GENERATING AN MTCNN MODEL AND INTEGRATING THE ALGORITHM WITH CUDA FOR FAST COMPUTATION

A PROJECT REPORT BY

Keshav Moorthy (16BCI0066)

Ashwath M (16BCE0701)

Sanjeev (16BCE0535)

M Viswanath (16BCE2250)

School of Computer Science and Engineering

CERTIFICATE

The project report entitled "GENERATING AN MTCNN MODEL AND INTEGRATING THE ALGORITHM WITH CUDA FOR FAST COMPUTATION" is prepared and submitted by Keshav Moorthy (16BCI0066), Ashwath M (16BCE0701), Sanjeev (16BCE0535) and M Viswanath (16BCE2250). It has been found satisfactory in terms of scope, quality and presentation as partial fulfilment of the requirements for the award of the degree of Bachelor of Technology in Computer Science and Engineering inVIT University, India.

Guide: Sairabanu J

Internal Examiner **External Examiner**

ACKNOWLEDGMENT

We would like to express our special thanks of gratitude to our ma'am Sairabanu J, who gave us the golden opportunity to do this wonderful project on the topic 'GENERATING AN MTCNN MODEL AND INTEGRATING THE ALGORITHM WITH CUDA FOR FAST COMPUTATION' and also gave her precious suggestions and constructive guidance in the completion of this project. This also helped us in going through diverse Research papers and useful websites and we came to learn about so many new things; we are really thankful to them.

Secondly we would also like to thank our friends who helped us a lot in our hard periods by giving their support and new ideas to finalize this project within the limited time frame.

INDEX

ABSTRACT

The Multi-task Cascaded Convolutional Networks (MTCNN) is a state of the art algorithm used extensively for face detection and alignment. In our project our aim is to build a simple MTCNN neural network on Tensorflow. CUDA is a parallel computing program ,that is supported by Tensorflow, created by Nvidia for general purpose computing on GPUs. Hence we would integrate our neural network with CUDA toolkit.

INTRODUCTION

With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs. In GPU-accelerated applications, the sequential part of the workload runs on the CPU – which is optimized for single-threaded performance – while the compute intensive portion of the application runs on thousands of GPU cores in parallel.

When using CUDA, developers program in popular languages such as C, C++, Fortran, Python and MATLAB and express parallelism through extensions in the form of a few basic keywords.
The CUDA Toolkit from NVIDIA provides everything you need to develop GPU-accelerated applications. The CUDA Toolkit includes GPU-accelerated libraries, a compiler, development tools and the CUDA runtime.

The proposed MTCNN consist of three stages. In the first stage, it produces candidate windows quickly through a shallow CNN. Then, it refines the windows by rejecting a large number of non-faces windows through a more complex CNN. Finally, it uses a more powerful CNN to refine the result again and output five facial landmarks positions. Thanks to this multitask learning framework; the performance of the algorithm can be notably improved.

Most of previous face detection and face alignment methods ignore the inherent correlation between these two tasks. Though several existing works attempt to jointly solve them, there are still limitations in these works.

Given an image, we initially resize it to different scales to build an image pyramid, which is the input of the following three-stage cascaded framework. We then aim to identify face regions with more supervision. In particular, the network will output five facial landmarks' positions.

OBJECTIVE

Face detection and alignment in unconstrained environment are challenging due to various poses, illuminations and occlusions. Recent studies show that deep learning approaches can achieve impressive performance on these two tasks. In this paper, we propose a deep cascaded multi-task framework which exploits the inherent correlation between them to boost up their performance. In particular, our framework adopts a cascaded structure with three stages of carefully designed deep convolutional networks that predict face and landmark location in a coarse-to-fine manner. In addition, in the learning process, we propose a new hard sample mining strategy that can improve the performance automatically without manual sample selection. Our method achieves superior accuracy over the state-of-the-art techniques on the challenging FDDB and WIDER FACE benchmark for face detection, and AFLW benchmark for face alignment, while keeps real time performance. And also saves, identifies and outputs the data for the recognized face.

