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

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:** Resize image to the desired dimension.
2. **Gradient Computation**

(a) **Compute Gradients:**

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

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

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

$$G = \sqrt{G_x^2 + G_y^2}, \quad \theta = \text{atan2}(G_y, G_x)$$

- Adjust the orientation θ to be within a $[0, 180°]$ 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);
```

- **Kernel, Block and Grid Configuration:**
  - `gridSize`: Specifies the number of blocks in the grid, which is determined based on the image dimensions and block size.
  - `blockSize`: Specifies the number of threads per block, configured as 16x16 in this case.
  - `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.

2.1.4 Colab and Personal Computer Implementation

In our project, we used a different librarie for Google Colab and personal computer due to WSL compatibility issues on Windows.

For Colab, we used the OpenCV library, which is well-supported and easy to use for image processing. However, on the personal computer, OpenCV faced compatibility problems with WSL . To resolve this, we switched to the stb library[1], a lightweight option that worked smoothly in the WSL environment. The Colab implementation can be found in the notebook on the repo called `Gradient_Computation.ipynb` or in the file `colab_code.cu` and the PC implementation in the file `gradient_computation.cu`.

[1]https://github.com/nothings/stb

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 CCTV footage images (PNG format) categorized into two classes:with humans (559) and without humans (362). Images were sourced from: CCTV footage from YouTube, open indoor image datasets and footage from personal CCTV cameras. This dataset is useful 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.

Note: The database 'INRIA' suggested by Dalal and Triggs is no longer available to the public, for that reason no comparison will be made against the original database.

3.2 EXPERIMENTS

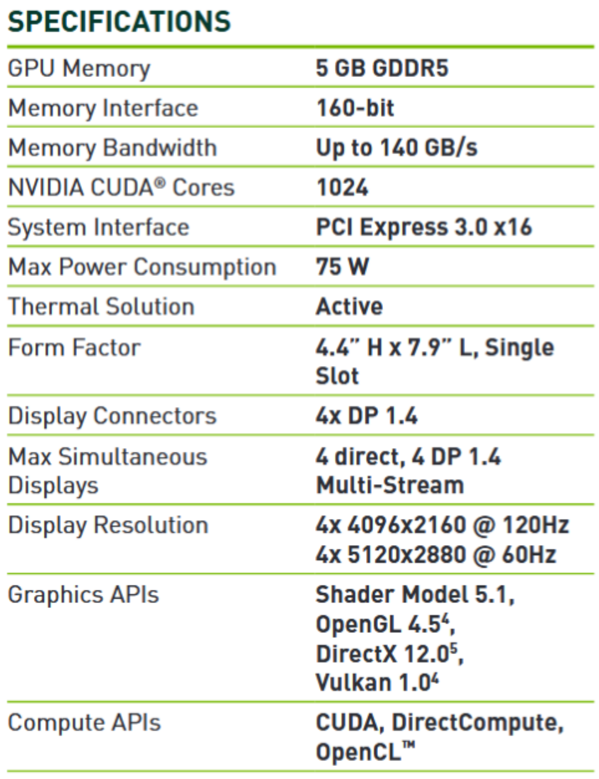
In this section, we present the experiments conducted to evaluate the performance of CUDA-accelerated and sequential implementations. The experiments focused mainly on execution time. 3 image dimensions were tested to assess the scalability and time/space efficiency of each approach.

1. Experimental Setup

The experiments were conducted on two environments:

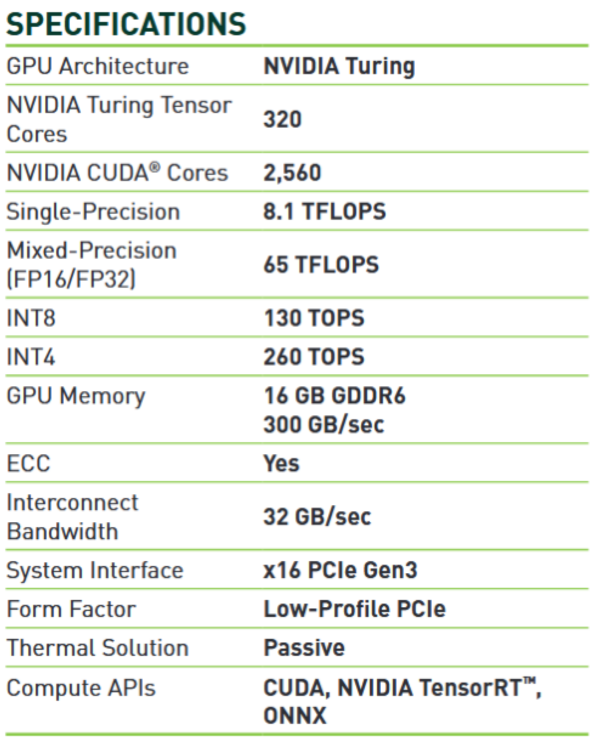
- Google Colab system which supplies an NVIDIA T4 GPU.
