Norwegian University of
Science and Technology

FPGA implementation of a Convolutional Neural Network for "Wake up word" detection

Ole Martin Skafså

Master of Science in Electronics
Submission date: June 2017
Supervisor: Kjetil Svarstad, IES
Co-supervisor: Florian Bochud, Cisco Systems Norway AS

Norwegian University of Science and Technology
Department of Electronic Systems

Project Assignment

Candidate name: Ole Martin Skafså

Assignment title: FPGA implementation of a Convolutional Neural Network for “Wake up word” detection

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 would be a natural device to implement that kind of processing, since it provides several thousand multiplications blocks that all can be run in parallel.

This master thesis will explore a “Wake up word” Convolutional Neural Network (CNN) implementation on FPGAs. This CNN detects a specific wake-up word and gives a single bit as an output.

To achieve that, the student to work on the following tasks:

- Setup an infrastructure to test various neural network. The CNN will be implemented on an Altera SOC dev kit. The infrastructure should provide an easy way to feed pattern into the network and retrieve back the result.
- Neural network with increasing complexity should be implemented and tested. The “wake up word” implementation, due to its size and complexity, would be implemented last.
- Ideally the implementation should be generalized to any type of CNN application.
- The result should be compared with a GPU implementation in terms of latency, power and cost.

In addition using High-level synthesis tool is required (Altera I++ or OpenCL).

Responsible professor: Kjetil Svarstad, IET

Supervisor: Florian Bochud, Cisco

Abstract

The popularity of machine learning has increased dramatically in the last years and the possible applications varies from web search, speech recognition, object detection, etc. A big part of this development is due to the use of Convolutional Neural Networks (CNNs), where high performance Graphics Processing Units (GPUs) has been the most popular device.

This thesis explores the use of a Field-Programmable Gate Array (FPGA), specifically an Arria 10 GX FPGA, to implement a "wake up word" CNN. The High-Level Synthesis (HLS) tool Intel FPGA SDK for OpenCL was used. During the project various neural networks has been implemented and tested on the FPGA with different attributes to understand their effect. An infrastructure to test various neural networks was made and used to implement the wake up word CNN. A solution to test the CNN in a setup with live recording was also made.

The final implementation of the wake up word CNN achieved a classification time of 3.6 ms and 0.54 Gmac/s, where a *mac* is the multiply-accumulate operation. Comparing to a CNN runnning on a NVIDIA Tegra X1 GPU, the GPU was 22.2 times faster with 11.99 Gmac/s. Although the classification time of 3.6 ms is acceptable for this application, future work should attempt to keep as much of the computation and memory transfers on the FPGA chip and with minimal interaction with the host machine to improve performance.

Sammendrag

Populariteten til maskinlæring har økt dramatisk de siste årene og de mulige applikasjonene varierer fra web søk, talegjenkjenning objektdeteksjon, osv. En viktig grunn for denne utviklingen er bruken av konvolusjonsnettverk, hvor grafikkprosessorer (GPUer) har vært den mest populære prosesseringsenheten.

Denne avhandlingen utforsker bruken av Field-Programmable Gate Array (FPGA), og da spesifikt en Arria 10 GX FPGA, for å implementere et konvolusjonsnettverk som kjenner igjen et "oppvåkningsord". For å utføre oppgaven har høy-nivå syntese (HLS) verktøyet Intel FPGA SDK for OpenCL blitt brukt. I løpet av prosjektet har forskjellige nevrale nettverk blitt implementert og testet på FPGAen med forskjellige attributter for å teste effekten de har. En infrastruktur for å teste forskjellige nevrale nettverk ble lagd og den ble brukt til å implementere konvolusjonsnettverket for "oppvåkningsordet". En løsning for å teste konvolusjonsnettverket med direkte opptak ble også laget.

Den endelige implementasjonen av konvolusjonsnettverket oppnådde en klassifikasjonstid på 3,6 ms og 0,54 Gmac/s, der en *mac* er multipliser-akkumuler operasjonen. Dette ble sammenlignet med et konvolusjonsnettverk kjørende på en Tegra X1 GPU, GPUen var 22,2 ganger raskere med 11,99 Gmac/s. Selv om en klassifikasjonstid på 3,6 ms er akseptabelt for denne applikasjonen, så bør fremtidig arbeid forsøke å beholde så mye som mulig av utregninger og dataoverføringer på FPGAen og med minimal interaksjon med vertsmaskinen for å øke ytelsen.

Preface

This report is the result of the Master's thesis conducted during the spring of 2017. It concludes a Master of Science degree in Electronics, with a specialization in Design of Digital Systems. The report is submitted to the Department of Electronic Systems at the Norwegian University of Science and Technology

The project *FPGA implementation of a Convolutional Neural Network for "Wake up word" detection* was proposed by Cisco. Cisco also provided the FPGA used in the project as well as other helpful support. During the work on the thesis I have learned a lot about OpenCL, Convolutional Neural Networks and FPGAs. I also did a project about High-Level Synthesis for FPGA in the autumn of 2016, which was also for Cisco and the same supervisors.

I would like to thank my supervisors Florian Bochud at Cisco and professor Kjetil Svarstad at NTNU, for valuable support, guidance and feedback through the project. I also want to thank my family and friends for their support and encouragement during this project and through the whole degree.

Ole Martin Skafså

Trondheim, 19th June 2017

Contents

1 Introduction **1**
1.1 Background and Motivation . . . 1
1.2 Contributions . . . 1
1.3 Thesis overview . . . 2

2 Background Theory **3**
2.1 Machine learning . . . 3
2.1.1 Neural networks . . . 4
2.1.2 Network training . . . 5
2.1.3 Convolutional neural network . . . 6
2.1.4 Layers . . . 7
2.2 OpenCL . . . 9
2.2.1 Intel FPGA SDK for OpenCL . . . 11
2.3 Speech signal preprocessing . . . 12

3 Related Work **15**
3.1 An OpenCL™Deep Learning Accelerator on Arria 10 . . . 15
3.2 PipeCNN . . . 16
3.3 FINN . . . 17

4 Neural Networks Implementation **19**
4.1 Linear classifier for the MNIST dataset . . . 19
4.1.1 Training . . . 20
4.1.2 OpenCL Implementation . . . 20
4.2 One hidden layer neural network . . . 23
4.2.1 Two hidden layer neural network . . . 26

5 CNN Implementation **27**
5.1 Architecture . . . 27
5.2 OpenCl implementation . . . 29

5.2.1 Getting it live 33

6 Results and Evaluation **37**
6.1 Experimental setup 37
6.2 Compiling and retrieving results 38
6.3 Resource utilization 39
6.3.1 Neural nets 40
6.3.2 CNN 42
6.4 Performance 43
6.4.1 Neural nets 43
6.4.2 CNN 45
6.5 Comparison 46

7 Discussion **47**
7.1 Project tasks 47
7.2 Results and improvements discussion 48

8 Conclusion **51**
8.1 Future Work 52

References **53**

Appendix **57**
A.1 Source code 57
A.1.1 Linear classifier 57
A.1.2 One hidden layer neural network 58
A.1.3 Two hidden layers neural network 61
A.1.4 CNN 77
A.1.5 Live setup code 101
A.2 Reports 105

List of Figures

2.1 Perceptron model and equation [26] 4
2.2 Activation functions [31] 5
2.3 Example of a CNN architecture, the LeNet-5. Each plane is a feature map [22]. 7
2.4 Convolution and pooling layer illustration [10]. 9
2.5 OpenCL platform model [20]. 10
2.6 NDRange index space example [24]. 11
2.7 OpenCL-to-FPGA framework [12]. 12
2.8 Mel-spaced filter bank for generating MFCCs [13]. 13

3.1 Overall DLA architecture. 16
3.2 PipeCNN architecture. 16
3.3 Generating an FPGA accelerator from a trained BNN [34]. 17

4.1 Example neural network [26]. 24

5.1 Topology of implemented wake up word CNN. 28
5.2 WAV format [2]. 35

6.1 Arria 10 GX FPGA Development Kit [15]. 38

List of Tables

6.1 Arria 10 GX resources [16] 39
6.2 Linear classifier resource utilization 40
6.3 One hidden layer NN resource utilization 40
6.4 Two hidden layer NN resource utilization 41
6.5 CNN resource utilization 42
6.6 Total resource utilization for NDRange conv 4 CUs 43
6.7 Neural networks performance 44
6.8 CNN performance 45

Acronyms

ALM Adaptive Logic Module.

ALUT Adaptive LUT.

AOC Altera Offline Compiler.

API Application Programming Interface.

BNN Binarized Neural Network.

BRAM Block RAM.

BSP Board Support Package.

CNN Convolutional Neural Network.

CPU Central Processing Unit.

CU Compute Unit.

DCT Discrete Cosine Transform.

DFT Discrete Fourier Transform.

DMA Direct Memory Access.

DNN Deep Neural Network.

DSP Digital Signal Processing.

FLOPS Floating point operations per second.

FPGA Field-Programmable Gate Array.

GPGPU General Purpose Graphics Processing Unit.

GPU Graphics Processing Unit.

HLS High-Level Synthesis.

LE Logic Element.

LUT Look-Up Table.

mac multiply–accumulate operation.

MFCC Mel-frequency cepstrum coeffisients.

MLAB Memory Logic Array Block.

MNIST Modified National Institute of Standards and Technology database.

OpenCL Open Computing Language.

OPS Operations per second.

PCIe Peripheral Component Interconnect Express.

PE Processing Element.

SDK Software Development Kit.

SIMD Single Instruction Multiple Data.

SW Software.

WAV Waveform Audio File Format.

Chapter 1

Introduction

1.1 Background and Motivation

The popularity of machine learning has increased dramatically in the last years, a big part of this development is due to the use of Convolutional Neural Networks (CNNs). Also the use of Graphics Processing Units (GPUs) as General Purpose Graphics Processing Units (GPGPUs), together with the rapid growth of the internet and easily available data has helped to boost the performance of CNNs. The possible uses of machine learning and Convolutional Neural Networks are many, some examples are: object detection, playing games or speech recognition, which will be explored in this report.

There are huge amounts of data being processed in Convolutional Neural Networks, where the computations often are solved with matrix-multiplications. This project will explore the use of an Field-Programmable Gate Array (FPGA) to implement a CNN for "wake up word" recognition. The inherent parallelism of FPGAs makes them natural to consider for this kind of processing.

In order to implement the CNN a High-Level Synthesis (HLS) tool from Intel will be used. Both the Intel FPGA SDK for OpenCL and Intel HLS Compiler are possible options. However, the Intel FPGA SDK for OpenCL was deemed most appropriate for this project due to the time available.

1.2 Contributions

The goal of this thesis have been to explore the implementation of CNNs on FPGAs, specifically to detect a "wake up word". An important part has also been to use

OpenCL and the Intel FPGA SDK for OpenCL to do the implementations.

A summary of the contributions made in this projects is listed below:

- Different neural networks with various complexity has been implemented and tested with different attributes to understand their effect.
- An infrastructure to test various neural networks consisting of the implemented convolution and fully connected layers has been made.
- With this infrastructure a CNN used to recognize a "wake up word" has been implemented.
- In order to test the wake up word CNN in a live setup, a way to record audio and perform preprocessing on the audio was necessary. Cisco provided a program for the preprocessing, while the recording was implemented using PortAudio library.
- Results from the implementations has been presented and evaluated based on resource utilization and performance. The CNN results are also compared against an GPU implementation.

1.3 Thesis overview

The thesis is divided into 8 chapters including this introductory chapter. Chapter 2 introduces some necessary background regarding machine learning and neural networks, some info on OpenCL and the Intel FPGA SDK for OpenCL, and also relevant theory for speech signal preprocessing. Chapter 3 presents some recent related work on the field. In Chapter 4, the implementation of three initial neural networks with increasing complexity is described, while Chapter 5 explains the implementation of the "wake up word" CNN, including some work to be able to test it with live speech. In Chapter 6 the implemented neural networks' results are presented and evaluated based on resource utilization and performance. A comparison between the FPGA implementation of the CNN, and a CNN implemented on a GPU is also included. Chapter 7 discusses the presented solution and its results with regards to the project tasks and the related work. Finally the thesis is concluded in Chapter 8, and proposes some thoughts and possible routes for future work. An Appendix is also included, it contains listings of the source code and summary reports for the final design.

Chapter 2

Background Theory

In this chapter some background and theory necessary to understand this subject is presented. First machine learning and neural networks are introduced, including some theory on training the networks. Then CNNs and its layers are presented. Some info on OpenCL and the Intel FPGA SDK for OpenCL is also included. The last section presents theory on speechh signal preprocessing relevant to the "wake up word" CNN.

2.1 Machine learning

Machine learning stems from computer science and is *a field of study that gives computers the ability to learn without being explicitly programmed* as originally defined by Arthur L. Samuel in 1959 [30]. Another newer and more formal definition was made by Tom M. Mitchell in [23] is:

> *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.*

One example that shows this is a computer program learning to recognize handwriting. Then the task T will be to recognize and classify handwritten words within images, the performance measure P is the percent of words correctly classified, and training experience E a database of words with known classifications.

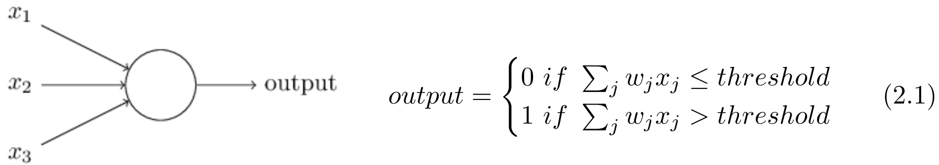
Machine learning has many applications including image classification, object detection, speech recognition, even learning to play games, etc.

A subset of machine learning are neural networks, also called Deep Neural Networks (DNNs) when the network consists of multiple layers. These networks are the focus

in this thesis, and specifically on Convolutional Neural Networks (CNNs)

2.1.1 Neural networks

Neural networks is one type of machine learning model which has had great practical value in the field of pattern recognition. The term *neural network* has its origin in attempts to find mathematical representations of information processing in biological systems, where the *perceptron* was one of the big influential outcomes of this research [10]. The perceptron is an artificial neuron developed by Frank Rosenblatt in the 1950s and 1960s, it takes several binary inputs, x_1, x_2, ..., and produces a single binary output. This is shown in Figure 2.1 along with its mathematical description.



$$output = \begin{cases} 0 \; if \; \sum_j w_j x_j \leq threshold \\ 1 \; if \; \sum_j w_j x_j > threshold \end{cases} \tag{2.1}$$

Figure 2.1: Perceptron model and equation [26]

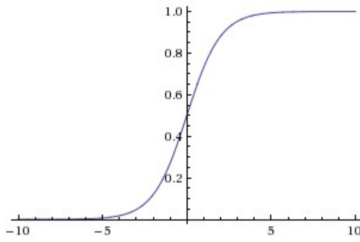
Rosenblatt introduced *weights*, w_1, w_2, ..., and the *threshold* value, which are all real numbers and parameters of the perceptron. Based on these parameters the perceptron will either output 1 or 0, depending on the input, as shown in the figure above.

