Speeding up LIP-Canny with CUDA programming

Rafael Palomar, José M. Palomares, Joaquín Olivares, José M. Castillo, and Juan Gómez-Luna [1]

Resumen— **The LIP-Canny algorithm outperforms traditional Canny edge detection in terms of edge detection under varying illumination. This method is based on a robust mathematical model (LIP paradigm), which is closer to the human vision system. However, this model requires more computations and more complex operations than the traditional paradigm. Non–parallel implementations of LIP–Canny do not fit Real-Time requirements because of the large amount of operations required. NVIDIA CUDA is a platform which enables the parallelization of this algorithm, obtaining very high performance. In this work, a comparison results between the non–parallel implementation (written in C/C++) and the NVIDIA CUDA one.**

Palabras clave— **NVIDIA CUDA, LIP-Canny, Speedup**

I. Introduction

Edge extraction is a very relevant stage for any visual processing–related algorithm, because it is usually the first step and any further processing relies heavily on them. Gradient map by image filtering is the most frequently method used to obtain the edges.

The *Canny* algorithm [1] is one of the most effective edge detector. This algorithm outperforms other algorithms, such as the Sobel edge detector [2], [3], when dealing with (gaussian) noisy images.

The *Logarithmic Image Processing* (LIP) is a robust mathematical model with several desirable properties for computer vision processing: operational results within a range, follow many human vision system laws, etc. In the work of Palomares *et al.*[4], a Canny algorithm based on LIP, namely LIP–Canny, is proposed. This new approach provides better edge maps than the traditional Canny algorithm (especially for images with shadows), although it is slower than the original one.

Regarding the problem of performance improvement, researchers are paying special attention to to GPGPU (General Purpose Graphic Processing Units) processing, that allows the execution of conventional algorithms in specialized graphic processing hardware. This technology has evolved rapidly and since NVIDIA developed CUDA, the interest around this technology has grown in many research fields. Examples of this fact could be the works of Riegel *et al.*[5] in fluid mechanics and Kavanagh *et al.*[6] in astrophysics.

This work pretends to cover the LIP-Canny algorithm under the perspective of its parallelization and optimization to archieve improvements of performance. It is based on Palomares *et al.*[4] and Luo and Duraiswami[7], and can be integrated as part of more complex algorithms that use an edge detection stages like Gómez-Luna *et al.*[8].

[1]Department of Computer Architecture, Electronics and Electronic Technology. University of Córdoba. Corresponding author:`jmpalomares@uco.es`

II. The LIP Model

In the late 80's and early 90's Jourlin and Pinoli[9], [10] exhibit a new paradigm for the processing of images using the projection of light (as in the case of microscope images). The aim was to solve common problems like the overflow that could be produced when two images are added. This new approach was named LIP (*Logarithmic Image Processing*).

Jourlin and Pinoli describe new mathematical operators to perform the addition and amplification of microscope images. These operators not only perform the operations that they were designed for, but they also preserve the physical nature of the images to be processed.

A. Definitions, naming conventions and ranges

Before starting to detail the underliying mathematical model of LIP, it is convenient to describe the ranges and the meaning of some variables.

M is referred to as the upper limit of the possible values of a pixel could have, which depends on the image depth. For instance, an 8-bit image (grayscale) would use an M value (unreachable) of 256 and a minimum value of 0 (Note that absolute darkness and absolute brightness do not occur in the real world). Gray-level images are denoted as capitalized letter such as I, J, etc. Gray-level functions are denoted as lowercase letters such as f where $f \in (0, M)$, while the *gray tone* of a gray-level function is denoted as the corresponding letter with a hat and in the case of f it is defined as $\hat{f} = M - f$. Finally, the LIP transformed gray tone function will be denoted as the corresponding lowercase letter with a tilde such as $\tilde{f}$.

B. A Brief outline of LIP

There are two possible ways to apply the LIP model to any digital image processing technique. In this section, we include a brief introduction to each of these methods.

B.1 Operating images in LIP space with traditional operators.

