Local Binary Pattern Descriptor

Simone Casini
E-mail address
simone.casini3@stud.unifi.it

Giacomo Ravara
E-mail address
giacomo.ravara@stud.unifi.it

Abstract

The Local Binary Pattern is an image descriptor. The procedure to calculate it is sequential by its definition, but a Multi-Block version can be used to take advantage of the modern multicore CPUs and CUDA development.

1. Introduction

Local Binary Pattern (LBP) is an effective texture descriptor for images which thresholds the neighboring pixels based on the value of the current pixel. The original LBP operator (Ojala et al. 1996[3]) forms labels for the image pixels by thresholding the 3 x 3 neighborhood of each pixel with the center value and considering the result as a binary number. The histogram of these 2^8 = 256 different labels can then be used as a texture descriptor. This operator used jointly with a simple local contrast measure provided very good performance in unsupervised texture segmentation (Ojala and Pietikäinen 1999 [2]). After this, many related approaches have been developed for texture and color texture segmentation.

The LBP operator was extended to use neighborhoods of different sizes (Ojala et al. 2002). Using a circular neighborhood and bilinearly interpolating values at non-integer pixel coordinates allow any radius and number of pixels in the neighborhood. So the sequence of operations is the following:

1. For every pixel in an image, I, choose P neighboring pixels at a radius R.
2. Calculate the intensity difference of the current pixel with the P neighboring pixels.
3. Threshold the intensity difference, such that all the negative differences are assigned 0 and all the positive differences are assigned 1, forming a bit vector.
4. Convert the P-bit vector to its corresponding decimal value and replace the intensity value at with this decimal value.

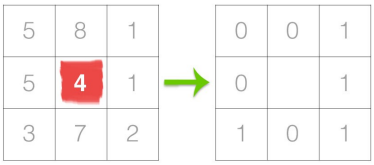

Figure 1: Example of step 3

Thus, the LBP descriptor for every pixel is given as

$$LBP(P,R) = \sum_{p=0}^{P-1} f(g_p - g_c)2^p$$

where g_p and g_c denote the intensity of the current and neighboring pixel, respectively. P is the number of neighboring pixels chosen at a radius R.

Algorithm 1 LBP Sequential

procedure LBP($pixels, samples, radium$)
  for all $pixels$ **do**
    NEIGHBORHOOD ($pixels, samples, radium$)
    THRESHOLD ($pixels, samples, radium$)

1.1. Multi-block LBP

In the LBP approach for texture classification, the occurrences of the LBP codes in an image are collected into a histogram. The classification is then performed by computing simple histogram similarities. However, considering a similar approach for facial image representation results in a loss of spatial information and therefore one should codify the texture information while retaining also their locations. One way to achieve this goal is to use the LBP texture descriptors to build several local descriptions of the face and combine them into a global description. Such local descriptions have been gaining interest lately which is understandable given the limitations of the holistic representations. These

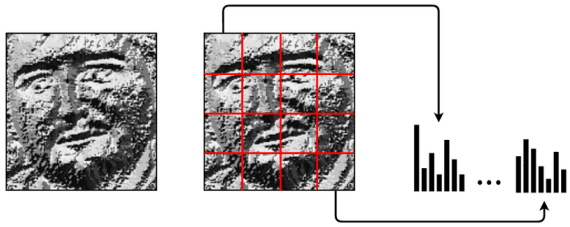

Figure 2: MB-LBP

local feature based methods are more robust against variations in pose or illumination than holistic methods. The basic methodology for LBP based face description proposed by Ahonen et al. (2006) is as follows: The facial image is divided into local regions (**blocks**) and LBP texture descriptors are extracted from each region independently. The descriptors (the histograms) are then concatenated to form a global description of the image.

2. Implementation

2.1. Preprocessing

Before performing the actual processing with the calculation of the LBP descriptor, some previous operations are necessary. The first step is to convert the image to grayscale. Then it is useful to identify the *offset vector* (based on the number of samples chosen and the radius), the elements of which will be added to the position of the pixel for the calculation of the descriptor. Each offset is identified by an integer approximation of $radius * cos(sample * angle))$ where angle is $2\pi/numberofsamples$.
It then becomes necessary to define the sub-image in which to actually calculate the descriptor, since for pixels less than the radius from the edges of the image, it becomes impossible to have the pixels at offset distance. So the processed image will have dimensions: $(height - 2 * radius) * (weight - 2 * radius)$.
Given the block dimension as a parameter, it is defined the number of the local regions used in multi-block LBP, given as: $(height - 2 * radius) * (weight - 2 * radius)/(blockEdge^2)$. In the Multi-Block LBP version each image block updates its histogram. Therefore the histogram lenght for a local sub-region is then defined as 2^P and the global histograms is given as $(HistogramLenght) * (HistogramNumber)$.