PROJECT DESCRIPTION

PARALLEL PLATFORM OF INTEREST

CUDA

The parallel platform that we're working on is CUDA, which is a parallel computing platform and API (Application Programming Interface) Model created by NVIDIA. It allows the usage of CUDA-enabled GPU for general purpose processing, termed as the General-Purpose computing on Graphics Processing Units (GPGPU). The CUDA platform acts as a layer of software that gives direct access to the virtual instruction set of the GPU as well as its parallel computational elements of it, carried out within the computer kernel.

The CUDA platform is enabled to work with C, C++, Fortran, Python etc. This helps in using its parallel computational capabilities, like our project for example.

Graphics chip producers, for example, NVIDIA and AMD have been seeing a flood in offers of their graphics processors (GPUs) because of digital currency miners and machine learning applications that have discovered utilizations for these designs processors outside of gaming and recreations. Essentially, this is on the grounds that GPUs offer abilities for parallelism that are not found when all is said in done reason processors that happens to be a decent counterpart for activities such, for example, vast scale hashing and framework figurings which are the establishments of mining, and machine learning outstanding tasks at hand.

CUDA from NVIDIA gives an enormously parallel engineering to graphics processors that can be utilized for numerical calculation. While a common broadly useful Intel processor may have 4 or 8 centers, a NVIDIA GPU may have a great many CUDA centers and a pipeline that supports parallel handling on a large number of strings, accelerating the processing extensively. This can be utilized to extensively diminish processing time in machine learning applications, which builds the quantity of investigations and iterations that can be maintained while fine-tuning a model.

INSTALLATION

Pre-Requirements

a. The first thing to be sure of is that the compute capability is > 3.0
b. The availability of 64-bit python is a must as TensorFlow doesn’t work on the 32-bit version.
c. The system must be up-to-date
d. Verification of CUDA-Capable GPU

Actual Installation

a. Installing all the required dependencies
    i. sudo apt-get install build-essential
    ii. sudo apt-get install cmake git unzip zip
    iii. sudo add-apt-repository ppa:deadsnakes/ppa
    iv. sudo apt-get update
    v. sudo apt-get install python2.7-dev python3.5-dev python3.6-dev pylint
b. Installing a kernel header
c. Installing the actual CUDA 9.2
    vi. sudo apt-key adv --fetch-keys http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1710/x86_64/7fa2af80.pub
    vii. echo "deb https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1710/x86_64 /" | sudo tee /etc/apt/sources.list.d/cuda.list
d. Rebooting the system to load the driver
e. To finally set it up
    viii. echo 'export PATH=/usr/local/cuda-9.2/bin${PATH:+:${PATH}}' >> ~/.bashrc
    ix. echo 'export LD_LIBRARY_PATH=/usr/local/cuda-9.2/lib64${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}' >> ~/.bashrc
    x. source ~/.bashrc
    xi. sudo ldconfig
    xii. nvidia-smi

RUNNING A SAMPLE CODE

Running it using a sample using Anaconda Python Distribution to help manage the different environments. After installing anaconda accelerate by using

```
 $ conda install accelerate
```

Using a basic program that makes two vectors with 100 million passages that are arbitrarily created. A calculation is then performed with the end goal that every passage from one vector is raised to the intensity of the relating section in the other and put away in a third vector, which is returned as the aftereffects of the calculation. Programming this straightly, we would utilize a for loop to play out this computation and return back the appropriate response. We have included a touch of timing code here to perceive to what extent it is taken.

Running it on our system, this program takes about 35 seconds.

```
import numpy as np
from timeit import default_timer as timer

def pow(a, b, c):
  for i in range(a.size):
      c[i] = a[i] ** b[i]

def main():
  vec_size = 100000000

  a = b = np.array(np.random.sample(vec_size), dtype=np.float32)
  c = np.zeros(vec_size, dtype=np.float32)

  start = timer()
  pow(a, b, c)
  duration = timer() - start

  print(duration)

if __name__ == '__main__':
  main()
```

But, after modifying it like so:

```
import numpy as np
from timeit import default_timer as timer
from numba import vectorize

@vectorize(['float32(float32, float32)'], target='cuda')
def pow(a, b):
    return a ** b

def main():
    vec_size = 100000000

    a = b = np.array(np.random.sample(vec_size), dtype=np.float32)
    c = np.zeros(vec_size, dtype=np.float32)

    start = timer()
    c = pow(a, b)
    duration = timer() - start

    print(duration)

if __name__ == '__main__':
    main()
```

After the inclusion of parallelism with the slight modification, it takes only 3.6 seconds.

