Neural Comput & Applic (2018) 30:3007–3018
https://doi.org/10.1007/s00521-017-2873-3

CrossMark

ORIGINAL ARTICLE

# CUDA-based parallelization of a bio-inspired model for fast object classification

**Daniel E. Hernández[1] · Gustavo Olague[2] · Benjamín Hernández[3] · Eddie Clemente[4]**

Received: 6 October 2014 / Accepted: 10 February 2017 / Published online: 24 February 2017
© The Natural Computing Applications Forum 2017

**Abstract** The need for highly accurate classification systems capable of working on real-time applications has increased in recent years. Nowadays, several computer vision tasks apply a classification step as part of bigger systems, hence requiring classification models that work at a fast pace. This rendered interesting the concept of real-time object classification to several research communities. In this paper, we propose to accelerate a bio-inspired model for object classification, which has given very good results when compared with other state-of-the-art proposals using the compute unified device architecture (CUDA) and exploiting computational capabilities of graphic processing units. The classification model that is used is called the artificial visual cortex, a novel bio-inspired approach for image classification. In this work, we show that through an implementation of this model in the CUDA framework it is possible to achieve real-time functionality. As a result, the proposed system is able to process images in average of up to 90 times faster than the original system.

**Keywords** CUDA · Real-time object classification · Artificial visual cortex

✉ Gustavo Olague
olague@cicese.mx

Daniel E. Hernández
daniel.hernandezm@tectijuana.edu.mx

Benjamín Hernández
benja@astrosen.unam.mx

Eddie Clemente
eclemente@ite.edu.mx

1 TecNM - Instituto Tecnológico de Tijuana, Calzada Del Tecnológico S/N, Fraccionamiento Tomas Aquino, C.P. 22414, Tijuana, B.C., Mexico

2 Centro de Investigación Científica y de Educación Superior de Ensenada, Carretera Ensenada-Tijuana No. 3918, Zona Playitas, C.P. 22860, Ensenada, B.C., Mexico

3 Instituto de Astronomía, Universidad Nacional Autónoma de México, Km. 103 Carretera Tijuana-Ensenada, Ensenada, B.C., Mexico

4 TecNM Instituto Tecnológico de Ensenada, Blvd. Tecnológico No. 150, Ex Ejido Chapultepec, C.P.22750, Ensenada, B.C., Mexico

## 1 Introduction

Real-time object classification is the basis of numerous computer vision applications, such as home service robots [1], autonomous vehicles [2], tracking [3], to mention but a few. These kinds of systems require high quality classification techniques that are able to function at a high pace. Today, there are many methods that attempt to approach the object classification problem, but it is far from being solved. The most popular methods are those based on visual features and descriptors working on categorical representation that are useful for determining if an object is a member of a given class. The description process is performed either by exploiting pixel-level, neighborhood-level, or region-level features [4].

Along these research lines, there are other approaches for the object recognition task including those based on the evolutionary computation paradigm [5–7]. For example the system called GenEigSda proposed in [8], whose authors implement the concept of slow feature analysis together with echo state networks in order to analyze temporal data from sensors, while classifying images from the MNIST digits database. Also, Yang et al. [9] propose the parent–offspring progressive learning method which extends the extreme learning machine paradigm by implementing a

multiple neural network setting, and at the same time increasing the systems classification accuracy. Another interesting system is the one described in [10], where Ng et al. define a vehicle classification system using histogram of oriented gradient features with a multi-class support vector machine. They increase the classification accuracy of the system by implementing a visual background extraction pre-processing stage.

One common aspect of all these approaches is that they are computationally expensive, mainly due in part to the high-information content that is necessary to process while working with large images or even worst when they need to operate over video. For such reason, many algorithms work on small images, for sizes that do not even reach the VGA resolution of 640 $\times$ 480 pixels. This kind of situation is encountered in real-life applications where it is common to find higher resolution images, which decrease the efficiency of these algorithms from a computational perspective.

In recent years, graphical processing units (GPUs) have transformed into a multi-core, multi-threaded, parallel processors with great computational power, very good memory conditions—capacity and speed—and economy, which render GPU technology attractive for computationally demanding procedures. Thanks to frameworks such as Nvidia's Compute Unified Device Architecture (CUDA) the GPU can be easily accessed for general purpose applications leading to a field called general purpose GPU (GPGPU) [11].

The GPGPU paradigm has been widely used in computer vision problems such as: local feature extraction, optical flow calculation, medical image segmentation, object detection and object recognition. For example, Lu presented a CUDA implementation of a neighborhood-level feature extraction algorithm that reaches speedups of up to 20 times faster than the CPU version of the SIFT algorithm [12]. Chase et al. [13] proposed a GPU implementation of a real-time optical flow algorithm which is more than three times faster than a CPU implementation. Jiang et al. [14] presented a CUDA-based system for processing magnetic resonance images (MRI). Ohmer et al. [15] suggest a CUDA-based face recognition system based on kernel principal component analysis techniques. Their system functions 28 times faster than the original sequential implementation. Uetz and Behnke proposed a CUDA-accelerated set of hierarchical neural networks for object recognition; as a result, their GPU implementation is 82 times faster than their CPU implementation [16]. Chikkerur described a CUDA implementation of the original HMAX model, achieving speedups of up to 20 times faster than the original CPU implementation [17]. Similarly, Mutch and Poggio present a CUDA implementation of an extended HMAX model described in [18]. That system was designed for and tested in the Caltech 101 and GRAZ-02 datasets, the original implementation is later compared to the one proposed in this work. Parks et al. [19] presented a CUDA implementation of a saliency system for detection and the HMAX model for recognition, both steps are 10 times faster compared with the original algorithms. Woodbeck et al. [20] presented a GPU implementation of a bio-inspired model—similar to the HMAX model—using the OpenGL framework that achieves speedups of up to three orders of magnitude. Note that these last three works are based on the HMAX model, a region-based visual feature system, which is similar to our proposed model called the *AVC* algorithm. In this way, a main distinction between both models is the change of paradigm from a data-driven into a function-driven processing. Thus, the latter replaces a set of image patches by a set of image transformations; hence, considerably reducing the computational complexity and achieving better results than the available parallel versions of the HMAX model [21]. Nevertheless, a major problem exists for the case of processing high-resolution images.

Besides the previous methods, there are others models whose main focus is on reducing the computational time of classifying objects within an image dataset. For example the case of the convolutional neural networks (CNN), originally presented in [22], whose main idea is to apply a kind of neural network as a way to divide the recognition task in two parts: the first part is responsible of feature extraction and is typically built by one or more convolutional layers interspersed with several sub-sampling layers in order to reduce the computational time and to build up further spatial and pose invariance of the visual features [23]; the second part is typically a fully connected perceptron which takes the extracted visual features as input. This paradigm has been applied in several image databases such as the MNIST hand written digits dataset [24], the NORB multi-view objects dataset [25] and the CIFAR10 ten class image dataset [26] to mention only a few. It is important to note that these methods achieve good classification rates while reducing the computational cost of the process, since all of them have CUDA-based implementations.

