Local Binary Pattern descriptor

comparison between sequential and two parallel implementations

Parallel Computing (9 CFU)

Francesca Del Lungo and Matteo Petrone

Prof. Marco Bertini

January 2021

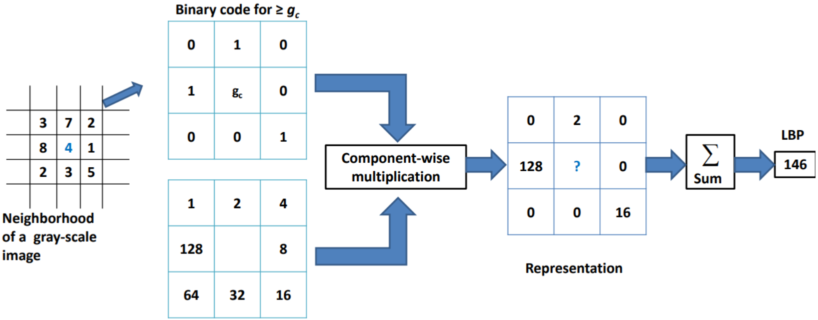
Contents overview

Introduction

LBP

- Non-parametric visual descriptor
- Analyse **local** structure of images to extract texture
- Works in a 3×3 pixels window
- LBP histogram
- **Embarrassingly parallel** algorithm structure

LBP procedure



Implementation

Sequential

- Classical LBP descriptor
- Implemented in java

Sequential

```
procedure LBP(img)
    neighbors ← 3
    radius ← ⌊neighbors/2⌋
    for row in img.height do
        for col in img.width do
            lbp_v := array[neighbors ∗ neighbors − 1]
            for r, c = −radius, ..., radius do
                if img[row + r][col + c] ≥ img[row][col] then
                    lbp_v[idx + +] ← 1
                else
                    lbp_v[idx + +] ← 0
            sum ← 0
            for c_idx in lbp_v do
                sum+ = 2^{c_idx} ∗ lbp_v[c_idx]
            new_img[row][col] ← sum
    ComputeHistogram(new_img)
```

Parallel: Java Threads

- CPU parallelism
- Each thread takes a split of the image and operates on it
- Each thread compute a **partial histogram** and the results are merged together
- **No synchronization** among threads needed during execution
- Multithreaded programming via *java.lang.Thread*

Parallel: CUDA

- GPU parallelism
- Naive and tiling versions
- Each pixel is assigned to a thread

LBP histogram

- For each block: local histogram on **shared memory**
- **Initialization** needed
- **Atomic operations** inside block and at the end to merge local histogram with the global one
- Atomic operations **slow down** the execution time of the CUDA kernel.

Parallel: CUDA naive procedure

```
procedure CudaLBPkernel(img, hist_gm)
    hist_sm := sm_array[n_bins]
    ctr_x ← blockIdx.x ∗ blockDim.x + threadIdx.x
    ctr_y ← blockIdx.y ∗ blockDim.y + threadIdx.y
    InitializeHistogram(hist_sm)
    __syncthreads()
    if  ctr_x < img.width and ctr_y < img.height then
        lbp_v ← ComputeLBPPixelValue(ctr_x, ctr_y)
        new_img[ctr_x][ctr_y] ← lbp_v
        AtomicAdd(hist_sm, pix_val)
        __syncthreads()
        for bin in hist_sm do
            AtomicAdd(hist_gm, bin)
```

Parallel: CUDA tiling

- Objective: decrease global memory accesses
- **Shared memory**: small, but fast
- Phases:
  - **Data loading** from global to shared memory
  - **Data processing**
  - Store data **back to global memory**
- Partition the data into subsets called **tiles**

CUDA tiling

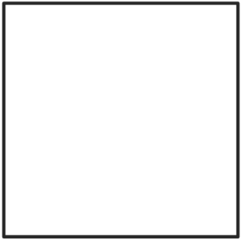
The size of each thread block matches the size of an output tile

$$BLOCK\_DIM = (TILE\_WIDTH, TILE\_WIDTH)$$

So:

- All threads participate in calculating output elements
- Some threads need to load more than one input element into the shared memory

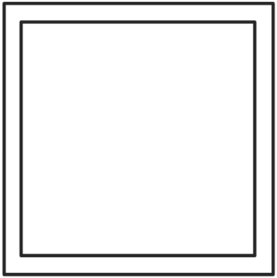
CUDA tiling: data loading I



- Portion of image to be processed
- Dimension:

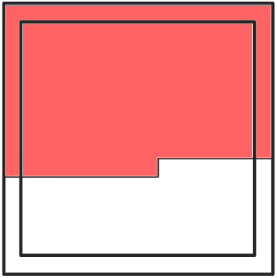
$$TILE\_WIDTH * TILE\_WIDTH$$

CUDA tiling: data loading II



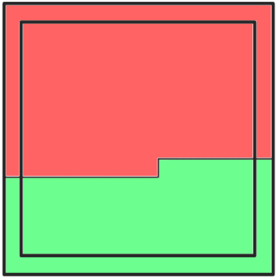
- Pixels needed for the computation
- Shared memory allocation: (TILE_WIDTH + neighbors-1) * (TILE_WIDTH + neighbors-1)

CUDA tiling: data loading III



- Two step loading phase
- First step: Load TILE_WIDTH * TILE_WIDTH elements

CUDA tiling: data loading IV



- Two step loading phase
- Second step: Load the data outside the TILE_WIDTH * TILE_WIDTH

CUDA tiling: data processing

- Same as CUDA naive solution
- Only shared memory access in LBP comparisons

Experiments

Dataset

Images of different resolution:

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

Experiments setting

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

- CPU: Intel Core i7-860 @ 2.80GHz, with 4 cores/ 8 threads
- GPU: NVidia GeForce GTX 980, 4 GB (with CUDA 10.1)
- Average over 15 runs

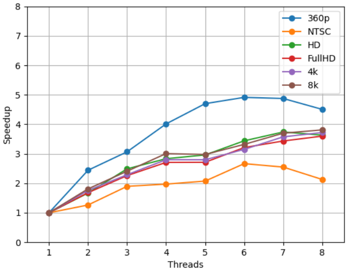
SpeedUp

To compare the performance of a sequential respect to a parallel implementation: **speedup** metric.

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

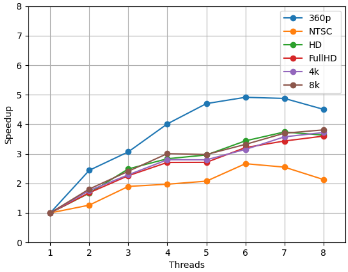
Ideally it should be equal to the number of processors of the parallel version (linear speedup).

Speedup: Sequential over Java Threads

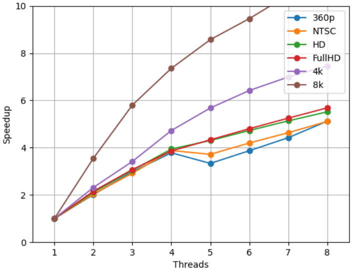


CPU: Intel(R) Core(TM) i7-860 @2.80GHz, with 4 cores/ 8 threads, Cache L2 1MB

Speedup: Sequential over Java Threads

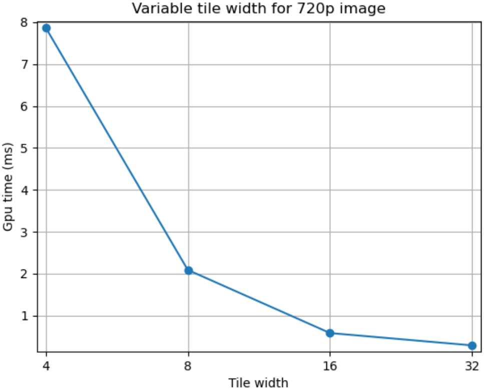


CPU: Intel(R) Core(TM) i7-860 @2.80GHz, with 4 cores/ 8 threads, Cache L2 1MB



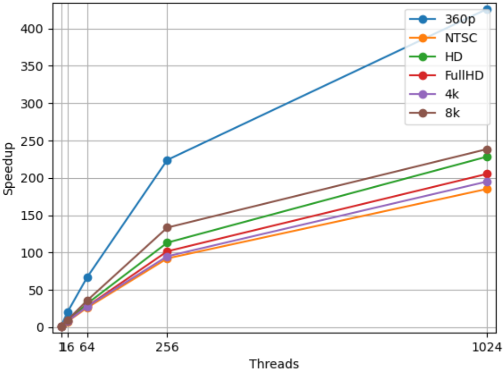
CPU: Apple M1 Octa-Core (4@3.20GHz, 4@2.00GHz), Cache L2 (4@12MB, 4@4MB).

Varying tile width



- $BLOCK\_DIM = TILE\_WIDTH^2$
- Best tile value 32

Speedup: Sequential over CUDA



CUDA naive and tiling

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

Conclusions

Conclusion

- **Embarrassingly parallel** structure makes it suitable for parallel implementation
- **Good results** both for CPU and GPU parallelism
- CUDA makes LBP **applicable to very large images** that cannot be processed sequentially

Thanks for the attention