



# **Deployment of AI Model inside Docker on ARM-Cortex-based Single-Board Computer**

**Technologies, Capabilities, and Performance**

**Helina Getachew WoldeMichael**

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering with Emphasis on Telecommunication Systems. The thesis is equivalent to 20 weeks of full time studies.

**Contact Information:**

Author: Helina Getachew WoldeMichael

E-mail: hege16@student.bth.se

**University advisor:**

Prof. Dr. Kurt Tutschku

Department of Telecommunication Systems

School of Computing

BTH, Karlskrona

Faculty of Computing  
Blekinge Institute of Technology  
SE-371 79 Karlskrona, Sweden

Internet : [www.bth.se](http://www.bth.se)  
Phone : +46 455 38 50 00  
Fax : +46 455 38 50 57

# ABSTRACT

IoT has become tremendously popular. It provides information access, processing and connectivity for a huge number of devices or sensors. IoT systems, however, often do not process the information locally, rather send the information to remote locations in the Cloud. As a result, it adds huge amount of data traffic to the network and additional delay to data processing. The later feature might have significant impact on applications that require fast response times, such as sophisticated artificial intelligence (AI) applications including Augmented reality, face recognition, and object detection. Consequently, edge computing paradigm that enables computation of data near the source has gained a significant importance in achieving a fast response time in the recent years. IoT devices can be employed to provide computational resources at the edge of the network near the sensors and actuators.

The aim of this thesis work is to design and implement a kind of edge computing concept that brings AI models to a small embedded IoT device by the use of virtualization concepts. The use of virtualization technology enables the easy packing and shipping of applications to different hardware platforms. Additionally, this enable the mobility of AI models between edge devices and the Cloud. We will implement an AI model inside a Docker container, which will be deployed on a Firefly-RK3399 single-board computer (SBC). Furthermore, we will conduct CPU and memory performance evaluations of Docker on Firefly-RK3399.

The methodology adopted to reach to our goal is experimental research. First, different literatures have been studied to demonstrate by implementation the feasibility of our concept. Then we setup an experiment that covers measurement of performance metrics by applying synthetic load in multiple scenarios. Results are validated by repeating the experiment and statistical analysis.

Results of this study shows that, an AI model can successfully be deployed and executed inside a Docker container on Arm-Cortex-based single-board computer. A Docker image of OpenFace face recognition model is built for ARM architecture of the Firefly SBC. On the other hand, the performance evaluation reveals that the performance overhead of Docker in terms of CPU and Memory is negligible.

The research work comprises the mechanisms how AI application can be containerized in ARM architecture. We conclude that the methods can be applied to containerize software application in ARM based IoT devices. Furthermore, the insignificant overhead brought by Docker facilitates for deployment of applications inside a container with less performance overhead. The functionality of IoT device i.e. Firefly-RK3399 is exploited in this thesis. It is shown that the device is capable and powerful and gives an insight for further studies.

**Keywords:** Artificial Intelligence, Docker, Edge Computing, Single Board Computer

## **ACKNOWLEDGMENT**

For most, my sincere gratitude goes to my advisor Prof. Dr. Kurt Tutschku for his patience and invaluable guidance. Without his support, it would not have been possible to accomplish this thesis. His door was always open whenever I had a question about my thesis. His feedbacks and research experience helped me in critically viewing key topics and writing this thesis.

I am grateful to PhD students Vida Ahmadi and Yuliyana Maksimov with whom I have had the pleasure to work in this thesis and ATS course. I am very thankful to Vida Ahmadi for her critical feedbacks and guidance. I would also like to thank Yuliyana Maksimov for his comments and suggestions on technical and research matters during ATS course.

I would like to extend my appreciation to Prof. Wlodek Kulesza for his course research methodology which was a guideline for me in writing this dissertation.

I must express my deepest gratitude to the Swedish Institute (SI) for generous scholarship which fully financed my university tuition fee and my accommodation expenses.

My heartfelt gratitude goes to my wonderful family and friends. Their constant prayer, support and motivation have been a great source of energy throughout my years of study.

# TABLE OF CONTENTS

<b>ABSTRACT .....</b>	<b>I</b>
<b>ACKNOWLEDGMENT .....</b>	<b>II</b>
<b>TABLE OF CONTENTS .....</b>	<b>III</b>
<b>LIST OF FIGURES.....</b>	<b>V</b>
<b>LIST OF TABLES.....</b>	<b>VI</b>
<b>LIST OF ABBREVIATIONS .....</b>	<b>VII</b>
<b>1 INTRODUCTION .....</b>	<b>1</b>
1.1 MOTIVATION .....	2
1.2 PROBLEM DESCRIPTION .....	2
1.3 AIM AND RESEARCH QUESTION .....	3
1.4 CONTRIBUTION .....	4
1.5 TARGET AUDIENCE .....	4
1.6 THESIS OUTLINE .....	4
<b>2 BACKGROUND.....</b>	<b>5</b>
2.1 BONSEYES .....	5
2.2 CLOUD COMPUTING .....	5
2.3 CLOUD SERVICE TYPES .....	6
2.3.1 IaaS .....	6
2.3.2 PaaS .....	6
2.3.3 SaaS .....	6
2.4 EDGE COMPUTING .....	6
2.5 WHY DO WE NEED EDGE COMPUTING? .....	7
2.5.1 Latency .....	7
2.5.2 Bandwidth .....	7
2.5.3 Security .....	7
2.6 VIRTUALIZATION .....	8
2.7 CONTAINER TECHNOLOGY .....	8
2.8 DOCKER CONTAINER .....	9
2.9 ARCHITECTURE OF DOCKER .....	9
2.10 FEATURES OF DOCKER CONTAINER .....	10
2.11 ROLE OF DOCKER IN EDGE COMPUTING PLATFORM .....	10
2.12 IOT DEVICES .....	11
2.13 ARM ARCHITECTURES .....	11
2.14 FIREFLY-RK3399 .....	12
<b>3 RELATED WORK.....</b>	<b>13</b>
<b>4 METHODOLOGY .....</b>	<b>15</b>
4.1 OVERALL CONCEPT .....	15
4.2 DEMONSTRATOR.....	16
4.3 EXPERIMENT TESTBED.....	16
4.4 IMPLEMENTATION OF THE DEMONSTRATOR .....	18
4.4.1 Environment Setup of Firefly-RK3399 .....	19
4.4.2 Running Docker .....	20

4.4.3	OpenFace .....	21
4.4.4	Containerization of OpenFace on Firefly-RK3399.....	22
4.5	PERFORMANCE EVALUATION .....	25
4.6	PERFORMANCE EVALUATION METHODOLOGY .....	25
4.7	EXPERIMENT SETUP.....	25
4.8	CPU PERFORMANCE EVALUATION.....	26
4.8.1	SYSBENCH .....	26
4.9	MEMORY PERFORMANCE EVALUATION .....	27
4.9.1	STREAM.....	27
4.9.2	STREAM2 .....	27
<b>5</b>	<b>RESULTS AND ANALYSIS .....</b>	<b>29</b>
5.1	PORTABILITY OF AI MODEL .....	29
5.2	CPU PERFORMANCE .....	31
5.2.1	SYSBENCH .....	31
5.3	MEMORY PERFORMANCE.....	34
5.3.1	STREAM .....	34
5.3.2	STREAM2 .....	35
<b>6</b>	<b>CONCLUSION AND FUTURE WORK.....</b>	<b>36</b>
6.1	ANSWERING RESEARCH QUESTIONS .....	36
	<b>REFERENCES .....</b>	<b>38</b>

## LIST OF FIGURES

Figure 2-1 Docker Virtualization Technology .....	9
Figure 2-2 Docker Architecture .....	10
Figure 2-3 Firefly-RK3399 board components .....	12
Figure 4-1 Experiment test bed.....	17
Figure 4-2 Model of Demonstrator.....	18
Figure 4-3 Flash operating system into Firefly-RK3399 board using Android Tool .....	20
Figure 4-4 Hello-world Docker container .....	21
Figure 4-5 OpenFace project structure .....	22
Figure 4-6 Sample observation method of an experiment.....	25
Figure 4-7 Experiment setup .....	26
Figure 4-8 Sysbench sample benchmarking output.....	27
Figure 5-1 Output of OpenFace on Firefly .....	29
Figure 5-2 Sequence diagram of transferring AI model in a Docker image.....	30
Figure 5-3 OpenFace AI model for ARM in Bonseyes Marketplace GUI.....	30
Figure 5-4 Execution time of Sysbench benchmark measured in a single CPU core and six CPU cores of Firefly-RK3399.....	33
Figure 5-5 STREAM Mean Memory bandwidth on COPY, SCALE, ADD and TRAIID operations in Native and Docker Platforms.....	34
Figure 5-6 STREAM2 Mean Memory bandwidth on FILL, COPY, DAXPY and DOT operations in Native and Docker Platforms.....	35

## LIST OF TABLES

Table 4-1 Experiment testbed description of IoT device.....	18
Table 4-2 STREAM operations.....	27
Table 5-1 Mean, standard deviation and confidence interval for Execution time of Firefly's single CPU core in Native Docker platforms .....	32
Table 5-2 Mean, standard deviation and confidence interval of Execution time of Firefly's six CPU cores in Native and Docker platforms.....	32
Table 5-3 Mean, standard deviation and confidence interval of Execution time of Firefly in Native and Docker platforms with different CPU cores with fixed input load.....	32
Table 5-4 STREAM Memory bandwidth of kernel operations for Native and Docker platforms .....	34
Table 5-5 STREAM2 Memory bandwidth of kernel operations for Native and Docker platforms .....	35



## LIST OF ABBREVIATIONS

AI	Artificial Intelligence
API	Application Program Interface
ARM	Advanced RISC Machine
AUFS	Advanced multi-layered Unification File System
Cgroups	Control Groups
CPU	Central Processing Unit
GPU	Graphical Processing Unit
GUI	Graphical User Interface
HOG	Histogram of Oriented Gradients
IaaS	Infrastructure as a Service
IDC	International Data Corporation
IoT	Internet of Things
IPC	Inter Process Communication
NIST	National Institute of Standard and Technology
NSC	Network Service Chain
PaaS	Platform as a Service
OS	Operating System
RFID	Radio Frequency Identification
RISC	Reduced Instruction Set Computer
SaaS	Software as a Service
SDN	Software Defined Network
SoC	System on Chip
WSANs	Wireless Sensor and Actuator Networks
VM	Virtual Machine
VNF	Virtual Network Function

# 1 INTRODUCTION

Internet of Things (IoT) vision is emerging and turn out to be a reality. IoT comprises several devices such as, sensors, smartphones, computers and actuators that can be connected, monitored or actuated. The number of these devices, as estimated by Cisco, will increase up to 50 billion in 2020 [1] . Globally, by 2019, data produced by people, machines, and things will reach 500 zettabytes as estimated by Cisco Global Cloud Index. But by that time, the global data center IP traffic will only reach 10.4 zettabytes [2] . Hence the computation of enormous amount of data from IoT to the cloud will cause a traffic on the network. The statistics raise the question whether the cloud will keep up with the performance requirement of applications running on IoT devices.

With widespread implementation of IoT we are heading to post-cloud era. As IDC [3] predicted, by 2019, at least 40% of IoT-Created data will be stored, processed, analyzed, and acted upon close to, or at the Edge of, the network. Edge computing is a model in which some part of data is computed locally, at the edge of, the network near the devices instead of transporting all data to the Cloud servers. However, the features of cloud computing are also crucial for the provision of high performance, scalable, reliable and efficient IoT services. The concept of cloud computing is characterized by on demand service provision, ubiquitous access, resource pooling, and elasticity [4]. Therefore, it is compelling to merge the Cloud and IoT paradigm [4].

Currently, Artificial intelligence is cutting edge technology which will affect our everyday life. With rapid growth of AI, it's evident that resources are needed to compute large amount of data instantly. Deploying AI applications on IoT devices can be advantageous for low latency and time-critical applications and with poor network bandwidth. However, the concept of moving neural-network-based AI inference from the cloud to the edge has not yet been exploited extensively. There are not been many commercial deployment of AI algorithms on IoT devices. Drones, surveillance camera and self-driving cars are few use cases in which inference is performed locally based on the sensor inputs [5].

Bonseyes [6] is EU H2020 collaborative project that aims to provide a platform for open development in building, improving and maintaining systems of Artificial Intelligence (AI). The target is for the adoption of AI in data center servers at the cloud and at the edge with low-power IoT devices. Sensors, devices, edge, and cloud are the four layers of AI computations. Bonseyes also addresses the challenges of closed monolithic development of deep-learning-based AI systems and overcomes the resistance to the exchange of AI artifacts.

This thesis is part of a group of three theses, which together implement and investigate a BTH demonstrator of the Bonseyes system. The combined aim of all thesis is to show and demonstrate by implementation the capabilities and feasibility of the Bonseyes system and the interworking of different Bonseyes components. The three theses are expected to contribute individually to the understanding of subparts of Bonseyes and on how the overall system needs be implemented. The following theses are conducted in VT 2018 for the implementation of BTH's Bonseyes demonstrator:

- Investigation and Deploying an AI model on Raspberry Pi IoT platform using FIWARE and Docker in the context of the Bonseyes

- Design and implementation of an AI based face recognition model in Docker container for use on Bonseyes IOT platform
- Deployment of AI Model inside Docker on ARM-Cortex-based Single-Board Computer

During this thesis, we will be studying the mechanisms to run Docker on the Firefly-RK3399 IoT device for deployment within the Bonseyes AI Ecosystem. We will research on the technologies to understand the requirements to implement this sub-part of the BTH demonstrator. The main goal of this thesis is to bring computation of AI application to embedded IoT device and ensure the portability of this application in a Docker container.

Edge computing shifts the attention into distributed lightweight virtualized resources instead of heavy-weight data center cloud. Docker is a light-weighted and flexible container that can be run on small devices (IoT devices) to large Cloud servers. Its performance, agility, portability, and low storage footprint makes it ideal for containerized deployment of applications on low energy consuming but highly powerful, i.e. very resource efficient edge devices [7].

Recently, many chipsets are launched with a processor features customized for AI models [5]. Independent of Cloud capacity, image processing and Augmented Reality (AR) applications can be performed using device-based inference or direct sensor inputs on processors such as NVidia's Jetson and ARM's CortexA-75. Firefly-RK3399 [8] is a single board computer which has dual-core Cortex-A72 and quad-core Cortex-A53 ARM processor. Its CPU+GPU hardware structure is promising for complex and high performance AI computing.