The current models of neurons are similar to Rosenblatts perceptron in many ways. Instead of the threshold value, the *bias*, b is introduced and is defined as $b \equiv -threshold$. In addition an activation function is introduced, its purpose is to allow small changes in the weights or bias to only cause a small change on the output, this property is helpful when training training a network. For the perceptron neuron such small changes could cause the output to flip, e.g. from 0 to 1. The new model and definition of a neuron is shown in the equation below:

$$y = f(\mathbf{w} \cdot \mathbf{x} + b) = f(\sum_j w_j x_j + b) \tag{2.2}$$

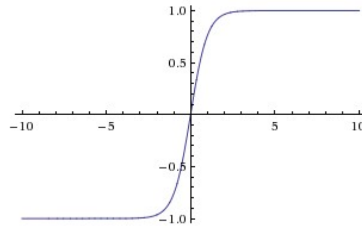
Where the output y, is given by the dot product the vector $\mathbf{w}$, containing the neurons weights, and the input vector $\mathbf{x}$, plus the bias. $f(\cdot)$ is the nonlinear activation function, in Figure 2.2 three of the most commonly used activation functions are displayed along with their equations.

A neural network consists of many neurons usually organized into multiple layers. The first layer of a neural network is the input layer, it is followed by one or more hidden layers. They are called *hidden layers* since the neurons in these layers are neither inputs nor outputs. After the last hidden layer, the output layer follows.



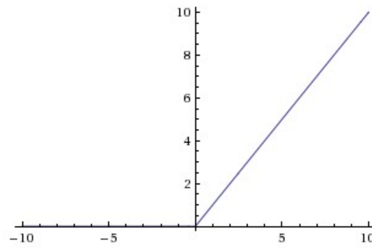
$$\sigma(z) = \frac{1}{1+e^{-z}} \qquad (2.3)$$

(a) Sigmoid



$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \qquad (2.4)$$

(b) tanh



$$RelU(z) = max(0, z) \qquad (2.5)$$

(c) RelU

Figure 2.2: Activation functions [31]

The number of neurons in the output layer depends on the depends on the task. For example to classify handwritten digits, it would be natural to use 10 output neurons, one for each digit. An example neural network with one hidden layer is shown in Figure 4.1. Neural networks where all the inputs to a layers neurons stems from the previous layer is called a *feedforward* neural network. If connections between neurons can form a directed cycle in the network, it would be called a *recurrent* neural network [26]. The focus here however will be on feedforward networks.

2.1.2 Network training

The process of training or learning a network involves a way to optimize the weights and biases of the network. As training of the networks are outside the scope of this project, only a general introduction of network training will be explained here.

In order to train a network a set of input vectors $\{\mathbf{x}_n\}$, where $n = 1, ..., N$, together

with a corresponding set of target vectors $\{\mathbf{x}_n\}$, is necessary. A cost function, $C(w, b)$, is introduced, and the goal is to minimize this function to get better classification results:

$$C(w, b) = \frac{1}{2N} \sum_{n=1}^{N} ||\mathbf{y}(\mathbf{x}_n) - \mathbf{t}_n||^2 \tag{2.6}$$

Here, w and b denotes the collection of all the weights and biases in the network. To minimize the cost function an algorithm known as *gradient decent* is used. The idea with gradient descent is to alter the values of the weights and biases by updating them with small steps in the direction of the negative gradient. This update for each weight component w_k and b_l is given by:

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C(w, b)}{\partial w_k} \tag{2.7}$$

$$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C(w, b)}{\partial b_l} \tag{2.8}$$

These updates are performed many times so the cost function converges towards a local or global minimum. The parameter η, is called the learning rate, and it decides how fast the cost function converges. Choosing too large η may cause the cost function to increase and thus not converge. Parameters such as η are called hyperparameters, they are not trained like the weights and biases, but must still be chosen appropriately and possibly be fine tuned [31].

Depending on the number of training inputs, the training time can be very long. A solution to speed up the process is called *stochastic gradient descent*, instead of using all N training inputs, a small number of samples are randomly chosen from the training set. This way each update operation from Equations 2.7 and 2.8 is performed faster.

An algorithm called the *backpropagation algorithm* is one of the most important approaches to train neural networks today since it provides a fast way to compute the gradient of the cost function. It will not be explained in detail here, but the idea is to first do a forward pass with an input vector $\mathbf{x}_n$ in order to find the activations of all the neurons. Next error values on the outputs are propagated backwards through the network, which then are used to calculate the gradients and perform the updates [26][10].

2.1.3 Convolutional neural network

The Convolutional Neural Network (CNN) is a feedforward neural network, meaning there can be no loops in the network. CNNs are very similar to ordinary neural networks, they are still made up of neurons that have learnable weights and biases.

The main difference between an ordinary neural network and a CNN is that besides the fully connected layers, there are two additional main layers used in CNN architectures: *convolutional layer* and *pooling layer*. These layers are stacked several times to form a CNN [31].

Figure 2.3 shows an example of a CNN, which consists of 7 layers (excluding the input). The first four layers are pairs of a convolutional layer followed by a pooling or subsampling layer. The fifth layer C5, is also a convolutional layer, however since S4's feature maps are 5x5, same as the filter sizes in C5, the layer is equivalent to a fully connected layer. The last two layers are a fully connected layer and the output layer.

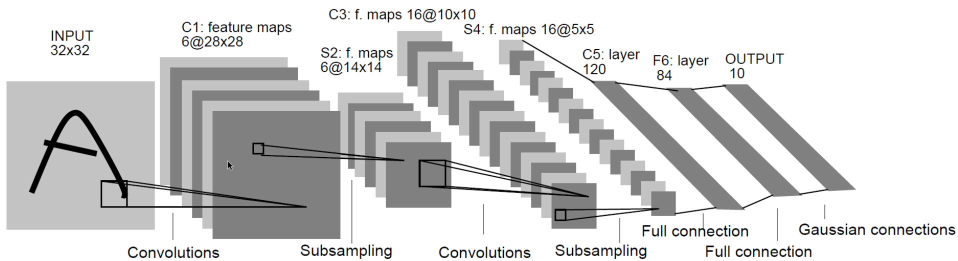


Figure 2.3: Example of a CNN architecture, the LeNet-5. Each plane is a feature map [22].

The next subsection will explain the main layers in a CNN, and they will show that CNNs use three basic ideas: *local receptive fields*, *shared weights and biases* and *pooling*.

2.1.4 Layers

Convolutional layer The convolutional layer is the core building block of a CNN. In the regular neural networks considered earlier, which are the same as a fully connected layer, the input is a vector, meaning 2D images are stretched into a vector. This means that a key property of images are ignored, which is that nearby pixels are more strongly correlated than more distant pixels. In a convolutional layer the input keeps its original shape in order to exploit this correlation between the pixels, and by using *local receptive fields* or small subregions in the input image, the convolutional layer is able to extract local features. The extraction is done by performing a convolution operation on the input image, using a kernel that acts like a filter. This can be thought of as sliding the filter across the entire image, and for each subregion in the input that the filter covers, a dot product operation is done between the weight values in the filter and the region, and the result is a value in what is called a *feature map*. The idea of *weight sharing* means that all the units of a feature map are constrained to share the same weight values, the filter. Using several filters in the convolution layer will yield several feature maps on the output, meaning more features can be extracted. After the dot product is

performed, a bias is added to every element in the output feature map, the bias is also shared across the entire feature map. By training the network the filter weights can be configured to extract the features needed to recognize the desired classes.

As an example consider layer C3 in the LeNet-5 architecture shown in Figure 2.3. This is the second convolution layer and gets 6, 14x14 feature maps from the previous layer, these are then convoluted with 16 filters which leads to 16 output feature maps. In general the filters has the same depth as the number of input feature maps, and typical widths and heights of the filters are 3x3 and 5x5, but this varies.

The filters spatial extent, i.e. width and height are hyperparameters, there are in total 4 hyperparameters required in a convolutional layer [31]:

- Number of filters **K**.
- Their spatial extent **F**, width and height of filters, can be different.
- The stride **S**, how many pixels the sliding filter moves.
- The amount of zero padding **P**, is used to control the spatial size of the output.

The values of these hyperparameters are used to decide the properties of a convolutional layer. Equation 2.9 is used to calculate the size of the output feature maps. W is the width, but the calculation is the same for the height H.

$$W_{out} = \frac{W_{in} - F + 2P}{S} + 1 \tag{2.9}$$

Pooling layer The convolutional layers are often paired with *pooling* layers, also called sub-sampling layers. These layers take the output feature maps from the convolutional layers, and its function is to reduce the size of the input feature maps to reduce the amount of parameters in the network, hence reducing the amount of computation needed and controls *overfitting*. Overfitting is a problem that may occur when a network fits the noise in the data. The pooling operation is applied separately on each feature map, meaning it is the same number of feature maps on both the input and output of the pooling layer. The pooling layer works in a similar way to the convolutional layer since it operates on a subregion on the input feature maps and performs a filter operation. However the filter operation is most commonly a MAX operation, working on 2x2 subregion and a stride of 2. This causes the width and height of the output to be halved.

The pooling layer requires two hyperparameters:

- Their spatial extent **F**.
- The stride **S**.

Similarly to the convolution layer the size of the output feature maps can be calculated with Equation 2.10:

$$W_{out} = \frac{W_{in} - F}{S} + 1 \tag{2.10}$$

There are two commonly seen variations of the hyperparameters in the pooling layer: The most common one is $F = 2$ and $S = 2$ as mentioned before, but another variant called overlapping pooling is $F = 3$ and $S = 2$. And also, though the most common form of pooling operation is max pooling, the pooling layer can also perform *average pooling* or *L2-norm pooling*. Average pooling has been the most popular historically, but max pooling has been shown to work better in practice [31].

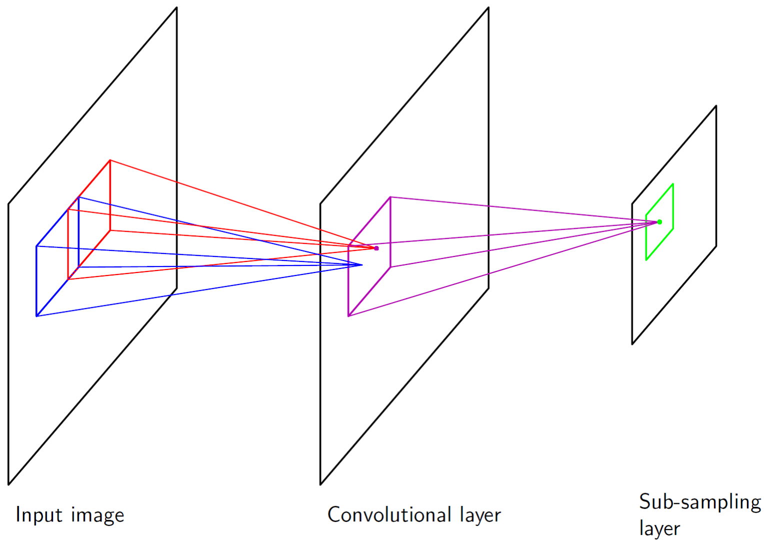


Figure 2.4: Convolution and pooling layer illustration [10].

Figure 2.4 shows how various receptive fields in the input image corresponds to one neuron on the output of the convolutional layer. It also shows how the pooling layer or sub-sampling layer reduces the size of the convolutional layer.

2.2 OpenCL

OpenCL is an industry standard framework for programming heterogeneous computer systems. These systems may contain a combination of CPUs, GPUs, and other devices with processing capabilities, like an FPGA used in this project [24].

OpenCL is a relatively new technology, it was first released in December of 2008 with some early products available in the fall of 2009. It was developed to standardize the method of parallel computation using GPGPUs. The current release of OpenCL is version 2.2 [19].

An OpenCL application consists of a single *host* and up to several *kernels*. The host interacts with the external environment to the OpenCL program, and it performs the necessary setup of the application, including: Discovering the devices of the system, probe the characteristics of the devices, create the kernels, set up and manipulate necessary memory objects, execute the kernels and collect the results. The kernels execute on the OpenCL devices and are functions written in the OpenCL C programming language. Figure 2.5 shows the OpenCL platform model, it shows the host connected to one or more OpenCL devices. The devices are further split into one or more *Compute Units* (CUs), and then *Processing Elements* (PEs) within the CUs.

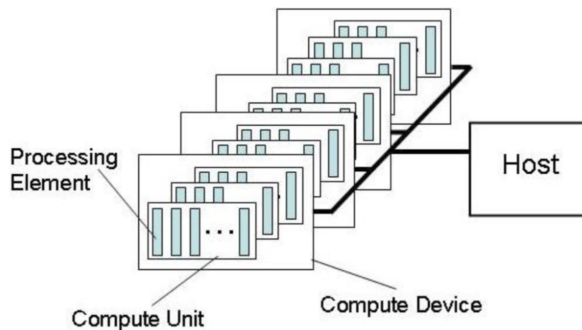


Figure 2.5: OpenCL platform model [20].

When a kernel is launched from the host program, it launches one or more instances of itself, each instance is called a *work-item*. In order to launch the kernel one of the two OpenCL Application Programming Interface (API) calls below are used:

- clEnqueueNDRangeKernel
- clEnqueueTask

The first execution call, *clEnqueueNDRangeKernel* creates a collection of work-items, where the number of work-items is defined by what is called the *NDRange* index space. The NDRange index space is an N-dimensional range of values, hence the name. The NDRange is also divided into work-groups which holds a subset of the work-items, each work-item in a work-group execute concurrently on the processing elements of a single compute unit. All work-items has a local ID within a work-group and a global ID within the NDRange, these ID's may be queried from the kernel with the function calls: *get_global_id(uint dimindx)* and *get_local_-id(uint dimindx)*, where *dimindx* is the specific dimension. Other kinds of info can also be acquired, e.g. the number of work-groups or the number of dimensions [20]. The second execution call *clEnqueueTask* launches a kernel using a single

work-item. Figure 2.6 shows an example NDRange index space of 2 dimensions with 3x3 work-groups and 4x4 work-items in each work-group, that means there is a total of 144 work-items.

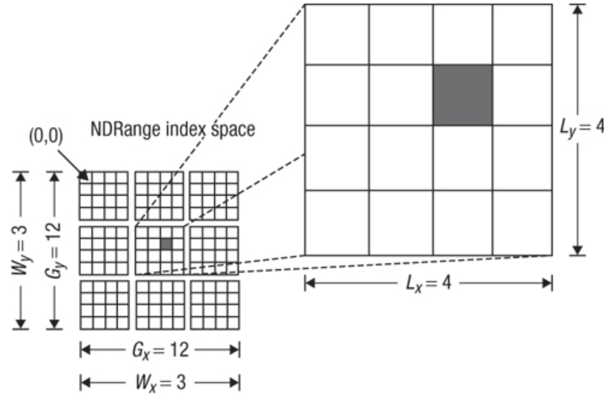


Figure 2.6: NDRange index space example [24].

