NTNU – Trondheim
Norwegian University of
Science and Technology

TFE4590 Electronic Systems Design Semester Project
Project Report

FPGA acceleration of neural network image recognition

Anders Nilsen

December 18, 2018

Project Assignment

Candidate name: Anders Nilsen
Assignment title: FPGA acceleration of neural network image recognition

Assignment text

Machine Learning has increased dramatically in popularity the last years and is now being used in various applications like web search, speech recognition, object detection, face recognition, etc. Huge set of data are used to train a neural network (Learn), before the network can be used stand-alone to classify new patterns (inference). Today's most common Machine Learning architecture is deep neural networks, which can be seen as layers of matrix multiplication. The network can have typically many layers with many weights (up to several 100k). Therefore, inference requires heavy processing resources, usually run on a GPU. FPGA are natural devices to implement that kind of processing, since they provide several thousand multiplications blocks that run in parallel. The two main FPGA vendors are now providing tools to accelerate inference of different Neural Networks types.

The goal of this project is to: - run standard and custom image recognition networks on these tools - explore their capabilities - benchmark them towards GPUs (which typically has been used so far at Cisco)

To implement these networks, the student will use - Xilinx SDAccel and XFDNN (https://www.xilinx.com/applications/megatrends/machine-learning.html) on an Amazon AWS F1 cloud platform. - Intel/Altera Deep Learning development toolkit (https://software.intel.com/en-us/computer-vision-sdk) running on an Intel development kit available in a NTNU server.

Assignment proposer/co-supervisor: Florian Bochud (flobochu@cisco.com)

Supervisor: Kjetil Svarstad (kjetil.svarstad@ntnu.no)

Abstract

Using convolutional neural networks for voice recognition has increased in popularity during the past few years, and Cisco has implemented a "wake-up word" using a CNN in their Webex platform. As neural networks require heayv computational power, efforts have been made to accelerate the CNN computation using GPUs and FPGAs. While GPUs offer high performance, research is being made into using FPGAs for computation as they offer high performance per watt compared to GPUs, which has led to the development of neural network acceleration toolkits from Intel and Xilinx. The goal of this project was to port an OpenCL-implementation of Cisco's wake-up word CNN to both toolkit: Intel OpenVINO and the Xilinx ML Suite.

The CNN was not ported to the ML Suite, but was successfully ported to OpenVINO and run on an Intel Xeon E5620 CPU and an Intel Arria 10 GX Development Kit FPGA on a server at NTNU. The result was a classification time of 5.85 ms and 3.10 ms, respectively. Though the FPGA implementation was on par with the earlier OpenCL-implementation with a classification time of 3.6 ms, the result was still slower than the 2.2 ms classification time achieved by Cisco on an Nvidia Tegra X1 GPU. The OpenVINO implementation was most likely limited by running on a server from 2011, lacking the recommended PCIe-port and with only one convolutional layer being accelerated. Nevertheless, the results are promising for OpenVINO acceleration on FPGAs, providing a relatively easy way of accelerating existing neural network applications compared to implementing the network in OpenCL.

Contents

1 Introduction **5**

2 Background **7**

2.1 Machine learning 7
2.2 Neural network topologies 8
2.3 Neural network training 10
2.4 Inference 10
2.5 Neural Networks on FPGA 11

3 Survey of existing frameworks **12**

3.1 OpenCL 12
3.2 Intel OpenVINO 13
3.3 Xilinx ML Suite 14
3.3.1 Amazon Web Service 15

4 Accelerating a speech-recognition CNN **17**

4.1 Previous work 17
4.2 Amazon AWS 19
4.2.1 Registering and financing 19
4.2.2 Server setup and connecting 19
4.2.3 Testing and importing 20
4.3 Intel OpenVINO 21
4.3.1 Setup 21
4.3.2 Running examples 22
4.3.3 Porting Skafså's code to OpenVINO 22
4.3.4 The ported code 23
4.3.5 Running the program 24

5 Results **26**

5.1 Test setup specifications 26
5.2 Program execution times 26

5.3 Layer execution times 26

6 Discussion 29

6.1 Result analysis 29

6.1.1 CPU vs FPGA 29

6.1.2 Test setup influence on the results 30

6.1.3 OpenVINO performance compared to Skafså's OpenCL implementation 30

6.1.4 Model Optimizer optimisation 31

6.2 Intel OpenVINO vs AWS/Xilinx 31

6.2.1 Feature comparison 31

6.2.2 Ease of use 31

6.3 Further work 32

6.3.1 Use the ML Suite locally to port the network to a Xilinx FPGA 32

6.3.2 Change the network to map conv1 to the FPGA 32

6.3.3 Run the inference on a more modern computer 33

7 Conclusion 34

References 35

A Source code 38

A.1 voice_recognition_OpenVINO.cpp 38

A.2 record_voice.cpp 48

A.3 write_wav.c 51

List of Tables

3.1 Performance and cost of Amazon Web Service EC2 F1 instances . . . 16

5.1 "Hey Spark"-CNN execution times and throughput from the Inference Engine . . . 27

5.2 "Hey Spark"-CNN execution times for each layer on CPU and FPGA . 28

List of Figures

2.1 Perceptron model as described by Frank Rosenblatt[8]. The activate function is a threshold for deciding whether the result of the input function is 1 or 0. The threshold is adjustable. 8

2.2 Types of artificial neural networks[10] 9

3.1 GoogleNet acceleration with batch size = 1 on Intel FPGAs using OpenCL 14

3.2 Comparison of GoogLeNet V1 neural networks inferred on Nvidia GPUs, Intel FPGA and Xilinx FPGAs [20] 15

4.1 Cisco "Hey Spark" wake-up word CNN topology 18

4.2 The SDx main menu using X11-forwarding from AWS EC2 F1 instance 21

4.3 Program flow using OpenVINO[26] 23

5.1 "Hey Spark" CNN throughput based on Inference Engine times. . . . 27

Chapter 1

Introduction

This project was done in co-operation with Cisco, and features acceleration of a convolutional neural network (CNN) for voice recognition, even though the title of the project was image recognition. Specifically, the recognition of the phrase "Hey Spark" used to wake up a Cisco conference system from sleep-mode. This type of phrase and function is also called a "wake-up word". The use of voice commands has increased in popularity the last decade, with Apple's Siri being a well-known example.

Convolutional neural networks usually require heavy computational power to function within a reasonable time, causing a need for performance in situations where it's not feasible due to power, cost or area constraints, such as on an embedded system. To provide more performance and lower latency, the CNN can be accelerated on a server with GPUs or FPGAs. While GPUs have a relatively long history of acceleration, having been used for the heavily parallel task of computer graphics calculations since the late 1990s, FPGA's have not been as widely adopted. FPGA's offer a high degree of reconfigurability, in theory being able to recreate most digital circuits, while using less power than a GPU [1]. This makes FPGAs attractive both on a local device or on a server.

Previously, the wake-up word CNN was implemented and accelerated using OpenCL by Skafså[2], achieving a throughput of 277 classifications per second using an Intel Arria 10 GX Development Kit. The result was compared to a larger network implementation on an Nvidia Tegra X1 GPU at Cisco, achieving a throughput of 454 classifications per second. The goal of this project has been to port Skafså's implementation to Intel OpenVINO and Xilinx ML Suite, two competing acceleration platforms utilising custom machine learning FPGA designs to accelerate high-level neural network descriptions.

The report will provide some background information on neural networks in chapter 2, with a survey of some existing frameworks for acceleration. Chapter 4 will provide more detailed information on how to use OpenVINO and the ML Suite along with how Skafså's program was implemented in OpenVINO, while chapter 5 will show the results of the implementation. Chapter 6 will discuss the results and compare them against Skafså's results while discussing some factors which might affect the measured performance, while chapter 7 is a conclusion.

Chapter 2

Background

This chapter will present some of the history of machine learning, provide a brief overview of the situation today and mention some frameworks for neural network design.

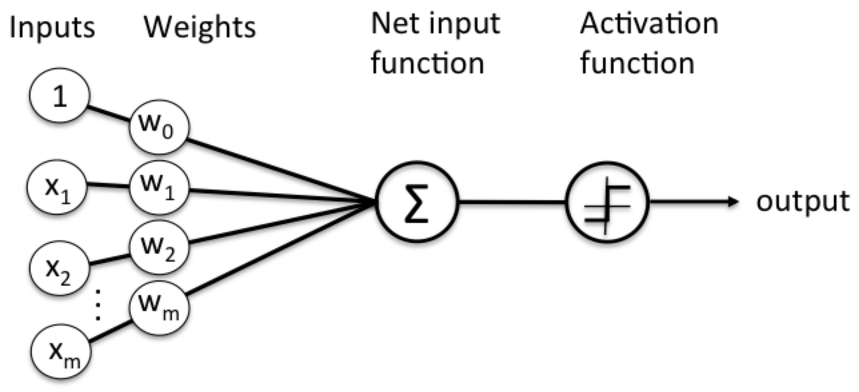
2.1 Machine learning

Machine learning is a concept which dates back to 1959 when the term was coined by Arthur Samuel. [3]. The concept revolves around using statistical techniques to make a computer "learn" an operation using sets of data instead of manually programming the response. Initially, machine learning was restricted to relatively simple tasks such as playing checkers and improving the quality of phone calls. The concept remained mostly within academia for the next four decades until the required computational power became affordable enough for individuals and companies to make use of the techniques developed. The use was further increased through the introduction of free, open-source frameworks such as TensorFlow[4] and Caffe[5].

According to an analysis of the machine learning application landscape by Moor Insights & Strategy, over 2300 investors had funded 1700 machine learning startups as of February 2017 [6]. Companies such as Amazon, Baidu, Facebook, Google, IBM and Intel are claiming that they are "AI-companies", investing in machine learning technology and utilising it in their services. FPGA manufacturers such as Intel Altera and Xilinx offer FPGA-accelerators for use in computers to satisfy the need for high-speed neural networks. Amazon are renting servers fitted with FPGA-clusters to customers who need computational power to run machine learning algorithms. Artificial neural networks are being deployed in all fields: Industry, public health, surveillance and transportation are all fields where machine vision and AI are being used.

