Journal of Real-Time Image Processing (2022) 19:591–605
https://doi.org/10.1007/s11554-022-01208-0

ORIGINAL RESEARCH PAPER

# GUD-Canny: a real-time GPU-based unsupervised and distributed Canny edge detector

**Antonio Fuentes-Alventosa[1] · Juan Gómez-Luna[2] · R. Medina-Carnicer[1]**

Received: 15 September 2021 / Accepted: 13 February 2022 / Published online: 5 March 2022


**Abstract**
The Canny algorithm is one of the most commonly used edge detectors due to its superior performance, especially in noisy environments. Its main limitation is that it is time consuming due to its multistage nature and the use of complex computational operations, primarily hysteresis thresholding. For this reason, many efficient implementations of the Canny edge detector have been developed on different accelerating platforms, such as ASICs, FPGAs and GPUs. The two main limitations of the GPU implementations developed to date are the bottleneck caused by the hysteresis process, and the use of fixed hysteresis thresholds. To overcome these issues, a novel GPU-based unsupervised and distributed Canny edge detector is proposed in this paper. Experimental evaluation showed that our Canny edge detector fully satisfies real time requirements, as it only requires 0.35 ms on average to detect edges on 512×512 images, and that it is faster than existing GPU and FPGA implementations.

**Keywords** Edge detection · Canny edge detector · GPU · CUDA · Parallel implementations

## 1 Introduction

Edge detection is an essential operation in different fields, such as image processing, computer vision and pattern recognition. Over the years, many edge detection algorithms have been proposed, including classical approaches, such as Roberts [1], Sobel [2], Prewitt [3] and Canny [4] methods, as well as more recent methods based on soft computing techniques, such as fuzzy logic [5], Artificial Neural Networks [6], genetic algorithms [7], particle swarm optimization [8], ant colony optimization [9] and adaptive neuro fuzzy inference system [10].

The Canny algorithm [4], also known as optimal detection method, is still one of the most widely used edge detection techniques due to its superior performance. It consists of the following four stages: (1) noise reduction, (2) gradient computation, (3) non-maximum suppression, and (4) hysteresis thresholding. First, the image noise is reduced by a Gaussian convolution. Next, first derivatives are calculated in both horizontal ($d_x$) and vertical dimensions ($d_y$). From these two images, the gradient magnitude ($G$) and direction ($\theta$) are computed for each pixel by the formulas $G = \sqrt{d_x^2 + d_y^2}$ and $\theta = tan^{-1}(\frac{d_y}{d_x})$. In the third stage, possible edges are obtained by suppressing all pixels which are not local maximums in the gradient direction. In the last stage, hysteresis thresholding determines which of possible edges are really edges using two thresholds values, *low* and *high*. First, the set of pixels with $G \geq high$ and the set of pixels with $G \leq low$ are directly classified as edges and non-edges, respectively. Then, the remaining possible edges (i.e., those with $low < G < high$) are classified as edges if and only if they are connected (directly or via other possible edges) to pixels with $G \geq high$. In the rest of the paper, the set of pixels with $low < G < high$ will be referred to as *instability zone* [11], and their classification process as *linking process* [11]. Additionally, we define the *instability map* as a binary image of the same dimensions as $G$, in which the value of pixel $(i, j)$ is 1 if the pixel $(i, j)$ of $G$ belongs to the instability zone, or 0 otherwise.

✉ Antonio Fuentes-Alventosa
antonio.fa@gmail.com

Juan Gómez-Luna
juang@ethz.ch

R. Medina-Carnicer
rmedina@uco.es

[1] Department of Computer Sciences and Numerical Analysis, University of Córdoba, Córdoba, Spain

[2] Department of Information Technology and Electrical Engineering, ETH Zürich, Zürich, Switzerland

The main drawback of the Canny edge detector is that it is time consuming, due to its high computational complexity. To overcome this limitation, many implementations of the algorithm have been presented on different accelerating platforms, such as ASICs [12–14], FPGAs [15–23] and GPUs [25–32].

There are several ASICs implementations of Deriche filters, which have been derived from Canny's criteria. Deriche [12] presented a network with four transputers that took 6 s to detect edges in a $256 \times 256$ image, which is far from real-time requirements. Torres et al. [13] proposed a faster solution that processed 25 frames/s at 33 MHz, but the area overhead was increased by the use of Last-In First-Out (LIFO) stacks in off-chip SRAM memories. Lorca et al. [14] presented a new design that improved that of [13] by reducing the memory size and the computation cost by a factor of two. Nevertheless, the number of clock cycles per pixel varies with the image size, and the processing time increases with the size of the image.

Some efforts have been made to accelerate Canny edge detection using FPGAs [15–23]. The proposals in [15] and [16] translated the software designs directly into hardware description languages (Handel-C and VHDL, respectively), which resulted in timing performance degradation. Gentsos et al. [17] presented a parallel architecture of simultaneous 4-pixel calculation that reduced the latency of the implementations of [15] and [16]. He et al. [18] proposed a self-adapt threshold Canny algorithm to overcome the drawback of setting the hysteresis thresholds manually in existing hardware implementations. In their method, hysteresis thresholds are calculated from the histogram of gradient magnitude. Their algorithm required about 2.5 ms to detect the edges of a 360×280 image on a FPGA chip EP1C60240C8 (Altera Cyclone) based platform. Li et al. [19] presented other solution for self-adapt threshold Canny algorithm, which adopted a Shifting-LUT-based direction calculation algorithm to improve the processing speed. The processing time was 5.24 ms for a 512×512 image on a Xilinx's Virtex-5 FPGA. Peng et al. [20] proposed an improved high-speed Canny edge detection algorithm based on FPGA, in which the gradient is calculated by the second harmonic of the variable parameters (SHOVP) to simplify complex arithmetic into logic operation. The feasibility and effectiveness of the algorithm was tested on Altera DE2 platform. Abdelgawad et al. [21] proposed an implementation of Canny algorithm on Zynq platform using Vivado High Level Synthesis (HLS). The achieved results showed that the collaboration of CPU and FPGAs enabled up to a 100x performance improvement. The CPU utilization dropped down and the frame rate was up to 60 fps for 1280×1024 resolution. Xu at al. [22] presented 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, their method uses a non-uniform gradient magnitude histogram to compute block-based hysteresis thresholds. The implementation of the algorithm on a Xilinx Virtex-5 FPGA platform 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. Sangeetha et al. [23] proposed a cost-effective robust Canny edge detection algorithm, whose keys contributions are the following: (1) computation of gradient magnitude and orientation using approximate method, (2) block classification techniques, and (3) adaptive threshold calculation of each block. Results on Xilinx Virtex-5 FPGA showed that the algorithm requires only 0.672 ms to detect the edges of 512×512 image when clocked at 100 MHz.

In the area of General Purpose Graphic Processing Unit (GPGPU), several efficient implementations of the Canny algorithm have been proposed [25–32]. Luo and Duraiswami [25] presented the first implementation of the Canny algorithm on the popular NVIDIA CUDA framework [33]. They mapped the entire algorithm to the GPU, and improved previous similar implementations on NVIDIA Cg [34] and Khronos Group GLSlang [24] that not included the hysteresis stage. The convolution steps (Gauss and Sobel filtering) are efficiently implemented using a separable filter algorithm, similar to the one supplied with the CUDA toolkit [35]. The gradient magnitude and direction are easily obtained by calculating the L2 norm and the arctangent, respectively, of the first derivatives on a simple pixel to thread mapping. The gradient direction of each pixel is quantized to one of the eight directions corresponding to the neighboring pixels ($\pi/8 + k\pi/4$). Non-maximum suppression is performed on a straightforward way by setting to 0 the gradient magnitudes that are not local maximums in the gradient direction. Hysteresis is performed by a kernel of 16×16 thread-blocks, each of which processes a separate 16×16 pixel-block of the gradient along with a one pixel wide apron around the 16×16 pixel-block, resulting in a 17×17 pixel-block. Each thread-block loads its assigned 17×17 pixel-block to shared memory, and executes a breadth first search (BFS) algorithm on it to classify the pixels of the internal 16×16 pixel-block as edges or non-edges. This classification is carried out by assigning -2 to the gradient magnitude, if the pixel is an edge, or 0, otherwise. Once a thread-block finishes the BFS process, it writes the edge states of all non-apron pixels in shared memory back into the gradient magnitude space in global memory. Subsequent calls to the hysteresis kernel will allow the linking among pixels that belong to different 16×16 pixel-blocks, thanks to the reloading of the updated edge states of apron pixels into shared memory. Due to this multi-pass approach, the implementation speed is dominated by the hysteresis process. Experimental evaluation showed that it occupies more than 70% of the total runtime. For testing purposes, the