## 1.1 Motivation

AI applications are playing a great role in the technology world by enabling human level accuracy and intelligence to machines. Embracing AI applications in our day-to-day activities makes our life easy. The infrastructure and platform that is used for development and deployment of these applications needs to be selected appropriately. It is crucial to investigate the development and deployment of AI models on embedded IoT devices. It would be interesting to explore the capability of low power IoT devices. As most of the computation is performed in the cloud, movable and low power IoT devices can offload the computation of huge amount of data produced/consumed during the training of AI models. It is also worthwhile to study how container technology can be utilized here to pack and move AI applications between IoT devices.

The motivation behind initiating this thesis is to explore the capability and feasibility Firefly-RK3399 for containerization of deep-learning based AI applications. Firefly is an IoT device which adopts CPU + GPU hardware structure for high performance AI computing. We demonstrate this by implementing our BTH Bonseyes system.

## 1.2 Problem description

The platform to develop and deploy AI applications must be appropriate for service providers, developers and, eventually to users. Most AI application are developed for a particular platform only. For example, Airpoly [9] is an iOS based AI app for visually

impaired people that can instantly identify and describe objects when pointing at the object. The application can only be installed and used in iPhones which makes it incompatible for mobile devices with a different platform. One way to overcome this limitation is to develop a compiler in which its binary format can be interpreted by the specific platform. These compilers are nowadays developed for different hardware platforms however there is not much distribution for ARM based single board computers. If compilers are not available, it is difficult for developers because they cannot reuse and share components of applications. Besides, when users want to use AI application they must fulfill the necessary hardware requirements, in our case iPhone mobile, which is not cost effective. An abstraction technology between the hardware and the applications is another method to solve such problem. Docker is very light-weighted abstraction technology which can be adopted to deploy such applications. It can be used to pack software applications and all its dependencies in a container. Hence, it could be possible to move these applications between different environments by enabling them to adopt to demanded platforms. This facilitate easy exchange of components of an application and therefore collaboration between developers.

Computation of Big data cannot be achieved only in central cloud servers. Rather shifting from the Cloud to low energy consuming IoT devices is one way to balance the computational load. Especially for AI application that demand faster response time. The proximity of IoT devices to the network edge makes them advantageous to give a better response time depending on their computational power. We used Firefly-RK3399 IoT device for implementation. The thesis will also comprise the capability of this IoT device and whether it fits into Bonseyes Ecosystem.

### **1.3 Aim and research question**

In this paper, we will be studying the deployment of deep learning based Artificial intelligence (AI) applications on an ARM based IoT device. The aim of this research is to compute AI applications on single board embedded IoT device under Docker abstraction technology. The first objective is to demonstrate by implementation the capabilities of BTH part of Bonseyes system and interworking of different Bonseyes components. This part of BTH's Bonseyes system consists of Firefly-RK3399 IoT device, face recognition AI model, Docker-Engine, Bonseyes repository, and Bonseyes's AI marketplace GUI. Firefly-RK3399 hosts AI model within Docker container. Open source face recognition application called OpenFace is used as an AI model for this experiment. The second objective is study the performance of Docker on Firefly-RK3399. We will quantify if there is any performance overhead due to Docker abstraction layer.

The research questions that will be addressed in this thesis are:

1. How can Docker be installed, configured, and instantiated (run) on a Firefly-RK3399 IoT device such that it demonstrates that an AI model can be executed on it? What are the implications for the Bonseyes environment?
2. How significant is the performance overhead of Docker when a load module is deployed in a container on Firefly-RK3399 platform in terms of CPU and memory utilization?

## 1.4 Contribution

Firstly, the contribution of this thesis is providing methods to setup Docker environment on Firefly-RK3399. The virtual environment by itself can be used to deploy applications. Secondly, the implementation of BTH part of Bonseyes system will illustrate how Bonseyes artefacts can be deployed on the Firefly-RK3399. Thirdly, the newly built Docker image for OpenFace AI model, which is compatible for Firefly-RK3399, will be available on Bonseyes repository. Furthermore, the thesis provides the recipe to build the Docker image on ARM architecture SoC. Fourthly, this work contributes in presenting the performance of Docker on Firefly-RK3399. Finally, our prototype contributes to the understanding of subparts of Bonseyes system and on how the overall system needs be implemented. Moreover, this eventually provide insight on how the Bonseyes system should be re-designed or improved.

## 1.5 Target Audience

Basically, this thesis is targeted to Bonseyes consortium that is currently working on the Bonseyes project of providing a platform for development, deployment and maintenance of systems of AI. Yet, it may also benefit anyone who needs the knowledge about the capability of Firefly-RK3399 to deploy and run AI applications. As well, enterprises who put their best effort to build infrastructure and platform to provide an efficient level of computation resources can take the advantage of exploring the capability of the IoT devices.

## 1.6 Thesis outline

This paper work is structured in the following way:

We started by introducing the general setting of thesis in terms of IoT system, and AI development. We provided overview of the current issues which leads us to the motivation to do this thesis. Then we clearly stated the aim, research questions and its contribution accordingly.

The technologies of the thesis areas are described in chapter two. These areas include Bonseyes project, Cloud computing, Edge computing, Container Technology (Docker) and IoT devices.

In Chapter 3, we will survey on the previous works which are related to our research.

Chapter 4 is where we describe the methodology that we follow to reach to the goal of this thesis. It consists of our Bonseyes Demonstrator, experimental set up, and measurements methods and tools.

In Chapter 5 we will present the result and findings from our experiment. The output of the implementation will be discussed here.

We conclude the thesis work on Chapter 6. We will try to answer our research questions based on our findings. Moreover, we will suggest a set works as outlook for other researchers.

## 2 BACKGROUND

### 2.1 Bonseyes

Bonseyes [10] is a platform for open development in building, improving and maintaining systems of Artificial Intelligence (AI). Its main components include AI Marketplace, a Deep Learning Toolbox, and Developer Reference Platforms. The goal is for the adoption of AI application development and deployment both on the cloud and at the edge with low-power IoT devices. The main objective [6] of Bonseyes project is to reduce the development and deployment time and cost of AI applications. Another perspective is to overcome the challenge of closed monolithic development of deep-learning-based AI systems and overcomes the resistance to the exchange of AI artifacts[10]. At the end it is expected that, developers will find the AI marketplace a place to trade their intelligent systems. On the other side end users can interact with the application from a friendly user interface to solve their challenges. All this process is taken place with comparatively efficiently managed resources geographically located at different areas.

The initial Bonseyes architecture in [10] has a three-layer architecture, AI marketplace layer, AI artefacts layer and resource layer, with focus on supporting the AI pipeline of distributed AI artefacts. The AI pipeline framework is constructed in a way that it contains tools with specific functions, a runtime which is a runtime instance of the tool and AI artefacts to store the output data from the pipeline. The basic structure of AI pipeline is the same as Network Service Chain (NSC). Although the technology of combining using software-defined network (SDN) with VNF is yet emerging, NSC ensures an optimum provision of different network services SDN together with virtual network function (VNF) to guarantee that the services are controlled, chained and mobile. Similarly, Bonseyes employed Docker container to build AI artifacts into AI pipelines so that components of AI artefacts can be reusable, easily connected and communicable using APIs.

### 2.2 Cloud Computing

NIST defines cloud computing as an information technology paradigm that provides ubiquitous, convenient, on-demand access to a shared pool of configurable computing resources with minimal management effort or service provider interaction [11]. Networks, servers, storage, applications and services can be considered as computing resources [11]. They are provided over the internet so that users can access it from anywhere in the world. In addition, the infrastructure is centralized and built as multi-tenant model. Users can pool, share, allocate, and/or deallocate among different physical and virtual resources depending on their demands without the need to understand how the components of system/infrastructure works. The service providers on the other hand ensures the delivery of the service by isolating the users from the underlying infrastructure.

Rapid elasticity is one of the characteristics of cloud computing [12]. It is the capability of the providing rapid cloud services that can scale upward and downwards based on demands. Since it is possible to monitor, control and report resource usage all the cloud services are measured. Hence customers are asked to pay based on their usage [12].

Cloud computing have five essential characteristics. These are:

1. Resource pooling- assign/reassign physical and virtual resources dynamically
2. Measured service- the service is monitored controlled and measured. Hence users pay per usage.
3. Rapid Elasticity- Scaling of resources based on user's demand
4. Broad network access- service provision over the internet
5. On-demand self service- no need to interact with service providers

## 2.3 Cloud Service types

Cloud computing can be divided into three major categories based on the type of service they are delivering. These are:

- Infrastructure as a service (IaaS)
- Platform as a service (PaaS)
- Software as a service (SaaS)

### 2.3.1 IaaS

In an IaaS service model, fundamental computing resources such as physical and virtual machines are provided to the user [13]. The service provision is also complemented by cloud storages and network capabilities. Relatively, users of IaaS have a control over the infrastructure. Either they build other services (PaaS or SaaS) or run software on the top of this model [11]. Amazon EC2 and Google Compute Engine are examples of IaaS services.

### 2.3.2 PaaS

PaaS provides a computing platform which additionally comprises of runtime environment and development tools over the underlying infrastructure [13]. Hence it saves the time and effort to install and setup the development environment.

### 2.3.3 SaaS

SaaS model delivers software to be used as a service. Users need not to install software on their local system instead they can access centralized commercially available software on-demands [13]. For example, software like Google Gmail are owned by the vendor (Google) and can be accessible to end users over the internet.

## 2.4 Edge Computing

In 1999, the term *Internet of Things* was introduced by Auto-ID center that was working on the area of networked Radio Frequency Identification (RFID) and emerging sensing technologies at Massachusetts Institute of Technology (MIT) [14]. Later, they began to design and propagate a cross-company RFID infrastructure [15]. The vision of IoT is connecting the everyday physical things with the virtual world using the internet thereby data can be remotely accessed and controlled from these objects [15].

Nowadays, the implementation of IoT become predominant and lots of data are generated by the devices near the edge of the network [16]. The edge also accompanies applications which consume and process these data. As estimated by Cisco Global Cloud Index, data produced by people, machines, and things will reach 500 zettabytes by 2019. But by that time, the global data center IP traffic will only reach 10.4 zettabytes [2] [17]. Besides, the number of



devices connected to the internet are also increasing. Depending on the type of application that will be run on the IoT devices the performance requirement can be different. For example, some applications might prioritize fast response time while others for high computational power. Such demands cannot be fully fulfilled by the cloud computing only. The edge of the network should offload some computing tasks by transforming from data consumer to data producer as well as data consumer. This brings the concept of *Edge computing* into the picture.

Edge computing is a model that allows computation to be performed at the edge of the network, which works both on downstream data on behalf of cloud services and upstream data on behalf of IoT services [17] [16]. It can perform computing offloading, data storage, caching and processing, as well as distribute request and delivery service from cloud to user. We refer edge device as, any computing or networking resource located between data sources and cloud-based datacenters. For instance, a gateway in a smart home is the edge between home things and cloud.

## **2.5 Why do we need Edge Computing?**

The rationale behind Edge computing is to make the computation at proximity of the source. This approach has its own advantages over the cloud computing. Here we discuss these advantages with respect to latency, bandwidth, and security.

### **2.5.1 Latency**

In edge computing rather than transporting all data to the cloud, computation happens near the devices. Hence the delay for request/response to/from the cloud will be avoided. Due to the proximity, the roundtrip time for computation will be minimized. A prototype platform is built by researchers in [18] to run a face recognition application. When the application is moved from the cloud to the edge its response time is reduced from 900ms to 169ms. This shows that, Edge computing is beneficial for application with low latency requirement. Local decision making is also vital for cognitive assistance systems [16] and augmented reality (AR) that demand fast response time.

### **2.5.2 Bandwidth**

As the number of devices that access cloud services and applications increases it causes high latency to application, and generates congestion and bottleneck to the network [7]. Despite of the intensive traffic, the bandwidth of a network is usually constrained and costly. Edge computing in some way reduces this traffic because computation happens locally. Besides it relieves the long back and forth data transmissions among cloud and Edge devices. Hence it saves bandwidth and improves user's quality of experience [19]. In addition, local computation reduces the reliance on internet connectivity. Even though the cloud has a high computation power the data transmission costs high bandwidth because of its distance from the IoT devices [19]. Thus, it cannot fulfill the demand of real-time computing intensive application.

### **2.5.3 Security**

Relative to the cloud, the computation of the data at the edge occurs close to the source. Keeping the data near its source ensures the privacy of the data. Moreover it protects the data from the exposure to possible threats during its transmission to the remote cloud server [16].



## 2.6 Virtualization

Virtualization is an abstraction technology that provides optimization of hardware resources and operational flexibility in a cloud infrastructure [20]. It enables to isolate different parts of application and services by creating a virtualization layer between multiple instances. However, this virtual layer causes resource overhead. There are three different categories of virtualization techniques. These are full-virtualization, para-virtualization and container-based virtualization [20]. Both full and para virtualization use a software called hypervisor to isolate host machine from guest operating system instances. They run full operating system for each isolation instances and running multiple instances demand large amount of resources. On the contrary, containers run as processes on the host operating system. As its isolation mechanism is very light-weighted container-based virtualization have less overhead and resources can be used more efficiently [21].

## 2.7 Container Technology

Container technology is Operating-system-level virtualization method in which multiple isolated user spaces instances called containers are created using the feature in the OS-kernel. The resource allocation is also controlled and limited to each container. Therefore, every container has its own unique process id and filesystem namespace. Standard IPC mechanisms (sockets, pipes) is used as a means of communication.

The basic features in the OS-kernel that allows the isolation of resources are the following:

**Chroot:** change the root directory and its child processes. It is used to isolate and share a file system.

**Kernel Namespace:** there are several namespaces [21] in Linux kernel that creates isolation between containers. Namespace isolation allows groups of processes to have unique id. It allows each container to have its own instances. This ensures that they cannot see resources in other groups. The different namespaces used for isolating various tasks are:

- Pid namespace– provides unique process id
- net namespace– provides network artifacts such as routing table, loopback interface, iptables.
- IPC namespace– isolate different IPC mechanism like semaphore, message queues and shared memory segment.
- mnt namespace– provides mount points
- UTS namespace– enables containers to view and change their hostnames.

**Cgroups:** limit the usage of resources such as cpu, and memory among the containers. A key to running applications in isolation is to have them only use the resources you want which allows containers to be multi-tenant guests within a host. Control groups also allow containers to share available hardware resources and, if required, set up limits and constraints. For example, limiting the memory available to a specific container [22] .

## 2.8 Docker Container

Docker is a light-weighted operating system level virtualization technology. It is developed in Go programming language and operates on the underlying operating system and infrastructure. The Linux kernel's functionality of cgroups and namespaces are used by Docker to allocate and limit its resources and isolate its processes from the host operating system respectively.