The GPGPU parallelization paradigm has also been used for classification problems, vector quantization, anomaly detection, to mention but a few applications. For example, Dhanasekaran and Rubin proposed a GPU-based algorithm for the k-mean clustering problem that works three times faster than the CPU-based implementation [27]. Also, Xiao et al. [28] presented a GPU-based implementation of the LGB and self-organizing maps training algorithms. Finally, Nantes et al. [29] described an artificial neural network paradigm for anomaly detection in virtual environments for game testing.

The main idea developed in this paper is to propose a CUDA implementation of a bio-inspired classification algorithm called the artificial visual cortex (*AVC*) proposed in [21], which has shown good classification performance at a high computational cost, caused by the amount of transformations that the input image undergoes in order to be classified. The system takes around 350 milliseconds on small images (140 × 105 pixels) and almost 4 seconds on a VGA image. Hence, this model may not viable on a real-life situation and definitely not viable for real-time applications. The goal of this work is to exploit the computational power of the GPU through the CUDA framework in order to reduce the computational time of the *AVC* algorithm.

The remainder of this paper is organized as follows. First, Sect. 2 is devoted to describing the Artificial Visual Cortex model along details of implementation with the CUDA framework. Later, Sect. 3 presents the experimental results by comparing the computational performance of the algorithm in its CPU implementation against the CUDA version, as well as comparing it with some state-of-the-art CUDA-based systems. Finally, Sect. 4 provides our conclusion and some ideas for future work.

## 2 Parallel AVC algorithm with CUDA

The purpose of this section is to describe the structure and functionality of the *AVC* model along with different stages of the algorithm that were accelerated through the CUDA framework. First, a brief description of the operations performed during each stage of the model is given. Then, we outline the resulting computational complexity by calculating the operations of the sequential system. Finally, we propose a way of exploiting the parallelism offered by the GPU in order to improve the overall performance of the algorithm.

The artificial visual cortex model is based on the functionality of the human visual system. It is based on the transformations performed by the two main information routes of the visual cortex—the dorsal and ventral streams—the first stream is charged of finding the salient features of the objects in the scene, while the second stream is charged of identifying the object [30, 31]. The proposed CUDA-based version of the model is called *AVC–CUDA*.

The *AVC–CUDA* algorithm is divided in several stages as depicted on the left side of Fig. 1, where each stage requires a series of image operations to transform a color image into a descriptor vector. Note that there are two ways in which this model could be parallelized: the first one could be achieved by performing all operations of a given stage during a single processing time, while the second option could be achieved through the parallelization of the

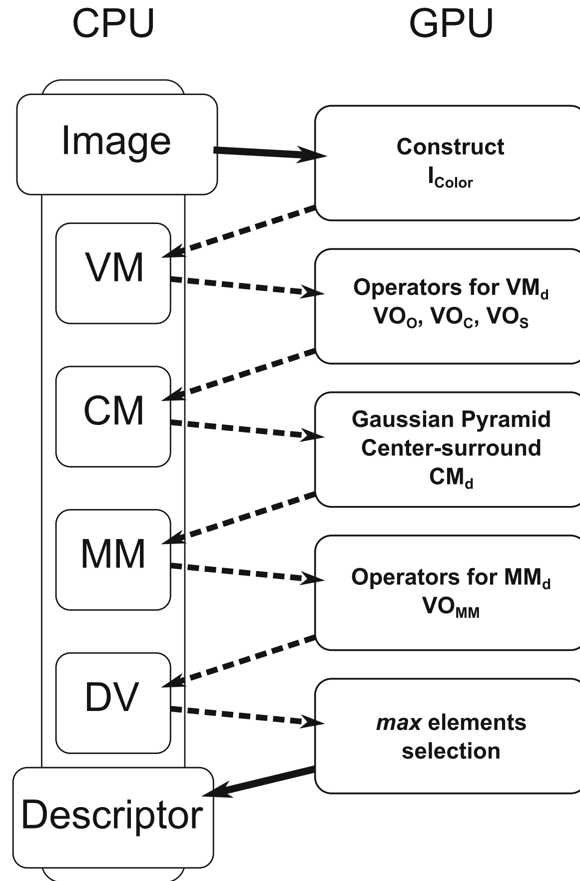


**Fig. 1** This flowchart of the *AVC–CUDA* provides a schematic representation of the image transformation process and the division of labor between CPU and GPU. The *solid lines* represent data transfer between host and device, while *dot lines* indicate only a change in the information processing

operations that are required to compute each function. For this work, we chose the second alternative since it represents the major bottleneck of the algorithm due to the complexity of the functions that compose the image transformations. In this way, Fig. 1 has two sides, the left one represents the CPU operations, defined by function calls and flow control, while the right side illustrates the image transformations performed by the GPU.

### 2.1 Initialization

An important aspect to consider while programming the GPU is to minimize the data transfer between CPU and GPU, since it is computationally expensive. Thus, in the proposed model we focus on developing a system with only two stages that are charged of the data exchange between host and device, marked as solid arrows in Fig. 1. First, the initialization step of the algorithm copies the

input image to the device. Then, the image is decomposed into three subimages, each containing information from a single color channel, $I_r$, $I_g$, $I_b$. Later, these channels are transformed into the *CYMK* and *HSV* color spaces resulting in color components separated into independent images, defining the set $I_{\text{color}} = \{I_r, I_g, I_b, I_c, I_m, I_y, I_k, I_h, I_s, I_v\}$ that corresponds to the *red, green, blue, cyan, magenta, yellow, black, hue, saturation and value* components of the input image.

## 2.2 Visual maps

The next step of the *AVC* model looks for relevant characteristics of the object captured by the image. This is accomplished through the analysis of the information in four different feature dimensions: orientation (*O*), color (*C*), shape (*S*) and intensity (*Int*). For this purpose, a visual operator (*VO*) is applied per dimension, where each operator extracts relevant information from the image resulting in a set of visual maps (*VMs*) containing prominent information at each dimension. Thus, the *VOs* are defined as mappings $VO_d : I_{\text{color}} \rightarrow VM_d$, where $d \in \{O, C, S, Int\}$ is the dimension index. These operators are constructed through function composition using the proposed operations of Table 1.