It is also worth noting OpenCL's memory model, it defines five distinct memory regions:

- **Host memory**: is only visible to the host.
- **Global memory**: permits read/write access to all work-items. May be cached depending on capabilities of the device.
- **Constant memory**: is a region of the global memory that remains constant during execution of a kernel.
- **Local memory**: is local to a work-group and therefore shared by all work-items in that work-group.
- **Private memory**: is private to a work-item and are not visible to any of the other work-items.

2.2.1 Intel FPGA SDK for OpenCL

In this project a HLS tool called Intel FPGA SDK for OpenCL will be used. It is a framework that gives OpenCL support for use with Intel FPGAs. The tool was first released as Altera OpenCL SDK in 2013 [36]. To compile OpenCL kernels for the FPGA, the Altera Offline Compiler (AOC) is used. AOC supports version 1.0 of the OpenCL specification, and some newer features from version 2.0 has preliminary support, such as Shared Virtual Memory (SVM) and *pipes*. Intel has also added an extension called *channels*, which is similar to pipes.

A Compute Unit (CU) in Intel FPGA for OpenCL terms is a pipelined circuit designed to execute a potentially large number of work-items concurrently [12]

[12]. Figure 2.7 displays the compilation flow for the AOC, the inputs are a set of OpenCL kernels, and the output is a *.aocx* file containing the FPGA image. The full compilation may take up to several hours, hence the functionality of the OpenCL code can be tested by calling the AOC with the *-march=emulator* option [4].

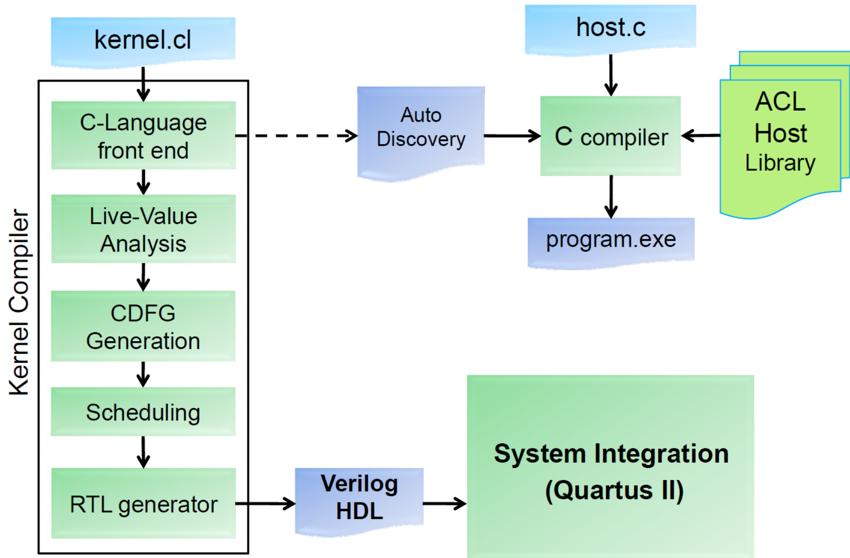


Figure 2.7: OpenCL-to-FPGA framework [12].

For OpenCL devices such as CPUs and GPUs, the kernels can be built during runtime when the *clBuildProgram* function is called. Since the compilation time for FPGA is so long, the kernels must be compiled in advance. However, the *clBuildProgram* function must still be called from the host in order to program the Field-Programmable Gate Array with the .aocx file.

2.3 Speech signal preprocessing

Instead of using the speech signals directly in a speech recognition application like the "wake up word" CNN that will be implemented here, it is common to extract a set of features from each speech signal to use as inputs in the neural network. These features should eliminate unnecessary information with regards to phonetic analysis and enhance the aspects of a signal that helps to detect phonetic differences.

One of the most commonly used methods to extract such features for a CNN is to convert the speech signals to Mel-frequency cepstrum coeffisients (MFCC). This and similar techniques use low level spectral information which conveys vocal tract characteristics. The spectral information is extracted from short time intervals

usually of 20-30ms of the speech signal and using the Discrete Fourier Transform (DFT) on it. Since the vocal tract is a slowly varying system, the speech signal is considered to be nearly stationary over this short time interval. The DFT is defined as follows:

$$X(k) = \sum_{n=1}^{N_s} x(n) e^{-j2\pi kn/M_s} \quad 1 \leq k \leq M_s \tag{2.11}$$

Where N_s is the number of samples in the signal window and M_s is the length of the DFT. Then, the energy spectrum is given as:

$$|X(k)|^2 = \left| \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N} \right|^2 \tag{2.12}$$

The next step to compute the MFCCs is to pass the energy spectrum through a bank of Q *Mel*-spaced triangular filters shown in Figure 2.8, this results in a set of filter bank energies $e(i)|_{i=1}^{Q}$. The *Mel*-scale is linear below 1000Hz and logarithmic above 1000Hz. The number of filters Q in Figure 2.8 varies. Finally, the Discrete Cosine Transform (DCT) is applied to the log of the filter bank energies $log[e(i)]|_{i=1}^{Q}$, and the final MFCCs C_m can be written as :

$$C_m = \sqrt{\frac{2}{Q}} \sum_{l=0}^{Q-1} log[e(l+1)] * cos \left[m * \left(\frac{2l-1}{2} \right) \frac{\pi}{Q} \right] \tag{2.13}$$

Where, $0 \leq m \leq R-1$, and R is the desired number of cepstral features [11].

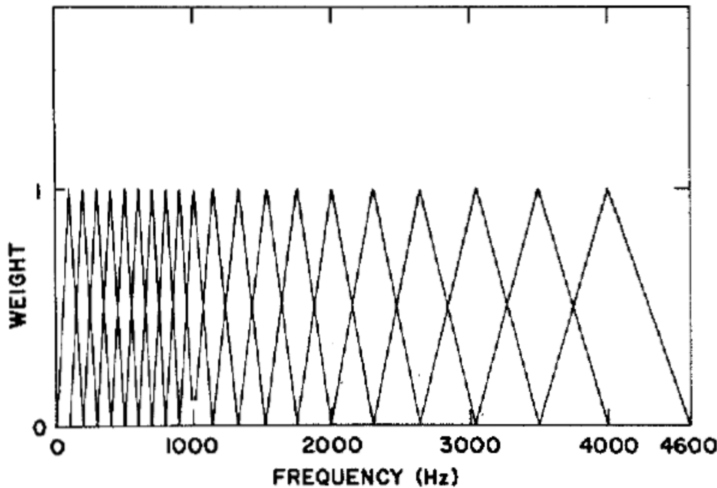


Figure 2.8: Mel-spaced filter bank for generating MFCCs [13].

Chapter 3

Related Work

This chapter will present some recent work that also uses FPGAs to accelerate inference of neural networks. The first two papers also uses the Intel FPGA SDK to implement CNNs, while the third paper takes another approach by using Vivado HLS to implement Binarized Neural Networks (BNNs).

3.1 An OpenCL™Deep Learning Accelerator on Arria 10

In [8], Intel presents their Deep Learning Accelerator (DLA) on Arria 10. They use the Intel FPGA SDK for OpenCL to create the DLA, which is able to implement all of the layers of AlexNet on the FPGA. Previous approaches to use FPGAs for CNNs have often been memory bound due to the limitied external memory bandwidth on the FPGA device, but the DLA architecture is compute bound due to significant reductions in memory bandwidth. They point out that in most CNN topologies, the total amount of computation is dominated by the convolution layers, in AlexNet the convolutions equate to 92% of the total floating point operations. Hence their DLA architecture focuses on optimizing the throughput of the convolution layers. Figure 3.1 shows the overall DLA architecture. The DLA uses several control signals, which are generated by the *Sequencer* unit. In order to implement different CNNs, it is enough to change the configuration of the Sequencer. All of the units in the architecture are OpenCL kernels that executes independently and concurrently. To connect the kernels, the channel extension from the SDK is used.

The results they achieved with Intel's Arria 10 were 1020img/s, or 23img/s/W when running the AlexNet CNN benchmark. This is the same as 1,382 GFLOPS and is 10x faster with 8.4x more GFLOPS and 5.8x better efficiency than the

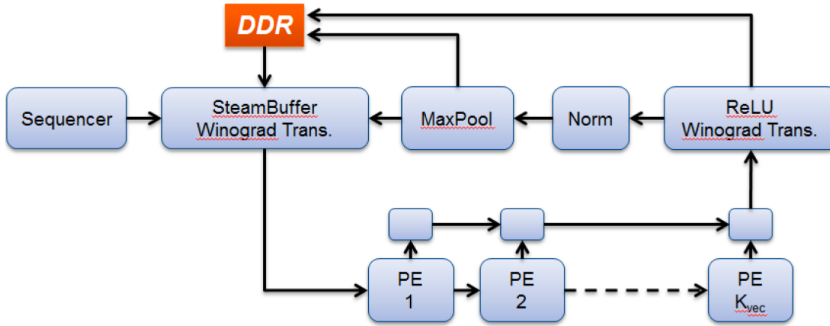


Figure 3.1: Overall DLA architecture.

state-of-the-art on FPGAs. And the 23img/s/W competes with the *best publicly known* implementation of AlexNet on NVIDIA's Titan X GPU. This paper was from January 2017.

3.2 PipeCNN

PipeCNN [35] is an open-source implementation of a CNN accelerator also using the Intel FPGA SDK for OpenCL. The main contributions from this work are: an efficient structure of pipelined kernels for large-scale CNNs. The proposed architecture was explored on Stratix-V A7 FPGA, and two large-scale CNN models, AlexNet and VGG, were implemented and tested.

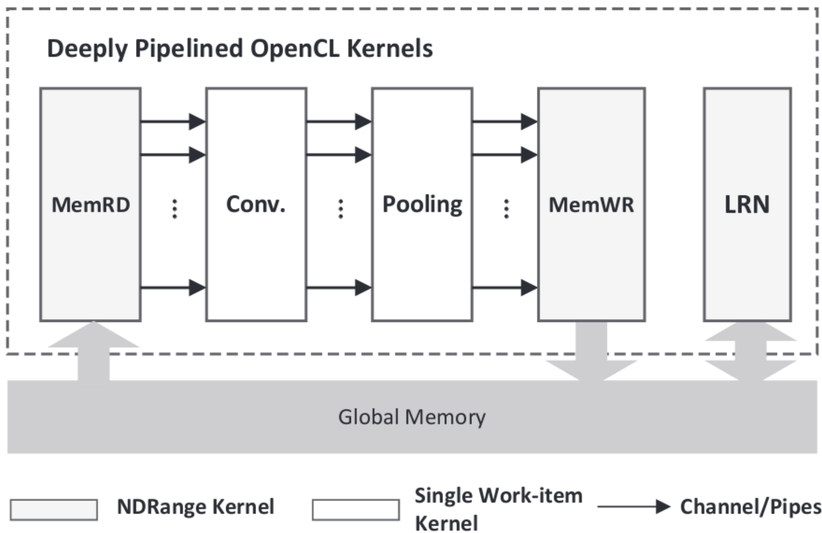


Figure 3.2: PipeCNN architecture.

The architecture of PipeCNN shown in Figure 3.2, includes two data-mover kernels, *MemRD* and *MemWR* which handles the transferring of data and weights between global memory and the FPGA. To transfer data from the MemRD kernel to the Conv. kernel, the Pooling kernel and to the MemWR kernel, the channel extension is used, and it allows the transfers between the kernels to stay on chip. The Local Response Normalization (LRN) kernel is kept separate from the others since it may require multiple memory access patterns.

The paper reports their shortest classification time achieved to be 43ms for AlexNet and 718ms for VGG-16. Also they get 33.9 GOPS while using 162 Digital Signal Processing (DSP) blocks.

3.3 FINN

Another related work is FINN: A Framework for Fast, Scalable Binarized Neural Network Inference [34]. While the previously mentioned work has focused on standard *float32* precision, FINN implements BNNs. To do this they used Vivado HLS, which is a HLS tool from Xilinx [37]. The Vivado HLS tool is more similar to the Intel HLS Compiler, which was an option to use here.

Figure 3.3 illustrates how their framework uses a trained BNN and its topology, to create a C++ description of a heterogeneous streaming architecture.

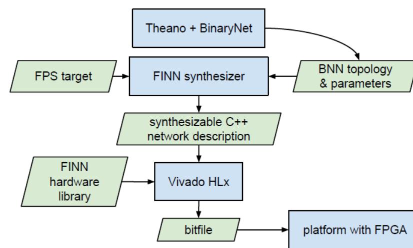


Figure 3.3: Generating an FPGA accelerator from a trained BNN [34].

With a ZC706 embedded FPGA platform drawing less than 25 W total system power, they achieved up to 12.3 million image classifications per second with 0.31 μs latency on the MNIST dataset with 95.8% accuracy. For the CIFAR-10 and SVHN datasets, they achieved accuracies of respectively 80.1% and 94.9%, with 21906 image classifications per second and 283 μs latency.

Chapter 4

Neural Networks Implementation

This chapter describes the implementation of three different neural networks with increasing complexity. The sections are divided into these three networks, explaining the differences in their implementations.

4.1 Linear classifier for the MNIST dataset

A natural network to start with when learning about machine learning and neural networks is a linear classifier. In addition the Modified National Institute of Standards and Technology database (MNIST) database [21] is very suitable for such a classifier. The MNIST dataset consists of 28x28 images of handwritten digits, it contains $60,000$ training examples and $10,000$ test examples. The digits range from 0-9, which means that in order to classify the results there has to be ten output values. The operation that the classifier will do is shown in Equation 4.1. The input data $\mathbf{x}$ is flattened into a single $[784 \times 1]$ column vector, thus the weight matrix $\mathbf{W}$, needs to be of the dimensions $[10 \times 784]$. The bias $\mathbf{b}$ and output $\mathbf{y}$ are both column vectors of size $[10 \times 1]$.

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{4.1}$$

A simplification can be done to make Equation 4.1 easier to implement, by increasing the number of columns in $\mathbf{W}$ by one and extending $\mathbf{x}$ with a constant 1, the problem is reduced to a single matrix multiplication:

$$\mathbf{y} = \mathbf{W}\mathbf{x} \tag{4.2}$$

4.1.1 Training

In order to train this linear classifier, a modified version of TensorFlows [32] tutorial file for the MNIST dataset were used. The only changes done were combining **W** and **b** as explained earlier. The training performs gradient descent on batches of 100 random training examples for each training step with a learning rate of 0.5, and this is done 1000 times. Ideally each step should include all the training data, but using 100 random training examples for each step is cheap and sufficient. This is the same as *stochastic gradient descent* as explained in Section 2.1.2. The trained weights were extracted to a header file to make it easy to use for the OpenCL implementation of the linear classifier.