The open source platform of Docker allows developers and sysadmins to build, pack and ship applications together with their libraries and dependencies within a container [23]. They are designed to provide fast, easy and consistent development platform to move codes between developers. It minimizes the time it takes to write and run a code in a production environment.

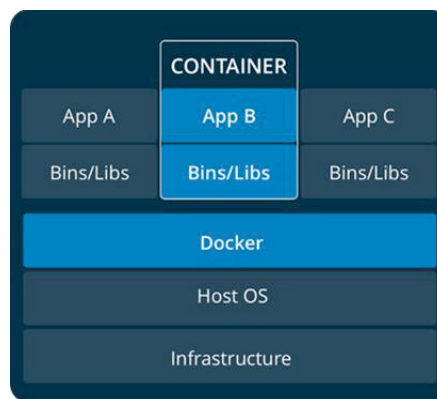


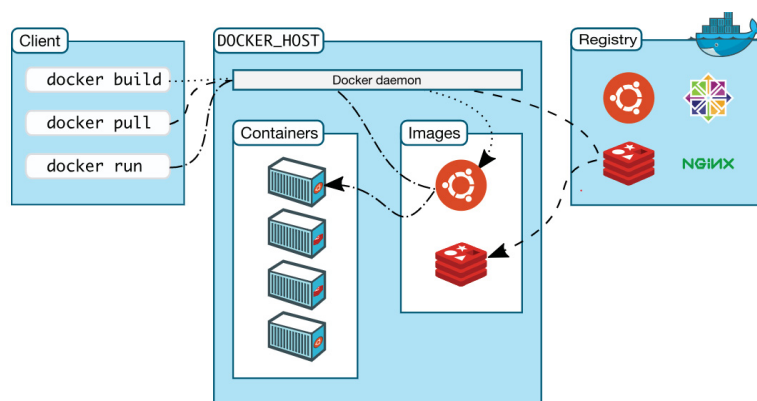
Figure 2-1 Docker Virtualization Technology [24]

## 2.9 Architecture of Docker

A client-server architecture of Docker consists of three main components: Docker client, Docker host and Docker registries as shown in the Figure 2-2 below. Client and Docker host or server are *Docker* and *Dockerd* respectively [7]. Communication between these entities is implemented in REST API. When requesting for service by using Docker commands, like Docker pull and Docker run, a user is acting as a Docker client. The daemon serves the client requests and manages objects like images, containers, networks and volumes [25]. Docker registry is repository which is used to store Docker images [25]. Docker Hub is Docker's public repository and is by default configured to be a registry to look for images.

Docker containers is generated with instructions that are constructed in the read-only Docker images [7]. Dockerfile is used to build and create an image. It can contain base image, the program to be executed, and all of its dependences. Docker images are instantiated in a container. Once the container is up and running, with the provided options of Docker it can perform the functionalities of the image. Since containers are stateless any changes made do not persist unless it is pushed back to the image. When containers run, UnionFS creates a writable layer on the top and this layer is used to update into a new image [26].

Docker uses AuFS (Advanced MultiLayered Unification Filesystem). Such filesystem allows to overlay one or more existing filesystems i.e. a modification of a file is done by first copying that file then it writes on the copied file (copy-on-write). This makes the images more virtualized through the image versioning management.



**Figure 2-2 Docker Architecture [25]**

## 2.10 Features of Docker Container

Docker is now popular and most adopted container technology. There are several reasons for its wide acceptance. First, it enables to build software applications and their dependencies locally and deployed in the cloud so that they can be run anywhere. With its light-weighted runtime and packaging tool and automation of workflow Docker allows to assemble components quickly and avoid friction between development and production environment [22]. This makes Docker containers easily portable. Another important feature of Docker is its agility. Even the most complex applications can be containerized into a small sized Docker images. Docker image use layering file system and only changes are saved on the top the layer. To persist changes in Docker container those changes need to be committed into the Docker image. Therefore, Docker has a low storage footprint which helps to save storage space [7].

## 2.11 Role of Docker in Edge Computing Platform

Edge computing model shift from large-scale cloud server to more distributed edge data centers. The service at the edge should be flexible and elastic. To meet the computing resource demand at the Internet of Things (IoT), we need to abstract the infrastructure at the edge of the network, IoT devices. The use of containers as an abstraction technology have advantages over VMs [27]. First it is very light weighted and portable runtime. Second it has the capability to develop, test and deploy applications to a large number of servers. Third, the capability to interconnecting containers.

Docker is a flexible platform which makes it ideal for deploying reusable services without dependency on heterogenous edge devices. It is easy to install, configure, manage and terminate services. It is also fault tolerant which ensures the availability and reliability of resources at the edge [7].

## 2.12 IoT Devices

In IoT, devices are connected often in a constrained communication bandwidth. These devices are usually simple, small and portable. They can be sensors/actuators, or smart devices. Depending on their design considerations, they are usually resource-constrained in processing power, memory and electrical power. IoT devices are used to gather information in buildings, homes and natural ecosystems and send data to server station [28]. Sometimes they act upon on the data. IoT has a distributed architecture that connect uniquely identifiable and addressable things. Other entities on the network such as controlling servers might have more computational and communication resources therefore could support interaction between constrained devices and applications [28]. Based on their computing power, IoT devices can be divided as RFID tags, Wireless sensor and actuator networks (WSANs), low-end computing devices, middle-end computing devices, and high-end computing devices.

## 2.13 ARM Architectures

ARM (Advanced RISC Machine) is RISC (Reduced Instruction Set Computer) processor. It has highly optimized instruction set. It is one the most widely used embedded architectures. The processor is applicable in smart phones, embedded devices and laptops. The principal advantages of RISC are 1)It comes with smaller die size, 2) it takes a shorter development time, and 3) it has a higher performance [29]. This in turn reduces the cost of ARM processors. ARM is compatible with most commonly used operating systems i.e. Linux, Windows, Symbian, Palm, and Android.

Cortex A, Cortex R and Cortex M are the series of ARM Cortex Processors [30]. ARM Cortex A-Application Processors are designed for application use. It consists of the 32-bit ARM Cortex-A5/A7/A8/A9/A12/A15/A17 MPCore, and the 64-bit ARM CortexA53/A57/A72. ARM Cortex R-Real-time Embedded Processors developed for real-time embedded applications such as automotive braking system and power train solutions. The last series is ARM Cortex M-Embedded Processors which is intended for microcontrollers like smart sensors that demand low power consumption and fast interrupt management.

Cortex-A72, for instance, is one of the processors used by RK3399. Cortex-A72 processor [31] is based on ARMv8-A, latest ARM version, architecture with high-performance, and low-power processor. It includes supports for AArch32 and AArch64 Execution states. In a single processor device, it may have one to four cores with L1 and L2 cache subsystems. Features supported by Cortex-A722 are Advanced Single Instruction Multiple Data (SIMD) operations, floating-point operations, and optional Cryptography Extension [31].

## 2.14 Firefly-RK3399

Firefly-RK3399 is a single board computer developed and delivered by Firefly and LovePi [32]. It is powered by Rockchip's RK3399 SoC, shown in figure 2-3 with red frame labeled with number 1, which is low power and high-performance processor. The processor can be used for high performance computation in mobile devices, IoT devices, and digital multimedia applications. RK3399 [33] has 64-bit ARMv8 architecture with dual 2.0GHz CPU clusters. The cluster consists of dual-core Cortex-A72 and quad-core Cortex-A53 with separate NEON coprocessor [33]. The performance of the new core shows 100% enhancement when compared to old Cortex- A15/A17/A57 core designs [32].

Additionally, RK3399 is equipped with ARM Mali T-864 GPU. GPU is a graphical processing unit with high parallel programmable processor [34]. It can be used for processing complex problems. RK3399 supports image processor and multi-format video decoders. Moreover, the GPU embedded in RK3399 enables it to support OpenGL, OpenCL and DirectX [33]. The GPU of Firefly-RK3399 makes it suitable for computer vision, Virtual Reality (VR), 4K 3D rendering, and gaming applications. Besides, the two MIPI-CSI interfaces capture up to 800Megapixels and can concurrently process two input streams [35]. The memory bandwidth of Firefly-RK3399 is up to 4GB DDR3 RAM and a hard disk storage space of up to 128GB. In the figure 2-3 the RAM and hard disk are labeled with number 3 and 4 respectively.

The communication for Firefly-RK3399 can either via Ethernet, Wi-Fi or Bluetooth technologies. There are two USB 3.0 channels with Type A and Type C and four USB 2.0 channels. For displaying HDMI 2.0 interface can be utilized. From figure 2-3 it can be depicted that multiple IO (input output) interfaces like MIPI-DSI, PCI-Express, M.2, and DP 1.2 are available. Android 6.0 and Ubuntu 16.04 are the operating systems supported by Firefly-RK3399. Dual-boot of these two systems is also allowed.

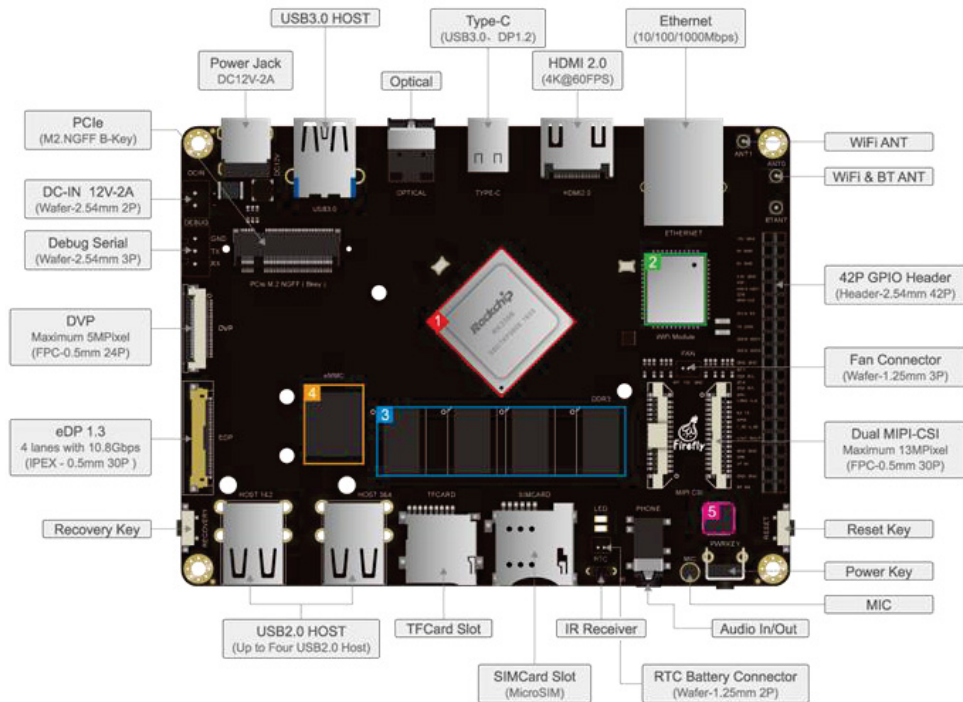


Figure 2-3 Firefly-RK3399 board components [36]

### 3 RELATED WORK

This section presents the previous works done that become a motivation for our thesis in several contexts. We presented researches that implemented a Docker container on small embedded devices. The performance evaluation of Docker on Embedded devices as well.

In [26] authors demonstrated an architecture for Edge cloud PaaS. They propose the architecture to meet the demand for infrastructure between IoT devices and data center clouds. Edge cloud is what they referred as multi-cloud platforms built from a range of nodes from data centers to small devices. The aim was to study the suitability and feasibility of container technology and container cluster management for Edge clouds built on single-board device clusters. Docker is used as a container technology.

The research work comprises implementation, experimentation and conceptual indication about key components of Edge cloud PaaS like orchestration. The result from the implementation shows that by using Docker and cluster management technology on Raspberry Pi we can have cost-efficient, low power consuming, and robust Edge cloud PaaS. Furthermore, they evaluated the architecture by using benchmarking tools. Their use case was a health care application where health status sensing devices are integrated on Raspberry Pi B. According to their finding, Raspberry Pi has the lowest power consumption compared with Marvell Kirkwood, MK802, Intel Atom 330, and dual core G620 Pentium.

The paper [37] presents a deployment of real-time emotion recognition application on IoT device, i.e. Raspberry Pi 2 (900 MHz ARM CPU and 4 GPU). The authors implemented an AI application that can recognize emotions from facial images. They used Active Shape Model (ASM) for extracting facial points and AdaBoost classifier. The image is captured from the webcam in real-time and then processed in AI application that is deployed on Raspberry Pi. They performed a test and compare their result with AI models deployed on Intel architectures. From their findings, they achieved a maximum recognition accuracy, which is 95% with an average execution time of 120ms.

A research in [38] also showed the implementation of motion detection on Raspberry Pi 2 (700 MHz ARM CPU). Raspberry Pi is connected to webcam to capture images. Data is processed locally. If an object is detected a notification will be sent and the image will be stored on remote FTP server. The authors emphasized the advantage small, portable and cheap devices like Raspberry Pi in distributed large-scale applications of IoT.

Another research direction is [39] which presents performance evaluation of container technologies on IoT devices. The authors investigated the performance overhead of Docker on Raspberry Pi 2. They performed synthetic benchmarking to measure CPU, Memory, Disk I/O, and Network throughput. In addition, they measure the performance of database application and response time of Apache request. The result shows that Docker has negligible overhead.

In a similar vein, authors in [40] perform several synthetic and application benchmarking on SoC devices. The experiment is conducted to study the performance overhead of Docker on Raspberry Pi B+ and Raspberry Pi 2. These are two different version of Raspberry Pi where the latter is the latest generation. Significant overhead is brought by the old version i.e.



Raspberry Pi B+. On the other hand, the result of Raspberry Pi is comparable to the performance evaluation of Linux containers on high-end servers in [41].

In [42] author provide performance evaluation of multiple container instances on low-power devices. According to their finding, containers has negligible impact on the performance even if multiple container instances are running. A tradeoff can be found between performance and power consumption for huge workloads.

Related researches in [43] are done to study the resource utilization of OS- level virtualization technologies on SoC computers. The research shows that LXC and Docker have similar processing performance as non-virtualized system. However, the isolation mechanism is still insufficient. The experiment shows that, when compared to Docker, LXC has stable and better performance on I/O bound applications on SoC devices.

Researches has been done to bring the computation from cloud to edge. Architectures are suggested using Docker container for resource optimization and hosting IoT applications. However, these researches did not address the detail implementation of containerization of applications in Docker on IoT devices. Although, researches have been done to study the performance of Docker on SoC devices, most researchers focus on Raspberry Pi only. They did not explore the performance of Docker on powerful IoT device, Firefly-RK3399.

