PC-2019/20 LBP Descriptor

Francesca Del Lungo
francesca.dellungo2@stud.unifi.it

Matteo Petrone
matteo.petrone@stud.unifi.it

Abstract

Local Binary Patterns (LBP) is a non-parametric descriptor whose aim is to efficiently summarize the local structures of images. In this paper we use it to compare computational time between a sequential and two different parallel versions, one of which involves CUDA.

1. Introduction

Local Binary Pattern (LBP) is a type of visual descriptor used for a variety of tasks in computer vision. It has been used for object, scene and face detection, with a good popularity due to computational simplicity and good performances [5]. As a non-parametric method, it summarizes local structures of images efficiently by comparing each pixel with just its neighboring pixels.

LBP was originally proposed for texture analysis [4], and has proved a simple yet powerful approach to describe local structures. Due to its simplicity yet very powerful discriminative capability, many LBP variants have also been developed since its first introduction.

In this paper we implement the classical version of LBP, with the purpose of analyzing the hardware acceleration obtained through parallelization. At first, we expose the basics of LBP operations; then we show the sequential version of the algorithm (3), our idea of Java parallelization (4) and finally the CUDA version of the algorithm, with particular emphasis on the differences between a naive and a tiling version.

1.1. LBP procedure

The classical version of LBP works in a block size of b×b pixels, in which the center pixel is used as a threshold for the neighboring pixels, and the LBP value for the center pixel is generated by encoding the computed threshold value into a decimal value.

Let's consider a generic LBP descriptor which operates on a P neighborhood pixels, the value of the LBP code at the pixel (x_c, y_c) will given by:

$$LBP(x_c, y_c) = \sum_{p=0}^{P-1} S(g_p - g_c) * 2^p \tag{1}$$

where g_c is the gray value of the center pixel (x_c, y_c), g_p are the gray values of the P pixels that belong to the neighborhood of the considered pixel and S defines the thresholding function as follow:

$$S(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & otherwise \end{cases} \tag{2}$$

Using a neighborhood of P pixels we have a total of 2^P possible combinations of LBP codes, that in the classical case of 3x3 neighborhood corresponds to $2^8 = 256$ possible values.

The last step is to compute the LBP histogram over all the LBP codes. Since a 3 x 3 neighborhood has $2^8 = 256$ possible patterns, the LBP value of each pixel will be $\in [0, 255]$ allowing us to construct a 256-bin histogram of LBP codes as final feature vector.

2. Implementation

Three different versions of classical LBP algorithm have been implemented in this work: a first sequential in *java* [1] and two parallel versions: one in *java threads* and a second one in *CUDA* [3].

We have considered the classical approach of the LBP algorithm where the neighborhood of each pixel is given by a matrix of 3x3 dimension centered on the considered pixel.

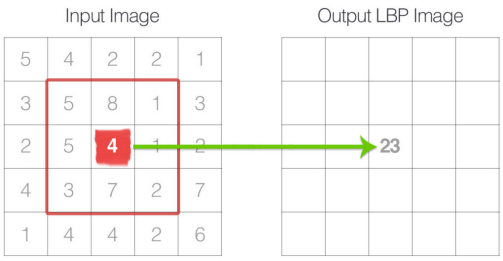

Figure 1. Example of sliding window on an input image and the corresponding value on the output.

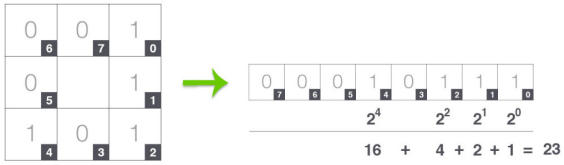

Figure 2. Example of computation of a pixel value through LBP algorithm.

Later to have compared neighborhood values with the center, we started from the top-left value and built the pixel value (Figure 1) proceeding left-to-right and row-by-row. Notice that it is equivalent to proceed in a clockwise or counterclockwise way.

In Appendix 8 there is a full example of LBP computation on an image from grayscale to LBP histogram, taken from a 1280x720 (HD) image from our dataset.

2.1. Image preprocessing

We started from RGB images, but to extract the texture and the relative histogram a couple of operations were necessary. First, the RGB image is transformed to grayscale. Moreover, we needed to pay attention on the borders: in our case, with a 3x3 window sliding on the matrix, we used padding with 0 values on the original image.

