Serial and Parallel Implementation of Viola Jones and Analysis

Ibrahim Binmahfood
Kunjan Vyas
Robert Wilcox

Agenda

Overview of Viola Jones

Serial Implementation

Parallel Implementation

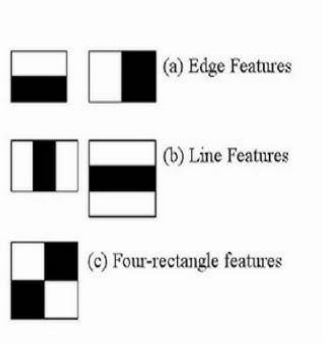
Outfiles

Comparison

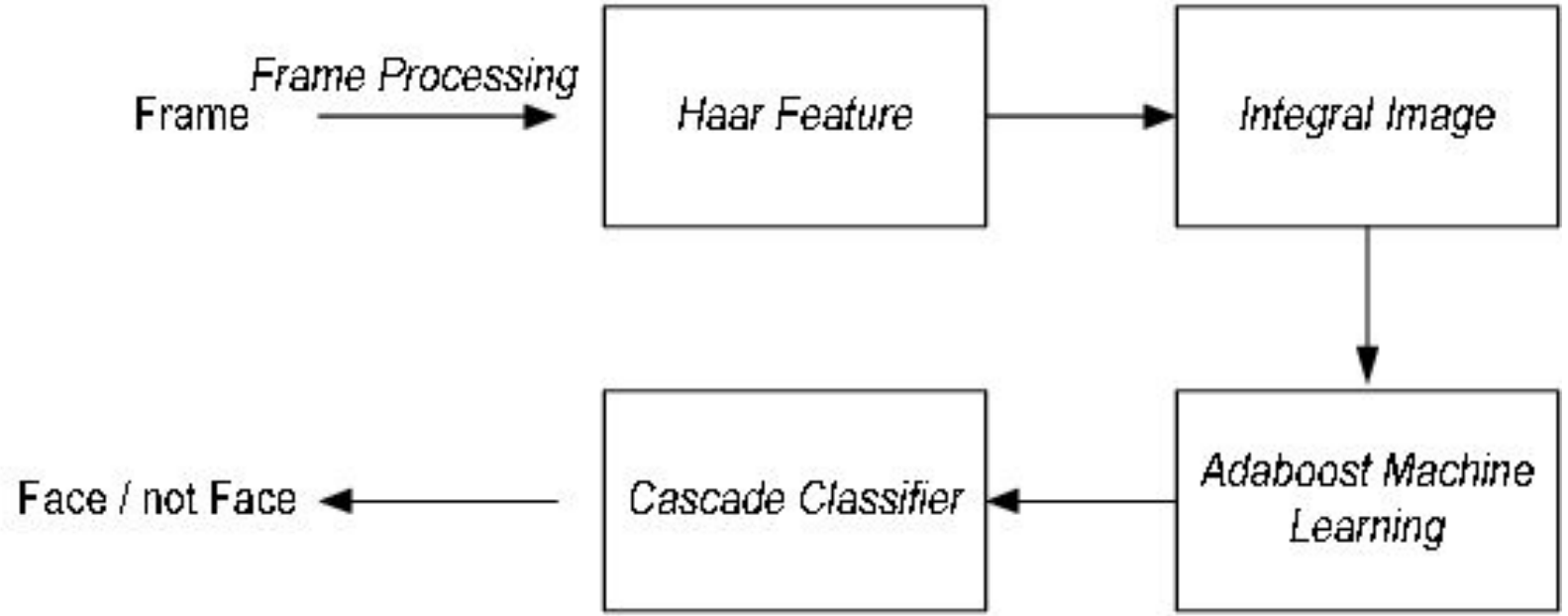
References

What is Viola Jones?

The **Viola-Jones face detection algorithm** is a fast and efficient approach for identifying faces in images. It begins by extracting features using Haar-like filters, which capture edges and textures to differentiate faces from the background. To speed up computation, the algorithm employs the Integral Image technique, which transforms the image into a summed representation, allowing rapid feature calculations. Next, the AdaBoost learning algorithm selects the most relevant features while discarding unnecessary ones, enhancing accuracy. Finally, a cascade classifier is used to filter out non-face regions in multiple stages, ensuring a balance between speed and precision. This structured approach makes the algorithm well-suited for real-time face detection in applications like security and photography.



Flow:


Frame
Frame Processing
Haar Feature
Integral Image
Adaboost Machine Learning
Cascade Classifier
Face / not Face

Serial Implementation and Directory Structure

The **Viola-Jones face detection algorithm** in its **serial** form processes an image step by step, applying Haar-like feature extraction, integral image computation, AdaBoost learning, and the cascade classifier in a sequential manner. The sliding window scans across the image at multiple scales, evaluating each region individually, which can be computationally expensive, especially for high-resolution images.

