Experiences accelerating features selection in Viola-Jones algorithm

Germán Lescano[1,2], Pablo Santana-Mansilla[1,2], Rosanna Costaguta[1]

[1]Instituto de Investigación en Informática y Sistemas de Información (IIISI)
Facultad de Ciencias Exactas y Tecnologías (FCEyT)
Universidad Nacional de Santiago del Estero (UNSE)
[2]Consejo Nacional de Investigaciones Científicas y Tecnológicas (CONICET)
{gelescano,psantana,rosanna}@unse.edu.ar

Abstract. Faces and facial expressions recognition is an interesting topic for researchers in machine vision. Viola-Jones algorithm is the most spread algorithm for this task. Building a classification model for face recognition can take many years if the implementation of its training phase is not appropriately optimized. In this study, several settings for implementing the training phase are analyzed. The aim was to share our experiences when we try to accelerate the training phase using one computer with a graphical processing unit (GPU). For each setting, the execution times were analyzed and compared with previous studies. Although we don't contribute to break new ground in topic or methodology, we decide to share our experience in order to show an antecedent working with a cheap GPU with the aim that this can be useful to another for to make comparisons.

Keywords: Adaboost, Viola-Jones Algorithm, feature selection, CUDA

1 Introduction

Face and facial expressions recognition is an interesting topic for researchers in machine vision [9]. An important stage in a face recognition algorithm is the building of a classification model that can discriminate faces. Building a classification model require a training phase during which a sample of images is analyzed with the aim of extracting those features that best describe a face.

Viola and Jones [11] proposed an algorithm that can detect faces in real time. This algorithm can be implemented on a wide range of small low power devices, including hand-helds devices and embedded processors. However, a drawback of this algorithm is that the training phase is extremely time-consuming.

In this work, we propose and analyze five settings to implement the training phase of Viola-Jones algorithm. Each setting tries to reduce the execution times when working on a single computer. Two settings involve the use of sequential computing and the other three involve the use of parallel computing, specifically CUDA architecture. CUDA is a parallel computing platform and programming model invented by NVIDIA [12]. It enables dramatic increases in computing performance by harnessing

the power of the graphics-processing unit (GPU). In order to reduce execution times, we had focus on feature selection because this process has a notable impact on training times.

This paper is organized as follows. In section 2, the purpose, the utility and the training phase of Viola-Jones algorithm are described. In section 3, five settings of development for training face are described. In section 4, our experimental results are showed. In section 5, we compare our proposal respect other alternatives we found in the literature. In section 6, conclusions and the future research directions are presented.

2 Revision of the training phase in Viola-Jones Algorithm.

The Viola-Jones algorithm describes a framework for object detection. It is widely used in a variety of software and hardware applications that incorporate elements of computer vision, like the face detection module in video conferencing, human-computer interaction, and digital photo cameras [6].

Viola and Jones [11] propose a variant of Adaboost algorithm [2] for the training phase that is related with the selection of a small number of features that best describe a face. These features are known as weak classification functions and are combined to build a stronger classifier.

The features used by Viola and Jones are reminiscent of Haar basis functions [8]. Figure 1 shows five simple features usually employed. These features are defined by two, three or four rectangles. To compute a feature, the sum of pixels within the white rectangles should be subtracted from the sum of pixels within the black rectangles.

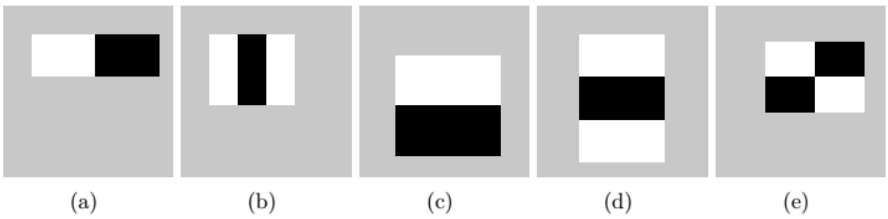

Fig. 1. Haar-Like patterns frequently employed [11].

Supposing f is a feature, θ is a threshold, $p \in \{-1,1\}$ is the polarity that indicates the direction of the inequality and x is a sub-window of an image, a weak classifier to detect a face in x can be defined by Equation 2.

$$h(x, f, p, \theta) = \begin{cases} 1 \; si \; pf(x) < p\theta \\ 0 \; en \; otro \; caso \end{cases} \qquad (2)$$

Each iteration of the boosting algorithm is designed to select the single rectangle feature which best separate the positive (face images) and negative examples (not face images). For each feature, the weak classifier determines the optimal threshold of

classification function, such that the minimum number of examples is misclassified. Figure 2 shows the boosting algorithm proposed by Viola and Jones [11].

