Parallel approach for GLCM matrix

Universidade Federal de Santa Catarina.
Aluno: Matheus Francisco Batista Machado
INE410129-41000025DO/ME - Parallel Computing

Introduction and Context

- The **Gray Level Co-occurrence Matrix (GLCM)** is a statistical method used in image processing for texture analysis. The technique captures the spatial relationship between pixel intensity values in an image, enabling the extraction of texture features. [1]
- GLCM fits in scenarios where analyzing these textures :
    - Medical Imaging;
        - Identify and classify textures in medical scans, such as X-rays, MRIs, CT scans
        - Segmenting tumor regions based on their texture characteristics.
    - Remote Sensing in satellite imagery and aerial photography
        - Classify land cover types
        - Detect textures that indicate geological features or vegetation health
    - Industrial Inspection quality control processes to
        - Detect surface defects in materials or products.
        - Analyze patterns in fabrics, metals, or composite materials for uniformity
    - Pattern Recognition and Computer Vision
        - Face recognition and biometrics systems
        - Identifying textures in natural images

Motivation

The GLCM is use to plenty of problem in computer vision, medical imaging etc. Usually texture features knows as Harlick features, derived from the Gray Level Co-occurrence Matrix (GLCM).

- Few lib that implements GLCM such as **skimage.feature.graycomatrix** in the **scikit-image**,also there are few non popular implementation in python, such as py-glcm, glcm with numpy, another glcm with numpy.

- Some C++ implementation: GLCM-OpenCV

- Some R implementation: GLCM Textures

Motivation to Implement CUDA GLCM

- The GLCM can be slow to generate for a large set of images.
- There is (glcm cuda python)
  - https://github.com/Eve-ning/glcm-cupy
  - https://github.com/Benjamintdk/GLCM-CUDA

Motivation

- Despite the availability of existing libraries, there are several motivations for creating a new implementation of GLCM:
  - Performance Optimization
  - Parallel Processing
  - Cross Platform Efficiency (GPU, CPU, OpenMP for CPUs)

Description of the problem

- You have **355 DICOM images** and **355 PNG images** that need to be processed efficiently. The goal is to **compare different approaches** for speeding up this processing task using parallel computing techniques. Specifically, you'll:
- 1. Implement parallelism using: Cuda for GPU, OpenMP (OMP) for multi-threading on CPUs, and combination of OpenMP and CUDA for hybrid parallelism
- 2. Then I will compare the implementation with some python and some github implementation.

Algorithm: Gray Level Co-occurrence Matrix (GLCM)

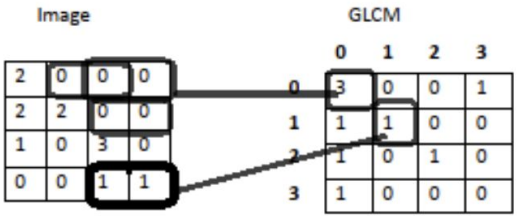
Counting Pairs: For each pair of pixels, the first pixel has an intensity I, and its neighbor (determined by a displacement vector d) has an intensity J For every such pair in the image, a count is incremented in the [I,J]-th cell of the matrix. The displacement vector defines the spatial relationship (e.g., neighbor one pixel to the right, one above, or diagonally).

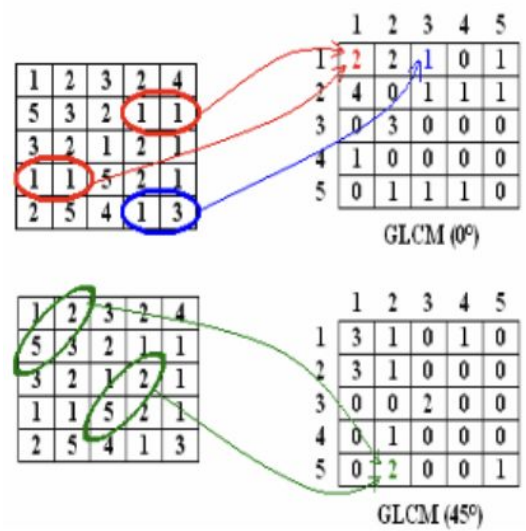
Populating the GLCM: Each [I,J]-th entry in the GLCM holds the count of occurrences where a pixel of intensity III is paired with a neighboring pixel of intensity J, based on the defined displacement.