hysteresis kernel was called four times per iteration, as no significant improvement was observed with higher values for the test images. Experiments showed a significant speedup against straightforward CPU functions, but a moderate improvement against multi-core multi-threaded CPU functions taking advantage of special instructions. The measured execution time for a 512×512 image was 3.40 ms. Ogawa et al. [26] presented a solution based on the work of Luo and Duraiswami [25], in which they described an issue in the traversing of all weak edge pixels, and proposed a stack-based mechanism to fix it. In the hysteresis thresholding stage, if the pixel assigned to a thread is a strong edge pixel, the thread uses a stack to traverse the adjacent weak edge pixels, which are labeled as final edge pixels. Experimental evaluation showed a runtime of 364.389 ms for a 10240×10240 image. The logarithmic image processing (LIP) model is a robust mathematical framework that is compatible with what is known about the human visual process [36]. In [27], Palomar et al. presented the implementation of two LIP-Canny methods, one operating images in LIP space with traditional operators, and the other operating images in natural space with modified operators. The work of Palomar et al. [27] was based on those of Palomares et al. [37] and Luo and Duraiswami [25]. As in [25], the number of iterations of the hysteresis kernel was fixed to 4. Experimental evaluation showed that CUDA implementations are 10–16 times faster than the corresponding C++ implementations. Moreover, LIP-Canny using modified operators is slightly faster than the alternative approach based on classical operators. The average runtimes for 512×512 images were 26.448 ms and 28.848 ms for the first and second method, respectively. Lourenço et al. [28] developed a CUDA implementation of the Canny algorithm for the Insight Segmentation and Registration Toolkit (ITK) using second-order derivatives (instead of Sobel filtering [25]) and a hybrid CPU-GPU approach for the hysteresis stage that closely followed the method proposed in [25]. Experimental evaluation showed that the CUDA implementation on three generations of NVIDIA GPGPUs was between 3.6 and 50 times more faster than the standard ITK Canny implementation on two CPU models. The main novelties of the CUDA implementation proposed by B. M. L. P. Vigil [29] are the application of Otsu method for automatic calculation of hysteresis thresholds, and the use of interpolation in the non-maximum suppression step to improve the quality of edge detection. The hysteresis thresholding is performed by the same hybrid CPU-GPU technique used in previous works, and, hence, it occupies a considerable percentage in the total execution time (more than 50%). The execution times of the CUDA Canny detector for 512× 512 Lena, Mandrill and Peppers images were 8.49 ms, 9.84 ms and 10.90 ms, respectively. Huang et al. [30] presented a CUDA implementation on the embedded CPU and GPU heterogeneous computing platform Jetson TK1 of NVIDIA. Noise reduction, gradient computation and non-maximum suppression are efficiently implemented in a similar way to that of [25]. However, the linking process is replaced by a simpler schema, which classifies a pixel of the instability zone as an edge pixel if at least one of its eight neighboring pixels is an edge pixel. Additionally, the hysteresis thresholds are obtained from the histogram of gradient magnitude. Experimental evaluation showed that the runtimes for 512× 512 Lena and Peppers images were approximately 3 ms. In [32], Emrani et al. presented a CUDA implementation of Canny algorithm in which the main novelty was the replacement of the Luo and Duraiswami's BFS algorithm [25] with a more efficient method. The kernel corresponding to this method checks whether a pixel belongs to the instability zone or not. If so, it will check its neighboring pixels. If a strong edge is found, the current pixel is classified as an edge pixel. A flag in global memory is used to indicate whether any pixel of the instability zone has been classified as an edge pixel. The kernel is launched as long as the flag is set. The execution time of the CUDA Canny detector for a 512× 512 image was 37.35 ms on a GeForce GTX 550 Ti GPU.

As we have just seen, the main bottleneck of GPU-based implementations of Canny algorithm is the hysteresis step, due to the need of calling the hysteresis kernel an indeterminate number of times (at least 4) executed on host side. On the other hand, in all implementations, except B. M. L. P. Vigil's [29] and Huang et al.'s [30], the hysteresis thresholds are adjusted manually. In this work, we propose a novel GPU-based implementation of the Canny algorithm on CUDA that overcomes these limitations. As in [22] and [23], the image is partitioned into sub-images, and the following steps are performed on each sub-image in parallel: (1) calculation of the optimal hysteresis thresholds, and (2) hysteresis process using the parameters obtained in the previous step. As each sub-image is processed independently, it is not necessary the costly hybrid CPU-GPU approach of previous implementations for hysteresis stage. The calculation of hysteresis thresholds is carried out with Medina-Carnicer's method [11], which, at present, is relevant for unsupervised edge detection because, since its introduction, it has been used to find automatically the hysteresis thresholds in many works [38–51]. Medina-Carnicer's method [11] outperforms those used in previous implementations of Canny algorithm [18, 19, 22, 23, 29], because the first searches the optimal values of both hysteresis thresholds *low* and *high*, while the latter do not, since they assume a constant ratio *low*/*high*. Experimental evaluation showed that our GPU-based unsupervised and distributed Canny edge detector, which we have named GUD-Canny, requires only between 0.33 and 0.48 milliseconds to detect edges on 512×512 images, which fully satisfies real-time requirements and outperforms reported runtimes of existing FPGA and GPU solutions.

The rest of the paper is organized as follows. Section 2 gives a brief overview of Medina-Carnicer's method. Section 3 presents GUD-Canny. Section 4 shows the experimental evaluation of our solution, and, finally, the main conclusions are stated in Sect. 5.

## 2 Medina-Carnicer's method for unsupervised determination of hysteresis thresholds

### 2.1 Background

In [11], Medina-Carnicer et al. presented a novel method to look for the hysteresis thresholds in an unsupervised way. Given a set of candidate thresholds pairs, the key idea is to combine the gradient information with that obtained from applying the linking process for all the candidate thresholds pairs. Experimental evaluation showed that the performance of Medina-Carnicer's algorithm is better than those of previous methods [52, 53]. The computational complexity of Medina-Carnicer's algorithm [11] is smaller than that of the solution presented in [53], but bigger than that of the proposal in [52]. Nevertheless, the approach in [52] only finds an approximate edge map and it is not able to find the hysteresis thresholds. The results obtained by Medina-Carnicer's method [11] have been validated only for the Canny edge detector, but there are no restrictions to apply it to any other edge detector whose strategy is based on the hysteresis mechanism.

The main innovations presented in [11] are the following:

1. In contrast to previous works [53–56], which are aimed at directly searching for hysteresis thresholds, it follows an indirect way, which consists of looking for the instability zone and then determining the hysteresis thresholds from it.
2. Unlike previous proposals [53, 55, 56], which only use gradient information, it combines the latter with that of the linking process.

### 2.2 Steps summary of Medina-Carnicer's method

Let $I$ be an image, $G$ its gradient magnitude after non-maximum suppression normalized in the interval [0,1], and $C$ a set of candidate thresholds pairs $\{(low, high), low, high \in (0, 1)\}$.

Given a hysteresis thresholds pair $(low, high)$, we define the following edge maps:

- *Hysteresis map* ($G_{low,high}$), which is obtained by performing the hysteresis process on $G$ with $(low, high)$.
- *High map* ($G_{high}$), which is the result of thresholding $G$ with $high$.
- *Linking map* ($\Delta G_{low,high}$), which is composed exclusively of the edges added by the linking process using $(low, high)$. Note that $\Delta G_{low,high} = G_{low,high} - G_{high}$.

The steps of Medina-Carnicer's method are the following:

1. Calculate a set $H$ of linking maps corresponding to the candidate thresholds pairs of $C$.

$$H = \{\Delta G_{low,high}, (low, high) \in C\} \tag{1}$$

2. Compute the sum $SM_H$ of the linking maps.

$$SM_H = \sum H \tag{2}$$

In this matrix, the value of each element is the number of times that the corresponding pixel of $G$ is classified as edge by the linking process for all the candidate thresholds pairs.

3. Calculate the division of $SM_H$ by the cardinality of $C$, which will be denoted as $Prob(SM_H)$.

$$Prob(SM_H) = SM_H/|C| \tag{3}$$

Each element of $Prob(SM_H)$ represents the probability that the corresponding pixel of $G$ is classified as edge by the linking process.

4. Compute the distribution $P(F(x)), \forall x \in (0, 1)$, defined as follows:

$$P(F(x)) = \begin{cases} \frac{|F(x)|}{|Prob_x(SM_H))|} & |Prob_x(SM_H)| > 0 \\ 0 & |Prob_x(SM_H)| = 0 \end{cases} \tag{4}$$

where

- $Prob_x(SM_H)$ is the binary edge map obtained by thresholding $Prob(SM_H)$ with $x \in (0, 1)$. Its elements with value 1 correspond to the pixels of $G$ that have a probability equal or greater than $x$ of being classified as edges by the linking process.
- $|Prob_x(SM_H)|$ is the number of elements with value 1 in $Prob_x(SM_H)$.
- $F(x) = G \circ Prob_x(SM_H)$, where $\circ$ is the Hadamard product.
- $|F(x)|$ is the number of elements with value $x$ in $F(x)$.