2.2 Neural network topologies

Machine learning uses artificial neural networks to simulate the human neural network. Although computers are more efficient at performing mathematical computations and storing data, humans are better at learning how to do things while a computer has to be told how to do something. Learning something new, recognising faces and voices and decision making are task which humans are better at, and which computers struggle with. These actions are, in humans, performed by a neural network in the brain. The neural network consists of approximately 10^{11} neurons which are interconnected into a decision tree[7]. The desire to mimic the human neural network motivated scientists to create a mathematical model of a neuron, called a perceptron. A model of the perceptron was described by Frank Rosenblatt in 1958.



Schematic of Rosenblatt's perceptron.

Figure 2.1: Perceptron model as described by Frank Rosenblatt[8]. The activate function is a threshold for deciding whether the result of the input function is 1 or 0. The threshold is adjustable.

Figure 2.1 shows a block diagram of the model proposed by Rosenblatt. This served as a basis for further computational models, and was implemented in software by IBM on 1958 to perform image recognition, but the network could only recognise one pattern. This was due to the network having only one layer, and the solution was to implement a feed-forward, multi-layered neural network consisting of perceptrons, as suggested by Stephen Grossberg in 1973[9].

These new types of networks consist of several different types. The most common network types are shown in figure 2.2. The network types can be divided into several categories: Feed-forward, recurrent, competitive and self-organising. Feed-forward networks, also called convolutional neural networks, like figure 2.2a and 2.2b, can be compared to combinatorial logical circuits. Recurrent networks, such as figure 2.2c and 2.2d, are more akin to sequential circuits with the recursive connections acting as a short-term memory. Competitive neural networks and self-organising networks, such as figure 2.2e and 2.2f, allow the networks to build self-organise and build input feature maps, a feature used during network "training".

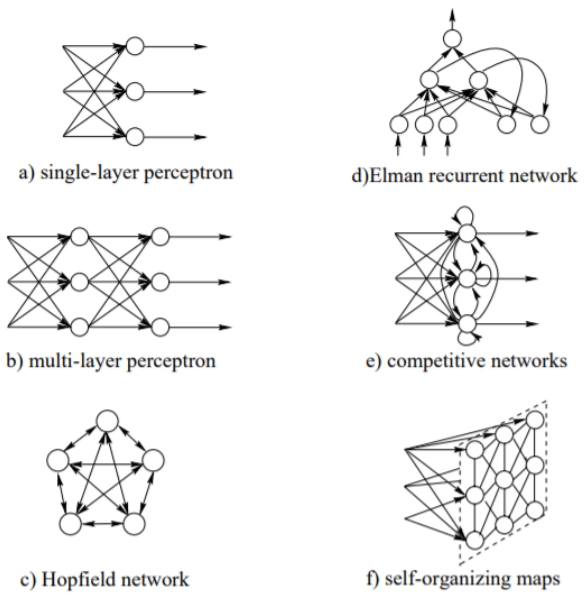

a) single-layer perceptron
d)Elman recurrent network
b) multi-layer perceptron
e) competitive networks
c) Hopfield network
f) self-organizing maps

Figure 2.2: Types of artificial neural networks[10]

What neural network type to use depends on the task of the network. For image recognition, convolutional neural networks can be used, while for handwriting and speech recognition, recursive neural networks can be used. After selecting a network type, the weighting of the connection between the nodes has to be calculated.

In general, the networks consist of multiple layers where each layer can be trained to look for a certain attribute. One example is a convolutional neural network for recognising cats. One layer looks for eyes, while another layer looks for fur, and another layer looks for a tail. To get the correct response from each layer, the weights for the connections between the layers are adjusted until they produce the correct response. This is done by "training" the network.

2.3 Neural network training

The algorithms can be "trained" in multiple ways. The most common way of training is supervised training where the algorithm is fed a set of specific data, e.g. an algorithm for detecting cats is fed pictures of cats. Another way is unsupervised training, where the algorithm gets a set of rules and "trains" itself. This was famously used by the AI research-company AlphaMind in 2017 to create a computer program, AlphaGo Zero [11]. In October 2017 it beat their previous Go-program "AlphaGo", which the previous year beat world Go champion Lee Sedol.

2.4 Inference

An artificial neural network usually requires substantial training data and computational power to perform the training. This makes creating an artificial network on a users cellphone difficult while delivering results with low latency. The solution is to create an optimised model of the trained network. This is done through a process called "inference".

Inference is the process of optimising a trained artificial neural network and deploy it on a computational device, such as a cellphone, a computer or an FPGA, depending on what the scale is. A cellphone could have a neural network running to analyse pictures the user takes, while a company might infer the network to a server to perform speech recognition on users' cellphones. How the network can be inferred to a device varies depending on how it was made. Usually, artificial neural networks are coded in either C++ or Python using frameworks such as OpenCL[12], OpenCV[13], TensorFlow[4] or Caffe[5].

These frameworks simplify the process of constructing and using artificial neural networks in the user's application. TensorFlow, developed by Google, for instance, is a high-level API in which the programmer describes the neural network layers. TensorFlow can be used for most artificial neural network-based tasks. Caffe, developed by Berkeley AI Research, is another commonly used API and is mostly used for image recognition. Both of these APIs are open-source, allowing them to be used, modified and distributed by users free of charge. OpenCV, short for Open Source Computer Vision Library, is an open source computer vision and machine learning software library for C++, Python, Java and Matlab[13]. It offers over 2500 computer vision and machine learning algorithms. In addition, it supports training on GPUs through OpenCL and CUDA, in addition to CPU's.

During inference, unnecessary parts of the network are identified based on their activity and removed. Secondly, layers can be fused together to reduce the overall number of layers. The result is an optimised algorithm in terms of speed and size, usually at the sacrifice of accuracy. Inference can be done manually by a programmer or by a computer program. Traditionally, artificial neural networks have been inferred on CPUs and GPUs, but due to the growing market of neural network applications, FPGA's have become a viable target.

2.5 Neural Networks on FPGA

Due to their parallel nature, FPGAs offer a high performance per watt, making it a strong candidate for neural network computations and inference. In addition to speed, FPGAs are reconfigurable allowing for several configurations to be programmed and run on the platform during development. Run-time reconfiguration allows for the FPGA to be reconfigured during the different stages of the artificial neural network algorithm to increase hardware density [14]. Artificial neural networks also allow for speedup when the inferred algorithm uses low numeric precision in calculations, e.g. using fixed point weighting and quantization data instead of 32-bit floating point can provide substantial speedup while maintaining reasonable accuracy [15]. Implementing this function when inferring to an FPGA results in significant speedup when using low-bit numeric precision [16].

Due to their performance per watt and speed, FPGAs are a desired platform for running inferred artificial neural networks. One of the main problems reducing the adoption rate is, and has been for many applications, how they are programmed. FPGAs are not programmed in the same way as micro-controllers or computer programs, using a programming language such as C++ or C which is assembled into machine-level instructions and run on a CPU. An FPGA is "programmed" by describing their functionality using a Hardware Descriptive Language, such as VHDL or Verilog. The HDL code is then synthesised into a netlist which is mapped onto the FPGA. This way of programming differs from regular programming and increases the difficulty of writing effective and good HDL-code, in most cases requiring specialised engineers. To reduce the difficulty of programming FPGAs, several tools exist to synthesise conventional programming languages, such as C, C++ and Python, into VHDL code. This is called High Level Synthesis, and can be utilised in conjunction with artificial neural networks to allow for inference of C++ code using OpenCV, TensorFlow, Caffe and other frameworks to FPGAs without the need for the designer to write HDL-code.

Chapter 3

Survey of existing frameworks

This chapter will explore the existing frameworks for running neural networks on FPGAs. Each section gives a brief overview of each framework.

3.1 OpenCL

OpenCL[12], short for Open Computing Language, is a platform heterogeneous framework for writing and running programs on several computing platforms, including CPUs, GPUs, FPGAs, DSPs and other hardware accelerators. OpenCL was launched in 2009 by Apple to utilise the acceleration possibilites of on-board GPUs. A collaborative group, the Khronos Compute Working Group, was created featuring representatives from several CPU, GPU, embedded-processing and software companies to maintain and improve the framework. As of this report, the newest version was 2.2 which incorporated more C++ features to the language.

The OpenCL framework is officially available for C and C++, but is unofficially available for Python, Java, Perl and .NET. An OpenCL implementation of a program is based around a host containing several compute devices, such as a CPU and a GPU, which is further divided into multiple processing elements. A function which is executed using OpenCL is called a kernel, and can run in parallel on all processing elements. A programmer can utilise the acceleration capabilities available on a system by getting the device information from the computer the program is running on.

While OpenCL provides good possibilities for acceleration and resource usage, it is limited by its low-level nature. While it has functions for standard operations like FFT, neural networks have to be manually declared unless the frameworks used to generate the network have OpenCL-branches. Caffe has such a branch[17], but it is currently under development. TensorFlow has an OpenCL-branch on its roadmap. The lack of neural network framework support limits its adoption. A more supported and similar framework to OpenCL is Nvidia's CUDA, although this only runs on Nvidia GPUs.