- Personal machine equipped with a NVIDIA P2000 Quadro (5GB).

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



SPECIFICATIONS

GPU Memory	**5 GB GDDR5**
Memory Interface	**160-bit**
Memory Bandwidth	**Up to 140 GB/s**
NVIDIA CUDA® Cores	**1024**
System Interface	**PCI Express 3.0 x16**
Max Power Consumption	**75 W**
Thermal Solution	**Active**
Form Factor	**4.4" H x 7.9" L, Single Slot**
Display Connectors	**4x DP 1.4**
Max Simultaneous Displays	**4 direct, 4 DP 1.4 Multi-Stream**
Display Resolution	**4x 4096x2160 @ 120Hz 4x 5120x2880 @ 60Hz**
Graphics APIs	**Shader Model 5.1, OpenGL 4.5[4], DirectX 12.0[5], Vulkan 1.0[4]**
Compute APIs	**CUDA, DirectCompute, OpenCL™**



SPECIFICATIONS

GPU Architecture	**NVIDIA Turing**
NVIDIA Turing Tensor Cores	**320**
NVIDIA CUDA® Cores	**2,560**
Single-Precision	**8.1 TFLOPS**
Mixed-Precision (FP16/FP32)	**65 TFLOPS**
INT8	**130 TOPS**
