Exploring the Benefits of Heterogeneous Computing to Accelerate
Face Detection Deep Learning Inference

A Thesis Presented
by

Julian Gutierrez

to
The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Master of Science

in

Electrical and Computer Engineering

Northeastern University
Boston, Massachusetts

September 2017
To my family.

I smile because you’re my family, I laugh because there’s nothing you can do about it!
Contents

List of Figures iv
List of Tables v
List of Acronyms vi
Acknowledgments vii
Abstract of the Thesis viii

1 Introduction 1

2 Background 4
  2.1 Machine Learning ................................................................. 4
    2.1.1 Artificial Neural Networks .............................................. 4
    2.1.2 Caffe .............................................................................. 6
  2.2 Parallel Computing ................................................................. 6
    2.2.1 Graphics Processing Units ............................................... 7
    2.2.2 Parallel Computer Memory Model ..................................... 8
    2.2.3 Synchronization .............................................................. 9
  2.3 Related Work ...................................................................... 9

3 Methodology 11
  3.1 State-of-the-art Algorithms .................................................... 11
  3.2 Environmental Setup ............................................................. 13
  3.3 Initial Implementation ........................................................... 14

4 Framework 15
  4.1 Data Packet ....................................................................... 16
  4.2 Auxiliary Functions .............................................................. 19
    4.2.1 Ordered ....................................................................... 19
    4.2.2 Non-Maximum Suppression (NMS) ................................... 19
    4.2.3 Generate Bounding Box ............................................... 19
    4.2.4 Pad Bounding Box ....................................................... 20
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Face detection example using deep neural networks.</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Resolution impact on accuracy due to low resolution (a) compared to higher resolution (b).</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Hyper-Q benefit example</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>System built with GTX1080 and a Core i7 6700k.</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>Simple instruction pipeline with 5 stages.</td>
<td>15</td>
</tr>
<tr>
<td>4.2</td>
<td>General structure of the threaded application.</td>
<td>24</td>
</tr>
<tr>
<td>4.3</td>
<td>Example execution of a pipelined framework, considering the FPS limit</td>
<td>24</td>
</tr>
<tr>
<td>4.4</td>
<td>Execution example of pipeline framework over FPS limit</td>
<td>25</td>
</tr>
<tr>
<td>4.5</td>
<td>Example output displayed on the terminal by the application. Interaction with terminal is done with the keyboard using the characters indicated by the display.</td>
<td>32</td>
</tr>
<tr>
<td>5.1</td>
<td>Execution time for our pipelined implementation while processing a blank video collected at 1080p.</td>
<td>34</td>
</tr>
<tr>
<td>5.2</td>
<td>Execution time for our implementation, while processing a crowd video at 1080p.</td>
<td>36</td>
</tr>
<tr>
<td>5.3</td>
<td>Sample output when processing the crowd video.</td>
<td>37</td>
</tr>
<tr>
<td>5.4</td>
<td>Execution time for the pipeline while processing the crowd video with a queue size of one element, at 1080p.</td>
<td>38</td>
</tr>
<tr>
<td>5.5</td>
<td>Execution example of the pipeline framework utilized for the application when the execution time is larger than the FPS limit, for the video with a queue size of one element.</td>
<td>39</td>
</tr>
<tr>
<td>5.6</td>
<td>Execution time for the pipeline while processing a live video at 1080p.</td>
<td>40</td>
</tr>
<tr>
<td>5.7</td>
<td>Execution time for the pipeline while processing fireworks video at 4K resolution.</td>
<td>41</td>
</tr>
<tr>
<td>5.8</td>
<td>Summary results, comparing the performance of the video for the parallel version, parallel version running on one thread and the sequential version.</td>
<td>43</td>
</tr>
<tr>
<td>5.9</td>
<td>Sample outputs from the FDDB database for our application.</td>
<td>44</td>
</tr>
<tr>
<td>5.10</td>
<td>ROC curve for the discrete measurements on the FDDB database</td>
<td>45</td>
</tr>
<tr>
<td>5.11</td>
<td>ROC curve for the continuous measurements on the FDDB database</td>
<td>46</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Desktop Characteristics .................................................. 13

5.1 Characteristics of the videos for the sequential implementation. ........................................ 42
List of Acronyms

ANNS  Artificial Neural Networks
API  Application Program Interface
CAM  Camera
DCNN  Deep Convolutional Neural Networks
DTB  Database
FPS  Frames Per Second
GPU  Graphics Processing Unit
GPGPU  General Purpose Graphics Processing Unit
ILP  Instruction Level Parallelism
IOM  intersection over minimum
IOU  intersection over union
NMS  None-Maximum Supression
SIMD  Single Instruction, Multiple Data
VID  Video
Acknowledgments

I want to express my deepest gratitude to Dr. Kaeli for all his guidance and expertise on the development of this project. I also want to thank my incredible friends/family from Boston for all their support and who made this experience enjoyable and unforgettable. My family has always been invaluable to me and I can’t thank my sisters, my mom and my dad enough for all their tips and support. Finally, I wish to thank the University of Costa Rica and the Ministry of Science and Technology from Costa Rica for sponsoring my Master’s Degree.
Abstract of the Thesis

Exploring the Benefits of Heterogeneous Computing to Accelerate Face Detection Deep Learning Inference

by

Julian Gutierrez

Master of Science in Electrical and Computer Engineering
Northeastern University, September 2017

David Kaeli, Ph.D, Advisor

Significant improvements in face detection accuracy have been achieved by an emerging class of deep learning algorithms. Despite the capability of these algorithms to achieve high accuracy, deep learning approaches can be computationally prohibitive. As a result, we need to trade off high accuracy with processing throughput, meaning robust face detection in real-time for full HD video streams is not possible today.

To overcome this challenge, we propose a parallel pipelined framework that enables efficient usage of our heterogeneous platform. We implement this pipeline framework using a state-of-the-art algorithm, exploiting the CPU and GPU available resources through C++ libraries, including pthreads and OpenCV, and use Caffe and cuDNN libraries to implement our deep learning models.

Our framework is capable of handling full HD video workloads in real-time, assuming typical video scenarios. We achieve a 2.4x faster frame-rate as compared to a sequential implementation that is GPU enabled. We are also capable of achieving up to 110 FPS for a standard definition video, while still retaining the high accuracy of the original algorithm. The resulting pipelined framework has a high degree of flexibility, enabling us to consider a range of deep learning algorithms as we try to map deep neural networks to a powerful CPU-GPU platform.
Chapter 1

Introduction

Face and facial landmark feature detection on images is an indispensable component required for accurate face recognition [1]. Facial landmark detection is fundamental in a wide range of face analysis tasks [2], ranging from face verification (recognition), to face morphing and overlapping masks on faces. With the increased use of face recognition on full-featured mobile devices, these algorithms are becoming even more essential. Popular applications, such as Facebook, use these algorithms to identify faces in images. Snapchat uses detection to add or modify faces in real-time. These are just a few applications that are leveraging accurate face recognition algorithms.

Figure 1.1: Face Detection example using deep neural networks (Source [3])

Deep learning algorithms have the ability to interpret a scene or image given a corpus of examples to learn from. Through this training process, learning algorithms are able to identify complex relationships between data-points in order to produce accurate identification engines. Face
CHAPTER 1. INTRODUCTION

detection is one of the most popular problems tackled by machine learning algorithms, as they are capable of identifying patterns in how faces are portrayed on images, as shown in Figure 1.1. Due to the emergence of online surveillance systems, real-time face detection and recognition is now require. While the training process can be slow, the online recognition algorithm needs to be high performance.

Fast face detection has been achieved since the 1980’s. The issue has been that these early implementations were only able of identifying faces under very controlled situations: correct lighting, frontal image, etc. While the detection problem in ideal settings can be done in real-time, it is rare to find situations where we can assume ideal conditions. More recently, robust face detection has been demonstrated through the use of neural networks [4]. This paper focused on detection of highly variable face patterns. These algorithms are capable of identifying faces under almost any kind of scenario, from poor lighting, to occlusions, to different poses and expressions [5].

The main drawback associated with adopting neural networks is that they require a significant amount of computational power. Through the use of GPU’s, these algorithms are now capable of processing the data at a much faster pace, allowing us to perform face detection/recognition in real-time.