3.2 Intel OpenVINO

The OpenVINO toolkit is Intel's solution for running neural networks on FPGAs, and aims to simplify the process compared to existing solutions. The OpenVINO toolkit was launched in 2018 by Intel, replacing the Open Vision SDK. It allows the user to program applications which can be accelerated on Intel processors, GPUs, FPGAs and VPUs. The suite consists of several programs, such as the Intel Deep Learning Deployment Toolkit with a Model Optimizer for optimising the neural network model and an inference engine for inferring the model to the target device. It also includes the FPGA Runtime Environment for running the accelerated software on an Intel Altera FPGA using Linux, as well as optimised computer vision libraries like OpenCV and OpenVX. The toolkit is available for Windows 10, CentOS, Ubuntu and Yocto Project Poky Jethro, but compatibility with different options such as FPGA acceleration varies between platforms. As of this report, the FPGA acceleration with OpenVINO works on the Altera Arria 10 GX FPGA development kit and the Intel Vision Accelerator Design with Intel Arria 10, with a retail price of $4,495, while there is no publicly available retail price for the Vision Accelerator. These FPGAs have a PCI-Express connector which allows them to easily be integrated into a computer. In addition to FPGA, OpenVINO supports CPUs, VPUs and GPUs[18]. The supported CPUs are 6th or 8th generation Intel CPUs and select Pentium processors. The supported GPUs included in some Intel CPUs. The supported VPU is the Intel Movidius and the Neural Compute Stick 2.

OpenVINO is mainly used for accelerating image recognition, but can be used for other purposes such as audio. It supports frameworks such as Caffe and TensorFlow, and deep learning architectures such as AlexNET and GoogleNET. It supports a set amount of layers for each framework out of the box, with custom layer support available for developers.

OpenVINO has an advantage over OpenCL in that it is relatively simple to incorporate into an existing C++ program. To use OpenVINO in a project, the neural network model is optimised using the model provided by the neural network framework, such as a .caffemodel (from Caffe), with the calculated weights with the Model Optimizer. The Optimizer provides an optimised intermediate representation which is loaded into the code using the Inference Engine API. The API prepares and infers the network to the target device and runs the network with the supplied input data. All pre- and post-processing is done in C++, so the only part which has to be replaced is the neural network of the previous implementation.

Intel has not published any whitepapers featuring the performance of OpenVINO, but have published a whitepaper showcasing the performance of their FPGAs using their Quartus Design Suite with OpenCL. As OpenVINO uses OpenCL to infer the neural network to the FPGA, the paper can serve as a reference for the achievable performance. Though, the network used in the test, the Deep Learning Network (DLA), was manually designed and not generated[19]. Figure 3.1 shows that the Arria 10 GX achieved a classification throughput of 760 images per second with a INT9 data and computation, while it achieved 250 images per seconds with FP16. This shows the advantage FPGAs have when using relatively low-resolution integers, as mentioned in chapter 2.5.

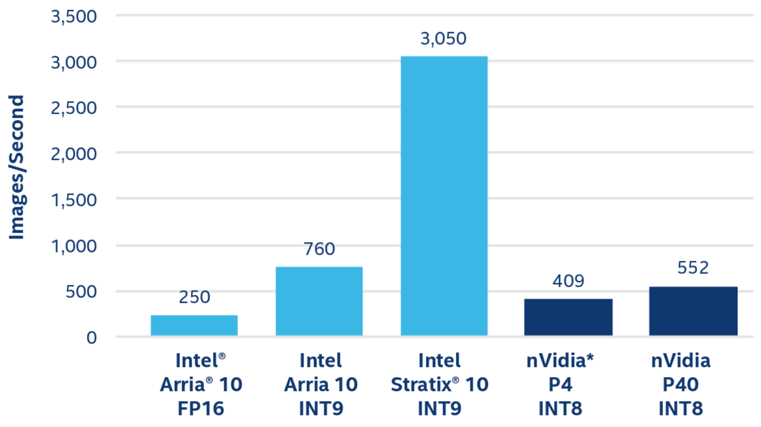


Figure 3.1: GoogleNet acceleration with batch size = 1 on Intel FPGAs using OpenCL

3.3 Xilinx ML Suite

To simplify neural network deployment on FPGAs, Xilinx has developed and released the Machine Learning (ML) Suite in 2018. The suite allows the user to accelerate neural networks on Xilinx FPGAs using inference from high-level programming languages such as C++ and Python. The ML suite features the xfDNN Middleware, a software library with APIs for for the aforementioned and several other programming languages which connects the neural network from a supported framework, such as Caffe and Tensorflow, to the xDNN IP core. The xDNN IP core runs on a Xilinx FPGA and acts as the target for the inferred neural network. The IP core consists of several general CNN processing elements, allowing it to be reconfigured to fit current neural network topologies and maybe future topologies.

The ML Suite has a programming flow similar to OpenVINO: Initially the model is compiled and afterwards quantized, usually from FP32, to Int16 or Int8. The final stage is deployment, which is done through integrating xfDNN Middleware API into the program which is to be accelerated. To showcase the performance of the framework, Xilinx has published a whitepaper comparing their framework to Intel and Nvidia[20]. The results are shown in figure 3.2

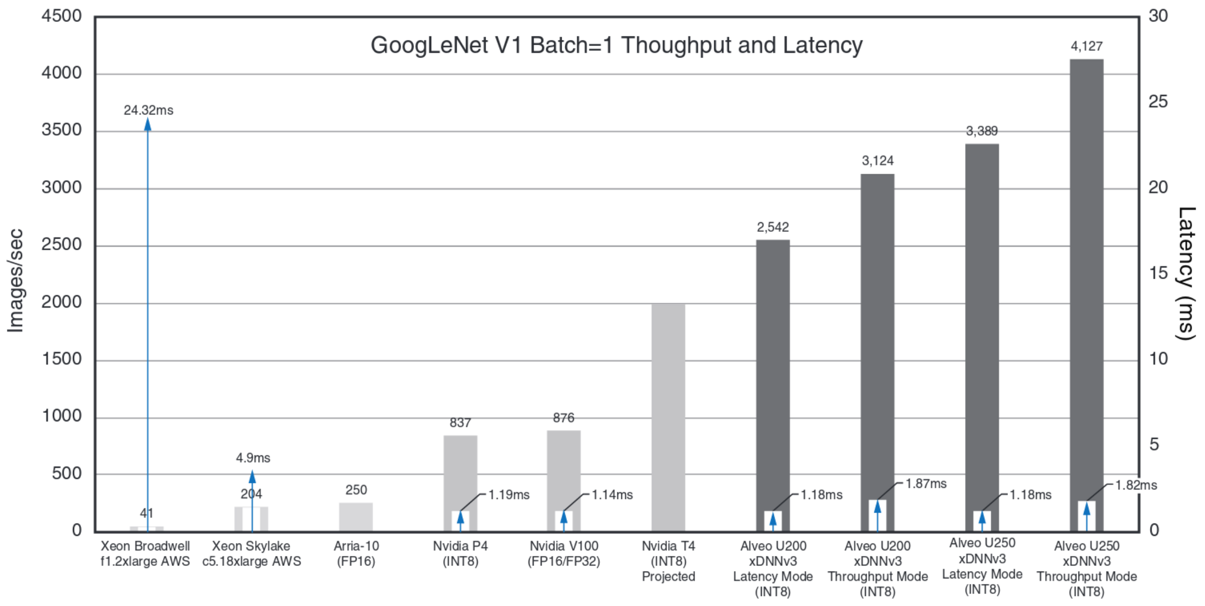


Figure 3.2: Comparison of GoogLeNet V1 neural networks inferred on Nvidia GPUs, Intel FPGA and Xilinx FPGAs [20]

In addition to the ML Suite, Xilinx offers the SDAccel development environment for accelerating C/C++ applications on FPGAs. The suite finds accelerateable parts of the user's code when using the SDAccel GUI interface, i.e. compatible code such as OpenCL-code or C-code.

All software is available for free if the user has a Xilinx account. As of this report, the FPGA acceleration works on the Alveo U200 Data Center Accelerator Card, Alveo U250 and the Xilinx Virtex UltraScale+ FPGA VCU1525 Acceleration Development Kit, with a retail price of $8,995, $12,995 and $10,000 respectively. An alternative to purchasing the cards is to use Amazon Web Service.

3.3.1 Amazon Web Service

To make it easier for programmers to accelerate their applications using their tools, Xilinx has teamed up with Amazon to provide servers for rent with up to 8 FPGA Ultrascale+ accelerator cards. The servers are available to rent on Amazon Web Services, with the the necessary tools available free of charge as Amazon Firmware Images which can be deployed on any available Amazon server. The cost of renting a server depends on the number of FPGAs and virtual CPUs, and is shown in table 3.1.

In addition, when registering a free account, 750 hours of server time per month on a t2.micro server is provided free of charge for 12 months. The specifications of the t2.tiny server is shown in table 3.1. The instance does not feature any dedicated storage, and the ECU rating is not available.

Name	vCPU	ECU[1]	Memory [GiB]	Storage [GB]	FPGAs	Cost[$/hour]
f1.2xlarge	8	26	122	470	1	1.815
f1.4xlarge	16	52	244	940	2	3.630
f1.16xlarge	64	188	976	3760	8	14.52
t2.micro	1	-	1	-	0	Free

Table 3.1: Performance and cost of Amazon Web Service EC2 F1 instances

[1]ECU = EC2 Compute Unit, a unit of measurement for performance used by Amazon.

Chapter 4

Accelerating a speech-recognition CNN

This chapter will briefly explain the related work by Skafså, and will look at Amazon Web Services with the Xilinx ML Suite and Intel OpenVINO and their features, along with how they were set up for this project. The chapter will also cover how Skafså's code was ported to the frameworks to OpenVINO and why it was not ported to the ML Suite.

4.1 Previous work

To test out the various acceleration platforms, a speech-recognition CNN was provided by Cisco along with an OpenCL-implementation from a prior master's thesis by Skafså[2]. In his thesis, Skafså implemented the pre-trained neural network using C++ and OpenCL through Intel Altera's C++ AOCL-library included in the Intel FPGA SDK for OpenCL for deployment on an Intel Altera Arria 10 GX Development kit. Figure 4.1 shows the CNN network topology for the Wake-up word detection network.