## 4 METHODOLOGY

### 4.1 Overall Concept

In this section we will describe the methodology that we applied to approach the problem. The selection of methods and their rationales together with their detail technical descriptions are included in this chapter.

IoT is now connecting things. Data from sensors, RFID tags and actuators are processed in the cloud. However, transporting all data into the cloud requires a large amount of bandwidth. Moreover, the distance between sensors and cloud server causes a delay in response time. Therefore, there is a need for infrastructure between IoT and Cloud that can offload some of the computing tasks. Once we identified the gap, we should choose a method to approach the problem. We begin our study by reviewing literatures on our subject matter. We dig into different mechanisms used to provide computing resource near sensors and actuator. After extensive research, we reach into a conclusion that Edge computing solves this problem with the provision of computing resources within proximity of the sensors and actuators[27] [44]. The hardware resource can be provided using energy constrained Embedded devices or mobile devices. Some or all the computation load can be offloaded by edge devices which are resource constrained yet movable and distributed at the edge of the network.

In this thesis, we proposed a **system** that can demonstrate the edge computing infrastructure on embedded device. We will use single-board system on chip embedded device. The aim is for the computation of AI model near the sensors and actuators with less cost of bandwidth and latency. We applied containerization [7] to ensure the portability and interoperability of the AI applications in a distributed edge cloud architecture.

The overall methodology is to validate the **system** by a “Demonstrator”. We demonstrate the capabilities of the **system** by implementing a Demonstrator. The Demonstrator explains how to implement and configure the **system**. It also shows the feasibility of the **system** and contribute the understanding on how to implement the **system**, its real-world capabilities and its real-world performance.

The requirement and features of the Demonstrator is considered as:

- An implementation, i.e. hard- and software running
- Can be shown and execute main functions (not all)
- It's not a product; it's prototype
- Must not comprise all features of the intended system, but the relevant ones
- Should show how the components of the system interact with each other
- May need some manual interference from user
- Enables the design/developer to understand the difficulties of implementing the system
- Enables the developer to gain knowledge about the system, its technologies and capabilities.
- The knowledge should focus on the important parts of the system

The thesis is based on experimental research. It will apply the concept of a Demonstrator to show the feasibility and interworking of the subcomponents investigated in this thesis.



## 4.2 Demonstrator

In order to answer our research questions, we propose a model that can be called as a Demonstrator. The main aim of the Demonstrator is to show the feasibility and capability of computation AI applications on Embedded devices. It illustrates by deploying and running AI application on resource constrained IoT device. Moreover, the application will be containerized in Docker container. In our *system*, we focused on three scientific and engineering contributions.

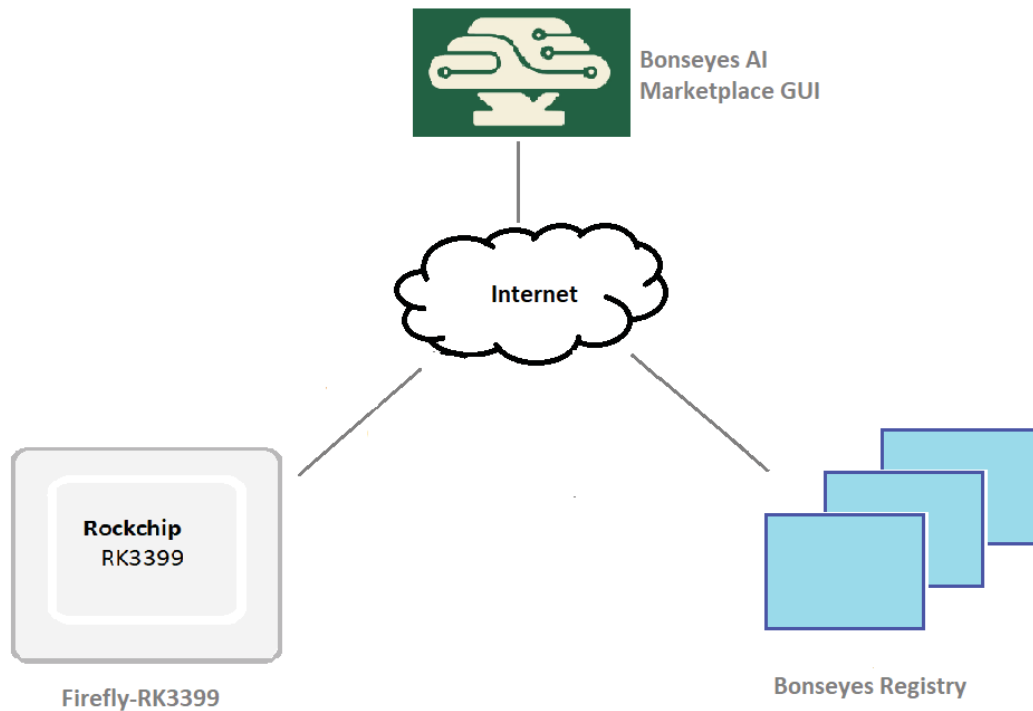
First, our *system* will bring the computation from high computing centralized cloud server to distributed edge devices that are located at the proximity of sensors and actuators. This will greatly reduce the cost of bandwidth and latency. We will show the capability of the system by running face recognition AI model which by nature demand fast response time.

Second, we will implement the *system* using a very powerful IoT device called Firefly-rk3399. As far as our knowledge, there is no literatures written about the implementation of Docker on Firefly-RK3399. Hence, we will provide the understanding and the capability of Firefly-rk3399 as an Edge device. Moreover, the capability and performance of this device in real-world application can be used by Bonseyes to integrate this device into a Bonseyes Ecosystem.

Third, we propose a method to move AI model between different architecture, from Intel x86 to ARM in our case. We design our Demonstrator to run the AI model inside a Docker container, the goal is to easily pack and ship the model with its dependencies. Therefore, successful implementation of our demonstrator will show the mechanism that enables the portability AI model among devices with different architecture.

## 4.3 Experiment Testbed

The experiment is done on the experiment testbed shown in figure 4-1. The experiment test bed comprises an embedded device, Bonseyes registry and Bonseyes GUI. The embedded device is Firefly-RK3399. The hardware and software specification of Firefly-RK3399 that is used for our experiment is described in table 4-1.



**Figure 4-1 Experiment test bed**

Firefly-RK3399 can be connected to the internet either in wired or wireless network. The registry is the Bonseyes repository. It is the repository where we store Docker images. The repository will also be used for transferring software application in a Docker image. Docker image will be built on the Firefly-RK3399 which has ARM architecture. Bonseyes's AI Marketplace GUI is a friendly user interface that contains information about available AI models in the Bonseyes repository. The image that is built on Firefly will be registered on the GUI for ease access of the user.

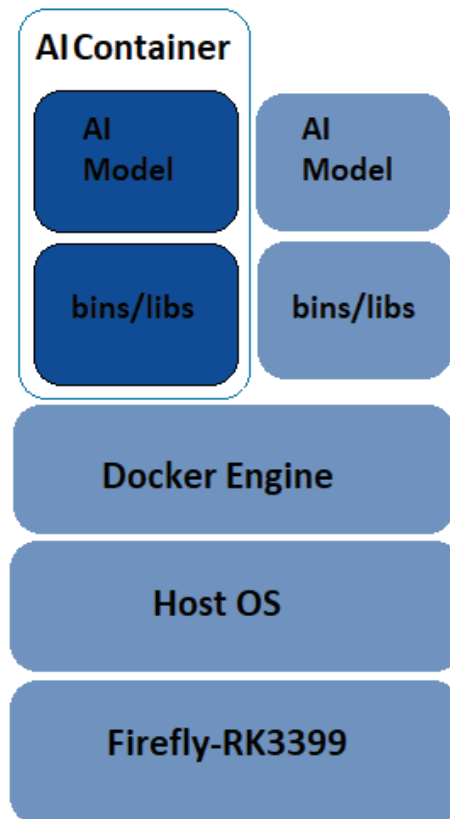
Specification	
Device name	Firefly-RK3399
System on Chip (SoC)	Rockchip RK3399
CPU	Six-Core ARM, Dual-Core Cortex-A72 and Quad-Core Cortex-A53 with separate NEON coprocessor 64-bit processor, up to 2.0GHz
GPU	ARM Mali-T860 MP4 Quad-Core GPU
Memory	4GB Dual-Channel DDR3
Storage	32GB High-Speed eMMC MicroSD (TF) Card Slot
Hardware Features	
Communication technology	Wireless- 2.4GHz/5GHz Dual-Band Wi-Fi Bluetooth 4.1 Ethernet- 10/100/1000Mbps Ethernet (Realtek RTL8211E)
Display	HDMI 2.0 (Type-A), DP 1.2(DisplayPort), MIPI-DSI and eDP 1.3 (4 lanes with 10.8Gbps)
Camera	13Mpixel or Dual 8Mpixel ISP, 2 x MIPI-CSI Camera Interface 5Mpixel DVP Camera Interface
USB	4 x USB2.0 HOST, 1 x USB3.0, and 1 x USB3.0 Type-C

Button	Reset, Power and, Recovery Button
Power supply	Battery - 5V Voltage and peak current of 3A
Software	
Operating system	Android 7.1, Android 6.0, Ubuntu 16.04, u-boot
Programming language support	C, C++, Java, Shell, Python etc.
Appearance	
Size	124mm x 93mm (8 Layer design, Immersion gold process)
Weight	89g (with cooling fan: 120g)

**Table 4-1**Experiment testbed description of Firefly-RK3399 [35]

## 4.4 Implementation of the Demonstrator

The main goal of the research is to move the computation of AI application to low power consuming IoT device, named Firefly-RK3399. The proposed model of our Demonstrator is shown in the figure 4-2. We wanted to demonstrate if it is possible and feasible to run Docker on Firefly-RK3399. The infrastructure built will be used for the deployment AI model. The implementation steps will be described in detail. The limitation and the problems encountered during the process will also be comprised in this paper. As the thesis is experimental we believe every finding is worthwhile mentioning.



**Figure 4-2** Model of Demonstrator

We start our implementation with fresh installation of operating system(OS) on Firefly-RK3399. The device supports both Android and Ubuntu operating systems. The aim of the thesis is to bring AI applications to Embedded devices. Later, we want to containerize the deployment an AI model by using Docker container. We choose Docker because it is very flexible and light-weighted [7][45] container technology. It also allows to pack and ship software application together with their dependencies from development to production environment. Docker uses the features of Linux kernel for isolation of its resources from the host OS. For this reason, Ubuntu is the appropriate OS for our host. After the device is flashed with Ubuntu 16.04, we update the system and configure the device's network connection.

Once we setup our system, we proceed to the installation of Docker on our device. But first we need to study the hardware and software requirements to run Docker. Additionally, we need to analyze the capability of Firefly-RK3399 platform for running Docker. If we make sure all the requirement is meet we will install Docker. At that point we will move to the next layer in our Demonstrator i.e. to deploy a face recognition AI model on a Docker container. We used an opensource AI model to show the computation of AI application inside Docker on Firefly-RK3399 platform. We will describe the technical details of each steps in the following subsections. But here are the general steps that we follow for the implementation:

- Understand the requirements to run Docker
- Analyze the capabilities of Firefly-RK3399 platform for running Docker
- Understand how to run and configure Docker on Firefly-RK3399 platform
- Analyze and choose techniques for implementation
- Connect the Firefly-RK3399 platform to a developer environment/network
- Configure Docker to run on the Firefly-RK3399 platform
- Conduct measurement on the performance of Docker on Firefly-RK3399

#### **4.4.1 Environment Setup of Firefly-RK3399**

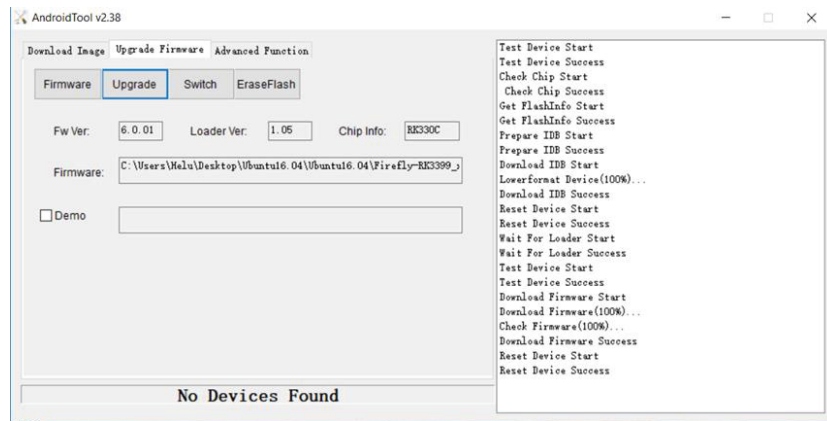
Before starting the implementation, we should setup our working environment. Firefly-RK3399 comes with Android OS when we first bought it. Since we need to work on Ubuntu, we will flash the device with Ubuntu16.04. To flash Ubuntu image on the board there are some software applications that should be downloaded and installed. The flashing process can be done either on Windows or Ubuntu OS. The requirement depends on the platform choice. Here, we will show how it can be accomplished on Windows 10. The installation process is described below.

First and for most the image, Ubuntu16.04 should be downloaded. We downloaded the image, Firefly-RK3399\_xubuntu1604\_201711301130.img, from the official t-firefly website [35] which is available at [46]. Baidu account is required to access the image file. Then we need to download and install RK USB driver, DriverAssitant\_v4.5. The installation is trivial, double click on DriverInstall.exec file will install the driver for Firefly-RK3399. The next software required is RK Android Tool, which is used to flash the image from our computer into Firefly board. We downloaded and installed, AndroidTool\_Release\_v2.38. The software's default language is Chinese. To change the language to English, we open config.ini (written in English) with text editor and set the value of "selected=2".

Now it is time to start the board and connect it to our computer. The process is stated as follows:

- Connect the power adapter to the board.
- Using USB Type-C cable connect the board to a computer
- Long press and hold RECOVERY key
- Short press RESET key
- After two seconds release RECOVERY key

The computer will prompt that a new device is detected. When we run RK Android Tool and will automatically detect the device. Then we go to the upgrade Firmware tab and click the firmware button and browse to the image file we downloaded before. Then clicking the Upgrade button will flash the image to the board, as shown in figure 4-3.



**Figure 4-3 Flash operating system into Firefly-RK3399 board using Android Tool**

During the process the detail information will be displayed on the right side of the window. When the process ends it will release the device as shown in the figure 4-3 and displays No Device Found.