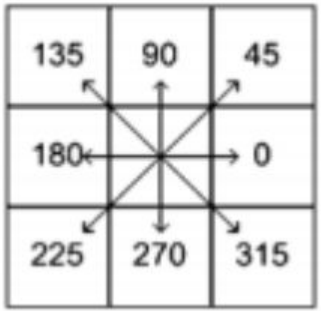
Symmetry and Counts: The matrix is not necessarily symmetric because the relationship is directional. For instance, counting a pixel III with a rightward neighbor J may not yield the same count as a pixel J with a rightward neighbor I. However, if symmetry is desired (e.g., for certain texture analysis methods), the GLCM can be made symmetric by adding it to its transpose.

Algorithm: Gray Level Co-occurrence Matrix (GLCM)

- For the all 8 degree







How I handle the images (PNG, DICOM)

- For PNG images

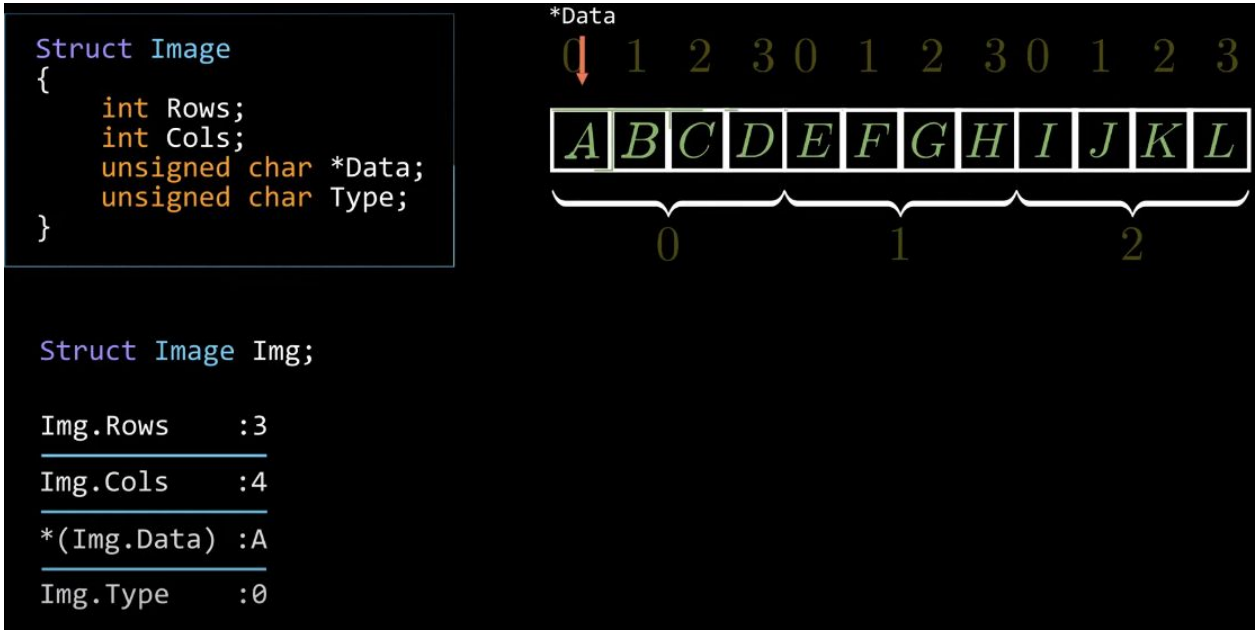
```
typedef struct {
  unsigned width;
  unsigned height;
  unsigned char *image; // RGBA

} png_image;
```

How I handle the images

- For DICOM Image

```
// Struct to hold DICOM image information
struct DICOMImage {
    std::string patientName;
    int rows;
    int cols;
    int bitsAllocated;
    int samplesPerPixel;
    bool isSigned;
    std::vector<int16_t> pixelData; // Adjust type based on Bits Allocated
};
```

Image matrix of pixels



GLCM on CPU

- (0°, 45°, 90°, 135°, 180°, 225°, 270°, 315°)
- ((0,1), (1,1), (1,0), (1,-1), (0,-1), (-1,-1), (-1,0), (-1,1))

```
void glcm_directions(int *matrix, int *glcm, int n_col, int n_row, int
                     int dx, int dy) {
  for (int row = 0; row < n_row; row++) {
    int new_row = row + dx;
    for (int col = 0; col < n_col; col++) {
      int new_col = col + dy;

      int current = matrix[row * n_col + col];

      // Check if the neighboring pixel is within bounds
      if (new_row >= 0 && new_row < n_row && new_col >= 0 && new_col <
        int next = matrix[new_row * n_col + new_col];

        // Validate gray levels to prevent out-of-bounds access
        if (current >= 0 && current < glcm_max && next >= 0 &&
            next < glcm_max) {
          glcm[current * glcm_max + next]++;
        }
      }
    }
  }
}
```

GLCM on GPU

The function entrypoint https://github.com/matheusfrancisco/glcm.cuda/blob/main/main.cu#L25

The function GLCM https://github.com/matheusfrancisco/glcm.cuda/blob/main/glcm_gpu.cu#L67

GLCM on GPU parameters

Int threads_per_block = 256

Int total_pairs = n_row * (n_col - 1) ;; total number of pixel pairs in the direction being calculated. For example in the dx=1, dy=0 this equals n_row * (n_col - 1) as each row has n_col - 1 valid pairs.

