Noname manuscript No.
(will be inserted by the editor)

A Non-Smooth, Non-Local Variational Approach to Saliency Detection in Real Time

Eduardo Alcaín · Ana I. Muñoz · Emanuele Schiavi · Antonio S. Montemayor

https://link.springer.com/article/10.1007/s11554-020-01016-4

Received: date / Accepted: date

Abstract In this paper, we propose and solve numerically a general non-smooth, non-local variational model to tackle the saliency detection problem in natural images. In order to overcome the typical drawback of the non-local methods in image processing, which mainly is the inherent computational complexity of non-local calculus, as the non-local derivatives are computed w.r.t every point of the domain, we propose a different scenario. We present a novel convex energy minimization problem in the feature space, which is efficiently solved by means of a non-local Primal-Dual method. Several implementations and discussions are presented taking care of the computing platforms, CPU and GPU, achieving up to 33 fps and 62 fps respectively for 300×400 image resolution, making the method eligible for real time applications.

Keywords Variational Methods, Convex, Primal-Dual, Non-local Image Processing, Saliency Segmentation, GPU, Superpixels

1 Introduction

Saliency object segmentation refers to an image processing system which aims to emulate the human visual attention system extracting the most relevant information of a scene. Saliency methods are usually applied as a pre-processing step in different areas in computer vision: adaptive compression of images [41], image retrieval [18] and content-aware resizing [6] and can be divided into bottom-up (pre-attentive data driven) and top-down (task dependent). In this work, we shall focus on bottom-up, image stimulus-driven models of attention. They are task-free and do no rely on learning, training or contextual information. They can also be considered as a fundamental building block for advanced, robust, hybrid bottom-up and top-down models.

Several algorithms and methods have emerged in the classical image processing field for the saliency detection problem like [3][13], machine learning approaches [40][23], variational methods [29][33] and combined [19][24].

In Itti et al. [25], it is determined the saliency map using center-surround operations on colour, intensity and orientation features using a Difference of Gaussians (DoG) approach in a multiscale framework. In [3] the authors use the $L*a*b$ colour space, subtracting the image mean colour to each of the components and producing the saliency map after a meanshift segmentation and a dynamic thresholding.

As a segmentation procedure, saliency detection methods can make use of graph-based segmentation techniques [26][10][15]. The starting point is an oversegmented image in which regions are progressively merged using different criteria or features, such as colour or contrast. The oversegmentation task can rely on fine-grained morphological operators such as the watershed segmentation algorithm [26][15] or more recent superpixels approach [27]. In [13] authors propose a histogram based contrast (HC) procedure to measure the saliency of pixels as well as a region based contrast (RC) metric for regions obtained after a graph partitioning algorithm application.

The application of modern machine learning methods has achieved very good results making full use of transfer learning techniques [40][23]. However, being com-

Eduardo Alcaín, Ana I. Muñoz, Emanuele Schiavi, Antonio S. Montemayor
C/Tulipán, S/N, 28933 Móstoles, Spain
Tel.: +34-914887190
E-mail: e.alcain@alumnos.urjc.es, {anaisabel.munoz, emanuele.schiavi, antonio.sanz}@urjc.es

putationally very intensive methods, they perform very slow even using modern hardware (about 1 fps) to produce saliency maps and they suffer from images without clear objects.

Variational methods have achieved great success when applied in many low-level image processing problems [36][12][17]. They can also accomplish more specific tasks, such as saliency segmentation, but new dynamics have to be introduced in order to facilitate the bottom-up (local analysis structure) variational approach. Some works have been published making use of the variational setting for this task. A non-local, pixel-wise convex model is proposed in [29] to segment natural images, and another one, non-convex, is proposed in [33] to detect and segment glioblastomas (tumors) in MRI images. In [37], we can find another non-local non-convex model, where it is considered the minimization of the L_0 semi-norm using superpixels and solving the minimization with the ADMM (Alternative Direction Method of Multipliers). In this case, the output has to be binarized. This model is further discussed and compared in [5].

Based on the framework considered in [5], we divide the image into regions (*superpixels*). This oversegmentation changes the domain of our problem, from pixels to superpixels (manifold). In this domain, we construct a variational model to solve the saliency segmentation (see Fig. 1).

In order to induce the binarization in the final output, we include a new *saliency* term in the model which pushes superpixels towards a binary classification (salient or no salient). The consideration of this saliency term is inspired in a simplified Ginzburg -Landau phase-transition model [21]. To precise, saliency is modelled by means of the introduction of a reactive term in the Euler-Lagrange equation which is solved to minimize a new energy functional defined in the manifold of characteristics. The numerical resolution is performed by a non local Primal-Dual algorithm, which proves to be more efficient than a naïve gradient descent algorithm or the ADMM used in [37] for a non-convex formulation.

It is well known that variational methods are computationally expensive tasks due to the iterative nature of their solvers. Iterative algorithms are not very parallel-friendly, due to the fact that the result of one iteration is used in the next one. Since the introduction of GPUs to the general purpose computation field (GPGPU), some works have tried to explore the parallelism of modern GPUs using different strategies and solvers like [35][32] and more recently, in [22], using a more general GPU computing framework such as NVIDIA CUDA. Our final contribution is a fast implementation

(a) (b) (c) (d) (e)

Fig. 1: The workflow of our proposal consists of: (a) input image, (b) oversegmentation by superpixels (SLIC), (c) weights to connect each supperpixel among them. A superpixel (yellow) is shown and its k-neighbours in cyan, (d) control map created from the superpixels by means of colour contrast and location priors, (d) variational method and saliency map.

in CPU and GPU, demonstrating the computing power of those platforms for achieving real time performance.

Overall, the contributions in this paper are summarized as follows:

- We propose a novel, general, total variation (TV) based variational model in graphs is proposed. It improves the visual quality of the segmentation and includes a new energy saliency term $H(u)$ which drives the dynamics in the manifold.
- We consider a non-local Primal-Dual algorithm for the resolution of the resulting saliency model in a convex scenario which assures well-posedness of the model.
- We present a performance study for achieving a fast solution for the variational problem using CPU optimizations and GPU computing using NVIDIA CUDA.

The paper is organized as follows: we introduce our proposal for the saliency segmentation in a non-local variational framework in section 2. Section 3 gives a brief introduction to CPU and GPU parallel computing. The description of our implementation and optimizations are discussed in the section 4. Experimental results are presented in section 5 and finally, some conclusions are drawn in section 6.

2 Variational approach for saliency detection

The pipeline of our method is summarized in Fig. 1 and presents two conceptual stages: 1) **initialization stage** with the superpixels extraction (see Fig. 1b), calculation of the weights (see Fig. 1c) and generation of a control map which models the a priori likelihood of being salient (see Fig. 1d), and 2) **iterative stage**, with the variational solver for the saliency segmentation (see Fig. 1e). First we shall present the model to better understand the preliminary calculations.