The distribution $P(F(x))$ represents the probability that a pixel has gradient level $x$ if it is a pixel with probability equal or greater than $x$ of being added by the linking process. It is the combined information used by Medina-Carnicer's method.

5. Compute the histogram of $Prob(SM_H)$ for the set $D = \{x \in (0, 1) | P(F(x)) \neq 0\}$, which represents the instability zone. The hysteresis thresholds are the values of $D$

corresponding to the first and last local maximums of the histogram.

The set $C$ is obtained by sampling an interval $[0.01, MAX\_HIGH]$, where $0.01 < MAX\_HIGH \leq 1.0$. In [11], Medina-Carnicer et al. showed that two selections of $C$ that ensure a good performance of their method are those obtained by sampling the interval [0.01, 0.25] with steps 0.01 and 0.03. Furthermore, the results presented in [53] indicate that their approach, in general, depends less on the initial set than the method of Yitzhaky and Peli [56] does.

## 3 GPU-based unsupervised and distributed Canny edge detector (GUD-Canny)

In this section, we describe *GUD-Canny*, our GPU-based unsupervised and distributed Canny edge detector, which has been developed using the popular NVIDIA CUDA framework [33]. In the presented algorithms, the following notation is employed:

- Prefixes *d_*, *s_* and *c_* in the names of the variables indicate that they are allocated in global, shared and constant memory spaces, respectively.
- Symbols&, |, ~, << and >> are the bitwise operators AND, OR, NOT, left shift and right shift, respectively.

Algorithm 1 provides a high-level description of GUD-Canny. As it can be seen, the inputs of our method are the following. First, a $W \times H$ image, which is provided in a vector of $P$ 8-bit unsigned integers (*d_image*), where $P$ is the number of pixels. Second, the standard deviation $\sigma$. Third, a set of *NCTP* candidate thresholds pairs, which is supplied in a vector of float pairs (*c_C*). On the other hand, the output of GUD-Canny are the edges of the input image, which are written in a vector of $P$ 8-bit unsigned integers (*d_edges*).

Steps 1–3 correspond to the classic first stages of Canny edge detection. To apply Medina-Carnicers's method (steps 4 to 7), the non-maximum suppression returns the gradient magnitude normalized in the interval [0, 1] (*d_G*). The gradient magnitude is partitioned horizontally into $NS = W/32$ sub-images of dimension $32 \times H$, and Medina-Carnicer's method [11] is used to calculate an optimal pair of hysteresis thresholds for each sub-image. Finally, in step 8, the hysteresis map is computed for each subimage using its assigned hysteresis thresholds pair, and written in the output vector *d_edges*.

Since the original width of the input image may not be a multiple of 32, the CUDA function *cudaMemcpy2D* [57] is used to copy the input image from host to device memory adding the necessary padding to each row, and the same function is called to copy the output edges from device to host memory.

In the following subsections, each step of GUD-Canny is described in detail.

**Algorithm 1:** GUD-Canny

```
function GUD-Canny(// output
                   uchar d_edges[P],
                   // input
                   uchar d_image[P],
                   float σ,
                   float2 c_C[NCTP]
                   )
  // Step 1: Gaussian filtering
  uchar d_smoothed_image[P]
  d_smoothed_image ← gaussian_filter(d_image, σ)

  // Step 2: gradient computation
  short d_grad_x[P], d_grad_y[P],
        d_grad_mag[P], d_max_grad_mag[1]
  (d_grad_x, d_grad_y, d_grad_mag,
   d_max_grad_mag) ← compute_gradient(d_smoothed_image)

  // Step 3: non-maximum suppression
  float d_G[P]
  d_G ← non_maximum_suppression(d_grad_x, d_grad_y,
                                d_grad_mag, d_max_grad_mag)

  // Step 4: calculation of matrices SM_H
  uint d_SM_H[P]
  d_SM_H ← calc_SM_H(d_G, c_C)

  // Step 5: calculation of matrices Prob(SM_H)
  float d_Prob_SM_H[P]
  d_Prob_SM_H ← calc_Prob_SM_H(d_SM_H, NCTP)

  // Step 6: calculation of distributions P(F(x)) and
  // histograms of matrices Prob(SM_H)
  float d_PF[NS][NX + 1]
  uint d_histo[NS][NX + 1]
  (d_PF, d_histo) ← calc_PF_histo(d_G, d_Prob_SM_H)

  // Step 7: searching of hysteresis thresholds for
  // each subimage of G
  float2 d_thresholds[NS]
  d_thresholds ← search_thresholds(d_PF, d_histo)

  // Step 8: hysteresis thresholding
  d_edges ← hysteresis(d_G, d_thresholds)
end function
```

### 3.1 Gaussian filtering

To reduce the impact of noise, the input image is smoothed by convolving it with two one-dimensional Gaussian filters in the horizontal and vertical dimensions.

Each Gaussian filtering is performed by a different CUDA kernel, in which each output pixel is computed by a different thread. Kernels implementations are similar to those

presented in [35], but with the difference that the shared memory is not used for caching data. Since the hardware cache system ensures a good performance [57], all read/write operations are performed directly to global memory.

Each thread initializes each element of the input image vector used to perform the convolution dot product as follows. If it corresponds to an existing pixel, i.e., the position of the pixel is not outside the borders of the image, it is read from the input image. Otherwise, it is assigned the value zero.

As in [27], the length of Gaussian filters is variable and depends on the standard deviation $\sigma$. Each kernel obtains the Gaussian filter from a table in constant memory, which stores the Gaussian filters corresponding to $\sigma$ values between 0.1 and 2.0. The first table entry corresponds to $\sigma = 0.1$, the second one to $\sigma = 0.2$, and so on up to 2.0.

### 3.2 Gradient computation

After Gaussian filtering, each gradient tuple $(d_x, d_y)$ is calculated using the first difference operator $(-1, 0, 1)$, and the associated gradient magnitude by the formula $\sqrt{d_x^2 + d_y^2}$. The results are written in the output vectors *d_grad_x*, *d_grad_y* and *d_grad_mag*, respectively. As in Gaussian filtering step, all read/write operations are made directly to global memory, and the border conditions are carefully checked.

Additionally, to compute the maximum gradient magnitude, each thread performs an atomic maximum operation (using the CUDA function *atomicMax* [57]) between the calculated gradient magnitude and a global memory variable (*d_max_grad_mag*[0]), which has been initialized to zero.

### 3.3 Non-maximum suppression

In this step, one kernel computes a new version of gradient magnitude (*d_G*) by performing non-maximum suppression and normalization on the gradient magnitude obtained in the previous stage (*d_grad_mag*). Given a pixel of value *p* in *d_grad_mag*, the value of the corresponding pixel in *d_G* is *p*/*d_max_grad_mag*[0] if the pixel is a maximum in the gradient direction, or zero otherwise. Each pixel of *d_G* is computed by a different thread of the kernel.

The method used for maximum suppression is the one employed in [29], which quantizes gradient direction to one of the eight directions $\{\pi/8 + k\pi/4\}$, and uses linear interpolation to calculate the values of the two neighboring pixels in the gradient direction.

Global memory operations and border conditions management are executed as in previous steps.

### 3.4 Hysteresis thresholds computation

As we said previously, the gradient magnitude is partitioned horizontally into $NS = W/32$ sub-images, and the method of Medina-Carnicer is applied on each one in parallel.

Images are processed by dividing them into groups of 32 consecutive pixels in the horizontal dimension, which will be referred to as *regions*. The numbers of regions of an image and of a sub-image will be denoted by *NRI* and *NRS*, respectively. For simplicity, the regions of instability/hysteresis/high/linking maps will be referred to as *instability/hysteresis/high/linking regions*, respectively.

Each of the steps 4–7 of Algorithm 1 is performed by a different kernel, whose actions are specified in the following subsections.

#### 3.4.1 Calculation of the matrices $SM_H$

Algorithm 2 presents the pseudo code of the kernel *calc_SM_H*, which calculates the matrix $SM_H$ for each sub-image of *G*. The inputs are *G*, which is provided in a vector of *P* 32-bit floats (*d_G*), and *C*, which is supplied in a vector of *NCTP* 32-bit float pairs, initialized statically in constant memory (*c_C*). The output are the *NS* matrices $SM_H$ corresponding to the *NS* sub-images of *G*, which are written in a vector of *P* 32-bit unsigned integers (*d_SM_H*), initialized to 0. Maps regions are represented by 32-bit unsigned integers, where the i-th bit stores the binary value of the i-th pixel of the region. Although the gradient regions reads in step 1 are not coalesced, as CUDA literature [58] [57] recommends, they satisfy the principle of spatial locality because each thread reads 32 consecutive elements of *d_G*, which are properly aligned. Therefore, the transparent cache hierarchy of modern GPU architectures ensures a good performance while reading the gradient regions. On the other hand, the writes in step 5 are carried out atomically using the CUDA function *atomicAdd* [57].