Boosting Algorithm

- Given example images (x_1,y_1), …, (x_n,y_n) where y_i=0, 1 for negative and positive examples respectively..
- Initialize weights $w_{1,i} = \frac{1}{2m}, \frac{1}{2l}$ for y_i=0,1 respectively. Where *m* is the number of positives examples and *l* the number of negatives examples.
- For t = 1,..,T (T weak classifiers)

1. Normalize the weights, $w_{t,i} = \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$
2. Select the best weak classifier with respect to the weighted error:

$$\varepsilon_t = min_{f,p,\theta} \sum_i w_i |h(x_i, f, p, \theta) - y_i|$$

Define $h_t(x)=h(x,f_t,p_t,\theta_t)$ where $f_t p_t y\ \theta_t$ are the minimizers of ε_t.

3. Update the weights: $w_{t+1,i} = w_{t,i}\beta_t^{1-e_i}$
Where e_i=0 if example x_i is classified correctly, e_i = 1 otherwise, and $\beta_t = \frac{\varepsilon_t}{1-\varepsilon_t}$

- The final strong classifier is:

$$C(x) = \begin{cases} 1\ if\ \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2}\sum_{t=1}^{T} \alpha_t \\ 0\ otherwise \end{cases}$$

Where $\alpha_t = log\frac{1}{\alpha}$

Fig. 2. Boosting Algorithm [11]

Viola-Jones algorithm to select weak classifier is depicted in Figure 3. This pseudo-code is an interpretation proposed by Morelli and Padovani [5] and its operation can be summarized as follow. For each feature, the images into the samples are sorted by feature value in ascendant way. The Adaboost optimal threshold for that feature can then be computed in a single pass over this sorted list. On each iteration over the sorted list, four sums are evaluated for each element: the total sum of positive examples weights T^+, the total sum of negative example weights T^-, the sum of positive weights below the current example S^+ and the sum of negative weights below the current example S^-. Also for each feature, an error is computed using the Equation 3. This value represents the error that we would produce if the element were considered the threshold for the feature. Once all errors have been computed, the lowest one is selected.

$$\varepsilon = \min\left(S^+ + (T^- - S^-), S^- + (T^+ - S^+)\right) \qquad (3)$$

$selectedFeature = \emptyset$
$selectedFeatureError = \infty$
$selectedThreshold = 0$
$T^+ \leftarrow total\ sum\ of\ positive\ examples\ weights$ $(\mathbf{1})$
$T^- \leftarrow total\ sum\ of\ negative\ examples\ weights$ $(\mathbf{2})$
for $all\ f\ \varepsilon\ features$ **do**
  $X^{(f)} \leftarrow training\ images\ sorted\ by\ f\ value$ $(\mathbf{3})$
  $e_f = \infty$
  $\theta_f = 0$
  for $i = 1$ **to** N
    $S_i^+ \leftarrow the\ sum\ of\ positive\ weights\ below\ the\ current\ example\ x_i^f$ $(\mathbf{4})$
    $S_i^- \leftarrow the\ sum\ of\ negative\ weights\ below\ the\ current\ example\ x_i^f$ $(\mathbf{5})$
    $e_i = \min\left(S_i^+ - (T^- + S_i^-), S_i^- - (T^+ - S_i^+)\right)$ $(\mathbf{6})$
    if $\left(e_i - e_f\right)$ **then**
      $e_f = e_i$
      $\theta_f = f\left(x_i^f\right)$
    end if
  end for
  if $selectedFeatureError > e_f$ **then**
    $selectedFeature = f$
    $selectedFeatureError = e_f$
    $selectedThreshold = \theta_f$
  end if
end for

Fig. 3. Algorithm for weak classifiers selection [5].

3 Experimental Settings for Implementing the Training Phase

Five experimental settings were proposed to implement the training phase of Viola-Jones algorithm. The C language was chosen to code each experimental setting because it enables us to make decision of programming at low-level and it is compatible with CUDA.