The first method comes from the tranformation of the "original" image (image expressed in natural space) to an special space called LIP. This is carried out by an *isomorphic transform* defined by

$$\tilde{f} = \varphi(\hat{f}) = -M \cdot \log\left(1 - \frac{\hat{f}}{M}\right) \qquad (1)$$

Once the corresponding processing technique is applied, the image has to be transformed through an *inverse transformation* defined by

$$\hat{f} = \varphi^{-1}(\tilde{f}) = M \cdot \left(1 - \exp\left(-\frac{\tilde{f}}{M}\right)\right) \qquad (2)$$

B.2 Operating images in linear space with modified operators.

The second method operates directly on the images expressed in natural space. For this purpose, we define:

- A new image with an inversion of the scale ($\widehat{I} = M - I$).
- A special sum operator $\triangle\!\!\!\!+$ defined as $\widehat{f} \triangle\!\!\!\!+ \widehat{g} = \widehat{f} + \widehat{g} - \frac{\widehat{f} \cdot \widehat{g}}{M}$
- A special scalar product operator $\triangle\!\!\!\!\times$ defined as $\alpha \triangle\!\!\!\!\times \widehat{f} = M - M \cdot \left(1 - \frac{\widehat{f}}{M}\right)^{\alpha}$
- Further operators have been proposed, for example, the LIP-Difference $\triangle\!\!\!\!-$, which permits this paradigm to be an algebraic linear space.

III. CUDA Parallel Processing Foundations

A common trend in the videogames industry and more concisely into the graphics industry, is to develop architectures based on processors capable to process a great amount of data parallely. These "graphics" processors were originally designed to rapidly transform complete 3D scenes into 2D graphics. Obviously, in an architecture especially designed for such a purpose, the tasks performed are specialized, but recent tests have demonstrated that these architectures have obtained processing speeds of more than one order of magnitude compared with CPUs. Nowadays new technologies such as NVIDIA CUDA provides the developers with tools for programming algorithms for their execution in graphic architectures, thus obtaining higher performance rates than the analogous algorithms executed on CPUs.

A. Architecture

The architecture is based on special processors called *streaming multiprocessors* (SMs) capable of keeping a huge number of threads running parallely, which allows developers to easily implement data-parallel kernels. Each SM contains a set of *streaming processors* (SPs), 32-bit units especially designed to perform profitable multiply-add operations of single-precision floating-point numbers. In addition the SMs have two special function units (SFUs) that execute more complex operations such as square root, sine, cosine, etc.

Because of SMs' high demand of data the system memory is divided physically in different spaces. The GPU has a *global memory* shared by all the processing units (SPs) whereas each SM has a small on-chip *shared memory* used by all the SPs of the SM. For read-only data that is accessed simultaneously by many threads there is a *constant memory* per SM[1] and a *texture memory* shared by clusters of SMs.

B. Execution Model

The CUDA execution model fits around two execution modes. Certain pieces of code of a CUDA program would be executed sequentially by the CPU while another code would do it as kernels executed parallely by the GPU.

The kernels are organized in a two-level hierarchy, threads in the lower level that maps directly with the SPs and block of threads in the higher level that map directly with the SMs. In order to feed the execution threads the data space will be divided in a SIMD-like style.

The maximum efficiency of the kernels can be reached when the occupancy of the SMs is as high as possible, the memory hierarchy is properly used and the correct access patterns for the different memory spaces is preserved. For further details about the architecture, execution model, access patterns and optimizations see [11] and [12].

IV. Implementing LIP-Canny with CUDA

The Canny algorithm and its variants that use LIP present certain features that enables its parallelization on GPUs. A common strategy to implement image processing algorithms using CUDA is to separate the implementation in kernels that individually, load the data in *shared memory*, perform some operations on the data and finally store the results back into the *global memory*.

In this section we present both the changes needed for applying LIP to the Canny algorithm as well as some additional changes applicable to the Canny algorithm, with regard to Luo[7].

A. LIP-Canny using a traditional operators' approach

As mentioned previously, using traditional operators to implement LIP-Canny requires a direct transformation to LIP space, processing of the image and a inverse transformation to linear space. The first intuitive approximation is to implement two separate kernels, one for the direct transformation operating before the filtering stages and another for the inverse transformation operating before the gradient computation stage. In order to avoid undesirable memory transferences between *global* and *shared* memories, each kernel must perform all the operations possible when the data is in *shared memory*, therefore some kernels have to be merged. Table I shows the kernels

[1]Altough this memory exist per SM, the content is common to and replicated for all SMs.

used in our implementation and their development nature.