2.1 A general variational model for saliency

A general variational model for saliency segmentation can be formulated as the following energy minimization problem: Given a function f (the data), compute a solution u in the image domain such that it minimizes the energy

$$E(u) = J(u) + \lambda F(u) - H(u) \tag{1}$$

where $J(u)$ is the non-local Total Variation operator, [20], $F(u)$ is the likelihood or fidelity term and $H(u)$ is a saliency term which promotes the binarization of the solution into salient (foreground) and not salient (background) regions.

This model can be formulated pixel-wise, in the image domain or in a manifold of featured superpixels, the region domain. The second option provides a dimensional reduction of the problem. We follow this approach with the aim to reduce the computational time of the whole process yet preserving the fundamental information of the image, i.e edges, through the superpixel partition.

2.2 Over-segmentation: superpixels

We briefly describe our setting. The input colour image f is first transformed from RGB into CIELAB colour space, which is perceptually more interesting for the independence of colour and intensity [13]. Then the image is partitioned into regions (*superpixels*). A superpixel is a cluster of pixels connected by some metric. Each pixel is assigned only to one superpixel so they do not overlap one each other. The finite union of all these regions allows to recover the image domain in a sort of over-segmented image provided by the partition. In our approach the initial partition is created by using the SLIC method (*Simple Linear Iterative Clustering*) [1] (see Fig. 1b) which has been proven to be accurate, computationally efficient and robust.

2.3 Weights: adjacency matrix W

Once the image has been partitioned, the spatial structure in the pixel domain is lost. A finite dimensional manifold can be constructed using the initial data f in the image domain. Let $\mathbf{f_p} = \mathbf{f}(p)$ be the vector value in the features space at a superpixel p which we identify with a vertex of a graph (un-directed, symmetric and weighted) $G = (V, E)$ where V is the set of vertexes and E the set of edges. The connections $pq \in E$ between superpixels are defined in the feature space by the weight function

$$\omega_{pq} = \exp\left(-\frac{\|\mathbf{f_p} - \mathbf{f_q}\|^2}{2\sigma^2}\right) \tag{2}$$

where $\mathbf{f}_p$ is a feature vector at superpixel p defined by $\mathbf{f}_p = (\alpha \mathbf{c}_p, \mathbf{l}_p)$, being $\mathbf{c}_p \in I\!R^3$ the mean of the superpixels for each component in the CIELAB colour space and $\mathbf{l}_p \in I\!R^2$ is the centroid position of the superpixel in the original spatial space. The parameter α controls the balance between the two features ($\alpha = 0.9$). Superpixels p and q are connected, say $p \sim q$ iff $w_{pq} > 0$. All the superpixels are initially connected. The graph G is represented by its adjacency matrix $W^{sp \times sp} = (\omega_{pq})_{pq \in E}$, where sp is the number of superpixels. In order to reduce computational cost and to exploit local relationships in the feature space, the number of total connections (edges of the graph) of each superpixel is decreased from sp to k-nearest neighbours (see Fig. 1c). The rest of the weights for a superpixel are set to zero. As a result of this process, we end up with a sparse weight matrix $W^{sp \times k}$ which is no longer symmetric.

2.4 Regularizer, Fidelity and Saliency terms

We fix the notation which is slightly adapted from [16]. Let $\mathbf{u}$ be a function on the set of vertices V in G representing the solution in the features domain and let $\partial_q u_p$ be the *weighted partial difference* at a vertex p in the direction of vertex q:

$$\partial_q u_p = \sqrt{w_{pq}}(u_p - u_q)$$

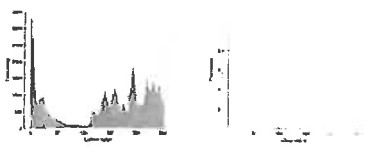
where u_p is the value at superpixel p and $\mathbf{u} = (u_p)_{p \in V}$. The *weighted gradient operator* as the vector of all partial differences at superpixel p:

$$\nabla_w u_p = (\partial_q u_p)_{q \in V} \tag{3}$$

Collecting all the contributions we have that $\nabla_\omega \mathbf{u} = (\nabla_w u_p)_{p \in V}$ is a matrix. In order to compute the Euler-Lagrange equation of 1 the non-local divergence div_w of a vector $\mathbf{d}$ is introduced:

$$div_\omega \mathbf{d}(p) = \sum_{p,\, qp \in E} (\mathbf{d}(p,q) - \mathbf{d}(q,p))\sqrt{w_{pq}} \tag{4}$$

We can now describe the nonlocal operators and energy terms in the features domain corresponding to 1. Our regularizer $J(u)$ is a semi-norm on graphs defined



(a) (b) (c)

Fig. 2: Our saliency algorithm with 300 superpixels and convergence criteria $\epsilon = 10^{-5}$ between the energy of two consecutive iterations has been applied to the Fig. 1 where (a) colour histogram of the input image, (b) represents the histogram of the saliency map and (c) profile of the energy of $NLTVSalTerm$.

by the non-local total variation operator, which preserves edges and induces the sparsity of the gradients of saliency maps.

Following [16], the NLTV norm in its discrete version can be defined as the isotropic L_1 norm of the weighted graph gradient

$$J_{NLTV,w}(\mathbf{u}) = \sum_{p\in G}\left(\sum_{q,\,pq\in E} w_{pq}|u_q - u_p|^2\right)^{1/2} \tag{5}$$

so we choose $J_{NLTV,w}(\mathbf{u})$ as a regularizer. The fidelity term is defined as in [37]. Using the data f, we compute a *control map* $\mathbf{v}^c$ where $\mathbf{v}^c = (v_p^c)_{p\in V}$ (see Fig. 1d). Each component is composed of a **contrast prior**

$$v_p^{con} = \sum_{q\neq p}\omega_{pq}^{(l)}||\mathbf{c}_p - \mathbf{c}_q||^2, \quad \omega_{pq}^{(l)} = e^{-\frac{||\mathbf{l}_p - \mathbf{l}_q||^2}{2\sigma^2}} \tag{6}$$

and an **object prior**

$$v_p^{obj} = e^{-\frac{||\mathbf{l}_p - \bar{\mathbf{l}}||^2}{2\sigma^2}}, \tag{7}$$

where $\bar{\mathbf{l}}$ are the coordinates of the center of the image. The parameter σ^2 is empirically set to 0.05. We can encode these priors in the saliency control map at superpixel p as $v_p^c = v_p^{con}v_p^{obj}$ with $\mathbf{v}^c = (v_p^c)_{p\in V}$ and define the fidelity term as

$$F(\mathbf{u}) = \frac{\lambda}{\alpha}||\mathbf{u} - \mathbf{v}^c||_2^2 = \frac{\lambda}{2\alpha}\sum_{p\in V}|u_p - v_p^c|^2 \tag{8}$$

where the positive parameters λ, α model the trade off between regularization and likelihood.