2.2. Dataset

We built our dataset by referencing to TVs screen resolutions. Specifically we collected images of the following resolution:

- 480x360
- 640x480 (NTSC)
- 1280x720 (HD)
- 1920x1080 (FullHD)
- 3840x2160 (4K)
- 7680×4320 (8K)
- 15360×8640 (16K)

The last image has been used only for the experiments with CUDA, it has not been possible to conduct experiments for the sequential and parallel version with java threads due to the image big dimension.

3. Sequential version

In Algorithm 1 is shown the pseudocode of the sequential implementation of the LBP descriptor algorithm that computes the new image and the relative histogram.

Algorithm 1 LBP algorithm

procedure LBP(img)
  $neighbors \leftarrow 3$
  $radius \leftarrow \lfloor neighbors/2 \rfloor$
  for row **in** $img.height$ **do**
    for col **in** $img.width$ **do**
      $lbp\_v := array[neighbors * neighbors - 1]$
                                ▷ 3 x 3 neighborhood
      for $r = -radius, ..., radius$ **do**
        for $c = -radius, ..., radius$ **do**
                                ▷ except for the center
          if $img[row+r][col+c] \geq img[row][col]$ **then**
            $lbp\_v[idx++] \leftarrow 1$
          else
            $lbp\_v[idx++] \leftarrow 0$

                                ▷ Binary to decimal value
      $sum \leftarrow 0$
      for c_idx **in** lbp_v **do**
        $sum += 2^{c\_idx} * lbp\_v[c\_idx]$
      $new\_img[row][col] \leftarrow sum$
  COMPUTEHISTOGRAM(new_img)

4. Parallel version with Java Threads

Efficient performance increase for parallel algorithm is ensured by an equal distribution of computational loads to the cores. In fact for the Java Threads version of the algorithm we generated as many splits from the original image as there are available threads. Starting from the original image, the splits are made horizontally in such a way that their dimensions are the same, except for the last one split, in the case that the height of the image is not a multiple of the number of threads.
In Figure 3 there is a representation of this process on an image taken from our dataset (2.2).
Each thread takes a split and operates on it computing the same algorithm described before (see Algorithm 1). Furthermore, it calculates a partial histogram that will be merged later.

Figure 3. Representation of splitting process on an image in an experiment with 4 threads.

Parallel part of the algorithm involves instantiation of

threads by the same time and finalizing them using barrier synchronization. In fact if the sub images are edge padded no synchronization communication among threads during execution is needed. Only barrier synchronization is used to detect whether threads completed their tasks.

Java has built in libraries to support multithreaded programming via *java.lang.Thread* class or*java.lang.Runnable* interface.

5. Parallel version with CUDA

CUDA (Compute Unified Device Architecture) [3] is a parallel computing platform and application programming interface (API) model created by Nvidia.

In this section we will look at some details of the parallel implementation of the proposed algorithm that takes advantage of the parallelism of the GPU. There are two variants of mean shift algorithm implemented on CUDA: the first one is the naive version, while the second one is a more optimized version that makes use of shared memory.

In the GPU, multiple threads form a block and multiple blocks together form a grid. Each thread within a thread block executes an instance of the kernel and has a 3-dimensional thread ID and also every thread block has a 3-dimensional block ID within its grid.

For the LBP histogram computation, depending on the input distribution, some bins will be used much more than others, so it is necessary to support efficient accumulation of the values across the full memory hierarchy. Our histogram implementation has two phases : in the first one each CUDA thread block processes a set of pixels of the image and accumulates a corresponding local histogram in shared memory, in the second phase the local histogram is stored in global memory. It's important to note that every thread of the block has access to the shared memory, where they have to write values, for this reason it's necessary to use *atomic operations*. This is required both for the access to the local histogram than for the final one in global memory. Obviously if on the one hand the atomic operations ensure that the final value of the histogram is correct, these inevitably slow down the execution time of the CUDA kernel.

Note that every bin of the histogram array in shared memory is initialized to zero value (in the pseudocode represented by the INITIALIZEHISTOGRAM($hist\_sm$)) and this is done through a loop indexed as follows:

```
for(i = threadIdx.x; i < n_bins; i +=
blockDim.x)
```

Moreover the `__syncthreads()` after that initialization assures that all the bins are set to zero before that any other thread can continue with the computation.

