HISTOGRAMS OF ORIENTED GRADIENTS FOR HUMAN DETECTION

GPU Computing

submitted by

Celeste Nicora - Michele Ghirardelli
(16535A - 16441A)

Department of Computer Science ”Giovanni Degli Antoni”
University of Milan
September 2024

DECLARATION

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.

Place	:	Signature of student	: .....
Date	: September 5, 2024	Name of students	: Celeste Nicora - Michele Ghirardelli

Chapter 1

INTRODUCTION TO THE PROJECT

The application of HOG descriptors in human detection has been extensively studied, including in the work "Histograms of Oriented Gradients for Human Detection" by Dalal and Triggs (2005), which demonstrated the effectiveness of HOG in detecting pedestrians in various poses and backgrounds. This report explores the implementation of a parallel gradient computation for human detection using CUDA, comparing it to a sequential implementation. The code provided is designed to extract features from images using Histogram of Oriented Gradients (HOG).

1.1 OBJECTIVES OF THE PROJECT

- **Measure and Compare Execution Times:** Evaluate and compare the execution times of both the CUDA and the sequential implementations across different image dimensions (64x64, 128x128, and 256x256 pixels) and varying HOG descriptor dimensions.
- **Scalability Assessment:** Investigate how execution time scales with increasing image sizes and descriptor dimensions, highlighting the advantages of CUDA's parallel processing capabilities in stabilizing computation time.
- **Sequential vs. CUDA Performance:** Directly confront the performance of the sequential algorithm with the CUDA algorithm to demonstrate the efficiency gains achieved through parallelization.

1.2 PSEUDOCODE OF THE HOG ALGORITHM IMPLEMENTED

1. **Input Image Preparation**
   (a) **Load Image:** Load the image in grayscale mode.
   (b) **Resize Image**
2. **Gradient Computation**

(a) **Compute Gradients:**

- Compute the gradients G_x and G_y for each pixel in the image using these filters:

$$Kx = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

$$Ky = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

- Calculate the gradient magnitude G and orientation θ for each pixel:

$$G = \sqrt{G_x^2 + G_y^2}$$

$$\theta = \text{atan2}(G_y, G_x)$$

- Adjust the orientation θ to be within a $[0, 180^\circ]$ for unsigned gradients.

3. **Spatial Binning**

(a) **Divide Image into Cells:**

- Partition the image into small, non-overlapping cells (e.g., 8x8 pixels).

(b) **Create Orientation Histograms for each Cell:**

- For each cell, create a histogram of gradient orientations.
- Bin the gradient orientations into a predefined number of bins (e.g., 9 bins).
- Each pixel in the cell votes for a bin based on its orientation, weighted by the gradient magnitude.

4. **Block Normalization**

(a) **Group Cells into Blocks:**

- Group several cells into overlapping blocks (e.g., 2x2 cells per block).
- This overlap ensures robustness to changes in illumination and contrast.

(b) **Normalize Histograms within each Block:**

- Normalize the concatenated histograms within each block to account for variations in lighting and contrast using:

– **L2-norm:**

$$v \rightarrow \frac{v}{\sqrt{\|v\|_2^2 + \varepsilon^2}}$$

5. **Construct Feature Vector - Descriptor Formation**

   (a) **Concatenate normalized Histograms:**

   - Concatenate the normalized histograms from all blocks in the detection window to form the final feature vector (HOG descriptor).

6. **Detection (Optional)**

   (a) **Apply a Classifier:**

   - Use a classifier like a Support Vector Machine (SVM) trained on HOG descriptors to detect objects (e.g., humans) in the image.

Chapter 2

CUDA IMPLEMENTATION

2.1 BLOCK AND GRID CONFIGURATION

In CUDA, the parallel execution of code is managed through a grid of blocks, where each block contains a set of threads.

2.1.1 Determining Block and Grid Sizes

In the provided `computeDescriptorsCUDA` function, the image processing task is parallelized by dividing the work with one thread per pixel.