The vectorize decorator on the pow work deals with parallelizing and diminishing the capacity over numerous CUDA centers. It does this by incorporating Python into machine code on the principal invocation, and running it on the GPU. The vectorize decorator takes as info the mark of the capacity that will be quickened, alongside the objective for machine code age. For this situation, 'cuda' suggests that the machine code is produced for the GPU. It likewise bolsters targets 'cpu' for a solitary strung CPU, and 'parallel' for multi-center CPUs. Engineers can utilize these to parallelize applications even without a GPU on standard multi center processors to remove each ounce of execution and put the extra centers to great utilize. And the majority of this, without any progressions to the code.

We at that point convert the pow capacity to work on a scalar rather than the whole vector. This would be a nuclear task performed by one of the CUDA centers which Numba deals with parallelizing. The invocation of the function is changed to get the third vector as opposed to passing it in as a parameter.

ALGORITHM EXPLANATION

- The idea of the whole algorithm can be divided into multiple segments as:

- The neural system identifies singular appearances, finds facial milestones (i.e. two eyes, nose, and endpoints of the mouth), and draws a bounded box around the face.

- To start with, they import OpenCV (to open, read, compose, and demonstrate pictures) and MTCNN. Checking ./mtcnn/mtcnn.py demonstrated the MTCNN class, which played out the facial discovery.

- At that point, a finder of the MTCNN class was made, and the picture read in with cv2.imread. The detect_faces work inside the MTCNN class is called, to "identify faces" inside the picture we go in and yield the appearances in "result".

- The result is by all accounts a lexicon that incorporated the directions of the bounded box and facial milestones, and additionally the system's trust in arranging that region as a face.

- We would now be able to isolate the directions, going in the bounded box directions to bounding_box and the facial milestone directions to key points.

- Presently we draw the square shape of the jumping confine by passing the directions, the shading (RGB), and the thickness of the case layout. Here, bounding_box[1] and bounding_box[0] speak to the x and y directions of the upper left corner, and bounding_box[3] and bounding_box[2] speak to the width and the tallness of the case, individually.

- Additionally, we can draw the purposes of the facial tourist spots by going in their directions, the span of the circle, and the thickness of the line.

- At last, we make another record and demonstrate the picture

- This, however, doesn't help facial location on the off chance that we could just go in pictures. It needs to be altered so that it can likewise test the paper's case that it is ready to find faces continuously.

- Now we use OpenCV and MTCNN, and at that point made a finder:

- To utilize the webcam, we make a VideoCapture protest. Since it's just one camera, we go in 0.

- cap.read() restores a boolean (True/False) that states regardless of whether a casing is perused in accurately. On the off chance that a mistake happens and an edge isn't perused it, it will return False and the while circle will be broken.

- Furthermore, there might be in excess of one face in the casing. All things considered, result will return back various arrangements of directions, one for each face. Thus, we will run a for loop in result to repeat through each individual face. For each face, we will draw out the bounded box outline and spot the 5 facial tourist spots.

- Running this new document, we will hopefully be able to see that the MTCNN system can in reality keep running progressively. Boxing the face and checking out the highlights. On the off chance that the face is movedout of the edge, the webcam keeps running also.