When the CUDA kernel is launched the execution configuration is defined as:

$$BLOCK\_DIM = (TILE\_WIDTH, TILE\_WIDTH) \tag{3}$$

$$GRID\_DIM = \lceil (width/BLOCK\_DIM, height/BLOCK\_DIM) \rceil \tag{4}$$

Algorithm 2 CUDA naive LBP algorithm

procedure CUDALBPKERNEL($img, hist\_gm$)
  $hist\_sm := sm\_array[n\_bins]$
  $ctr\_x \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
  $ctr\_y \leftarrow blockIdx.y * blockDim.y + threadIdx.y$
  INITIALIZEHISTOGRAM($hist\_sm$)
  $\_\_syncthreads()$
  if $ctr\_x < img.widthctr\_y < img.height$ **then** ▷ Ensure that threads do not attempt illegal memory access
    $pix\_val \leftarrow 0$
    $idx \leftarrow 0$
    for $r = -radius, ..., radius$ **do**
      for $c = -radius, ..., radius$ **do** ▷ except for the center
        $curr\_x \leftarrow row + r$
        $curr\_y \leftarrow col + c$
        if $curr\_x$ **and** $curr\_y$ **are valid then**
          if $img[curr\_x][curr\_y] \geq img[ctr\_x][ctr\_y]$ **then**
            $lbp\_v[idx++] \leftarrow 1$
          else
            $lbp\_v[idx++] \leftarrow 0$
    for c_idx **in** lbp_v **do**
      $sum+ = 2^{c\_idx} * lbp\_v[c\_idx]$
    $new\_img[ctr\_x][ctr\_y] \leftarrow sum$
    ▷ Add pixel value to local histogram
    ATOMICADD($hist\_sm, pix\_val$)
    $\_\_syncthreads()$
    ▷ Local to global histogram
    for bin **in** $hist\_sm$ **do**
      ATOMICADD($hist\_gm, bin$)

5.1. CUDA-Tiling version

There are several levels of memory on the GPU device, each one with distinct characteristics.

Every thread in a thread block has access to a unified *shared memory* (in addition to the private 'local' memory of the single thread) that is shared among all threads for the life of that thread be.

Therefore there is a natural tradeoff in the use of device memories in CUDA: the global memory is large but slow, whereas the shared memory is small but fast. A common

strategy is to partition the data into subsets called tiles so that each tile fits into the shared memory [2]. An important principle is that the kernel computation on these tiles can be done independently of each other.

The main idea is to use the shared memory available to each block so as to contain a tile and by doing this, the reading/writing costs of the global memory are significantly amortized. There are two main phases of which this CUDA-tiling algorithm is composed and of which it is worthwhile to deepen: the first consists in the loading on the data from the global memory to the shared memory, and the second consists in the computation part (which operates on the loaded data).
The main idea is to divide the image into tiles and let the Block made by TILE_WIDTHxTILE_WIDTH threads. To operate on the tile, each Block needs to borrow some pixels from neighbor tile so that even pixels on the border tile can be significant. For this reason the occupied shared memory area will be (TILE_WIDTH+MASK_WIDTH-1)*(TILE_WIDTH+MASK_WIDTH-1). Considering that each thread should move at most two elements from global to shared memory, this load is convenient to be managed in two-stages: the first loading the tile, and the second loading the remaining pixels. When the shared memories are correctly loaded, and after making sure that every thread finished doing that (__syncthreads() necessary), the second step consists in computing LBP pixel in the same previously explained manner (Algorithm 2), with the main difference that now the values of the pixels to compare are not read from global memory but from shared memory.

6. Experiments

6.1. Speedup

Many experiments have been done to evaluate the performance of the different implementations of the LBP algorithm.

The speedup metric has been used to compare these differences in the performances, where the **speedup** is defined as the ratio of the time for solving a problem sequentially t_S to the time taken by the parallel algorithm to solve the same problem t_P.

$$S = \frac{t_S}{t_P}$$

All the tests have been performed on a machine equipped with:

- CPU: Intel(R) Core(TM) i7-860 @ 2.80GHz, with 4 cores/ 8 threads (L1 cache
- GPU: NVidia GeForce GTX 980, 4 GB (with CUDA 10.1)

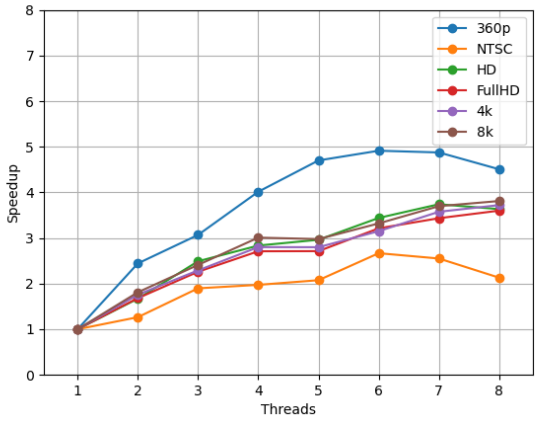

Figure 4. Speed up sequential over java threads time results varying the number of threads.