We based our implementation of Viola-Jones on Francesco Comaschi’s code from 2012. Comaschi’s implementation was straightforward, open source, and was structured well for conversion to cuda code.

Parallel Implementation and Cuda Details

Francesco Comaschi's object detection code uses a Run-time Adaptive Sliding Window, which dynamically adjusts the step size to improve speed without reducing accuracy, which is where parallelization comes into play.

1. **main.cpp:** Initializes the detection pipeline, handles input/output, and manages CUDA detection calls.(Host)
2. **haar_cuda.cpp/h:** Manages cascade classifier logic, setting up integral images and classifier parameters.
3. **image_cuda.c/h:** Reads and writes PGM images and computes integral images on the CPU.
4. **class.txt & info.txt:** Hard-coded model data from trained classifier
5. **cuda_detect.cu/h:** CUDA kernel implements parallel detection. The primary GPU computation occurs here, running sliding window detection in parallel.
6. **rectangles_cuda.cpp/h:** Groups detection results and draws bounding rectangles around detected objects.

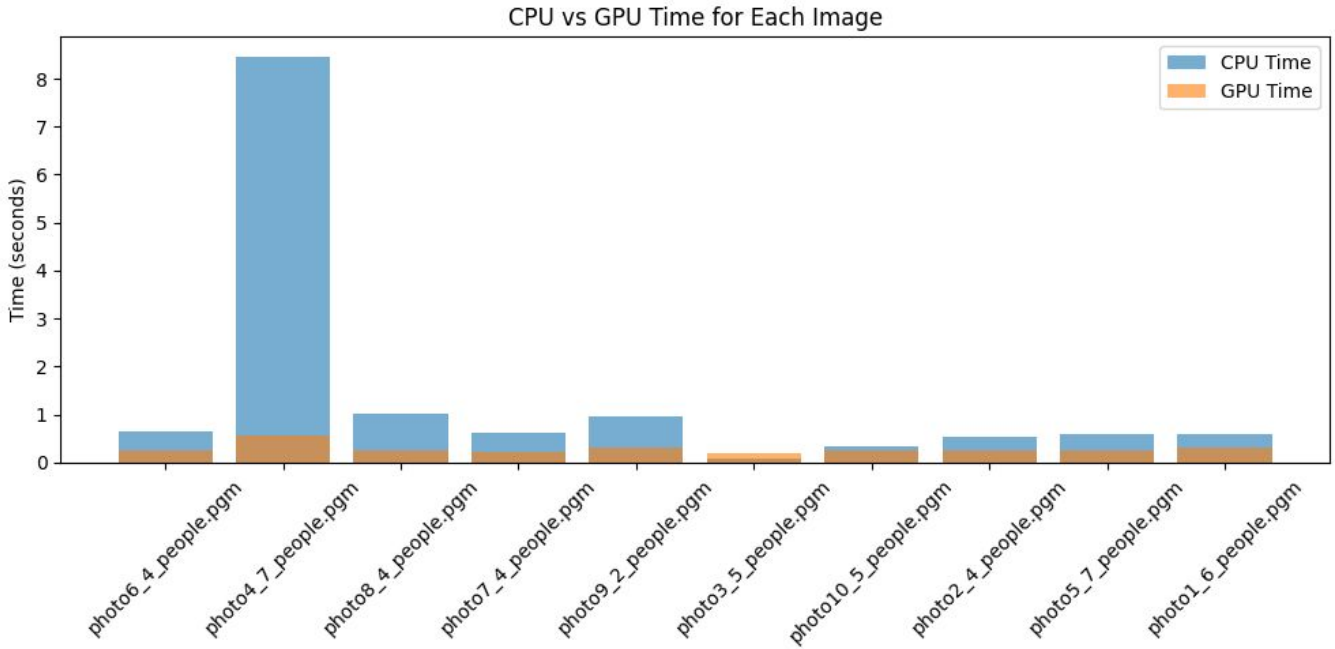
Test Setup

- We have a flexibility to build a CPU-based implementation of the VJ algorithm or a GPU-accelerated CUDA version.
- We have a sophisticated Make file in place which sets appropriate compiler flags based on the program.
- To test out GPU Version we used SM_86 capability 3060 Ti or a T4 with SM_75 compute capability via Google Colab.
- Upon the execution of code, the generated result files such as .pgm output images with detected faces and comprehensive logs containing execution time and number of faces detected are generated.
- The post processing script takes in this data to plot trends analyzing GPU and CPU outputs.
- We used Github to effectively collaborate and track progress.