After the successful installation of host OS, the wireless network connection needs to be configured. We power on the board and connect the board to a display monitor using HDMI cable, USB mouse and keyboard. Then we configure the SSID, protocol, security type, username and password based on Eduroam Wi-Fi configuration information. We also registered the mac address of Firefly-RK3399 in a BTH network to have a static IP address.

#### **4.4.2 Running Docker**

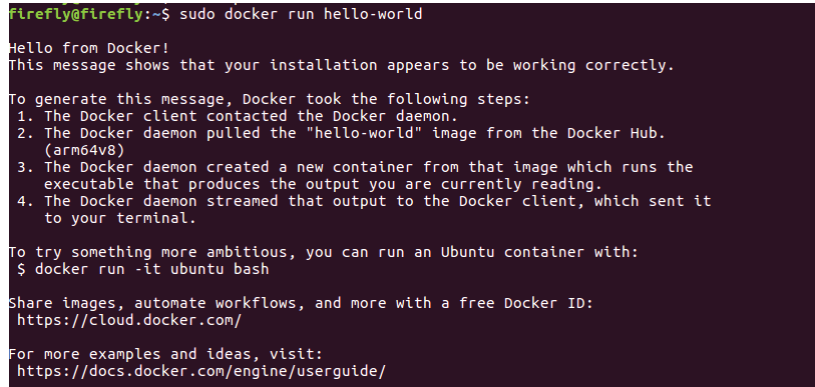
Since it is suitable for developers and small teams looking to get started with Docker and experimenting with container-based apps, we will install Docker Community Edition (CE) [47]. One of the prerequisites for installing Docker CE on Ubuntu is to have 64-bit Ubuntu version of Artful 17.10 (Docker CE 17.11 Edge and higher only), Xenial 16.04 LTS (Long Term Support), or Trusty 14.04 LTS. This requirement is already satisfied as we installed the Xenial 16.04 (LTS) version of Ubuntu.

Firefly-RK3399 has ARM architecture which fulfills the second requirement to install Docker. Hardware architectures supported by Docker are x86\_64, armhf, s390x (IBM Z), and ppc64le (IBM Power).

Now it is evident that we can install Docker CE on Firefly-RK3399. We install Docker by downloading the .deb file [48]. Docker CE has two update channels stable and edge. We choose stable channel because it gives reliable updates.

```
$ sudo dpkg -i /home/firefly/Downloads/Docker-ce_17.09.0_ce-0_ubuntu_arm64.deb
```

The Docker daemon starts automatically. We verified Docker is correctly installed by running hello-world image:



```
firefly@firefly:~$ sudo docker run hello-world
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (arm64v8)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://cloud.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/engine/userguide/
```

**Figure 4-4 Hello-world Docker container**

We should have a root privilege to access the Docker daemon otherwise we can only access it using “sudo”. This is because Docker daemon binds to Unix socket which is owned by root user. To avoid typing password when using Docker command, we created a Unix group known as Docker and add users to it. When the Docker daemon starts, it makes the ownership of the Unix socket read/writable by the Docker group [49].

We create a group called Docker and add user i.e. firefly to the group:

```
$ sudo groupadd Docker
```

```
$ sudo usermod -aG Docker firefly
```

The system is rebooted for changes to take effect.

### 4.4.3 OpenFace

We used an opensource face recognition AI model called OpenFace [50] to demonstrate the containerization of AI models on Firefly-RK3399. The details of OpenFace development and algorithm description is beyond the scope this thesis. However, general description and workflows will be discussed to give some insights.

OpenFace is an open source face recognition AI model, created by Brandon Amos, Bartosz Ludwiczuk, and Mahadev Satyanarayanan. It is a Python and Torch implementation with deep neural networks and based on the FaceNet [50]. FaceNet [50] is a Unified Embedding for Face Recognition and Clustering at Google.

Torch [51] is a scientific computing framework for LuaJIT. It offers a wide support for machine learning algorithms that puts GPUs first. It uses scripting language, LuaJIT and an underlying C/CUDA implementation. Torch’s deep learning framework is used for executing very complex tensor computation and for constructing deep neural networks.

The logical OpenFace project structure is shown in figure 4-5. The process of OpenFace face recognition which is described in [50] [52] can be summarized as follows:

1. Detect the face- Histogram of Oriented Gradients (HOG) algorithm [53] is used to detect the face.
2. Posing and projecting faces- to identify the same image from different poses and projections a face landmark estimation algorithm is implemented. It will come up with 68 specific points called landmarks that exist on every face the top of the chin, the outside edge of each eye, the inner edge of each eyebrow, etc. Then the image is rotated, scaled and sheared (affine transformation) to make the eyes and mouth as centered as possible.
3. Encoding faces- approximately the same 128 measurements called embedding are generated for each person from different pictures of the person. Deep Convolutional Neural Network is trained to generate these measurements for each face.
4. Finding the person's name from the encoding- SVM [54] classifier is trained to take in the measurements from a new image and search for closest match from the known persons. Then the output of the classifier is the name of the person.

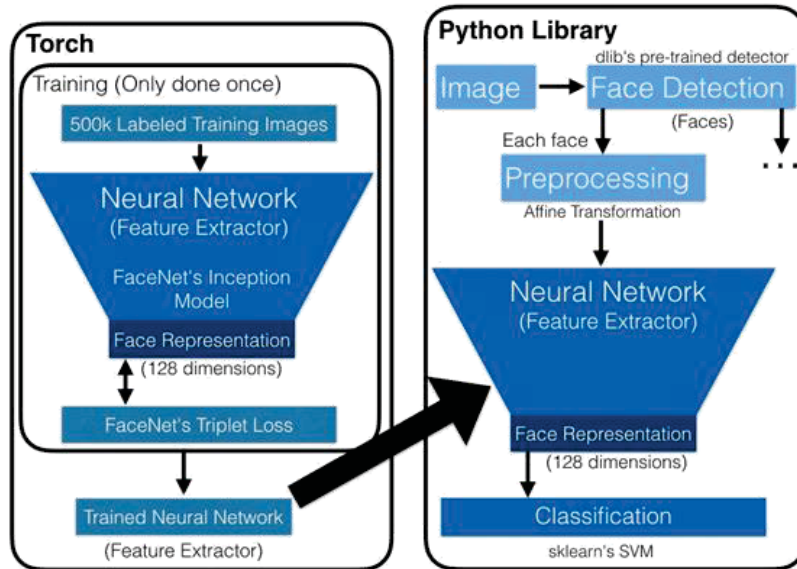


Figure 4-5 OpenFace project structure [50]

#### 4.4.4 Containerization of OpenFace on Firefly-RK3399

Docker's main purpose is to pack applications in a container so that it is ready to be shipped [55]. The basic idea is to move around the software application at different environments, particularly between development and production environments. Our main objective is to bring AI applications into embedded devices and pack them in a container to ensure their portability.

OpenFace has already built Docker image called *bamos/OpenFace* which is publicly available at Docker hub. However, we cannot use this image because it uses Ubuntu14.04 intel x86 base image which has a binary format different from ARM architectures. Thus, we must build our own Docker image that can be interpreted by ARM architecture.

The process to build a Docker image for OpenFace starts from pulling the available Docker image from Docker hub. The Docker image, *bamos/OpenFace*, should be tested on x86 intel



architecture and validated if it gives the expected output. Similarly, we will do the same testing and validation after we build OpenFace Docker image for Firefly-RK3399.

Dockerfile is used to create Docker images, as mention in section 2.8. The Dockerfile contains commands to install and run a given software and all its dependencies and build the image all at once. This method has its own advantage i.e. the Dockerfile can later be used by anyone who wants to build the same application on the same platform. There is also another approach to create a Docker image. The Docker image can be build layer by layer inside a base image container. We select and start base image and every dependency of the application will be containerized inside the base image container. Each application should be tested that it is containerized and works properly before changes is saved on the top of base image layer. We prefer this method as OpenFace has a lot of dependencies and we want the containerization process to go accurately. There is a case that Dockerfile builds image successfully, but the Docker image might not work as expected. It is difficult to trace the failure. However, this approach is easier to debug errors as we validate each step. Afterwards, we can create the Dockerfile using the verified commands in the containerization steps for later use.

The selection of the base image is very important in the building process. The base image must be compatible to the underlying hardware architecture and should provide support for the application and its dependencies. We selected **arm64v8/ubuntu** Docker image as a base image which meets the requirement for our device. Firefly-RK3399 has ARM 64bits version 8 architecture. In addition, it provides the support for OpenFace AI model.

The process to generate a Docker image for OpenFace will be described as follows:

First open a terminal and pull Ubuntu 16.04 Docker image from Docker hub on Firefly-RK3399

```
$ docker pull arm64v8/ubuntu
```

We run the image as a daemon to keep it running in background. The container can be name by using *--name* option, we named it as *ubuntu-base*. To get inside the running container we can use *docker exec* command in bash by referring to the container name.

```
$ docker run --name ubuntu-base -i -d arm64v8/ubuntu
```

```
$ sudo docker exec -it ubuntu-base bash
```

Now we are inside the container, system update needs to be done since it is a fresh installation

```
# apt-get update
```

The next step is to install all dependencies of OpenFace [56]. This includes python-opencv, dlib, and Torch. We verify each software is containerized properly. The change will be saved to ubuntu-opencv-dlib-torch using Docker commit command.

```
$docker commit <container ID> ubuntu-opencv-dlib-torch
```

Container ID can be found using **docker ps** command



Since we don't need the previous container, we will kill it and execute the newly saved image i.e. ubuntu-opencv-dlib-torch.

```
$ docker stop ubuntu-base
```

```
$ docker run --name OpenFace-test -i -d ubuntu-opencv-dlib-torch
```

```
$ sudo docker exec -it OpenFace-test bash
```

It is time to put OpenFace into the container. The source can either be copied from host OS or can be cloned inside the container. To copy we open a new terminal and execute the command:

```
$ docker copy <host-path-to-source-file> <containerid>:<container-destination-path>
```

The next step is to train the model as mentioned in [52] and we commit the change to *openface-trained* image. We will test it inside the container. After the image is trained and tested, we push the image into the Bonseyes repository using the following commands.

```
$ docker login bonseyes.zhdk.cloud.switch.ch/
```

```
$ docker tag <imageid> bonseyes.zhdk.cloud.switch.ch/trained-openface: arm
```

```
$ docker push bonseyes.zhdk.cloud.switch.ch/trained-openface:arm
```

Finally, to test our image we pull it back from Bonseyes repository and execute it. The Docker pull command can be found on the Bonseyes GUI AI marketplace under OpenFace model.

```
$ docker pull bonseyes.zhdk.cloud.switch.ch/trained-openface:arm
```

To run the image the arguments are the classifier, the pkl file and the image of the person we want to recognize. Infer is command mode which tells the classifier.py to do the recognition. The output will give us the confidence in which a person is recognized, usually it will be more than 0.8 where the maximum is 1. The program gives confidence even for unknown person. If the number is less than 0.6 it means that the person in the image is unknown. We used three images for training the model in which Steve Job's images are one of them.

```
$ docker run -it bonseyes.zhdk.cloud.switch.ch/trained-openface:arm bash -l -c  
'/home/project/openface/demos/classifier.py infer /home/project/openface/generated-  
embeddings/classifier.pkl /home/project/openface/test_images/steve-  
jobs/steve_jobs8.png'
```

At this point we managed to build the Docker image for OpenFace on Firefly-RK3399, hence we move OpenFace from x86 into ARM container.

## 4.5 Performance Evaluation

After the completion of configuration of Docker, we will investigate its performance on Firefly-RK3399. We did literature review to get understanding about the different methods of performance analysis. We then select benchmarking tools to measure the performance. The measurement and analysis are done both on host OS and Docker. The result of Docker will be compared with the host OS. In theory, Docker is very light-weighted virtualization technology. Hence, we hypothesize that it does not have significant performance overhead when compared to the underlying host. However, we will determine the performance difference by doing an experiment. The aim of the experiment is to quantify the Docker's overhead with respect to CPU and Memory on Firefly-RK3399. In addition, this paper can be used as an input to present the performance evaluation of Docker on Firefly-RK3399 which, in our knowledge, has not been written before. The methodology of the experiment and the measurement tools will be described on the following subsequent sections.

## 4.6 Performance Evaluation Methodology

The experiment will be conducted by applying load into the system under test. The system has two test cases i.e. system with Docker and system without Docker. We define a random variable and statistical objects that are of interest for the performance evaluation. The random variable  $[X]$  will be the synthetic load. The performance metrics are CPU and memory. We used standard tools to apply load and measure aforementioned metrics. The measurement will be repeated to verify the result. Then we will compute Expectation ( $E[X]$ ), Standard deviation and confidence interval. The statistics will be used to study the behavior of our test cases under the variable load.

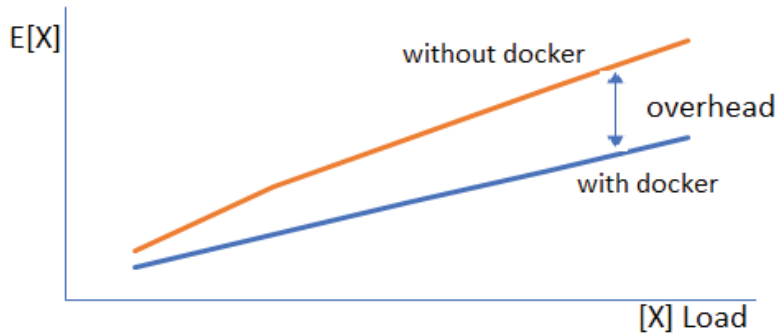


Figure 4-6 Sample observation method of an experiment

## 4.7 Experiment Setup

The environment where the experiment will be conducted is shown in figure 4-7. On the top of Firefly-RK3399 infrastructure we setup two test cases for CPU and Memory benchmarking. We used Sysbench 0.4.12 for CPU benchmarking. STREAM 5.10 and STREAM2 5.10 are used as Memory benchmarking. The description of the tools is stated in section 4.8 and 4.9. In Test Case 1, the performance of host operating system i.e. without virtualization layer will be measured. The tools are installed from a package and by compiling a source file. The second Test Case setup comprises of a Docker virtualization

layer. Hence, the measurement will be inside a Docker container. For this case, a docker image is customized for each tool which is compatible on Firefly-RK3399.