These functions work over the set of images $I_{\text{color}}$ resulting in highly expensive operations that depend on the size of the images. The computational time required to evaluate the *VOs* for the *VMs* creates a bottleneck in the algorithm. For this reason, we propose to parallelize through the CUDA framework all functions involved in the definition of the *VOs*; thus, considerably reducing the computational cost when applied over large images. The process to paralelize the functions of Table 1 is described next.

- First, point-by-point operations such as arithmetic functions, square, square root, logarithm and exponential functions have to perform the corresponding operation at each pixel of the image. Hence, in a sequential machine the time complexity is $O(n * m)$ where $n$ and $m$ are the image width and height, respectively. On the other hand, on a parallel architecture such as the GPU the time complexity is given by

$$O\left(\frac{n * m}{T}\right), \quad (1)$$

where $T$ is the number of available processors, in this case, the number of threads that will be called by the GPU to evaluate a given function. Note that if the number of threads is large enough, that is $T = n * m$, the functions could be performed in constant time $O(1)$. Hence, the idea is to have a thread per pixel where each thread performs the required operation over all elements in the image, resulting in a point-by-point function. In this way, the number of thread blocks of the CUDA kernels for our implementation is given by:

**Table 1** Functions for the visual operators (*VOs*)

| Function | Description |
|---|---|
| $A + B$, $A - B$, $A \times B$, $A / B$ | Arithmetic functions between two images $A$ and $B$ |
| $\log(A)$, $\exp(A)$ | Transcendental functions over the image $A$ |
| $(A)^2$ | Square function over the image $A$ |
| $\sqrt{A}$ | Square root function over the image$A$ |
| $(A)^c$ | Image complement over the image A |
| $\text{Op}_{r-g}(I)$, $\text{Op}_{b-y}(I)$ | Color opponency red–green and blue–yellow |
| $\text{thr}(A)$ | Dynamic threshold function over the image $A$ |
| $k + A$, $k - A$, $k \times A$, $A / k$ | Arithmetic functions between an image $A$ and a constant $k$ |
| *round*(A), *half*, $\lfloor A \rfloor$, $\lceil A \rceil$ | Round, half, floor and ceil functions over the image $A$ |
| $A \oplus \text{SE}_d$, $A \oplus \text{SE}_s$, $A \oplus \text{SE}_{dm}$ | Dilation operator with disk, square and diamond structure element (SE) |
| $A \ominus \text{SE}_d$, $A \ominus \text{SE}_s$, $A \ominus \text{SE}_{dm}$ | Erosion operator with disk, square and diamond structure element (SE) |
| *Sk*(A) | Skeleton operator over the image $A$ |
| *Perim*(A) | Find perimeter of objects in the image $A$ |
| $A \circledast \text{SE}_d$, $A \circledast \text{SE}_s$, $A \circledast \text{SE}_{dm}$ | Hit or miss transformation with disk, square and diamond structures |
| $T_{hat}(A)$, $B_{hat}(A)$ | Performs morphological top-hat and bottom-hat filtering over the image $A$ |
| $A \circledcirc \text{SE}_s$, $A \odot \text{SE}_s$ | Opening and closing morphological operators on $A$ |
| $\lvert A \rvert$, $\lvert A + B \rvert$, $\lvert A - B \rvert$ | Absolute value applied to $A$, and the addition and subtraction operators |
| *inf*(A, B), *sup*(A, B) | Infimum and supremum functions between the images $A$ and $B$ |
| $G_{\sigma=1}(A)$, $G_{\sigma=2}(A)$ | Convolution of the image $A$ and a Gaussian filter with $\sigma = 1$ or 2 |
| $D_x(A)$, $D_y(A)$ | Derivative of the image $A$ along direction $x$ and $y$ |

$$\text{Thread blocks} = \left\lfloor \frac{D+T-1}{T} \right\rfloor,$$

where $D$ is the data size and $T$ is the number of threads per block. Thus, creating a thread for each pixel of the input image as seen in Fig. 2.

- In contrast to point-by-point operations, neighborhood-based functions require the convolution of a kernel over the image in order to evaluate the output [32]. For example, the Gaussian smoothing filter is implemented through convolution of a Gaussian kernel with the image. In this way, the time complexity for this kind of functions in the CPU is $O(n * m * k^2)$ where $k$ is the size of the filter kernel. Therefore, the theoretical speedup achieved by a parallel computer is

$$O\left(\frac{n * m * k^2}{T}\right). \quad (2)$$

In this last case, the idea is to have one thread for each element of the output matrix, where each thread is charged of performing the neighborhood operation required by its corresponding output pixel as depicted by Fig. 3. The resulting speedup for this step is experimentally shown later in this paper.

### 2.3 Conspicuous maps

After obtaining the *VMs*, the system applies a center-surround process, whose goal is to further expose the previously proposed characteristics of the objects over the four dimensions. This process transforms the *VMs* into conspicuity maps (*CMs*); and it is divided in two parts: first, the *CMs* are calculated by the differences between fine and

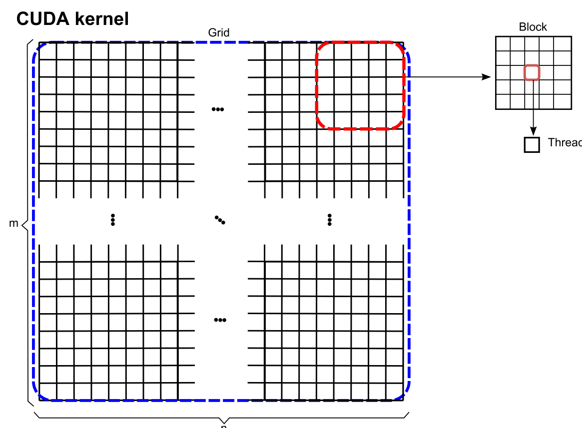


**Fig. 2** CUDA kernels are divided into thread blocks, each containing $T$ threads and each thread performing the required operation over a pixel or matrix position

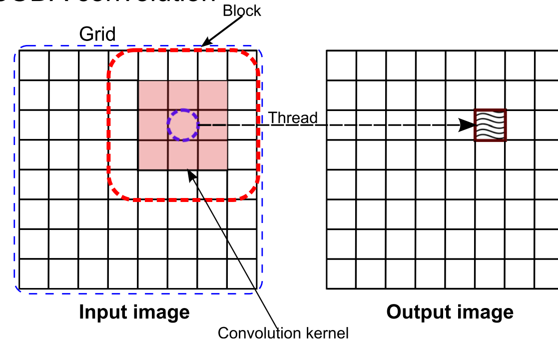


**Fig. 3** CUDA kernels used for convolution are divided into thread blocks containing $T$ threads, where each thread performs the required neighborhood operation over a region of the image considering the corresponding output pixel

coarse scales, which are computed through a nine-level Gaussian pyramid

$$P_d^{\sigma} = \{P_d^{\sigma=0}, P_d^{\sigma=1}, P_d^{\sigma=2}, P_d^{\sigma=3}, \ldots, P_d^{\sigma=8}\}.$$