AREAS OF PARALLELISM

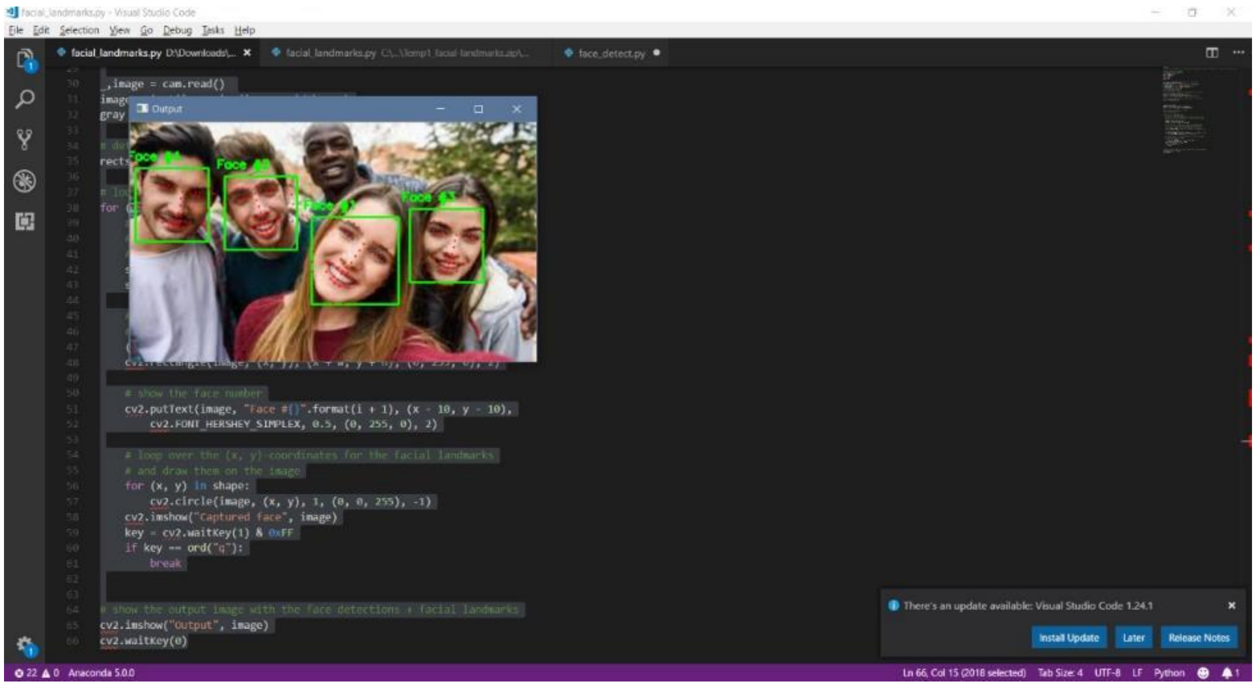
Since we are running it in a GPU, using TensorFlow, we would be forcing the entire code to run in parallel.

A contemporary most loved for parallelism in suitable circumstances, and focal point of this article, is using broadly useful figuring on illustrations handling units (GPGPU), a methodology abusing the various preparing centers found on top of the line present day designs preparing units (GPUs) for the concurrent execution of computationally costly errands. While not all machine learning assignments, or some other gathering of programming undertakings so far as that is concerned, can profit by GPGPU, there are without a doubt various computationally costly and time-cornering errands to which GPGPU could be a benefit. Adjusting calculations to enable sure of their errands to exploit GPU parallelization can exhibit essential gains in both undertaking execution and finishing speed.

A considerable measure of undertakings in (3D) designs are free from one another, so the plan to parallelize those assignments isn't new. Be that as it may, while parallel processors are these days exceptionally normal in each work area PC and even on more up to date cell phones, the manner in which GPUs parallelize there work is very extraordinary. Understanding the manners in which a GPU works can help understanding the execution bottlenecks and is critical to plan calculations that fit the GPU engineering.

In all actuality present day CPU centers are very mind boggling: they accelerate the program execution by reordering autonomous tasks on the fly (out-of-arrange execution), they endeavor to figure the result of a branch before it was assessed to keep the profound pipelines filled (branch forecast) and actualize parallel ideas in a solitary center: SIMD and synchronous multithreading. A GPU center is more straightforward with regards to execution rationale, they are all together and attempt less to be shrewd. GPUs likewise actualize the last two traps (SIMD, synchronous multithreading), even in more outrageous ways.