4.1.2 OpenCL Implementation

As explained in Section 2.2, OpenCL programs is divided into a host file and different kernels which can run on various devices, in this case an FPGA. As a starting point a matrix multiplication example design provided on Intel FPGA's webpages [18] was used.

The linear classifier is implemented with one kernel which performs the matrix multiplication in Equation 4.2.

Host file The host program is the part of the application that handles initialization and execution of the kernels. Here, the host program is divided into five functions: *read_mnist*, *init_opencl()*, *run()*, *verify()* and *cleanup()*.

The first function *read_mnist()* prepares the image data by reading from binary files containing the MNIST data. The image data was downloaded from [1], which is derived from the original database, but is separated into ten binary files, one for each class 0 to 9. The image values in these files range from 0-255, while the network expects values between 0-1, hence they also had to be normalized. The function stores each image in *input_x* which is initialized as: *float* input_x[NUM_IMAGES]*. *NUM_IMAGES* is the total number of images to read from the binary files, and 64 byte aligned memory allocation is used. Without 64 byte aligned memory allocation, a warning is returned which states that Direct Memory Access (DMA) is not used due to the lack of alignment. A consequence of not using DMA to transfer the data is that unnecessary CPU resources may be used.

The necessary initializations is performed in *init_opencl()*. First it calls *findPlatform("Altera")* to check if the FPGA device is available. The function is not a standard OpenCL API call but is provided in the common utility functions mentioned earlier. Since the version of Intel FPGA SDK for OpenCL used here is

version 16.0.2 Build 222, the platform is *Altera*. For newer versions of the SDK, the platform name is *Intel(R) FPGA OpenCL*. Next the context is created using the API call *clCreateContext()*, it consists of the number of Altera devices found in the system, which in this case is an Arria 10 FPGA. Another function from the common utilities, *createProgramFromBinary()* is used to create the program object, it reads a *.aocx* file that is generated when compiling the kernel file, and then the program is built using *clBuildProgram()*. Next the command queue for the device and a kernel object is created. And finally buffer objects are created for all the inputs and outputs: *input_x_buf[NUM_IMAGES* and *output_buf[NUM_IMAGES]*. All of the inputs are also written to the input buffers at this stage, *clEnqueueWriteBuffer()* does this.

When the initialization is done the *run()* function is called. It starts by setting the arguments of the kernel using *clSetKernelArg()* inside the loop launching the kernels. Listing 4.1. There are two arguments that need to be set, and they correspond to the arguments shown for the kernel in Listing 4.2. The arguments are changed for each kernel execution since a new input is needed, *input_x_buf[i]* and *output_buf[i]*, represents the host side of the arguments. When the arguments are set, the kernel is launched with an NDRange of 10 global work-items.

```
1  const size_t global_work_size[1] = {10};
2  const size_t local_work_size[1]  = {10};
3
4  for(unsigned i = 0; i < NUM_IMAGES; i++) {
5    // Set correct output buffer for argument
6    status = clSetKernelArg(kernel[0], 0, sizeof(cl_mem), &
         output_buf[i]);
7    checkError(status, "Failed to set argument %d", 0);
8
9    // Set correct input buffer
10   status = clSetKernelArg(kernel[0], 1, sizeof(cl_mem), &
         input_x_buf[i]);
11   checkError(status, "Failed to set argument %d", 1);
12
13   // Execute the kernel
14   status = clEnqueueNDRangeKernel(queue[0], kernel[0], 1,
         NULL,
15        global_work_size, local_work_size, 0, NULL, &
             kernel_event[0]);
16   checkError(status, "Failed to launch kernel");
17 }
18
19 // Wait for all kernels to finish.
20 clWaitForEvents(num_devices, kernel_event);
```

Listing 4.1: Execution loop in run() function

The output arrays contains 10 elements which corresponds to each digit, the *verify()* function reads all the outputs from the output buffers using *clEnqueueReadBuffer()*, and verifies that the highest value in each output array is correct, and then calculates the accuracy. The final part is to clean up, for this linear classifier this includes releasing the kernel, the queue, the buffers, the program and the context. And also freeing up *input_x* which holds the image data.

A challenge faced here was to figure out how to run the kernel several times in order to process all the images. The optimal solution would be to continuously stream the data into the kernel, however a way to do this was not found and does not seem to be possible at this point. Hence a more straightforward solution was to write new input to the buffer and execute the kernel each loop iteration. Another possible solution could be to change the kernel to handle an input of several images in one execution, this would probably increase the throughput since the overhead of writing to the buffer and executing the kernel would be smaller. It was also not clear at once when the kernels are executed, it does not happen when the kernels are queued up, instead it happens when a *flushing* of the command queue is performed. In this case the flush is performed implicitly when calling *clWaitForEvents()*

Kernel For this linear classifier only a single kernel is necessary, its task is to calculate the matrix multiplication shown in 4.2. Due to the bias $\mathbf{b}$ being merged with the weights $\mathbf{W}$, the dimensions of the matrices are: $[10 \times 785]$ for $\mathbf{W}$ and $[785 \times 1]$ for $\mathbf{x}$. The focus when implementing this kernel was to get a simple kernel, not to create a highly optimized kernel. The main goal is to end up with a functioning CNN.

A standard way to implement a matrix multiplication in software would be to use three for-loops to iterate the matrices. However since the input $\mathbf{x}$ is a column vector, a good simplification was to reduce the kernel to perform only a dot product and execute it with an NDRange of one dimension and 10 work-items. 10 work-items is used because of the number of elements in the output $\mathbf{y}$.

The linear classifier kernel code is presented in Listing 4.2. The kernel takes two arguments, namely Y and X, which corresponds to $\mathbf{y}$ and $\mathbf{x}$. Both arguments use the *__global* identifier, which means each access will be to the global memory region, this is off-chip memory. The *restrict* keyword informs the compiler that there is no overlap between the two pointers Y and X [4]. Instead of passing the weight matrix W as an argument, it is defined in the *weights_linear_classifier.h* header file as global memory and then included in the kernel file. The kernels are launched with only one work-group consisting of 10 work-items, this means that a call to *get_global_id(0)* is identical to a *get_local_id(0)* call.

```
1  #include "../host/inc/linear_classifier.h"
2  #include "../host/inc/weights_linear_classifier.h"
3
4  #ifndef SIMD_WORK_ITEMS
5  #define SIMD_WORK_ITEMS 2
```

```
6  #endif
7  #ifndef NUM_COMPUTE_UNITS
8  #define NUM_COMPUTE_UNITS 2
9  #endif
10
11 __kernel
12 __attribute((reqd_work_group_size(10,1,1)))
13 //__attribute((max_work_group_size(10)))
14 __attribute((num_simd_work_items(SIMD_WORK_ITEMS)))
15 //__attribute((num_compute_units(NUM_COMPUTE_UNITS)))
16 void linear_classifier(
17                      // Input and output matrices
18                      __global float *restrict Y,
19                      __global float *restrict X)
20 {
21     // Global ID index
22     int global_id = get_global_id(0);
23
24     float running_sum = 0.0f;
25
26     for (int k = 0; k < W_x; ++k) {
27         running_sum += W[global_id][k] * X[k];
28     }
29     barrier(CLK_LOCAL_MEM_FENCE);
30
31     // Store result
32     Y[global_id] = running_sum;
33 }
```

Listing 4.2: Linear classifier kernel code

4.2 One hidden layer neural network

The second network to be implemented was a neural network with one hidden layer, and each neuron in the hidden layer and the output layer uses *sigmoid* as its activation function. The *sigmoid* function was presented earlier and shown in Figure 2.2. Same as the linear classifier this network also has ten outputs since it is the same classification problem, however the hidden layer has 25 neurons. The input vector is also slightly different from the linear classifier due to the MNIST images being scaled down to 20x20 here, this is because the network and training was based on a Coursera course [25]. Similar to the linear_classifier, stochastic gradient descent was used with 200 iterations.

Figure 4.1 below shows an example of a neural network that is similar to the one being implemented here. The differences are that the input layer consists of 20x20

= 400 neurons, and the hidden layer 25 neurons. The linear classifier also has the neurons connected the same way (fully connected), but the input layer is directly connected to the output layer, and no activation function is used.

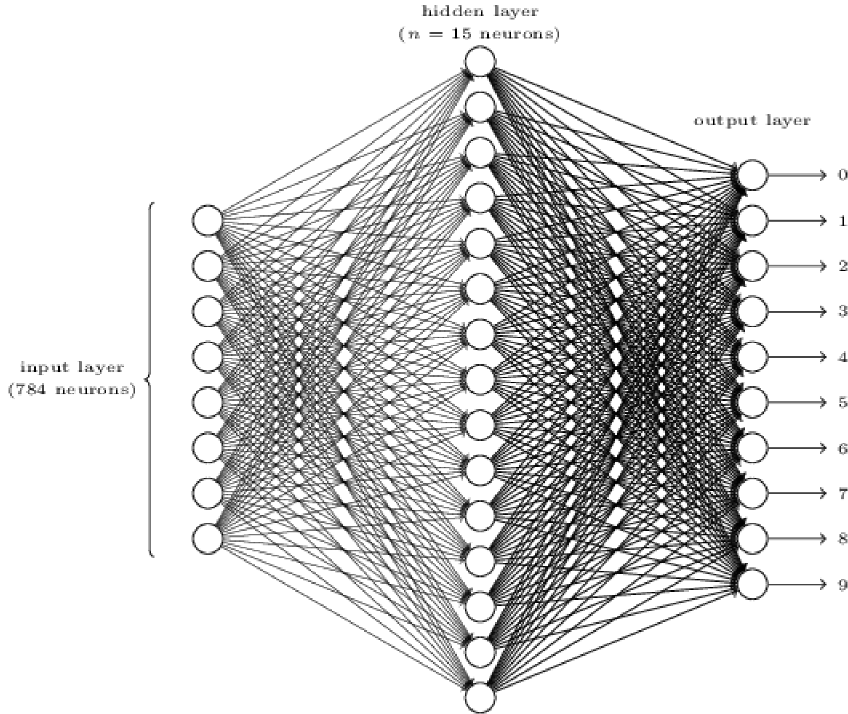


Figure 4.1: Example neural network [26].

Host The host program here is very similar to the linear classifier's version, it is divided the same way with the same types of functions. However there are some differences since the image data here is scaled coming from the Coursera course, also there are two fully connected layers to be executed for each image.

The difference in the image data is that in addition to the size being 20x20, it does not need normalizing, but otherwise *read_mnist()* does the same as before.

The context for this neural network with one hidden layer comprises the FPGA device, a program object, two kernels and three buffers. The linear classifier had one kernel and two buffers in comparison. A single kernel could also have been used here, but that would require a small change in the kernel code which will be explained in the next paragraph. The extra buffer here is a temporary buffer used to store intermediate results from the hidden layer, and is then used as an input for the output layer.

Between the two layers or kernel executions it was first considered necessary to

use *clEnqueueReadBuffer()* and *clEnqueueWriteBuffer()* to store the intermediate results, fortunately this is not necessary as the results stays in the buffer. Instead both the kernels are simply added to the queue. For the linear classifier the number of work-items was 10, for this neural network two kernels are launched with a different amount of work-items. The first kernel, representing the hidden layer of the network need 25 work-items, this corresponds to the 25 neurons in this layer. Similarly the second kernel requires 10 work-items for the output neurons.

Kernels The two kernels in this neural network is essentially the same as the linear classifier, but since this is a neural network an activation function has to be applied to the output. A sigmoid activation is used here and Listing 4.3 shows how it is defined in the code.

```
1 #define SIGMOID(x) (1.0f / (1 + exp(-x)))
```

Listing 4.3: *sigmoid* activation function in the kernel file

The kernels are split into one kernel for each layer, this has to be done when using the attributes in Listing 4.4. These attributes specifies the required amount of work-items in a work-group for the kernel, and they allow the compiler to optimize the generated hardware. For this neural network the required number of work-items are 25 and 10 and the NDRange is of only one dimension. If the kernels is executed with a different NDRange from the host, it would result in an error. There is also a *max_work_group_size(N)* attribute which works in a similar way, but only specifies the **maximum** number of work-items in a work-group. Note that this attribute only requires one parameter while *reqd_work_group_size(X,Y,Z)* requires the size in each dimension of the work-group. This attribute could be used here to avoid defining two separate kernels by setting *max_work_group_size(25)* instead.

```
1 __attribute((reqd_work_group_size(25,1,1)))
2 __attribute((reqd_work_group_size(10,1,1)))
```

Listing 4.4: Attributes used with neural network kernels

In addition to *reqd_work_group_size(X,Y,Z)* and *max_work_group_size(N)*, two attributes shown in Listing 4.2 can be used to control vectorization and CU replication of the kernels. The *num_simd_work_items(SIMD_WORK_ITEMS)* can be used to vectorize the kernel, to use this it is also necessary to specify the required work-group size which must be divisible with the number *SIMD_WORK_ITEMS*, the number must also be a power of two. The attribute *num_compute_units(NUM_COMPUTE_UNITS)* is used to replicate the compute units. This attribute must be used with the maximum work group size attribute. Some variations of these attributes will be tested to see the impact on resource usage and performance in Chapter 6

4.2.1 Two hidden layer neural network

A neural network with two hidden layers was also implemented to test how the resource usage varies with increasing neural net sizes. This also used the MNIST dataset from Coursera and was trained the same way. This network also uses sigmoid as its activation function, and it uses the same *reqd_work_group_size(X,Y,Z)* attribute for the kernels, which means there are three kernels here: two for the hidden layers, and one for the output layer. The first hidden layer has 100 neurons, corresponding to 100 work-items, the next hidden layer has 25 neurons and finally the output layer has 10 neurons as it is still the same classification problem. A difference in the kernel here is that the weight matrix is included in the arguments to the kernel functions. This means that the host has to create buffer objects for the weights and add the extra argument. However, since the weights are constant, the arguments can be set outside the execution loop.

Since these neural networks are so similar, only the host code for this neural network with two hidden layers is added to the Appendix A.1, all the kernels and headers are added.

Chapter 5

CNN Implementation

While the last chapter focused on implementing neural networks with increasing complexity, this chapter will present the implementation of a "Wake up word" CNN. The first part of the chapter describes the architecture of the CNN, and the latter part discusses the implementation in OpenCL. The live setup code is also explained. Some source code is listed here, but the rest of the code is in Appendix A.1.4

5.1 Architecture

The CNN implements a trained network able to recognize a so-called "wake up word", where the phrase to be recognized is "hey spark". Cisco provided the architecture and performed the training for this network, and the topology of the CNN is shown in Figure 5.1. As shown in the figure the network consists of three convolutional layers and three fully connected layers. The fully connected layers are called ip1, ip2 and ip3 in the figure, where ip is the inner product and is the same as a fully connected layer. In the text the layers will be called FC1, FC2 and FC3 instead. There are no pooling layers in this network, instead the hyperparameter values of the convolutional layers decides the spatial reduction.