In most videos today, the frame rate is 30 frames per second (FPS), which says a detection algorithm needs to complete the inference task in under 33ms. A further complication associated with this challenge arises when we start considering the resolution of the image. Neural networks do not scale linearly in terms of image resolution. Working with a higher resolution image, the computational requirements grow significantly [6]. These algorithms are not capable of performing inference under this hard limit, requiring alternative schemes to achieve real-time performance.

The improved resolution of today’s cameras allows us to capture richer information in a single image, enabling vision systems to identify individuals more accurately, as shown on Figure 1.2. But the challenge is that given this increase in data, the processing time of the neural network increase significantly. Despite the impressive computational power available in a GPU, increasing image resolution has a significant cost in terms of speed. As a result, we may not be able to leverage the best detection algorithm when real-time processing on full HD resolution videos is needed.
CHAPTER 1. INTRODUCTION

Figure 1.2: Resolution impact on accuracy due to low resolution (a) compared to higher resolution (b).

The objective of this thesis is to deliver an application that can provide robust face and landmark detection, and leverages both GPU and CPU architectures to reduce inference time and enable processing of a full HD video in real-time. This is only possible by introducing a pipelined framework to parallelize the processing of each frame. By separating the algorithm into clearly defined stages, we are capable of running these stages in parallel to process different frames at the same time.

This thesis is organized as follows. Chapter 2 presents background information on machine learning and provides a general introduction to parallel computing and related work. Chapter 3 describes the algorithms that are used in the thesis. Chapter 4 discusses the implementation of the framework. Chapter 5 reports face detection results from our experiments. In Chapter 6, we conclude the thesis by first summarizing our work, and then identifying directions for future work.
2.1 Machine Learning

Machine learning is an important research area where the main goal is to develop algorithms that are capable of processing information, “without being explicitly programmed” \cite{7} to do so. These algorithms are capable of learning how to process the information through a training routine, which involves knowing what the correct solution is to be able to teach the program the correct answer.

Recently, this class of algorithms have increased in popularity thanks to their great flexibility for solving problems. These algorithms are commonly used in image processing, and have attracted significant attention in their ability to perform object detection, object tracking, image manipulation and interpretation. There are many types of machine learning algorithms, from decision tree learning, artificial neural networks, deep learning, support vector machines, clustering, bayesian networks, genetic algorithms, and many others \cite{8}. In this thesis we focus on Artificial Neural Networks (ANNs).

2.1.1 Artificial Neural Networks

Artificial Neural Neworks are a sub-area of machine learning. This class of algorithms tries to mimic the behavior of biological neural networks in the brain in order to process information. Similar to many other machine learning algorithms, an ANN has to go through a process of training in order to understand the patterns in the data. These systems can learn to do tasks by considering prior examples. For example, identifying whether an image contains a cat, a dog or a mouse, using the
CHAPTER 2. BACKGROUND

results from training, we can then use this trained system to identify these animals in other images.

These systems are composed of a large number of highly interconnected processing elements known as neurons [9]. An ANN is typically configured for a specific application, where the structure of the model used is designed to be able to tackle the problem in an efficient manner. The learning process mimics the synaptic adjustments [9] that occur in our brains when we learn. Learning is facilitated through a backward propagation mechanism that propagates the differences between the correct result and the output of the algorithm in order to modify the values of the connections between neurons, with the goal of interatively improving our prediction results.

Some advantages of ANNs include:

- Robustness and Accuracy: As mentioned before, they are capable of providing better results than most rule-based algorithms.

- Adaptive learning: They have the ability to learn how to perform tasks based on training data [9].

- Self-Organization: An ANN can create its own organization or representation of the information it receives during the learning process [9].

- Real-Time Operation: thanks to current hardware resources, ANN computations can be carried out in parallel, with many hardware vendors building custom solutions specifically for accelerating these algorithms.

A popular application for a neural network is to perform face detection/recognition. There are multiple ways of approaching face and facial landmark detection in a rule-based system, but recent studies have shown that deep learning approaches with neural networks can achieve impressive performance in terms of accuracy and robustness [10]. Robustness is a major requirement for facial algorithms to produce the correct output. Even if a landmark point is not visible, it is typical for a facial landmark detection algorithms to guess a position for the point [11]. This is important if we want to detect the rotation of the face. Providing robustness, while also satisfying performance guarantees, poses some great challenges for the research community.

Multiple frameworks have been proposed to provide a simple interface for the development of neural networks. Frameworks such as TensorFlow [11], Theano [12], Keras [13] and Caffe [14] provide a set of APIs for developers to effectively train and test their models under multiple hardware configurations. Caffe is one of the most popular frameworks in the deep learning field for computer vision, and serves as the main tool in this thesis for modeling neural networks.
CHAPTER 2. BACKGROUND

2.1.2 Caffe

Caffe is a deep learning framework developed by the Berkeley Vision and Learning Center (BVLC) [14]. This framework allows us to use either CPUs or GPUs to develop a model [14]. Due to its recent popularity, Caffe’s community has increased significantly, and is used in a wide range of academic research projects in vision, speech and multimedia. Caffe contains an interface for Matlab, Python, Fortran and C++. Most research with Caffe has leveraged the Python and Matlab interfaces.

In this thesis, we explore how to apply deep neural networks to perform face detection. Neural network applications need to leverage algorithms that can execute efficiently. A major limitation of Convolutional Neural Networks (CNNs) is that they require a huge amount of computation to train their networks. CNNs can be slow, as discussed by Sun et al. [15]. Using Caffe [14], we can achieve high performance throughput for neural networks, especially when using Caffe with GPUs for model training and inference. Shi et al. [16] determined there is a significant performance improvement on training and testing of ANNs when using GPUs versus CPU only. Shi et al. [16] also demonstrated Caffe performed similarly to other popular frameworks when using GPU hardware.

2.2 Parallel Computing

In today’s world, technology has been advancing rapidly, especially in the computing technology markets. The latest generation of CPUs and GPUs offer impressive performance capabilities, while also maintaining power efficiency. The design of CPUs has shifted from a focus on single-threaded performance to multi-core CPUs. CMOS technology hit the power wall such that increasing the frequency became unsustainable. Designers started adding cores into their designs, and a new realm of computing began.

Parallel computing is an approach to leverage multiple computational resources concurrently [17]. The operating system needs to be designed appropriately to support efficient usage of these resources. When processors have multiple cores, a well-designed application is capable of using all resources efficiently. This class of applications are usually broken into discrete parts that can be solved concurrently.

CPUs are now designed with concurrency at multiple layers, from the bit level to the task level. Today’s CPUs are highly parallel, allowing programmers to make use of computational and memory resources more efficiently [17]. On the other hand, GPUs are designed to expose a high thread-level parallelism for high-throughput applications. Flynn’s taxonomy [18] describes
CHAPTER 2. BACKGROUND

these parallel systems as SIMD (Single Instruction, Multiple Data). GPUs are capable of executing thousands of threads concurrently, all executing the same instructions over multiple data streams. Heterogeneous systems utilize these characteristics from both CPU and GPU resources to enable a more efficient execution of applications.

2.2.1 Graphics Processing Units

GPUs were originally designed for processing vertexes and polygons, rendering 3D objects for video games, and display information on screens in near real-time. It was not until the introduction of programmable shaders and high level languages, the GPUs became a major accelerator for a wide class of computations. In this new era of GPU computing, demanding applications can leverage the massive parallelism available on a GPU. Given the thousands cores present on a GPU, they are able to accelerate data-parallel applications, while also achieving lower power budgets (as compared to a CPU). GPUs have evolved rapidly in the general purpose market, and are used extensively by the scientific community to accelerate computations [19].

NVIDIA introduced CUDA as the programming standard for their GPUs. The CUDA runtime enables us to execute programs developed in high-level languages, including C, C++, Fortran, Matlab, OpenCL, and other languages [20]. The key idea in the CUDA model is that serial programs that are loop-oriented can be easily expressed to run in parallel on a GPU. The key idea is to identify a thread that can be replicated to applied to a large amount of data. Linear algebra applications are well designed to leverage this straightforward data-parallel execution model.