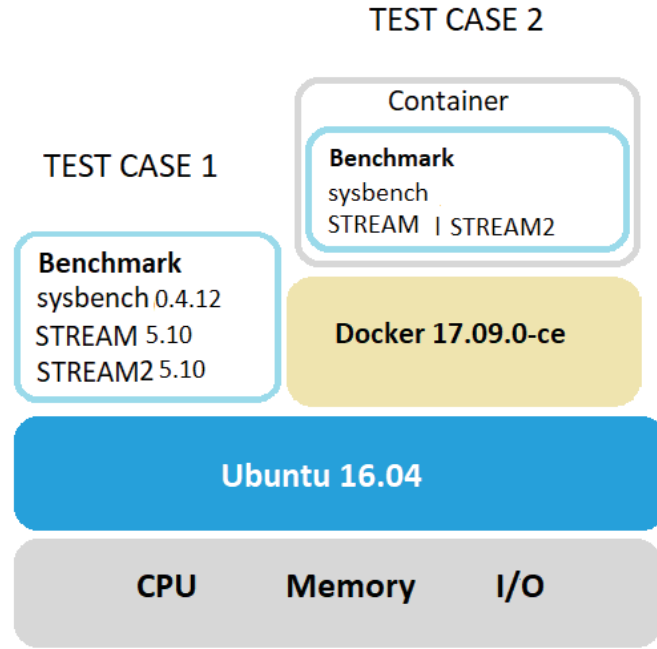


Figure 4-7 Experiment setup

## 4.8 CPU Performance Evaluation

The objective of measuring CPU is to study the amount of time consumed by Docker, in relative to host OS, to compute a certain amount of instructions. From our extensive literature review, we selected a well-known standard tool to measure the CPU performance i.e. Sysbench. Here, we will describe how the tool operates and later the output from the tool will be presented and analyzed in result section. In both cases, a synthetic workload is applied to CPU to measure the execution time. The output from the tool does not reflect the overall performance of a given system but it does reflect the performance of a dedicated system for computing a complex mathematical task.

### 4.8.1 SYSBENCH

Sysbench is LuaJIT based scriptable multi-threaded benchmark tool. It can be used to create arbitrarily complex workload [57]. It has a command line options to set the number of threads (*--num-threads*) and input number of requests (*--max-requests*). The CPU intensive workload consists of computing prime numbers by doing standard division of the input number by all numbers between 2 and the square root of the number. The sample output is shown in figure 4-8. We are interested in the total time which is the time it takes to compute the maximum prime number.

```

firefly@firefly:~$ sysbench --test=cpu --num-threads=4 --max-requests=10000 run
sysbench 0.4.12: multi-threaded system evaluation benchmark

Running the test with following options:
Number of threads: 4

Doing CPU performance benchmark

Threads started!
Done.

Maximum prime number checked in CPU test: 10000

Test execution summary:
total time: 1.8686s
total number of events: 10000
total time taken by event execution: 7.4640
per-request statistics:
  min: 0.55ms
  avg: 0.75ms
  max: 2.60ms
  approx. 95 percentile: 1.05ms

Threads fairness:
events (avg/stddev): 2500.0000/732.01
execution time (avg/stddev): 1.8660/0.00

```

Figure 4-8 Sysbench sample benchmarking output

## 4.9 Memory Performance Evaluation

In memory performance evaluation, we measure the access speed by requesting a block of memory. Different access speeds can be observed for the different sizes of blocks. We will use STREAM and STREAM2 memory benchmarking tools.

### 4.9.1 STREAM

STREAM is a benchmarking tool that measures sustainable memory bandwidth (in MB/s) [58]. It calculates the computation rate for simple vector kernels. Table 4-2 illustrates the four set of operations for the vector kernels performed by STREAM. The array size must be at least four times the size of the sum of all caches in the memory. We computed the change in memory bandwidth using equation 4.1.

$$\%change = \frac{M_{docker} - M_{native}}{M_{native}} * 100 \quad (4.1)$$

Where:  $M_{Docker}$ - memory bandwidth of Docker platform and

$M_{native}$ - memory bandwidth of native platform

Operation type	Kernel	Bytes/iter	Flops/iter
COPY	$a(i) = b(i)$	16	0
SCALE	$a(i) = q*b(i)$	16	1
SUM	$a(i) = b(i) + c(i)$	24	1
TRAID	$a(i) = b(i) + q*c(i)$	24	2

Table 4-2 STREAM operations [58]

### 4.9.2 STREAM2

STREAM2 [59] is based on STREAM but it uses a different set of vectors. It also has four kernel operations FILL, COPY, DAXPY and SUM. FILL fills a given vector with a constant

number. DAXPY is like TRAIID except it writes the output by overwriting one of the input vectors. SUM is also a similar operation, but the input is a single vector.

## 5 RESULTS AND ANALYSIS

This section presents the results obtained from several experiments on the test environment. Besides, the outcome of moving OpenFace AI model from Docker repository to ARM board will be discussed.

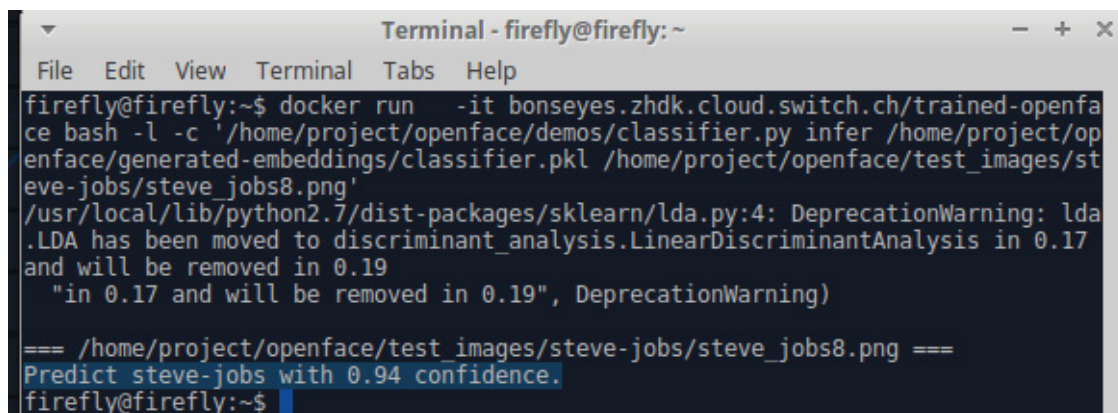
The performance analysis comprises experiments by combining multiple scenarios and tools on Native and Docker platforms on Firefly-RK3399. The selected metrics are CPU and Memory performance. In all the experiments, each measurement is repeated for 20 times. The first measurement is not considered in the result due to transient state. Once data is collected, we will analyze by applying statistical objects. The resulting values will be given in a table. Moreover, we generate plots for graphical representation of the results.

There are two main results obtained from this thesis work. The primary result of this thesis is the portability of AI model in Docker on firefly-RK3399 which we will describe first. After that, we will present the second result which is performance analysis. The analysis of each performance metrics will be presented when various loads and tools are applied.

### 5.1 Portability of AI Model

The portability of Docker image is experimented by our demonstrator. The result from the implementation of the demonstrator will be discussed here. First, the demonstrator shows how Docker Engine can be installed and configured in Firefly-RK3399. This is one of the research questions if we recall from section 1.4. Then we demonstrated by running an AI model on Docker container.

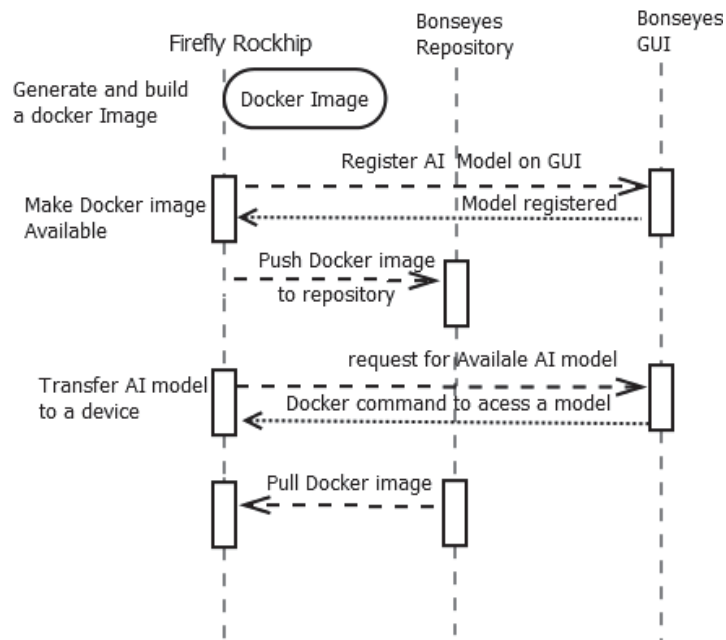
The outcome of the demonstrator is that we can transfer OpenFace which is x86 based Docker image into ARM based Firefly-RK3399 board. We cannot directly run the Docker image because ARM and x86 have different binary format. Hence a new Docker image is built which is compatible for ARM architecture. The image is built, tested and pushed into the Bonseyes repository. It consists of OpenFace and all its dependencies. From figure 5-1 it is clearly shown that output of OpenFace container predicts a face with 94% confidence which is confidence interval of OpenFace if the face is known and recognized.

A terminal window titled "Terminal - firefly@firefly: ~" with a menu bar (File, Edit, View, Terminal, Tabs, Help). The terminal shows a Docker command being executed: `firefly@firefly:~$ docker run -it bonseyes.zhdk.cloud.switch.ch/trained-openface bash -l -c '/home/project/openface/demos/classifier.py infer /home/project/openface/generated-embeddings/classifier.pkl /home/project/openface/test_images/steve-jobs/steve_jobs8.png'`. The output includes a deprecation warning for LDA and a final prediction: `=== /home/project/openface/test_images/steve-jobs/steve_jobs8.png === Predict steve-jobs with 0.94 confidence.` The prompt `firefly@firefly:~$` is visible at the bottom.

```
Terminal - firefly@firefly: ~
File Edit View Terminal Tabs Help
firefly@firefly:~$ docker run -it bonseyes.zhdk.cloud.switch.ch/trained-openface bash -l -c '/home/project/openface/demos/classifier.py infer /home/project/openface/generated-embeddings/classifier.pkl /home/project/openface/test_images/steve-jobs/steve_jobs8.png'
/usr/local/lib/python2.7/dist-packages/sklearn/lda.py:4: DeprecationWarning: lda.LDA has been moved to discriminant_analysis.LinearDiscriminantAnalysis in 0.17 and will be removed in 0.19
  "in 0.17 and will be removed in 0.19", DeprecationWarning)

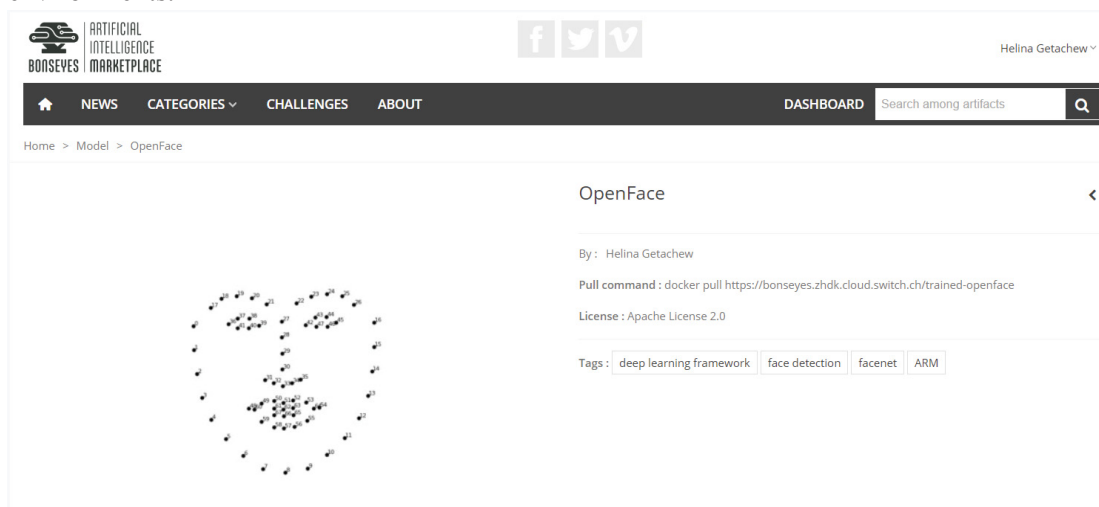
=== /home/project/openface/test_images/steve-jobs/steve_jobs8.png ===
Predict steve-jobs with 0.94 confidence.
firefly@firefly:~$
```

Figure 5-1 Output of OpenFace on Firefly



**Figure 5-2 Sequence diagram of transferring AI model in a Docker image**

Figure 5-2 shows the process of transferring the AI model in a Docker container. The built OpenFace Docker image should be stored either on a public or private repository to make it available for legitimate users. OpenFace is first registered in GUI to signify that the Docker image for OpenFace AI model is available in Bonseyes repository. Short description of OpenFace and how we run the model in a container is also included in the marketplace GUI. In our thesis, we did not implement a separate GUI rather a Bonseyes marketplace GUI is considered. The AI marketplace, as stated in section 2.1, is the place where all available AI models are advertised but the access to Docker image is allowed only to authorized users. The next step is to push the Docker image to a Bonseyes repository. Therefore, anyone who is working on a Firefly-RK3399 can transfer AI models by visiting the marketplace for available AI models and get the Docker command for that specific image. As shown in figure 5-3 the Docker pull command is written on Bonseyes GUI, any legitimate user can use this command to pull the image to their device. In this way, AI models can be easily transferred in a Docker image between different working environments.



**Figure 5-3 OpenFace AI model for ARM in Bonseyes Marketplace GUI**



According to our result, we ensured the portability of OpenFace independent of the hardware architecture. It can be advantageous for developers to exchange their codes and runtime instances by applying this mechanism to build Docker images for different architectures. Therefore, developers working on different architectures can collaborate their work without worrying about how to set up their working environment for the application development process. Moreover, in the future end users can use applications independent of the hardware they have. For example, they can use the android mobile apps for iOS iPhones and run it inside a container.

It is also worth mentioning the challenge of the mobility of AI application in Docker containers. Building from Dockerfile requires a base image compatible for ARM architecture. Furthermore, all the version of dependencies of an application must be supported by the base image. Otherwise the built will not be successful. One challenge we come across is that there is not much Docker images available for ARM as there is for x86 in Docker hub. This is one of the reason we choose to build Docker image layer by layer. In this approach we pull only base image from Docker repository. We run the base image in a container and add layers of our applications. But this time we install all the application's dependencies as we are installing in host OS. It is stated in methodology section 4.4.4 that this method enabled us to pack and ship applications in a controlled way.