3. Parallel Version C++

In a parallel view, the multi-block variant leads to another important advantage. The processing becomes easily parallelizable and moreover, if each block has its own histogram, there aren't race condition problems. In the parallel version the array of pixels obtained from the image in input is divided between the threads. In this way each thread works on a part of the image, applying the LBP operator.

Algorithm 2 LBP Parallel Version

procedure LBP($pixels, samples, radium, edge$)
  for $t < num\_threads$ **do**
    THREADLBP ($t, pixels, samples, radium, edge$)
    COMPUTEHISTOGRAM
  return Histogram

Algorithm 3 LBP Parallel Thread

procedure THREADLBP($pixels, samples, radium, edge$)
  $region_t$ = COMPUTEREGION($pixels$)
  COMPUTELBP($region_t, samples, radium, edge$)
return LocalHistogram

When the threads have terminated the local histograms are used to define a global description.

4. CUDA

The multiblock approach suits CUDA[1] perfectly by using a grid type formation with blocks that exploits CUDA parallelization and GPU's architecture. The block dimension is given as a parameter and then the grid dimension is computed as in the sequential implementation. The kernel function to computes the LBP is then launched: each block of threads corresponds to a sub-region of the image, in this way each block of threads computes the LBP for a sub-region.

Algorithm 4 LBP CUDA Version

procedure COMPUTELBP($pixels, samples, radium, edge$)
  COMPUTESIZE ($pixels, samples, radium, edge$)
  COMPUTELBP$<<< lbpGridSize, lbpBlockSize >>>$
return Histogram

Each threadId is used to identify the pixel to work with and then is compared with the pixels defined in its neighborhood. The value obtained is then used in the histogram of the current block.
There may be race condition when each thread try to update the histogram of its block. Therefore an `atomicAdd` becomes necessary.

Algorithm 5 LBP kernel

procedure COMPUTELBP($pixels, samples, radium, edge$)

$position \leftarrow threadId + blockId * blockEdge$

$pixel \leftarrow$ GETPIXEL($position$)

$vectorP \leftarrow$LBPVALUE($pixel$)

ATOMICADD($histogram, vectorP$)

return Histogram

5. Results

The performance of the sequential algorithm and the different parallel versions is measured using the speedup.
The speedup equals to the ratio between the time of execution of the sequential version and the parallel one, using the same input.
Moreover, the speedup is computed with respect to different parameters and input size.
The parameters of the algorithm are indeed the number of pixels to use for the LBP descriptor, the radius of the neighborhood and the blockedge that defines the image subregion processed.
The results were produced using a Microsoft Azure Virtual Machine, 2.60 GHz with 6 cores and NVIDIA Tesla M60.
The tables show the performance speedup between the sequential algorithm and CUDA, and between the parallel version and CUDA, changing input size and using P = 9, R = 1 and blockEdge = 16.

Size	Sequential	CUDA	SpeedUp
(750x562)	109 ms	5 ms	21.80
(1000x750)	193 ms	7 ms	24.13
(1500x1125)	441 ms	16 ms	27.56
(2000x1500)	774 ms	27 ms	28.67
(3000x2000)	1720 ms	57 ms	30.18
(4000x3000)	3234 ms	98 ms	33.45

Table 1: Comparison between sequential and CUDA

Size	Parallel	CUDA	SpeedUp
(750x562)	25 ms	5 ms	5.00
(1000x750)	45 ms	8 ms	5.62
(1500x1125)	92 ms	16 ms	5.75
(2000x1500)	160 ms	27 ms	5.93
(3000x2000)	349 ms	57 ms	6,13
(4000x3000)	629ms	98 ms	6,41

Table 2: Comparison between parallel and CUDA.

The speedup grows with respect to the input size.

The image shows the performance using number of samples = 10, radium = 2.0, edges = 32, and using different image sizes = (500x375), (750x562), (1000x750), (1500x1125), (2000x1500), (3000x2000), (4000x3000).