We model the saliency term with a concave quadratic energy function $-H(u)$, where:

$$H(\mathbf{u}) = \frac{1}{2\alpha^2}\sum_{p\in V}(1 - \delta u_p)^2, \tag{9}$$

with $\delta > 0$ acting as a threshold. Without no mechanism to keep the solution in the (saliency) range $[0, 1]$, the reaction and forcing terms in the Euler-Lagrange equation can produce (depending on the parametric values) changing sign solutions with very high absolute value. We then perform an hard truncation to fulfill the constraint. It models the probability of each superpixel to be salient.

2.5 Numerical resolution: Primal-Dual algorithm

Replacing the terms in the Eq. 1 by their analogous Eq. 5, 8 and 9 our proposal is the minimization of the following energy:

$$\min_{\mathbf{u}} \quad J_{NLTV,w}(\mathbf{u}) + F(\mathbf{u}) - H(\mathbf{u}) \tag{10}$$

The solution $\mathbf{u}$ is calculated on the superpixel domain and must be projected back onto the pixel domain in order to create the final output u in the image domain (see Fig. 1c).

The resolution of the minimization problem (Eq.10) is achieved by a Primal-Dual algorithm. This algorithm encompasses an alternate *maximization* (update dual variable $\mathbf{d}$) and *minimization* (update primal variable $\mathbf{u}$) steps [11] [28]. Both steps are repeated until the energy convergence is reached. As a stopping criteria, we compute the difference between consecutive values of the energy functional (Eq. 10) during iterations until this is less than a fixed tolerance ϵ (see Fig. 2c).

In summary, given the $k-$step solution in terms of the primal and dual components $(\mathbf{u}^k, \mathbf{d}^k)$, the update is

- Maximization step: Fixed an ascent dual discretization time τ_d, we compute for every superpixel q

$$\mathbf{d}_q^{k+1} = \frac{\mathbf{d}_q^k + \tau_d\nabla_w\mathbf{u}_q^k}{max(1, |\mathbf{d}_q^k + \tau_d\nabla_w\mathbf{u}_q^k|_\infty)} \tag{11}$$

  and set $\mathbf{d}^{k+1} = (\mathbf{d}_q^{k+1})_{q\in V}$.
- Minimization step: Fixed an descent primal discretization time τ_p and given the matrix $\mathbf{d}^{k+1}$, we compute

$$\mathbf{u}_q^{k+1} = (1 + a\tau_p)\mathbf{u}_q^k + \tau_p\left(div_w(\mathbf{d}_q^{k+1}) - \mathbf{b}_q\right) \tag{12}$$

  and set $\mathbf{u}^{k+1} = (\mathbf{u}_q^{k+1})_{q\in V}$ where

$$a = \frac{\delta^2}{\alpha^2} - \frac{\lambda}{\alpha} \qquad \mathbf{b} = \frac{\delta}{\alpha^2} - \frac{\lambda}{\alpha}\mathbf{v}^c \tag{13}$$

The parametric values of λ, α and δ modify the general behaviour of the model and generate different scenarios to be explored. In this work, we focus on the case $a < 0$ where the energy functional is strictly convex and the minimization problem is well-posed. The forcing term $\mathbf{b} = (b_p)_{p \in V}$ is a changing sign function driven by the control map which favours saliency detection.

Notice that the iterative splitting between the primal and dual variable generate a strong data dependency that penalizes the full potential of a parallel computation.

A pseudocode of our Primal-Dual algorithm for saliency is shown in Algorithm 1 where we experimentally fixed $\tau_p = 0.3$ and $\tau_d = 0.33$.

Algorithm 1 Saliency estimation on non-local TV with Saliency Term

```
procedure NLTVSalTerm(inputImage,sp,k,λ,δ,α,ϵ)
    Calculate Superpixels over inputImage
    Extract superpixel features f
    W^{sp×sp} ← Create adjacency matrix by Eq. 2
    Connections in W^{sp×sp} to k-nearest neighbours
    v^c ← Calculate controlMap by Eq. 6, Eq. 7
    Calculate a and b by Eq. 13
    Initialization: u = v^c, d = 0
    repeat
        Compute d by Eq. 11
        Compute u by Eq. 12
        Compute E(u) by Eq. 10
    until |E(u^k) − E(u^{k+1})| ≤ ϵ
    return u
end procedure
```

3 CPU and GPU computing

In this paper, we propose effective implementations using CPU and GPU. However, some forms of optimization are needed to exploit them appropriately.

3.1 CPU performance considerations

Performance improvements on a single CPU are usually achieved by using parallel computing techniques with threading or vector instructions. In the first case, there are many concurrent primitives such as mutex, locks, monitors, etc. which are used in order to synchronize threads while computing and avoid race conditions in shared memory problems. However, the use of these mechanisms when scaling from two processes to more computing threads is not trivial and some interfaces like OpenMP [14] or Intel TBB try to alleviate the effort of programming with them. Typically, these strategies need from much data and quite independent computation to be beneficial. This is not the case of our saliency problem, where there is not so much data and exhibits data dependencies.

In the second case, using vector instructions provided by the CPU manufacturer can be a very low level task and difficult, but some efforts can be made in order to ease some transparent autovectorization. Although compilers identify and optimize part of the code automatically [2], the human help organizing the code could produce important speedups as well as test different compilers. In our problem, we make use of this loop organization with Intel compilers to achieve important improvement in the performance.

3.2 GPU Computing

From desktop computers to supercomputers, the GPU is one the most computationally effective resource, especially since the introduction of proper GPU programming interfaces such as NVIDIA CUDA [31] in 2007 or OpenCL [30] in 2008. With these kind of interfaces, the GPGPU (general purpose computation using GPUs) style of using the GPU with a graphics context such as 3D primitives and transformations, textures, shaders, and so on, derived to a broader GPU computing area, in which a general programmer could launch a massive number of computing threads in a data parallel problem. These interfaces are extensions of the C/c++ languages with specific functions for: device memory allocation/deallocation, data transfers between main memory and video memory, synchronization barriers, etc.

In this hybrid CPU-GPU computing ecosystem, there exist two parts: host and device. The host is the CPU that controls the device computation whereas the device is the graphics hardware that moves and processes the data through a number of multiprocessors (streaming multiprocessors or SMs). The instructions to accomplish a specific procedure executed by these SMs are encoded in *kernels*. The GPU makes the full use of thousands of lightweight threads that execute the kernels code in parallel over the data that resides on the device memory (*global memory*). This is usually a very large memory and can be accessed and modified from both, the CPU or the GPU sides. Due to the fact that it can be read at any time by any thread is considered a slow memory in comparison to other types of GPU memories. Threads have the ability to cooperate with each other in small neighbourhoods (*blocks*), through a fast chip memory (*shared memory*) about 10 times faster than the global memory [38]. The thread possesses a private local memory (*registers*) which offer the fastest access to the memory. Their use is limited

to auxiliary calculations within a kernel because their lifetime is equal to the thread lifetime. There is a maximum number of registers available in a block and the number of registers that a thread can use depends on the number of threads in the block. So this is not a trivial resource but usually a scarce one.

