edge detection is much faster than the frame-based FPGA and CPU implementations.

TABLE IX
COMPUTATION TIME OF THE PROPOSED DISTRIBUTED CANNY FPGA IMPLEMENTATION USING A NON-OVERLAPING BLOCK SIZE OF 64×64 AND A 3×3 GRADIENT MASK AND OF EXISTING FRAME-BASED CANNY FPGA IMPLEMENTATIONS

	Image Size	FPGA Device	Used Slices	Used MEM. (KB)	Freq (MHz)	Time (ms)	Norm. Time (ms)
[4]	256× 256	Xilinx Vertex-E	□	□	16	4.2	2.688
[5]	256×	Altera	1,530/	1,116/	264	0.25	2.64
	256	Stratix II	48,352	2,044			
[7]	360× 280	Altera Cyclone	□	□	27	2.5	0.72
[6]	512× 512	Xilinx Virtex-5	4,553/ 71,680	192/ 5,328	292.8	0.57	1.669
Our	512× 512	Xilinx Virtex-5	23,904/ 37,440	16,184/ 18,576	250	0.15	0.372

TABLE X
RUNTIMES OF THE GPGPU AND OUR 32-CE FPGA IMPLEMENTATIONS

	GTX 80 [10]	GT 80 [12]	GT 200 [12]	Fermi [12]	Our
Image Size	512×512	321×481	321×481	321×481	512×512
Freq (MHz)	768	1500	1300	1150	100
Time (ms)	3.4	5.47	2.95	2.3	0.721
Norm. Time(ms)	26.112	139.31	65.11	44.91	0.721

This illustrates the efficiency of the proposed distributed Canny edge detector.

3) Comparison With Other Existing GPU and FPGA Implementations: A long standing challenge for the use of the Canny edge detection algorithm for real-time applications is its high computational complexity. Several FPGA and GPU implementations have been proposed to meet the requirements of real-time applications. Comparisons between our proposed FPGA implementation and other existing FPGA and GPU implementations are discussed below.

As far as we know, the existing FPGA-based approaches implemented the frame-based Canny algorithm and only the computation time for detecting edges are presented in papers [4]–[6]. The computation time and resource utilization results of these frame-based Canny FPGA implementations and our distributed Canny FPGA implementation are shown in Table IX. Since these approaches are implemented for different image sizes and operated at different clock frequencies, their results have to be normalized for a fair comparison. The rightmost column of Table IX gives the normalized computing time, where the normalization is done with respect to an image of size 512×512 and an FPGA clock frequency of 100 MHz. We see that the computation time of the proposed distributed implementation is much faster than the existing implementations. Furthermore, the implementations [5]–[7] result in degraded performance compared to our implementation since the thresholds are fixed to predetermined values, while our implementation computes the hysteresis thresholds adaptively based on the input image characteristics. In addition, compared to other frame-based FPGA implementations, the proposed implementation enables a higher throughput due to its

distributed nature, at the expense of more memory and slice usage.

Also, the GPU-based Canny implementations in [10] and [12] are compared to our 32-CE FPGA Canny implementation. One NVidia GPGPU was used in [10] and three NVidia GPGPUs were used in the recent GPU Canny implementation [12]. The GPU configuration details are given below:

- GTX 80 [10]: NVidia GeForce 8800 GTX with 128 575-MHz cores and 768MB RAM.
- GT 80 [12]: NVidia GeForce 8800 GT with 112 1.5-GHz cores and 512MB RAM.
- GT 200 [12]: NVidia Tesla C1060 with 240 1.3-GHz cores and 4GB RAM;
- Fermi [12]: NVidia Tesla C2050 with 448 1.15-GHz cores and 3GB RAM.

Since the FPGA runtime does not include the image loading time from the host to the external SRAMs, the loading time for GPU implementations was removed from the total runtime for fair computation. Table X illustrates the runtimes of the GPGPU implementations and our 32-CE FPGA implementation (with a 64×64 block size and a 3×3 gradient mask) for processing a 512×512 image. Even though all these GPGPU implementations do not have the adaptive threshold calculation stage, our 32-CE FPGA implementation is faster than the GPGPU implementations even though the FPGA is operated at a much slower frequency.

VII. Conclusion

The original Canny algorithm relies on frame-level statistics to predict the high and low thresholds and thus has latency proportional to the frame size. In order to reduce the large latency and meet real-time requirements, we presented a novel distributed Canny edge detection algorithm which has the ability to compute edges of multiple blocks at the same time. To support this, an adaptive threshold selection method is proposed that predicts the high and low thresholds of the entire image while only processing the pixels of an individual block. This results in three benefits: 1) a significant reduction in the latency; 2) better edge detection performance; 3) the possibility of pipelining the Canny edge detector with other block-based image codecs. In addition, a low complexity non-uniform quantized histogram calculation method is proposed to compute the block hysteresis thresholds. The proposed algorithm is scalable and has very high detection performance. We show that our algorithm can detect all psycho-visually important edges in the image for various block sizes. Finally, the algorithm is mapped onto a Xilinx Virtex-5 FPGA platform and tested using ModelSim. The synthesized results show 64% slice utilization and 87% BRAM memory utilization. The proposed FPGA implementation takes only 0.721ms (including the SRAM read/write time and the computation time) to detect edges of 512×512 images in the USC SIPI database when clocked at 100 MHz. Thus the proposed implementation is capable of supporting fast real-time edge detection of images and videos including those with full-HD content.