Here's how the blocks and grid sizes are implemented in the code:

- **Block Size (`dim3 blockSize`):**
  - A block is typically defined by its dimensions, usually in 2D for image processing tasks. In this case, a block of size 16x16 threads is used:

    ```
    dim3 blockSize(16, 16);
    ```

  - This means each block contains $16 \times 16 = 256$ threads.
- **Grid Size (`dim3 gridSize`):**
  - The grid is composed of multiple blocks, where each block handles a portion of the image. The grid dimensions are calculated based on the image dimensions and the block size:

    ```
    dim3 gridSize((image.cols + blockSize.x - 1) / blockSize.x,
                  (image.rows + blockSize.y - 1) / blockSize.y);
    ```

  - Here's the breakdown:
    - `image.cols` is the width of the image, and `image.rows` is the height.
    - `blockSize.x` and `blockSize.y` are the dimensions of the block (16x16 in this case).

    - The grid size in each dimension is calculated by dividing the image dimension by the corresponding block size dimension. The `+ blockSize.x - 1` and `+ blockSize.y - 1` parts ensure that the grid size rounds up when the image dimensions are not perfectly divisible by the block size. This ensures all pixels are covered by the threads.
  - For example, if the image has dimensions 128x128 pixels, the grid size would be:

    `gridSize.x = (128 + 16 - 1) / 16 = 8` and `gridSize.y = (128 + 16 - 1) / 16 = 8`

  - This results in a grid of $8 \times 8 = 64$ blocks, where each block processes a 16x16 pixel area of the image.

2.1.2 Launching the CUDA Kernel

Once the block and grid sizes are determined, the CUDA kernel `computeGradients` is launched:

```
computeGradients<<<gridSize, blockSize>>>(
    d_image, d_histograms, image.cols, image.rows,
    cellSize_g, binWidth, numBins_g, histSize_vec);
```

- **Grid Configuration:**
  - `gridSize`: Specifies the number of blocks in the grid, which is determined based on the image dimensions and block size.
- **Block Configuration:**
  - `blockSize`: Specifies the number of threads per block, configured as 16x16 in this case.
- **Kernel Parameters:**
  - `d_image`: Pointer to the image data on the GPU.
  - `d_histograms`: Pointer to the histogram data on the GPU.
  - `image.cols`, `image.rows`: The dimensions of the image.
  - `cellSize_g`: The size of each cell in the image for histogram computation.
  - `binWidth`: The width of each bin in the histogram.
  - `numBins_g`: The number of bins in the histogram.
  - `histSize_vec`: The total size of the histogram vector.

This configuration ensures that we can utilize the parallel processing capabilities of the GPU to execute the gradient computation per pixel.

2.1.3 Parallelizing the HOG Computation

In the pseudocode presented in **Chapter 1, Section 2**, we focus on points 2 and 3 of the HOG computation process. Each thread is responsible for processing one pixel of the image, calculating the gradient at that pixel, and contributing to the corresponding histogram bin.

By assigning one pixel to each thread, the following steps are performed in parallel:

1. **Gradient Calculation (Point 2):** The kernel computes the gradient magnitude and orientation for each pixel based on its neighboring pixels.

2. **Histogram Formation (Point 3):** Each thread contributes to a histogram for its respective cell, ensuring that the gradient information is distributed across the correct bins in parallel.

Chapter 3

DATA AND EXPERIMENTS

3.1 DATA

The dataset used for this project is the Human Detection Dataset, available on Kaggle[1]. This dataset contains a collection of images categorized into two classes: images with humans and images without humans.

Dataset Description

- **Content:** - The dataset contains CCTV footage images, including both indoor and outdoor scenes. - Half of the images contain humans, and the other half do not.
- **Sources:** - The dataset images are sourced from:
  1. CCTV footage from YouTube.
  2. Open indoor image datasets.
  3. Footage from personal CCTV cameras.
- **Categories:** The dataset is divided into two main categories:
  - **Humans Present:** Images that contain one or more human figures. (559 images)
  - **No Humans:** Images that do not contain any human figures. (362 images)
- **File Formats:** Images are provided in common formats such as JPEG and PNG.

This dataset is particularly useful for evaluating human detection algorithms due to the diversity in poses, lighting conditions, and backgrounds. For this project, the images were preprocessed (using bilinear interpolation[2]) by resizing them to 64x64, 128x128, and 256x256 pixels to test the performance of the CUDA-accelerated and sequential implementations for computing HOG descriptors.