Hence, one pyramid per dimension is constructed, each with its corresponding $VM_d$ per dimension by applying a Gaussian smoothing filter. Second, a new pyramid of six levels is built by evaluating the difference between the levels of the previous pyramid as follows

$$Q_d^j = P_d^{\sigma=\left\lfloor\frac{j+9}{2}\right\rfloor+1} - P_d^{\sigma=\left\lfloor\frac{j+2}{2}\right\rfloor+1}$$

where $j = \{1, 2, \ldots, 6\}$. Note that the levels of the Gaussian pyramid have different sizes, due to the smoothing process; therefore, the nine maps are scaled down to the smallest map size that corresponds to the size of the top level. Next, the six levels of the second pyramid are normalized and combined by applying the summation operation. Then, the resulting map is normalized and scaled up to the size of the original $\mathrm{VM}_d$ using a polynomial interpolation, hence resulting in the corresponding $\mathrm{CM}_d$, which are the input to the next process of the *AVC* algorithm.

This stage of the algorithm is computationally demanding, due in part to the number of required convolutions. It consists of nine operations for each of the pyramids resulting in 27 Gaussian filtering processes. The convolution is a blending operation that integrates an image with a kernel, where each output pixel is the result of the sum of a pointwise multiplication of the kernel with the neighborhood of the corresponding input pixel and is formally written as:

$$r(i) = (s * k)(i) \sum_n s(i-n)k(n).$$

In the context of image processing, a filter is applied by performing a convolution of the input image with a filter

kernel. The aim is to evaluate the scalar product of the elements of the filter kernel with the pixels of the input image within a neighborhood around each of the output pixels. This scalar product is a parallelizable operation that is well suited for computation on highly parallel hardware such as the GPU [33].

In general, the convolution of an image with a bi-dimensional filter kernel requires $l * k$ multiplications per output pixel, leading to a computational complexity given by $O(n * m * l * k)$ where $n$ and $m$ are the image dimensions and $l$ and $k$ represent the kernel size. In our case, a Gaussian smoothing filter is a special kind of kernel with the property of being a separable kernel. A separable filter kernel can be expressed as the integration of two one-dimensional filters; one for the rows and one for the columns of the image. This kind of filter can be divided into two consecutive one-dimensional convolution operations over the image, hence it requires only $l + k$ multiplications for each output pixel. Therefore, the time complexity of a separable filter is given by $O(m * n(l + k))$. In this way, the parallel version of this process is described next.

- The CUDA implementation of a separable filter has a time complexity of

$$O\left(\frac{m * n(l + k)}{T}\right) \quad (3)$$

  where $T$ is the number of threads for the required operations. Our approach to implement the convolution in CUDA is based on the proposal presented in [34]. The idea is to separate the image in sections of the same size so each part is loaded into the shared memory of a thread block; then, the thread executes a pointwise multiplication of a kernel size portion of the image section and writes the sum into the output image in the device memory. In this way, each thread block processes a section of the image, so each thread is in charge of evaluating a pixel of the output image, hence each one performing $l + k$ multiplications. Thus, reducing considerably the computational time as soon as the image grows in size. In this way, the process of a neighborhood operation performed by each thread is depicted in Fig. 3 but using one-dimensional convolutional kernels. The speedup results for this stage of the algorithm are shown later in this paper.

### 2.4 Mental maps

As seen in Fig. 1, the following process of the *AVC–CUDA* algorithm is to calculate the mental maps *MMs*. Once the *CMs* are obtained, the system implements a set of *VOs* to transform the $\mathrm{CM}_d$ per dimension into a new map that is called mental map ($\mathrm{MM}_d$) that contains the most prominent regions per feature dimension. Similarly to previous stages the computational time required to evaluate the *VOs* for the *MMs* creates a bottleneck in the algorithm. For this reason, we propose again to parallelize this stage through the CUDA framework. Each $MM_d$ per dimension is evaluated as follows:

$$\mathrm{MM}_d = \sum_{i=1}^{k} (\mathrm{VO}_{\mathrm{MM}_i}(\mathrm{CM}_d))$$

where $k$ is the cardinality of the set of operators $\mathrm{VO}_{\mathrm{MM}}$. Since $k$ varies together with the visual operators from one instance of the algorithm to another, which can be user defined or found through an optimization process. Note that the models presented in this work were constructed through a random search process, which was sufficient to compete with the algorithms reported in the state-of-the-art. Thus, the output of this stage is a map $\mathrm{MM}_d$ per dimension that results of applying the $k$ visual operators to the respective CM. Then, these $k$ maps are joined into a final $\mathrm{MM}_d$ per feature dimension using a summatory function. Note also that the operators are used along all dimensions instead of applying a given operator per dimension as in a previous stage of the algorithm.

The process for evaluating the *MMs* is similar to the procedure that is used to obtain the *VMs*. In this way, the required set of CUDA-based functions can be found in Table 2. Note that the functions are similar to those required by the *VMs*; therefore, the analysis to obtain the speedup per operator—from a CPU implementation to the CUDA—is the same one as that presented earlier. Nevertheless, in our work the number of *VOs* per dimension varies from three to 12, considering different instances of the algorithm and that each operator is applied to the four-feature dimensions. Hence, the computational time required for this stage of the algorithm is greater than that reported for the *VMs* stage.

### 2.5 Descriptor vector

The final goal of the *AVC–CUDA* model is to obtain the descriptor vector associate to the input image. The four *MMs* represent the regions in the image where prominent characteristics were found according to each feature dimension. Hence, they symbolize the information of the object that can be useful for classification tasks. In this way, the maps $MM_d$ are concatenated into a single array of numbers. Then, the *max* operation is applied over all image values to find the $n$ highest elements to be selected as the elements of the descriptor vector in order to symbolize the object in the image. Note that in a sequential implementation, the process of finding the 200 highest values has a computational complexity of

**Table 2** Functions used for building the visual operators for the mental maps ($VO_{MM}$)

| Function | Description |
|---|---|
| $A + B$, $A - B$, $A \times B$, $A\ /\ B$ | Arithmetic functions between two images $A$ and $B$ |
| $\|A + B\|$, $\|A - B\|$ | Absolute value applied to the addition and subtraction operators |
| $\log(A)$ | Transcendental functions over the image $A$ |
| $(A)^2$ | Square function over the image $A$ |
| $\sqrt{A}$ | Square root function over the image $A$ |
| $G_{\sigma=1}(A)$, $G_{\sigma=2}(A)$ | Convolution of the image $A$ and a Gaussian filter with $\sigma = 1$ or 2 |
| $D_x(A)$, $D_y(A)$ | Derivative of the image $A$ along direction $x$ and $y$ |

$$O(200 * n * m * 4) = O(n * m)$$

since the system must compute all elements associate to the *MMs* and that we would like to minimize the computational cost for building the descriptor vector, the four *MMs* are concatenated into a single array and the 200 highest values are extracted using the *max* operation using the *cuBLAS* library that has a time complexity of

$$O(\log(n * m * 4)) = O(\log(n * m)).$$

Such a function provides the *max* element of an array. Since the operation is performed several times, the resulting time is given by

$$O(200 * log(n * m * 4)) = O(log(n * m)), \quad (4)$$

which is a great improvement and this will be appreciated in the experiments.