The network consists of 14 layers, including input and output. The input data is a 1x1x40x90 input matrix of speech sample data in MFCC form, and the output is a 1x2x1x1 output matrix. The outputs are "Wake up word not detected" and "Wake up word detected" for output 1 and 2, respectively. The convolutional layers are most likely to benefit from acceleration as they feature heavy computation in the form of interconnected perceptrons. For more information on the CNN, see chapter 5.1 in [2].

MFCC, short for Mel-Frequency Cepstral Coefficients, is a technique for low level spectral information to convey vocal tract characteristics, eliminating unnecessary information with regards to phonetic analysis[2]. The MFCC representation of the recorded input signal is generated using a custom program provided by Cisco. The

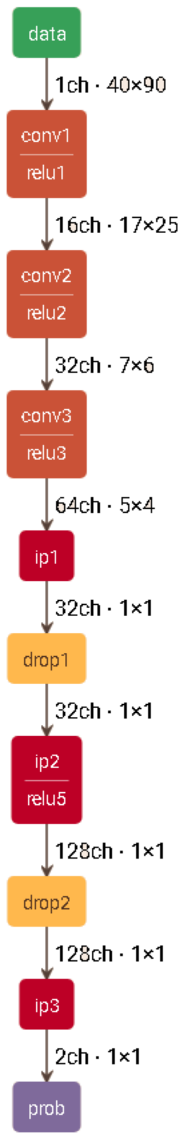


Figure 4.1: Cisco "Hey Spark" wake-up word CNN topology

program is called after exporting the .wav-file to a hard-coded location.

Two programs were created by Skafså: One using a pre-recorded sample as the speech input and a live implementation using the PortAudio API[21] to record audio. The pre-recorded sample is imported into the program using the open-source cnpy-library[22] for C++ to import the sample. The sample was generated using the scientific computing library NumPy for Python, and was exported as an .npy-file during development.

The program is mainly split into two parts: pre- and post-processing in C++ and neural network computations implemented in OpenCL. Skafså calculates the execution time based on the reports from the OpenCL implementation, based on the time spent initialising and running the kernels.

Using the Arria 10 GX Development kit with and the C++/OpenCL-implementation, Skafså was able to perform a classification in 3.6 ms. To have something to compare with, Cisco ran a bigger CNN implementation on an NVIDIA Tegra X1 GPU, which performed a classification in 2.2 ms. Skafså also measured the performance in MAC/s, or multiply-accumulate operations per second. The implementations had a computational speed of 0.54 Gmac/s and 11.9 Gmac/s.

Looking at table 6.8 in [2], most of the computational time of the FPGA implementation is caused by the convolutional layers. Optimising the neural network model using either OpenVINO or Xilinx ML Suite could help accelerate these networks and speed up the classification time. In addition, using a network on the FPGA optimised by the FPGA manufacturer will most likely be optimised for the target architecture.

4.2 Amazon AWS

4.2.1 Registering and financing

As the F1 instances have a fixed rate per hour, financing was required to run simulations and develop. The free server instance, the t2.micro instance, is not suited for extensive programming and compiling with only 1 CPU core and 1 GiB of memory. It is therefore necessary to rent larger server instances with greater computational capabilities, especially if any HDL-synthesis is required. Luckily, Amazon has a student program with affiliated colleges and universities, called AWS Educate, which NTNU is a part of. AWS Educate gives access to special courses to learn the AWS platform, educative communities and $100 of free credit each year. Server instance options are still limited to the free-tier, making the F1-instances unavailable. This can be circumvented by requesting access through Amazon's support service. Server instances are free to start and only start accumulating a bill once they are in the "running" state. The instance can be stopped, which is similar to pausing as the instance is not terminated, to stop billing.

4.2.2 Server setup and connecting

After registering an account, access is granted to the Amazon AWS dashboard with shortcuts to all services, such as EC2-instances. Starting a server is quick and can be done by following a guide[23] for server setup by Amazon. EC2-instances use AMIs, Amazon Machine Images, as the starting point for a server. AMI's are essentially virtual machine images with an operating system, such as Windows, Ubuntu and CentOS, and software. The AMI for FPGA development and acceleration is the FPGA Developer

AMI, which features the AWS EC2 FPGA Hardware and Software Development Kit[24]. The kit includes SDAccel, Vivado and SDx amongst other programs related to FPGA development and acceleration. The kit also includes examples for SDAccel acceleration and high-level synthesis using Vivado. After customising the FPGA Developer AMI to the user's liking, an AMI can be created. This allows data and configurations to quickly be deployed on different server types.

After setting up the server, the user has to define a security group and whitelist the current IP address to be allowed access to the server. Security groups with user rights are managed through IAM roles. IAM, or Identity and Access Management, is the AWS service for securely controlling access to AWS resources. Creating and administering IAM roles is done through the IAM Management Console. To whitelist IP addresses, the security group for the instance has to be edited. By editing the inbound rules of the security group, IP addresses can be added to the whitelist.

Connecting to the server can be done in several ways, but the simplest is using SSH. The connection details are listed in the AWS EC2 management console with a guide on how to connect. The default connection is SSH using terminal, and X11-forwarding has to be enabled manually to enable GUI interaction. Some form of GUI is necessary to use most of the applications, such as Vivado or SDAccel.

4.2.3 Testing and importing

To test the acceleration capabilities, AWS recommends starting with the "Hello World OCL" example. This will show the programming flow using SDAccel to accelerate an OpenCL-application by synthesising it and running it on an FPGA, as well as generating an AFI, an Amazon FPGA Image. An AFI can easily be transferred between servers and can also be sold on the Amazon AWS Marketplace. This example is also meant to showcase SDx-capabilities.

After finishing the example, an attempt was made at importing Skafså's program. This can be done by using SDx after setting up the server with X11-forwarding or using some other remote-desktop application. Importing the project files is simple, and after importing the application looks for parts of the code which can be accelerated. This proved unsuccessful, possibly due to Skafså using the aocl-library in his C++ application. No further attempt to accelerate the code using AWS was done after this.

Due to the server being located in Ireland, there is a considerable delay when interfacing with it using X11-forwarding. In addition, X11-forwarding is unreliable, causing applications to crash seemingly at random. On one occasion it was not possible to load SDx past the main menu shown in figure 4.2. This, along with the delay, makes it difficult to program efficiently. Due to this, along with a limited amount of time, the focus was shifted to OpenVINO, which was being worked on concurrently.

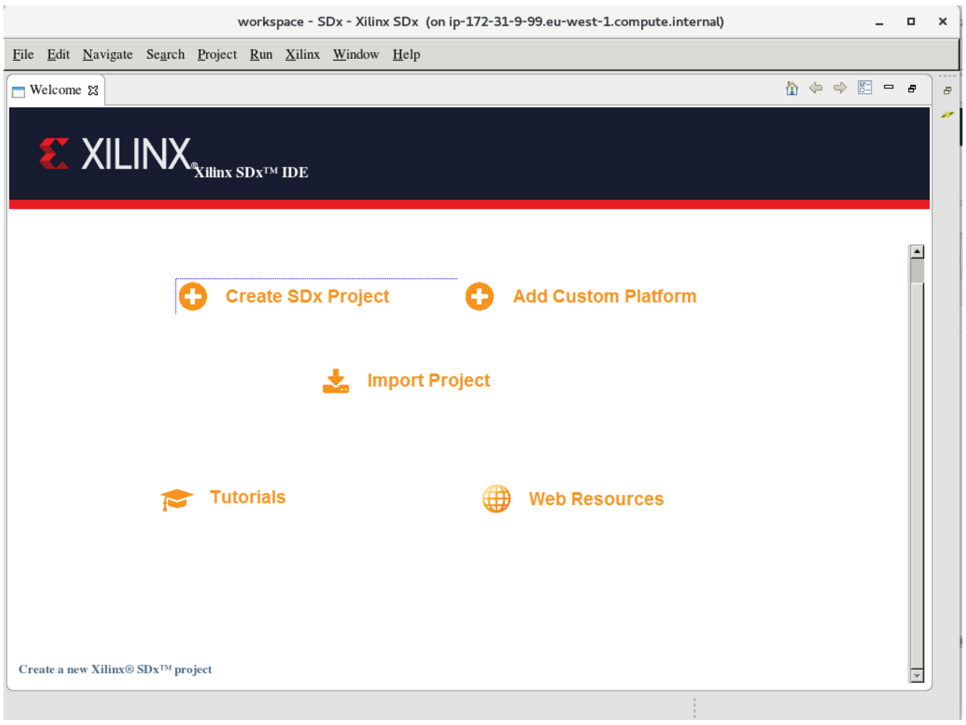

workspace - SDx - Xilinx SDx (on ip-172-31-9-99.eu-west-1.compute.internal)
File Edit Navigate Search Project Run Xilinx Window Help
Welcome
XILINX Xilinx SDx™ IDE
Create SDx Project
Add Custom Platform
Import Project
Tutorials
Web Resources
Create a new Xilinx® SDx™ project

Figure 4.2: The SDx main menu using X11-forwarding from AWS EC2 F1 instance

4.3 Intel OpenVINO

4.3.1 Setup

As specified in the project description, a server at NTNU was available with an Intel Arria 10 GX FPGA Development kit. The server, macron.iet.ntnu.no, had previously been worked on by a masters student and was supposedly properly set up with the accelerator card. The server used CentOS as the operating system, which is supported by OpenVINO.

Initially, the server was unavailable as it was in storage, so CentOS along with OpenVINO was installed on a Lenovo E470 laptop with an Intel i7-7500U CPU to get familiar with the software and the operating system. OpenVINO installation is quick and easy and can be done either in GUI or terminal. Following Intel's installation guide [25] to set up the FPGA, the process can be done in a couple of hours. Note that when using Quartus Lite, it seems that it's not possible to flash bitstreams to the FPGA accelerator. It can still be programmed and will function properly, but it has to be programmed with the bitstreams from step 1 in the installation guide every time the computer is rebooted.