While many applications can keep an entire GPU occupied with work, sometimes an application only uses a portion of the GPU. In this case, we would like to run two (or more) different kernels at the same time. NVIDIA introduced Hyper-Q concurrency with the Kepler architecture 3.5. Hyper-Q enables a GPU to launch/run multiple kernels asynchronously, enabling multiple CPU processes or threads to launch work on the same GPU, increasing GPU concurrency and utilization. This is achievable by assigning independent GPU operations to different CUDA streams. A CUDA stream is a sequence of operations that execute in order on the GPU [21]. Figure 2.1 shows how Hyper-Q assigns different work queues to each stream, allowing concurrent execution. Previous generation of GPUs only had one work queue, so if 2 kernels associated with the same stream were added to the queue in succession, this would create false-serialization of independent kernels [22].
CHAPTER 2. BACKGROUND

2.2.2 Parallel Computer Memory Model

One of the key components of any computer system design is the memory model. There are two main ways of implementing a memory system for a multi-processor chip, using either a shared memory or distributed memory. Shared memory enables all processors to access all of memory, treating it as a global address space [17]. Multiple processors can execute independently, but they share the same memory resources. One of the main advantages of this approach is that data shared between tasks is fast and uniform due to the proximity of the memory to the CPUs [17]. The other advantage is that the coherency of any data address is managed by the hardware, and does not require any programming effort.

Distributed memory logically (and generally physically) divides up memory into partitions, and relies on explicit communication to share data across memory partitions [17]. Each processor has its own memory with a separate memory address space. If a processor has to share a value, it has to explicitly send the data through an interconnect network. The advantage of this approach is if sharing is not needed often, performance scales nicely. If you add more processors to a shared memory model, there will be contention for resources, while a distributed model is more capable on handling scaling, up to a certain limit.

In a heterogeneous system, the GPU and the CPU commonly have separate memory systems. In order to execute work on the GPU, the CPU has to explicitly copy data from the CPU memory to the GPU memory, and then back once the GPU finished processing it. NVIDIA’s CUDA API enables this communication through simple function calls. Caffe’s interface builds upon these
functions, and masks these copies in their APIs, so programmers do not need to worry about these operations.

### 2.2.3 Synchronization

When working on a distributed system with distributed memory, communication performance is key when the degree of data sharing is high. Communication performance in a shared memory system is also important, though it is handled by the underlying hardware. When running multiple tasks or processes that share a common memory, CPU cores are required to synchronize shared data to make sure each core is looking at the correct data at any point in time [17].

Synchronization is key to make sure that communication is carried out correctly. In order to support synchronization, a number of Application Programming Interfaces (APIs) have been developed, such as Pthreads [23]. Synchronization can be implemented using mutexes, conditional variables, explicit synchronization functions, and many more.

The Pthreads library and POSIX threads, are a popular thread-based programming abstraction that has become a standard [23]. The Pthreads model provides a set of APIs to control multiple tasks that are assigned to threads and executed in parallel. Most Unix operating systems support Pthreads, including FreeBSD, Linux and Android.

### 2.3 Related Work

There has been a significant body of prior work exploring deep learning approaches for face detection [15] using SVMs [24, 25], Regression Trees [26], and Deep Convolutional Neural Networks (CNNs) [1, 2, 10, 15], to mention a few. Given the large visual variations of faces, including occlusions, large pose variations and extreme lighting conditions, accurate face recognition includes a number of challenges [10]. Face detection algorithm have recently been attracting a lot of interest from industry and the research community [1].

CNNs have been found to perform effectively for identifying faces, but require a significant amount of training and computation to obtain accurate results. With the increased use of GPUs, new CUDA implementations of CNNs have been developed [27]. APIs, such as CuDNN and Caffe [14], have become key in order to provide an effective path to efficient face detection.

There have been prior work that looks to accelerate CNNs. Fast R-CNN [28] and Faster R-CNN [29] restructure models for faster and better performance during inference, as well as during
training. While the focus of this thesis build on this prior work, we focus on tuning an implementation that maximizes the hardware resource usage. As an example, Fast R-CNN was implemented in C++ and uses Caffe, but did not consider how to tune execution on the CPU.

YOLO [30] and YOLOv2 [31] are popular approaches for performing object detection, similar to the face detection problem. The main goal of their approach is to only look at the image once (YOLO) in order to predict the locations of the objects on the image; one single convolutional network performs all calculations. This approach has proven to be the fastest in terms of model performance because there is no need for extra processing between stages. The problem is that given its unique model, scaling becomes an issue, and at higher resolutions, the model is unable to satisfy real-time performance, thus, there is no way to improve the performance without affecting the accuracy by making the model simpler.

In this thesis, we build on the work of Zhang et al. [10] to reimplement their robust algorithm. Their approach is divided into three stages. The first stage contains a DCNN that detects possible candidates for faces on the image by applying the model at different scales of the image, starting at 0.6 (matching the convolution filter size of 12x12 to a 20x20 bounding box after scaling). It continues to scale the image using a natural scaling factor until it reaches a scaled image with the size of the image smaller than the convolution filter. The second stage fine-tunes the detection positions of a bounding box through applying bounding-box regression and incorporating a more detailed scoring. And the final stage applies a final bounding box regression, and includes the calculation of landmark feature locations. Despite their good performance compared to other algorithms, this approach is unable to handle real-time inference on a full HD video stream. Creating a pipeline framework implemented in C++ with Caffe and Pthreads library will allow us to satisfy this objective.
Chapter 3

Methodology

Designing a framework to implement a real-time inference face detection requires intimate familiarity with state-of-the-art algorithms. In this chapter we discuss the key factor required by any face detection algorithm in order to provide robust and expedient detection. We discuss our targeted execution environment, and the algorithm selected that is used in this thesis.

3.1 State-of-the-art Algorithms

To select the right machine learning algorithm for real-time face detection, we start by identifying the variables that will be most critical in this challenging problem. We summarize these qualities as follows:

- robustness and accuracy of a detection are fundamental - robustness can be measured in many ways, including sensitivity to noise, pose, occlusions, etc.,
- scalable (in terms of accuracy),
- open source,
- adaptability to work with GPU API libraries, and
- pipelined in stages, facilitating performance optimization.

First and foremost, we need any algorithm to be able to identify faces and landmarks under different circumstances, including occlusions, side shots, low and high resolution, etc. Most deep
CHAPTER 3. METHODOLOGY

Learning algorithms are capable of obtaining excellent precision when running benchmark datasets such as FDDB [32] and Wider Face [33].

Second, we require an algorithm to be scalable, ensuring that accuracy is maintained when we modify the image resolution. Algorithms that process different scales of an image are ideal, as we can apply the same concepts when using higher resolution images. We also require the code is open source so that others can replicate our results.

Third, this implementation should leverage well established deep learning APIs such as Caffe, TensorFlow, Theano, etc., which are capable of offloading work to a GPU. The goal is to greatly reduce the computation time for the deep learning models drastically using state-of-the-art hardware (i.e., a GPU).

Fourth, in order to create a parallel pipeline, we would like to use a deep learning algorithm that is divided into multiple stages. Deep learning algorithms can be generalized into two classes. The first class of algorithms are implemented with a single deep learning model and the second type with multiple models as sequential stages [34].

Using a single model that is capable of interpreting the input data and producing the final output is a common approach for designing deep learning models [34, 30, 35]. These models can be very complex, but still provide advantages. Post processing (e.g., bounding box regression or non-maximum suppression) can be integrated, and can be much faster for smaller or low resolution images [30]. The downside for this approach is they do not scale well with increased input data size, causing significant delays in processing when using high resolution images.

Using multiple models as sequential stages is a less common approach for deep learning model designs, though leads to a more portable design, since these models tend to be simpler [10]. Some benefits to this approach include changing individual stages with a better solution, and enables us to run each stage in parallel. The disadvantages of this approach include the overhead of sending data between stages, and high variations in execution time of each stage. Two images that are the same size, but contain different levels of detail, can produce significantly different execution times. Given the tradeoffs, we have adopted a multi-stage pipeline approach, given portability and adaptability of the resulting neural network model. To provide for real-time face detection and landmark detection utilizing a neural network, we selected the algorithm proposed by Zhang et al. [10].
CHAPTER 3. METHODOLOGY

3.2 Environmental Setup

To provide us with an appropriate computing platform, we selected a desktop system equipped with a powerful GPU in order to enable us to achieve near real-time performance. Table 3.1 describes the characteristics of this machine. We exploit the acceleration provided by the NVIDIA Pascal Architecture. This GPU consists of 2560 CUDA cores and has 8 GB of GDDR5X RAM memory, enabling high computational power for accelerating neural network models. The Core i7 CPU has 4 cores with 8 logical cores through hyper-threading, with 16 GB of DDR3 RAM memory overclocked to 2.7 GHz.