As we described from the beginning of this paper, installing Docker is not the only requirements to adopt application for different architectures. To build a Docker image for different architecture requires the possession of the hardware and a basic understanding on the requirements how to set up Docker environment and build Docker images. The process of containerization for ARM architecture is shown and written in this paper. The process might vary for different hardware architecture because the interpreter or compiler of binary format of one hardware is different from the other. For a developer who is not familiar with the hardware architecture, the process might not be easy and could be time consuming. But this paper can be used to create basic understanding and as a guideline on how to containerize and transfer an application using Docker images.

## **5.2 CPU Performance**

In this section, results of CPU performance are presented and analyzed when various synthetic workloads are applied.

### **5.2.1 SYSBENCH**

In this experiment, we used Sysbench benchmarking tool to measure the execution time of a synthetic load. While running the benchmark all applications are closed. The measurement is first performed in the Native and then in Docker platform. Each measurement is repeated 20 times and there is a 5 minutes gap between each iteration. For all measurements mean, standard deviation and 95% confidence interval is calculated. In the first scenario, Firefly's CPU execution time is measured in a single thread by varying the number of input load. The option "--max-requests" is used to increase the number of prime number to be calculated. We define the input load to be 10,000, 20,000, 30,000, and 40,000. The result is presented in table 5-1 and figure 5-4 (a). Firefly-RK3399 has 6 CPU core processors. Since Sysbench supports multithreading we can make use of all CPU cores. The number threads can be increased by using "--num-threads" option. We set the value of thread as 1, 2, 4 and 6. Table



5-3 shows a sample execution time for these thread numbers with input load of 10,000. We did not present output of 2 and 4 thread for other input loads as they have similar trend. Table 5-2 and figure 5-4 (b) shows the execution time of Docker and Native platforms for 6 CPU cores.

Input Load	Firefly Native			Firefly Docker		
	Mean	Standard deviation	95% CI Half-size	Mean	Standard deviation	95% CI Half-size
10,000	5.629	0.012	0.005	5.630	0.021	0.009
20,000	11.227	0.017	0.008	11.239	0.070	0.031
30,000	16.628	0.014	0.006	16.636	0.036	0.016
40,000	22.523	0.021	0.009	22.534	0.027	0.012

**Table 5-1 Mean, standard deviation and confidence interval for Execution time of Firefly's single CPU core in Native Docker platforms**

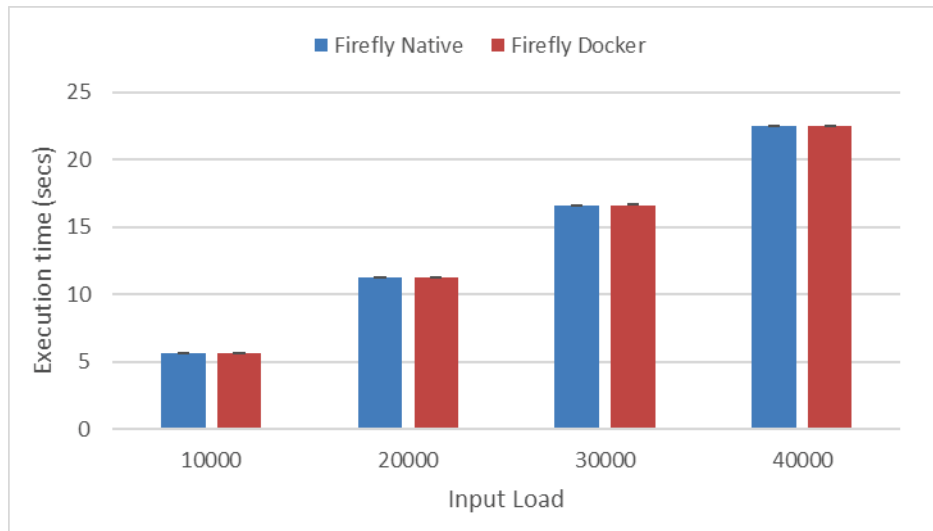
Input Load	Firefly Native			Firefly Docker		
	Mean	Standard Deviation	95% CI Half-size	Mean	Standard deviation	95% CI Half-size
10,000	1.424	0.024	0.011	1.455	0.029	0.013
20,000	2.851	0.099	0.043	2.924	0.125	0.055
30,000	4.297	0.175	0.077	4.352	0.122	0.053
40,000	5.582	0.143	0.063	5.742	0.132	0.058

**Table 5-2 Mean, standard deviation and confidence interval of Execution time of Firefly's six CPU cores in Native and Docker platforms**

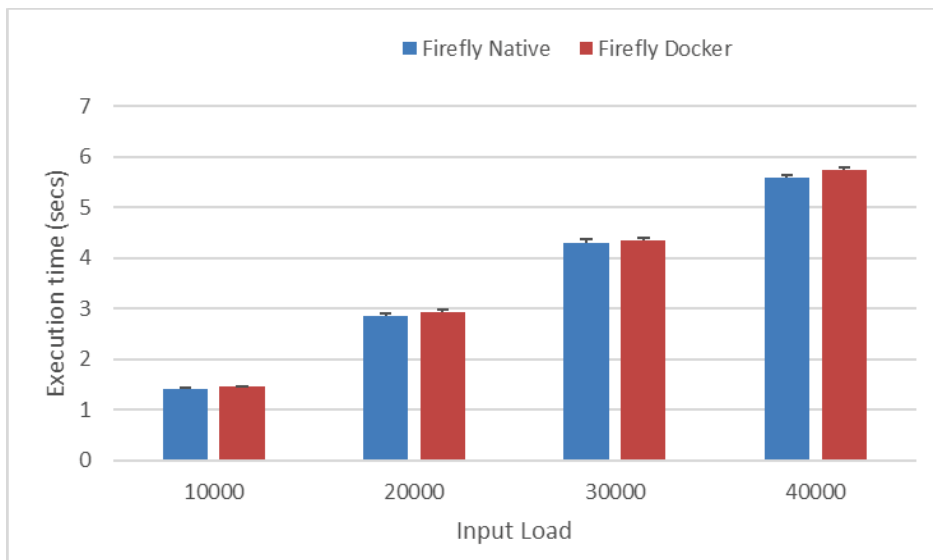
Number of Threads	Firefly Native			Firefly Docker		
	Mean	Standard Deviation	95% CI Half-size	Mean	Standard deviation	95% CI Half-size
1	5.629	0.012	0.005	5.630	0.021	0.009
2	2.852	0.024	0.011	2.864	0.028	0.012
4	1.872	0.011	0.005	1.871	0.013	0.006
6	1.424	0.024	0.011	1.455	0.029	0.013

**Table 5-3 Mean, standard deviation and confidence interval of Execution time of Firefly in Native and Docker platforms with different CPU cores with fixed input load**

From the result of the experiments, we can see the CPU performance of Docker with respect to Host (Native) OS. The overhead brought by Docker can be depicted from table 5-1. For input load of 10,000 the execution time overhead is 0.001secs (1ms). Maximum input load i.e. 40,000 on Docker brought 11ms delay. Figure 5-4 (a) also shows the result of this scenario. The errors bars and the standard deviation is very low which indicates the distribution of the measured values are close to the mean value. In the second scenario, we increased the number of threads. As shown in table 5-2 and figure 5-4 (b) the execution time decrease which is expected because it uses all CPU cores therefore it will take less time to execute the task. If we consider the overhead, Docker took 31ms, 76ms, 55ms and 160ms more than Native platform respective with the input loads. Generally, we observed the increase in load will affect the overhead both in single thread and 6 threads cases. But we cannot say they have a linear relationship. In table 5-3 the CPU execution time is shown for different threads the time decreases as number of threads increases. Increasing the number of threads above six doesn't make difference as the task will be queued anyways.



(a) Execution time of Sysbench in a single thread



(b) Execution time of Sysbench in 6 threads

**Figure 5-4 Execution time of Sysbench benchmark measured in a single CPU core and six CPU cores of Firefly-RK3399**

## 5.3 Memory Performance

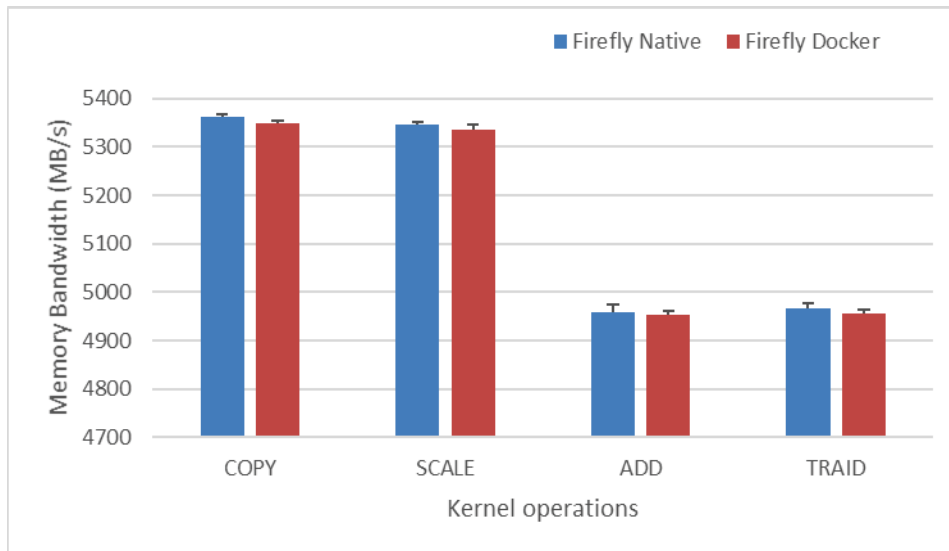
### 5.3.1 STREAM

The experiment is conducted to measure the memory bandwidth by doing four kernel operations. The measurement for each operation is repeated 20 times. The mean, standard deviation and the 95% confidence interval is calculated as shown in table 5-4. Additionally, the change between Docker and native is computed using equation 4.1. The rule of STREAM is that we should use an array size at least 4x the total available CPU cache. Firefly has both L1 and L2 cache with a total of 1.928MB. By default, STREAM's array size is 1 million elements that will occupy 8 million bytes per array, which is more than 4 times the size of all the available cache on the system ( $> 4 \times 1.928$ ). This will occupy a memory of 22.4MB of memory. But Firefly has 4GB of RAM so array size can be increased. Hence, we decided to take from the larger end array size for STREAM (64,000,000) and lower end for STREAM2 (2,000,000).

Operation Type	Firefly Native			Firefly Docker			Percentage decrease in Docker (%)
	Mean (MB/s)	Standard Deviation	95% CI Half-size	Mean (MB/s)	standard deviation	95% CI Half-size	
COPY	5361.49	12.78	5.60	5347.53	15.79	6.92	-0.26
SCALE	5345.35	15.22	6.67	5335.34	21.74	9.53	-0.18
ADD	4958.08	36.91	16.18	4954.03	14.47	6.34	-0.08
TRAID	4965.28	28.08	12.31	4956.90	16.79	7.36	-0.17

**Table 5-4 STREAM Memory bandwidth of kernel operations for Native and Docker platforms**

It can be clearly depicted from the bar graph in figure 5-5 that the Docker's memory bandwidth is very close that of native's. The percentage change column in table 5-4 tell us the maximum degradation occurs on COPY operation i.e. 0.26%.



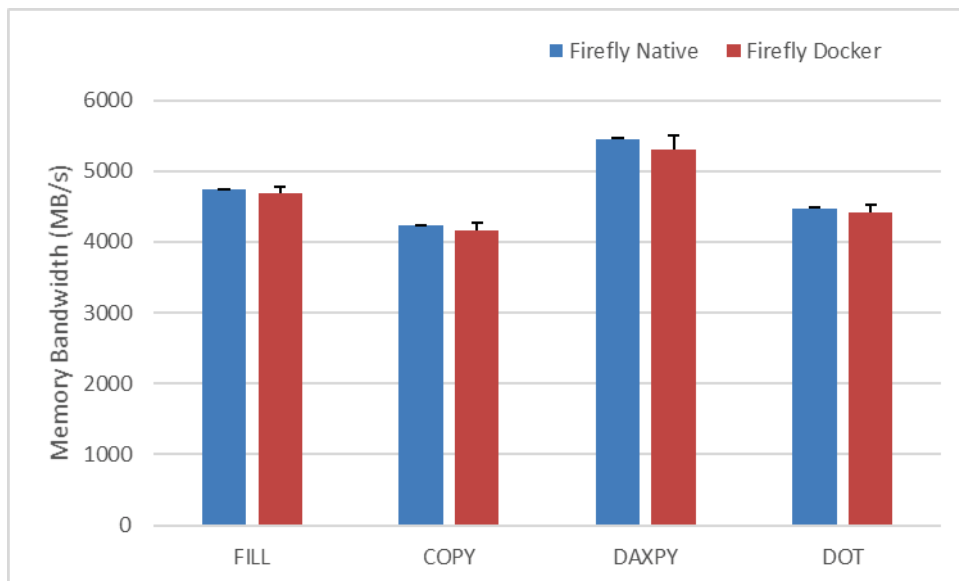
**Figure 5-5 STREAM Mean Memory bandwidth on COPY, SCALE, ADD and TRAID operations in Native and Docker Platforms**

### 5.3.2 STREAM2

Similarly, the memory bandwidth is computed using STREAM2. For consistency we used the same layout for the table and graph. As shown in table 5-5 2.73% decrease is observed at DAXPY operation (equivalent to STREAM's TRAIID operation). But it can be comprehended from figure 5-6 that, for other operations although the number is slightly increased when compared to STREAM it does not show much difference. From the output of STREAM and STREAM2, we can say that Docker performs almost as native platform.

Operation Type	Firefly Native			Firefly Docker			Percentage decrease in Docker (%)
	Mean (MB/s)	Standard Deviation	95% CI Half-size	Mean (MB/s)	standard deviation	95% CI Half-size	
FILL	4735.75	10.10	6.26	4693.16	131.06	81.23	-0.89
COPY	4226.79	26.16	16.21	4163.45	185.27	114.83	-1.49
DAXPY	5455.00	15.74	9.75	5306.04	304.85	188.95	-2.73
DOT	4477.38	16.16	10.02	4415.46	161.26	99.95	-1.38

**Table 5-5 STREAM2 Memory bandwidth of kernel operations for Native and Docker platforms**



**Figure 5-6 STREAM2 Mean Memory bandwidth on FILL, COPY, DAXPY and DOT operations in Native and Docker Platforms**

## 6 CONCLUSION AND FUTURE WORK

In this thesis work, we attempted to identify the methods for local inference of AI model on IoT device inside a Docker container. The aim was to bring the computation of AI applications to small embedded IoT devices. Additionally, we tried to find a method for containerizing AI model from different hardware platform.