TABLE I
Kernels for LIP-Canny using a traditional operators' approach

`Kernel`	`Development`
`Loading data + LIP Transform`	`Custom`
`Gaussian Row Convolution`	`Based on [13]`
`Gaussian Column Convolution`	`Based on [13]`
`Derivative Gaussian Row Convolution`	`Based on [13]`
`Derivative Gaussian Column Convolution`	`Based on [13]`
`Gradient Mag./Dir. Computation + Inverse LIP Transform`	`Custom`
`Non-max. Suppression`	`Based on [7]`
`Hysteresis`	`Based on [7]`
`Unloading Data`	`Based on [7]`

A.1 Loading Data and Performing LIP Transformation Kernel.

Usually the images are given in a 1-byte per pixel (gray-scale) format. As mentioned previously, the processing units are especially designed to perform 32-bit floating-point operations, hence these image data must be represented in a more convenient format. Luo[7] presents a conversion technique based on logical *AND* and shift operations while mantaining the correct memory access patterns, in addition we have included the LIP transformation operations based on the Equation 1.

A.2 Filtering Operations.

The filtering operations widely differ from the Luo[7] proposal. While the authors have used fixed length filters we propose a pseudo-parameterization which allows the processing to benefit from a variable filter length. This pseudo-parameterization establishes a wide filter length and fills it with the corresponding centered gaussian or derivative gaussian function, which avoids some of the calculations needed for pure variable length filters. Fig. 1 shows examples of this pseudo-parameterization. Regarding the convolution operation, Podlozhnyuk[13] offers an efficient approach that has been used in this work.

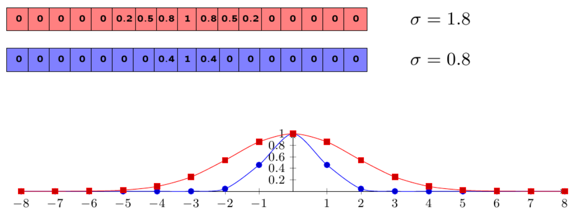


Fig. 1. Pseudo-parameterized kernel generation.

A.3 Gradient Magnitude and Direction Computation and Inverse LIP Transformation.

This kernel arises from including of the inverse LIP transformation operations in the gradient and direction computation processes. The philosophy of the kernel is the same as that of the rest of the kernels, that is, loading from *global* to *shared* memory, processing and storing data back into the *global* memory. Luo[7] uses the classical magnitude of the gradient equation and a discretization for the direction, but we have included the inverse LIP transformation for each vector component, according with the Equation 2.

B. *LIP-Canny using a modified operators' approach*

Using modified operators requires a new convolution composed from the LIP operators previously introduced. However, Palomares et al.[14] demonstrate that it is possible to obtain a new operation $conv2D_{\triangle}$ from traditional separable filter convolution operations and by using "transformed" images. This new operation is equivalent and more efficient than the pure LIP operators' approach. Furthermore, the authors establish a new operation $grad_{\triangle}$ that requires the definition of $conv2D_{\triangle}$ in terms of its LIP transformation, as shown in Equation 3.

$$\varphi(conv2D_{\triangle}(\hat{I},F)) = M \cdot \big(K \cdot \log M - conv1D\left(conv1D\left(lnI, \mathbf{a^T}\right), \mathbf{b}\right)\big) \qquad (3)$$

where $K = \sum_{j=0}^{m-1} \mathbf{a(m-j)} \cdot \sum_{\mathbf{i=0}}^{\mathbf{n-1}} \mathbf{b(n-i)}$. We can keep the derivative filtering operations invariant to the *Traditional Operators* approach, but it is necessary to obtain a new "transformed" image resulting from the application of the logarithm to the original image as well as that of performing gaussian filtering as seen Equation 3. As in the *Traditional Operators* approach, we take advantage of this in order to add some processing in the load data kernel and the gaussian filtering (y) kernel. Below, we further develop these changes.

TABLE II
Kernels for LIP-Canny using a modified operators' approach

`Kernel`	`Development`
`Loading data + LIP Transform`	`Custom`
`Gaussian Row Convolution`	`Based on [13]`
`Gaussian Column Convolution`	`Custom`
`Derivative Gaussian Row Convolution`	`Based on [13]`
`Derivative Gaussian Column Convolution`	`Based on [13]`
`Gradient Mag./Dir. Computation + Inverse LIP Transform`	`Custom`
`Non-max. Suppression`	`Based on [7]`
`Hysteresis`	`Based on [7]`
`Unloading Data`	`Based on [7]`

B.1 Loading Data and Getting the Logarithm Image.