![System built with GTX1080 and a Core i7 6700k.](image)

Table 3.1: Desktop Characteristics

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>6700k OC @ 4.5 GHz</td>
</tr>
<tr>
<td>GPU</td>
<td>GTX1080</td>
</tr>
<tr>
<td>RAM</td>
<td>16 GB OC @ 2.7 GHz</td>
</tr>
<tr>
<td>SSD</td>
<td>240 GB</td>
</tr>
</tbody>
</table>

The operating system is Ubuntu 16.04.2 LTS 64-bit version, with the Linux 4.8.0-54-generic kernel. The following list describes the libraries involved in the development of our neural
CHAPTER 3. METHODOLOGY

network model for real-time face detection.

- GCC compiler version 5.4.0 with pthread library included by default.
- CUDA release V8.0.61 with CUDNN enabled for better performance of the Caffe models.
- OpenCV version 3.2.1 with CUDA header support enabled.
- Caffe 1.0 version with CUDA and CUDNN support enabled, as well as Matcaffe for the Matlab interface.
- Additional libraries where installed including Ncurses for the terminal manipulation, and other Caffe dependencies.

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library \[36\]. OpenCV provides a rich set of library functions for computer vision applications. The library supports multiple languages, including C, C++, Java and Python, and is supported on a variety of operating systems including Linux, MacOS, Android, and Windows \[37\]. The library has more than 2500 optimized algorithms \[37\], ranging from common image manipulation to state-of-the-art computer vision algorithms.

To correctly translate the original implementation in Matlab by Zhang et al.\[10\] to C++, we used OpenCV as the API to replace the Matlab Image processing library \[38\]. Caffe was also installed with the Matlab interface to test the original algorithm. We need to made changes to the original OpenCV source code and the makefile in order to resolve library dependencies.

3.3 Initial Implementation

After the initial testing Zhang et al.’s Matlab code \[10\], we proceeded to convert the Matlab code into C++, where we would use the C++ Caffe interface instead of the Matcaffe interface. A sequential version was constructed without any sort of parallelization in mind, except for the use of a GPU through the Caffe API. The C++ Caffe API is considered low level, which in turn makes any modification to the model a bit more complex, requiring manual pointer manipulation. Once these modifications were complete, we introduced explicit threading with the Pthread library. We describe our parallelization approach in the next chapter.
Chapter 4

Framework

Next, we discuss the implementation of our parallelized framework for implementing a Caffe neural network model. Our implementation is written in C++, and combines additional API functions from Caffe, OpenCV, Neurses, and Pthreads. Our application uses the prior models by Zhang et al. [10], available in their github repository. This repository includes trained models, enabling us to avoid retraining the models. The Caffe models have been trained using three different methods: Face/non-face classification, bounding box regression, and facial landmark localization.

![Figure 4.1: Simplified instruction pipeline with 5 stages. When maximum instruction overlap is achieved, it is capable of completing one instruction per clock cycle.](image)

<table>
<thead>
<tr>
<th>Instruction Number</th>
<th>Pipeline Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>2</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>3</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>4</td>
<td>IF ID EX MEM</td>
</tr>
<tr>
<td>5</td>
<td>IF ID EX</td>
</tr>
<tr>
<td>Clock Cycle</td>
<td>1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

We use the concept of pipelining to drive the architecture for our framework. A CPU pipeline breaks the task of executing an instruction into steps or stages. This allows the system to execute instructions executing in different stages of the pipeline concurrently. Figure 4.1 shows an example of a simple instruction pipeline with 5 stages, resembling the execution of a assembly line [17]. Based on this idea, we divided our neural network algorithm into 7 stages:

1. Main Stage: in charge of reading user input and handling input files/cameras,
CHAPTER 4. FRAMEWORK

2. Preprocessing Stage: in charge of processing the input frame to feed the correct data into the models,

3. PNET Stage: first stage of the algorithm by Zhang et al. [10],

4. RNET Stage: second stage of the same algorithm,

5. ONET Stage: third stage of the same algorithm,

6. Post Processing Stage: in charge of converting the output from the neural network models into the correct output, and

7. Output Stage: in charge of handling the display of the images and handling output files.

In this Chapter we discuss the data structures used to pass the data between stages, the functions that are used to process the output from the deep learning models, the main queuing mechanism that allows proper synchronization between stage. We also provide a more detailed explanation of each stage, as well as how the user interfaces with the framework.

4.1 Data Packet

The data packet is the essential object of the framework. It contains all the necessary information for the processed frame. The definitions used to wrap the data for each frame are defined in Program 4.1. `Package_type` is used to define which type of input is being processed. STU is the type of package used to synchronize the master thread with all the remaining threads during start up. IMG, VID and CAM are used to indicate what type of package is being processed, an image, video or a video from a camera, respectively. DTB denotes the input will be multiple images with different sizes. END is used to indicate that processing can end, and all threads can exit. The BBox structure is used to define the information necessary to store the bounding boxes, including the score and the bounding box regression adjustments. The landmark structure is used to define the 5 landmarks that will be analyzed:

- LE: Left Eye
- RE: Right Eye
- N: Nose
- LM: Left Mouth
- RM: Right Mouth
CHAPTER 4. FRAMEWORK

```c
enum package_type { STU, IMG, VID, CAM, DTB, END, ILL }; 

typedef struct { 
   // Bounding Box
   cv::Point2f p1;
   cv::Point2f p2;

   // Score
   float score;

   // Bounding Box Regression adjustment
   cv::Point2f dP1;
   cv::Point2f dP2;
} BBox;

typedef struct { 
   cv::Point2f LE;
   cv::Point2f RE;
   cv::Point2f N;
   cv::Point2f LM;
   cv::Point2f RM;
} Landmark;
```

Program Listing 4.1: Definitions of structures used for the data representation.

Program 4.2 describes the data object definition. The package type is defined as described previously. Two OpenCV Mats are used to store the original frame and the processed frame. A vector of "BBoxs" are used to store the bounding boxes and a vector of "landmarks" to store the landmarks for those bounding boxes. Synchronization variables are included, since they are used for scenarios where threads need to update the data atomically (the PNET stage, as described in subsection 4.5.3). Timing variables are used to keep track of the execution time for each stage, and the total execution time for each frame. We also have 3 functions that are used to control synchronization variables:

1. "WaitForCounter" uses a conditional variable to wait until the counter has reached the value passed to the function,

2. "IncreaseCounter" increases the counter by 1, (to indicate the master thread that one of the children has completed their update), and

17
CHAPTER 4. FRAMEWORK

3. "ResetCounter" will reset the counter to 0.

```cpp
class Data {
public:
    // File name
    std::string name;

    // Type indicator
    package_type type;

    // Actual Data
    cv::Mat frame;
    cv::Mat processed_frame;

    std::vector<BBox> bounding_boxes;
    std::vector<Landmark> landmarks;

    // Synchronization Variables
    int counter;
    pthread_mutex_t *mut;
    pthread_cond_t *done;

    // Timing variables
    double start_time;
    double end_time;
    double stage_time [STAGE_COUNT+1]; // PreP−PNET−RNET−ONET−PostP−Out−Main

    // Functions
    void WaitForCounter(int num);
    void IncreaseCounter(void);
    void ResetCounter(void);

    Data();
    ~Data();
};
```

Program Listing 4.2: Data Class definition.
CHAPTER 4. FRAMEWORK

4.2 Auxiliary Functions

Many auxiliary functions are used across multiple stages of the framework. In this section, we describe each of these functions in detail.

4.2.1 Ordered

std::vector<int> ordered(std::vector<float> values);

Program Listing 4.3: Ordered function declaration.

This function receives a vector of float values and outputs a vector of ints of the same size, with the indices of the values in the original vector ordered, with the highest value first.

4.2.2 Non-Maximum Suppression (NMS)

std::vector<int> nms(std::vector<BBox> total_boxes,
                      float threshold,
                      bool type);

Program Listing 4.4: NMS function declaration.

The NMS function receives a vector of bounding boxes, a threshold and a type. NMS then returns a vector of ints, specifying the indices of the chosen bounding boxes. It uses the ordered function to find the window of the maximum score, and removes all bounding boxes that intersect, based on an overlap threshold. Type is used to identify which type of decision to use, IOU, or intersection over minimum (IOM).

4.2.3 Generate Bounding Box

std::vector<BBox> generateBoundingBox(
                      std::vector<std::vector<float>> data,
                      std::vector<int> shape_map,
                      float scale,
                      float threshold);

Program Listing 4.5: Generate bounding box function declaration.
CHAPTER 4. FRAMEWORK

This function is specific for the PNET model object, which grabs the output from this model and creates the bounding boxes using the bounding box regressor output and the index for each box with a score higher than a given threshold.