3.1 First Experimental Setting

The implementation that was done with this experimental setting was sequential. We allocated in memory a matrix that stores the identifier of each training file, the category of the image (face or not face), its weight and the category (face or not face) assigned by the classifier algorithm. In addition, we use as many files as training images in order to save the features of the images. Each file has 162336 lines (this number correspond to the total quantity of features that can be generated for an image of 24 x 24 pixels) and each line registers the details of one feature.

In Figure 3, the instructions that can have in memory the data necessary for computing them were tagged with (1), (2), (4), (5) and (6). Meanwhile, the instruction tagged with (3) has a lot of access to hard disk because it needs data saved in files. In instruction (3) the sorting task is made through the Bubble sort method, so we have a loop that iterates over 162336 features. In each iteration, the values of a feature are extracted from each training image file. These values are employed to re-sort the training images. The access to details of a feature in the file is sequential.

3.2 Second Experimental Setting

This experimental setting is similar to the first one in the sense that it operates with data saved on files. The difference is that with this second setting we create one file to save the values of all features on a unique place. Each line of this file contains all values of a feature extracted from each training image. This change reduces in a 99,99% the amount of access to files, from 2.435.040.000 (162336 features x 15000 sample images) to 162336, during the sorting task. In this second setting, we changed the Bubble method for the Quick Sort method because the last one generally has best performance during sorting tasks. The way of accessing to the feature values was also changed. Sequential access was changed for random access through the *fseek* function.

3.3 Third Experimental Setting

This experimental setting continues with the use of a unique file in order to store for each feature the values that are observed in each one of the sample files. However, we employ the Thrust library. Thrust is a parallel algorithms library that enhances programmer productivity while enabling performance portability between GPUs and multicore CPUs [13]. This library was used for sorting, operation tagged with (5) in Figure 3, and in (4) and (6) operations of this figure for accomplish them through a reduction operation.

3.4 Fourth Experimental Setting

In this experimentation, we changed the way of applying the sum operations involved in the process of weak classifiers selection. The operations to sum the weight of positive samples (4) and negative samples (5) of Figure 3 were implemented in the

same procedure so as to take advantage of the processing cycle, and the procedure was parallelized based on the algorithm proposed by Harris [3]. Small changes were applied to the algorithm in order to make two sum operations at the same time.

This experimental setting does not sort the values of the samples with respect to a particular feature. For each value in the samples with respect to a particular feature, the calculation of $\varepsilon = \min(S^+ + (T^- - S^-), S^- + (T^+ - S^+))$ was made in fully parallel way. This way 15000 processing threads were launched, one for each sample.

3.5 Fifth Experimental Setting

On previous experimentations, the data matrix was stored in a disk file, but on this experimental setting, all data are stored in the memory of the computer. To be sure data are not paged for the operative system we use *cudaHostAlloc* instruction that allocates a buffer of page-locked host memory [7].

The concept of stream it is also used. A CUDA stream represents a queue of GPU operations that are executed in a specific order [7]. On each execution, a stream is responsible for processing a row of the matrix, asynchronously loading the data into memory. Two kernels are launched at the same time when data are loaded, one kernel sorts the files of the samples considering the value of the analyzed feature. Since the order of sample files can vary according to the feature and that kernels execution is asynchronous, it is necessary to use other matrix to keep a reference to the order with regard to the feature that the stream analyzes. In this second matrix, each row represents a feature and it stores indexes of the files corresponding to the samples. This matrix is also stored in pinned memory and it is copied in asynchronous way into the streams.

Once the sorting kernel is launched, the following step consists on launching a kernel for performing operations (4), (5) and (6) of Figure 3 for each value of sample files for a particular feature. The results of these calculations are copied asynchronously to an array that stores, for each feature, the mistake that would commit if the value in the sample for this feature was chosen as threshold value. This array is stored in pinned memory.

At the end of the execution of all the streams, the *min_element* function of Thrust library is used to find the smallest mistake in the above mentioned array. This final step allow one to get the selected feature, the threshold value and its corresponding mistake.

4 Results

Table 1 shows the results of executing the experimental settings designed in a computer with the following hardware configuration: i7 processor of 3.4 Ghz and GPU GeForce GT 730 with 2 GB of global memory. The numbers into the first column of Table 1 are operations tagged in Figure 3.