After the server was put online, it turned out that the Arria 10 GX card had not been set up at all, only plugged into a PCI-express slot on the motherboard of the server. In addition, shortly after being put online, the server crashed and was unable to boot. The solution was to re-install CentOS, and at the same time install the FPGA accelerator according to the installation guide.

4.3.2 Running examples

To test the functionality of OpenVINO and the accelerator card, Intel has included examples with common machine learning tasks, mostly image recognition examples. The initial example is the "hello-classification" example, which is an image classification program for recognising objects in a picture of a car. This can either be run on a CPU or an FPGA, depending on which example. The Intel installation guide features the "hello-classification"-demo for FPGA and CPU as a part of the installation guide. The demo can be compiled and run after setting the correct environment-variables for Linux, which is done using a *setupenv*-script located in the OpenVINO folders. The demo runs in the console and prints out the classification results in addition to the throughput. Following Intel's installation guide, the demo was initially run on the CPU with a batch size of 1, followed by FPGA with a batch size of 1 and finally on FPGA with a batch size of 100. The result was a throughput of 22FPS, 288FPS and 338FPS, respectively, on the server. The reason for increased performance when running increased batch sizes on FPGA is, according to Intel, due to the initial performance cost of inferring the network onto the FPGA during the first classification.

4.3.3 Porting Skafså's code to OpenVINO

Although Skafså used Intel FPGA OpenCL SDK, the code could not be directly imported to the OpenVINO toolkit. Instead, the functionality of the code had to be included in a new program which used the OpenVINO API. As mentioned, Intel provides many sample applications in C++ to help understand how to use the API and the program flow, although the program flow and API has changed since the initial release of OpenVINO. While the still work, the examples use old API calls can make it confusing when trying to learn the API. Intel has provided a guide [26] for integrating the Inference Engine into pre-existing applications.

Running Skafså's code required installing some additional libraries in addition to modifications. The use of cnpy to read the pre-recorded voice sample was practical for training purposes when generating the file in Python, but for inference testing it was more practical to convert the file to .csv and read it into the application using *std::ifstream* and *getline* to read the samples. Using PortAudio to record the voice for the live demo required building the PortAudio library, which has to be done after making sure that a suitable audio API, such as ALSA, is installed on the target system.

Otherwise, the PortAudio API won't detect any recording devices when being called in the C++ application.

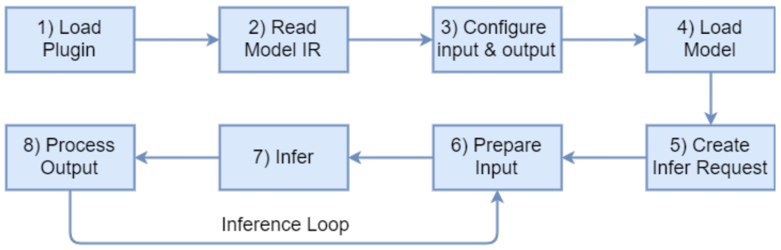


Figure 4.3: Program flow using OpenVINO[26]

As the main part of Skafså's code was initialising and running the OpenCL-code, only the helper functions could be used in the OpenVINO implementation. Functions for recording live audio, timestamping, and writing .wav-files were kept. Otherwise, the main functionality was replaced with the OpenVINO API. Using the Intel's examples and guides, the program was re-written to use the OpenVINO API. Figure 4.3 shows the recommended program flow from Intel in their Inference Engine Developer guide [26].

4.3.4 The ported code

Using the suggested flow, the program was ported to OpenVINO. The code for the main-loop and some helper functions is included in appendix A.1, while helper functions for voice recording and .wav-export are included in appendix A.2 and A.3, respectively. The code can be downloaded from Github[1].

Initially, the input arguments are read using the *argc* and *argv* command line options. The input arguments are used to set:

1. The path to the optimised network files generated by the Model Optimizer.
2. The number of times the program will loop. The program will start looping at stage 6 shown in figure 4.3, reading either the pre-recorded sample or recording a new input using PortAudio.
3. Whether to infer to CPU or FPGA and CPU.
4. The amount of information printed out during runtime.

After reading the input arguments, the program follows the flow shown in figure 4.3, with the following actions for in each stage:

[1]`https://github.com/andernil/OpenVINO_project`

1. The plugin is selected based on the input argument and can be either CPU or FPGA. The Inference Engine has several plugins, usually one for each compatible device, but also for using multiple inference targets. This program uses *HETERO:FPGA,CPU* which runs the inference of the neural network layers primarily on the FPGA with CPU as the fallback device in case the layer is incompatible with the FPGA.

2. The network files are loaded into the inference engine.

3. The input and output of the inference are initialised with layout and precision, in this case NCHW and FP32 for layout and input, respectively, and FP32 for output. The input and output sizes are printed to the console.

4. The model is loaded into the inference engine.

5. An inference request is created based on the network and the input/output precision.

6. The input data precision is set and the input is loaded or generated and stored in the input buffer of the Inference Engine.

7. The inference is performed synchronously, causing the program to halt execution until the inference is completed. A timestamp is taken here to calculate the time spent pre-processing.

8. The output data is read out of the Inference Engine and printed. Additionally, output statistics are read for each layer in the neural network. These statistics include execution time for each layer, whether they were run or not and if they were executed on CPU or FPGA. The result is printed for each iteration, and finally for the whole program.

As the program can run for several iterations, the execution time for each iteration is stored in an array. The execution time is measured in two ways: After the input data is generated but before .wav-creation and MFCC-transform, and based on the execution time for each layer reported by the Inference Engine. The execution times are used for calculating the average, median and shortest times. This is useful as some of the tasks performed executed on the CPU. CentOS is not a real-time system, and the program execution will therefore have lower priority than OS-tasks, causing the execution time to vary each iteration.

4.3.5 Running the program

To run the program, have OpenVINO installed, either with or without FPGA support, for FPGA and CPU, respectively, and have programmed the FPGA with a topology bitstream located in *bitstreams/a10_devkit_bitstreams*. Set up the environment variables by using *source* on either *setup_env.sh* in *fpga_support_files* or *setupvars.sh* in *computer_vision_sdk/bin*. Optimise the model using mo.py in the Model Optimizer, and build the program using *cmake ./* when in the *Hey_Spark_OpenVINO/*-folder. Make the program and run it with the appropriate options. OpenVINO has several pre-combiled bitstreams with different common neural network topologies, including AlexNET, MobileNET and TinyYolo. All topologies have an 11-bit floating point precision.

An example on how to do this for FPGA with 1 iteration is with the following bash-commands:

```
source ../fpga_support_files/setup_env.sh
aocl program acl0 /opt/intel/computer_vision_sdk/bitstreams/
    a10_vision_design_bitstreams/4-0_PL1_FP11_SqueezeNet.aocx
./voice_recognition_OpenVINO network/wakeword.xml network/wakeword.bin 1
    LIVE FPGA RELEASE
```

Running the code will generate printouts during execution, the amount depending on whether the program is run as release or debug.

Chapter 5

Results

This chapter will show the results achieved using OpenVINO to accelerate the application on an Intel Altera Arria 10 GX FPGA accelerator card.

5.1 Test setup specifications

The server at NTNU had the following specifications: HP Z800 workstation with Intel Xeon E5620 2.4 GHz with 8 GB 800 MHz DDR3 RAM, 2 TB Seagate Barracuda Green HDD with 64 MB cache and SATA 6 Gbit s^{-1}, CentOS 7.6.1810 with Linux 3.10.0-957.1.3.el7.x86_64 kernel, HP Z800 Workstation Motherboard 460838-003. Intel OpenVINO 2018 R4 and PortAudio version v.19.06.00 were used in the C++ program, which was built using cmake version 2.8.12.2 and compiled using gcc version 4.8.5 20160623 (Red Hat 4.8.5-28).

5.2 Program execution times

Running the program on CPU and FPGA resulted in the execution times and throughputs listed in table 5.2 for the execution times reported by the Inference Engine. The results are based on the 100 iterations for CPU and for each of the FPGA topologies.

To easily compare the different implementations, figure 5.1 shows the throughput for each topology based on the times reported by the Inference Engine.

5.3 Layer execution times

Table 5.3 shows the execution time for each layer. Note that the execution times for the CPU and FPGA differ both in speed and what layers are executed as some of the layers are accelerated.

	Inference Engine Execution Time[ms]			Throughput [recordings/sec]		
Topology	Average	Median	Minimum	Average	Median	Minimum
CPU	6.86	6.81	5.85	145.73	146.84	171.03
AlexNet	5.29	4.09	3.44	189.00	244.68	290.95
ELU	4.83	4.07	3.71	206.95	245.58	269.69
Generic	5.83	3.83	3.58	171.47	261.37	279.10
MobileNet	5.78	4.48	3.93	173.01	223.02	254.71
SqueezeNet	4.84	3.45	3.10	206.44	290.28	322.27
TinyYolo	5.61	4.17	3.83	178.32	239.92	261.10

Table 5.1: "Hey Spark"-CNN execution times and throughput from the Inference Engine

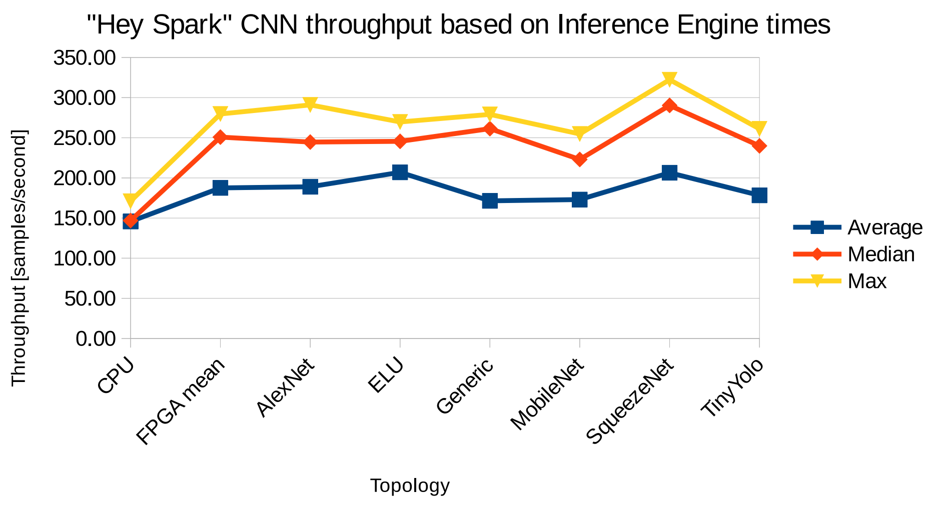


Figure 5.1: "Hey Spark" CNN throughput based on Inference Engine times.

