Sequential vs CUDA LBP Histogram Computation

Davide Tarlini
E-mail address
davide.tarlini@edu.unifi.com

July 18, 2025

Abstract

1 Introduction and Problem Statement

The *Local Binary Pattern* (LBP) operator is a visual descriptor for gray-scale images used to capture local texture features in images. LBP encodes the local spatial structure of an image by comparing each pixel with its surrounding neighbors and computing an associated binary pattern.

The core idea behind LBP is this: given pixel p, a binary code is generated by thresholding the intensity values of its neighboring pixels against p. This binary pattern is then converted to a decimal number, which serves as a descriptor for that pixel. Given a collection of such descriptors (one for every pixel of an image). The histogram of these patterns can be used to summarize the texture features of the image.

Although computing a single LBP is computationally lightweight, the histogram of these patterns across an image usually requires thousands if not millions of repetitions of such task. Massive gains can be obtained by parallelizing the task via GPU-based acceleration. It is an instance of an embarrassingly parallelizable problem.

This report presents a comparative study between a sequential and CUDA-parallelized implementation of LBP histogram computation. The goal is to analyze the performance gains achievable through GPU acceleration across different image sizes and thread block configurations.

2 LBP and Histogram Computation

To be more precise, the LBP code associated to a pixel p is obtained by visiting in a clockwise (or counterclockwise) order its neighboring pixels and then computing the following summation:

$$lbp_p = \sum_i 1[p >= p_i] * 2^i \tag{1}$$

for $i = 0, ..., 7$ and p_i the i-th neighbour of p. The summation can be viewd as the decimal representation of a binary string in wich each bit is 0 or 1 according to the result of the indicator function in the summation.

2.1 Padding

Given an image, computing the LBP for border pixels requires the handling of invalid neighbours. The chosen strategy has been to pad the given image with an extra by adding two rows (one at the top and one at the bottom) and two columns (on on the left and one on the right). By default the padded cells are set to 0.

3 Implementations

The general idea of how to perform this task can structured in a very simple format:

- Initialize a padded version of the source image
- Compute the LBPs of each pixel
- Compute the LBPs histogram

The sequential and parallel implementations of the algorithm differ essentially in how they compute these steps. Let us look first at the initialization.

3.1 Initializations

To get a padded version of the source image thew most straightforward way is to initialize a zero-valued image of the appropriate (padded) size and then fill the center with the source image pixel values. Here the pseudocode of the sequential implementation:

Algorithm 1 seq init

Require: Grayscale image I of size $rows \times cols$

Initialize padded image $P \leftarrow$ zero matrix of size $(rows + 2) \times (cols + 2)$
for $i \leftarrow 1$ to $rows$ **do**
  for $j \leftarrow 1$ to $cols$ **do**
    $P[i][j] \leftarrow I[i-1][j-1]$
  end for
end for

Parallelizing this step with cuda is rather straightforward. we first allocate space on the device global memory for the source image and the padded image. We then copy the source image into the allocated space and then make use of the *memset* function to initialize to 0 all the space allocated for the padded image. We then parallelize the filling of this padded image by calling a cuda kernel with the following internal logic:

Algorithm 2 `cuda init`

Require: Input image I, padded image P

$r \leftarrow$ `blkIdx.y` $\cdot$ `blkDim.y` + `thIdx.y`
$c \leftarrow$ `blkIdx.x` $\cdot$ `blkDim.x` + `thIdx.x`
if $r < rows$ and $c < cols$ **then**
  $P[r+1][c+1] \leftarrow I[r][c]$
end if

So every position gets filled by a different thread.

3.2 LBP and histogram computation

This is the more computationally intensive part of the taks and where we can expect to gain more from GPU parallelization. Approaching the computation sequentially is pretty straightforward. We can organize the code with a loop traversing the padded image:

Algorithm 3 seq hist accumulation

Require: Padded image P

for $i \leftarrow 0$ to $rows - 1$ **do**
  for $j \leftarrow 0$ to $cols - 1$ **do**
    $code \leftarrow$ `lbp_seq`$(P, i, j, cols)$
    $H[code] \leftarrow H[code] + 1$
  end for
end for

And the LBP computation is performed by simply accumulating values across the neighbors:

Algorithm 4 seq lbp

Require: Padded image P, coordinates (i, j) in original image

$r \leftarrow i + 1, c \leftarrow j + 1$
$c_0 \leftarrow P[r][c]$
$code \leftarrow 0$
$code \leftarrow code + (P[r-1][c-1] \geq c_0) \cdot 128$
$code \leftarrow code + (P[r-1][c] \geq c_0) \cdot 64$
$code \leftarrow code + (P[r-1][c+1] \geq c_0) \cdot 32$
$code \leftarrow code + (P[r][c+1] \geq c_0) \cdot 16$
$code \leftarrow code + (P[r+1][c+1] \geq c_0) \cdot 8$
$code \leftarrow code + (P[r+1][c] \geq c_0) \cdot 4$
$code \leftarrow code + (P[r+1][c-1] \geq c_0) \cdot 2$
$code \leftarrow code + (P[r][c-1] \geq c_0) \cdot 1$
return $code$

3.2.1 Parallel lbps and histogram computation

Parallelizing this step can be done firstly in a very simple way by applying the same exact logic in Algorithm 4, but invoking it in parallel for many different pixels. With cuda it is very simple to write such code, since we almost need to just copy algorithm 4 into a cuda kernel. What needs to be taken care of is populating the histogram, since this steps must be performed by allowing each cuda thread to atomically acces the histogram in global memory. An obvious criticality in this implementation is the direct access to the global histogram by each thread, and we can easily expect high memory contention. We could add a private

histogram to every thread block, stored in shared memory, a very simple modification of the kernel.

Algorithm 5 CUDA lbp without Shared Histogram

Require: Padded image P and histogram H in device global memory

$r \leftarrow$ `blkIdx.y` $\cdot$ `blkDim.y` $+$ `thIdx.y`
$c \leftarrow$ `blkIdx.x` $\cdot$ `blkDim.x` $+$ `thIdx.x`
if $r < rows$ and $c < cols$ **then**
    $r \leftarrow i + 1, c \leftarrow j + 1$
    $c_0 \leftarrow P[r][c]$
    $code \leftarrow code + (P[r-1][c-1] \geq c_0) \cdot 128$
    $code \leftarrow code + (P[r-1][c] \geq c_0) \cdot 64$
    $code \leftarrow code + (P[r-1][c+1] \geq c_0) \cdot 32$
    $code \leftarrow code + (P[r][c+1] \geq c_0) \cdot 16$
    $code \leftarrow code + (P[r+1][c+1] \geq c_0) \cdot 8$
    $code \leftarrow code + (P[r+1][c] \geq c_0) \cdot 4$
    $code \leftarrow code + (P[r+1][c-1] \geq c_0) \cdot 2$
    $code \leftarrow code + (P[r][c-1] \geq c_0) \cdot 1$
    `atomicAdd(&H[code], 1)`
end if

Algorithm 6 CUDA lbp with Shared Histogram

Require: Padded image P and histogram H in device global memory

`shared_hist`
$tid \leftarrow$ `thIdx.y` $\cdot$ `blkDim.x` $+$ `thIdx.x`
$T \leftarrow$ `blkDim.x` $\cdot$ `blkDim.y`
for $i \leftarrow tid$ to 255 step T **do**
    `shared_hist[i]` $\leftarrow 0$
end for
`__syncthreads()`
$r \leftarrow$ `blkIdx.y` $\cdot$ `blkDim.y` $+$ `thIdx.y`
$c \leftarrow$ `blkIdx.x` $\cdot$ `blkDim.x` $+$ `thIdx.x`
if $r < rows$ and $c < cols$ **then**
    $r \leftarrow i + 1, c \leftarrow j + 1$
    $c_0 \leftarrow P[r][c]$
    $code \leftarrow code + (P[r-1][c-1] \geq c_0) \cdot 128$
    $code \leftarrow code + (P[r-1][c] \geq c_0) \cdot 64$
    $code \leftarrow code + (P[r-1][c+1] \geq c_0) \cdot 32$
    $code \leftarrow code + (P[r][c+1] \geq c_0) \cdot 16$
    $code \leftarrow code + (P[r+1][c+1] \geq c_0) \cdot 8$
    $code \leftarrow code + (P[r+1][c] \geq c_0) \cdot 4$
    $code \leftarrow code + (P[r+1][c-1] \geq c_0) \cdot 2$
    $code \leftarrow code + (P[r][c-1] \geq c_0) \cdot 1$
    `atomicAdd(&shared_hist[code], 1)`
end if
`__syncthreads()`
for $i \leftarrow tid$ to 255 step T **do**
    `atomicAdd(&H[i], shared_hist[i])`
end for

4 Experiments and Results

The performances of the three implementations have been compared across a range of image sizes and thread block sizes. For each configuration of the experiment a total of 100 runs have been executed and the performances averaged. The test images are created by populating each pixel with a random value in $[0, 255]$. The experiments have been performed on a machine with a Nvidia GPU Geforce 950m, processor intel i7-7700HQ.