Finally, once the descriptor vector is built, it should be transferred from the device to the host, and this step corresponds to the second data exchange between GPU and CPU. This concludes the description of the *AVC–CUDA* algorithm. The following sections present experimental results that show the achieved speedup by comparing the proposed model with the original CPU implementation along the several stages of the algorithm.

## 3 Experiments and results

In this work, two experiments were carried out. The first one focuses on the presentation around the quality of the solutions by considering a well-known classification problem, which was one of the main motivations to accelerate the original model, since it achieves an accuracy that is comparable to other models of the state-of-the-art. The second experiment shows the speedup achieved by the proposed model by exploiting the capabilities of a graphics card using the CUDA framework. For both experiments, 100 AVC solutions were studied, the models were built through a random process by selecting arbitrarily functions from Tables 1 and 2 to define the *VOs* required for the *VMs* and *MMs*, respectively. Then, all solutions were implemented in the CUDA framework and tested as follows.

All experiments were carried out on a Dell Precision T7500 workstation, with Intel Xeon eight core processor, using 8 GB RAM and an Nvidia Quadro 4000 with 256 CUDA cores and 2 GB of memory.

In order to test the classification accuracy of the solutions, the GRAZ database was used as the testing protocol. GRAZ belongs to the PASCAL object recognition database collection proposed by Opelt et al. [36]; it is composed of two challenging datasets: GRAZ-01 and GRAZ-02. The first one contains two image classes, Bikes and Persons, as well as the non-class set known as the background set. The images for this database were captured in different locations, the objects were captured at different scales and from various viewpoints in order to make the recognition task more complex. The second set, GRAZ-02, is an improvement over the first set since the number of locations—where the images were capture—was increased with the aim of achieving object's independence to the image background. In addition, the complexity of the images is increased by adding occlusions and varying the object's location over the image plane. Also, the Cars dataset was added to the test as a new object category. For our work, the protocol followed for testing the *AVC* solutions is the one proposed in [36]. In this way, 100 positive images and 100 negative images were randomly selected as training samples from the classes in the GRAZ-01 database, together with 50 positive and 50 negatives images for testing purposes. On the other hand, for the GRAZ-02 classes, 150 positive and 150 negative images were randomly selected as training data, along with 75 positive and 75 negative images for testing. The classification performance comparison of the AVC model, in both CPU and CUDA implementations, can be seen in Tables 3 and 4, the values are the equal error rate and area under the curve for GRAZ-01, calculated for the binary classification of the images from the testing set for GRAZ-01 and GRAZ-02, respectively. These are common evaluation metrics for binary classification and correspond to the results reported in [36]. The quality of the solutions is comparable to those reported in the state-of-the-art. Note that the *AVC–CUDA* achieves the same results as the original *AVC* model, which was the expected result after the parallelization.

The idea for the second set of experiments is to compare the performance between the original *AVC* implementation and the *AVC–CUDA* that is proposed in this work. It is interesting to compare the performance of the system

**Table 3** This table reports a comparison between several feature extraction methods together with the AVC and *AVC–CUDA* average performance for the GRAZ-01 dataset

| Methods | Bikes | | Persons | |
|---|---|---|---|---|
| | EER | AUC | EER | AUC |
| EBIM [35] | 84.1 | 90.5 | 86.0 | 91.8 |
| SM [36] | 83.5 | 89.6 | 56.5 | 59.1 |
| HMAX-GA [37] | 80.2 | 88.2 | 84.0 | 90.8 |
| SIFT [36] | 78.0 | 86.5 | 76.5 | 80.8 |
| Moment invariants [36] | 73.5 | 76.5 | 63.0 | 68.7 |
| AVC | 75.8 ±1.3 | 87.8 ±0.9 | 75.1 ±0.6 | 87.7 ±0.7 |
| AVC–CUDA | 75.8 ±1.3 | 87.8 ±0.9 | 75.1 ±0.6 | 87.7 ±0.7 |

**Table 4** This table shows a comparison between several feature extraction methods and the AVC and *AVC–CUDA* average performance computed with the EER measure and the GRAZ-02 dataset

| Methods | Bikes | Persons | Cars |
|---|---|---|---|
| HMAX-GA [37] | 82.6 | 82.3 | 75.6 |
| EBIM [35] | 80.8 | 83.2 | 72.2 |
| Mutch et. al. [38] | 80.5 | 81.7 | 70.1 |
| Basic Moments [36] | 76.5 | 77.2 | 70.2 |
| SIFTs [36] | 76.4 | 10.0 | 68.9 |
| SM [36] | 74.5 | 74.1 | 56.5 |
| Moment invariants [36] | 72.5 | 81.1 | 67.0 |
| AVC | 75.7 ±4.9 | 75.2 ±3.6 | 75.4 ±2.7 |
| AVC–CUDA | 75.7 ±4.9 | 75.2 ±3.6 | 75.4 ±2.7 |

**Table 5** $G01Persons_{368}$ is one of the system instances that was implemented in the original CPU version and reprogrammed with the GPU paradigm of the *AVC* model to compare the computational time requirements of our algorithm

| $G01Persons_{368}$ | |
|---|---|
| $\mathrm{VO}_O$ | $\lvert(I_s - 0.25) - (half(I_g))\rvert$ |
| $\mathrm{VO}_C$ | $\lceil exp(I_c)\rceil$ |
| $\mathrm{VO}_S$ | $(I_g \ominus SE_d)/0.24$ |
| $\mathrm{VO}_{\mathrm{MM}}$ | $\lvert\lvert D_x(\mathrm{CM}) + \mathrm{CM}\rvert - \lvert D_x(D_x(\mathrm{CM}))\rvert\rvert$ |
| | $\mathrm{half}(\mathrm{CM}) - (D_x(D_x(\mathrm{CM})) + D_y(\mathrm{CM}))$ |
| | $D_x(\sqrt{log(\mathrm{CM})})$ |
| | $\lvert\log(D_y(D_y(\mathrm{CM})))\rvert$ |
| | $\sqrt{\lvert D_y(\mathrm{CM})\rvert}$ |
| | $\frac{\sqrt{\mathrm{CM}}}{D_y(D_y(\mathrm{CM}))}$ |
| | $\mathrm{half}(\lvert D_y(CM) + D_y(CM)\rvert)$ |
| | $\lvert\frac{D_y(D_y(\mathrm{CM}))}{D_x(D_x(\mathrm{CM}))} - \mathrm{CM}^2\rvert$ |
| | $G_{\sigma=1}(D_y(D_y(D_y(\mathrm{CM}))))$ |
| | $\log(\log(\mathrm{CM}))$ |

within the most demanding stages of the algorithm such as: the *VMs* calculation, the center-surround procedure to obtain the *CMs*, the visual operations required to evaluate the *MMs* and finally the max search process to build the descriptor vector (*DV*). The comparison for each of these steps as well as the overall execution of the whole system considering several image sizes is presented next.