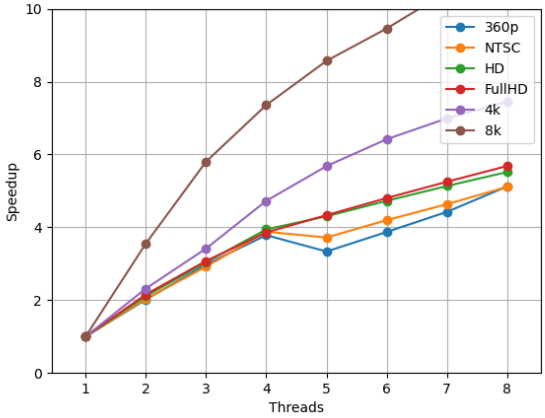

Figure 5. Speed up sequential over java threads time results varying the number of threads. Experiments on Apple M1.

All the reported results have been obtained running each test 15 times (both for the sequential and the parallel versions) and averaging the values obtained.

Thanks to the availability of a new machine with a different processor architecture, we executed both the sequential and the parallel (Java Thread) versions of LBP even on an Apple M1 chip:

- CPU: Apple M1 Octa-Core (4@3.20GHz, 4@2.00GHz), Cache L2 (4@12MB, 4@4MB).

6.2. SpeedUp: sequential and parallel with java threads

In the plot in Figure 4 we can compare how speedups change varying the number of threads. The trend of almost all curves is sub-linear from two to four threads and then decrease as the number of threads increase. For the 360p we

Image size	360p		
Algorithm version	**Threads**	**Time (ms)**	**Speedup**
Sequential	-	30.46	-
Java Threads	2	12.60	2.442
	3	9.95	3.067
	4	7.53	4.039
	5	6.48	4.713
	6	6.20	4.91
	7	6.25	4.87
	8	6.73	4.51

Table 1. Speedup. Experiments done on a 480x360 image on an Intel-17. Mean values over 15 runs of the algorithm.

Image size	480p (NTSC)		
Algorithm version	**Threads**	**Time (ms)**	**Speedup**
Sequential	-	19.73	-
Java Threads	2	15.60	1.26
	3	10.40	1.89
	4	10.0	1.97
	5	9.58	2.07
	6	7.40	2.66
	7	7.73	2.55
	8	9.00	2.13

Table 2. Speedup. Experiments done on a 640x480 image on an Intel-17. Mean values over 15 runs of the algorithm.

Image size	720p (HD)		
Algorithm version	**Threads**	**Time (ms)**	**Speedup**
Sequential	-	66.13	-
Java Threads	2	39.60	1.67
	3	26.46	2.49
	4	23.33	2.83
	5	22.33	2.96
	6	19.20	3.44
	7	17.66	3.74
	8	16.2	3.83

Table 3. Speedup. Experiments done on a 1280x720 image on an Intel-17. Mean values over 15 runs of the algorithm.

can notice that in the first part the speedup is linear. Specifically, for the 360p and 480p images we can see the highest peak with 6 threads, after which the speedup decreases.

We can observe in Figure 5 different curves wrt Figure 4. This is because Apple M1 chip has 8 physic cores and 8 threads, unlike the other machine that has 4 cores with 8 threads (like many modern machines it uses *hyper-threading*). This hardware differences bring better performances increasing the number of threads participating to the computation, with super-linear trends (justified by bigger caches).

Image size	1080p (FullHD)		
Algorithm version	**Threads**	**Time (ms)**	**Speedup**
Sequential	-	129.13	-
Java Threads	2	76.733	1.68
	3	56.94	2.27
	4	47.66	2.71
	5	47.60	2.71
	6	40.20	3.21
	7	37.60	3.43
	8	35.86	3.6

Table 4. Speedup. Experiments done on a 1920x1080 image on an Intel-17. Mean values over 15 runs of the algorithm.

Image size	2160p (4K)		
Algorithm version	**Threads**	**Time (ms)**	**Speedup**
Sequential	-	478.6	-
Java Threads	2	272.34	1.76
	3	208.93	2.3
	4	170.73334	2.80
	5	170.0	2.81
	6	151.86	3.15
	7	133.86	3.57
	8	128.60	3.72

Table 5. Speedup. Experiments done on a 3840x2160 image on an Intel-17. Mean values over 15 runs of the algorithm.