**Algorithm 2:** calc_SM_H

```
function calc_SM_H(// output
                   uint d_SM_H[P],
                   // input
                   float d_G[P],
                   float2 c_C[NCTP]
                   )
  // Declarations
  float low, high, grad_reg[32]
  uint high_reg, inst_reg, hyst_reg, link_reg
  uint *d_SM_H_reg

  // Step 1: read input data
  {low, high} ← read candidate thresholds pair assigned
                to current thread-block from c_C
  grad_reg ← read gradient region assigned to current
             thread from d_G

  // Step 2: calculate high and instability regions
  high_reg ← 0
  inst_reg ← 0
  for i ← 0 to 31 do
      if (grad_reg[i] ≥ high) then
         set i-th bit of high_reg to 1
      else if (low < grad_reg[i] < high) then
         set i-th bit of inst_reg to 1
      end if
  end for

  // Step 3: calculate hysteresis region
  hyst_reg ← calc_hyst_map(high_reg, inst_reg)

  // Step 4: calculate linking region
  link_reg ← hyst_reg & ~high_reg

  // Step 5: update SM_H region
  d_SM_H_reg ← pointer to d_SM_H subvector corresponding
               to SM_H region assigned to current thread
  for i ← 0 to 31 do
      if (i-th bit of link_reg is 1) then
         atomicAdd(d_SM_H_reg[i], 1)
      end if
  end for
end function
```

In step 3, of Algorithm 2 each thread gets its hysteresis region (*hyst_reg*) by calling the function *calc_hyst_map*, which receives as inputs the high and instability regions of the calling thread (*high_reg* and *inst_reg*, respectively). The actions performed by this function are presented in Algorithm 3. As it can be seen, each thread-block computes its hysteresis map in a shared memory 32-bit unsigned int vector (*s_hyst_map*) of size *NRS*. Each hysteresis region $i$ is managed by the thread $i$, and held in the element *s_hyst_map*[$i$].

**Algorithm 3:** calc_hyst_map

```
function calc_hyst_map(// output
                   uint hyst_reg,
                   // input
                   uint high_reg,
                   uint inst_reg
                   )
  // Declarations
  shared uint s_hyst_map[NRS]
  shared bool s_flag_updated
  uint top_hyst_reg, bottom_hyst_reg
  uint link_mask, new_hyst_edges

  // Step 1: initialize the hysteresis map
  // to the high map
  s_hyst_map[thread_idx] ← high_reg
  hyst_reg ← high_reg

  repeat
      // Step 2: reset the flag s_flag_updated, which
      // is used to control the execution of the loop
      if thread_idx = 0 then
         s_flag_updated ← false
      end if
      synchronize all threads of the block

      // Step 3: check if the instability region
      // has edges
      if inst_reg ≠ 0 then
         // Step 4: read the top and bottom
         // hysteresis regions from the hysteresis map
         top_hyst_reg ← s_hyst_map[thread_idx - 1]
         bottom_hyst_reg ← s_hyst_map[thread_idx + 1]

         // Step 5: get the new edges to include
         // in the hysteresis region
         link_mask ← (hyst_reg << 1) & (hyst_reg >> 1)
            & top_reg & (top_reg << 1) & (top_reg >> 1)
            & bottom_reg & (bottom_reg << 1)
            & (bottom_reg >> 1)
         new_hyst_edges ← inst_reg & link_mask

         // Step 6: check if there are new edges
         if new_hyst_edges ≠ 0 then
            // Step 7: add the new edges to the
            // hysteresis region, and exclude them
            // from the instability region
            hyst_reg ← hyst_reg | new_hyst_edges
            inst_reg ← inst_reg & ~new_hyst_edges

            // Step 8: update the hysteresis map
            s_hyst_map[thread_idx] ← hyst_reg

            // Step 9: Register that the hysteresis map
            // has been updated
            s_flag_updated ← true
         end if
      end if

      // Step 10: thread-block synchronization
      synchronize all threads of the block

  // Step 11: if the hysteresis map has not been
  // updated in the last iteration, the loop ends
  until s_flag_updated = false
end function
```

An alternative way to divide the gradient magnitude into sub-images is by partitioning it vertically into $NS = H/32$ sub-images of dimension $W \times 32$. In this case, the spatial locality of accesses to global memory is improved, because consecutive threads access consecutive regions. On the other hand, the advantage of the horizontal partition is that the

number of operations in the linking process is reduced (step 5 of the function *calc_hyst_map*). The reason is that it is only necessary to examine the top and bottom regions; in the case of a vertical partition, the six remaining neighbor regions (left, top left, bottom left, right, top right and bottom right) have also to be taken into account. As will be shown in Sect. 4, GUD-Canny is slightly faster for sub-images of dimension $32 \times H$.

#### 3.4.2 Calculation of the matrices $Prob(SM_H)$

The matrix $Prob(SM_H)$ for each sub-image of $G$ is obtained by dividing each element of the corresponding matrix $SM_H$ by $NCTP$. The matrices $Prob(SM_H)$ are written in a vector of $P$ 32-bit floats ($d\_Prob\_SM\_H$).

The number of threads of the grid equals to $P$ divided by 4, and each thread $i$ performs the following actions:

1. Reads the group $i$ of four consecutive elements from $d\_SM\_H$ through one vectorized load.
2. Calculates the division of each element by $NCTP$.
3. Writes the four computed float values to the 4-elements group $i$ of $d\_Prob\_SM\_H$ through one vectorized store.

Vectorized accesses are an important GPU optimization, because they increase bandwidth and reduce both instruction count and latency [59].

#### 3.4.3 Calculation of the distributions $P(F(x))$ and the histograms of the matrices $Prob(SM_H)$

For each sub-image of $G$, the distribution $P(F(x))$ and the histogram of $Prob(SM_H)$ are computed by one kernel for $x \in \{0.01, 0.02, ..., MAX\_HIGH\}$. The number of $x$ values, which is $MAX\_HIGH/0.01$, will be denoted by $NX$.

The number of thread-blocks of the grid is $NS \times NX$. Each thread-block calculates $P(F(x))$ and the histogram of $Prob(SM_H)$ for one sub-image of $G$ and one $x$ value. The size of thread-blocks is $NRS$.

The actions performed by the kernel are shown in Algorithm 4, where *div* and *mod* are the quotient and remainder operators, respectively. The three parallel reductions are efficiently executed using the CUDA function *__shfl_down_sync* [57] and fast device memory atomic operations, as described in [60].

**Algorithm 4:** calc_PF_histo

```
function calc_PFI_histo(// output
                        float d_PF[NS][NX + 1],
                        uint d_histo[NS][NX + 1],
                        // input
                        float d_G[P],
                        float d_Prob_SM_H[P]
                        )
  // Declarations
  uint x_idx, subimage_idx, region_idx
  float x, Prob_reg[32], G_reg[32], dist_value
  uint Prob_sum, total_Prob_sum, G_sum, total_G_sum,
       hist_sum, total_hist_sum

  // Step 1: initializations
  x_idx ← 1 + (thread_block_index mod NX)
  subimage_idx ← thread_block_index div NX
  region_idx ← thread_index
  x ← 0.01 × x_idx

  // Step 2: process region of Prob(SM_H)
  Prob_reg ← read region region_idx from sub-image
             subimage_idx of d_Prob_SM_H
  Prob_sum ← count all elements of Prob_reg that are
             greater than or equal to x

  // Step 3: process region of G
  if (Prob_sum > 0) then
     G_reg ← read region region_idx from sub-image
             subimage_idx of G
     G_sum ← count all elements of G_reg that are equal
             to x whose associated elements in Prob_reg
             are greater than or equal to x
  end if

  // Step 4: thread-block reductions
  total_Prob_sum ← thread-block reduction of Prob_sum
  total_G_sum ← thread-block reduction of G_sum

  // Step 5: the first thread of the thread-block
  // calculates the P(F(x)) value and writes it
  // to d_Prob_SM_H
  if (thread_idx = 0) then
     if (total_Prob_sum = 0) then
        dist_value ← 0
     else
        dist_value ← total_G_sum / total_Prob_sum
     end if
     d_PF[subimage_idx][x_idx] ← dist_value
  end if

  // Step 6: check if the histogram value is non-zero
  if (total_G_sum > 0) then
      // Step 7: process region of Prob(SM_H)
      hist_sum ← count all elements of Prob_reg that
                 are equal to x
      total_hist_sum ← thread-block reduction of
                       hist_sum

      // Step 8: the first thread of the thread-block
      // writes the histogram value to d_histo
      if (thread_idx = 0) then
          d_histo[subimage_idx][x_idx] ← total_hist_sum
      end if
  end if
end function
```

### 3.4.4 Searching of hysteresis thresholds

The hysteresis thresholds searching for each sub-image of $G$ is performed by the kernel described in Algorithm 5. The number of warps of the grid is $NS$, and each warp $i$ searches for the hysteresis thresholds pair of sub-image $i$. It is assumed that $NX < 32$.