[1] https://www.kaggle.com/datasets/constantinwerner/human-detection-dataset
[2] https://docs.opencv.org/3.4/da/d54/group__imgproc__transform.html

3.2 EXPERIMENTS

In this section, we present the experiments conducted to evaluate the performance of the CUDA-accelerated and the sequential implementations for computing HOG descriptors. The experiments focused on two primary metrics: execution time and memory usage. 3 image dimensions were tested to assess the scalability and time/space efficiency of each approach.

1. Experimental Setup

The experiments were conducted on a system equipped with an NVIDIA T4 GPU (Google Colab). The image dimensions tested were 64x64, 128x128, and 256x256 pixels. For each image size, the HOG descriptors were computed using both the CUDA and sequential algorithms, and the following metrics were recorded:

- **Execution Time:** The time taken to compute the HOG descriptors for each image dimension.
- **Memory Loading Time:** The time load the memory on GPU (This metrics was not taken in the sequential algorithm)
- **Memory Usage:** The amount of GPU memory used at various stages of the CUDA implementation, including before and after memory allocation and after memory deallocation.

2. Execution Time Analysis

2.1 Sequential vs. CUDA Execution Time

The first set of plots (Figure 2.1) compares the execution time of the sequential (SEQ) and CUDA implementations as a function of the HOG descriptor dimension for different image sizes.

- **SEQ Execution Time (Figure 2.2 - Top Plot):** The sequential implementation shows a consistent increase in execution time as the image dimension increases, particularly noticeable with larger image sizes (256x256 pixels). This is expected as the CPU processes each pixel and gradient in sequence, leading to longer computation times with increasing image size.
- **CUDA Execution Time (Figure 2.2 - Bottom Plot):** The CUDA implementation demonstrates significantly lower execution times across all image dimensions compared to the sequential method. Notably, the execution time remains relatively stable as the descriptor dimension increases, indicating the effectiveness of GPU parallelization in handling larger images.

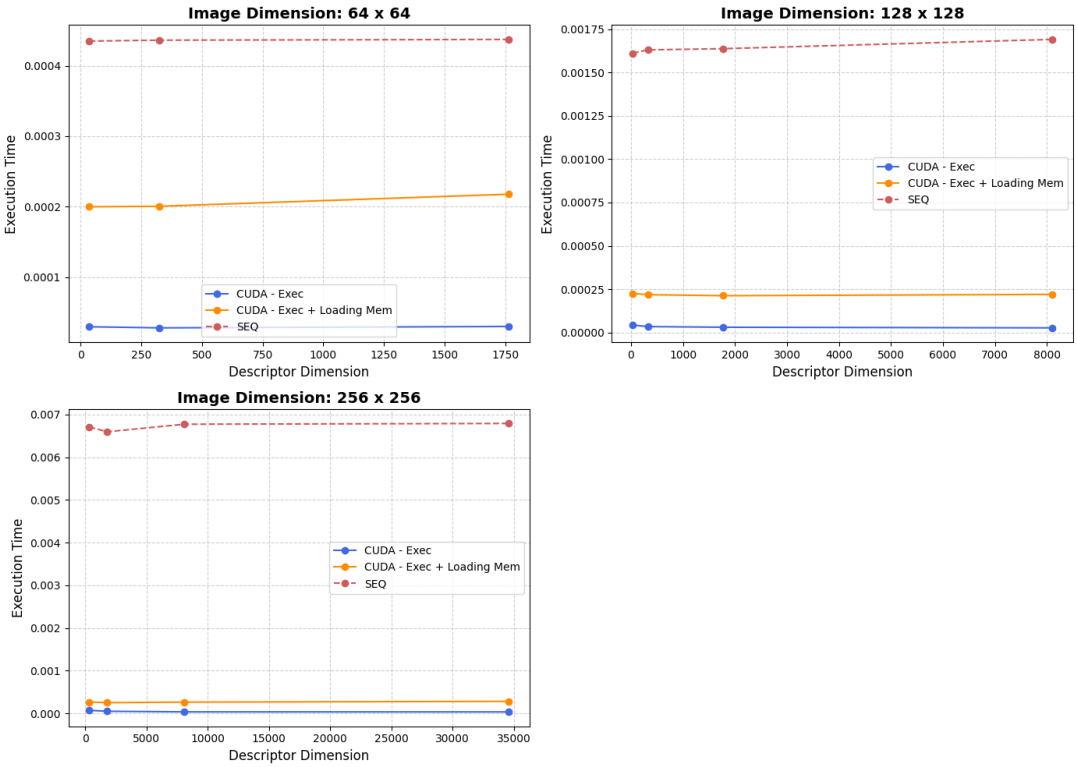

Figure 3.1: Plots image dimension ExecutionTime x Descriptor Dimension

These plots clearly illustrate the substantial performance improvement gained by using CUDA, especially for larger images. The difference in execution times becomes more pronounced as the computational load increases, with CUDA maintaining low execution times while the sequential method's times increase sharply.