The input to this network is no longer images, fortunately it is still of two dimensions similar to an image, in the form of Mel-frequency cepstrum coeffisients (MFCC), which was explained in Section 2.3. One such input has the size $[90 \times 40]$, note that while the input data for the networks in the last chapter were flattened into a column vector, here the data is still treated in its original form. As mentioned Cisco trained the network, and to do it they used a machine learning framework called *Caffe* [9], as a standard way of describing the dimensions of a data object they use the format: number N x channel C x height H x width W or (N, C, H, W).

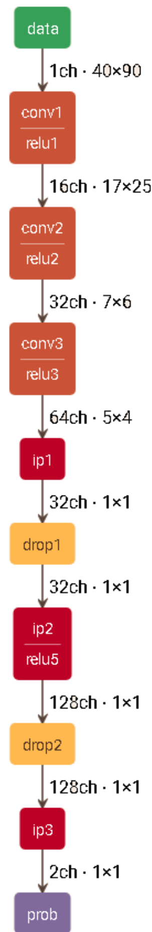


Figure 5.1: Topology of implemented wake up word CNN.

N may for instance be the number of filters. C is the depth or number of channels, for instance in an RGB image C would be 3 due to the three color channels, while H and W would be the height and width of the image. This format will be used from now on to describe the various weights and data objects in the network. With this format the input size is (1, 1, 90, 40).

The CNNs layers will now be described with more detail in this list:

- **Layer 1, convolution, C1**: Takes an input of size (1, 1, 90, 40), and creates 16 output feature maps with a total size of (1, 16, 25, 17). The 16 filters are of size (16, 1, 16, 8) and the stride height direction *stride_h* is 3, and along the width *stride_w* is 2.
- **Layer 2, convolution, C2**: The input here is the output feature maps from

the last layer of size (1, 16, 25, 17). There is 32 filters in this layer of size (32, 16, 8, 4) which results in an output of size (1, 32, 6, 7). The strides are the same as the last layer.

- **Layer 3, convolution, C3**: The third layer also takes the output from the previous layer as input. This layer uses 64 filters of size (64, 32, 3, 3). Here the strides are 1 in each direction, hence the output size is (1, 64, 4, 5).
- **Layer 4, fully connected, FC1**: This layer is the first fully connected layer and it takes a flattened version of the output from the previous layer, which results in a $65 \times 4 \times 5 = 1280$ long vector. The layer has 32 neurons, hence the weight matrix is of size (1, 1, 32, 1280) and bias (1, 1, 32, 1).
- **Layer 5, fully connected, FC2**: This layer has 128 neurons, which then has a weight matrix of size (1, 1, 128, 32) and bias (1, 1, 128, 1).
- **Layer 6, fully connected, FC3**: The last layer has 2 output neurons which tells if the correct phrase was spoken. The weight matrix is of size (1, 1, 2, 128) and the bias (1, 1, 2, 1).

Each of the layers, except FC1 and FC3, uses the RelU activation function on every neuron in the layer. The activation functions are often presented as a separate layer in architecture overviews, but in the OpenCL implementation they are added directly in the kernel of a convolutional or fully connected layer. FC1 does not use any activation function, while FC3 appends a *softmax* layer on the output, which is represented as the *prob* layer in the figure. The softmax layer uses a *softmax function* to normalize the set of outputs, so they add up to 1. This way their values represent probabilities. In addition to the softmax layer, drop-out layers (*drop* layers in the figure) was used during training of the network, the drop-out layer is used to prevent overfitting. However these layers are not necessary during inference of the network, and are therefore not mentioned in the list above or implemented.

5.2 OpenCl implementation

This section will explain the implementation of the wake up word CNN and the infrastructure making it easy to configure the networks architecture.

Host Since host files generally does the same tasks, which mainly includes initializing and execution of kernels, the overall structure of this host program is similar to those described earlier.

The first task performed is preparation of the data objects, this is done in the function *prepare()*. It is similar to *read_mnist()*, but in addition to reading input data, it also reads weights and biases, and a golden reference for verification. All this information is stored in binary files of the *numpy* format since the training was

done with Caffe in python. In order to read these files a library called *cnpy* [29] was used. The included method *npy_load(filename)* returns a struct containing shape information and the data of the numpy array. The data part of these structs are copied into 64 byte aligned memory objects using *memcpy()*. In addition to the data, information about the sizes of the various arrays are also stored for use later in the program.

The next part of the host is initialization, this is still done in a function called *init_opencl()*. The memory objects created here are: input and output buffers as usual, there are also a weight and bias buffer for each layer, and lastly between the layers two temporary buffers are used. The temporary buffers will hold varying amount of data, hence a *TMP_BUF_SIZE* constant is used, and it needs to be set to the maximum buffer size needed. For this architecture it is 6800, which is the total size of the output from the first layer. Two kernel objects are created, one for each kernel type, that is the convolution kernel and the fully connected kernel. There are also separate command queues for each kernel.

The idea of the application is to run constantly and listen for the phrase "hey spark" to get live wake up word detection, hence the main part of the *run()* function is placed inside an infinite while loop. The current version of this live setup is not optimal however since it requires one to speak at certain times in order to record the voice. A new voice input is read at the beginning of each iteration of the while loop, more details on how this is done is described in Section 5.2.1. Further each layer is executed in a loop with *num_layers* iterations, where the first part of the loop body involves setting the arguments of the current layer. If it is the first layer the input is written to *input_buf*, then for the convolution layers the taskconv kernel is executed with *cLEnqueueTask()* since it is a single work-item kernel. The ndrconv kernel is executed as an NDRange kernel as earlier. And for the fully connected layers, the kernels are executed in the same way as the neural networks with the appropriate number of work-items. Weights and biases are now arguments to the kernels, while the weights were included as headers in the neural networks. They must therefore also be written to the kernels. After the execution of each layer in the CNN, the output buffer is read, and if the second output element is higher than the first element, then the network considers the recorded phrase to be the wake up word. There is also a verification function, but it is only necessary when using test input to test against the golden reference.

To ease the readability and generalization of the code, an enum called *config_-item* and an array called *layer_config* are introduced, inspired by PipeCNN [35] mentioned in Section 3.2. The array contains information about each layer, e.g. the type of the current layer or the dimensions of the data. Listing 5.2 shows a part of the layer_config array with the values for the first convolution layer. These values are used in various parts of the host file and as arguments for the kernels. The config_item enum shown in Listing 5.1 gives names to the indices of the layer_-config array, which makes the code easier to read. Also this implementation makes it easy to try various CNN architectures or to test separate layers in order to verify correct behaviour.

```
enum config_item{
    layer_type,
    type_num,
    data_h, data_w, //data_n,
    weight_n, weight_c, weight_h, weight_w,
    conv_h, conv_w, conv_stride_h, conv_stride_w,
    relu_on,
    memrd_src,
    memwr_dst
};
```

Listing 5.1: Layer config enum

```
unsigned int layer_config[][15] = {
        { // C1
        0,
        1,
        90, 40,
        16, 1, 16, 8,
        25, 17, 3, 2,
        1,
        0,
        2
        }
};
```

Listing 5.2: Layer config example showing configuration for first layer

Kernels The CNN is implemented using two kernels, one kernel is the fully connected kernel, which is almost the same as the kernel used in the neural networks, but there are some differences. First of all the activation function used here is RelU instead of sigmoid, Listing 5.3 shows its definition in code. Since not all of the layers require RelU, an argument *relu_on* is used to control this, in addition to this argument *W_width* is passed to set the loop limit. Input, output, weights and bias are also arguments in the fully connected kernel as well as the convolution kernel. The convolution kernel also has several arguments that controls the layers stride, filter size, input and output sizes, etc.

```
#define RELU(x) (x > 0 ? x : 0)
```

Listing 5.3: *RelU* activation function in the kernel file

Listing 5.4 below shows the convolution kernels code. The kernel performs the convolution with the use of six loops, where the multiply accumulate operation of the dot product is performed in the inner loop. The bias is added and RelU applied in the third loop since the three first loops iterates the output neurons. Since the data is transferred as 1D arrays the indexing gets somewhat complicated.

This is not an optimal solution since it does not utilize any parallelism and the data is not buffered, but it was necessary to implement a basic convolution kernel achieve a functional CNN. This kernel does not support padding since it would be difficult to test without having a CNN that requires padding.

```
1  __kernel
2  __attribute__((task))
3  void taskconv(
4        // Params Ports
5        unsigned in_h,
6        unsigned in_w,
7        unsigned out_c,
8        unsigned in_c,
9        unsigned K_h,
10       unsigned K_w,
11       unsigned out_h,
12       unsigned out_w,
13       unsigned S_h,
14       unsigned S_w,
15       unsigned relu_on,
16
17       // Data Ports
18       __global float *restrict output,
19       __global float *restrict input,
20       __global float *restrict weights,
21       __global float *restrict bias
22       )
23 {
24   unsigned filter_size_2d = K_h * K_w;
25   unsigned filter_size_3d = filter_size_2d*N;
26
27   unsigned ifm_size = in_h*in_w;
28   unsigned ofm_size = out_h*out_w;
29
30   float running_sum = 0.0f;
31
32   for (unsigned row = 0; row < out_h; row++) { // output
       rows
33     for (unsigned col = 0; col < out_w; col++) { // output
         cols
34       for (unsigned to = 0; to < out_c; to++) { // output
           feature maps
35         for (unsigned ti = 0; ti < in_c; ti++) { // input
             feature maps
36           for (unsigned i = 0; i < K_h; i++) { // filter
               height
```

```
37              for (unsigned j = 0; j < K_w; j++) { //  filter
                    width
38                running_sum += weights[to*filter_size_3d + ti
                      *filter_size_2d + i*K_w + j] * input[ti*
                      ifm_size + (S_h*row + i)*in_w + (S_w*col +
                      j)];
39              } // j
40            } // i
41          } // ti
42          running_sum += bias[to];
43          if (relu_on)
44            output[to*ofm_size + row*out_w + col] = RELU(
                  running_sum);
45          else
46            output[to*ofm_size + row*out_w + col] =
                  running_sum;
47          running_sum = 0.0f;
48        } // to
49      } // col
50    } // row
51  }
```

Listing 5.4: Convolution kernel

In addition to the *taskconv* kernel, a *ndrconv* kernel was also attempted. Its code is not listed here, but its implementation is similar to fully connected layers with only one loop. While the *taskconv* kernel is a task, the *ndrconv* is a NDRange kernel that allows kernel replication and vectorization.

5.2.1 Getting it live

In order to test the wake up word implementation in a realistic setting, it was desirable to run with live recording. This gives both an easy way to test the network with various inputs, and it shows how responsive the implementation is. To implement this requires a way to record the voice of a speaker and then do some preprocessing on the recorded audio input.

Preprocessing The preprocessing is covered first since the recording part depends on it. Fortunately Cisco provided an executable file that performs the preprocessing, it takes a 48 kHz Waveform Audio File Format (WAV) file as input where each sample is a 32-bit float. The exexutable will generate a binary file containing 40 MFCCs for each 10ms frame, this means that a WAV file that is 900ms long yields the desired $[90 \times 40]$ input for the CNN. In code, the executable is run as shown in Listing 5.5:

```
1 system("./mfcc_preprocess --input voice_rec.wav --output
     voice_rec_mfcc");
```

Listing 5.5: Execute preprocessing file

Recording To record audio a library called *PortAudio* [28] was used. PortAudio is free, cross-platform and open-source, which make it suitable for use on both CentOS 7, which is used here and other operating systems. A function *record_-audio(float* recorded_samples)* was made to be called from the host file when a new input is required. The function utilizes PortAudio's blocking read function presented in Listing 5.6 below:

```
1 PaError Pa_ReadStream(PaStream* stream,
2                       void* recorded_samples,
3                       unsigned long total_frames
4                      )
```

Listing 5.6: PortAudio blocking read function

Prior to this function call it is necessary to call *Pa_OpenStream()* that opens a stream with the desired behaviour, in this case: blocking read, a sample rate of 48 kHz, 1 audio channel and float32 data type. Next *Pa_StartStream()* commences audio processing on the opened stream object. After these two functions are called the *Pa_ReadStream()* function is called with $total_frames = 0.9s \times 48kHz$, the samples will be stored in the *recorded_samples* array and is ready to be written to a WAV file. Finally the stream is closed with *Pa_CloseStream()*.

Write to WAV Since the preprocessing executable requires a WAV file as input, it was necessary to find a way to do this. There probably exists good libraries to achieve this, but doing it without a library was probably just as easy. Figure 5.2 shows the format of a simple WAV file, some extensions with more *sub-chunks* exists, but they were not necessary here. A function *write_wav()* was created and is called from the host application when the recorded audio data is ready to be written. Each field in the header part of the WAV format from Figure 5.2 (everything except the data) is declared in a struct. Then in the *write_wav()* function each field of the struct is given the appropriate value for this application. After the header fields are set, the WAV file is simply created by first writing the header to a binary file and then the recorded audio samples to the same file.

The code for the *write_wav()* function and the *read_audio()* function is included in Appendix A.1.5.

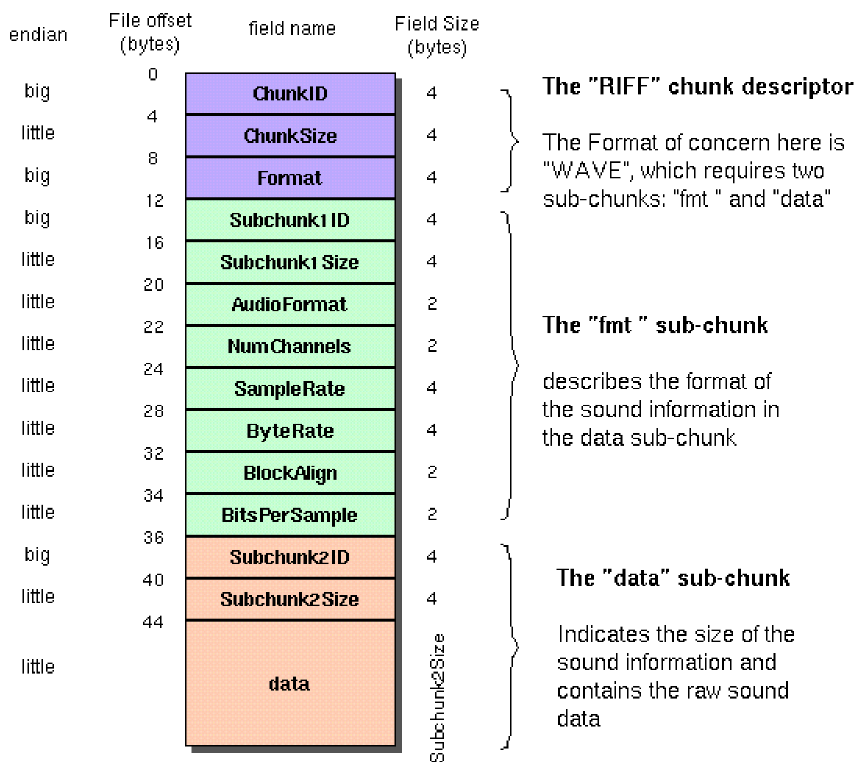


Figure 5.2: WAV format [2].