4.2.4 Pad Bounding Box

```c
void padBoundingBox(std::vector<BBox> &boundingBoxes, int imgHeight, int imgWidth);
```

Program Listing 4.6: Pad bounding box function declaration.

This function is used to make sure bounding boxes are inside the limits of the image (errors with OpenCV libraries). Any box with an edge outside the image will be moved inside.

4.2.5 Write Output Image

```c
void writeOutputImage(Data* Packet);
```

Program Listing 4.7: Write output image function declaration.

This function writes the bounding boxes and landmarks that were detected into the original image, to be displayed on the screen.

4.2.6 Timing Functions

```c
double CLOCK();
void avginit();
double avgfps(double _avgfps);
double avgdur(double newdur, double _avgdur);
```

Program Listing 4.8: Additional timing functions declaration.

These functions are used to calculate the execution time of the different stages, as well as to calculate the average total duration and approximate average FPS of the video.
CHAPTER 4. FRAMEWORK

4.3 Neural Network Wrappers

To simplify the interaction with the Caffe models, we implemented an API that wraps the models in order to send and retrieve data with simple function calls. Caffe’s C++ API is fairly low level and requires multiple function calls to send input data to the models or to retrieve output data. By creating this higher-level API, we reduce the complexity of our program. To implement these wrappers, we created a base layer called BNET described in the next subsection.

4.3.1 BNET

```cpp
class BNet {
public:
    BNet(const string& model_file,
         const string& trained_file);

    std::shared_ptr<Net<float>> GetNet(void);
    void SetInputGeometry (cv::Size input);
    void FeedInput (std::vector<cv::Mat>& imgs);
    void Forward (void);

private:
    void WrapInputLayer (std::vector<cv::Mat>* input_channels);
    void PreProcess (std::vector<cv::Mat>* input_channels,
                     std::vector<cv::Mat>* imgs);

    std::shared_ptr<Net<float>> net;
    cv::Size input_geometry;
};
```

Program Listing 4.9: Base model class declaration.

This base model class is the main interface between Caffe and our application. Given that all 3 models in our application adopt the same interface to feed and forward data into the models, this class is in-charge of defining these functions.

- Public Functions

  - GetNet: Returns pointer to the Caffe object.
CHAPTER 4. FRAMEWORK

- SetInputGeometry: Sets input geometry of the model which varies depending on the size of the image, the number of boxes, etc.
- FeedInput: Correctly writes the data into the CPU input data pointer of the Caffe object.
- Forward: Call forward function from Caffe object.

- Private Functions
  - WrapInputLayer: Used by FeedInput to allocate the memory needed at the pointer of the Caffe object.
  - PreProcess: Used by FeedInput to save input data into the allocated memory for the input of the Caffe object.

4.3.2 PNET, RNET, and ONET

```
class ONet : public BNet{
  public:
    ONet(const string& model_file,
          const string& trained_file) : BNet(model_file, trained_file){}

    void RetrieveOutput( std::vector<int>& shape,
                        std::vector<std::vector<float>>& data);
};
```

Program Listing 4.10: ONET inheritance declaration.

Each model processes the output data differently. To provide more flexibility and reuse, we define a base class and use inheritance to specify the exact functions needed to retrieve the output for each model. The previous code reflects the same behavior for the 3 models.

- Public Functions
  - RetrieveOutput: each model has its own implementation since each output varies according to the model. Receives a reference to the shape of the output and a reference to the data where the data will be accommodated based on the neural network output layers.
4.4 Queueing Mechanism

The queuing mechanism is used to manage coherency in the framework and enforcing synchronization between stages. As describe in Program 4.11, the class is defined as a template to enable us to flexibly extends its use for a range of data types. Remove and Insert functions use the mutex and conditional variables for synchronization. If a thread tries to remove an object from the queue and the queue is empty, this thread will suspend until a new object has been inserted into the queue. Each stage in the pipeline has an input queue and an output queue, except for the main function (the initial stage has no input queue) and the output stage (the last stage has no output queue). This implementation supports a classical "producer-consumer" relationship between threads.

```cpp
template <class Element>
class Queue {
  public:
    Queue (void);
    ~Queue (void);
    void Insert (Element in);
    Element Remove (void);
    Element Snoop (void);

  private:
    void Add (Element in);
    Element Del (void);
    Element buf[QUEUESIZE];
    long head;
    long tail;
    int fullFlag;
    int emptyFlag;
    pthread_mutex_t *mut;
    pthread_cond_t *full;
    pthread_cond_t *empty;
};
```

Program Listing 4.11: Queue template declaration.
CHAPTER 4. FRAMEWORK

4.5 Pipeline Structure

As previously discussed, to enable full parallelization of our application, we adopt a pipeline structure. A pipeline allows us to process multiple frames at the same time. Figure 4.2 shows the structure of our threaded application, and shows all the stages and queues in the framework. An initialization mechanism is called to allow the application enough time to read the Caffe models into the program and setup all the threads before starting.

![Figure 4.2: General structure of the threaded application.](image)

Figure 4.2: General structure of the threaded application.

Figure 4.3 shows an example of how our framework executes on a video, where all stages execute under the frames per second (FPS) limit. The FPS limit given by a video is equal to 1000/FPS (in ms.). For example, a 30 FPS video sets the maximum execution time for a stage to 33 ms. If all stages take less time then the FPS limit, the application can run successfully without delays.

![Figure 4.3: The example application when the stage execution time is smaller than the FPS limit, based on the pipelined algorithm from [10].](image)

Figure 4.3: The example application when the stage execution time is smaller than the FPS limit, based on the pipelined algorithm from [10].

If one of the stages is slightly over the FPS limit, this will impact the FPS of the entire application. For example, if a stage takes 50 ms, the maximum FPS possible for the application is 20 FPS. Figure 4.4 shows an example of when this occurs in the third stage of the computing pipeline.
CHAPTER 4. FRAMEWORK

This is why it is important that our application is capable of handling all the stages under this limit. On the other hand, the total execution time can be over this limit, without impacting the FPS rate. The human eye is incapable of identifying delays under 100 ms, which means, the total execution time should ideally remain under this limit to be perceived as real-time.

![Timeline](image)

Figure 4.4: Execution example of the pipeline framework utilized for the application with a FPS rate violation.

Each stage executes independently, with a synchronized queue between each stage to pass frame data between stages. Each stage are described next.

### 4.5.1 Main Thread

This thread is in charge of executing the main program, as well as instantiating all threads involved in the main pipeline. In summary, the tasks handled by the main thread include:

1. parsing input arguments provided in the command line when executing the program,
2. instantiating all main process threads involved on the pipeline, including: Pre Processing, PNET, RNET, ONET, Post Processing and Output threads,
3. configuring the camera (if necessary) and setting the ncurses terminal for input retrieval during the program execution,
4. reading the frame from a file or camera, and creating a Packet with the image data.
5. printing configuration and general execution data on the terminal with ncurses,
6. inserting the packet into the next queue to be processed by the next stage,
7. reading user input from terminal (if any), and
8. jump to step 4 until the entire video is processed.
CHAPTER 4. FRAMEWORK

4.5.2 Pre Processing Thread

This thread is in charge of preparing the data image for processing through the Caffe models. This thread handles the following tasks:

1. remove packets from previous queue,
2. converting image data type from int to float,
3. normalizing the image data from -1 to 1 as needed for the Caffe models,
4. transposing the image - due to the move from Matlab to C++,
5. storing new image data in the processed frame Mat,
6. inserting the packet into the next queue to be processed by the next stage, and
7. iterating through all steps until the entire video is processed.

4.5.3 PNET Thread

Each DCNN stage is modeled with a Caffe prototype [10]. The PNET stage (Proposal stage) applies the neural network to different scaled versions of the image. A maximum of 12 scales is used as a limit to avoid the creation of too many threads. This stage outputs multiple candidates for the bounding boxes of faces for each scaled image.