The host program transfers the data to and from the device for computation, and also sets the launch configuration (the global number of computing threads organized in regular blocks of up to 1024 threads). One of the basic strategies to improve performance either in CPU or in GPU environments is a good aligned memory access pattern. Bad patterns can cause performance penalties when using memory (specially global memory). Write and read operations are executed by memory block transactions. A transaction fetches or inserts a continuous block from/into the memory. A good strategy to optimize the code is to reduce the number of transactions. Furthermore, each SM executes a block of threads and the execution inside a block in CUDA is achieved by groups of 32 threads (*warps*). The performance is optimal if all the threads in a warp fetch aligned memory (*coalesced memory access*) and all threads follow the same execution path. The thread block size should be multiple of 32 threads as a direct consequence of executing instructions in warps but also small enough to allocate most local variables in fast registers.

4 Accelerated non-local TV saliency

The complete saliency algorithm (Algorithm 1) can be divided into two stages: initialization stage (lines 2-8) and the iterative stage (lines 9-13).

4.1 Initialization stage

As explained in section 2.3, the weight matrix is very sparse and guides the complete calculations in the iterative stage of the algorithm: non-local gradient and non-local divergence. Usually, instead of performing the operations in the whole matrix $W^{sp\times sp}$, the matrix is transformed into a more compact one, $W^{sp\times k}$, following the CSR (Compressed Sparse Row) representation.

$$\underbrace{\begin{pmatrix} 5&0&1&0&2\\ 3&5&0&0&1\\ 0&2&8&0&1\\ 1&0&0&1&3\\ 4&0&9&0&2 \end{pmatrix}}_{\text{Weights}} \Rightarrow \underbrace{\begin{pmatrix} 5&1&2\\ 3&5&1\\ 2&8&1\\ 1&1&3\\ 4&1&2 \end{pmatrix}}_{\text{CSR Values}}$$

This CSR format can represent any arbitrary sparse pattern like our case where the non-zero entries (NNZ) vary in each different input image. It has three vectors: **Rows** and **Cols** to store the row/column indices of the non-zero entries in the original matrix and **Values** to store the values of the matrix.

$$\begin{aligned} \textbf{Rows} &= [0,3,6,9,12,15]\\ \textbf{Cols} &= [0,2,4,0,1,4,1,2,4,0,3,4,0,2,4]\\ \textbf{Values} &= [5,1,2,3,5,1,2,8,1,1,1,3,4,9,2] \end{aligned}$$

The size of vectors $\textbf{Cols} \in \mathbb{N}^{NNZ}$ and $\textbf{Values} \in \mathbb{R}^{NNZ}$ are always proportional to the number of non-zeros, whereas $\textbf{Rows} \in \mathbb{N}^{rows+1}$ to the number of rows. Row indices introduce a fast way to access values in the matrix and easy calculation of the number of non-zeros in a specific row ($rows[i+1]-rows[i]$). For these reasons, the compressed row format is commonly used for sparse matrix-vector multiplication [8].

The representation explained above reduces drastically the memory needs for storing the matrix as well as reducing the number of operations performed in the non-local gradient and non-local divergence. In addition, the optimization in CPU makes use of the vectorization guidelines for the Intel compiler [2] to improve the performance. The GPU implementation starts by migrating this optimized CPU approach into CUDA. Now we describe the GPU implementation in detail.

The first part of the pipeline shown in Figure 1b, superpixels extraction, uses the SLIC implementation found in [34], which reports GPU speedups of $10x \sim 20x$ compared to a CPU sequential implementation. After the region partition, we need to extract the features for each superpixel (colour and location). This operation involves selecting the pixels from each L^*a^*b channel that belong to a specific superpixel and apply the mean. This procedure is trivial in CPU but not for GPU because some coordination is needed to collect the data and sum them up. The coordination is achieved by reduction techniques. A reduction is a class of parallel algorithms that produces a scalar result from a collection [38]. Using a NVIDIA Kepler GPU, we employ the *atomicAdd* operation, as according to [1] it is an optimized implementation since CUDA 7.5. In any case, there are several reduction implementations and technologies with different performances depending on the GPU family.

In order to perform a norm between two different superpixels we need normalized feature vectors ($\overline{L}$,$\overline{a}$,$\overline{b}$,$\overline{x}$,$\overline{y}$):

$$\mathbf{v} = \frac{\mathbf{v} - min(\mathbf{v})}{max(\mathbf{v}) - min(\mathbf{v})} \forall \mathbf{v} \in (\mathbf{L}, \mathbf{a}, \mathbf{b}, \mathbf{x}, \mathbf{y})$$

this normalization requires the computation of *min* and *max* of each feature and, again, we accomplish this at the same time with a reduction kernel.

Algorithm 2 Weights & Control Map

```
1: procedure ControlmapAndWeights(L,a,b,x,y)
2:     Calculate w_pq by Eq. 2
3:     shared memory ←v_pq^con by Eq. 6
4:     sync
5:     reduce v_p^con in shared memory
6:     sync
7:     Calculate v_p^obj by Eq. 7
8:     Calculate v_p^c by Eq. 6, Eq. 7
9:     return W^{sp×sp}, v^c
10: end procedure
```

Before implementing the numerical solution of our model, we compute the weights (Eq. 2) and the control map through (Eq. 6, Eq. 7), which are very suitable and efficient for the CUDA environment (elementwise operations). For optimization purposes, both operations are unified into a single CUDA kernel with grid configuration of *sp* threads and the process is shown in Algorithm 2. Note that the *sync* instructions in lines 4 and 9 are synchronization barriers to ensure all the threads reach that instructions in their independent evolution through the kernel execution. The control map procedure requires a reduction operation inside the kernel (line 5) because the contrast prior (Eq. 6) must be repeated from one superpixel to all superpixels. Again, the control map must also be normalized in the range $[0, 1]$ for the Primal-Dual algorithm.

The next operation in the initialization stage is the selection of the k-largest values for each superpixel in the $W^{sp\times sp}$ matrix (Algorithm 3). As the matrix is not very large, in CPU is more efficient to sort the values in rows ($O(Nlog_2N)$) and extract the k-largest values in each row than finding directly the k-largest values in each unsorted row. However, due to the insufficient data and few number of rows (number of sorts), this approach is unfortunately not very efficient for the GPU. Our strategy is based on a loop of reductions to find the k-largest values in each row until all the rows are completed.