Chapter 6

Results and Evaluation

This chapter will present and evaluate the results achieved with regards to resource usage and performance. However, first the experimental setup is described and how the results were retrieved is explained. A comparison with a GPU implementation is also included.

6.1 Experimental setup

The FPGA used to test the OpenCL kernels is an Arria 10 GX FPGA Development Kit ES3 from Intel, it is depicted in Figure 6.1. As stated earlier the Intel FPGA SDK for OpenCL is necessary to develop OpenCL applications for the FPGA, along with the SDK, Quartus Prime Pro is also necessary since it is used as a part of the compilation process. Due to the FPGA used, the newest version of the SW possible to use was 16.0.2.222.

The Arria 10 is the latest FPGA in the Arria series and is built on 20 nm technology, which is considered to be the mid-range series from Intel FPGA. Table 6.1 shows the resources available on the FPGA.

The Adaptive Logic Module (ALM) is the basic building block of the FPGAs logic fabric, it uses an 8-input Adaptive Look-Up Table (ALUT) together with four registers and two adders to implement various logic. The 8-input ALUT can be configured to implement different functions, e.g. two independent 4-input functions, or a 5-input and 3-input function, and thus has the flexibility to implement up to 2.5 Logic Elements (LEs), [16] and [3]. In addition to the ALM and its registers and LEs, the other relevant resources are the memory blocks M20K and Memory Logic Array Block (MLAB), and the DSPs.

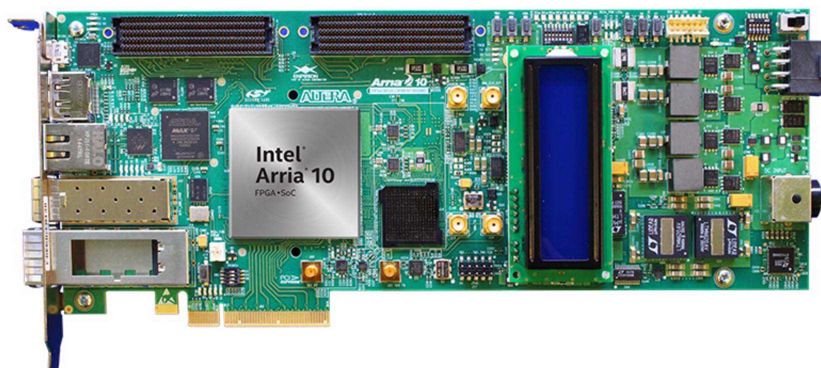


Figure 6.1: Arria 10 GX FPGA Development Kit [15].

The Arria 10 device has a Peripheral Component Interconnect Express (PCIe) 3.0 x8 connector which is used as the interface to the host computer. Since PCIe 3.0 or Gen3 is backwards compatible it supports speeds from 20 Gbps full-duplex for Gen1, and up to 64 Gbps full-duplex for Gen3 [5]. In order to fully utilize the maximum transfer speed, the host computer must also support PCIe 3.0.

One of the main advantages of using the Intel FPGA SDK for OpenCL compared to other HLS solutions such as Intel HLS Compiler, is that the logic necessary to communicate with the host computer is already implemented in the form of a Board Support Package (BSP).

The performance of the OpenCL implementation is compared with a GPU implementation of a CNN. The GPU used for this was a NVIDIA Tegra X1 which is a 20 nm mobile processor with 256 GPU cores based on NVIDIAs Maxwell architecture [27].

6.2 Compiling and retrieving results

To compile a set of OpenCL kernels the command in Listing 6.1 is used, this creates the *.aocx* file that the host loads and uses to program the FPGA. Compiling with the *-g* option gives a detailed area breakdown by source line in the resulting area report. The host program is compiled with g++ since it is an ordinary C++ program

```
1  $ aoc -v -g --report --board a10gx_es3 device/kernels.cl -o
       bin/kernels.aocx
```

Listing 6.1: Compilation command

When a project is compiled it is possible to get a utilization report with estimated values for the kernels, this is done with the command below in Listing 17. It is

Resource		#
Logic Elements (LE) (K)		1,150
ALM		427,200
Register		1,708,800
Memory (Kb)	**M20K**	54,260
	MLAB	12,984
Variable-precision DSP Block		1,518
18 x 19 Multiplier		3,036
PLL	**Fractional Synthesis**	32
	I/O	16
17.4 Gbps Transceiver		96
GPIO		768
LVDS Pair		384
PCIe Hard IP Block		4
Hard Memory Controller		16

Table 6.1: Arria 10 GX resources [16]

this report that shows the detailed area breakdown when compiling with the *-g* option. In addition to this report the standard reports from Quartus is also available after the *.aocx* file has been generated. These reports are called *top.fit.place.rpt* and *top.fit.summary*. Resources taken up by the BSP is found in a file called *base.fit.summary*.

```
1 $ aocl analyze-area bin/cnn.aocx
```

Listing 6.2: Analyze area command

6.3 Resource utilization

This section presents the resource utilization results from the different implemented networks. As mentioned in the last section the resource utilization was found mainly in the top.fit.place.rpt, while the detailed estimated reports was helpful to evaluate the results. It should be noted that the resource results presented here only shows the utilization for the kernels, the total resources used are much higher since that includes the BSPs usage and extra resources required. Table 6.6 shows the total resource usage for the biggest design.

As previously stated the synthesized resource results also includes other necessary logic on the board, such as the BSP. Hence the results in Table 6.2 is much higher than those from the previous table.

6.3.1 Neural nets

Each design was compiled with some variations of the attributes mentioned in Section 4.2.

Linear classifier The linear classifiers resource utilization is presented in Table 6.2. The table shows the resources used for the various attributes, in the SIMD2 column the kernel uses *num_simd_work_items(2)* in order to execute two work-items in parallel and in the same CU. The 2 CU column on the other hand uses *num_compute_units(2)* to generate two separate CUs. Both should increase throughput but at the expense of more resources being utilized.

The resource utilization of the linear classifier shows that the standard implementation and SIMD2 uses a similar amount of resources, however one expected difference is that the amount of DSPs is doubled, due to the amount of parallelization. This is the same for the 2 CUs implementation. A big portion of the BRAM and memory bits used is due to a 2048 kilobit cache being generated. This cache is generated when a global load operation is performed [4], which happens at line 27 in Listing 4.2. The amount of BRAM is doubled for the 2 CUs implementation, and the resource utilization overall is basically doubled, this is logical since the extra CU requires the same amount resources as the original CU.

Version	ALM	Register	BRAM	DSP	Memory bits
Standard	2,195	4,501	45	2	319,488
SIMD2	2,624	4,775	45	4	540,672
2 CUs	4,620	9,110	90	6	638,976

Table 6.2: Linear classifier resource utilization

One hidden layer The resources for the implementation of a neural network with one hidden layer is presented in Table 6.3. The table divides the resources between the two layers in the network, namely the hidden layer and the output layer, which implements 25 and 10 neurons respectively.

Version	Layer type	ALM	Register	BRAM	DSP	Memory bits
Standard	**Hidden**	2,703	5,607	65	15	596,480
	Output	2,710	5,573	36	15	95,488
SIMD2	**Hidden**	2,773	5,604	65	15	596,480
	Output	3,652	7,190	24	30	107,648
2 CUs	**Hidden**	5,438	11,296	130	32	1,192,960
	Output	5,516	11,256	72	30	190,976

Table 6.3: One hidden layer NN resource utilization

The same tendencies as for the linear classifier can be seen here, for instance when using the *num_compute_units(2)* attribute, all the resources are approximately doubled, and the output layer in the SIMD2 version shows some increase in resources and a doubling of the DSPs. However since the hidden layer for this network has 25 neurons it cannot be vectorized with *num_simd_work_items(2)*, thus the resources used by this layer is the same as the standard implementation. The biggest change in this network is the addition of the *sigmoid* activation function, it causes the amount of DSPs to increase from 2 to 15 in the linear classifier. This is well within the total number of DSPs on this Arria 10 FPGA (see Table 6.1), but when optimizing a neural network it may be desirable to increase the parallelism more than tested here, which could quickly consume a big part of the available DSPs.

Another thing to note is that the BRAM and memory bits usage is lower in the output layer than the hidden layer, this is caused by the loop limits being defined at compile time for this neural network and the linear classifier, which means that the global load operation requires a smaller cache.

Two hidden layers Table 6.4 shows the resource utilization for the third implemented neural network, which has three layers, two hidden and an output layer. The layers implements 100, 25, and 10 neurons in that order.

Version	Layer type	ALM	Register	BRAM	DSP	Memory bits
Standard	**Hidden1**	4,481	7,510	154	17	2,167,808
	Hidden2	4,301	7,432	153	17	2,164,736
	Output	4,425	7,333	153	17	2,164,224
SIMD max	**Hidden1**	10,240	17,338	366	68	5,622,912
	Hidden2	4,453	7,435	153	17	2,164,736
	Output	6,353	10,657	210	34	3,238,272
2 CUs	**Hidden1**	8,082	14,716	204	34	2,191,616
	Hidden2	7,967	14,673	200	34	2,180,096
	Output	8,024	14,514	200	34	2,180,096

Table 6.4: Two hidden layer NN resource utilization

The kernels implementing the layers here has some differences compared to the last two networks, here both the weights and the loop limit are passed as arguments instead of being defined in a header. Due to this change the weight array must be indexed as a 1-dimensional array which requires an extra multiplication and an extra DSP some probably some extra logic. Also since the loop limit is no longer known at compile time, extra resources is used to accommodate a variable limit. The table shows a significant increase for all the categories except the DSPs which only increased by 2, however this was expected due to the indexing of the weight array.

Since the first hidden layer has 100 neurons it was possible to use a vectorization factor of 4 instead of 2. The second layer has 25 neurons and can not be vectorized, and the output layer has 10 neurons which allows a vectorization factor of 2. The SIMD max row shows a very high increase in resources for the vectorized layers, the DSP amount is expected, but the usage of ALMs and registers seems strangely high. However by reviewing the detailed estimation report, it seems most of the resource increase is caused by the extra logic surrounding the global loads, and for increased vectorization it is obviously necessary with more loads.

The extra CU gives an expected doubling of resources as shown on the 2 CU row.

6.3.2 CNN

The implemented CNN consists of 6 layers, 3 convolution layers and 3 fully connected layers. Figure 5.1 shows the CNNs architecture. In the previously considered neural networks each layer had its own OpenCL kernel, although it was not necessary to implement the networks this way, instead the same kernel could have been used for each layer. For the CNN it was desirable to have a generalized implementation, which can be used for CNNs with different architectures, hence the two kernel types does not use *reqd_work_group_size(x,y,z)* or *max_work_group_size(n)* to limit the number of work-items. This also means that *num_simd_work_items(n)* can not be used to vectorize the kernels.

Table 6.5 shows the resource utilization of the CNN with some variations. The first version in the table uses the convolution layer implemented as a task or single work-item kernel, and can therefore not be vectorized or replicated. The other versions uses the convolution layer implemented as an NDRange kernel, since the convolution layer does most of the computation, it is the only layer tested with varying number of CUs. The fully connected, FC layers were not changed across the versions, this is shown in the table as the FC layers resources are almost identical.

Version	Layer type	ALM	Register	BRAM	DSP	Memory bits
Task conv	**FC**	3,964	7,069	117	5	1,463,808
	CONV	11,658	25,581	110	23	1,523,456
NDRange conv	**FC**	3,968	7,083	117	5	1,463,808
	CONV	7,010	13,534	140	23	2,162,176
NDRange conv 2 CUs	**FC**	3,974	7,083	117	5	1,463,808
	CONV	13,293	25,256	280	46	4,324,352
NDRange conv 4 CUs	**FC**	4,000	7,082	117	5	1,463,808
	CONV	26,181	48,795	560	92	8,648,704

Table 6.5: CNN resource utilization

In the previously mentioned neural networks the sigmoid activation function was used, this resulted in an increase of DSP usage compared to the linear classifier. For this network the activation function is *RelU*, and it does not require DSPs as it only performs a comparison, this is reflected in the number of DSPs used here. Instead, most of the DSP usage comes from the complicated indexing, some variables that are calculated and the dot product calculation. Interestingly both the task based convolution kernel and the NDrange convolution kernel use the same amount of DSPs, however despite the difference in implementation, they use almost the same variables and the dot product is similarly calculated.

The table shows that the task based convolution kernel uses more ALMs and registers than the NDRange version without replication. A lot of the resource usage for both kernels seems to come from the global load operation, which then includes the necessary indexing logic. In addition all the loops with variable limits in the task based version is most likely the reason to the extra logic resources used. The BRAM and memory bits usage is somewhat higher for the NDRange kernel, but it is not clear why. The other two versions with 2 and 4 CUs of the NDRange convolution kernel gives the expected increase in resource utilization.

Table 6.6 shows the total resource utilization and the percentages of the total available resources on the Arria 10 FPGA board used. The total resources consists of resources the kernels use and extra logic surrounding the kernels. The BSP usage is also included and amounts to a relatively large portion of the total usage.

	ALM	**Register**	**BRAM**	**DSP**	**Memory bits**
Total	123,589 (29%)	206,342 (12%)	1,140 (42%)	99 (7%)	13,823,124 (25%)

Table 6.6: Total resource utilization for NDRange conv 4 CUs

6.4 Performance

This section shows the performance of the different neural networks and discusses the impact of the different attributes that were tested.

6.4.1 Neural nets

As before, the initial neural networks and the linear classifier is considered first. When testing the performance of these networks it was found that the *num_simd_work_items(n)* and *num_compute_units(n)* attributes had no effect, and the extra CUs even made the performance worse depending on the work-group size. It is hard to know exactly why the increased parallelism did not help, but

since the amount of processing done by each work-item and the number of work-items in the layers is quite low, it is likely that these attributes only creates more overhead. Especially when using several CUs, which requires a hardware scheduler in the FPGA to dispatch the work-groups [4], a low amount of work-items and thus small work-groups gives a lot of unnecessary overhead.

Since these attributes gave no improvements Table 6.7 displays the results from the standard implementation, which then represents the best results for each network. Each of the networks tests 500 images from the MNIST dataset, however as explained in Chapter 4 the linear classifier takes 28x28 input images and the other two networks uses a scaled down version of the images with size 20x20 as input.