Layer	CPU [µs]	FPGA [µs]
conv1	1002	974
conv2	2767	-
FPGA pre-processing	-	279
FPGA input transfer to DDR	-	594
FPGA execution	-	304
FPGA output from DDR	-	195
FPGA output post-processing	-	60
FPGA copy to IE output blob	-	74
ip1	1428	641
ip2	32	18
ip3	7	5
pool1	93	78
pool1 reorder	-	41
prob	9	10
relu5	8	-
Total	5436	3273

Table 5.2: "Hey Spark"-CNN execution times for each layer on CPU and FPGA

Chapter 6

Discussion

This chapter will discuss the results achieved with OpenVINO and compare OpenVINO and the Xilinx ML Suite on AWS. The comparison will be mostly based on the user experience of using the frameworks. There is also a section for further work, which includes porting Skafså's code to the ML Suite and improving the neural network topology.

6.1 Result analysis

6.1.1 CPU vs FPGA

Looking at table 5.2 and 5.3 in addition to figure 5.1, it's clear that the FPGA accelerates the neural network computations. The sum of the FPGA-parts in table 5.3 is 1547 µs, and based on the reported layers it is layer conv2 which is accelerated. On the CPU, the execution time of conv2 was 2767 µs. relu5 was also apparently performed on the FPGA, but the time it took on the CPU, 8 µs, is negligible. The remaining layers, such as ip1, are run on the CPU, making them susceptible to being halted due to OS-tasks with higher priorities, increasing their execution time. As such, the only point of comparison is layer conv2, and based on that the FPGA provides a speed increase of 79%.

During testing, the possibility of computing all layers on the FPGA was attempted by using only the FPGA-plugin in the Inference Engine API. This resulted in an error message stating that kernel height of layer conv1 divided by the stride was greater than the maximum height allowed supported by the architecture on the FPGA. Editing the neural network to use a shorter height or increasing the stride might lead to layer conv1 being accelerated on the FPGA, increasing the throughput even further.

6.1.2 Test setup influence on the results

The test server at NTNU has several drawbacks which will affect performance:

- The PCIe-port is only generation 1 instead of generation 3, which makes the maximum transfer speed to the FPGA 2 Gbit s^{-1} instead of 8 Gbit s^{-1}.
- The CPU is not officially supported by OpenVINO.
- The relatively slow DDR3 RAM at 800 MHz will cause slower performance compared to a newer system.

Using a modern platform would increase the computational speed substantially. To test this theory, the program was run on a laptop with an Intel i7-7500U CPU with 4 GB of DDR4 RAM and an SSD. The inference was done targeting the CPU, and resulted in a computation time of 3.72 ms, similar to the performance of using the accelerator card on the test server at NTNU.

6.1.3 OpenVINO performance compared to Skafså's OpenCL implementation

Skafså's OpenCL implementation had a classification time of 3.6 ms and Cisco achieved a classification time of 2.2 ms[2]. Skafså does not provide any information on what processor the test computer had, making it more difficult to compare. Nevertheless, the fastest classification time of the OpenVINO implementation, based on the reported time by the Inference Engine, was 3.10 ms. Skafså's implementation had all the convolutional layers on the FPGA while only the second convolutional layer was on the FPGA in the OpenVINO implementation. Comparing table 5.3 and table 6.8 in [2], the layer conv2 were both executed on the FPGA, with an execution time of 1.547 ms and 0.691 ms, respectively. The OpenVINO implementation has additional overhead from transfer to and from the CPU in addition to some pre- and post-processing. Skafså's implementation has the advantage of having all interconnects between the layers on the FPGA. As such, comparing just the execution time on the FPGA, the OpenVINO implementation is faster with 0.304 ms against 0.691 ms.

Skafså's implementation is also custom made, while the OpenVINO implementation runs on a non-custom architecture, which will most likely affect performance.

The overall code size was reduced substantially by using OpenVINO compared to OpenCL, from 1672 to 432, not counting header files, for each project. While OpenCL required all layers to be coded manually, OpenVINO's use of pre-compiled bitstreams and the Model Optimizer help reduce the code size while providing almost the same classification time. OpenVINO has the advantage of being more user friendly, but has the disadvantage of being relatively homogenous, only working on Intel platforms while OpenCL works on many devices and platforms.

6.1.4 Model Optimizer optimisation

Based on the layer information provided by the Inference Engine, using the OpenVINO Model Optimizer on the CNN model by Cisco removed some layers, speeding up the execution. Comparing the information with figure 4.1, layers relu1, relu2, relu3, drop1 and drop2 were optimised out. This might lead to increased performance if all the layers can be implemented on the FPGA through OpenVINO, but this was not the case for this project.

6.2 Intel OpenVINO vs AWS/Xilinx

One of the goals of the project was to compare the OpenVINO toolkit and Xilinx ML Suite. While no inference was done with the ML Suite, it is still possible to do some comparison between the two toolkits.

6.2.1 Feature comparison

- The features of OpenVINO and the Xilinx ML Suite are similar, both including some form of OpenCL-integration with the FPGA SDK for OpenCL and SDAccel, respectively.
- Both feature APIs for using CNN-models for inference using high-level languages such as C++.
- Both are both free of charge and run on accelerator cards, though OpenVINO can run on CPUs, VPUs and Intel GPUs, while it’s not apparent that the ML Suite can deploy on other devices.
- Xilinx xDNN currently supports more bit precisions than OpenVINO, which will most likely lead to faster acceleration when used.
- The cost of accelerating on an FPGA is cheaper with Xilinx due to their partnership with AWS
- OpenVINO can use a CPU while the ML Suite is either on FPGA or simulated on an FPGA.
- OpenVINO has more documentation and code examples than the ML Suite.

6.2.2 Ease of use

While good features might lead to a good product, ease of use is a big factor when it comes to FPGA acceleration as complicated development leads to more work and higher integration costs.

Intel has provided a good amount of documentation with OpenVINO, along with getting-started guides, installation guides and API guides. Installation is easy and the amount of available samples make it easier to understand how to integrate the Inference Engine into your program. As mentioned, some examples are out-dated in regarads to the current API and the recommended program flow, but it is still understandable. Deploying the program to an FPGA instead of CPU required a small change to the program in the plugin-loading.

Based on the FPGA AMI on Amazon Web Services, the amount of documentation is not on the same level as Intel. The main documentation is, seemingly, on Github, and there are not many sample applications which showcase xDNN. Using the software on the server is also cumbersome due to the delay when using X11-forwarding. The overall experience might be better if the ML Suite is installed on a local computer, but the lack of proper documentation and examples is still a problem.

In addition, using Amazon Web Services is difficult. It is a huge platform with lots of different options and possibilities, making it confusing to navigate. It can be difficult to find the information you are looking for, especially considering the vast amount of documentation available.

6.3 Further work

6.3.1 Use the ML Suite locally to port the network to a Xilinx FPGA

As the only attempt to use the ML Suite was done through AWS, the impression from using it will most likely be improved by running it locally. This would remove the need to use AWS and remove the delay from the X11-forwarding. It would give a better insight into the functionality of the ML Suite, in addition to providing more comparison points in terms of user friendliness, as well as give performance statistics for comparison with OpenVINO.

Skafså's OpenCL implementation could possibly have been accelerated using SDAccel by changing using the regular OpenCL-library instead of the aocl-library. This would require the code to be altered, probably on the same scale as porting to OpenVINO, but would provide a good point of comparison for acceleration on the Xilinx FPGAs.

6.3.2 Change the network to map conv1 to the FPGA

Changing the network topology to fit the restrictions of the FPGA architecture of OpenVINO would increase the performance and classification speed. It would also make a for a fairer comparison with Skafså's implementation.

6.3.3 Run the inference on a more modern computer

Running the program and inference on a modern computer would provide more relevant results for today’s computers. It would better show whether accelerating a neural network on an accelerator card would be worth it in terms of performance gains versus price.

Chapter 7

Conclusion

Porting the program to OpenVINO showcased the acceleration capabilities of the Intel Arria 10 GX Development Kit, increasing the classification time of the conv2-layer by 79%. The accuracy of the results can be questioned due to inconsistent execution times on the CPU, caused by the operating system having higher priority than the program. Nevertheless, the average results show that the FPGA accelerated the application by 28%. The fastest classification time was 3.10 ms compared to Skafså's implementation with a time of 3.6 ms, while being slower than Cisco's GPU-implementation at 2.2 ms. Using OpenVINO to accelerate the program reduced the overall code size by almost 75%, while providing classification times on par with the OpenCL implementation. While the use of a relatively old computer might reduce the relevancy of the project, it was still shown that the FPGA acceleration using OpenVINO does provide a performance speedup.

References

[1] J. Fowers, G. Brown, P. Cooke, and G. Stitt, “A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications,” Feb. 2012, pp. 47–56.

[2] O. M. Skafså, “FPGA implementation of a Convolutional Neural Network for "Wake up word" detection,” Master’s thesis, Norwegian University of Science and Technology, 7491 Trondheim, Norway, 2018.

[3] A. Samuel, “Some studies in machine learning using the game of checkers,” *IBM Journal of Research and Development*, vol. 3, 1959.