Image size	4320p (8K)		
Algorithm version	**Threads**	**Time (ms)**	**Speedup**
Sequential	-	2331.8	-
Java Threads	2	1282.2	1.82
	3	965.6	2.41
	4	775.2	3.01
	5	783.33	2.97
	6	702.8	3.32
	7	630.5333	3.70
	8	612.0	3.81

Table 6. Speedup. Experiments done on a 7680×4320 image on an Intel-17. Mean values over 15 runs of the algorithm.

6.3. Cuda tiling: tile width

In Figure 6 are shown GPU times for the CUDA with tiling version varying the tile width. Here we can observe the advantage of shared memory pre-loading data that we explained in detail on section 5.1: increasing the Tile dimension (up to 32x32, which represents the max number of threads a block can contain), the GPU time computation decreases from just under 8ms to little more than 0.2ms.

For all the subsequent experiments the tile width will be set to 32 as it is the value that assures the minimum time for the computation.

In Table 7 are shown the times for the sequential and CUDA tiling implementations are the speedups obtained.

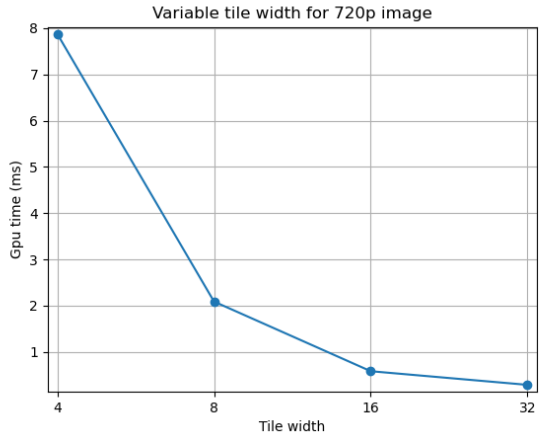

Figure 6. GPU times varying the tile width for HD image. Times in milliseconds. Mean values over 15 runs of the cuda algorithm.

Image size	Sequential	CUDA tiling	Speedup
360p	30.46 ms	0.0715 ms	430.9
NTSC	19.73ms	0.106 ms	197.3
HD	66.13 ms	0.289 ms	236.2
FullHD	129.13 ms	0.628 ms	208.3
4K	478.6 ms	2.453 ms	199.4
8K	2331.8 ms	9.775 ms	240.5

Table 7. Sequential version in comparison with cuda tiling version with tile width set to 32. The results reported are the average over 15 runs.

We can notice that, as expected, with the GPU parallelism the speedup is higher respect to the speedup obtained with the CPU parallelism.

Finally, in Table 8 are shown the time values for the different CUDA versions.

Image	CUDA naive	CUDA tiling	Speedup
360p	98.599 μs	71.466 μs	1.38
NTSC	126.266 μs	106.533 μs	1.19
HD	377.33 μs	289.46μs	1.30
FullHD	760.33 μs	628.79 μs	1.21
4K	2990.067 μs	2453.13 μs	1.22
8K	11739.467 μs	9775.400μs	1.20
16K	41791.469 μs	39585.867 μs	1.08

Table 8. CUDA naive version vs tiling version with tile width set to 32. The results reported are the average over 15 runs. Times are expressed in microseconds (μs).

7. Resources

The code of is publicly available on GitHub:

- Sequential and parallel CPU (Java): `https://github.com/matpetrone/LBP_Descriptor.git`
- Parallel GPU (CUDA): `https://github.com/matpetrone/LBP_Descriptor_CUDA.git`

References

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[2] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[3] NVIDIA, P. Vingelmann, and F. H. Fitzek. Cuda, release: 10.2.89, 2020.

[4] T. Ojala, M. Pietikäinen, and D. Harwood. A comparative study of texture measures with classification based on featured distribution. *Pattern Recognition*, vol. 29:51–59, 1996.

[5] T. Ojala, M. Pietikäinen, and T. Maenpaa. Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(1):32–40, 2002.

8. Appendix

Image size	360p		
Algorithm version	**Threads**	**Time (ms)**	**speedup**
Sequential	-	42.82	-
Java Threads	2	20.36	2.10
	3	14.4	2.97
	4	11.36	3.76
	5	12.84	3.33
	6	11.06	3.87
	7	9.68	4.42
	8	8.39	5.12

Table 9. Speedup. Experiments done on a 480x360 image on an 8-core Apple M1 chip. Mean values over 15 runs of the algorithm.