4.2 Iterative part: First GPU implementation

The iterative part (Algorithm 1 lines 9-13) has data dependencies among its operations. In the maximization step (Eq. 11 for the $\mathbf{d}^k$ calculation), the denominator is based on the $\max(|\cdot|_\infty)$ of the numerator, which needs from the $\mathbf{u}^k$ information. The first approach is

Algorithm 3 Select k-largest values in $W^{sp\times sp}$

```
1: procedure CalculateK-NhKernel(weights,sp,k)
2:     h = 0, th = actual thread index
3:     repeat
4:         sdata = row-weights, maxs = row-indices
5:         sync, j = 0
6:         repeat
7:             sdata[maxsFinal[j]] = 0, j = j + 1
8:         until j < h
9:         sync
10:        Find max in sdata  store index in maxsFinal
11:        sync
12:        h = h + 1
13:    until h < k
14:    if th ≠ maxsFinal then
15:        Write zero in actual position in weights
16:    end if
17: end procedure
```

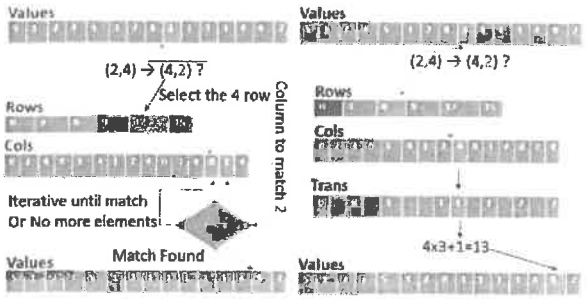
to split the maximization step into three sub operations/kernels: 1) numerator calculation with the non-local gradient of $\mathbf{u}^k$ (**nltvGradient**) (Eq. 3). The CSR representation makes this operation achieve full coalesced memory access. 2) max calculation, which requires a reduction (**Thrust::Max**) achieved by Thrust library [9]. 3) Divide the numerator by the max (**DivByMax**).

In the minimization step (Eq.12), the non-local divergence ($div_w(\mathbf{d}^{k+1})$) must be calculated before updating the primal variable. We can again split the procedure into two kernels: 1) the calculation of the non-local divergence (**nltvDivergence**). Although the CSR format also improves this procedure by reducing the number of operations, the access to transpose elements (needed in Eq. 4) is not direct and requires an iterative procedure inside the kernel to find the transpose (see Fig. 3a), and 2) Update the solution, which is an elementwise GPU efficient operation (**UpdateUk**).

The energy calculation (Eq. 10) for the stop criteria exhibits similar GPU data reduction inefficiencies as the extraction of superpixel features.

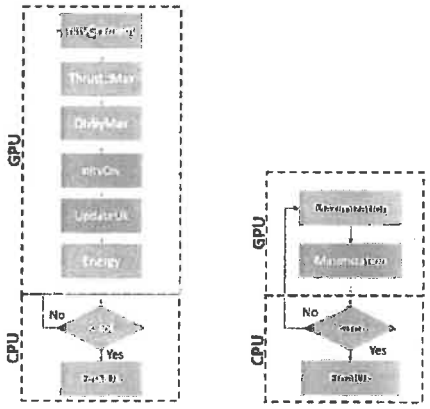
4.3 Iterative part: Optimized GPU version

The first GPU approach solves the data dependencies with the introduction of different kernels. However, as our formulation in superpixels decreases the amount of data to analyze compared to a pure pixel approach, the time of launching kernels becomes very important and therefore this direction has not produced good results (see Table 2). This data dependency could be solved by using thread synchronization barriers inside kernels. In order to improve the timing, we merge the maximization and minimization kernels shown in green and purple boxes, respectively, in Fig. 4. Accommodating the kernel launch configuration (CUDA thread grid),



(a) (b)

Fig. 3: Given an initial position (2,4), the selection of its transpose element (4,2) in the CSR format: (a) First GPU implementation. After selecting the row, we iterate through the col vector to match the specific col. (b) Optimized GPU proposal. A lookup table (**Trans**) is used to find directly whether an index in the compact matrix $W^{sp\times k}$ has transpose or not.



(a) (b)

Fig. 4: These diagrams show the kernels used to calculate the iterative part (Eq. 10). (a) First GPU implementation, where each operation in CPU has been implemented as a CUDA kernel and (b) optimized proposal grouping GPU kernels. **FinalUk** transfers the final $\mathbf{u}^k$, normalizes data [0,1] and projects the final result onto pixel domain.

the energy (Eq. 10) can be included in the minimization step with reductions.

Apart from the data dependencies, the non-local divergence (Eq. 4) implies finding transpose elements in its calculation for the dual variable $\mathbf{d}^{k+1}$ (accessing $\mathbf{d}$(p,q) and $\mathbf{d}$(q,p)). In the first GPU implementation, a procedure inside the kernel to find the transpose value has been implemented. This is inefficient because most of the values are zero and the loop for finding the match in the column vector is inside the kernel (in the worst case it iterates k times). However, as $\mathbf{d}$ has only values

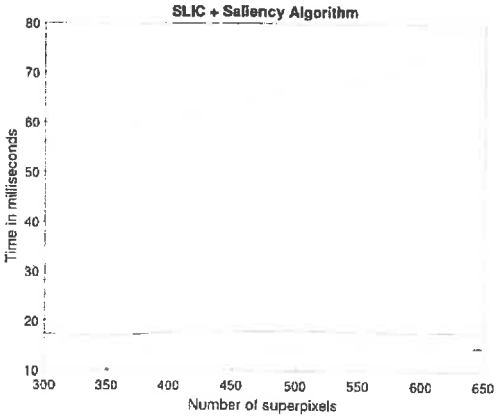


Fig. 5: Computation times (SLIC + Saliency Algorithm) for CPU-mvcc configuration (red), CPU-icc (cyan), GPU Tesla K40C (blue) and GPU Tesla K40C with no energy calculation (green), varying the number of superpixels [300,650].

in the indices where the weights are not zero, and the weights matrix is constant during the algorithm, we can create an offline lookup table (LUT) with the transpose information (**Trans** $\in \mathbb{N}^{NNZ}$ in Fig. 3b). This LUT vector stores the transpose result for each index in the compact matrix $W^{sp\times k}$ where "-" entry means the transpose value is zero and a value $\in [0, k]$ indicates the index of the column where the value in the compact matrix is located.

5 Experimental results

This section provides visual and performance results comparing our non-local total-variation including saliency term (*NLTVSalTerm*) against different approaches.