[4] Google Brain Team, *Tensorflow*, 2017. [Online]. Available: `https://www.tensorflow.org/`.

[5] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” *arXiv preprint arXiv:1408.5093*, 2014.

[6] K. Freund, “A machine learning application landscape and appropriate hardware alternatives,” Mar. 2017.

[7] S. Herculano-Houzel, “The human brain in numbers: A linearly scaled-up primate brain,” *Frontiers in Human Neuroscience*, vol. 3, p. 31, 2009. [Online]. Available: `https://www.frontiersin.org/article/10.3389/neuro.09.031.2009`.

[8] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain,” *Psychological Review*, vol. 65, no. 6, 1958.

[9] S. Grossberg, “Contour Enhancement, Short Term Memory, and Constancies in Reverberating Neural Networks,” *Studies in Applied Mathematics*, vol. 52, no. 3, pp. 213–257, [Online]. Available: `https://onlinelibrary.wiley.com/doi/abs/10.1002/sapm1973523213`.

[10] A. Perez-Uribe, “Artificial Neural Networks: Algorithms and Hardware Implementation,” in *Bioinspired Computing Machines: Towards Novel Computational Architectures*, D. Mange and M. Tomassini, Eds. PPUR Press, 1998, ch. 11, pp. 289–316.

[11] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, “Mastering the game of go without human knowledge,” *Nature*, vol. 550,

[12] Khronos Group, *Opencl*, 2008. [Online]. Available: `https://www.khronos.org/opencl/`.

[13] OpenCV Team, *About - OpenCV library*, Accessed 2018-09-27, 2018. [Online]. Available: `https://opencv.org/about.html`.

[14] J. G. Eldredge and B. L. Hutchings, “Density enhancement of a neural network using FPGAs and run-time reconfiguration,” in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1994, pp. 180–188.

[15] D. Lin, S. Talathi, and S. Annapureddy, “Fixed point quantization of deep convolutional networks,” in *International Conference on Machine Learning*, 2016, pp. 2849–2858.

[16] P. Colangelo, N. Nasiri, A. Mishra, E. Nurvitadhi, M. Margala, and K. Nealis, “Exploration of Low Numeric Precision Deep Learning Inference Using Intel FPGAs,” *arXiv preprint arXiv:1806.11547*, 2018.

[17] Community, *OpenCL Caffe*, 2018. [Online]. Available: `https://github.com/BVLC/caffe/tree/opencl`.

[18] Intel, *Computer Vision Hardware*, Accessed 2018-12-09. [Online]. Available: `https://software.intel.com/en-us/openvino-toolkit/hardware`.

[19] U. Aydonat, S. O’Connell, D. Capalija, A. C. Ling, and G. R. Chiu, “An OpenCL™Deep Learning Accelerator on Arria 10,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’17, Monterey, California, USA: ACM, 2017, pp. 55–64. [Online]. Available: `http://doi.acm.org/10.1145/3020078.3021738`.

[20] Xilinx, “Accelerating DNNs with Xilinx Alveo Accelerator Cards,” Tech. Rep., 2018, Accessed 2018-12-05. [Online]. Available: `https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf`.

[21] PortAudio community, *Portaudio*, version v19.06.00, Dec. 5, 2018. [Online]. Available: `http://www.portaudio.com/`.

[22] C. Rogers, *Cnpy*, version 4e8810b, Jun. 1, 2018. [Online]. Available: `https://github.com/rogersce/cnpy`.

[23] Amazon, *Getting Started with Amazon EC2 Linux Instances*, Accessed 2018-12-09. [Online]. Available: `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EC2_GetStarted.html`.

[24] AWS, *AWS EC2 FPGA Hardware and Software Development Kit*, Accessed 2018-12-09. [Online]. Available: `https://github.com/aws/aws-fpga/`.

[25] F. Boyle, D. Deuermeyer, D. D., and K. O'Neill, *Installation Guide for Intel® Distribution of OpenVINO™ toolkit with Support for FPGA 2*, Accessed 2018-12-10. [Online]. Available: `https://software.intel.com/en-us/articles/installation-guide-for-intel-distribution-of-openvino-toolkit-with-support-for-fpga-2`.

[26] D. Deuermeyer, F. Boyle, A. Z, and A. R, *Inference Engine Developer Guide*, Accessed 2018-12-10. [Online]. Available: `https://software.intel.com/en-us/articles/OpenVINO-InferEngine`.

Appendix A

Source code

A.1 voice_recognition_OpenVINO.cpp

```cpp
#include <iostream>
#include <map>
#include <fstream>
#include <string>
#include <algorithm>
#include <iomanip>
#include <inference_engine.hpp>
#include "record_voice.h"
#include "write_wav.h"
#include <cstdlib>

using namespace InferenceEngine;

#define INPUT_W      40
#define INPUT_H      90
#define INPUT_SIZE   INPUT_W*INPUT_H

// Global buffer array for the input audio
float input_audio[INPUT_SIZE];

// Path of the recorded input audio file and sample
const char* input_processed_recording_name = "recordings/
    voice_rec_live_processed";
```

```
const char* input_processed_sample_name = "recordings/
    data.csv";

// Function prototypes
void load_sample(float* input_buffer);
void record_input(float* input_buffer);
double getCurrentTimestamp();

int main(int argc, char *argv[]) {
  // Input argument variables
  std::string DEVICE = "CPU";
  int DEBUG = 0;
  int USE_SAMPLE = 0;
  if(argc != 7){
    std::cerr << "Incorrect amount of arguments, use : [
        NETWORK_PATH XML] [WEIGHTS_PATH BIN] [NUM
        ITERATIONS] [SAMPLE or LIVE] [CPU or FPGA] [
        RELEASE or DEBUG]" << std::endl;
    return 1;
  }
  std::string NETWORK = std::string(argv[1]);
  std::string WEIGHTS = std::string(argv[2]);
  const int NUM_LOOPS = std::stoi(argv[3]);
  if(std::string(argv[4]) == "SAMPLE")
    USE_SAMPLE = 1;
  if(std::string(argv[5]) == "FPGA")
    DEVICE = "HETERO:FPGA,CPU";
  if(std::string(argv[6]) == "DEBUG")
    DEBUG = 1;

  // Timestamps
  long long execution_time_buffer[NUM_LOOPS];

  // Processing time average
  long long processing_avg = 0;

  // 1. Load plugin
  if(DEBUG)
    std::cout << "Loading Plugin" << std::endl;
```

```
InferencePlugin plugin = PluginDispatcher({"../../../
    lib/intel64", ""}).getPluginByDevice(DEVICE);

// 2. Read intermediate representation
if(DEBUG)
  std::cout << "Reading intermediate representation" <<
       std::endl;
CNNNetReader network_reader;
network_reader.ReadNetwork(NETWORK);
network_reader.ReadWeights(WEIGHTS);
CNNNetwork network = network_reader.getNetwork();

// 3. Configure input and output
// Get input info and set input layout and precision
if(DEBUG)
  std::cout << "Configuring input and output" << std::
      endl;
InputInfo::Ptr input_info = network.getInputsInfo().
    begin()->second;
std::string input_name = network.getInputsInfo().begin
    ()->first;

input_info->setLayout(Layout::NCHW);
input_info->setPrecision(Precision::FP32);

// Print input info
std::cout << "Getting network info" << std::endl;
static size_t num_channels = input_info->getTensorDesc
    ().getDims()[1];
static size_t width = input_info->getTensorDesc().
    getDims()[3];
static size_t height = input_info->getTensorDesc().
    getDims()[2];
if(DEBUG){
  std::cout << "Num. input channels: " << num_channels
      << std::endl;
  std::cout << "Input dimensions: " << width << "x" <<
      height << std::endl;
}