The image sizes used to compare the proposed system were: $256 \times 256, 512 \times 512, 1024 \times 1024, 2048 \times 2048$, and $4096 \times 4096$. All the 100 solutions were tested 30 times for each image size while comparing the original version with the CUDA-based algorithm in order to verify the computational time between the two implementations. Table 5 shows one of the solutions that was tested. This particular instance of the *AVC* algorithm had three operators for the visual maps and 10 operators that are used for building the mental maps over each of the feature dimensions. The results for time comparison are presented next for all solutions. Note that due to the difference in execution time within the biggest image resolution, it was necessary to plot the results in a semilogarithmic scale.

Figure 4 shows a time comparison of the *VMs* evaluation stage for 100 *AVC–CUDA* models against their original CPU implementations. Note that the time difference grows significantly for the largest image. This is consistent with the theoretical speedup gain of Eqs. (1) and (2), that is achieved through the parallelization of the VO*s*. The theoretical execution time is also shown in the figure as a reference. The execution time was calculated considering the image sizes, a convolution kernel of size 15 and using a system with 256 processors, performing 10 million floating point operations per second (Gigaflops). It is important to note that a single function was considered for the theoretical case, when in reality a *VO* is composed of several functions, that causes the difference between the two curves. Nevertheless, the CUDA execution time follows the expected behavior.

Figure 5 shows the results that were obtained after computing the solutions in the CM level. In this stage, we can observe a big difference between both systems since this is a highly parallel stage and because the transformations performed by the images are constant, hence the importance of the performance gain of the parallelized convolution operation defined by Eq. (3).

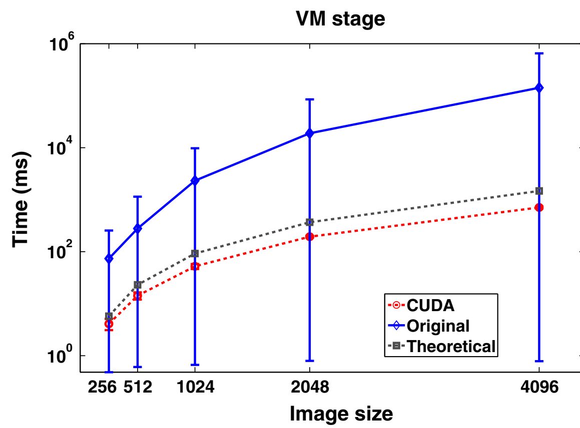


**Fig. 4** Computational time comparison of the VM*s* stage for 100 individuals of the *AVC* model using the CPU and their CUDA implementation. Additionally, the theoretical execution time is depicted as a reference

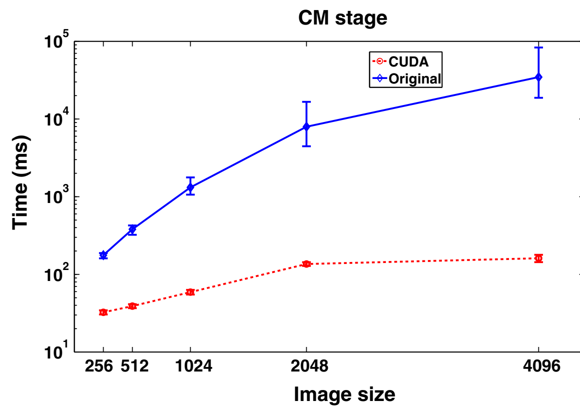


**Fig. 5** Computational time comparison of the CM*s* stage for 100 individuals of the *AVC* model using the CPU and their CUDA implementation

The comparison of the systems for the evaluation of the *MMs* is shown in Fig. 6, where we can observe that the performance gain is similar to that achieved in the *VMs* stage since the nature of the functions is similar to those applied for building the *VOs* of the *MMs*.

The computational time results achieved by the *DV* stage are depicted in Fig. 7. On this plot it is clear that the growth in time for larger images of the GPU-based model is less drastic than the CPU version due to the improvement in its computational order, since it goes from quadratic to logarithmic order; see Eq. (4).

Finally, the overall computational time comparison for both implementations, the original *AVC* model and its CUDA-based parallelized version the *AVC–CUDA* algorithm are presented in Fig. 8. We can appreciate that the difference in performance is significantly larger for the

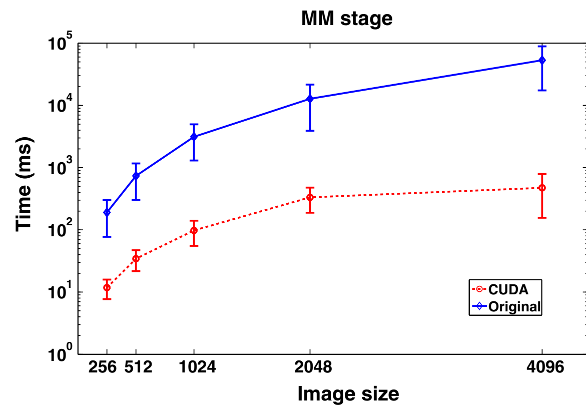


**Fig. 6** Computational time comparison of the MM*s* stage for 100 individuals of the *AVC* model using the CPU and their CUDA implementation

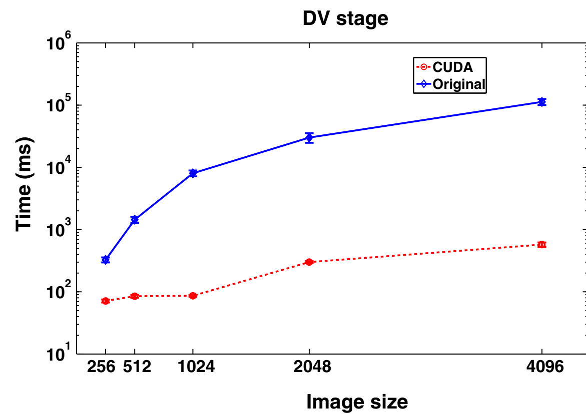


**Fig. 7** Computational time comparison of the *DV* extraction stage for 100 individuals of the *AVC* model using the CPU and their CUDA implementation