Due to the high computational requirements of this stage, we further parallelized this stage by performing each of the 12-scaled image analyzes with a different thread. After analyzing the scaled image, these threads grab the detected bounding boxes and merge the highly overlapped candidates by creating a heat map with the highest potential candidates. This might not be the fastest way to implement this process, as each scaled image will be processed at a very different rate (larger images take more time). Given the simplicity of the approach, we adopted this strategy in our framework. This thread handles the following tasks:

1. selects the scales used for each thread,
2. spawns threads, based on the number of scales, and creates a queue for each,
3. removes packets from the previous queue,
4. inserts packets into the PNET queues,
CHAPTER 4. FRAMEWORK

5. waits for the child threads to complete the task by using the auxiliary “WaitForCounter” function,

6. applies NMS to the collection of all possible bounding boxes detected by the child threads,

7. inserts the packet into the next queue to be processed by the next stage, and

8. jumps to step 3 until the entire video processed.

4.5.3.1 PNET Scale Thread

This thread is in charge of PNET Caffe model processing. These are the tasks carried out:

1. creates PNET object (using the Caffe model wrapper described in section 4.3),

2. removes packet from previous queue,

3. resizes frame to the scale provided by the main PNET thread,

4. feeds the Caffe model and forward propagates the input,

5. retrieves the output from the model and generates the bounding boxes based on this data,

6. applies NMS to the collection of bounding boxes detected by this thread,

7. updates the bounding boxes in the data packet atomically to avoid any race conditions with the other threads,

8. tells the main PNET thread that the counter has been updated, and

9. jumps to step 2 until the entire video processed.

4.5.4 RNET Thread

The RNET stage (Refine Network) further identifies false candidates, and decreases the number of possible bounding boxes. It does so by performing a deeper analysis of the features in the bounding boxes and scoring these appropriately. It also outputs a bounding box regression to better identify the face in the image. It then applies NMS again to better merge overlapped candidates. This thread handles the following tasks:

1. creates an RNET object (using the Caffe model wrapper described in section 4.3),
2. removes packets from previous queue,
3. crops all bounding boxes from the processed image and scales them to a 24x24 resolution,
4. feeds the Caffe model and forward propagates the input,
5. retrieves the output from the model and applies the scoring threshold,
6. applies NMS to the collection of detected bounding boxes,
7. applies bounding box regression,
8. converts the generated boxes into squares and pads them using the padding function described previously,
9. inserts packets into the next queue to be processed by the next stage,
10. Jumps to step 2 until the entire video processed.

4.5.5 ONET Thread

The ONET (Output Network) \cite{10} provides the final bounding boxes and the five facial landmark positions for each face. This thread handles the following tasks:

1. create ONET object (using the Caffe model wrapper described in section 4.3),
2. removes packet from previous queue,
3. crops all bounding boxes from the processed image and scales them to a 48x48 resolution,
4. feeds the Caffe model and forward propagates the input,
5. retrieves the output from the model, including the landmarks, and applies the scoring threshold,
6. applies bounding box regression,
7. applies NMS to the collection of detected bounding boxes,
8. inserts the packet into the next queue to be processed by the next stage,
9. jumps to step 2 until the entire video processed.
CHAPTER 4. FRAMEWORK

4.5.6 Post Processing Thread

This thread is in charge of correcting the final output of the system to correctly map into the original frame coordinates. It also applies a simple averaging of bounding boxes that have a high IOU, using the bounding box detected in the previous frame. This step reduces jitter in the detected faces, particularly when the faces are large. This thread handles the following tasks:

1. removes packet from previous queue,
2. corrects the bounding box coordinates to map correctly to the original image,
3. corrects the landmark coordinates to map correctly to the original image,
4. acquires feedback from the previous frame by averaging the detected bounding boxes that have a high IOU with bounding boxes from the previous frame,
5. inserts the packet into the next queue to be processed by the next stage,
6. jumps to step 1 until the entire video is processed.

4.5.7 Output Thread

This thread is in charge of handling the output of the application. The thread either displays the video, or stores a video or snapshot. This thread handles the following tasks:

1. opens an OpenCV window,
2. removes packets from the previous queue,
3. adds bounding boxes and landmarks in the image,
4. displays a frame (optional),
5. saves a snapshot (optional),
6. saves a video (optional),
7. calculates the execution time metrics and sends the value to main thread to display on the terminal,
8. writes metrics to a file (optional).
CHAPTER 4. FRAMEWORK

9. writes results to a file in the FDDB format (optional),

10. deletes the packet,

11. jumps to step 2 until the entire video is processed.

4.6 Application Setup and Execution

After a successful compilation of the application, we can execute the application without any options to print the following help information:

Usage: face-detector [options]

General Options:
   —help,  -h print this message
   —verbose  -e basic verbosity level
   —debug   enhanced verbosity level

Exclusive Options:
   —image,  -i <file_name>: Process image file
   —video,  -v <file_name>: Process video file
   —database,  -d <file_name> <image_directory>: Process a database with multiple images.
   <file_name>: File with the name of the images (no extension, jpg only).
   <image_directory>: Relative path to where the images are stored.
   —cam,  -c <#> : Process video from camera
   <#>: Indicates which camera (if only 1 present, use 0)

Additional Options:
   —nodisp,  -nd Don’t display image/video
   —record,  -r Record video.
   —snapshot,  -s Save image(s).
   —log,  -l Write log with performance results.
   —fddb,  -f Write log with results based on FDDB database.
   —output,  -o <folder_name>: Store outputs in this directory.

As an example, we can process a video without displaying the video or displaying the execution times using the following command:

./face_detector -e -l -nd -v input/video/1080p/Recording\_1.mp4
CHAPTER 4. FRAMEWORK

4.7 Terminal Interface

The information displayed when the program is running is shown in Figure 4.5. The interface displays general information about the video, including the frame size, the frame rate and the current runtime. This interface is available when processing either a video file or a video from a camera. It also displays the following options:

1. Show video: Press ‘v’ to change: This option will open up the OpenCV display to show the video if its value is set to 1, or else it won’t display it. This is useful when running the application remotely. The cost of forwarding a video through ssh is really high which can slow down the application.

2. Pause Video: Press ‘p’ to change: This option will pause the current video.

3. Record video: Press ‘r’ to toggle recording: This option is to activate or deactivate the storing of the resulting frames into a file in .mp4 format.

4. Take snapshot: Press ‘s’ to take snapshot: This option allows the user to take a snapshot of the output at any given moment the application is running.

5. Write log file: Press ‘l’ to toggle writing: This option allows the user to indicate to the application if the execution metrics that are recollected for each frame should be stored into a .csv file or not. This is useful to be able to display the real-time behavior of the application based on the time it takes to process each frame.
CHAPTER 4. FRAMEWORK

Figure 4.5: Example output displayed on the terminal by the application. Interaction with terminal is done with the keyboard using the characters indicated by the display.

This terminal also displays the data for the execution times of each stage if debug or verbose options were in the initial command for the application. This data includes the following metrics.

1. Average FPS: An estimate of the average FPS the application is actually processing at (it can be off by a couple of frames but it’s a good estimation).

2. Average Time: An average of the total time it takes to process a given frame, including the time it takes on each stage, plus the time each frame remains in a queue (if it happens).

3. Total Time: Same as the previous one but this is the real value for each frame.

4. Main Time: Total time the frame takes on the main thread, including reading from the video stream up to inserting the frame into the next queue.

5. PreP Time to Output Time: Total time it takes for the frame to be read from the input queue of that stage and inserted into the next queue.
Chapter 5

Evaluation

In this chapter we evaluate our ability to achieve real-time performance for our neural network. In order to evaluate performance, we use a variety of video files, individual images, image databases and live video streams. The live video stream was obtained using a Logitech HD Pro Webcam C920, capable of 1080p resolution streaming at 30 FPS.

5.1 Static Image Video Streams

We performed two controlled experiments to understand the limits of our implementation. The first test involved a 2 minute long, 1080p, video. The video was collected at 30 FPS, capturing a completely blank (i.e., white) image. This scenario represent the smallest amount of computational load and the shortest execution time to process a 1080p frame that has zero faces present.
CHAPTER 5. EVALUATION

Figure 5.1: Execution time for our pipelined implementation while processing a blank video collected at 1080p.

Figure 5.1 presents the execution time for all of the stages and the total execution time to process each frame. The execution time of each stage starts when the frame is read from the input/previous queue and stops when the frame is added to the next queue. The total execution time starts the moment the frame is read from the file/camera by the main thread and ends the moment the frame is deleted by the output thread. The red line in the graph indicates the FPS limit, which in this case is 33.33 ms.