5.1 Timing results

Experimentation was performed on an Intel Xeon E5-1650v3, 3.5GHz hexa-core processor, from the 2014 Intel Haswell architecture (Haswell-EP), 1.5MB L2 cache and 15MB L3 cache with 64GB DDR3 RAM as a CPU platform using Microsoft Windows Server 2012R2 as operating system. The algorithms have been implemented in C++ using Microsoft Visual C++ compiler (mvcc) as well as Intel C++ compiler (icc).

The GPU platform is a NVIDIA Tesla K40C from the Kepler family (2014) with 12GB onboard memory and 2880 CUDA cores organized in 15 streaming multiprocessors (SM). In both platforms we average timing results running the method 100 times.

Table 1: Computational results in milliseconds of the saliency method *NLTVSalTerm* in CPU for an image [400 × 334] from the MSRA10K dataset and 100 iterations. **CPU-mvcc** stands for Microsoft Visual C++ compiler, **CPU-icc** for Intel C/C++ 19.0 compiler and **Tesla K40C** for Kepler. SP is the number of superpixels used in the algorithm.

	CPU-mvcc			CPU-icc					
SP	SLIC	Sal	Total	SLIC	Speedup	Sal	Speedup	Total	Speedup
300	49.33	6.50	55.83	21.33	2.31X	4.25	1.53X	25.58	2.18X
350	49.62	8.09	57.71	21.41	2.32X	5.36	1.51X	26.77	2.16X
400	49.87	11.56	61.43	21.69	2.30X	7.78	1.49X	29.47	2.08X
450	49.83	14.07	63.90	22.41	2.22X	9.57	1.47X	31.98	2.00X
500	50.18	17.42	67.60	22.30	2.35X	11.90	1.46X	34.20	1.98X
550	50.00	18.09	68.09	21.87	2.39X	12.40	1.46X	34.27	1.99X
600	50.32	22.90	73.22	22.50	2.24X	15.78	1.45X	38.28	1.91X
650	50.16	28.55	78.71	22.35	2.24X	19.92	1.43X	42.27	1.86X

Table 2: Computational results in milliseconds of the optimized saliency method *NLTVSalTerm* for an image [400 × 334] from the MSRA10K dataset and 100 iterations in CPU and GPU. **CPU-icc** stands for the CPU implementation using Intel C/C++ 19.0 compiler and **GPU Tesla K40C** is the GPU platform. SP is the number of superpixels used in the algorithm. Note that **Sal** column represents the saliency in the first GPU implementation, **Sal+** the optimized GPU version, the **Sal*** column represents the saliency results without calculating the energy of the functional and that the speedups (**Speedup+** and **Speedup***) shown are compared to the CPU-icc timing results and **Speedup** is the comparison against the first GPU implementation.

	CPU-icc			GPU Tesla K40C									
SP	SLIC	Sal	Total	SLIC	Sal	Sal+	Speedup+	Speedup	Total	Sal*	Speedup*	Speedup	Total*
300	21.33	4.25	25.58	8.95	22.44	8.41	0.51X	2.67X	17.36	6.08	0.70X	3.69X	14.93
350	21.41	5.36	26.77	8.57	22.87	8.40	0.64X	2.72X	16.97	6.02	0.89X	3.80X	14.51
400	21.69	7.78	29.47	8.66	23.39	9.02	0.86X	2.59X	17.68	6.17	1.26X	3.79X	14.72
450	22.41	9.57	31.98	8.52	23.63	9.23	1.04X	2.56X	17.75	6.34	1.51X	3.73X	14.85
500	22.30	11.90	34.20	8.61	24.61	9.51	1.25X	2.59X	18.12	6.43	1.85X	3.83X	14.90
550	21.87	12.40	34.27	8.16	25.31	9.57	1.30X	2.64X	17.73	6.52	1.9X	3.88X	14.70
600	22.50	15.78	38.28	8.21	25.43	9.73	1.62X	2.61X	17.94	6.66	2.37X	3.82X	14.84
650	22.35	19.92	42.27	8.23	25.18	9.79	2.03X	2.57X	18.02	6.60	3.02X	3.81X	14.78

Figure 5 shows the total time in milliseconds of different configurations, in CPU and GPU varying the number of superpixels in the range [300, 650]. The first thing to note is the great difference between timing results using different compilers in CPU. The Intel C++ compiler (CPU-icc) creates more efficient autovectorized code for the CPU using the same -O2 compiler directive than the Visual C++ compiler (CPU-mvcc) when organizing loops for improving memory aligned accesses.

Tables 1 and 2 show the breakdown of the total computational time between the two most independent methods: superpixel method (SLIC) and the variational saliency algorithm (Primal-Dual). Again, the tests have been carried out varying the number of superpixels in the range [300, 650] and forcing 100 iterations in order to avoid different results depending on the type of image and the convergence of the algorithm. Note again in Table 1 the great difference in CPU timing results when using different compilers.

In Table 2 we report timing results of 100 iterations comparing the best CPU solution (CPU-icc) against three different GPU implementations. First, note that we use a GPU version of SLIC from [34] to keep a complete GPU solution but we only report GPU speedups of our contribution (the saliency method), not the SLIC method.

The first implementation is the straightforward GPU implementation described in section 4.2. This implementation performs slightly better than the CPU-mvcc version but up to 4 times worse than the CPU-icc version. The second GPU implementation (*Sal+*) is the optimized version described in section 4.3 and is about 2.6× (column *Speedup*) faster than the first GPU implementation. However, in this implementation the resulting energy is transferred back to main memory for

evaluating the stopping criterion in CPU, although the number of iterations is fixed to 100. This transfer in each iteration is one of the historical bottlenecks of GPU implementations and many strategies can be adopted to reduce this penalty, such as processing batches and comparing the resulting energy after a number of iterations or directly use a fixed number of iterations. As we keep a fixed number of iterations of the algorithm for comparison purposes, we report a third GPU approach (*Sal**) avoiding the energy calculation and data transfer from GPU memory to host memory in each iteration. This strategy is reasonable as the implementation is focused on the hardware limitations, reducing the bottleneck, but ensures convergence when compared to an implementation that uses the stopping criterion with fewer iterations (as shows Fig. 2*c* reaching an acceptable convergence after about 50-60 iterations). It is important to note that, as expected, the speedup of the optimized GPU version is below 1 when compared to the CPU-icc results for low number of superpixels (column *Speedup+*). It needs more than 400-450 superpixels to compensate the performance overheads of data transfers and bad GPU suitability in iterative algorithms with parallel computation. Removing the energy calculation (*Sal**) we achieve a benefit of up to 3.02× when compared to the CPU-icc results for configurations of 650 superpixel and 3.8× compared to the first GPU version.

The results show that the method reachs real time frame rate capability either in CPU (∼ 25 FPS for 650 superpixels) or in GPU (∼ 60 FPS for 650 superpixels).