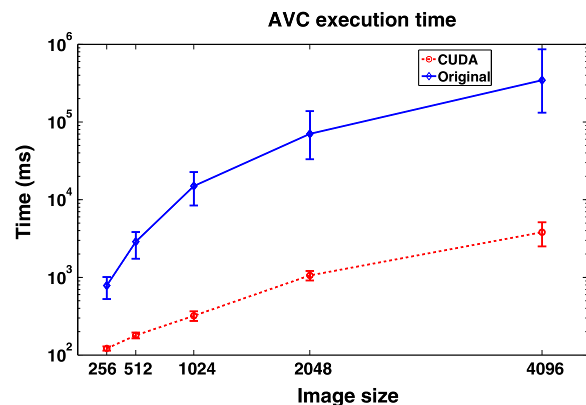


**Fig. 8** This last graph shows the computational time comparison of the *AVC–CUDA* and the original CPU implementation of the *AVC* algorithms

bigger images. So the results for the image size of 256 × 256 of the *AVC–CUDA* model is 6 times faster on average than the original model. In the case of an image resolution of 512 × 512 the system runs 16 times faster, while the GPU version is 47 times faster than the CPU implementation for an image resolution of 1024 × 1024. Finally, the CUDA-based model is 70 and 90 faster than the original model for the image sizes of 2048 × 2048 and 4096 × 4096, respectively.

### 3.1 Performance comparison with CNN and HMAX models

In this section, we compare our system with two state-of-the-art methods based on CUDA implementations. The *HMAX–CUDA* model presented by Mutch et al. [18] and the CUDA-CNN implementation presented by Sirotenko et al. [39]. On the one hand, the *HMAX–CUDA* system was implemented to work over the Caltech 101 and GRAZ-01 image datasets. This system has the constraint of being designed to function over relatively small images. The system only accepts images of up to 512 × 512 pixels, hence it resizes any input image larger than that in order to process it. On the other hand, the CUDA-CNN system was developed to work on the MNIST digits dataset. Therefore, this system focuses on large sets of small images. The system processes thousands of images with a resolution of 28 × 28 pixels for the provided dataset, but it is capable of processing images of up to 64 × 64 pixels. Due to image size limitations for both systems, we decided to compare our system by using two image sizes, 256 × 256 and 512 × 512 pixels. Since the CNN system works with images of 64 × 64 pixels, a fair comparison would be to test the system using the same amount of information treated by the other two techniques while working with larger images. Hence, for the test with the images of 256 × 256 pixels, the CNN system processes 16 images of size 64 × 64. Similarly, for images of 512 × 512 pixels the CNN system processed 64 small images. As a result, the three systems were executed one hundred times for each image size and the execution time comparison can be observed in Fig. 9. Note that the *HMAX–CUDA* system provides almost the same execution time for both image sizes. This is due to the fact that only the first stage of the system uses the original image size, the most expensive tasks are processed in all subsequent stages of the model whose images are of 256 × 256 pixels. Thus, resulting in similar processing times for both image sizes.

The results show that our system outperforms both methods in the two tests. In the case of images of 256 × 256 pixels the *AVC–CUDA* is almost 10 times faster than the *HMAX–CUDA* model, and its performance results are very similar to the *CNN–CUDA* model. Similarly, for the

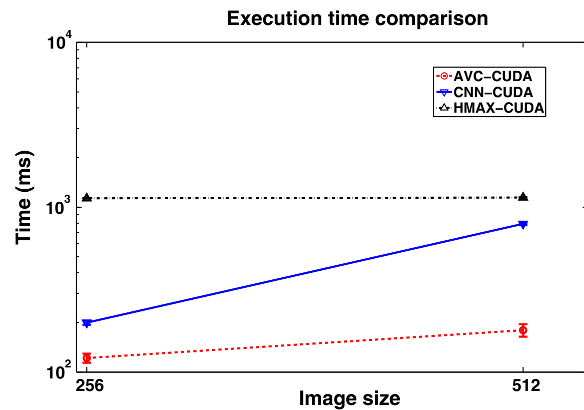


**Fig. 9** This graph shows the computational time comparison of the *CNN–CUDA* system presented in [39] and the *HMAX–CUDA* model from [18], against our proposed system

case of images of 512 × 512 pixels, our implementation is five times faster than the *HMAX − CUDA* method and four times faster than the *CNN–CUDA* system.

## 4 Conclusions and future work

The results obtained in our work show that the CUDA base implementation of the *AVC* model considerably improves the computational performance of the algorithm. The proposed approach follows a parallelization principle focusing on the numerous transformations that the AVC model applies to replicate an artificial visual cortex. The new algorithm achieves a consistent speedup, and the results are similar to those reported in other models working on image-related problems and that have been implemented with the CUDA framework without sacrificing its classification capabilities. Note that it is now possible to exploit the classification efficiency of the *AVC* model in a real-time environment since it is now capable of achieving a processing speed of three images per second using an image resolution of 1024 × 1024, five images per second for an image resolution of 512 × 512 and up to 10 images per second considering the smaller tested resolution of 256 × 256. Furthermore, the system was compared against other CUDA-based techniques for images sizes of 256 × 256 and 512 × 512 pixels. Our results outperform both systems from an execution time perspective for both test cases.

Further improvements can be achieved with our implementation. A first advance could be to parallelize fully the complete system by processing the four-feature dimension at the same time, either by including several graphics cards on the same computer or by establishing a computer cluster where each node will be in charge of evaluating the

corresponding maps at each dimension. Another important step could be to apply the model in a real-world application.

**Acknowledgements** This research was funded by CONACyT through Project 155045—"Evolución de Cerebros Artificiales en Visión por Computadora" and by CICESE project number 634119. Dr. Olague graciously acknowledges the support of the Seventh Framework Programme of the European Union through the Marie Curie International Research Staff Scheme, FP7-PEOPLE-2013-IRSES, Grant 612689 ACoBSEC, project Analysis and Classification of Mental States of Vigilance with Evolutionary Computation.