INT4	**260 TOPS**
GPU Memory	**16 GB GDDR6 300 GB/sec**
ECC	**Yes**
Interconnect Bandwidth	**32 GB/sec**
System Interface	**x16 PCIe Gen3**
Form Factor	**Low-Profile PCIe**
Thermal Solution	**Passive**
Compute APIs	**CUDA, NVIDIA TensorRT™, ONNX**

The image dimensions tested were 64x64, 128x128, and 256x256 pixels, each computed using both the CUDA and sequential algorithms and the following metrics recorded:

- **Execution Time:** Time needed to compute the HOG descriptors for each image dimension.
- **Memory Loading Time:** Time to 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/after memory allocation and after memory deallocation.

2. Execution Time Analysis

Sequential vs. CUDA Execution Time

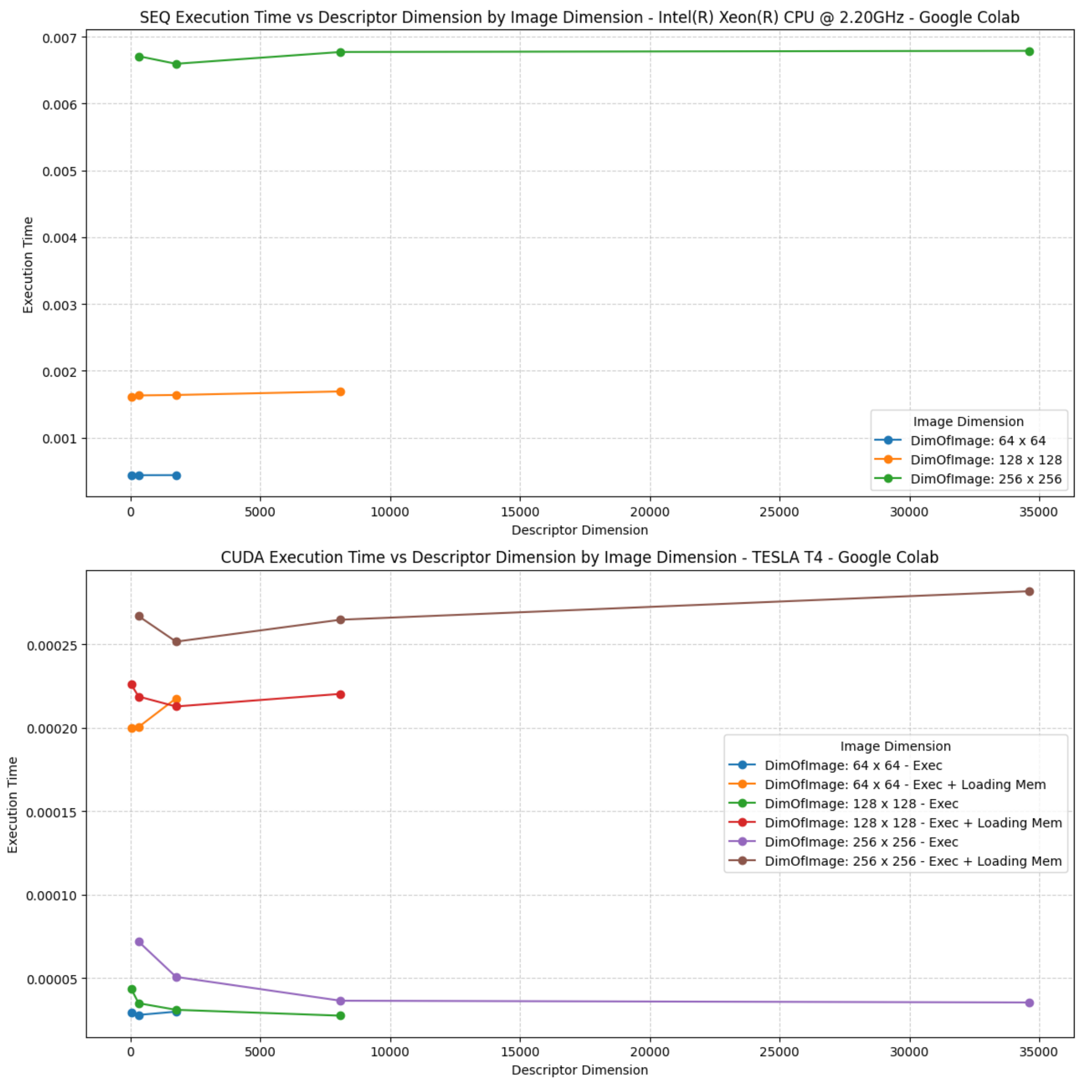


Figure 3.1: SEQ and CUDA execution times compared to descriptor dimensions in Colab