RESULTS AND DISCUSSION



```
_,image = cam.read()
image
gray

rects

for

    cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)

    # show the face number
    cv2.putText(image, "Face #{}".format(i + 1), (x - 10, y - 10),
        cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

    # loop over the (x, y)-coordinates for the facial landmarks
    # and draw them on the image
    for (x, y) in shape:
        cv2.circle(image, (x, y), 1, (0, 0, 255), -1)
    cv2.imshow("Captured face", image)
    key = cv2.waitKey(1) & 0xFF
    if key == ord("q"):
        break


# show the output image with the face detections + facial landmarks
cv2.imshow("Output", image)
cv2.waitKey(0)
```

facial_landmarks.py - Visual Studio Code
File Edit Selection View Go Debug Tasks Help
facial_landmarks.py D:\Downloads\...
facial_landmarks.py C:\...\lcmp1_facial-landmarks.zip\...
face_detect.py
Output
Face #4
Face #2
Face #1
Face #3
There's an update available: Visual Studio Code 1.24.1
Install Update
Later
Release Notes
22 0 Anaconda 5.0.0
Ln 66, Col 15 (2018 selected) Tab Size: 4 UTF-8 LF Python

The program saves the values of the face via scanning using the camera and adds landmarks of the face to a file which contains all the data of the face under a name. Once called for running, the function reads the face data and identifies the face by comparing and contrasting the values and the input name.

If identified, the program draws a box around the person’s face along with the name, else showing “Unknown”.

We’ve been able to calculate this to 100% accuracy at a particular line of sight with the camera and under favorable lighting. The accuracy drops by 26% or so when conditions are not favorable.

CONCLUSION

In this paper, we have presented an extensive review of recent research development on face recognition. We’ve also focused on face recognition systems, detection and localization, feature extraction, and recognition aspects of the face recognition problem. We herein present a comprehensive and critical survey of face detection algorithms.

Face detection is a necessary first-step in face recognition systems, with the purpose of localizing and extracting the face region from the background. In terms of image features, the extraction techniques are classified into four types: knowledge-based, mathematical transform based, neural networks or fuzzy theory based, and other. Feature extraction is an old problem in the field of pattern recognition; however, it has been the most fundamental and important problem. Whether feature extraction is effective is always the key to solving the problem or completing the task of image recognition. It is believed that better methods for extraction will be improved to represent intrinsic and more attributions of images, reduce the information redundancy, and increase simultaneously the entropy, which will merit further recognition for better results.

REFERENCES

[1] S. Abell, N. Do, J.J. LeeGPU-LMDDA: a Bit-Vector GPU-Based Deadlock Recognition Algorithm for Multi-Unit Resource Systems International Journal of Parallel, Emergent and Distributed Systems, 31 (6) (2016), pp. 562-590

[2] B.F. AlBdaiwi, H.M. AboElFotohA GPU-Based Genetic Algorithm for the P-Median Problem The Journal of Supercomputing, 73 (10) (2017), pp. 4221-4244

[3] D.R. AmancioA Complex Network Approach to Stylometry PLoS One, 10 (8) (2015), p. e0136076

[4] A.S. Arefin, C. Riveros, R. Berretta, P. MoscatoGPU-FS-kNN: A Software Tool for Fast and Scalable kNN Computation Using GPUs PloS ONE, 7 (8) (2012), pp. 1-13

[5] V.D. Blondel, J.-L. Guillaume, R. Lambiotte, E. LefebvreFast Unfolding of Communities in Large Networks Journal of Statistical Mechanics: Theory and Experiment, 2008 (10) (2008), p. P10008

[6] S. Busygin, O. Prokopyev, P.M. PardalosaBiclustering in Data Mining Computers and Operations Research, 35 (9) (2008), pp. 2964-2987

[7] J. Shi and J. Malik, "Normalized Cuts and Image Segmentation," Proc. IEEE CS Conf. Computer Vision and Pattern Recognition, pp. 731-737, 1997.

[8] P. Ekman and W. V. Friesen. Emotional facial action coding system. Unpublished manuscript. University of California at San Francisco, 1983.

[9] Kang Yuanyuan,Li Bin,Tian Lianfang ,Mao Zongyuan. Multi-modal Medical Image Fusion Based On Wavelet Transform And Texture Measure. Proceedings of the 26th Chinese Control Conference, pp.697-699,2007

[10] OpenCL conformant products. [Online]. Available: https://www.khronos.org/conformance/adopters/conformant-products#opencl