## References

1. Hsuan L, Chih-Yin L, Chih-Jui L, Chien-Feng H, Ri-Wei D, Tzuu-Hseng S (2014) Implementation of real-time object recognition system for home-service robot by integrating SURF and BRISK. IEEE international conference on system science and engineering (ICSSE), pp 273–278
2. Muller UA, Jackel LD, LeCun Y, Flepp B (2013) Real-time adaptive off-road vehicle navigation and terrain classification. Proc SPIE, Unmanned Syst Technol XV 8741:87410A–1
3. Salas-Moreno RF, Newcombe RA, Strasdat H, Kelly PHJ, Davison AJ (2013) SLAM++: simultaneous localisation and mapping at the level of objects. IEEE conference on computer vision and pattern recognition (CVPR) 1352(1359):23–28
4. Zhang X, Yang Y-H, Han Z, Wang H, Gao C (2013) Object class detection: a survey. ACM Comput Surv 46(1, 10):53
5. Olague G (2016) Evolutionary computer vision: the first footprints. Springer, Berlin, p 411
6. Dozal L, Olague G, Clemente E, Hernández DE (2014) Brain programming for the evolution of an artificial dorsal stream. Cognit Comput 6(3):528–557
7. Hernández DE, Clemente E, Olague G, Briseño JL (2016) Evolutionary multi-objective visual cortex for object classification in natural images. J Comput Sci 17(1):216–233
8. Malik ZK, Hussain A, Wu J (2014) Novel biologically inspired approaches to extracting online information from temporal data. Cognit Comput 6(3):595–607
9. Yang Y, Wu QMJ, Wang Y, Zeeshan KM, Lin X, Yuan X (2014) Data partition learning with multiple extreme learning machines. IEEE Trans Cybern 45(8):1463–1475
10. Ng LT, Suandi SA, Teoh SS (2014) Vehicle classification using visual background extractor and multi-class support vector machines. The 8th international conference on robotic, vision, signal processing & power applications. Lecture notes in electrical engineering, vol 291, pp 221–227
11. Asano S, Maruyama T, Yamaguchi Y (2009) Performance comparison of FPGA, GPU and CPU in image processing. International conference on field programmable logic and applications. FPL, pp 126–131
12. Lu M (2013) Fast implementation of scale invariant feature transform based on CUDA. Appl Math Inf Sci 7(2):717–722
13. Chase J, Nelson B, Bodily J, Wei Z, Lee D (2008) Real-time optical flow calculations on FPGA and GPU architectures: a comparison study. 16th international symposium on field-programmable custom computing machines. FCCM '08, pp 173–182
14. Jiang S, Wang Y, Chen Z, Sun K (2014) Real-time brain extraction method from cerebral MRI volume based on graphic processing units. Neural Comput Appl 25(5):1145–1151
15. Ohmer J, Maire F, Brown R (2006) Implementation of kernel methods on the GPU. Digital image computing: techniques and applications, DICTA 05. Queensland, Australia, pp 1–8
16. Uetz R, Behnke S (2009) Large-scale object recognition with CUDA-accelerated hierarchical neural networks. IEEE international conference on intelligent computing and intelligent systems, ICIS 09, pp 1–6
17. Chikkerur S (2008) CUDA implementation of a biologically inspired object recognition system. MIT technical report
18. Mutch J, Knoblich U, Poggio T (2010) CNS: a GPU-based framework for simulating cortically-organized networks. MIT technical report
19. Parks D, Jain A, McInerney J, Itti L (2010) GPGPU-based real-time object detection and recognition system. J Vis 10(7):997
20. Woodbeck K, Roth G, Huiqiong C (2008) Visual cortex on the GPU: biologically inspired classifier and feature descriptor for rapid recognition. IEEE computer society conference on computer vision and pattern recognition workshops. CVPRW '08, pp 1–8
21. Olague G, Clemente E, Dozal L (2014) Evolving an artificial visual cortex for object recognition with brain programming. EVOLVE—A bridge between probability set oriented numeric and evolutionary computation III, studies in computational intelligence, Springer, vol 500, pp 97–119
22. LeCun Y, Bottou L, Orr GB, Müller KR (1998) Efficient backprop. Neural Netw Tricks Trade LNCS 1524:9–50
23. Bouvrie J (2006) Notes on convolutional neural networks. Center for biological and computational learning, MIT tutorial
24. Lecun Y, Bottou L, Bengio Y, Haffner P (1998) Gradient-based learning applied to document recognition. Proc IEEE 86(11):2278–2324
25. Ciresan D, Meier U, Masci J, Gambardella LM, Schmidhuber J (2011) Flexible, high performance convolutional neural networks for image classification. IJCAI'11 proceedings of the twenty-second international joint conference on artificial intelligence, vol 2, pp 1237 –1242
26. Srivastava N, Hinton G, Krizhevsky A, Sutskever I, Salakhutdinov R (2014) Dropout: a simple way to prevent neural networks from overfitting. J Mach Learn Res 15(10):1929–1958
27. Dhanasekaran B, Rubin N (2011) A new method for GPU based irregular reductions and its application to K-means clustering. In: Proceedings of the fourth workshop on general purpose processing on graphics processing units, vol 2, pp 2- 8
28. Xiao Y, Leung CS, Ho T, Lam P (2011) A GPU implementation for LBG and SOM training. Neural Comput Appl 20(7):1035–1042
29. Nantes A, Brown R, Maire F (2013) Neural network-based detection of virtual environment anomalies. Neural Comput Appl 23(6):1711–1728
30. Milner AD, Goodale MA (2006) The visual brain in action, 2nd edn. Oxford University Press, Oxford, p 297
31. Clemente E, Chavez F, Fernandez de Vega F, Olague G (2015) Self-adjusting focus of attention in combination with a genetic fuzzy system for improving a laser environment control device system. Appl Soft Comput 32:250–265
32. Olesen SM, Lyder S (2010) Applying 2D filters using GPU's and CUDA. Vision 3—advanced topics in computer vision
33. Sanders J, Kandrot E (2010) CUDA by example: an introduction to general-purpose gpu programming. Addison-Wesley, Boston, p 290
34. Podlozhnyuk V (2007) Image convolution with CUDA. NVIDIA corporation technical report, pp 1–21
35. Huang Y, Huang K, Tao D, Tan T, Li X (2011) Enhanced biologically inspired model for object recognition. IEEE Trans Syst Man Cybern Part B 41(6):1668–1680
36. Opelt A, Pinz A, Fussenegger M, Auer P (2006) Generic object recognition with boosting. IEEE Trans Pattern Anal Mach Intell 28(3):416–431

37. Ghodrati M, Khaligh-Razavi S, Ebrahimpour R, Rajaei K, Pooyan M (2012) How can selection of biologically inspired features improve the performance of a robust object recognition model? Plos ONE 7(2):1–15
38. Mutch J, Lowe DG (2008) Object class recognition and localization using sparse features with limited receptive fields. Int J Comput Vis 80(1):45–57
39. Pshikhopov VKh, Medvedev MY, Sirotenko MY, Kostjukov VA (2009) Control system design for robotic airship. 9th IFAC symposium on robot control. Gifu, Japon, vol 42, no 16, pp 26–31