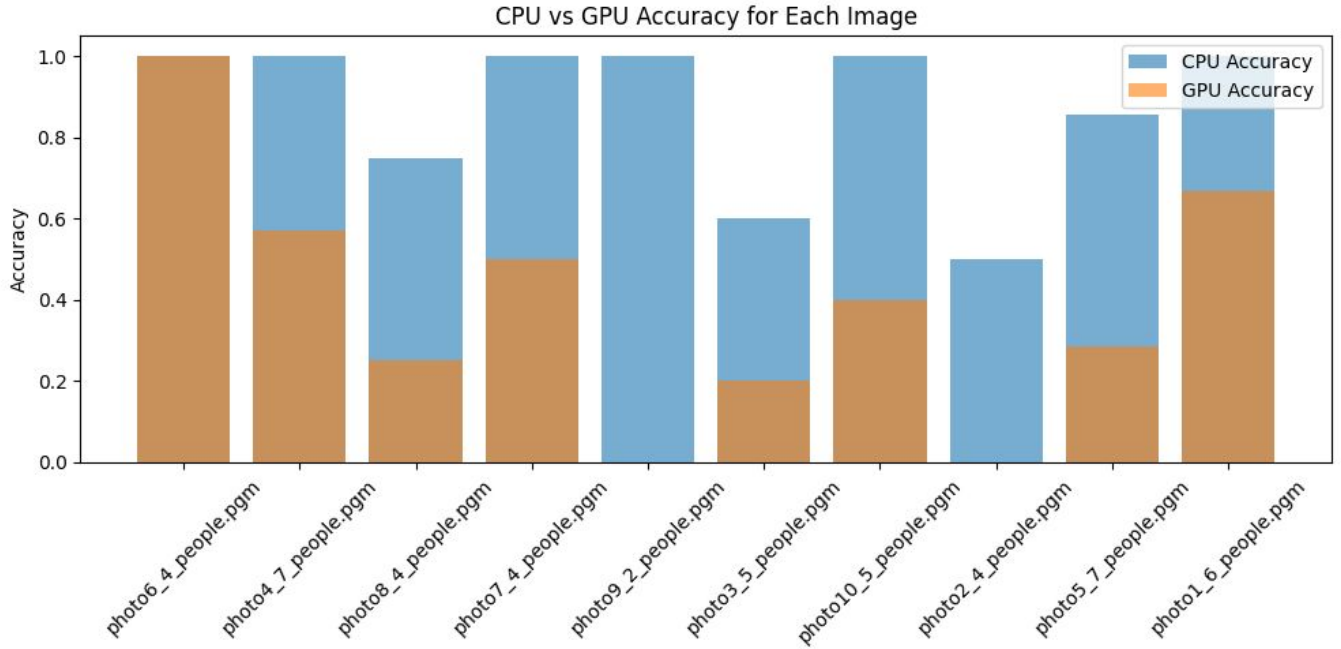
Results - Logs and Generated Images

```
- entering main function --
ound 1 CUDA device(s).

eading image file: PGM_Images/photo6_4_people.pgm
- loading cascade classifier --
- cascade classifier loaded --
- linking integral images to cascade --
- After setImageForCascadeClassifier call --
- integral images linked to cascade --
omputed extra offsets: extra_x = 24, extra_y = 24
djusted detection window size: (48, 48)

UDA detection detected 4 candidates.
ime = 239928432 nanoseconds    (0.239928432 sec)

DEBUG] CUDA Candidate 0: x=273, y=143, width=28, height=28
DEBUG] CUDA Candidate 1: x=345, y=167, width=26, height=26
DEBUG] CUDA Candidate 2: x=451, y=152, width=29, height=29
DEBUG] CUDA Candidate 3: x=519, y=167, width=35, height=35
- saving output --
- image saved as ./Detected_Images/gpu/photo6_4_people.pgm -
```

Results - GPU vs CPU Time Taken



Results - GPU vs CPU Accuracy



Results - Speedup

```
!python3 run_parser.py logs/cpu/ logs/gpu/ logs/
```

```
Parsing logs...
Plots generated in logs/plots/
python3 run_parser.py ./logs/cpu ./logs/gpu ./logs
photo6_4_people.pgm - Speedup: 2.66
photo4_7_people.pgm - Speedup: 14.80
photo8_4_people.pgm - Speedup: 4.00
photo7_4_people.pgm - Speedup: 2.64
photo9_2_people.pgm - Speedup: 2.99
photo3_5_people.pgm - Speedup: 0.43
photo10_5_people.pgm - Speedup: 1.33
photo2_4_people.pgm - Speedup: 2.15
photo5_7_people.pgm - Speedup: 2.43
photo1_6_people.pgm - Speedup: 1.88
```

References

[1] F. Comaschi, S. Stuijk, T. Basten and H. Corporaal, "RASW: A run-time adaptive sliding window to improve Viola-Jones object detection," 2013 Seventh International Conference on Distributed Smart Cameras (ICDSC), Palm Springs, CA, USA, 2013, pp. 1-6, doi: 10.1109/ICDSC.2013.6778224.

[2] Yi-Qing Wang, An Analysis of the Viola-Jones Face Detection Algorithm, Image Processing On Line, 4 (2014), pp. 128–148.

Any ?s

Thank You!