Int number_of_blocks = (total_pairs + threads_per_block -1)/threads_per_blocks; this is the total number of Cuda blocks required to process all pixel pairs, this ensures that all pixel pairs are processed even if the total number of pairs isn't a multiple of threads_per_block;

For dx=0, dy=1 **glcm_cuda_direction<<<n_col, n_row>>>(d_matrix, d_glcm, n_col, n_row, max, dy, dx);**

- Each pixel in a column has a pair below it (expect the last row)

For dx = 1, dy =1 **glcm_cuda_direction<<<number_of_blocks, threads_per_block>>>(d_matrix, d_glcm, n_col, n_row, max, dy, dx);**

- **Each thread processes pixel pairs at (row, col) and (row+1, col + 1)**

GLCM on GPU parameters

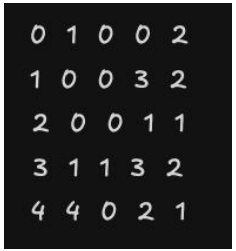
Why Different Configurations for Each Direction?

- **0,180 degrees**: Uses (n_col, n_row) because the calculation is simpler (entire columns can be processed directly).
- **45 and 90 degrees**: Use threads_per_block and number_of_blocks to balance the workload across threads and blocks, especially when pairs are distributed unevenly.
-

GLCM on GPU

The CUDA kernel **glcm_cuda_direction<<<n_col, n_row>>>(...)** is launched with:

- Number of blocks = 5, number of threads per block = 5



Matrix = [0 1 0 0 2 1 0 0 3 2 2 0 0 1 1 3 1 1 3 2 4 4 0 2 1]

The max intensity = 4

Directions = (dx=1, dy =0),

GLCM on GPU

For blockIdx.x = 0,

threadIdx.x = 0 → idx = 0 * 5 + 0 = 0

threadIdx.x = 1 → idx = 0 * 5 + 1 = 1

threadIdx.x = 2 → idx = 0 * 5 + 2 = 2

threadIdx.x = 3 → idx = 0 * 5 + 3 = 3

threadIdx.x = 4 → idx = 0 * 5 + 4 = 4

…

The indexes

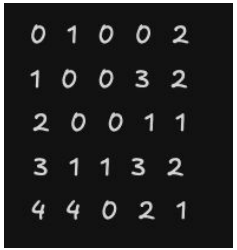
Block 0: 0, 1, 2, 3, 4

Block 1: 5, 6, 7, 8, 9

Block 2: 10, 11, 12, 13, 14

Block 3: 15, 16, 17, 18, 19

Block 4: 20, 21, 22, 23, 24



GLCM on GPU

Thread-by-Thread Updates:

- **idx=0:** (row=0, col=0) neighbor=(0,1)
  - current=matrix[0]=0, neighbor=matrix[1]=1
  - glcm(0,1)++
- **idx=1:** (0,1) neighbor=(0,2)
  - current=1, neighbor=0
  - glcm(1,0)++

1) **idx=2:** (0,2) neighbor=(0,3)
   a) current=0, neighbor=0
   b) glcm(0,0)++
2) **idx=3:** (0,3) neighbor=(0,4)
   a) current=0, neighbor=2
   b) glcm(0,2)++
3) **idx=4:** (0,4) neighbor=(0,5 out of bounds) no update
4) **idx=5:** (1,0) neighbor=(1,1)
   a) current=1, neighbor=0
   b) glcm(1,0)++ (again)
5) **idx=6:** (1,1) neighbor=(1,2)
   a) current=0, neighbor=0
   b) glcm(0,0)++ (again)
6)

GLCM on GPU

1. **idx=7:** (1,2) neighbor=(1,3)
    a. current=0, neighbor=3
    b. glcm(0,3)++
2. **idx=8:** (1,3) neighbor=(1,4)
    a. current=3, neighbor=2
    b. glcm(3,2)++
3. **idx=9:** (1,4) neighbor=(1,5 out of bounds) no update
4. **idx=10:** (2,0) neighbor=(2,1)
    a. current=2, neighbor=0
    b. glcm(2,0)++

….