The warp votes are performed by calling the CUDA function *__balloc_sync* [57], which, given a predicate, evaluates it for all threads in the current warp, and returns a 32-bit binary mask, in which each bit $j$ is set if the predicate evaluates to non-zero for the lane $j$.

The searches of bits within the masks are performed efficiently using the CUDA integer intrinsic functions *__ffs* and *__brev* [61]. The first one finds the position of the least significant bit set to 1 in a 32-bit integer, and the second one reverses the bit order of a 32-bit unsigned integer.

**Algorithm 5:** search_thresholds

```
function search_thresholds(// output
                  float2 d_thresholds[NS],
                  // input
                  float d_PF[NS][NX + 1],
                  uint d_histo[NS][NX + 1]
                  )
  // Declarations
  uint curr_x_idx, subimage_idx, left_x_idx, right_x_idx,
    x_index_of_first_local_max, x_index_of_last_local_max
  float curr_x, PF_x, hist_x, hist_left_x,
        hist_right_x, low, high
  uint PF_mask, hist_mask
  bool local_mask

  // Step 1: initializations
  curr_x_idx ← warp_lane
  subimage_idx ← grid_warp_idx
  curr_x ← 0.01 × curr_x_idx

  // Step 2: each thread of the warp reads its assigned
  // values in P(F(x)) and the histogram of Prob(SM_H(x))
  if curr_x_index ≤ NX then
     PF_x ← d_PF[subimage_idx][curr_x_idx]
     hist_x ← d_histo[subimage_idx][curr_x_idx]
  else
     PF_x ← 0
     hist_x ← 0
  end if

  // Step 3: each thread of the warp gets the indices
  // of the x values for which P(F(x)) and the histogram
  // of Prob(SM_H(x)) are non-zero
  PF_mask ← warp-voting(PF_x ≠ 0)
  hist_mask ← warp-voting(hist_x ≠ 0)

  // Step 4: determine the local maximums of the
  // histogram of Prob(SM_H(x)) for which P(F(x))≠0
  if PF_x = 0 then
     local_max_mask ← false
  else
     left_x_index ← find index of first one bit of
           PF_mask at right of bit of index curr_x_index
     right_x_index ← find index of first one bit of
            PF_mask at left of bit of index curr_x_index

     if left_x_index and right_x_index were found then
       hist_left_x ← d_histo[subimage_idx][left_x_idx]
       hist_right_x ← d_histo[subimage_idx][right_x_idx]
       local_max ← (hist_x > hist_left_x) and
                   (hist_x > hist_right_x)
     else
        local_max ← false
     end if
  end if
  local_max_mask ← warp-voting(local_max = true)

  // Step 5: search the first and last local maximums,
  // which correspond to the low and high hysteresis
  // thresholds, respectively
  x_index_of_first_local_max ← find index of first one
                               bit of local_max_mask
  x_index_of_last_local_max ← find index of last one
                              bit of local_max_mask

  // Step 6: return the hysteresis thresholds pair
  low ← 0.01 × x_index_of_first_local_max
  high ← 0.01 × x_index_of_last_local_max
  d_thresholds[subimage_idx] ← {low, high}
end function
```

### 3.5 Hysteresis thresholding

The hysteresis thresholding is carried out by the kernel presented in Algorithm 6, which is very similar to Algorithm 2. The number of thread-blocks of the grid is *NS*, and the size of each thread-block is *NRS*. The i-th thread-block calculates the hysteresis map corresponding to the i-th sub-image of *G* following the same steps of Algorithm 2. Then, the j-th thread of the thread-block writes the pixels values specified in its hysteresis mask (*hyst_reg*) to the j-th region of the corresponding output edges sub-image.

To write the hysteresis region, each thread accesses the output edges image through a pointer to a structure of 32 8-bit unsigned int members. As in the case of gradient regions reading, although the accesses to global memory are not coalesced, they satisfy the principle of spatial locality, and are properly aligned.

**Algorithm 6:** hysteresis

```
function hysteresis(// output
                    uchar d_edges[P],
                    // input
                    float d_G[P],
                    float2 d_thresholds[NS]
                    )
  // Declarations
  float low, high, grad_reg[32]
  uint high_reg, inst_reg, hyst_reg
  uchar *d_edges_reg
  uchar edges_reg[32]

  // Step 1: read input data
  {low, high} ← read thresholds pair assigned to current
                thread-block from d_thresholds
  grad_reg ← read gradient region assigned to current
             thread from d_G

  // Step 2: calculate high and instability regions
  high_reg ← 0
  inst_reg ← 0
  for i ← 0 to 31 do
      if (grad_reg[i] ≥ high) then
         set i-th bit of high_reg to 1
      else if (low < grad_reg[i] < high) then
         set i-th bit of inst_reg to 1
      end if
  end for

  // Step 3: calculate hysteresis region
  hyst_reg ← calc_hyst_map(high_reg, inst_reg)

  // Step 4: calculate edges region
  for i ← 0 to 31 do
      edges_reg[i] ← i-th bit of hyst_reg
  end for

  // Step 5: write edges region to the output edges image
  d_edges_reg ← pointer to d_edges subvector correspon-
                ding to edges region assigned to current
                thread
  *d_edges_reg ← edges_reg
end function
```

## 4 Experimental evaluation

To evaluate the performance of GUD-Canny edge detection, we used the ground truth images of Heath's dataset [62], that can be downloaded from ftp://figment.csee.usf.edu/pub/Edge_Comparison/images/results/. The 28 gray reference images of this dataset were selected by humans from a limited set of edge maps, which were obtained using the Canny edge detector with different values for its parameters.

We utilized the same two candidate thresholds sets selected in [11], which were those obtained by sampling the interval [0.01, 0.25] with steps 0.01 and 0.03, and that will be denoted by $C_{0.01}$ and $C_{0.03}$, respectively.

Our test machine had a 3.50Ghz Intel Core i7-7800X CPU and 32 GB of RAM. The GPU that we used was a GeForce RTX 2080 (Turing architecture with compute capability 7.5), and no optimization flags were utilized in our implementation.

### 4.1 Quality evaluation

In the first experiment, we compared the quality obtained by applying Medina-Carnicer's method to the entire W×H image (classical *frame-level approach*) with the quality resulting from executing the same method on each 32×H sub-image (*distributed approach*, which is the focusing of GUD-Canny). Table 1 shows the mean-square errors (*MSE*) obtained for sets $C_{0.01}$ and $C_{0.03}$. In each row, for each candidate thresholds set, the minimum MSE is highlighted in bold. As it can be seen, the good performance of Medina-Carnicer's method not only remains in the distributed approach, but it even slightly outperforms that of frame-level approach. For the set $C_{0.01}$, the average MSEs for classical and distributed approaches were 0.0534 and 0.0498, respectively. In the case of the set $C_{0.03}$, the values were 0.0534 and 0.0502, respectively.

On the other hand, it can be observed that there is no big difference between the quality obtained using $C_{0.01}$ with respect to that resulting from utilizing $C_{0.03}$, as the average MSEs are 0.0498 and 0.0502, respectively.

### 4.2 Temporal efficiency evaluation

Table 2 presents the GUD-Canny edge detection times for sets $C_{0.01}$ and $C_{0.03}$. At the end of each column, statistics (average, minimum and maximum) are presented for all images, and for those of size 512×512. Additionally, Table 3 shows the statistics of GUD-Canny speedup for $C_{0.03}$ with respect to $C_{0.01}$. From the presented results, we can see the following points:

**Table 1** MSE values for frame-level Canny edge detection and distributed Canny edge detection using Medina-Carnicer's method for unsupervised determination of hysteresis thresholds