- **SEQ Execution Time in Colab (Figure 3.1 - 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 in Colab (Figure 3.1 - 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.

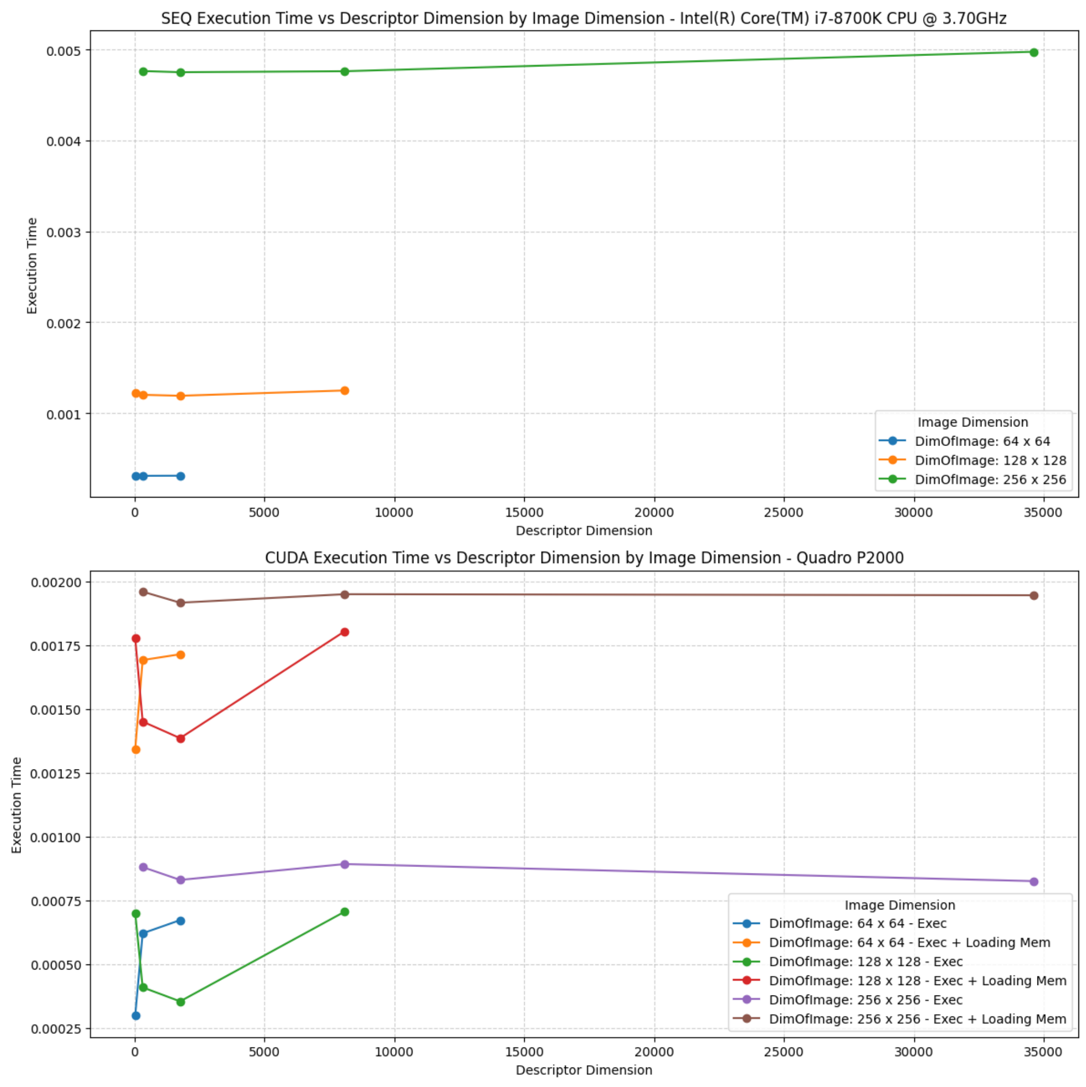


Figure 3.2: SEQ and CUDA execution times compared to descriptor dimensions in Colab

- **SEQ Execution Time in PC (Figure 3.2 - Top Plot):** The sequential implementation on the i7 CPU is slightly faster compared to Xeon(R) CPU on Colab but it still maintains a similar increase as the image dimension increases.

- **CUDA Execution Time in PC (Figure 3.2 - Bottom Plot):** The CUDA implementation on the Quadro GPU follows a different pattern from the T4 GPU. It maintains a higher time, even if we take into consideration the time it takes to transfer data. We can also observe that on smaller images the descriptor dimension has a noticeable influence on the total time.

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.