	Layer num	Linear classifier	One hidden layer	Two hidden layers
Avg. kernel times	**1**	0.055ms	0.0473ms	0.169ms
	2	-	0.0128ms	0.0416ms
	3	-	-	0.0196ms
Total kernel time		27.5ms	30ms	111ms
Total wall time		36ms	62ms	180ms
macs		3,920,000	5,125,000	21,375,000
Mmac/s		108.9	82.7	118.9

Table 6.7: Neural networks performance

The total wall time in the table represents the time it takes for the host to start executing the kernels and wait until they are finished. The total kernel time on the other hand is the total execution time of the kernels, which is retrieved using the *clGetEventProfilingInfo()* API call. The separate kernels or layers average execution time also uses this method. The performance is also measured in *mac/s*, where a mac is the multiply–accumulate operation. The number of mac operations per layer in the networks is calculated by Equation **??**:

$$mac_per_layer = input_size * output_size * NUM_IMAGES \quad (6.1)$$

The total number of macs and the performance in Mmac/s is shown in the table. The performance is highest for the network with two hidden layers, this is probably because the first layer is allowed to do the highest amount of processing per kernel invocation, thus reducing the overhead. By looking at the difference between the total kernel times and total wall times, it is apparent that each kernel execution causes a significant overhead. The total kernel time for the linear classifier and one hidden layer network, shows that they perform a similar amount of computation, but the total wall time is much higher for the one hidden layer network which has two layers.

6.4.2 CNN

Table 6.8 shows the performance of the CNN implementation with the two different convolution layer types and extra CUs for the NDRange version. For these networks only a single test were run to measure the performance.

The task based version has the worst performance, which was expected due to its implementation. The first NDRconv version was also expected to have similar results since both versions use a single CU, but there is a difference of 5ms, which is a considerable time here. However, the NDRange convolution kernel has a simpler implementation which is also reflected in the resource utilization presented earlier, hence it is probably easier for the compiler to create a kernel requiring less cycles per mac. Increasing the number of CUs for the previous networks had no impact and even made the performance worse. For the NDRange convolution kernel on the other hand, doubling the amount of CUs almost halved the total kernel time. Increasing to 4 CUs almost halved the total kernel time again, but for the total wall time, the overhead of executing kernels becomes a bigger part of the execution time. These observations fits well with the reasoning made in the last section, for those networks there were probably too few work-items and work-groups to utilize the parallelism, in the convolution layers used here however, there are a lot more work-items and data to process.

	Layer num	**Task conv**	**NDRconv**	**2 CUs**	**4 CUs**
Kernel times	**CONV1**	4.35ms	3.33ms	1.62ms	0.874ms
	CONV2	4.51ms	2.62ms	1.33ms	0.691ms
	CONV3	4.19ms	1.48ms	0.757ms	0.4ms
	FC1	0.202ms	0.204ms	0.196ms	0.237ms
	FC2	0.035ms	0.0399ms	0.0319ms	0.0531ms
	FC3	0.0387ms	0.0343ms	0.0283ms	0.0592ms
Total kernel time		13.3ms	7.71ms	3.96ms	2.31ms
Total wall time		14.9ms	9.34ms	5.46ms	3.66ms
macs		1,972,480			
Mmac/s		132.4	211.2	361.3	538.9

Table 6.8: CNN performance

The number of macs for the fully connected layers are calculated with Equation 6.1, for the convolution layers Equation 6.2 is used:

$$mac_per_layer = out_h * out_w * filter_h * filter_w * channels_in * channels_out \tag{6.2}$$

6.5 Comparison

Since GPUs has been the go-to device for implementation of CNNs recent years, it is natural to compare the FPGA implementation with a GPU implmementation.

Performance To do this Cisco ran a bigger CNN on a NVIDIA Tegra X1 GPU. One classification for this network takes 26.38M macs, and the Tegra X1 chip performed this in about 2.2 ms. This results in 11.99 Gmac/s, while the best performing FPGA implementation made here reached about 0.54 Gmac/s. That means the GPU implementation is 22.2 times faster. It should also be noted that since the networks are so fast, the times may vary somewhat. Hence, using a bigger network when testing, especially on the GPU, gives more accurate results.

Cost Comparing the cost between using FPGAs or GPUs is difficult due to several reasons. First of all, both devices come in many different types, from low end to high end devices. The Tegra X1 for example is a high end mobile processor with a GPU, but since it is a mobile processor it is a relatively low end GPU compared to a NVIDIA Tesla P40 for instance. The Tesla P40 is a high end maximum throughput GPU from NVIDIA, especially designed for deep learning inference, the price for this GPU is around 5700$ according to this site [33]. On this site [7] the Tegra X1 module price is 400$. On the FPGA side, the smallest Arria 10 GX 160 has a price of 320$ according to [6]. Although it is possible to find some prices, these prices may vary for companies ordering large quanta, they do however, give an indication of the devices cost.

The solution from this project is far from an optimized solution which also makes it hard to do a performance/cost comparison it with the GPU implementation. Some points to note though is that the available resources of an Arria 10 GX 160, which seems to be at a similar price point as the Tegra X1, is a lot less than the one used in this project, Arria 10 GX 1150. While the Arria 10 used here has 2713 M20K memory blocks available, the GX 160 only has 440 [14], this is less than the 4 CU version the CNN design uses, which is 677 just for the OpenCL kernels. In addition to this a more optimized design will likely use even more resources to achieve competing performance with GPUs. Therefore bigger and more expensive FPGAs than the GX 160 will be necessary.

Power It was also desirable to include a comparison of power usage between the GPU and the FPGA. Intel provides a program called Board Test System [15] which includes a power monitor, unfortunately due to problems with this program, a power measurement from the Arria 10 was not performed.

Chapter 7

Discussion

This chapter discusses the achieved solution and the project tasks. The results are also discussed with thoughts on improvements.

7.1 Project tasks

The main goal of this project was to implement an infrastructure to test various neural networks and then finally a CNN that is used to recognize a "wake up word". And also ideally it should be generalized to any CNN application or architecture. In order to have the implementation adaptable for various neural networks, the CNN host file uses the *layer_config* array explained in Section 5.2 to set the properties for a network. This setup can be used to implement networks containing convolution layers and fully connected layers with different variations. Specifically for the convolution layer, parameters such as filter size and strides can be controlled. These layers and their parameters was able to implement the wake up word CNN architecture provided by Cisco, however some other architectures may require other layers and properties. Hence the solution is not fully generalized.

To make the solution more generalized, one feature that should be added to the convolution layer is padding, this allows the spatial size of the input to be preserved on the output. There is also the pooling layer, which is not implemented here, but is used in some CNNs to reduce the size of the output feature maps. According to [31] there are many that thinks the pooling layer is not really necessary since convolution layers with larger strides can also be used to reduce the size, but for a generalized CNN framework a pooling layer should be available.

There was also a point to provide an easy way to feed patterns to the network. For the live *wake up word* implementation inputs are simply fed into the network

from a recording microphone. If not using it live and instead using test inputs, the name of the input file must be set in the host file source code. With regards to the networks' weights, they are read from *.npy* files containing numpy arrays, this is a practical format since many machine learning frameworks that are used to train neural networks use Python and numpy. Weights for other networks should follow the same naming scheme as used here, this makes it easy to control each layers weights and initialize them in the host application. Although there are probably other good ways to do this. Presenting or using the results is very application specific, hence the results are read and verified for test inputs with a golden reference. For the live wake up word version there are only two outputs which is used to check whether the correct wake up word was recorded or not.

7.2 Results and improvements discussion

The results presented in the last chapter showed that the CNN implementation was able to classify one input in about 3.66 ms. When comparing this against a GPU implementation, it was found that the GPU was 22 times faster. Also considering the total amount of resources used shown in Table 6.6, it is apparent that there is room for improvements, both by optimizing the resources used and better utilizing the available resources on the FPGA.

For the live wake up word version, the total time after the word has been said takes around 14 ms. This extra time comes from the preprocessing part when reading a new input, and equals about 10 ms since the CNN uses around 3.66 ms. Since this wake up word implementation is a practical application that should interact with a person, it is natural to consider how 14 ms is perceived by humans. A good comparison for this delay is the time between frames on a standard computer monitor, which is 1/60 Hz = 16.67 ms and thus is higher than the CNNs delay. That means the result from the wake up word CNN should be available on the next frame. 60 Hz is a relatively high refresh rate, hence a somewhat slower classification time should also be acceptable.

Although one can evaluate the speed of this network to be adequate as a wake up word application and probably other applications, improving the implementation is desirable. Chapter 3 presented some related work where for example Intel's DLA [8] was able to achieve 1,382 GFLOPS on an Arria 10 device. Their architecture, shown in Figure 3.1, is able to compute all the layers of a CNN without having to execute the kernels multiple times, this is done with the help of the *Sequencer* kernel and using the Intel FPGA channel extension to keep data on the chip.

The PipeCNN [35] architecture also used the Intel FPGA channel extension to communicate feature data and weights between the 4 main kernels. However in comparison to Intel's solution, PipeCNN needs to read and write back data between each convolution layer and fully connected layer. And then also re-execute the kernels for each layer, same as it is done here. Anyway keeping as much as possible of

the computation and data on chip and without host interaction is an improvement that should be considered for this implementation.

For a GPU it is normal to store batches of samples before processing it, due to time demanding context switches, this helps to minimize memory accesses and executing its tasks as fast as possible. The downside of this batching is that it adds latency since it has to wait for samples. An FPGA on the other hand can process incoming samples as soon as it is ready, this difference makes the FPGA more suitable for low latency applications. The implementation does however need to utilize this advantage. The implementation done here as well as PipeCNN, needs to execute the kernels for each layer and also for new inputs, which means that a new input is processed as soon as the prior is finished, but each execution call adds extra delay. Tables 6.7 and 6.8 shows this delay. As mentioned the classification time for the wake up word CNN is about 3.66 ms, which is less than the preprocessing time at around 10ms. The current input will therefore be finished processing before a new input is ready, meaning the execution delay is acceptable for this application.

The FINN [34] framework took a different approach by implementing BNNs instead of traditional CNNs with 32 bit floats. Advantages of using BNNs is higher classification speeds, less resource usage and since weights are binary they require less storage capacity. Instead of using Intel FPGA SDK for OpenCL, they used Vivado HLS to make the framework. Vivado HLS has the ability to define new data types with arbitrary precision [38], which is useful for applications such as a BNN, and to save FPGA resources. The Intel FPGA SDK for OpenCL version used here (16.0.2.222), supports data types down to 8 bits. It is also possible to use bit masking and then let AOC disregard the unnecessary bits [4], whether this is suitable to implement a BNN is unknown. Fortunately in version 17.0 support is added for arbitrary precision integers [17]. The FINN paper points to research that shows neural networks can accurately classify with reduced precision weights and activations. For many applications somewhat reduced accuracy may be acceptable, hence using a similar approach as FINN might be a good direction to improve performance.

Chapter 8

Conclusion

The work presented in this thesis has resulted in an infrastructure that is capable of testing various neural networks. Different network architectures can be easily configured in a header file by setting parameters such as sizes and number of layers. The two most important types of layers for a CNN were implemented, namely the convolution and fully connected layers. The Intel FPGA SDK for OpenCL was used to implement the OpenCL kernels and host code, and to perform full compilation of the code to a final FPGA image. An Arria 10 GX FPGA was used to test the implemented neural networks.

The "wake up word" CNN that was implemented gave correct results for two provided test inputs. In addition a way to record audio and perform the necessary preprocessing was added, with this it was possible to test the implementation in a live setup, running the CNN on the FPGA and speaking directly to it. This made testing with different inputs easier and further verified correct behaviour of the CNN. The addition of live testing was not a main goal of the project, so more can be done to make it work more optimally alongside the CNN. It was however useful to test the network in a more realistic setup to see the extra delay from the preprocessing.

Before the CNN was implemented, three smaller networks was created as a mean to find out how to properly do several layers and try the impact of some optimization techniques. It was found that since these networks processed a small amount of data per execution, the attributes increasing the parallelism had zero impact on performance, or even made it worse. For the CNN on the other hand, extra parallelism increased the performance in mac/s from 211.2 Mmac/s with 1 CU to 538 Mmac/s with 4 CUs. This resulted in the best classification time of about 3.6 ms. Comparing it to a CNN running on the NVIDIA Tegra X1 GPU, the GPU was 22.2 times faster with 11.99 Gmac/s. The live wake up word version used about 14 ms with preprocessing and classification. This time is considered acceptable for

this application.

A comparison between the prices of GPUs and FPGAs was also discussed. However, it is difficult to find out what the normal price points are for these devices, especially considering prices are usually lower when ordering in large quanta. Some prices found along with the resources used, may indicate that bigger and more expensive FPGAs will be necessary to create highly optimized CNNs. Due to difficulties measuring power usage on the FPGA a power comparison was not performed.

8.1 Future Work

A natural way forward will be to improve the performance of the CNN, to do this the most impactful layer in a CNN, the convolution layer, should be the main focus. As discussed earlier, keeping as much of the computation and data movement on chip will be important. Intel FPGA's channel extension is one way to move data between kernels. The OpenCL standard provides pipes that also may be used, an advantage of these are that they can be used for other platforms as well. In addition buffering of data should also be implemented. Although some parallelism was used, there were still available resources on the Arria 10, hence utilizing more of these resources by for example increasing the parallelism of the kernels should be explored.

The solution should be generalized further by implementing a kernel for pooling layers and also adding support for padding in the convolution layers. Layers for normalization and softmax can also be considered.

Another route, but still relevant to the previously mentioned improvements, is to explore the use of fixed point data types with reduced precision, for example 8 or 16 bit fixed point instead of 32 bit floating point used here. Or possibly looking into BNNs, especially if version 17.0 of the Intel FPGA SDK for OpenCL can be used, which introduces arbitrary precision data types.

References

[1] Handwritten digit database. Derived from original MNIST, http://cis.jhu.edu/ sachin/digit/digit.html, Accessed May, 2017.

[2] WAVE PCM soundfile format. http://soundfile.sapp.org/doc/WaveFormat/, Accessed May, 2017.

[3] Altera. FPGA Architecture White Paper. WP-01003-1.0, July 2006.

[4] Altera. *Altera SDK for OpenCL Best Practices Guide UG-OCL003*, Quartus Prime Design Suite: 16.0 edition, December 2016.

[5] Altera. *Arria 10 FPGA Development Kit User Guide UG-20007*, February 2016.

[6] Arrow. Arria 10 GX 160. https://www.arrow.com/en/products/10ax016c4u19e3sg/alteraintel-programmable-solutions, Accessed June, 2017.

[7] Arrow. NVIDIA Tegra X1 module. https://www.arrow.com/en/products/900-82180-0001-000/nvidia, Accessed June, 2017.

[8] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu. An OpenCL(TM) Deep Learning Accelerator on Arria 10. *CoRR*, abs/1701.03534, 2017.

[9] Berkeley Artificial Intelligence Research (BAIR). Caffe. http://caffe.berkeleyvision.org/, Accessed May, 2017.

[10] C. M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.

[11] S. Chakroborty and G. Saha. Feature selection using singular value decomposition and QR factorization with column pivoting for text-independent speaker identification. *Speech Communication*, 52(9):693–709, 2010.