Figure 5.1 shows that the total execution time to process each frame is over the FPS limit. Processing this video in real-time would not be achievable if our application was implemented without a pipeline. Even though the total execution time is higher than the FPS limit, the framework is capable of achieving the desired FPS given the execution time for all the individual stages remain under the FPS limit. This pipeline structure increases the throughput of the application, while still maintaining the total execution time per frame approximately the same, as compared to the non-pipelined implementation.
CHAPTER 5. EVALUATION

Figure 5.1 also shows that there is an issue with the execution time of the PNET stage for the first frame. This execution time is 73.3 ms, which is 40 ms over the FPS limit. This is relatively high, but can be attributed to the OS scheduling mechanism. This effect is not observed in later frames, which is good news for our implementation. The time it takes to feed all full HD scaled images through the Caffe models in the PNET stage takes on average 19.2 ms. Given that the image has no potential candidates, the time to run the post-model functions is negligible. This test provides positive results and insights into application performance.

The next experiment was to evaluate a 2 minute long, 1080p at 30 FPS video with a static image of a crowd (278 detected faces). This video should allow us to identify the real bottlenecks in dense images, and it provides us with a look at the upper limit in terms of the capabilities of our framework when working with a full HD video stream. Figure 5.2 shows the results for this experiment. The execution time for the PNET and RNET stages is over the FPS limit due to the large number of detected candidates in these stages. The CPU threads perform more work in order to sort and apply NMS, padding, and the post-processing functions associated with these stages. This suggests that our algorithm is unable to maintain the same real-time quality of service that we observed for the blank video.
CHAPTER 5. EVALUATION

Figure 5.2: Execution time for our implementation, while processing a crowd video at 1080p.

The PNET stage execution time ranges from 130 ms to 195 ms. This high variability can be attributed to the multiple threads that are instantiated for this stage. The post-model functions from the "PNET scale" threads require more CPU computation due to the high number of detected candidates. NMS involves a sorting algorithm - the performance of the sort is dependent on the number of detected candidates. The same can be said for the remaining functions. Given our computing platform can run a maximum of 8 concurrent threads (more are possible, but would degrade performance due to the number of physical cores), the operating system has to handle the scheduling of all the threads, suspending threads to grant others access to the shared hardware resources.
CHAPTER 5. EVALUATION

Figure 5.3 is a sample output from the video that was used in this analysis. In this scenario, the average FPS rate is 6.7 FPS, which meets our FPS target for video. The FPS rate is defined by the slowest stage in the pipeline, which is the PNET stage. This stage takes approximately 148 ms on average. A side-effects of this delay are that the total execution time is over our target rate. Figure 5.3 shows, on average, the total execution time is 3.4 seconds. This is misleading, given that the effective execution time per frame is 285 ms, on average. The reason this number is so high is due to the queueing mechanism. This behavior is complex, due to the interaction of the stages and the queues, and the resulting stalls. To better understand this behavior, we ran this same video with a queue size of 1 instead of 10 (default value).
The results in Figure 5.4 show similar results to the ones observed in Figure 5.2, where the main difference is the total execution time. With a queue size of 1, we obtain an average total execution time of 850 ms. Figure 5.5 shows an example timeline of the execution of our application with a queue size of 1, where the PNET stage takes longer to execute than the FPS limit.
CHAPTER 5. EVALUATION

Figure 5.5: Execution example of the pipeline framework utilized for the application when the execution time is larger than the FPS limit, for the video with a queue size of one element.

Figure 5.5 explains how the PNET stage for frame 2 is delayed until the PNET stage for frame 1 has completed. This is displayed as "Next Stage Busy" in the timeline. When frame 3 starts and the main stage finishes processing this frame, it passes the frame to the preprocessing stage. When this stage is complete, it should add this frame to the input queue of the PNET stage. This is not possible, because the PNET stage is still processing frame 1, and frame 2 is still waiting in the input queue. This shows how the preprocessing stage is waiting for the queue to free up to insert frame 3. This is displayed in the timeline with the "Next Queue Full" label.

This domino effect propagates, causing the following frames to be delayed even further. This has no effect in the final output FPS, which is still limited by the slowest stage. The total time it takes to process frame 7 is approximately equal to 5x the time it takes the PNET stage to finish, plus the execution time for the remaining stages. Using these findings, we can calculate that the total execution time would be 820 ms. The real total execution time in our experiment was on average 812 ms. As stated above, the total execution time is measured from the time the frame is read until it is no longer needed. Any delay in these stages or in between them, including wait in a queue, will affect the total execution time.

The reason we are using a queue size of 10 is to minimize the impact of random frame delays that can impact following frames, especially when the execution time for one stage is over the FPS limit. The idea is to minimize the performance impact of outliers. With a larger queue size, the total execution time for each frame goes up, but the frame rate is less affected by outliers.
CHAPTER 5. EVALUATION

5.2 Real-Time Video Stream

We followed our experiments with a real-time test using our 1080p 30 FPS camera. This test involved only one person in the video. Diverse movements were used to qualitatively measure the accuracy of the algorithm.

Figure 5.6: Execution time for the pipeline while processing a live video at 1080p.

Figure 5.6 shows the total time delay at the beginning is still observed. The reason this occurs was explained in the Blank experiment. On average, most stages execute within the FPS limit. This allows us to achieve real-time performance for this test. The total execution time does affect the perception of real-time performance from the application. As shown by Miller et al. [39], applications with a response delay under 100 ms can be considered instantaneous by the human brain. If the response time is over this limit, the human brain is capable of detecting delays, so the application loses the illusion of real-time. On average, the total execution time remains under this threshold. Finally, we can appreciate from the same figure that there are certain places where spikes in execution time where the Total, Main and PreP metrics occur. These appear sporadically.
CHAPTER 5. EVALUATION

throughout different runs and at different moments in time, mainly caused by the operating system scheduling mechanism.

5.3 4K Video Stream

The next test was to stress our application with a 4K video input, as shown in Figure 5.7. A 4K video has a resolution of 3840x2160 pixels. This was a video sample found online, which shows fireworks over Taipai City. Real-time performance is not achievable with this video because the execution time for most stages is above the FPS limit. We can also observe that the execution time for the Main and PreP stages (which only execute on the CPU) are above the limit. The Main and PreP stages dominate this delay, which is mainly due to the lack of CPU resources and the scheduling mechanism used. The PreP stage can also attribute this delay to the lack of scalability of the OpenCV functions, given that the image resolution has significantly increased.

Figure 5.7: Execution time for the pipeline while processing fireworks video at 4K resolution.

Figure 5.7 shows a high peak in execution time for the PNET and RNET stages at frame
CHAPTER 5. EVALUATION

1000. This happens when the camera is pointed at a relatively large crowd. Computing resources are already stressed to their maximum and adding a significant number of face detections presents additional performance overhead. The application achieves a rate of less than 3 FPS during this segment. We require better hardware resources if we expect the frame rate to approach even half of real-time throughput.

The preprocessing stage can be improved through more efficient functions that utilize the GPU. We tried using OpenCV CUDA primitives to improve the execution time for this stage, but this did not improve performance. To improve the Caffe stages, we would need a more highly tuned framework capable of using multi-GPU systems efficiently, which was not explored in this thesis.

5.4 Video Processing with Sequential Implementation

This experiment involved comparing the results of our parallel pipelined implementation with two different approaches: 1) a sequential implementation, and 2) our parallel implementation running on a single core (we will refer to this scenario as parallel-one-thread). The sequential implementation uses the same framework but only reads the next frame when the current frame is finally proceeded by the output thread. We also serialized the execution of the PNET scale threads. All of these implementations still use the GPU for the DCNN Caffe model processing.

<table>
<thead>
<tr>
<th>Resolution</th>
<th>FPS</th>
<th>Length (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blank</td>
<td>1920x1080</td>
<td>29.97</td>
</tr>
<tr>
<td>Crowd</td>
<td>1920x1080</td>
<td>29.97</td>
</tr>
<tr>
<td>NT</td>
<td>1920x768</td>
<td>23.98</td>
</tr>
</tbody>
</table>

Table 5.1: Characteristics of the videos for the sequential implementation.

Three different videos were used for testing: the Blank video, the crowd video, and a snippet from the movie "National Treasure" (NT). All these videos are described in table 5.1. Figure 5.8 shows the average FPS for the 3 videos run with the 3 configurations. We measure the average FPS by counting how many frames were completed in a given amount of time. We found that one second was the most reliable time slice. This value was chosen since smaller values introduced sampling issues. As a result, the values collected represent relative, versus exact, FPS values. They are accurate enough for evaluation in this thesis. We observe that the implementations need to warm up before they reach a steady FPS.
CHAPTER 5. EVALUATION

Our parallel pipelined implementation is capable of handling the real FPS from both the NT and Blank videos. Our implementation averaged 30 FPS for the Blank video. The minor variations in FPS, as compared to the original frame rate, can be attributed to the method used to calculate the FPS rate. We find that our framework can handle the original FPS for this experiment. The sequential and parallel-one-thread implementations are unable to achieve the desired frame-rate for these tests.