Detailed Comparison for Each Image Dimension

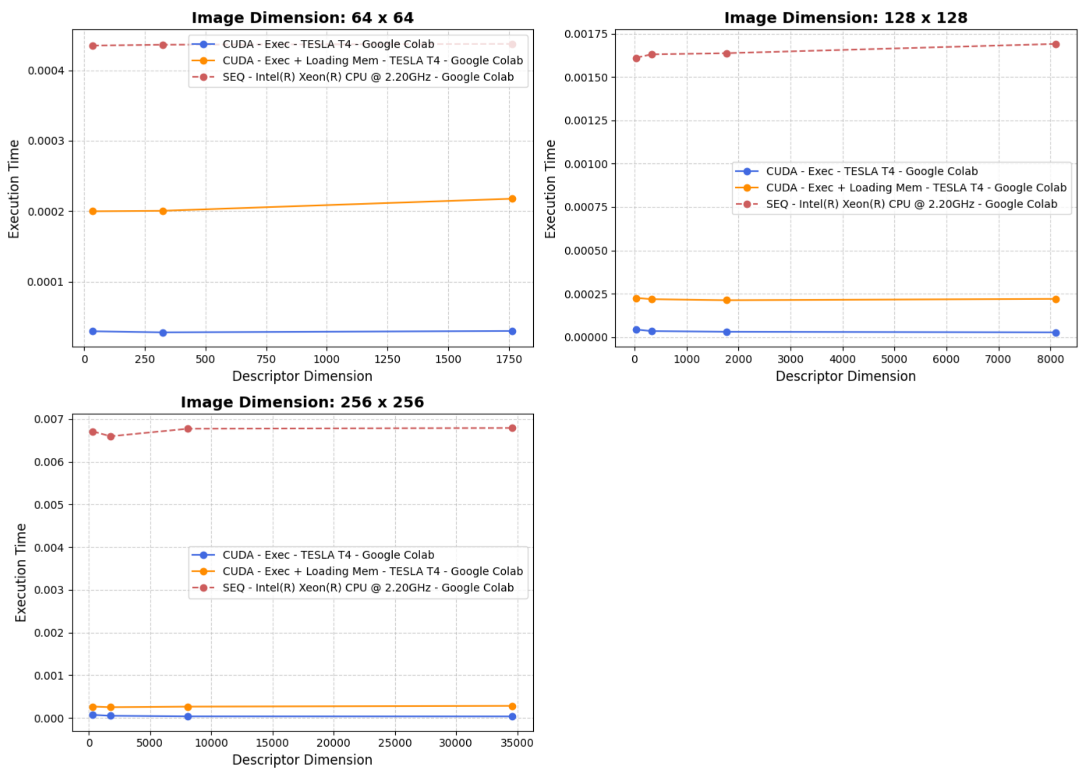


Figure 3.3: Plots with image dimensions and their execution time using Colab

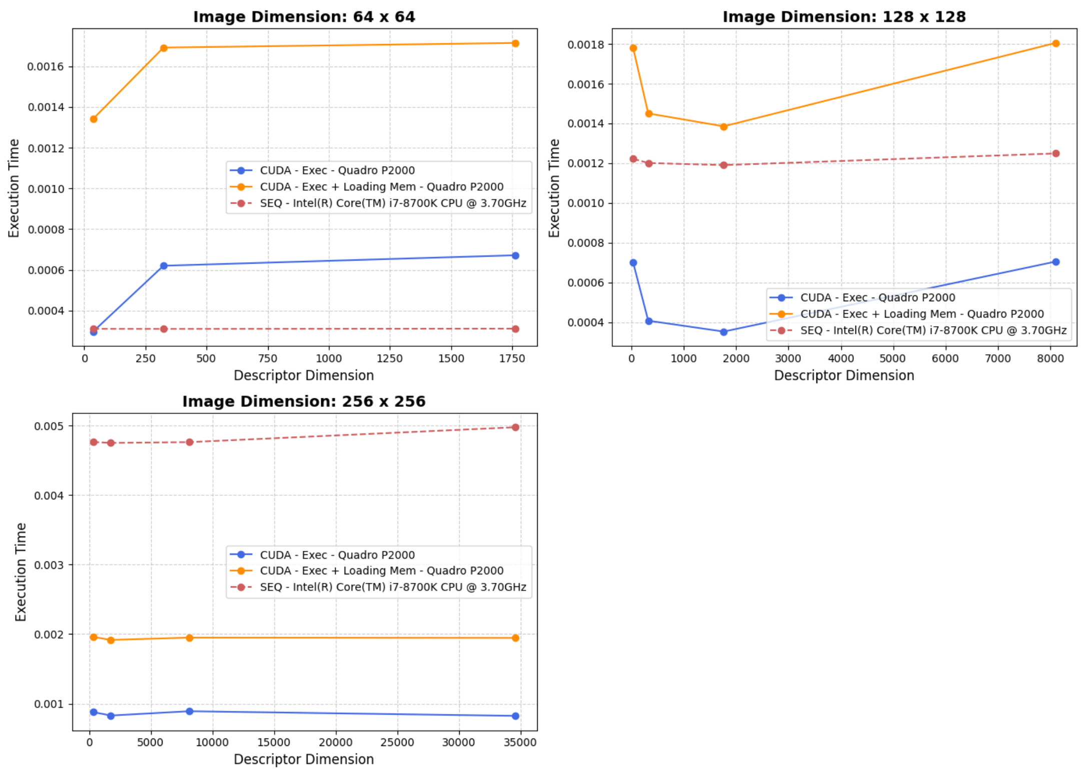


Figure 3.4: Plots with image dimensions and their execution time using PC

Figures 3.3 and 3.4 provides a more detailed comparison of execution times for each specific image dimension (64x64, 128x128, and 256x256 pixels).