This thesis dealt with designing, implementing, and testing an Edge computing concept that ensures the portability of AI model in a Docker image on Firefly-RK3399. After requirements to run Docker are studied and fulfilled, Docker is used to demonstrate the execution of AI model. We choose an OpenFace AI model that is based on x86 Intel architecture to demonstrate the portability of Docker image between different architectures. The containerization process is implemented so that the architecture is adopted from x86 to ARM architecture. OpenFace is trained using test images and the result is tested multiple times. From the finding of this thesis, OpenFace successfully executed on Firefly-RK3399 inside Docker container with an average confidence of 94%. Therefore, we deduce that this method can be used as one mechanism to move an AI model from x86 to ARM architecture on embedded device which runs full Linux OS. Furthermore, this thesis provides an understanding in making the Docker image available in repository and transferring this image from repository to another end device.

The performance evaluation of Docker on Firefly-RK3399 is another perspective studied in the thesis. Multiple synthetic load is applied to Docker container in a controlled environment. CPU and Memory performance evaluation is conducted to study the overhead of Docker when compared to native OS. CPU execution time of a given load in Docker yields comparable speed as native OS. Moreover, Docker had memory bandwidth of 3% less than native OS. With the scope of the data analyzed in this thesis, the performance overhead of Docker on Firefly-RK3399 in terms of CPU and Memory is very low.

### 6.1 Answering Research Questions

Here, we will try to answer our research questions.

1. How can Docker be installed, configured, and instantiated (run) on a Firefly-RK3399 IoT device such that it demonstrates that an AI model can be executed on it? What are the implications for the Bonseyes environment?

Firefly-RK3399 is an ARM based embedded device which supports Ubuntu OS. One of the major requirements to run Docker on this device is a Linux distribution, as Docker's isolation mechanisms depend on the features of Linux kernel i.e. namespace and Cgroups. We also investigated other requirements as mentioned in section 4.4.2. Docker is then installed from a source file and configured as it is done in Linux distribution. For demonstration, OpenFace AI model is executed on Docker in which a customized Docker image is generated for Firefly-RK3399. As Bonseyes environment aims to provide platform for the development and deployment of AI applications, there are possible implication from this thesis work to Bonseyes project. Cross compilation of Docker image is identified as a technical challenge. To overcome this difficulty, this thesis provides a solution to cross compile an AI model on ARM-Cortex based device. This is achieved by creating a Docker image compatible for Firefly-RK3399 platform. The success of executing an AI model inside Docker on Firefly-RK3399 implies it is

possible and feasible to use this device for deployment of AI model inside Docker container. Therefore, Bonseyes can consider the technical challenge and use our solution in cross compilation of a Docker image on Firefly-RK3399 platform. Additionally, the research exploited the capability of a single-board device. Firefly-RK3399 is small, cheap and low power consuming yet very powerful embedded device which can be beneficial for Bonseyes environment to compute AI applications either totally locally or partially coupled with the cloud.

2. How significant is the performance overhead of Docker when a load module is deployed in a container on Firefly-RK3399 platform in terms of CPU and memory utilization?

A performance evaluation is carried out to assess the overhead caused by running load inside Docker on Firefly-RK3399. The execution time of synthetic load in multiple scenario is measured to evaluate the CPU speed of Docker. Docker shows a near zero performance degradation compared to the native OS. Also, sustainable memory bandwidth is measured using well known tools. From our finding, Docker shows a very low overhead. With the scope of data analyzed in this thesis, we can say that Docker virtualization overhead on Firefly-RK3399 in terms of CPU and Memory is negligible.

This paper demands future work on investigating the performance of Docker by applying application benchmarking i.e. AI application. The execution time, accuracy, and power consumption of AI model can be considered as performance metrics. So that Docker container can be applied for deployment of AI application based on its performance in terms of reliability and resource efficiency. Especially the accuracy of AI applications is the key metrics that should be evaluated. Furthermore, result from our thesis indicates Firefly-RK3399 is a powerful device. This suggests another research direction towards exploiting the capability of this device as an IoT gateway and as platform for development and deployment of applications, particularly for AI applications.

## REFERENCES

- [1] D. Evans, "How the Next Evolution of the Internet Is Changing Everything," *Cisco Internet Bus Solutions Group IBSG*, pp. 1–11, 2011.
- [2] D. Dicochea, "Cisco Global Cloud Index: Forecast and Methodology, 2014–2019 White Paper," p. 44, 2015.
- [3] "IDC.com IDC Table of Contents," *IDC: The premier global market intelligence company*. [Online]. Available: <https://www.idc.com/research/viewtoc.jsp?containerId=US40755816>. [Accessed: 02-May-2018].
- [4] A. R. Biswas and R. Giaffreda, "IoT and cloud convergence: Opportunities and challenges," in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, 2014, pp. 375–376.
- [5] Lei Shi, "The convergence of AI and IoT, are we there yet?," *Blog from Northstream*, 2017.
- [6] T. Llewellynn *et al.*, "BONSEYES: Platform for Open Development of Systems of Artificial Intelligence: Invited paper," 2017, pp. 299–304.
- [7] B. I. Ismail *et al.*, "Evaluation of Docker as Edge computing platform," in *2015 IEEE Conference on Open Systems (ICOS)*, 2015, pp. 130–135.
- [8] T-Chip Intelligent Technology Co., Ltd., "Firefly-RK3399 - Firefly wiki." [Online]. Available: [http://opensource.rock-chips.com/wiki\\_Firefly-RK3399](http://opensource.rock-chips.com/wiki_Firefly-RK3399)
- [9] Poly AI Inc, "Aipoly - Fully Autonomous Markets." [Online]. Available: <https://www.aipoly.com/>. [Accessed: 12-Jul-2018].
- [10] "BONSEYES – The Artificial Intelligence Marketplace." [Online]. Available: <https://bonseyes.com/>. [Accessed: 18-Dec-2017].
- [11] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," *NIST Special Publication (SP) 800-145*, p. 7, Oct. 2011.
- [12] "Essential characteristics of Cloud Computing." [Online]. Available: <https://www.isaca.org/Groups/Professional-English/cloud-computing/GroupDocuments/Essential%20characteristics%20of%20Cloud%20Computing.pdf>.
- [13] Y. Jadeja and K. Modi, "Cloud computing - concepts, architecture and challenges," in *2012 International Conference on Computing, Electronics and Electrical Technologies (ICCEET)*, 2012, pp. 877–880.
- [14] K. Ashton, "That 'Internet of Things' Thing," *RFID Journal*, p. 1, 2010.
- [15] F. Mattern and C. Floerkemeier, "From the Internet of Computers to the Internet of Things," in *From Active Data Management to Event-Based Systems and More*, vol. 6462, K. Sachs, I. Petrov, and P. Guerrero, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 242–259.
- [16] W. Shi and S. Dustdar, "The Promise of Edge Computing," *Computer*, vol. 49, no. 5, pp. 78–81, May 2016.
- [17] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.
- [18] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog Computing: Platform and Applications," in *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, 2015, pp. 73–78.
- [19] T. H. Luan, L. Gao, Z. Li, Y. Xiang, G. We, and L. Sun, "Fog Computing: Focusing on Mobile Users at the Edge," *arXiv:1502.01815 [cs.NI]*, p. 11, Feb. 2015.
- [20] T. Scheepers, "Virtualization and Containerization of Application Infrastructure: A Comparison." [Online]. Available: <https://pdfs.semanticscholar.org/b06b/c9d88762f5146445bd44e9a9deab174591dd.pdf>.
- [21] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs Containerization to Support PaaS," in *2014 IEEE International Conference on Cloud Engineering*, 2014, pp. 610–614.



- [22] D. Liu and L. Zhao, "The research and implementation of cloud computing platform based on docker," in *2014 11th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)*, 2014, pp. 475–478.
- [23] P. E. N, F. J. P. Mulerickal, B. Paul, and Y. Sastri, "Evaluation of Docker containers based on hardware utilization," in *2015 International Conference on Control Communication Computing India (ICCC)*, 2015, pp. 697–700.
- [24] "What is Docker? - DZone Cloud," *dzone.com*. [Online]. Available: <https://dzone.com/articles/what-is-docker-1>. [Accessed: 11-May-2018].
- [25] "Docker overview," *Docker Documentation*, 11-May-2018. [Online]. Available: <https://docs.docker.com/engine/docker-overview/>. [Accessed: 11-May-2018].
- [26] M. T. Chung, N. Quang-Hung, M. T. Nguyen, and N. Thoai, "Using Docker in high performance computing applications," in *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*, 2016, pp. 52–57.
- [27] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee, "A Container-Based Edge Cloud PaaS Architecture Based on Raspberry Pi Clusters," in *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, 2016, pp. 117–124.
- [28] C. Bormann, M. Ersue and A. Keranen, "RFC 7228 - Terminology for ConstrainedNode Networks." [Online]. Available: <http://www.rfc-base.org/txt/rfc-7228.txt>. [Accessed: 15-Jul-2018].
- [29] S. B. Furber, *ARM System-on-chip Architecture*. Pearson Education, 2000.
- [30] J. V. Vijay and B. Bansode, "ARM Processor Architecture," vol. 4, no. 10, p. 3, 2015.
- [31] ARM, "ARM® Cortex®-A72 MPCore Processor Technical Reference Manual," p. 577, 2014.
- [32] "Firefly-RK3399: Six-Core 64-bit High-Performance Platform," *Kickstarter*. [Online]. Available: <https://www.kickstarter.com/projects/1771382379/firefly-rk3399-six-core-64-bit-high-performance-pl>. [Accessed: 15-Jul-2018].
- [33] Rockchip Electronics Co.,Ltd, "Rockchip\_RK3399\_Datasheet\_V0.7\_20160219.pdf." [Online]. Available: [http://www.t-firefly.com/download/Firefly-RK3399/docs/Chip%20Specifications/Rockchip\\_RK3399\\_Datasheet\\_V0.7\\_20160219.pdf](http://www.t-firefly.com/download/Firefly-RK3399/docs/Chip%20Specifications/Rockchip_RK3399_Datasheet_V0.7_20160219.pdf). [Accessed: 18-Jul-2018].
- [34] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.
- [35] t-firefly, "Firefly | Make technology more simple, Make life more intelligent." [Online]. Available: <http://en.t-firefly.com/Product/Rk3399/spec.html>. [Accessed: 13-Jun-2018].
- [36] "Firefly-RK3399.png - Rockchip open source Document." [Online]. Available: [http://opensource.rock-chips.com/wiki\\_File:Firefly-RK3399.png](http://opensource.rock-chips.com/wiki_File:Firefly-RK3399.png). [Accessed: 12-Jul-2018].
- [37] Suchitra, S. P, and S. Tripathi, "Real-time emotion recognition from facial images using Raspberry Pi II," in *2016 3rd International Conference on Signal Processing and Integrated Networks (SPIN)*, 2016, pp. 666–670.
- [38] A. N. Ansari, M. Sedky, N. Sharma, and A. Tyagi, "An Internet of things approach for motion detection using Raspberry Pi," in *Proceedings of 2015 International Conference on Intelligent Computing and Internet of Things*, 2015, pp. 131–134.
- [39] R. Morabito, "A performance evaluation of container technologies on Internet of Things devices," in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPs)*, 2016, pp. 999–1000.
- [40] A. Krylovskiy, "Internet of Things gateways meet linux containers: Performance evaluation and discussion," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, 2015, pp. 222–227.
- [41] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 171–172.



- [42] R. Morabito, "Virtualization on Internet of Things Edge Devices with Container Technologies: A Performance Evaluation," *IEEE Access*, vol. 5, pp. 8835–8850, 2017.
- [43] D. Beserra, M. K. Pinheiro, C. Souveyet, L. A. Steffemel, and E. D. Moreno, "Comparing the performance of OS-level virtualization tools in SoC-based systems: The case of I/O-bound applications," in *2017 IEEE Symposium on Computers and Communications (ISCC)*, 2017, pp. 627–632.
- [44] H. Chang, A. Hari, S. Mukherjee, and T. V. Lakshman, "Bringing the cloud to the edge," in *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2014, pp. 346–351.
- [45] C. Arango, R. Dernat, and J. Sanabria, "Performance Evaluation of Container-based Virtualization for High Performance Computing Environments," *ArXiv170910140 Cs*, Sep. 2017.
- [46] "Ubuntu16.04." [Online]. Available: <https://pan.baidu.com/s/1hsOCOJU#list/path=%2F>. [Accessed: 14-Jun-2018].
- [47] "Community Edition," *Docker*, 29-May-2016. [Online]. Available: <https://www.docker.com/community-edition>. [Accessed: 14-Jun-2018].
- [48] "Index of /linux/ubuntu/dists/xenial/pool/stable/arm64/." [Online]. Available: <https://download.docker.com/linux/ubuntu/dists/xenial/pool/stable/arm64/>. [Accessed: 14-Jun-2018].
- [49] Docker, "Post-installation steps for Linux," *Docker Documentation*, 11-Jun-2018. [Online]. Available: <https://docs.docker.com/install/linux/linux-postinstall/>. [Accessed: 14-Jun-2018].
- [50] B. Amos, B. Ludwiczuk, and M. Satyanarayanan, "OpenFace: A general-purpose face recognition library with mobile applications," *CMU-CS-16-118*, p. 20, 2016.
- [51] "Torch | Scientific computing for LuaJIT." [Online]. Available: <http://torch.ch/>. [Accessed: 14-Jun-2018].
- [52] A. Geitgey, "Machine Learning is Fun! Part 4: Modern Face Recognition with Deep Learning," *Medium*, 24-Jul-2016. [Online]. Available: <https://medium.com/@ageitgey/machine-learning-is-fun-part-4-modern-face-recognition-with-deep-learning-c3cfc121d78#.ds8i8oic9>. [Accessed: 14-Jun-2018].
- [53] N. Dalal and B. Triggs, "Histograms of Oriented Gradients for Human Detection," presented at the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), 2005, vol. 1, pp. 886–893.
- [54] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, Sep. 1995.
- [55] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux J*, vol. 2014, no. 239, Mar. 2014.
- [56] "OpenFace: Face recognition with deep neural networks," 16-Jun-2018. [Online]. Available: <https://github.com/cmusatyalab/openface>. [Accessed: 16-Jun-2018].
- [57] A. Kopytov, "sysbench-manual.pdf." [Online]. Available: <http://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf>. [Accessed: 16-Jun-2018].
- [58] "MEMORY BANDWIDTH: STREAM." [Online]. Available: <http://www.cs.virginia.edu/stream/>. [Accessed: 16-Jun-2018].
- [59] "STREAM2." [Online]. Available: <http://www.cs.virginia.edu/stream/stream2/>. [Accessed: 16-Jun-2018].