In the first setting, the amount of access has a notable influence in execution times because if we have 162336 features and 15000 training image, then we will require

2435040000 accesses to files. In the second setting, the amount of accesses to files reduces in 99.99% and this enables the reduction of the execution times. However, in the third setting, the execution times go to the bad. In the third setting, the times for making the sorting task are reduced but we lose performance when computing the error for obtaining the threshold for a feature. In the fourth setting, the execution times are improved again and they are better than second setting. This was achieved because in the fourth setting we did not make the sorting task and the execution times for calculating the threshold was reduced. Overall, we got the best performance in the fifth setting. If we compare fifth setting with setting number 4, the fifth setting represents an increasing of about 11x in speed. It seems reasonable to assume that this improvement in performance was obtained thanks to the use of streams and pinned memory.

Table 1. Comparison of time needed to execute the different operations of the method for selecting the weak classifier

Operations	**Time (milliseconds)**				
	Exp. Setting 1	**Exp. Setting 2**	**Exp. Setting 3**	**Exp. Setting 4**	**Exp. Setting 5**
(3)	938993.904	3.437	0.540	--	44.381 (8192 streams)
(4), (5) y (6)	0.046	0.046	0.601	0.019	
Approximate time to evaluate a feature	939682.127	696.716	9090.446	519.515	
Approximate time to find a weak classifier	4.84 years	31.42 hs	17.08 days	23.43 hs	2 hs

5 Discussion

A bibliographical exploration regarding to time reduction when building images classifiers with boosting algorithm allowed us to identify research works such as Huang and Shi [4], Abualkibash et al. [1] and Tsai et al. [10]. Huang and Shi [4] report the results that are showed in Table 2 when working with 65230 features and 18676 samples. In their experiments Huang and Shi used computers with a 1.8 Ghz processor.

Abualkibash et al. [1] describe the same experiment that was made by Huang and Shi [4]. Abualkibash et al. [1] utilized computers equipped with quad-core processors but they did not give details about processing capacity of CPUs. Table 3 shows the results reached by Abualkibash et al. [1].

Tsai et al. [10] experimented with a GPU Nvidia Tesla K20c. This GPU, according to technical documentation, has 2496 cores, 706 Mhz of memory frequency, and 5 GB of global memory. Tsai et al. [10] made three experiments: the first one with 1119 features and 19575 samples; the second experiment with 6090 features and 19161

features; and the last experiment with 10640 features and 19140 samples. The results of these three experiments can be seen in Table 4.

Table 2. Comparison between execution time of alternative 5 and execution time during experiments conducted by Huang and Shi [4]

Time to select a feature	
(Huang et al. 2010)	**Alternative 5 implemented in Geforce GT 730 of 2GB memory**
3.31 minutes (when working with 2 computers)	38.52 minutes (when working with 8192 streams)
1.97 minutes (when working with 4 computers)	

Table 3. Comparison between execution time of alternative 5 and execution time during experiments conducted by Abualkibash et al. [1]

Time to select a feature	
(Abualkibash et al. 2013)	**Alternative 5 implemented in Geforce GT 730 of 2GB memory**
24.6 seconds (when working with 6 computers)	38.52 min (when working with 8192 streams)
6.4 seconds (when working with 21computers)	
5.2 seconds (when working with 26 computers)	
4.8 seconds (when working with 31 computers)	

Table 4. Comparison between execution time of alternative 5 and execution time during experiments conducted by Tsai et al. [10]

Experiments	**Time to select a feature**	
	Tsai et al. [10]	**Alternative 5 implemented in Geforce GT 730 of 2GB memory**
1119 features 19575 samples	0.788 seg	1.413 min with 1119 streams
6090 features 19161 samples	1.157 seg	7.313 min 4096 streams
10640 features 19140 samples	1.217 seg	6.2497 min Con 8192 streams

Considering results that are showed in Tables 2, 3 and 4 one could conclude that the fifth experimental setting proposed in this paper is not a good option to reduce the time of classifiers building. Nevertheless, one should consider that the fifth alternative

of solution was tested in a computer with a GPU of limited processing capacity. For this reason, it would be interesting to run the fifth solution in a computer with better hardware capacities and then compare it with the results achieved by Huang and Shi [4], Abualkibash et al. [1] and Tsai et al. [10]. For example, a GPU like the one used by Tsai et al. [10] not only has more computing capacity but also it is possible to execute more streams.