```

```
// Get output info and set output precision
OutputsDataMap outputInfo(network.getOutputsInfo());
std::string firstOutputName;

for (auto & item : outputInfo)
{
  if(firstOutputName.empty())
  {
    firstOutputName = item.first;
  }
  DataPtr outputData = item.second;
  if(!outputData)
  {
    std::cout << "Data output pointer is invalid" <<
        std::endl;
  }
  item.second->setPrecision(Precision::FP32);
}
// Print output info
const SizeVector outputDims = outputInfo.begin()->
    second->getDims();
if(DEBUG)
  std::cout << "Output dims: " << outputDims[0] << "x"
      << outputDims[1] << std::endl;

// 4. Load the model
if(DEBUG)
  std::cout << "Loading model" << std::endl;
auto executable_network = plugin.LoadNetwork(network,
    {});


// 5. Create infer request
if(DEBUG)
  std::cout << "Creating infer request" << std::endl;
auto infer_request = executable_network.
    CreateInferRequest();

// 6. Prepare input
if(DEBUG)
```

```
  std::cout << "Assigning input data" << std::endl;
Blob::Ptr input = infer_request.GetBlob(input_name);
auto input_data = input->buffer().as<PrecisionTrait<Precision::FP32>::value_type *>();

// Loop for either inference speed testing or speech recognition testing
for(int loop = 0; loop < NUM_LOOPS; loop++)
{
  if(DEBUG)
    std::cout << "Preparing data" << std::endl;
  if(USE_SAMPLE){
    std::cout << "Using sampled data" << std::endl;
    load_sample(&input_audio[0]);
  }
  else
    record_input(&input_audio[0]);

  std::cout << "Filling input buffer" << std::endl;
  for(int i = 0; i < num_channels*width*height; i++)
  {
    input_data[i] = input_audio[i];
  }

  // 7. Start synchronous inference
  if(DEBUG)
    std::cout << "Starting synchronous inference" << std::endl;
  infer_request.Infer();

  // 8. Process output data
  if(DEBUG)
    std::cout << "Retrieving output data" << std::endl;
  Blob::Ptr output = infer_request.GetBlob(firstOutputName);

  // Get performance statistics for each layer
  if(DEBUG)
    std::cout << "Getting performance statistics for each layer" << std::endl;
  auto Info = infer_request.GetPerformanceCounts();
  std::map<std::string, InferenceEngineProfileInfo >::
      iterator it = Info.begin();
  if(DEBUG)
    std::cout << "----PERFORMANCE STATISTICS----" <<
        std::endl;
  execution_time_buffer[loop] = 0;
  while(it != Info.end())
  {
    InferenceEngineProfileInfo layer_info= it->second;
    execution_time_buffer[loop] += layer_info.
        realTime_uSec;
    if(DEBUG)
    {
      std::cout << "Performance stats for: " << it->
          first << std::endl;

      std::string status = layer_info.status == 0 ? "
          Not run":
                              layer_info.status == 1 ? "
                                  Optimized out":
                                                      "
                                                          Executed
                                                          "
                                                          ;

      std::cout << "Layer status: " << status << std::
          endl;
      if(status != "Not run"){
        std::cout << "Exec type: " << layer_info.
            exec_type << std::endl;
        std::cout << "layer type: " << layer_info.
            layer_type << std::endl;
        std::cout << "Realtime run: " << layer_info.
            realTime_uSec << "us" << std::endl;
      }
      std::cout << "----" << std::endl;
    }
    it++;
```

```cpp
    }

    std::cout << "Results:" << std::endl;
    // Print output data and execution time
    auto output_data = output->buffer().as<PrecisionTrait<Precision::FP32>::value_type*>();
    std::cout << "Neural Network output" << std::endl;
    for(int i = 0; i < outputDims[0] * outputDims[1]; i++)
    {
      std::cout << output_data[i] << std::endl;
    }

    std::cout << "Execution time from IE:      " << std::setprecision(4) << double(execution_time_buffer[loop])/1000 << "ms" << std::endl;
    std::cout << "----------------------------------------" << std::endl;

    // Add to running average and buffer
    processing_avg += execution_time_buffer[loop];
  }

  // Calculate average and sort buffer
  double average = processing_avg / NUM_LOOPS;

  // Sort the time arrays
  std::sort(execution_time_buffer, execution_time_buffer + NUM_LOOPS);

  // Print time and throughput calculations
  std::cout << "Inference Engine measurements" << std::endl;
  std::cout << "Time          Avg: " << std::setprecision(4) << average/1000 << "ms.          Median: "
            << double(execution_time_buffer[int(NUM_LOOPS/2)])/1000 << "ms.          Fastest: "
            << double(execution_time_buffer[0])/1000 << "ms." << std::endl;

  std::cout << "Throughput  Avg: " << 1000*1000/average << " samples/s. Median: "
            << 1000*1000/double(execution_time_buffer[int(NUM_LOOPS/2)]) << " samples/s. Fastest: "
            << 1000*1000/double(execution_time_buffer[0]) << " samples/s." << std::endl;
}

////////// HELPER FUNCTIONS ///////////////////////////////
/*  load_sample
*
*   Function for loading the sample data saying "Hey Spark"
*   from a .csv file to the input_audio buffer via the
*   input_buffer pointer
*/

void load_sample(float* input_buffer){
  std::cout << "Loading array" << std::endl;
  std::ifstream in(input_processed_sample_name);
  std::string line;
  int i = 0;
  while(getline(in, line))
  {
    std::stringstream ss(line);
    std::string data;
    std::string::size_type string_size;
    while(getline(ss, data, ','))
    {
      input_buffer[i] = std::stof(data, &string_size);
      i++;
    }
  }
  std::cout << "Read " << i << " input values" << std::endl;
}

/*  record_input
*
```

```
*   Function for calling record_voice in record_voice.c and
*   store the output data to a .wav-file using write_wav in write_wav.c
*   Also calls and the MFCC-preprocess function and reads the data to
*   input_audio via the input_buffer pointer.
*/

void record_input(float* input_buffer)
{
  // Allocate memory and record voice
  std::cout << "Allocating memory for recording" << std::endl;
  int num_bytes = NUM_SECONDS * SAMPLE_RATE  * NUM_CHANNELS * sizeof(float);
  float* recorded_samples = (float*)malloc(num_bytes);

  // Start recording
  record_voice(recorded_samples);
  std::cout << "Recorded " << num_bytes << " of data" << std::endl;

  // Export .wav file
  write_wav("recordings/voice_rec_live.wav", recorded_samples, num_bytes, NUM_CHANNELS, SAMPLE_RATE, 32); //

  // Preprocess data
  system("preprocessing/mfcc_preprocess --input recordings/voice_rec_live.wav --output recordings/voice_rec_live_processed");

  // Read preprocessed data
  std::ifstream input_file(input_processed_recording_name, std::ios::in | std::ios::binary);
  if(input_file.is_open())
  {
    input_file.read((char*)(input_buffer), sizeof(float)*INPUT_SIZE);
    input_file.close();
  }
  else
  {
    std::cout << "Input file could not be read" << std::
        endl;
  }
  free(recorded_samples);
}
double getCurrentTimestamp(){
  timespec a;
  clock_gettime(CLOCK_MONOTONIC, &a);
  return(1000*((double(a.tv_nsec) * 1.0e-9) + double(a.
      tv_sec)));
}
```

A.2 record_voice.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <ctime>
#include "portaudio.h"
#include "record_voice.h"

void record_voice(float* recorded_samples) {

  PaStreamParameters input_parameters;
  PaStream* stream;
  PaError err;
  int total_frames;
  int num_samples;
  int num_bytes;
  float max, average, val;

  total_frames = NUM_SECONDS * SAMPLE_RATE;
  num_samples = total_frames * NUM_CHANNELS;
  num_bytes = num_samples * sizeof(float);
  if (recorded_samples == NULL){
    printf("Could not allocate record array.\n");
    exit(1);
  }

  for (int i = 0; i < num_samples; i++)
    recorded_samples[i] = 0;

  err = Pa_Initialize();
  if (err != paNoError)
    goto error;

  input_parameters.device = Pa_GetDefaultInputDevice();
  if (input_parameters.device == paNoDevice) {
    fprintf(stderr, "Error: No default input device.\n");
    goto error;
  }
  input_parameters.channelCount = NUM_CHANNELS;
  input_parameters.sampleFormat = PA_SAMPLE_TYPE;
input_parameters.suggestedLatency = Pa_GetDeviceInfo(input_parameters.device)->defaultLowInputLatency;
input_parameters.hostApiSpecificStreamInfo = NULL;

err = Pa_OpenStream(&stream,
                    &input_parameters,
                    NULL,              // &output_parameters
                    SAMPLE_RATE,
                    FRAMES_PER_BUFFER,
                    paClipOff,         // we won't output out of range samples so don't bother clipping them
                    NULL,              // no callback, use blocking api
                    NULL);             // no callback, so no callback userData
if (err != paNoError)
  goto error;

err = Pa_StartStream(stream);
if (err != paNoError)
  goto error;
printf("\n------NOW RECORDING!!------\n\n"); // fflush(stdout);


err = Pa_ReadStream(stream, recorded_samples, total_frames);
if (err != paNoError)
  goto error;

err = Pa_CloseStream(stream);
if (err != paNoError)
  goto error;

Pa_Terminate();

return;

```

```
error:
  Pa_Terminate();
  fprintf(stderr, "An error occured while using the
      portaudio stream\n");
  fprintf(stderr, "Error number: %d\n", err);
  fprintf(stderr, "Error message: %s\n", Pa_GetErrorText(
      err));
  return;
}
```

A.3 write_wav.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <fstream>

#include "write_wav.h"



void write_wav(const char* file_name, float* audio_data, int num_bytes, short num_channels, int sample_rate, short bits_per_sample) {
  wav_header_t header;
  // RIFF wave header
  header.chunk_id[0] = 'R';
  header.chunk_id[1] = 'I';
  header.chunk_id[2] = 'F';
  header.chunk_id[3] = 'F';
  header.chunk_size = 36 + num_bytes;
  header.format[0] = 'W';
  header.format[1] = 'A';
  header.format[2] = 'V';
  header.format[3] = 'E';

  // Format subchunk
  header.subchunk1_id[0] = 'f';
  header.subchunk1_id[1] = 'm';
  header.subchunk1_id[2] = 't';
  header.subchunk1_id[3] = ' ';
  header.subchunk1_size = 16;                // 16 for PCM, size for rest of subchunk
  header.audio_format = 3;                   // 1 for PCM, 3 for float it seems
  header.num_channels = num_channels;
  header.sample_rate = sample_rate;
  header.bits_per_sample = bits_per_sample;
  header.byte_rate = header.sample_rate * header.num_channels * header.bits_per_sample/8;
  header.block_align = header.num_channels * header.
```

```
        bits_per_sample/8;
/*
    // Fact subchunk
    header.subchunk2_id[0] = 'f';
    header.subchunk2_id[1] = 'a';
    header.subchunk2_id[2] = 'c';
    header.subchunk2_id[3] = 't';
    header.subchunk2_size = 4;
    header.unknown2_1 = 43207;

    // PEAK subchunk
    header.subchunk3_id[0] = 'P';
    header.subchunk3_id[1] = 'E';
    header.subchunk3_id[2] = 'A';
    header.subchunk3_id[3] = 'K';
    header.subchunk3_size = 16;
    header.unknown3_1 = 1;
    header.unknown3_2 = 0x59036d67;
    header.unknown3_3 = 0x3ealf380;
    header.unknown3_4 = 0x24fc;
*/
    // Data subchunk
    header.subchunk2_id[0] = 'd';
    header.subchunk2_id[1] = 'a';
    header.subchunk2_id[2] = 't';
    header.subchunk2_id[3] = 'a';
    header.subchunk2_size = num_bytes; // or num_samples *

    // Write
    std::ofstream file(file_name, std::ios::binary); // std::ios::out
    file.write((char*)&header, sizeof(header));
    file.write((char*)audio_data, num_bytes);
    file.close();

}
```