4.1 Padd Performance Comparison

From figure 1 we can see that the parallelization of the padding step is beneficial only over a certain image size,

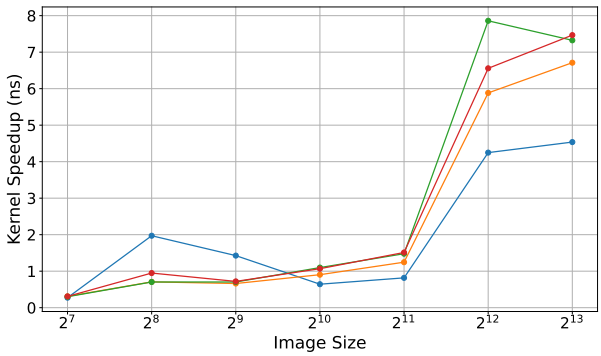

Figure 1: Speedups of the padd kernel. Each curve correspond to a thread block size.

with performances that for smaller images are worse than the sequential version of the padding operation. Overall, there is still somewhat small even for grater image sizes.

4.2 LBP and Histogram Performance Comparison

To compare the lbp and histogram computation performances we can look at the heatmaps in figure 3. This step of the algorithm is where the big performance improvements happens, with speedups that reach over 130 in the best case. The usage of a shared histogram at thread block level has clear advantages for since it always shows much higher speedups expect for very small and not optimal block sizes.

There are also two other patterns to note. First, for the problem at hand the bigger the block size the higher the speedup, independently of image size and usage of the shared memory. Second, for every block size the speedups form a sort of U-shaped curve with respect to the image sizes. Again, this is not affected by the usage of shared memory. Images between 512×512 and 2048×2048 seems harder configurations for the implemented kernels. (Figure 2).

4.3 Global Comparison

We can also look at the performance speedup of the whole procedure, including the data transfer to and from

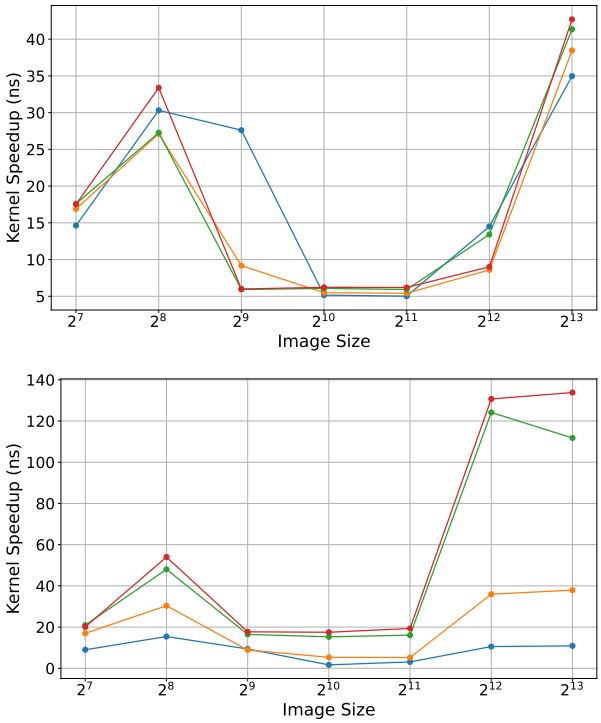

Figure 2: Speedups of non shared version (above) and shared version (below) of the lbp kernel. Each curve correspond to a thread block size. The green and red curves correspond respectively to tile sizes 16×16 and 32×32