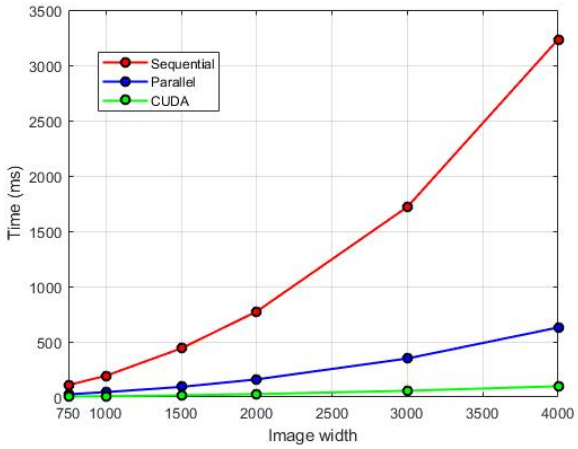

Figure 3: Performance comparison with different image sizes.

While the execution time of the sequential algorithm grows almost esponentialy with the input size, the parallel algorithm and CUDA grow more linearly.
The next tests analyze the speedup obtained with the same input image by changing the parameter ”samples”, indicated as ”P”.

Parameters	Sequential	CUDA	SpeedUp
P=15, R=2, e=32	5069 ms	1004 ms	5.05
P=14, R=2, e=32	3885 ms	514 ms	7.56
P=13, R=2, e=32	3237 ms	264 ms	12.26
P=12, R=2, e=32	2866 ms	139 ms	20.62
P=11, R=2, e=32	2552 ms	76 ms	33.58
P=10, R=2, e=32	2299 ms	45 ms	51.09
P=9, R=2, e=32	2084 ms	27 ms	77.19
P=8, R=2, e=32	1926 ms	18 ms	107.00
P=6, R=2, e=32	1592 ms	10 ms	159.20
P=4, R=2, e=32	1297 ms	7 ms	185.29

Table 3: Comparison between sequential and CUDA, image size (3840x2160)

Now, decreasing the number of samples, the speedup increase. This happens because with less samples, the computing of LBP descriptor is faster.

Figure 4: Example of LBP Descriptor

Parameters	Parallel	CUDA	SpeedUp
P=15, R=2, e=32	2468 ms	1004 ms	2.24
P=14, R=2, e=32	1442 ms	514 ms	2.81
P=13, R=2, e=32	907 ms	264 ms	3.44
P=12, R=2, e=32	646 ms	139 ms	4.65
P=11, R=2, e=32	502 ms	76 ms	6.61
P=10, R=2, e=32	408 ms	45 ms	9.07
P=9, R=2, e=32	349 ms	27 ms	12.93
P=8, R=2, e=32	318 ms	18 ms	17.67
P=6, R=2, e=32	261 ms	10 ms	26.10
P=4, R=2, e=32	213 ms	7 ms	30.43

Table 4: Comparison between parallel and CUDA, image size (3840x2160)

Parameters	Sequential	CUDA	SpeedUp
P=15, R=2, e=16	11249 ms	4025 ms	2.79
P=14, R=2, e=16	7001 ms	1984 ms	3.52
P=13, R=2, e=16	4971 ms	1016 ms	4.89
P=12, R=2, e=16	3701 ms	514 ms	7.20
P=11, R=2, e=16	3010 ms	263 ms	11.44
P=10, R=2, e=16	2565 ms	138 ms	18.58
P=9, R=2, e=16	2243 ms	74 ms	30.31
P=8, R=2, e=16	2007 ms	42 ms	47.78
P=6, R=2, e=16	1631 ms	16 ms	101.93
P=4, R=2, e=16	1301 ms	8 ms	162.62

Table 5: Comparison between sequential and CUDA, image size (3840x2160)

Parameters	Parallel	CUDA	SpeedUp
P=15, R=2, e=16	8508 ms	4025 ms	2.11
P=14, R=2, e=16	4468 ms	1984 ms	2.25
P=13, R=2, e=16	2438 ms	1016 ms	2.39
P=12, R=2, e=16	1428 ms	514 ms	2.77
P=11, R=2, e=16	896 ms	263 ms	3.40
P=10, R=2, e=16	609 ms	138 ms	4.41
P=9, R=2, e=16	447 ms	74 ms	6.04
P=8, R=2, e=16	371 ms	42 ms	8.83
P=6, R=2, e=16	277 ms	16 ms	17.31
P=4, R=2, e=16	224 ms	8 ms	28.00

Table 6: Comparison between sequential and CUDA, image size (3840x2160)

6. Conclusions

The LBP descriptor in his Multi-Block version is suitable for parallelization. Moreover CUDA architecture is easily linkable to the key concept of the algorithm. This allows a faster version of a common texture descriptor.

References

[1] NVIDIA. Cuda documentation. https://docs.nvidia.com/cuda/.

[2] M. Pietikäinen. Local binary patterns, 2010. www.scholarpedia.org/article/Local_Binary_Patterns.

[3] M. P. T. Ojala and D. Harwood. Pattern recognition, 1996.