- **64x64 Pixels:** For small images, the CUDA implementation outperforms the sequential method only on the Tesla T4 where execution times remain nearly constant, while Quadro P2000 shows noticeable memory loading overhead.
- **128x128 Pixels:** The gap between CUDA and the sequential implementation widens on Colab and CUDA's execution time remains negligible compared to the sequential method. Instead the Quadro P2000 starts to outperform the CPU when considering only execution times but struggles to maintain low latency with larger descriptor sets.
- **256x256 Pixels:** For the largest image size, the sequential implementation shows a significant rise in execution time in both environments. In contrast, CUDA implementation stabilizes also on the Quadro P2000 and maintains a low and stable execution time, highlighting its scalability and efficiency for large-scale image processing tasks.

The Tesla T4 GPU on Colab outperforms the Quadro P2000 across all image dimensions and descriptor sizes, especially for larger descriptors and higher image dimensions. Memory transfer overhead is significant with the Quadro P2000 GPUs, in contrast, Tesla T4 handles memory loading more efficiently, leading to minimal execution time increase even when descriptor dimensions grow.

3. 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 metric calculated with the training and testing of an SVM(Gaussian) using python to evaluate the performance of both CUDA-accelerated and sequential implementations of HOG descriptors for human detection and to check if the results are coherent to the starting paper.

The aim is 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** allow us to ensure that the model performs well. Here are the results:

Table 4.1: Sorted Results Table

Cell	Block	Accuracy	Recall	Descriptor	Image	Type	Processor
32	2	0.795	0.920	36	64	SEQ	Intel i7-8700K
32	2	0.800	0.938	36	64	CUDA	Quadro P2000
16	2	0.757	0.894	324	64	SEQ	Intel i7-8700K
16	2	0.762	0.903	324	64	CUDA	Quadro P2000
8	2	0.762	0.965	1764	64	SEQ	Intel i7-8700K
8	2	0.768	0.973	1764	64	CUDA	Quadro P2000
64	2	0.805	0.920	36	128	CUDA	Quadro P2000
64	2	0.805	0.929	36	128	SEQ	Intel i7-8700K
32	2	0.751	0.894	324	128	CUDA	Quadro P2000
32	2	0.757	0.894	324	128	SEQ	Intel i7-8700K
16	2	0.611	1.000	1764	128	SEQ	Intel i7-8700K
16	2	0.611	1.000	1764	128	CUDA	Quadro P2000
8	2	0.616	1.000	8100	128	SEQ	Intel i7-8700K
8	2	0.611	1.000	8100	128	CUDA	Quadro P2000
64	2	0.762	0.947	324	256	SEQ	Intel i7-8700K

Cell	Block	Accuracy	Recall	Descriptor	Image	Type	Processor
64	2	0.762	0.947	324	256	CUDA	Quadro P2000
32	2	0.611	1.000	1764	256	SEQ	Intel i7-8700K
32	2	0.611	1.000	1764	256	CUDA	Quadro P2000
16	2	0.611	1.000	8100	256	SEQ	Intel i7-8700K
16	2	0.611	1.000	8100	256	CUDA	Quadro P2000
8	2	0.622	0.991	34596	256	CUDA	Quadro P2000
8	2	0.627	0.991	34596	256	SEQ	Intel i7-8700K
32	2	0.610	1.000	1764	128	CUDA	TESLA T4
32	2	0.610	1.000	1764	128	SEQ	Intel Xeon @ 2.20GHz
64	2	0.708	0.903	36	128	CUDA	TESLA T4
64	2	0.708	0.903	36	128	SEQ	Intel Xeon @ 2.20GHz
8	2	0.627	1.000	8100	128	CUDA	TESLA T4
8	2	0.632	1.000	8100	128	SEQ	Intel Xeon @ 2.20GHz
16	2	0.616	1.000	8100	256	CUDA	TESLA T4
16	2	0.611	0.991	8100	256	SEQ	Intel Xeon @ 2.20GHz
32	2	0.611	1.000	1764	256	CUDA	TESLA T4
32	2	0.611	1.000	1764	256	SEQ	Intel Xeon @ 2.20GHz
64	2	0.714	0.947	324	256	CUDA	TESLA T4
64	2	0.719	0.947	324	256	SEQ	Intel Xeon @ 2.20GHz
8	2	0.632	1.000	34596	256	CUDA	TESLA T4
8	2	0.632	1.000	34596	256	SEQ	Intel Xeon @ 2.20GHz
16	2	0.746	0.973	324	64	CUDA	TESLA T4