References

[1] R. Deriche, "Using canny criteria to derive a recursively implemented optimal edge detector," *Int. J. Comput. Vis.*, vol. 1, no. 2, pp. 167–187, 1987.

[2] L. Torres, M. Robert, E. Bourennane, and M. Paindavoine, "Implementation of a recursive real time edge detector using retiming technique," in *Proc. Asia South Pacific IFIP Int. Conf. Very Large Scale Integr.*, 1995, pp. 811–816.

[3] F. G. Lorca, L. Kessal, and D. Demigny, "Efficient ASIC and FPGA implementation of IIR filters for real time edge detection," in *Proc. IEEE ICIP*, vol. 2. Oct. 1997, pp. 406–409.

[4] D. V. Rao and M. Venkatesan, "An efficient reconfigurable architecture and implementation of edge detection algorithm using handle-C," in *Proc. IEEE Conf. ITCC*, vol. 2. Apr. 2004, pp. 843–847.

[5] H. Neoh and A. Hazanchuck, "Adaptive edge detection for real-time video processing using FPGAs," Altera Corp., San Jose, CA, USA, Application Note, 2005.

[6] C. Gentsos, C. Sotiropoulou, S. Nikolaidis, and N. Vassiliadis, "Real-time canny edge detection parallel implementation for FPGAs," in *Proc. IEEE ICECS*, Dec. 2010, pp. 499–502.

[7] W. He and K. Yuan, "An improved canny edge detector and its realization on FPGA," in *Proc. IEEE 7th WCICA*, Jun. 2008, pp. 6561–6564.

[8] I. K. Park, N. Singhal, M. H. Lee, S. Cho, and C. W. Kim, "Design and performance evaluation of image processing algorithms on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 91–104, Jan. 2011.

[9] J. D. Owens *et al.*, "A survey of general-purpose computation on graphics hardware," *Comput. Graph. Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[10] Y. Luo and R. Duraiswami, "Canny edge detection on NVIDIA CUDA," in *Proc. IEEE CVPRW*, Jun. 2008, pp. 1–8.

[11] R. Palomar, J. M. Palomares, J. M. Castillo, J. Olivares, and J. Gómez-Luna, "Parallelizing and optimizing lip-canny using NVIDIA CUDA," in *Proc. IEA/AIE*, Berlin, Germany, 2010, pp. 389–398.

[12] L. H. A. Lourenco, "Efficient implementation of canny edge detection filter for ITK using CUDA," in *Proc. 13th Symp. Comput. Syst.*, 2012, pp. 33–40.

[13] Q. Xu, C. Chakrabarti, and L. J. Karam, "A distributed Canny edge detector and its implementation on FPGA," in *Proc. DSP/SPE)*, Jan. 2011, pp. 500–505.

[14] J. F. Canny, "A computation approach to edge detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 8, no. 6, pp. 769–798, Nov. 1986.

[15] S. Nercessian, "A new class of edge detection algorithms with performance measure," M.S. thesis, Dept. Electr. Eng., Tufts Univ., Medford, MA, USA, May 2009.

[16] P. Bao, L. Zhang, and X. Wu, "Canny edge detection enhancement by scale multiplication," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 9, pp. 1485–1490, Sep. 2005.

[17] J. K. Su and R. M. Mersereau, "Post-processing for artifact reduction in JPEG-compressed images," in *Proc. IEEE ICASSP*, vol. 3. May 1995, pp. 2363–2366.

[18] P. Arbelaez, C. Fowlkes, and D. Martin. (2013). *The Berkeley Segmentation Dataset and Benchmark* [Online]. Available: http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/

[19] N. D. Narvekar and L. J. Karam, "A no-reference image blur metric based on the cumulative probability of blur detection (CPBD)," *IEEE Trans. Image Process.*, vol. 20, no. 9, pp. 2678–2683, Sep. 2011.

[20] (2013). *USC SIPI Image Database* [Online]. Available: http://sipi.usc.edu/database/database.php?volume=misc

[21] (2013). *Standard Test Image Database* [Online]. Available: http://www.imageprocessingplace.com/

[22] L. V. Scharff, A. Hill, and A. J. Ahumada, "Discriminability measures for predicting readability of text on textured backgrounds," *Opt. Exp.*, vol. 6, no. 4, pp. 81–91, Feb. 2000.