5.2 Visual results

The visual results have been carried out on different benchmarks: **MSRA10K** benchmark [13] which has 10000 images and each image has a unique salient object, **ECSSD** benchmark [39] with 1000 images semantically meaningful but structurally complex and **iCoseg** benchmark [7] with 643 images from Flickr Website in different situations. Following [13] typical metrics such as Precision-Recall (PR), F-measure and Mean Absolute Error (MAE) are used to evaluate the results of the saliency methods.

For fair comparision among methods Non-local L_0 (NL_0) [37], non-local TV ($NLTV$) [5] and the new model non-local convex TV with saliency term ($NLTVSalTerm$) in the datasets, we binarize the saliency maps using a threshold starting by 0 up to 255. In each threshold, we measure the four metrics above reporting the mean for the whole range $[0, 255]$. We use for NL_0 our own implementation and the parameters setting according to [37], $NLTV$ has the same parameters as in [5] and for the new model $NLTVSalTerm$, we set $\lambda = 1$ $\alpha = 1.5$ and $\delta = 0.2$. For all methods, we use a fixed number of iterations (100). The quantitative comparison shows that the $NLTVSalTerm$ provides the best results for almost all configurations (see Table. 3). A visual comparison among different methods is shown in Fig. 6. We can observe that our results, shown in Fig.6*e*, can remove much better the background in the input images than the rest of methods.

6 Conclusions

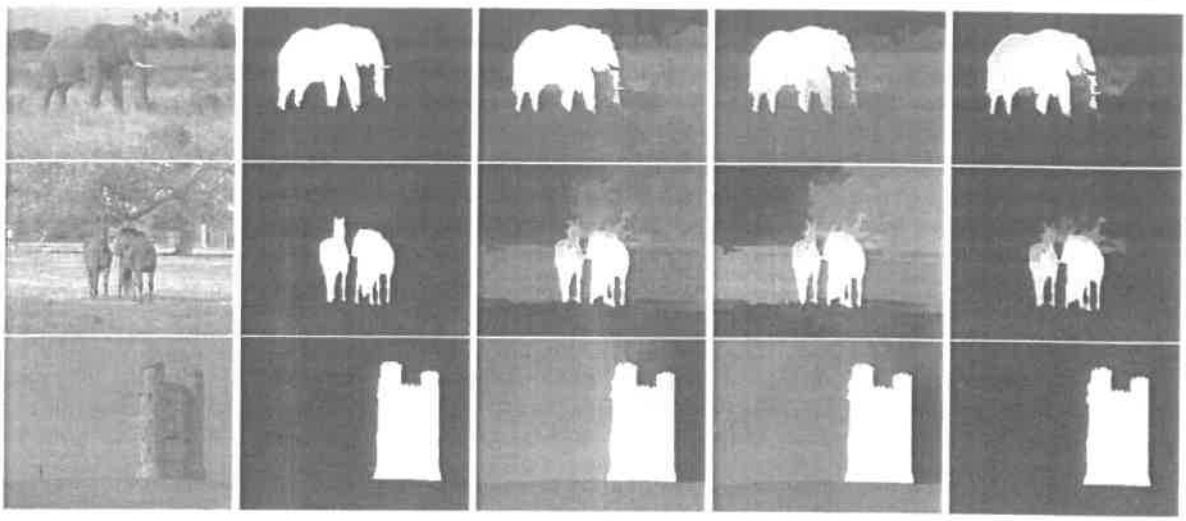
In this paper, a novel non-smooth and non-local variational method on manifold for saliency segmentation has been proposed. We solve the minimization problem with a Primal-Dual algorithm which proves to converge quick to the solution in few iterations. Although the method is based on simple priors, experimental results in the well known datasets indicate that the method produces very high quality results removing the background and highlighting the salient part of the image. Furthermore, we drive an exhaustive performance comparison between CPU and GPU using different configurations, programming tools, technologies and platforms demonstrating real-time performance either in CPU or GPU, even with computationally intensive configurations.

Acknowledgements

This work was supported in part by the Spanish government, Ministerio de Ciencia, Innovación y Universidades, RTI2018-098743-B-I00 (MICINN/FEDER) and Comunidad de Madrid, Y2018/EMT-5062.

References

1. Nvidia developer blog: Optimized filtering with warp-aggregated atomics. https://devblog.nvidia.com/. Accessed: 2019-08-20.
2. Programming guidelines for vectorization. https://software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf.
3. R. Achanta, S. Hemami, F. Estrada, and S. Susstrunk. Frequency-tuned salient region detection. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1597–1604, June 2009.
4. R. Achanta, A. Shaji, K. Smith, A. Lucchi, P. Fua, and S. Ssstrunk. Slic superpixels compared to



(a) (b) (c) (d) (e)

Fig. 6: Examples of saliency detection in datasets: iCoseg [7], ECSSD [39] and MSRA10K [13]. (a) original image, (b) ground truth, (c) saliency map using NL_0 [37], (d) saliency map using $NLTV$ [5], (e) saliency map using $NLTVSalTerm$ (our proposal).

Table 3: Quantitative results for the three datasets for the NL_0, $NLTV$ and $NTVSalTerm$ methods using Precison (PR), Recall (RC), F-Measure (F_β) and Mean Absolute Error (MAE).

iCogSeg												
	NL_0[37]				$NLTV$[5]				$NLTVSalTerm$			
Superpixels	PR	RC	F_β	MAE	PR	RC	F_β	MAE	PR	RC	F_β	MAE
300	0,678	0.524	0.589	0.220	0.653	0.518	0.580	0.236	**0.802**	**0.563**	**0.709**	**0.150**
450	0.713	0.526	0.603	0.210	0.662	0.506	0.577	0.229	**0.818**	**0.534**	**0.701**	**0.155**
600	0.722	0.529	0.606	0.207	0.683	0.509	0.587	0.225	**0.819**	**0.530**	**0.705**	**0.158**
ECSSD												
	NL_0[37]				$NLTV$[5]				$NLTVSalTerm$			
Superpixels	PR	RC	F_β	MAE	PR	RC	F_β	MAE	PR	RC	F_β	MAE
300	0.644	**0.517**	0.544	0.265	0.534	0.505	0.479	0.329	**0.680**	0.471	**0.586**	**0.186**
450	0.683	**0.522**	0.561	0.248	0.596	0.510	0.516	0.294	**0.738**	0.467	**0.609**	**0.175**
600	0.700	**0.521**	0.568	0.238	0.635	0.515	0.538	0.277	**0.763**	0.450	**0.607**	**0.177**
MSRA10K												
	NL_0[37]				$NLTV$[5]				$NLTVSalTerm$			
Superpixels	PR	RC	F_β	MAE	PR	RC	F_β	MAE	PR	RC	F_β	MAE
300	0.748	0.544	0.630	0.209	0.664	0.523	0.582	0.248	**0.831**	**0.583**	**0.730**	**0.125**
450	0.773	0.557	0.643	0.197	0.714	0.538	0.615	0.227	**0.861**	**0.559**	**0.729**	**0.126**
600	0.787	**0.559**	0.646	0.191	0.742	0.542	0.630	0.215	**0.873**	0.532	**0.716**	**0.131**

state-of-the-art superpixel methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(11):2274–2282, Nov 2012.