As we explained in the analogous kernel for the traditional operators' approach, the data must be converted into a more convenient format. The difference introduced in this kernel is the application of the logarithm operation to all the pixels of the original image, which we have included in the loading data kernel.

B.2 Gaussian Column Convolution

Concerning the $conv2D_\triangle$, we have used the Podlozhnyuk[13] convolution for the row filtering while some modifications are introduced for the column filtering. These modifications consist of the addition of the extra operations needed to perform $\varphi(conv2D_\triangle)$ as seen in the Equation 3.

B.3 Gradient Magnitude and Direction Computation and Inverse Transformation.

Here we apply the same principles that lead the gradient magnitude and direction computation previously explained in the *Traditional Operators* approach. This time, the magnitude computation is performed using the $grad_\triangle$ operator defined as:

$$grad_\triangle(\hat{\mathbf{g}}) = M \cdot \left(1 - \exp\left(-\frac{\sqrt{\varphi(\hat{g_x})^2 + \varphi(\hat{g_y})^2}}{M}\right)\right) \tag{4}$$

V. Results

In order to evaluate the performance of the different proposals, the two approaches previously described have been implemented using the NVIDIA CUDA framework for their execution on GPU just as we are using C/C++ for their execution on CPU. The hardware platform used for the CPU execution is *Intel P8400 (2.26GHz), 4GB RAM DDR2* while the hardware used for the GPU execution is *NVIDIA 9200GS, 512GB global memory.*

A maximum length filter of 15 for the pseudo-parameterized filtering process has been used with $\sigma = 1$. As Luo[7] states, the implementation of hysteresis using NVIDIA CUDA has to be iterated sometimes. We did not experimented a significant improvement over 4 iterations for our test images, hence we fixed this number of iterations.

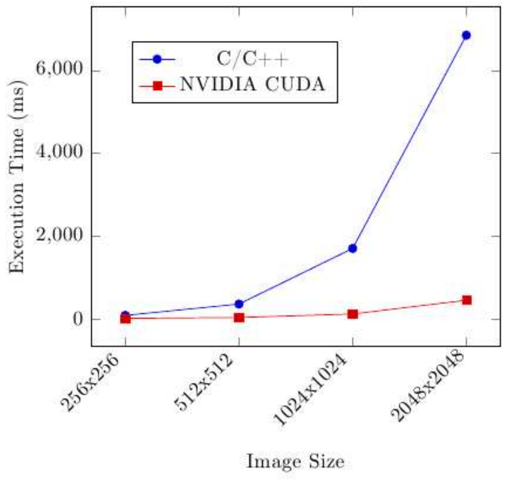
The experimentation process consisted of 100 executions per implementation. The results obtained are expressed in terms of average time in milliseconds, *speedup* and time consumption percentage.

A. Comparing NVIDIA CUDA and C/C++ implementations.

Fig. 2 and 3 exhibit the average execution times for all the implementations. The NVIDIA CUDA implementations behave better in terms of execution time compared to the C/C++ approaches. Generally, the *speedup* grows as the image size becomes larger.

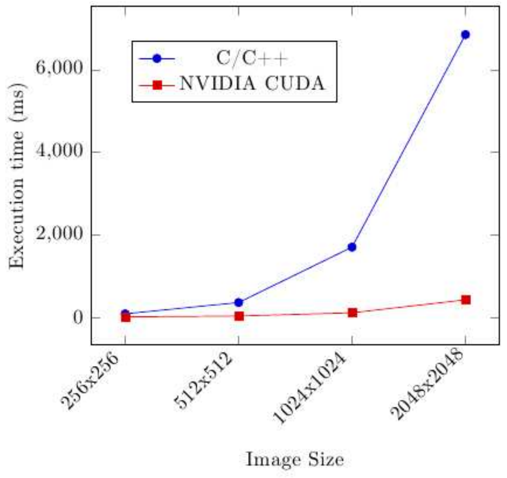
The *Modified Operators* approach give more efficient implementations. Fig. 4 shows the *speedup* obtained upon comparing this approach with the *Traditional Operators* approach, revealing that the NVIDIA CUDA implementation makes a better exploitation of the *Modified Operators* approach than C/C++.