2.2 Detailed Comparison for Each Image Dimension

Figure 2.1 provides a more detailed comparison of execution times for each specific image dimension (64x64, 128x128, and 256x256 pixels).

- **64x64 Pixels (Top Left Plot):** Even for small images, the CUDA implementation outperforms the sequential method, with execution times remaining nearly constant. Also the ExecutionTime + Loading Mem outperforms the sequential method.
- **128x128 Pixels (Top Right Plot):** The gap between CUDA and sequential performance widens as the image size increases. CUDA's execution time remains negligible compared to the sequential method, which continues to increase with larger descriptor dimensions.
- **256x256 Pixels (Bottom Plot):** For the largest image size, the sequential implementation shows a significant rise in execution time. In contrast, the CUDA implementation still maintains a low and

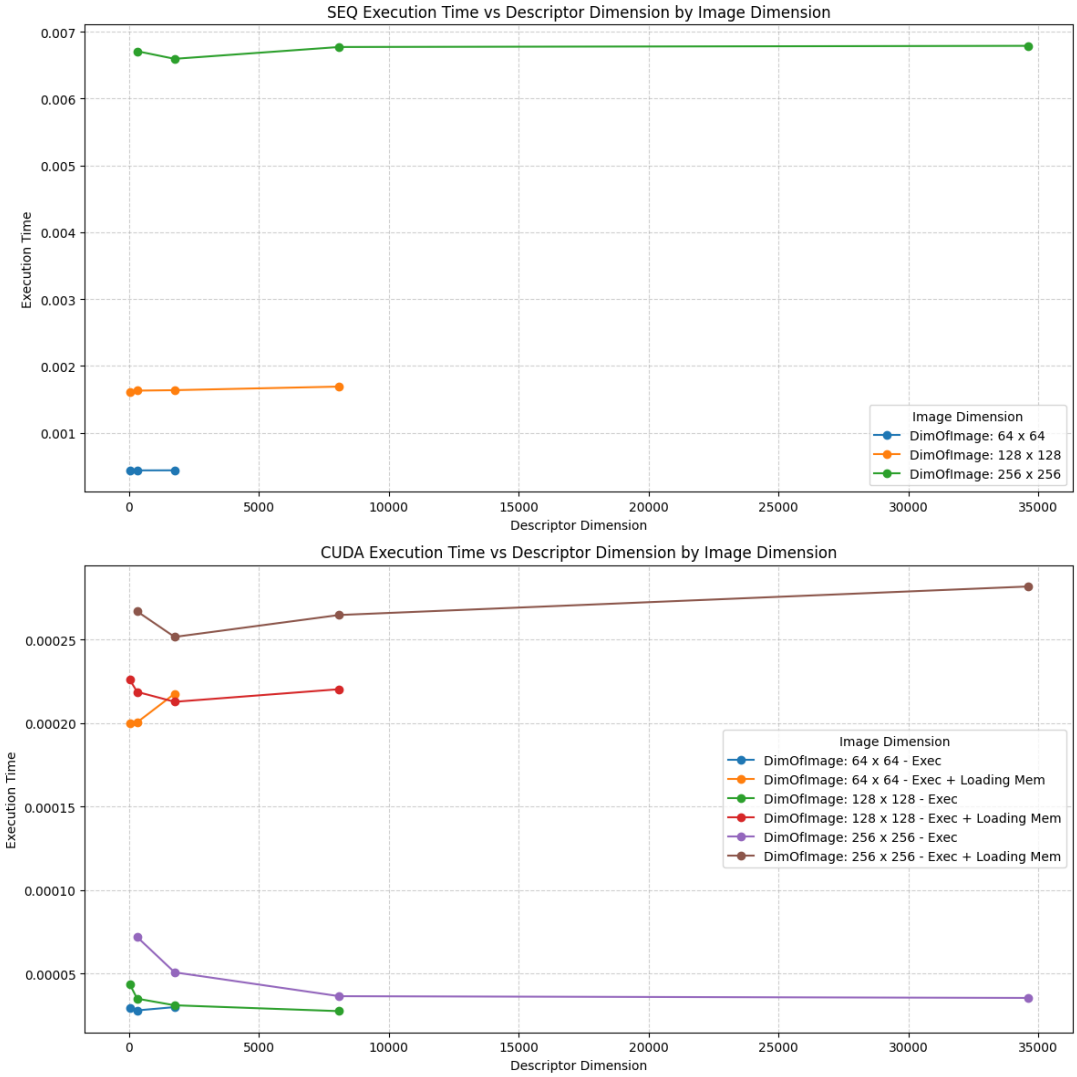

Figure 3.2: Sequential and Cuda Plot ExecutionTimexDescriptorDimension

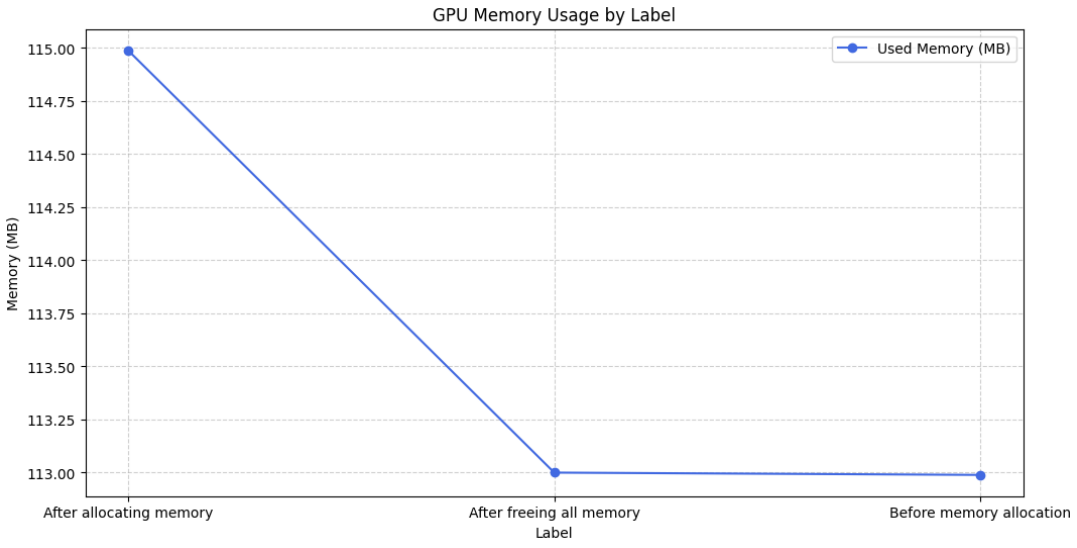

Figure 3.3: Memory Usage mean for every dimension image

stable execution time, highlighting its scalability and efficiency for large-scale image processing tasks.

These detailed comparisons confirm that the CUDA implementation is not only faster but also more scalable than the sequential approach.

3. Memory Usage Analysis

The final plot (Figure 2.3) presents the GPU memory usage at different stages of the CUDA implementation. The three stages measured include:

- **Before Memory Allocation:** Represents the baseline GPU memory usage before any processing begins.
- **After Allocating Memory:** Shows the memory usage after allocating space for image data and gradient computations.
- **After Freeing All Memory:** Indicates the memory usage after all allocated resources have been freed.

The plot demonstrates that there is no significant increase in memory usage during the CUDA implementation, particularly all smaller image dimensions. Across all tested stages—before memory allocation, after memory allocation, and after freeing memory—the GPU memory usage remains relatively stable for every image dimension

4. Conclusion from Experiments

The experiments conducted demonstrate the clear benefits of using CUDA for HOG descriptor computation over the sequential CPU-based approach. The CUDA implementation significantly reduces execution time, particularly as image size and descriptor dimensions increase, and efficiently manages GPU memory usage. These advantages make CUDA a highly effective solution for real-time image processing tasks where speed and efficiency are critical.