The sequential strategy is unable to achieve the original FPS of the videos because it is limited by the total execution time for each frame. We observed a 47.5 ms average execution time per frame, which translates in the graph to an average FPS of 20.9. This implementation runs all PNET scale threads in order, adding an overhead of 7 ms for each frame, as compared to the frame rate in the parallel version. The parallel-one-thread version is capable of handling a higher FPS than the sequential version, despite the limited hardware resources. The PNET-Scale threads suspend while the GPU is processing the data, allowing other threads to use the hardware resources in the meantime. This can be observed for the Blank experiment. In the other experiments, this effect is
CHAPTER 5. EVALUATION

less noticeable because the performance bottlenecks associated with a CPU dominate performance. The NT experiment shows a decrease in performance for both sequential and parallel-one-thread versions. This effect is caused by a sudden appearance of multiple detected faces in the frames when the camera is pointed at a large group of people.

5.5 FDDB Accuracy

Guaranteeing our application maintains the same accuracy as the original algorithm is crucial when considering tradeoffs. To test accuracy of our implementation, we ran the application
on the Face Detection Data Set and Benchmark (FDBB) [32]. This database involves over 2800 images with multiple faces and positions for testing. Figure 5.10 shows the Receiver Operating Characteristic (ROC) curves for the discrete results.

When using an ROC curve, the true positive rate (y-axis) is plotted as a function of the false positive rate, or the false positives count (x-axis). The true positive rate is the probability that an observation is positive, when the actual result is positive. A false positive is a positive prediction to an actual negative result. The higher the accuracy of the application, the closer the ROC curve will be to the upper left corner.

This test compares the accuracy of the application by assigning a good mark (1) if the bounding box detected has a 50% intersection over the union (IOU) with the correct result. As observed in this figure, our application is slightly worse than the original algorithm (MTCNN), but still has better results than other state-of-the-art algorithms [40, 41, 42, 43, 10].

Figure 5.10: Receiver Operating Characteristic (ROC) curve for the discrete measurements on the FDBB database.
CHAPTER 5. EVALUATION

Figure 5.11 shows the ROC curve for the continuous results. In this experiment, the IOU metric is used as a continuous function, assigning a value between 0 and 1, based on the IOU between the detected bounding box and the label from the database. The same behaviour can be observed in this figure, though in this case, the true positive rate for the original algorithm and our implementation is worse than some of the state-of-the-art algorithms. These figures demonstrate that the algorithm is great at detecting faces, but does not perform that well when identifying the boundaries for the faces.

![ROC curve for the continuous measurements on the FDDB database.](image)

5.6 Discussion

The results shown in this chapter suggest that our framework is indeed capable of handling 1080p resolution videos in real-time under normal circumstances. When introducing worst case scenarios such as the crowd test, the execution time for the PNET stage is too high to be able maintain real-time FPS rates. This could be improved by separating the PNET stage from the post-processing done from the PNET stage.
CHAPTER 5. EVALUATION

Sub-1080p resolution videos are easily handled by the framework. Under certain circumstances, the original algorithm should be able of handling low-resolution videos without our parallel framework. When handling high resolution videos, our system struggles to provide enough performance for the deep learning models to maintain real-time throughput, lowering the frame rate to 7 FPS for the 4K test.

Even though performance is good when using our pipelined implementation, the accuracy of this approach is worse than the original. As observed by visual inspection of the outputs from videos and images, the algorithm’s accuracy and robustness have slightly deteriorated compared to the original implementation. This is due to the slight variations in preprocessing libraries used in our approach, as compared to the original algorithm. We used the trained models from Zhang et al. [10] to simplify the implementation of the framework. Due to the differences between the Matlab environment and the Image Processing tools, versus our C++ implementation with OpenCV Image Processing tools, these small variations introduce some differences in output. Some of the issues detected were false positives, inaccurate bounding box results, and bad landmark locations, particularly in the more challenging scenarios that include occlusions and side faces.

Our algorithm was not well designed for video processing. Video processing algorithms correlate the results obtained in previous frames with the current frame processing, versus processing each frame individually, as is done in our framework. Given our algorithm processes each frame individually, small changes in the frames can produce a large change in the output. This can be easily observed in the videos when a bounding box around a face changes its size from frame to frame, despite the frames being almost identical. This effect is also known as jitteriness. To reduce this effect, an average is done across detected bounding boxes on the current frame and the previous frame that shares a high IOU. Despite this, jitter can still be observed in some cases.

Another key factor is the wide variability in execution time for individual frames on a video. The main factor causing these changes is the input frame, since the runtime will depend on how many candidates are detected as potential face objects in the image. This effect can be observed in the crowd test, with over 250 faces in an image, resulting in a significant performance slowdown. The scheduling mechanism also plays a role in the performance of the algorithm. Due to the fact that the implementation is done with multiple threads running on the CPU (approximately 19 threads), the operating system has to divide the threads between the available cores (8 in our environment), and resort to scheduling them to share computational resources. Adding more running processes can significantly impact the execution time. Given that we utilize the CPU and GPU resources extensively, any additional process that requests resources will affect the overall execution. This
delay occurs sporadically, when another process starts executing, such as the backup mechanism. Most of the work after running the Caffe model is executed on the CPU thread, and if this work is extensive (for instance, an image with multiple candidates after the PNET stage), this thread is very likely to be paused in the middle of this post processing.

In order to achieve higher FPS rates on the 4K videos, we need to add more GPUs and consider using smaller Caffe models (but this may affect accuracy). An experiment was performed that utilizes the CUDA API from OpenCV to reduce the computation time for some of the work done by the CPU threads, but due to the lack of robustness for the main functions in the OpenCV API, the performance obtained on the preprocessing stage with OpenCV and CUDA was worse. The main issue is that these interfaces are not sufficiently robust to produce performance improvements, given that they are only capable of handling 1 channel at a time (for an RGB image, we require 3 function calls). Having more cores would allow us to have less variation in the execution time, as all threads can execute concurrently and would not have suspend as often.
Chapter 6

Conclusions

6.1 Summary

The main objective of this thesis was to explore the feasibility of providing real-time face detection in video streams. We were able to demonstrate the framework can process a 1080p resolution video in real time using our parallel pipeline framework. Our framework was created with flexibility in mind, allowing it to integrate other algorithms easily. We can use this framework for additional applications, with the goal of demonstrating real-time performance for face detection. Using an accelerated system that can utilize the performance of a GPU, combined with the speed and flexibility of a CPU with multiple cores, there are a large number of image/video processing applications that can benefit from a similar CPU-GPU platform.

To develop a broader understanding of the face detection pipeline, we carried out a large number of experiments. We have reported on the results of those experiments, breaking down performance across each stage of the processing pipeline. The resulting project is available on github: https://github.com/jgutierrezm113/Face-Detection

6.2 Future Work

This thesis has shown that achieving real-time face detection using a pipelined neural network, we are able to optimize for throughput versus latency. To further expand the portability of this framework, one direction is to leverage the power of the GPU, transferring more work to the GPU from the CPU threads. One area ripe for this optimization is the preprocessing stage. Creating a kernel that can receive a Mat object and apply all of these preprocessing steps in a single kernel
CHAPTER 6. CONCLUSIONS

would result a much faster preprocessing stage, one that should be able of handling 4k resolution frames in real time.

One other attractive aspect of this framework is the versatility of the thread design. If we can identify a better PNET stage implementation that is faster than the present implementation, we can easily integrate it into our framework, providing a system that could achieve even better performance. We can also consider adding more stages (e.g., adding another stage that refines the landmark location). The original algorithm from Zhang et al. [10] has recently been updated with an additional stage to refine the landmark location, so this would be a logical first choice to consider.

Additional optimizations could also be pursued to improve performance of the framework. Retraining the model with better preprocessing tools, we could reduce the preprocessing time significantly, eliminating the transpose processing of the image. We could also focus on improving the accuracy and robustness. There is plenty of room for improvements and we are hope to expand the functionality of the framework in the near future.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