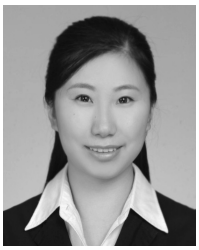
[23] (2013). *Altera Available IP* [Online]. Available: http://www.altera.com/products/ip/processors/nios2/features/ni2-peripherals.html

[24] (2013). *Xilinx Available IP* [Online]. Available: http://www.xilinx.com/ise/embedded/edk/_ip.htm

[25] (2007). *Xilinx IP LogiCore FIR Datasheet* [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/fir_compiler_ds534.pdf

[26] (2012). *Xilinx Vertrix-5 Family Datasheet* [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf

[27] (2012). *SRAM Datasheet* [Online]. Available: http://www.cypress.com/?docID=31696



Qian Xu received the B.E. degree in optoelectronics engineering from the Huazhong University of Science and Technology, Wuhan, China, and the M.E. degree in electrical engineering from the Shanghai Institute of Technical Physics, Chinese Academy of Sciences, China, in 2006 and 2009, respectively.

She is pursuing the Ph.D. degree in electrical engineering, specializing in image processing and analysis, with Arizona State University, Tempe. Her research interests include image processing, computer vision, pattern recognition, and algorithm-architecture codesign.

Srenivas Varadarajan received the B.E. degree in electronics and communications engineering from the PSG College of Technology, Coimbatore, India, and the M.S. degree in electrical engineering from Arizona State University, in 2003 and 2009, respectively, where he is currently pursuing the Ph.D. degree in electrical engineering in the area of image. His research interests include texture analysis and synthesis, image and video compression, computer vision, 3-D modeling, and embedded-software optimizations for media processing algorithms. He has about seven years of Industrial experience in image and video processing in companies, including Texas Instruments, Qualcomm Research, and Intel Corporation.

Chaitali Chakrabarti received the B.Tech. degree in electronics and electrical communication engineering from IIT Kharagpur, India, and the M.S. and Ph.D. degrees in electrical engineering from the University of Maryland, College Park, in 1984, 1986, and 1990, respectively. She is a Professor with the School of Electrical Computer and Energy Engineering, Arizona State University (ASU), Tempe. Her research interests include all aspects of low-power embedded systems design and VLSI architectures, and algorithms for signal processing, image processing, and communications.

Dr. Chakrabarti was the recipient of the Best Paper Awards at SAMOS'07, MICRO'08, SiPS'10, and HPCA'13. She is a Distinguished Alumni with the Department of Electrical and Computer Engineering, University of Maryland. She was the recipient of several teaching awards, including the Best Teacher Award from the College of Engineering and Applied Sciences from ASU in 1994, the Outstanding Educator Award from the IEEE Phoenix Section in 2001, and the Ira A. Fulton Schools of Engineering Top 5% Faculty Award in 2012. She served as the Technical Committee Chair of the DISPS subcommittee, IEEE Signal Processing Society, from 2006 to 2007. She is currently an Associate Editor of the *Journal of VLSI Signal Processing Systems* and the IEEE TRANSACTIONS OF VLSI SYSTEMS.

Lina J. Karam (F'13) received the B.E. degree in computer and communications engineering from the American University of Beirut, Beirut, Lebanon, and the M.S. and Ph.D. degrees in electrical engineering from the Georgia Institute of Technology, Atlanta, GA, USA, in 1989, 1992, and 1995, respectively.

She is a Full Professor with the School of Electrical, Computer and Energy Engineering, Arizona State University, Phoenix, AZ, USA, where she directs the Image, Video, and Usability (IVU) Research Laboratory. Her industrial experience includes image and video compression development at AT&T Bell Laboratories, Murray Hill, NJ, USA, multidimensional data processing and visualization at Schlumberger, and collaboration on computer vision, image/video processing, compression, and transmission projects with industries, including Intel, NTT, Motorola/Freescale, General Dynamics, and NASA. She has authored more than 100 technical publications, and she is a co-inventor on a number of patents.

Dr. Karam was the recipient of the U.S. National Science Foundation CAREER Award, the NASA Technical Innovation Award, the 2012 Intel Outstanding Researcher Award, and the Outstanding Faculty Award by the IEEE Phoenix Section in 2012. She has served on several journal editorial boards, several conference organization committees, and several IEEE technical committees. She served as the Technical Program Chair of the 2009 IEEE International Conference on Image Processing, the General Chair of the 2011 IEEE International DSP/SPE Workshops, and the Lead Guest Editor of the PROCEEDINGS OF THE IEEE, PERCEPTION-BASED MEDIA PROCESSING ISSUE (SEP. 2013). She has co-founded two international workshops (VPQM and QoMEX). She is currently serving as the General Chair of the 2016 IEEE International Conference on Image Processing and a member of the IEEE Signal Processing Society's Multimedia Signal Processing Technical Committee and the IEEE Circuits and Systems Society's DSP Technical Committee. She is a member of the Signal Processing, Circuits and Systems, and Communications societies of the IEEE.