Cell	Block	Accuracy	Recall	Descriptor	Image	Type	Processor
16	2	0.735	0.965	324	64	SEQ	Intel Xeon @ 2.20GHz
32	2	0.708	0.912	36	64	CUDA	TESLA T4
32	2	0.703	0.903	36	64	SEQ	Intel Xeon @ 2.20GHz
8	2	0.611	1.000	1764	64	CUDA	TESLA T4
8	2	0.611	1.000	1764	64	SEQ	Intel Xeon @ 2.20GHz

The best accuracy was produced by the Intel and Quadro runs, the reason could be various but the Quadro P2000 and Tesla T4 have different architectures optimized for different tasks. Quadro GPUs are designed for rendering/visual computing, which means they are optimized for tasks involving high precision and real-time rendering, as well as single-precision floating-point performance. On the other hand, Tesla T4 is primarily designed for inference tasks in machine learning and deep learning models, which may not directly align with the requirements of HOG descriptor computation. The Tesla architecture is also not fully utilized for our task, especially on smaller images. Overall we can see that CUDA runs are very similar in accuracy to sequential ones, so the speed up gained doesn't come at the cost of loss in accuracy.

Chapter 5

CONCLUSIONS

In this last 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, especially when utilizing latest generation GPUs. 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, although the memory loading part is often a bottleneck and can lead to higher total times in older GPUs. 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 3.4).

5.2 SCALABILITY

As demonstrated in the experiments, CUDA offers significant scalability benefits. For example, in the case of 64x64 pixel images, the increase in execution time as descriptor dimensions grow is substantially smaller in the CUDA implementation compared to the sequential implementation. For larger images (128x128 and 256x256 pixels), although both implementations show increased execution time, CUDA still outperforms the sequential approach with far less incremental slowdown, as illustrated in Figures 3.1 and 3.2.

5.3 ACCURACY CONSIDERATIONS

Both the CUDA and sequential implementations demonstrated high accuracy in human detection tasks. These results confirm that the performance improvements provided by CUDA do not compromise accuracy, maintaining the system's reliability.

Cell	**Block**	**Accuracy**	**Recall**	**Descriptor**	**Image**	**Type**	**Processor**
64	2	0.805	0.929	36	128	SEQ	$Inteli7-8700K$
64	2	0.805	0.920	36	128	$CUDA$	$QuadroP2000$
32	2	0.800	0.938	36	64	$CUDA$	$QuadroP2000$
32	2	0.795	0.920	36	64	SEQ	$Inteli7-8700K$
64	2	0.762	0.947	324	256	$CUDA$	$QuadroP2000$

The highest accuracy (0.805) is observed in both CUDA and sequential implementations using 64 cells on 128x128 images, demonstrating that CUDA improves performance without compromising accuracy. On smaller images (64x64), CUDA slightly outperforms the sequential method (0.800 vs. 0.795). Larger images (256x256) with 324 descriptors show a slight accuracy drop (0.762), but CUDA remains effective. Results indicate that smaller descriptors (36) are optimal for accuracy, and overall, CUDA offers comparable accuracy to sequential processing but with faster execution.

5.4 FUTURE IMPROVEMENT AND CONSIDERATIONS

Further optimizations could explore the use of Tesla GPUs not limited by the usage of Colab on larger and/or RGB images. While assigning threads to pixels has proven an effective strategy, we could try implementing a logic where a thread have the responsability of compute a portion of pixels.