- **Count how many times each pair incremented**
    a) glcm(0,0): incremented at idx=2,6,11 → total 3
    b) glcm(0,1): idx=0,12 → total 2
    c) glcm(0,2): idx=3,22 → total 2
    d) glcm(0,3): idx=7 → total 1
    e) glcm(1,0): idx=1,5 → total 2
    f) glcm(1,1): idx=13,16 → total 2
    g) glcm(1,3): idx=17 → total 1
    h) glcm(2,0): idx=10 → total 1
    i) glcm(2,1): idx=23 → total 1
    j) glcm(3,1): idx=15 → total 1
    k) glcm(3,2): idx=8,18 → total 2
    l) glcm(4,0): idx=21 → total 1
    m) glcm(4,4): idx=20 → total 1

GLCM on GPU result

Explanation:

- Each (i,j) entry in the GLCM tells how many times a pixel of value i was found adjacent to a pixel of value j in the specified direction.
- Here, (0,0)=3 means we found three pairs where a pixel with gray-level 0 was followed by another pixel with level 0 to the right.
- Similarly, (3,2)=2 means there were two occurrences where a pixel with gray-level 3 had a neighbor to the right with gray-level 2.

By using 5 blocks and 5 threads per block, we covered all 25 pixels, ensuring each pixel index `idx` was processed exactly once, and the resulting GLCM is as computed above.

You can find the function here https://github.com/matheusfrancisco/glcm.cuda/blob/main/glcm_gpu.cu#L67

	j=0	j=1	j=2	j=3	j=4
i=0	3	2	2	1	0
i=1	2	2	0	1	0
i=2	1	1	0	0	0
i=3	0	1	2	0	0
i=4	1	0	0	0	1

GLCM on GPU+OpenMP

I did the same approach for from GPU but with OpenMP for parallel the images.

For e.g 355 images I had do calculate 8 directions, só I run 1 thread for each image and inside this thread I lunch cuda kernels for the 8 directions

GLCM on with skimage and with glcm_cuda.py

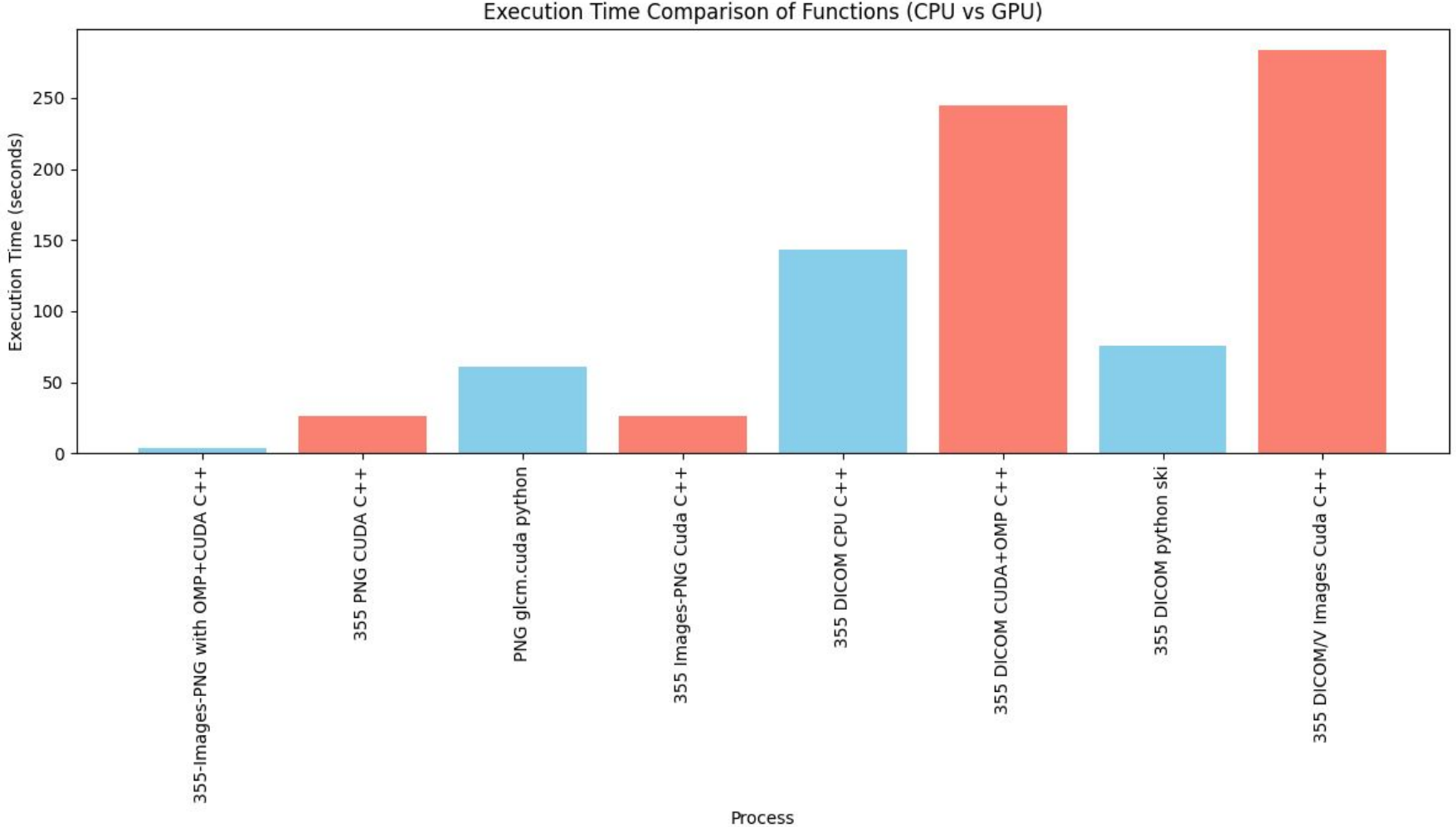
- https://github.com/matheusfrancisco/glcm.cuda/blob/main/benchmark/glcm_py.py#L52
  - Just open the images and apply the GLCM
- https://github.com/matheusfrancisco/glcm.cuda/blob/main/benchmark/glcm_cuda.py#L51