| Image | Frame, $C_{0.01}$ | Dist., $C_{0.01}$ | Frame, $C_{0.03}$ | Dist., $C_{0.03}$ |
|---|---|---|---|---|
| Airplane (659×409) | 0.0095 | **0.0081** | 0.0103 | **0.0071** |
| Banana (512×468) | **0.0289** | 0.0422 | **0.0310** | 0.0351 |
| Basket (512×512) | 0.0670 | **0.0603** | **0.0499** | 0.0524 |
| Beehive (512×512) | **0.0270** | 0.0284 | **0.0270** | 0.0278 |
| Briefcase (577×419) | **0.0237** | 0.0253 | 0.0269 | **0.0263** |
| Brush (572×512) | 0.0407 | **0.0243** | 0.0407 | **0.0259** |
| Coffeemaker (461×665) | **0.0275** | 0.0277 | 0.0291 | **0.0289** |
| Egg (512×512) | **0.0522** | 0.0540 | **0.0534** | 0.0550 |
| Elephant (512×456) | **0.0523** | 0.0661 | 0.0828 | **0.0727** |
| Feather (512×512) | 0.0797 | **0.0640** | 0.0643 | **0.0624** |
| Flower (536×509) | **0.0207** | 0.0260 | 0.0249 | **0.0247** |
| Golfcart (548×509) | 0.0607 | **0.0577** | 0.0914 | **0.0721** |
| Grater (512×438) | **0.0204** | 0.0210 | 0.0252 | **0.0224** |
| Mailbox (512×512) | **0.0461** | 0.0479 | **0.0531** | 0.0550 |
| Orange (412×472) | 0.0691 | **0.0679** | **0.0676** | 0.0688 |
| Pillow (552×468) | 0.0394 | **0.0360** | **0.0341** | 0.0357 |
| Pinecone (512×512) | 0.0687 | **0.0629** | 0.0603 | **0.0566** |
| Pitcher (568×419) | **0.0165** | 0.0169 | 0.0195 | **0.0188** |
| Pond (512×512) | **0.0719** | 0.0724 | 0.0778 | **0.0735** |
| Shopping cart (512×512) | 0.1188 | **0.0761** | 0.0949 | **0.0781** |
| Stairs (579×441) | **0.0496** | 0.0498 | 0.0635 | **0.0540** |
| Stapler (529×510) | **0.0335** | 0.0373 | **0.0360** | 0.0376 |
| Tiger (512×512) | 0.1811 | **0.1376** | 0.1554 | **0.1270** |
| Tire (512×512) | **0.1018** | 0.1102 | **0.1018** | 0.1105 |
| Traffic Cone (437×604) | 0.0768 | **0.0636** | 0.0662 | **0.0617** |
| Trashcan (539×433) | 0.0528 | **0.0521** | 0.0528 | **0.0525** |
| Turtle (512×512) | **0.0142** | 0.0145 | **0.0139** | 0.0165 |
| Videocamera (577×435) | **0.0441** | 0.0445 | **0.0420** | 0.0464 |
| Average | 0.0534 | **0.0498** | 0.0534 | **0.0502** |
| Minimum | 0.0095 | **0.0081** | 0.0103 | **0.0071** |
| Maximum | 0.1811 | **0.1376** | 0.1554 | **0.1270** |

For each image and candidate thresholds set, the minimum MSE is highlighted in bold

1. GUD-Canny fully satisfies real time requirements, as its execution times are on average 1.2736 ms and 0.3637 ms for sets $C_{0.01}$ and $C_{0.03}$, respectively.
2. For the set $C_{0.03}$, the edge detection times are between 0.2814 and 0.3932 milliseconds for 512×512 images. Hence, GUD-Canny outperforms the temporal efficiency of existing GPU and FPGA implementations, like the solution of Sangeetha et al. [23], whose edge detection time is 0.672 ms for 512×512 images.
3. The speedup obtained using $C_{0.03}$ instead of $C_{0.01}$ is significant, as its values are between 2.99x and 3.90x. The reason is that the number of linking maps that have to be calculated for $C_{0.03}$ ($36 \times NS$) is much less than that for $C_{0.01}$ ($300 \times NS$). This contrasts with the small difference between the quality of edge maps obtained with these candidate thresholds sets.

### 4.3 Distribution of execution times

Tables 4 and 5 show the statistics (average, minimum and maximum) of kernels execution time proportions (expressed as percentages) for sets $C_{0.01}$ and $C_{0.03}$.

Unlike the case of existing GPU-based Canny edge detectors, the hysteresis stage is executed efficiently, as its average time proportions are 2.02% and 7.70% for sets $C_{0.01}$ and $C_{0.03}$, respectively.

As expected, due to their higher computational complexity, the most time-consuming operations are the calculation of matrices $SM_H$ (whose average time proportions are 82.38% and 39.39% for sets $C_{0.01}$ and $C_{0.03}$, respectively) followed by the computation of distributions $\{P(F(x))\}$ and histograms of matrices $Prob(SM_H)$(whose average time

**Table 2** Edge detection times (ms) for candidate thresholds sets $C_{0.01}$ and $C_{0.03}$

| Image | $C_{0.01}$ | $C_{0.03}$ |
|---|---|---|
| Airplane (659×409) | 1.2805 | 0.3385 |
| Banana (512×468) | 1.0715 | 0.3108 |
| Basket (512×512) | 1.2234 | 0.3612 |
| Beehive (512×512) | 1.0518 | 0.3153 |
| Briefcase (577×419) | 1.2838 | 0.3496 |
| Brush (572×512) | 1.4066 | 0.3853 |
| Coffeemaker (461×665) | 2.1042 | 0.5396 |
| Egg (512×512) | 1.0554 | 0.3325 |
| Elephant (512×456) | 1.0943 | 0.3458 |
| Feather (512×512) | 1.3306 | 0.3607 |
| Flower (536×509) | 1.4382 | 0.3703 |
| Golfcart (548×509) | 1.5631 | 0.4238 |
| Grater (512×438) | 1.1136 | 0.3106 |
| Mailbox (512×512) | 1.3599 | 0.3932 |
| Orange (412×472) | 0.9376 | 0.2671 |
| Pillow (552×468) | 1.4201 | 0.3870 |
| Pinecone (512×512) | 1.1283 | 0.3770 |
| Pitcher (568×419) | 1.2057 | 0.3286 |
| Pond (512×512) | 1.2288 | 0.3480 |
| Shopping cart (512×512) | 1.2945 | 0.3721 |
| Stairs (579×441) | 1.4954 | 0.4707 |
| Stapler (529×510) | 1.1602 | 0.3283 |
| Tiger (512×512) | 1.3492 | 0.3739 |
| Tire (512×512) | 1.2050 | 0.3686 |
| Traffic Cone (437×604) | 1.5956 | 0.4115 |
| Trashcan (539×433) | 1.0523 | 0.3500 |
| Turtle (512×512) | 1.0314 | 0.2814 |
| Videocamera (577×435) | 1.1798 | 0.3833 |
| Average | 1.2736 | 0.3637 |
| Minimum | 0.9376 | 0.2671 |
| Maximum | 2.1042 | 0.5396 |
| Average (512×512) | 1.2053 | 0.3531 |
| Minimum (512×512) | 1.0314 | 0.2814 |
| Maximum (512×512) | 1.3599 | 0.3932 |

**Table 3** Statistics of GUD-Canny speedup for $C_{0.03}$ with respect to $C_{0.01}$

| Images | Average | Minimum | Maximum |
|---|---|---|---|
| All | 3.50x | 2.99x | 3.90x |
| 512×512 | 3.42x | 2.99x | 3.69x |

proportions are 7.21% and 24.87% for sets $C_{0.01}$ and $C_{0.03}$, respectively).

Figure 1 presents the statistics (average, minimum and maximum) of the total GPU time proportions corresponding to memory transferences. As it can be seen, the penalty is moderate because the percentages are less than 12% and 30% for sets $C_{0.01}$ and $C_{0.03}$, respectively.

### 4.4 Horizontal partitioning vs. vertical partitioning

Table 6 shows the edge detection times (ms) on 512×512 images using the candidate thresholds $C_{0.03}$ for sub-images sizes $32 \times H$ (horizontal partition) and $W \times 32$ (vertical partition). For each image, the minimum execution time is highlighted in bold. In all cases, the number of sub-images is 16 and the number of regions per sub-image is 512.

From the presented results, we can see that the execution times are slightly lower using the horizontal partitioning. The average speedup is 1.14x. As explained in Sect. 3.4.1, although the spatial locality of accesses to global memory is improved using vertical partitioning, the number of operations in the linking process is reduced if the sub-image size is $32 \times H$. Experimental evaluation has shown that the performance improvement due to the second factor is greater than that of the first.

## 5 Conclusions

This work has presented GUD-Canny, a novel GPU-based unsupervised and distributed implementation of Canny edge detector. Our solution overcomes the two main limitations of current Canny algorithm implementations, which are the bottleneck caused by the hysteresis process, and the use of fixed hysteresis thresholds.

Given a W×H image, GUD-Canny computes the normalized gradient magnitude, partitions it into 32×H sub-images, and calculates the optimal pair of hysteresis thresholds for each sub-image using Medina-Carnicer's method [11]. Once the hysteresis thresholds are obtained, instead of running one costly multipass CPU-GPU hysteresis process on the entire image, hysteresis thresholdings (one per sub-image, using its specific hysteresis thresholds) are executed entirely on GPU, independently and in parallel. Each thread-block performs the hysteresis process on one sub-image in shared memory, and represents each pixel of the hysteresis map with only one bit to optimize the use of the limited space of shared memory.

Experimental evaluation showed that GUD-Canny only requires 0.35 ms on average to detect edges on 512×512 images. Hence, it fully satisfies real time constraints, and is faster than existing GPU and FPGA implementations.