Considering it was wanted to execute the fifth experimental setting in a computer with more computing capacity and that the laboratory at the university had not have more powerful computers than the described in point 4 (i7 processor of 3.4 Ghz and GPU Geforce GT 730), it was decided to hired the computer instance g2.2xlarge from amazon.com. The instance g2.2xlarge has: 8 virtual CPU Intel Xeon E5-2670; 15 GB of RAM; and a GPU Nvidia Grid K520 with 4 GB of RAM, 797 Mhz of frequency and 1536 cores. Table 5 shows a comparison of time needed for executing the experiments of Table 4 (the same amount of features and sample files) in the hired instance and the GPU GeForce GT 730. It was expected that execution in the GPU available in g2.2xlarge instance would be faster than the execution in GeForce GT 730. Nonetheless, the results do not confirm the previous assumption. A reason might be the virtualization effect although this needs further research.

Table 5. Comparison of execution time when implementing solution 5 in GeForce GT 730 and in instance g2.2xlarge.

Experiments	**Time to select a feature (Alternative 5 implemented in GeForce Gt 730)**	**Time to select a feature (Alternative 5 implemented in GPU of g2.2xlarge)**
1119 features 19575 samples	1.413 min	4.1789 min
6090 features 19161 samples	7.313 min	22.039 min
10640 features 19140 samples	6.2497 min	38.472 min

6 Conclusions

This work analyzed the implementation of a training process for generating a classifier with the capacity of face recognition. The aim was to reduce the time needed to train the classifier using a single computer. The focus was the process for selection of weak classifiers because this stage is the most invoked during classifiers building and it is the most demanding in terms of execution time. With the several alternatives implemented, sequential and parallel through CUDA architecture, it was possible to achieve substantial improvements.

The use of GPUs for developing and accelerating applications is a feasible alternative because any programmer can acquire one and there are cheap GPU. In addition, there is an effort made by manufacturer of GPU for development software that helps

programmers to use them, for example NVIDIA with her CUDA platform. With the use a GPU, applications can compute many data without wasting too much time.

Further research may involve the measurement of the performance of fifth experimental setting in a computer with bigger computing capacity in its GPU and exploring a multi-GPU approach. Collaterally, we believe that further research is needed for evaluate if virtualization affect in the performance GPU.

7 References

1. Abualkibash, M.; ElSayed, A.; Mahmood, A.: Highly scalable, parallel and distributed Adaboost algorithm using light weight threads and web services on a network of multi-core machines. International Journal of Distributed and Parallel Systems (IJDPS), 4(3): 29-40, May 2013.
2. Freund, Y.; Schapire, R.E.: A decision-theoretic generalization of on-line learning and an application to boosting. Journal of Computer and System Sciences, 55(1): 23-37. August 1997.
3. Harris, M.: Optimizing parallel reduction in CUDA. Reporte técnico. 2007. Disponible en: http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf
4. Huang, Z.; Shi, X.: A distributed parallel AdaBoost algorithm for face detection. 2010 IEEE International Conference on Intelligent Computing and Intelligent Systems (ICIS). Vol 1: 147-150, Oct 2010.
5. Morelli A., Padovani S.: Detección y Reconocimiento de Cara. Tesis de Licenciatura en Ciencias de la Computación. Universidad de Buenos Aires. 2011.
6. Obukhov, A.: Haar Classifiers for Object Detection with CUDA. In: Wen-Mei W. Hwu (Ed.), GPU Computing Gems. 517-544. Burlington, MA 01803, USA, 2011.
7. Sanders, J.; Kandrot, E.: CUDA C on multiple GPUs. In: CUDA by Example. An Introduction to General-Purpose GPU Programming. 213-236. Boston, MA 02116, USA, 2011.
8. Papageorgiou, C.; Oren, M.; Poggio, T.: A general framework for object detection. International Conference on Computer Vision. 555-562, 04 January - 07 January, 1998.
9. Taheri, S.; Patel, V.; Chellappa, R.: Component-Based Recognition of Faces and Facial Expressions. IEEE Transactions on A ffective Computing, 4(4): pp. 360-371, October-December 2013.
10. Tsai, P.; Hsu, Y.; Chiu, C; Chu, T.: Accelerating AdaBoost algorithm using GPU for multi-object recognition. 2015 IEEE International Symposium on C ircuits and Systems (ISCAS), 738-741, May 2015.
11. Viola P., Jones M.: Robust Real-Time Face Detection. International Journal of Computer Vision, 57(2): 137–154, May 2004.
12. NVidia CUDA technology, http://www.nvidia.com/object/cuda_home_new.html
13. Thrust, http://thrust.github.io/