The behaviour of the *speedup* curve is different for the NVIDIA CUDA implementation than for the C/C++ implementation. The former decreases as the image size increases while the latter slightly increases with the image size.



Size Image	Ex. Time C/C++	Ex. Time CUDA	*Speedup*
256x256	80.200	7.599	10.553
512x512	353.000	28.848	12.236
1024x1024	1696.000	113.288	14.970
2048x2048	6846.400	445.168	15.379

Fig. 2. Results for the *Traditional Operators* approach.



Size Image	Ex. Time C/C++	Ex. Time CUDA	*Speedup*
256x256	79.000	6.786	11.640
512x512	346.200	26.448	13.089
1024x1024	1660.600	104.533	15.885
2048x2048	6704.100	418.563	16.016

Fig. 3. Results for the *Modified Operators* approach.

B. Execution time distribution using NVIDIA CUDA.

As mentioned previously, the NVIDIA CUDA executions are distributed around kernels. Figs. 5 and 6 show the distribution for LIP-Canny using both approaches for a 512 × 512 input image. The distribution of the kernels may vary slightly for different image sizes, but the order of time consumption remains invariant except for the memory transfers. The results exhibit similar distributions for the

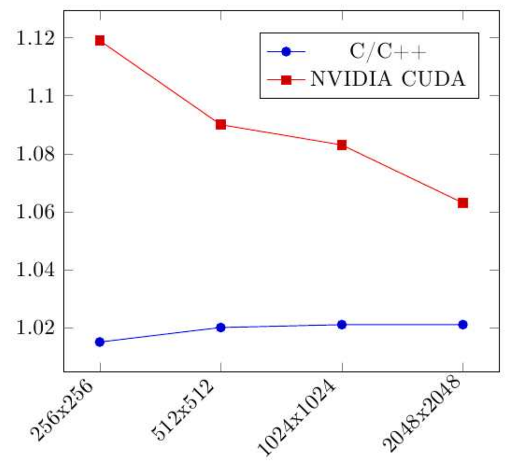


Image Size	C/C++	CUDA
256x256	1.015	1.119
512x512	1.020	1.090
1024x1024	1.021	1.083
2048x2048	1.021	1.063

Fig. 4. *Speedup* for the *Modified Operators* approach compared to the *Traditional Operators* approach.

different LIP-Canny implementations using NVIDIA CUDA. Hysteresis takes a special importance due to its high time consumption.

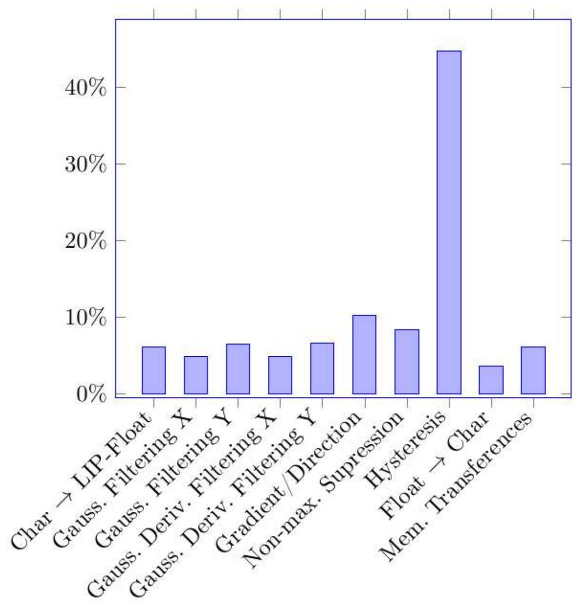


Fig. 5. Distribution for the *Traditional Operators* approach.

VI. Conclusions

In this work, we have presented two different approaches to implement LIP-Canny using NVIDIA CUDA, one through traditional operators applied to modified images and another through modified operators applied to traditional images.