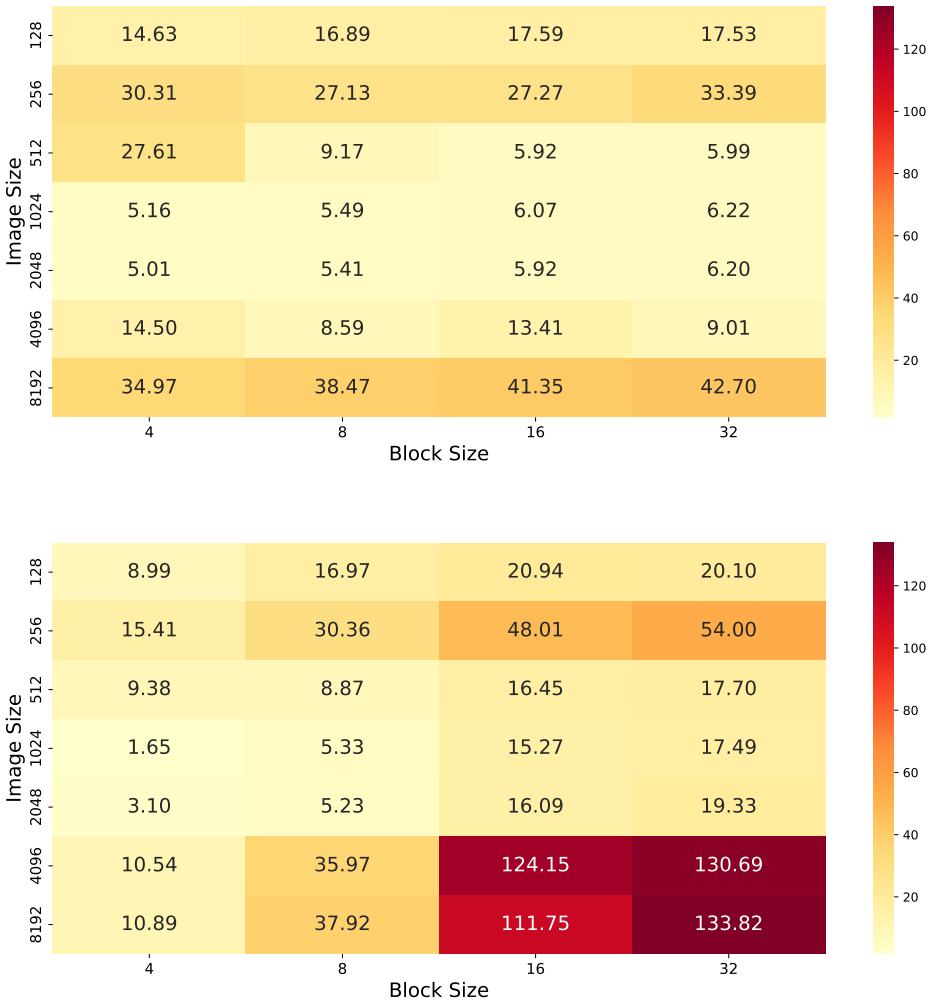

Figure 3: Heatmaps of mean speedups for non sharing lbp kernel (above) ans shared lbp kernel (below). The image sizes and block sizes labeled refers to the size of a side of an image or of a thread block.

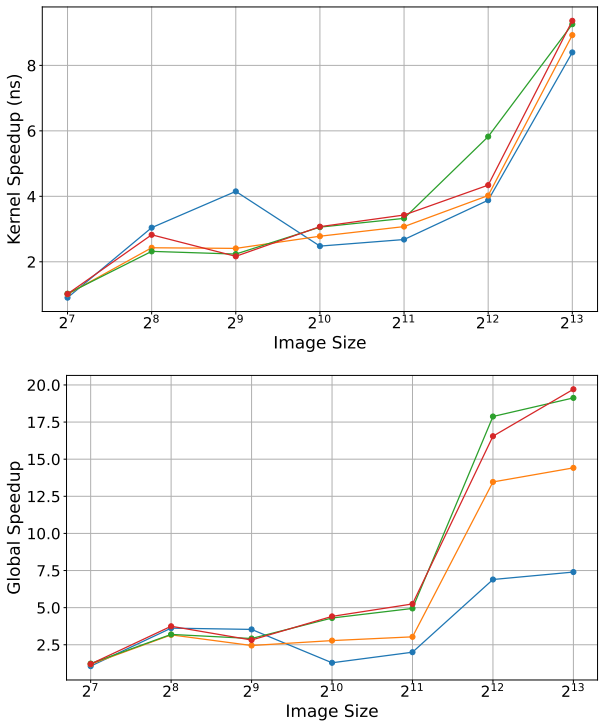

Figure 4: Global speedups for non shared (above) and shared (below) versions. The tile sizes are: Red=32×32; Green=16×16; Yellow=8×8; Blue=4

the GPU memory. These operations do indeed take up much of the time for smaller image size, with global speedups that surpass the value of 5 only for images over 4096×4096. Again we can see the benefit of moving part of the lbp and histogram computation in shared memory.

4.4 Conclusions

In conclusion we can say that the parallelizaton of the task of computing an histogram of LBPs is beneficial with much speed to gain. The choice of tile size and the usage of shared memory to reduce global memory contention on the gpu during the update of the histogram makes huge differences.