Chapter 4

ACCURACY

In the context of this project, accuracy is a crucial metric for evaluating the performance of both the CUDA-accelerated and sequential implementations of the HOG (Histogram of Oriented Gradients) descriptor for human detection.

We aim to compare the performance of different configurations based on various parameters such as cell size, block size, image dimension, and descriptor dimension. Evaluating accuracy alongside recall allows us to ensure that the model performs well not only on average but also in correctly identifying positive cases (images with humans). Here the results:

Cell	Block	Accuracy	Recall	DescriptorDim	ImageDim	Type
32	2	0.7568	0.9912	324	128	SEQ
32	2	0.7514	0.9912	324	128	CUDA
16	2	0.7459	0.9735	324	64	CUDA
16	2	0.7351	0.9646	324	64	SEQ
64	2	0.7189	0.9469	324	256	SEQ
64	2	0.7135	0.9469	324	256	CUDA
64	2	0.7081	0.9027	36	128	SEQ
64	2	0.7081	0.9027	36	128	CUDA
32	2	0.7081	0.9115	36	64	CUDA
32	2	0.7027	0.9027	36	64	SEQ
16	2	0.6162	1.0	8100	256	CUDA
16	2	0.6108	1.0	1764	128	CUDA
16	2	0.6108	1.0	1764	128	SEQ
32	2	0.6108	1.0	1764	256	CUDA
32	2	0.6108	1.0	1764	256	SEQ
8	2	0.6324	1.0	34596	256	CUDA
8	2	0.6324	1.0	34596	256	SEQ
8	2	0.6324	1.0	8100	128	SEQ

Cell	Block	Accuracy	Recall	DescriptorDim	ImageDim	Type
8	2	0.6270	1.0	8100	128	CUDA
8	2	0.6108	1.0	1764	64	CUDA
8	2	0.6108	1.0	1764	64	SEQ
16	2	0.6108	0.9912	8100	256	SEQ

Chapter 5

CONCLUSIONS

In this chapter, we summarize the key findings and conclusions from our project on parallelized HOG computation using CUDA.

5.1 PERFORMANCE BENEFITS OF CUDA PARALLELIZATION

The CUDA implementation significantly outperforms the sequential CPU-based approach in terms of execution time. For example, while the sequential method shows a substantial increase in execution time as the image size increases, the CUDA approach remains stable across image sizes. This difference is most evident with larger images, such as 256x256 pixels, where CUDA maintains low and constant execution times while the sequential method grows exponentially (Figure 3.3).

5.2 SCALABILITY

As demonstrated in the experiments, CUDA offers considerable scalability advantages. For instance, in the case of 64x64 pixel images, the CUDA implementation exhibits almost no increase in execution time as descriptor dimensions increase. For larger images (128x128 and 256x256 pixels), CUDA continues to outperform the sequential implementation, as shown in Figures 3.3 and 3.3.

5.3 GPU MEMORY EFFICIENCY

Memory usage analysis revealed that the CUDA implementation efficiently manages GPU memory across different stages, including memory allocation and deallocation (Figure 3.3). For all image dimensions, the GPU memory usage remained consistent, indicating that the CUDA implementation not only accelerates computation but also ensures that memory resources are utilized efficiently.

5.4 ACCURACY CONSIDERATIONS

Both the CUDA and sequential implementations provided high accuracy in human detection tasks. For example, the accuracy for detecting humans in 128x128 pixel images using the CUDA implementation with a 32-cell/2 block configuration was 0.7514, compared to 0.7568 for the sequential approach (Table 5.1). These values indicate that the performance gains from CUDA do not come at the cost of reduced accuracy, ensuring that the system remains reliable.

Table 5.1: Comparison of Accuracy and Recall for CUDA and Sequential Implementations

Cell	**Block**	**Accuracy**	**Recall**	**DescriptorDim**	**ImageDim**
32	2	0.7568	0.9912	324	128 (SEQ)
32	2	0.7514	0.9912	324	128 (CUDA)
64	2	0.7189	0.9469	324	256 (SEQ)
64	2	0.7135	0.9469	324	256 (CUDA)

5.5 FUTURE IMPROVEMENT

Further optimizations could explore the use of GPUs on larger images and implement a logic where a thread have the responsability of compute a portion of pixels