5. E. Alcaín, A. Muñoz, I. Ramírez, and E. Schiavi. *Modelling Sparse Saliency Maps on Manifolds: Numerical Results and Applications*, pages 157–175. Springer International Publishing, Cham, 2019.
6. Shai Avidan and Ariel Shamir. Seam carving for content-aware image resizing. In *ACM Trans. Graph*, page 10. SIGGRAPH, 2007.
7. Dhruv Batra, Adarsh Kowdle, Devi Parikh, Jiebo Luo, and Tsuhan Chen. icoseg: Interactive co-segmentation with intelligent scribble guidance. In *CVPR*, pages 3169–3176, 2010.
8. Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

9. Nathan Bell and Jared Hoberock. Thrust: Productivity-oriented library for cuda. *Astrophysics Source Code Library*, 7:12014–, 12 2012.
10. Theofilos Chamalis and Aristidis Likas. Region merging for image segmentation based on unimodality tests. In *3rd International Conference on Control, Automation and Robotics (ICCAR)*, April 2017.
11. Antonin Chambolle and Thomas Pock. A first-order primal-dual algorithm for convex problems withapplications to imaging. *Journal of Mathematical Imaging and Vision*, 40(1):120–145, May 2011.
12. Tony F. Chan and Jianhong (Jackie) Shen. Variational image inpainting. *Communications on Pure and Applied Mathematics*, 58(5):579–619, 2005.
13. M. Cheng, N. J. Mitra, X. Huang, P. H. S. Torr, and S. Hu. Global contrast based salient region detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(3):569–582, March 2015.
14. L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan 1998.
15. Abraham Duarte, Ángel Sánchez, Felipe Fernández, and Antonio S. Montemayor. Improving image segmentation quality through effective region merging using a hierarchical social metaheuristic. *Pattern Recognition Letters*, 27(11):1239–1251, 2006.
16. Abderrahim. Elmoataz, Matthieu. Toutain, and Daniel. Tenbrinck. On the p-laplacian and ∞-laplacian on graphs with applications in image and data processing. *SIAM Journal on Imaging Sciences*, 8(4):2412–2451, 2015.
17. P. Favaro and S. Soatto. A geometric approach to shape from defocus. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(3):406–417, March 2005.
18. Hong Fu, Zheru Chi, and Dagan Feng. Attention-driven image interpretation with application to image retrieval. *Pattern Recognition*, 39(9):1604 – 1621, 2006.
19. Lee Gayoung, Tai Yu-Wing, and Kim Junmo. Deep saliency with encoded low level distance map and high level features. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
20. Guy. Gilboa and Stanley. Osher. Nonlocal operators with applications to image processing. *Multiscale Modeling & Simulation*, 7(3):1005–1028, 2009.
21. V. L. Ginzburg. On the theory of superconductivity. *Il Nuovo Cimento (1955-1965)*, 2(6):1234–1250, Dec 1955.
22. Carlos A. S. J. Gulo, Henrique F. de Arruda, Alex F. de Araujo, Antonio C. Sementille, and João Manuel R. S. Tavares. Efficient parallelization on gpu of an image smoothing method based on a variational model. *Journal of Real-Time Image Processing*, 16(4):1249–1261, Aug 2019.
23. Q. Hou, M. Cheng, X. Hu, A. Borji, Z. Tu, and P. Torr. Deeply supervised salient object detection with short connections. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5300–5309, July 2017.
24. F. Huang, J. Qi, H. Lu, L. Zhang, and X. Ruan. Salient object detection via multiple instance learning. *IEEE Transactions on Image Processing*, 26(4):1911–1922, April 2017.
25. L. Itti, C. Koch, and E. Niebur. A model of saliency-based visual attention for rapid scene analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(11):1254–1259, Nov 1998.
26. Jianbo Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, Aug 2000.
27. Z. Li, X. Wu, and S. Chang. Segmentation using superpixels: A bipartite graph partitioning approach. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 789–796, June 2012.
28. A. Martin, J. Garamendi, and E. Schiavi. *Two efficient primal-dual algorithms for MRI Rician denoising*, pages 291–296. 01 2013.
29. Yi Zhan Meng Li and Lidan Zhang. Nonlocal variational model for saliency detection. *Mathematical Problems in Engineering*, vol. 2013, Article ID 518747, 7 pages, 2013.
30. Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 1st edition, 2011.
31. NVIDIA Corporation. NVIDIA CUDA C programming guide, 2017. Version 9.0.
32. Thomas Pock, Markus Grabner, and Horst Bischof. Real-time computation of variational methods on graphics hardware. In *Proceedings 12th Computer Vision Winter Workshop*, pages 67–74, 2007.
33. I. Ramírez, G. Galiano, and E. Schiavi. Non-convex non-local flows for saliency detection. *CoRR*, abs/1805.09408, 2018.
34. C. Y Ren, V. A. Prisacariu, and I. D Reid. gSLICr: SLIC superpixels at over 250Hz. *ArXiv e-prints*, September 2015.
35. M. Rumpf and R. Strzodka. Nonlinear diffusion in graphics hardware. In David S. Ebert, Jean M. Favre, and Ronald Peikert, editors, *Data Visual-*

ization 2001, pages 75–84, Vienna, 2001. Springer Vienna.

36. E. Strekalovskiy and D. Cremers. Real-time minimization of the piecewise smooth mumford-shah functional. In *European Conference on Computer Vision (ECCV)*, pages 127–141, 2014.
37. Yiyang Wang, Risheng Liu, Xiaoliang Song, and Zhixun Su. Saliency detection via nonlocal $$l_{0}$$minimization. In Daniel Cremers, Ian Reid, Hideo Saito, and Ming-Hsuan Yang, editors, *Computer Vision - ACCV 2014*, pages 521–535, Cham, 2015. Springer International Publishing.
38. N. Wilt. *Cuda Handbook: A Comprehensive Guide to Gpu Programming*. CreateSpace Independent Publishing Platform, 2017.
39. Q. Yan, L. Xu, J. Shi, and J. Jia. Hierarchical saliency detection. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1155–1162, June 2013.
40. Ting Zhao and Xiangqian Wu. Pyramid feature attention network for saliency detection. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.
41. F. Znd, Y. Pritch, A. Sorkine-Hornung, S. Mangold, and T. Gross. Content-aware compression using saliency-driven image retargeting. In *2013 IEEE International Conference on Image Processing*, pages 1845–1849, Sep. 2013.