Comparative results between C/C++ implementations and NVIDIA CUDA implementations are presented. These results show that NVIDIA CUDA implementations are faster (aproximately 10 to 16 times) than the analogue using C/C++. Furthermore, as occurs in C/C++ implementations, LIP-Canny using modified operators is faster than LIP-Canny through traditional operators, when based on NVIDIA CUDA. The *speedup* associated with the *Modified Operators* approach compared with the *Traditional Operators* approach is better exploited

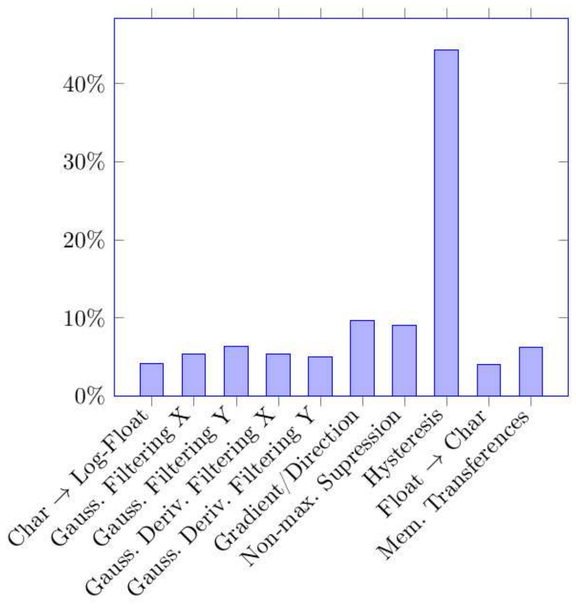


Fig. 6. Distribution for the *Modified Operators* approach.

by the NVIDIA CUDA implementation than by the C/C++ implementation.

The results show that the hysteresis process is the most expensive kernel in terms of execution time due to its iterative nature (hysteresis process was repeated 4 times). For further efforts to optimize the approaches presented in this work, the hysteresis process could have to be considered carefully, given the relevance of the process in terms of execution time.

The impact of the distribution when choosing another number of iterations will be significant, making the existing difference between the hysteresis kernel an the other kernels grow upon increasing the number of iterations, obtaining thus the opposite effect when decreasing this number which is highly dependent on the image to be processed.

Referencias

[1] John Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 679–698, Nov. 1986.

[2] Irwin Sobel and G. Feldman, “A 3x3 isotropic gradient operator for image processing,” Talk at the Stanford Artificial Project, 1968.

[3] Irwin Sobel, *Camera Models and Machine Perception*, Ph.d. thesis, Standford University, 1970.

[4] José M Palomares, Jesús González, and Eduardo Ros, “Detección de bordes en imágenes con sombras mediante LIP–Canny,” in *AERFAI 2005*, 2005.

[5] E. Riegel, T. Indinger, and N.A. Adams, “Implementation of a Lattice-Boltzmann method for numerical fluid mechanics using the nVIDIA CUDA technology,” 2009.

[6] G.D. Kavanagh, M.C. Lewis, and B.L. Massingill, “GPGPU planetary simulations with CUDA,” in *Proceedings of the 2008 International Conference on Scientific Computing, CSC 2008*, 2008, pp. 180–185.

[7] Y.M. Luo and R. Duraiswami, “Canny edge detection on NVIDIA CUDA,” in *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops*, 2008.

[8] Juan Gómez-Luna, José González-Linares, José Benavides, and Nicolás Guil, “Parallelization of a video segmentation algorithm on CUDA–Enabled graphics processing units,” in *Euro-Par 2009 Parallel Processing*, pp. 924–935. 2009.

[9] Jourlin and Pinoli, “A model for logarithmic image processing,” *Journal of Microscopy,*, vol. 149,, pp. 21–35, 1988.

[10] J.C. Pinoli, “The logarithmic image processing model: Connections with human brightness perception and contrast estimators,” *Journal of Mathematical Image Processing*, vol. 7,, pp. 341–358., 1997.

[11] NVIDIA, *NVIDIA CUDA Programming Guide*, 2009.

[12] NVIDIA, *NVIDIA CUDA C Programming Best Practices Guide. CUDA Toolkit 2.3*, 2009.

[13] Victor Podlozhnyuk, *Image Convolution with CUDA*.

[14] José M Palomares, Jesús González, Eduardo Ros, and Alberto Prieto, “General logarithmic image processing convolution,” *IEEE Transactions on Image Processing*, vol. 15, no. 11, pp. 3602–3608, Nov. 2006, ISSN: 1057–7149.