**Table 4** Statistics of kernels execution time proportions for set $C_{0.01}$

| Kernel | gauss_x (%) | gauss_y (%) | calc_grad (%) | non_max_supp (%) | calc_SM_H (%) | calc_Prob_SM_H (%) | calc_PF_histo (%) | search_thre | hyst (%) |
|---|---|---|---|---|---|---|---|---|---|
| Average | 1.83 | 1.96 | 0.97 | 2.41 | 82.38 | 0.86 | 7.21 | 0.36 | 2.02 |
| Minimum | 1.33 | 1.34 | 0.69 | 1.67 | 79.81 | 0.53 | 4.85 | 0.21 | 1.50 |
| Maximum | 2.80 | 3.17 | 1.20 | 3.21 | 87.13 | 1.16 | 8.81 | 0.59 | 2.86 |

**Table 5** Statistics of kernels execution time proportions for set $C_{0.03}$

| Kernel | gauss_x (%) | gauss_y (%) | calc_grad (%) | non_max_supp (%) | calc_SM_H (%) | calc_Prob_SM_H (%) | calc_PF_histo (%) | search_thre (%) | hyst (%) |
|---|---|---|---|---|---|---|---|---|---|
| Average | 6.10 | 6.57 | 3.25 | 8.03 | 39.39 | 2.90 | 24.87 | 1.19 | 7.70 |
| Minimum | 4.57 | 4.71 | 2.47 | 6.25 | 35.26 | 2.34 | 19.14 | 0.79 | 4.53 |
| Maximum | 8.93 | 10.17 | 4.04 | 10.35 | 44.57 | 3.96 | 29.55 | 1.48 | 12.62 |

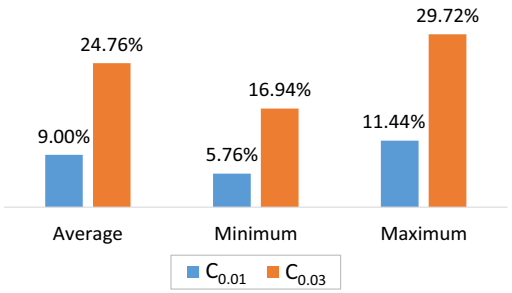


**Fig. 1** Statistics of memory transferences time proportions for sets $C_{0.01}$ and $C_{0.03}$

**Table 6** Edge detection times (ms) for sub-images sizes $32 \times H$ (horizontal partition) and $W \times 32$ (vertical partition)

| Image | $32 \times H$ | $W \times 32$ |
|---|---|---|
| Basket (512×512) | **0.3612** | 0.3914 |
| Beehive (512×512) | **0.3153** | 0.3539 |
| Egg (512×512) | **0.3325** | 0.4000 |
| Feather (512×512) | **0.3607** | 0.4054 |
| Mailbox (512×512) | **0.3932** | 0.4224 |
| Pinecone (512×512) | **0.3770** | 0.3876 |
| Pond (512×512) | **0.3480** | 0.4532 |
| Shopping cart (512×512) | **0.3721** | 0.4282 |
| Tiger (512×512) | **0.3739** | 0.4782 |
| Tire (512×512) | **0.3686** | 0.3922 |
| Turtle (512×512) | **0.2814** | 0.3178 |
| Average | **0.3531** | 0.4028 |
| Minimum | **0.2814** | 0.3178 |
| Maximum | **0.3932** | 0.4782 |

The candidate thresholds set is $C_{0.03}$

For each image, the minimum edge detection time is highlighted in bold

## References


1. Roberts., L.: Machine perception of 3-D solids, optical and electro-optical information processing (1965)
2. Sobel, I., Feldman., G.: A 3 × 3 isotropic gradient operator for image processing. a talk at the Stanford Artificial Project in, 271–272 (1968)
3. Prewitt, J.M.: Object enhancement and extraction. Pict. Process. Psychopictorics **10**(1), 15–19 (1970)
4. Canny, J.: A computational approach to edge detection. IEEE Trans. Pattern Anal. Mach. Intell. **6**, 679–698 (1986)
5. Gonzalez, C.I., Melin, P., Castro, J.R., Mendoza, O., Castillo, O.: An improved sobel edge detection method based on generalized type-2 fuzzy logic. Soft. Comput. **20**(2), 773–784 (2016)
6. Gunawan, T.S., Yaacob, I.Z., Kartiwi, M., Ismail, N., Za'bah, N.F., Mansor, H.: Artificial neural network based fast edge detection algorithm for mri medical images. Indones. J. Electr. Eng. Comput. Sci. **7**(1), 123–130 (2017)
7. ElAraby, W.S., Madian, A.H., Ashour, M.A., Farag, I., Nassef, M.: Fractional edge detection based on genetic algorithm. In 2017 29th International Conference on Microelectronics (ICM) (pp. 1-4). IEEE (2017, December)
8. Dagar, N.S., Dahiya, P.K.: Edge detection technique using binary particle swarm optimization. Procedia Comput. Sci. **167**, 1421–1436 (2020)
9. Sengupta, S., Mittal, N., Modi, M.: Improved skin lesion edge detection method using Ant Colony Optimization. Skin Res. Technol. **25**(6), 846–856 (2019)
10. Dhivya, R., Prakash, R.: Edge Detection Using Adaptive-Neuro-Fuzzy-Interference-System in Remote Sensing Images. J. Comput. Theor. Nanosci. **15**(9–10), 2720–2723 (2018)
11. Medina-Carnicer, R., Munoz-Salinas, R., Yeguas-Bolivar, E., Diaz-Mas, L.: A novel method to look for the hysteresis thresholds for the Canny edge detector. Pattern Recogn. **44**(6), 1201–1211 (2011)
12. Deriche, R.: Using Canny's criteria to derive a recursively implemented optimal edge detector. Int. J. Comput. Vis. **1**(2), 167–187 (1987)
13. Torres, L., Robert, M., Bourennane, E., Paindavoine, M.: Implementation of a recursive real time edge detector using retiming techniques. In Proceedings of ASP-DAC'95/CHDL'95/VLSI'95 with EDA Technofair (pp. 811-816). IEEE (1995, August)