- Problems glcm_cuda
  - It had some memory errors for images with big number NxN and it shrinks the GLCM for 255x255
  - https://eve-ning.github.io/glcm-cupy/glcm/binning.html#binning

Problems

It was really slow the GLCM on GPU C++, faster than the sync version but not so fast than Skimage version.

- DICOM images
- Most code out there apply GLCM in gray images and PNG format which reduce the intensity pixel
- I had problem for image with high intensity pixels


Execution Time Comparison of Functions (CPU vs GPU)
Execution Time (seconds)
0
50
100
150
200
250
355-Images-PNG with OMP+CUDA C++
355 PNG CUDA C++
PNG glcm.cuda python
355 Images-PNG Cuda C++
355 DICOM CPU C++
355 DICOM CUDA+OMP C++
355 DICOM python ski
355 DICOM/V Images Cuda C++
Process

Conclusion

There are plenty of recent papers using features extracted from GLCM.

- There still have few lib that calculates GLCM matrix with a high performance
- It is possible to improve the performance of GLCM matrix calculation and feature extraction in a new lib easy to use
- https://github.com/matheusfrancisco/glcm.cuda?tab=readme-ov-file#references-papers
-

References

1. Shamas, S., Panda, S. N., & Sharma, I. Efficient lung cancer detection in CT scans through GLCM analysis and hybrid classification. *Affiliation*: Chitkara University Institute of Engineering and Technology, Chitkara University, Punjab, India.

2. ZULPE, Nitish; PAWAR, Vrushsen. *GLCM Textural Features for Brain Tumor Classification*. College of Computer Science and Information Technology, Latur, Maharashtra, India; Department of Computational Science, SRTM University Nanded. Available at: https://d1wqtxts1xzle7.cloudfront.net/93568374/IJCSI-9-3-3-354-359-libre.pdf. Accessed on: 29 Oct. 2024.

3. NEHRA, Shalini; RAHEJA, Jagdish; BUTTE, Kaustubh; ZOPE, Ameya. *Detection of Cervical Cancer using GLCM and Support Vector Machines*. In: **2018 6th Edition of International Conference on Wireless Networks & Embedded Systems (WECON)**, 2018. DOI: 10.1109/WECON.2018.8782065.

4. Shahbahrami, A., Pham, T. A., & Bertels, K. (2012). Parallel implementation of Gray Level Co-occurrence Matrices and Haralick texture features on cell architecture. *Journal of Supercomputing, 59*(3), 1455–1477. https://doi.org/10.1007/s11227-011-0556-x

5. Zhou, Y., Jiang, H., Luan, Q., Li, Y., Li, X., & Pei, Y. Tumor classification algorithm via parallel collaborative optimization of single- and multi-objective consistency on PET/CT. *Affiliations*: Software College, Northeastern University, Shenyang, China; Key Laboratory of Intelligent Computing in Medical Image, Ministry of Education, Northeastern University, Shenyang, China; Department of Nuclear Medicine, The First Affiliated Hospital of China Medical University, Shenyang, China.

6.