Image size	480p (NTSC)		
Algorithm version	**Threads**	**Time (ms)**	**speedup**
Sequential	-	68.58	-
Java Threads	2	33.68	2.03
	3	23.5	2.91
	4	17.68	3.87
	5	18.44	3.71
	6	16.34	4.19
	7	14.8	4.63
	8	13.42	5.11

Table 10. Speedup. Experiments done on a 640x480 (NTSC) image on an 8-core Apple M1 chip. Mean values over 15 runs of the algorithm.

Image size	720p (HD)		
Algorithm version	**Threads**	**Time (ms)**	**speedup**
Sequential	-	223.0	-
Java Threads	2	105.74	2.10
	3	73.32	3.04
	4	56.56	3.94
	5	51.84	4.30
	6	47.18	4.72
	7	43.44	5.13
	8	40.42	5.51

Table 11. Speedup. Experiments done on a 1280x720 (HD) image on an 8-core Apple M1 chip. Mean values over 15 runs of the algorithm.

Image size	1080p (FullHD)		
Algorithm version	**Threads**	**Time (ms)**	**speedup**
Sequential	-	530.44	-
Java Threads	2	246.02	2.15
	3	172.46	3.07
	4	137.44	3.85
	5	122.4	4.33
	6	110.44	4.80
	7	101.04	5.24
	8	93.32	5.68

Table 12. Speedup. Experiments done on a 1920x1080 (FullHD) image on an 8-core Apple M1 chip. Mean values over 15 runs of the algorithm.

Image size	2160p (4K)		
Algorithm version	**Threads**	**Time (ms)**	**speedup**
Sequential	-	2767.4	-
Java Threads	2	1193.96	2.31
	3	810.0	3.41
	4	585.56	4.72
	5	487.26	5.67
	6	431.16	6.41
	7	395.86	6.99
	8	371.32	7.45

Table 13. Speedup. Experiments done on a 3840x2160 (4K) image on an 8-core Apple M1 chip. Mean values over 15 runs of the algorithm.

Image size	4320p (8K)		
Algorithm version	**Threads**	**Time (ms)**	**Speedup**
Sequential	-	19531.56	-
Java Threads	2	5501.36	3.55
	3	3367.46	5.80
	4	2657.48	7.35
	5	2281.84	8.56
	6	2066.5	9.45
	7	1867.26	10.46
	8	1701.35	11.48

Table 14. Speedup. Experiments done on a 7680x4320 (8K) image on an 8-core Apple M1 chip. Mean values over 15 runs of the algorithm.

Figure 7. Example of input grayscale image for LBP algorithm. Image of size 1280x720 (HD).

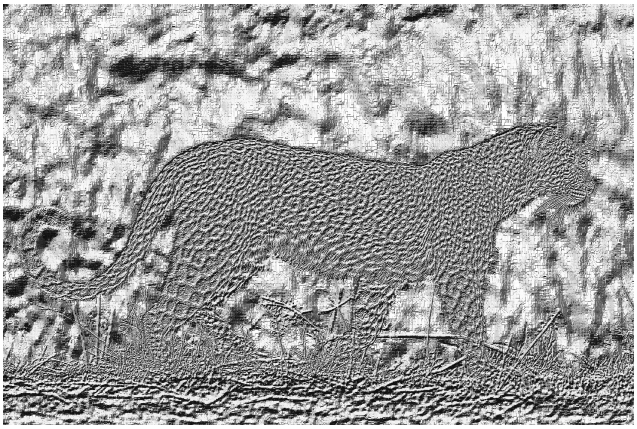

Figure 8. LBP output image

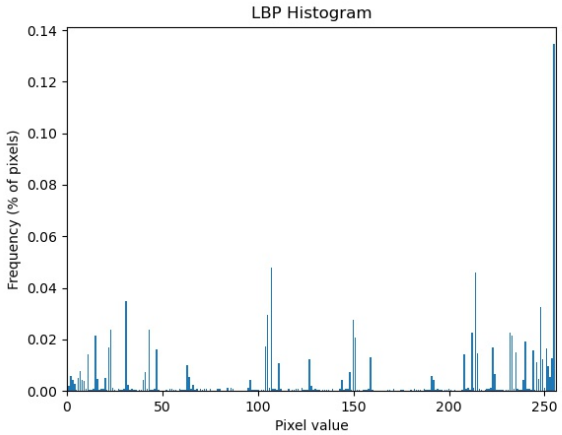

Figure 9. LBP histogram