14. Lorca, F.G., Kessal, L., Demigny, D.: Efficient ASIC and FPGA implementations of IIR filters for real time edge detection. In Proceedings of International Conference on Image Processing (Vol. 2, pp. 406-409). IEEE (1997, October)
15. Rao, D.V., Venkatesan, M.: An efficient reconfigurable architecture and implementation of edge detection algorithm using Handle-C. In International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. (Vol. 2, pp. 843-847). IEEE (2004, April)
16. Neoh, H.S., Hazanchuk, A.: Adaptive edge detection for real-time video processing using FPGAs. Global Signal Process. **7**(3), 2–3 (2004)
17. Gentsos, C., Sotiropoulou, C.L., Nikolaidis, S., Vassiliadis, N.: Real-time canny edge detection parallel implementation for FPGAs. In 2010 17th IEEE International Conference on Electronics, Circuits and Systems (pp. 499-502). IEEE (2010, December)
18. He, W., Yuan, K.: An improved Canny edge detector and its realization on FPGA. In 2008 7th World Congress on Intelligent Control and Automation (pp. 6561-6564). Ieee (2008, June)
19. Li, X., Jiang, J., Fan, Q.: An improved real-time hardware architecture for Canny edge detection based on FPGA. In 2012 Third International Conference on Intelligent Control and Information Processing (pp. 445-449). IEEE (2012, July)
20. Peng, F., Lu, X., Lu, H., Shen, S.: An improved high-speed canny edge detection algorithm and its implementation on FPGA. In Fourth International Conference on Machine Vision (ICMV 2011): Computer Vision and Image Analysis; Pattern Recognition and Basic Technologies (Vol. 8350, p. 83501V). International Society for Optics and Photonics (2012, January)
21. Abdelgawad, H.M., Safar, M., Wahba, A.M.: High level synthesis of canny edge detection algorithm on Zynq platform. Int. J. Comput. Electr. Autom. Control Inf. Eng **9**(1), 148–152 (2015)
22. Xu, Q., Varadarajan, S., Chakrabarti, C., Karam, L.J.: A distributed canny edge detector: algorithm and FPGA implementation. IEEE Trans. Image Process. **23**(7), 2944–2960 (2014)
23. Sangeetha, D., Deepa, P.: FPGA implementation of cost-effective robust Canny edge detection algorithm. J. Real-Time Image Proc. **16**(4), 957–970 (2019)
24. Roodt, Y., Visser, W., Clarke, W.: Image processing on the GPU: Implementing the Canny edge detection algorithm. In International Symposium of the Pattern Recognition Association of South Africa (pp. 1-6) (2007, November)
25. Luo, Y., Duraiswami, R.: Canny edge detection on NVIDIA CUDA. In 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (pp. 1-8). IEEE (2008, June)
26. Ogawa, K., Ito, Y., Nakano, K.: Efficient Canny edge detection using a GPU. In 2010 First International Conference on Networking and Computing (pp. 279-280). IEEE (2010, November)
27. Palomar, R., Palomares, J.M., Castillo, J.M., Olivares, J., Gómez-Luna, J.: Parallelizing and optimizing lip-canny using nvidia cuda. In International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (pp. 389-398). Springer, Berlin, Heidelberg (2010, June)
28. Lourenço, L. H., Weingaertner, D., Todt, E.: Efficient implementation of canny edge detection filter for ITK using CUDA. In 2012 13th Symposium on Computer Systems (pp. 33-40). IEEE (2012, October)
29. Vigil, B.M.L.P.: 2015, November. Accelerating the Canny edge detection algorithm with CUDA/GPU, International Congress COMPUMAT (2015)
30. Huang, Y., Bai, Y., Li, R., Huang, X.: Research of Canny edge detection algorithm on embedded CPU and GPU heterogeneous systems. In 2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD) (pp. 647-651). IEEE (2016, August)
31. Mogale, H.: High Performance Canny Edge Detector using Parallel Patterns for Scalability on Modern Multicore Processors. (2017) arXiv preprint arXiv:1710.07745
32. Emrani, Z., Bateni, S., Rabbani, H.: A new parallel approach for accelerating the gpu-based execution of edge detection algorithms. J. Med. Signals Sens. **7**(1), 33 (2017)
33. NVIDIA: CUDA Zone (2021) https://developer.nvidia.com/category/zone/cuda-zone https://developer.nvidia.com/category/zone/cuda-zone
34. Fung, J.: Computer Vision on the GPU. GPU Gems **2**(649–665), 34 (2005)
35. Podlozhnyuk, V.: Image convolution with CUDA. NVIDIA Corporation white paper, June, 2097(3) (2007)
36. Jourlin, M., Pinoli, J.C.: A model for logarithmic image processing. J. Microsc. **149**(1), 21–35 (1988)
37. Palomares, J.M., González, J., Ros, E.: Detección de bordes en imágenes con sombras mediante LIP-Canny. In Memoria del Simposio de Reconocimiento de Formas y Análisis de Imágenes del I Congreso Nacional de Informática. Granada, España. pp (pp. 71-76) (2005)
38. González-Hidalgo, M., Massanet, S., Mir, A., Ruiz-Aguilera, D.: A Comparison Study of Some Configurations of the Uninorm Morphological Edge Detector. In International Conference on Fuzzy Computation Theory and Applications (Vol. 2, pp. 410-419). SCITEPRESS (2012, October)
39. González-Hidalgo, M., Massanet, S.: A fuzzy mathematical morphology based on discrete t-norms: fundamentals and applications to image processing. Soft. Comput. **18**(11), 2297–2311 (2014)
40. González-Hidalgo, M., Massanet, S., Mir, A., Ruiz-Aguilera, D.: A new edge detector based on uninorms. In International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (pp. 184-193). Springer, Cham (2014, July)
41. Gonzalez-Hidalgo, M., Massanet, S., Mir, A., Ruiz-Aguilera, D.: On the choice of the pair conjunction-implication into the fuzzy morphological edge detector. IEEE Trans. Fuzzy Syst. **23**(4), 872–884 (2014)
42. González-Hidalgo, M., Massanet, S., Mir, A., Ruiz-Aguilera, D.: On the generalization of the uninorm morphological gradient. In International Work-Conference on Artificial Neural Networks (pp. 436-449). Springer, Cham (2015, June)
43. González-Hidalgo, M., Massanet, S., Mir, A., Ruiz-Aguilera, D.: On the pair uninorm-implication in the morphological gradient. In Computational Intelligence (pp. 183-197). Springer, Cham (2015)
44. Bibiloni, P., González-Hidalgo, M., Massanet, S., Mir, A., Ruiz-Aguilera, D.: Mayor-torrens t-norms in the fuzzy mathematical morphology and their applications. In Fuzzy Logic and Information Fusion (pp. 201-235). Springer, Cham (2016)
45. González-Hidalgo, M., Massanet, S., Mir, A., Ruiz-Aguilera, D.: Edge image aggregation method using ordered weighted averaging functions. In 2016 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE) (pp. 1355-1362). IEEE (2016, July)
46. Bustince, H., Barrenechea, E., Sesma-Sara, M., Lafuente, J., Dimuro, G.P., Mesiar, R., Kolesárová, A.: Ordered directionally monotone functions: justification and application. IEEE Trans. Fuzzy Syst. **26**(4), 2237–2250 (2017)
47. Sun, G., Zhang, A., Ren, J., Ma, J., Wang, P., Zhang, Y., Jia, X.: Gravitation-based edge detection in hyperspectral images. Remote Sens. **9**(6), 592 (2017)
48. Valero, M.M., Rios, O., Mata, C., Pastor, E., Planas, E.: An integrated approach for tactical monitoring and data-driven spread forecasting of wildfires. Fire Saf. J. **91**, 835–844 (2017)
49. Valero, M.M., Rios, O., Pastor, E., Planas, E.: Automated location of active fire perimeters in aerial infrared imaging using unsupervised edge detectors. Int. J. Wildland Fire **27**(4), 241–256 (2018)

50. Sussner, P., Carazas, L.C.: An Approach Towards Image Edge Detection Based on Interval-Valued Fuzzy Mathematical Morphology and Admissible Orders. In 11th Conference of the European Society for Fuzzy Logic and Technology (EUSFLAT 2019) (pp. 690-697). Atlantis Press (2019, August)
51. Marco-Detchart, C., Bustince, H., Fernandez, J., Mesiar, R., Lafuente, J., Barrenechea, E., Pintor, J.M.: Ordered directional monotonicity in the construction of edge detectors. Fuzzy Sets and Systems (2020)
52. Medina-Carnicer, R., Madrid-Cuevas, F.J., Muñoz-Salinas, R., Carmona-Poyato, A.: Solving the process of hysteresis without determining the optimal thresholds. Pattern Recogn. **43**(4), 1224–1232 (2010)
53. Medina-Carnicer, R., Carmona-Poyato, A., Muñoz-Salinas, R., Madrid-Cuevas, F.J.: Determining hysteresis thresholds for edge detection by combining the advantages and disadvantages of thresholding methods. IEEE Trans. Image Process. **19**(1), 165–173 (2009)
54. Medina-Carnicer, R., Madrid-Cuevas, F.J., Carmona-Poyato, A., Muñoz-Salinas, R.: On candidates selection for hysteresis thresholds in edge detection. Pattern Recogn. **42**(7), 1284–1296 (2009)
55. Hancock, E.R., Kittler, J.: Adaptive estimation of hysteresis thresholds. In Proceedings. 1991 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (pp. 196-201). IEEE (1991, June)
56. Yitzhaky, Y., Peli, E.: A method for objective edge detection evaluation and detector parameter selection. IEEE Trans. Pattern Anal. Mach. Intell. **25**(8), 1027–1033 (2003)
57. NVIDIA: CUDA C Programming Guide (2021) https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
58. NVIDIA: CUDA C Best Practices Guide (2021) https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html
59. Luitjens, J.: "CUDA Pro Tip: Increase Performance with Vectorized Memory Access", (Dec. 2013). https://devblogs.nvidia.com/cuda-pro-tip-increase-performance-with-vectorized-memory-access/https://devblogs.nvidia.com/cuda-pro-tip-increase-performance-with-vectorized-memory-access/
60. Luitjens, J.: "Faster Parallel Reductions on Kepler", (Feb. 2014). https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/
61. NVIDIA: CUDA Math API (2021) https://docs.nvidia.com/cuda/cuda-math-api/index.html https://docs.nvidia.com/cuda/cuda-math-api/index.html
62. Heath, M.D., Sarkar, S., Sanocki, T., Bowyer, K.W.: A robust visual method for assessing the relative performance of edge-detection algorithms. IEEE Trans. Pattern Anal. Mach. Intell. **19**(12), 1338–1359 (1997)




**Antonio Fuentes-Alventosa** received the B.S. and M.S. degrees in Telecommunication Engineering from the University of Málaga, Spain, in 1999, the B.S. degree in Computer Science from the National Distance Education University (UNED), Spain, in 2006, and the M.S. degree in Intelligent Systems from the University of Córdoba, Spain, in 2014. Since 2001, he is software developer at Aplicaciones Informáticas Prosur in Córdoba, Spain. His research interest is in the parallelization and optimization of scientific algorithms on GPUs and heterogeneous systems.

**Juan Gómez-Luna** is a postdoctoral researcher at ETH Zürich. He received the BS and MS degrees in Telecommunication Engineering from the University of Sevilla, Spain, in 2001, and the PhD degree in Computer Science from the University of Córdoba, Spain, in 2012. Between 2005 and 2017, he was a lecturer at the University of Córdoba. His research interests focus on Processing-in-Memory, GPUs and heterogeneous systems, medical imaging, and bioinformatics.

**R. Medina-Carnicer** received the B.S. degree in Mathematics from University of Sevilla, Spain, and the Ph.D. degree in Computer Science from the Polytechnic University of Madrid, Spain, in 1992. Since 1993, he has been a lecturer of Computer Vision at Cordoba University, Spain. His research is focused on Edge Detection, 3-D Vision, Augmented Reality and Pattern Recognition.