[12] T. S. Czajkowski, D. Neto, M. Kinsner, U. Aydonat, J. Wong, D. Denisenko, P. Yiannacouras, and J. Freeman. OpenCL for FPGAs: Prototyping a Compiler. *Int'l Conf. Reconfigurable Systems and Algorithms, ERSA'12*, 2012.

[13] S. B. Davis and P. Mermelstein. Comparison of Parametric Representations for Monosyllabic Word Recognition in Continuously Spoken Sentences. *IEEE Trans ASSP*, 28(4):357–366, 1980.

[14] Intel. Arria 10 Features. https://www.altera.com/products/fpga/arria-series/arria-10/features.html, Accessed June, 2017.

[15] Intel. Arria 10 GX FPGA Development Kit. https://www.altera.com/products/boards_and_kits/dev-kits/altera/kit-a10-gx-fpga.html, Accessed May, 2017.

[16] Intel. *Arria 10 Device Overview*, 2016.10.31 edition, October 2016.

[17] Intel. *Intel FPGA SDK for OpenCL Programming Guide UG-OCL002*, Quartus Prime Design Suite: 17.0 edition, May 2017.

[18] Intel FPGA. SDK for OpenCL Developer Zone. https://www.altera.com/products/design-software/embedded-software-developers/opencl/developer-zone.html, Accessed May, 2017.

[19] Khronos Group. OpenCL. https://www.khronos.org/opencl/, Accessed May, 2017.

[20] Khronos OpenCL Working Group. The OpenCL Specification, Version 1.0, 2009. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-1.0.pdf, Accessed May, 2017.

[21] Y. LeCun. The mnist database of handwritten digits. MNIST, http://yann.lecun.com/exdb/mnist/, Accessed May, 2017.

[22] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[23] T. M. Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math, March 1997.

[24] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg. *OpenCL Programming Guide*. Addison-Wesley, July 2011.

[25] A. Ng. Machine learning. Coursera, https://www.coursera.org/learn/machine-learning/, Accessed Feb, 2017.

[26] M. Nielsen. Neural Networks and Deep Learning. http://neuralnetworksanddeeplearning.com/index.html, Online book, Accessed May, 2017.

[27] NVIDIA. NVIDIA Tegra X1. http://www.nvidia.com/object/tegra-x1-processor.html, Accessed June, 2017.

[28] PortAudio. PortAudio Portable Cross-platform Audio I/O. http://www.portaudio.com/, Accessed May, 2017.

[29] C. Rogers. Cnpy. https://github.com/rogersce/cnpy, Accessed May, 2017.

[30] A. L. Samuel. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development* , 3(3):210–229, 1959.

[31] Stanford. CS231n Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io, Accessed May, 2017.

[32] TensorFlow. Mnist for ml beginners. https://www.tensorflow.org/get_started/mnist/beginners, Accessed May, 2017.

[33] Thinkmate. NVIDIA Tesla P40. http://www.thinkmate.com/product/nvidia/900-2g610-0000-000, Accessed June, 2017.

[34] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. H. W. Leong, M. Jahre, and K. A. Vissers. FINN: A framework for fast, scalable binarized neural network inference. *CoRR*, abs/1612.07119, 2016.

[35] D. Wang, J. An, and K. Xu. PipeCNN: An OpenCL-Based FPGA Accelerator for Large-Scale Convolution Neuron Networks. *CoRR*, abs/1611.02450, 2016.

[36] S. Windh, X. Ma, R. J. Halstead, P. Budhkar, Z. Luna, O. Hussaini, and W. A. Najjar. High-Level Language Tools for Reconfigurable Computing. *Proceedings of the IEEE*, 103(3):390–408, 3 2015.

[37] Xilinx. Vivado High-Level Synthesis. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, Accessed June, 2017.

[38] Xilinx. *Vivado Design Suite User Guide, High-Level Synthesis UG902*, v2016.2 edition, June 2016.

Appendix

A.1 Source code

For the three initial neural networks, only the two layer neural networks' host code is added due to their similarities. Hovewer, everything is included in the attachements.

A.1.1 Linear classifier

```
1  #ifndef LINEAR_CLASSIFIER_H
2  #define LINEAR_CLASSIFIER_H
3
4  // Dimensions
5  #define W_y  10
6  #define W_x  785 // 28x28 + 1 for bias
7
8  #define X_y  785 // 28x28 + 1 for bias
9  #define X_x  1
10
11 #define Y_y  10
12 #define Y_x  1
13
14
15
16 #endif // LINEAR_CLASSIFIER_H
```

Listing 1: linear_classifier.h

```
1  #include "../host/inc/linear_classifier.h"
2  #include "../host/inc/weights_linear_classifier.h"
3
4  #ifndef SIMD_WORK_ITEMS
5  #define SIMD_WORK_ITEMS 2 // must be a power of two, and
      divisible with number of kernels
```

```
6  #endif
7
8  #ifndef NUM_COMPUTE_UNITS
9  #define NUM_COMPUTE_UNITS 2
10 #endif
11
12 __kernel
13 __attribute((reqd_work_group_size(10,1,1)))
14 //__attribute((max_work_group_size(10)))
15 __attribute((num_simd_work_items(SIMD_WORK_ITEMS)))
16 //__attribute((num_compute_units(NUM_COMPUTE_UNITS)))
17 void linear_classifier(
18                  // Input and output matrices
19                  __global float *restrict Y,
20                  __global float *restrict X)
21 {
22     // Global ID index (offset within the NDRange)
23     //int global_id = get_global_id(0);
24
25     // Local ID index (offset within work group)
26     int local_id = get_local_id(0);
27
28     float running_sum = 0.0f;
29
30     for (int k = 0; k < W_x; ++k)
31     {
32         running_sum += W[local_id][k] * X[k];
33     }
34     barrier(CLK_LOCAL_MEM_FENCE); // try remove this
35
36     // Store result in matrix Y
37     Y[local_id] = running_sum;
38 }
```

Listing 2: linear_classifier.cl

A.1.2 One hidden layer neural network

```
1  #ifndef ONE_LAYER_NN_H
2  #define ONE_LAYER_NN_H
3
4  #define IMG_SIZE    20*20
5
6  // Dimensions
7  // Input
8  #define X_y  401 // 20x20 + 1 for bias
```

```
 9 #define X_x  1
10
11 // Output
12 #define Y_y  10
13 #define Y_x  1
14
15 // Theta1
16 #define theta1_y 25
17 #define theta1_x 401 // 20x20 + 1 for bias
18
19 // Theta2
20 #define theta2_y 10
21 #define theta2_x 26
22
23
24 #endif // ONE_LAYER_NN_H
```

Listing 3: one_layer_nn.h

```
 1 #include "../host/inc/one_layer_nn.h"
 2 #include "../host/inc/theta1_weights.h"
 3 #include "../host/inc/theta2_weights.h"
 4
 5 #ifndef SIMD_WORK_ITEMS
 6 #define SIMD_WORK_ITEMS 2 // must be a power of two, and
      divisible with number of kernels
 7 #endif
 8
 9 #ifndef NUM_COMPUTE_UNITS
10 #define NUM_COMPUTE_UNITS 2
11 #endif
12
13 #define SIGMOID(x) (1.0f / (1 + exp(-x)))
14
15 __kernel
16 //__attribute((reqd_work_group_size(25,1,1)))
17 __attribute((max_work_group_size(25)))
18 //__attribute((num_simd_work_items(SIMD_WORK_ITEMS))) //
      Not possible here since work group size must be
      divisible and a power of 2
19 __attribute((num_compute_units(NUM_COMPUTE_UNITS)))
20 void one_layer_nn_hidden_layer(
21                  // Input and output matrices
22                  __global float *restrict Y,
23                  __global float *restrict X)
24 {
```

```
25      // Global ID index (offset within the NDRange)
26      int global_id = get_global_id(0);
27
28      // Local ID index (offset within a work group)
29      //int local_id = get_local_id(0);
30
31      float running_sum = 0.0f;
32
33      for (int k = 0; k < theta1_x; ++k)
34      {
35          running_sum += Theta1[global_id][k] * X[k];
36      }
37      barrier(CLK_LOCAL_MEM_FENCE);
38
39      // Store result in matrix Y
40      Y[global_id] = SIGMOID(running_sum);
41  }
42
43  __kernel
44  //__attribute((reqd_work_group_size(10,1,1)))
45  __attribute((max_work_group_size(10)))
46  //__attribute((num_simd_work_items(SIMD_WORK_ITEMS)))
47  __attribute((num_compute_units(NUM_COMPUTE_UNITS)))
48  void one_layer_nn_output_layer(
49                      // Input and output matrices
50                      __global float *restrict Y,
51                      __global float *restrict X)
52  {
53      // Global ID index (offset within the NDRange)
54      int global_id = get_global_id(0);
55
56      // Local ID index (offset within a work group)
57      //int local_id = get_local_id(0);
58
59      float running_sum = 0.0f;
60
61      running_sum += Theta2[global_id][0];
62
63      for (int k = 1; k < theta2_x; ++k)
64      {
65          running_sum += Theta2[global_id][k] * X[k-1];
66      }
67      barrier(CLK_LOCAL_MEM_FENCE);
68
69      // Store result in matrix Y
```

```
70      Y[global_id] = SIGMOID(running_sum);
71  }
```

Listing 4: one_layer_nn.cl

A.1.3 Two hidden layers neural network

```
1   #ifndef TWO_LAYERS_NN_H
2   #define TWO_LAYERS_NN_H
3
4   #define IMG_SIZE    20*20
5
6   // Dimensions
7   // Input
8   #define X_y  400 // 20x20
9   #define X_x  1
10
11  // Output
12  #define Y_y  10
13  #define Y_x  1
14
15  // Theta1
16  #define theta1_y 100
17  #define theta1_x 401 // 20x20 + 1 for bias
18
19  // Theta1
20  #define theta2_y 25
21  #define theta2_x 101 // 20x20 + 1 for bias
22
23  // Theta3
24  #define theta3_y 10
25  #define theta3_x 26
26
27
28  #endif // TWO_LAYERS_NN_H
```

Listing 5: two_layers_nn.h

```
1   #include "../host/inc/two_layers_nn.h"
2
3   #ifndef SIMD_WORK_ITEMS
4   #define SIMD_WORK_ITEMS 2 // must be a power of two, and
       divisible with number of kernels
5   #endif
6
7   #ifndef NUM_COMPUTE_UNITS
8   #define NUM_COMPUTE_UNITS 2
```

```
 9 #endif
10
11 #define SIGMOID(x) (1.0f / (1 + exp(-x)))
12
13 __kernel
14 __attribute((reqd_work_group_size(100,1,1)))
15 //__attribute((max_work_group_size(100)))
16 __attribute((num_simd_work_items(4)))
17 //__attribute((num_compute_units(NUM_COMPUTE_UNITS)))
18 void hidden_layer_nn_100(
19                     // Input, weights and output matrices
20                     __global float *restrict Y,
21                     __global const float *restrict W,
22                     __global float *restrict X,
23                     // Widths of matrices.
24                     int W_width)
25 {
26     // Global ID index (offset within the NDRange)
27     //int global_id = get_global_id(0);
28
29     // Local ID index (offset within a work group)
30     int local_id = get_local_id(0);
31
32     float running_sum = 0.0f;
33
34     running_sum += W[local_id*W_width];
35
36     for (int k = 1; k < W_width; ++k)
37     {
38         running_sum += W[local_id*W_width + k] * X[k-1];
39     }
40     barrier(CLK_LOCAL_MEM_FENCE);
41
42     // Store result in matrix Y
43     Y[local_id] = SIGMOID(running_sum);
44 }
45
46 __kernel
47 //__attribute((reqd_work_group_size(25,1,1)))
48 __attribute((max_work_group_size(25)))
49 //__attribute((num_simd_work_items(SIMD_WORK_ITEMS)) //
       Not possible here since work group size must be
       divisible and a power of 2
50 __attribute((num_compute_units(NUM_COMPUTE_UNITS)))
51 void hidden_layer_nn_25(
```

```
52                      // Input, weights and output matrices
53                      __global float *restrict Y,
54                      __global const float *restrict W,
55                      __global float *restrict X,
56                      // Widths of matrices.
57                      int W_width)
58  {
59      // Global ID index (offset within the NDRange)
60      int global_id = get_global_id(0);
61
62      // Local ID index (offset within a work group)
63      //int local_id = get_local_id(0);
64
65      float running_sum = 0.0f;
66
67      running_sum += W[global_id*W_width];
68
69      for (int k = 1; k < W_width; ++k)
70      {
71          running_sum += W[global_id*W_width + k] * X[k-1];
72      }
73      barrier(CLK_LOCAL_MEM_FENCE);
74
75      // Store result in matrix Y
76      Y[global_id] = SIGMOID(running_sum);
77  }
78
79  __kernel
80  __attribute((reqd_work_group_size(10,1,1)))
81  //__attribute((max_work_group_size(10)))
82  __attribute((num_simd_work_items(SIMD_WORK_ITEMS)))
83  //__attribute((num_compute_units(NUM_COMPUTE_UNITS)))
84  void hidden_layer_nn_10(
85                      // Input and output matrices
86                      __global float *restrict Y,
87                      __global const float *restrict W,
88                      __global float *restrict X,
89                      // Widths of matrices.
90                      int W_width) //int X_width)
91  {
92      // Global ID index (offset within the NDRange)
93      //int global_id = get_global_id(0);
94
95      // Local ID index (offset within a work group)
96      int local_id = get_local_id(0);
```

```
97
98      float running_sum = 0.0f;
99
100     running_sum += W[local_id*W_width];
101
102     for (int k = 1; k < W_width; ++k)
103     {
104         running_sum += W[local_id*W_width + k] * X[k-1];
105     }
106     barrier(CLK_LOCAL_MEM_FENCE);
107
108     // Store result in matrix Y
109     Y[local_id] = SIGMOID(running_sum);
110 }
```

Listing 6: two_layers_nn.cl

```
1  #include <sys/stat.h>
2  #include <iostream>
3  #include <fstream>
4  #include <iomanip>
5  #include <math.h>
6  #include "CL/opencl.h"
7  #include "AOCLUtils/aocl_utils.h"
8  #include "two_layers_nn.h"
9  #include "../inc/theta1_weights_2h_1d.h"
10 #include "../inc/theta2_weights_2h_1d.h"
11 #include "../inc/theta3_weights_2h_1d.h"
12
13 using namespace aocl_utils;
14
15 #define NUM_IMAGES  500
16 #define TEST_NUM    3
17
18 // Kernels.
19 enum Kernel {
20   KERNEL_H1_LAYER = 0,
21   KERNEL_H2_LAYER,
22   KERNEL_OUTPUT_LAYER,
23
24   NUM_KERNELS
25 };
26
27 unsigned int correct_count = 0;
28
29 // OpenCL runtime configuration
```

```
30 cl_platform_id platform = NULL;
31 unsigned num_devices = 0;
32 scoped_array<cl_device_id> device; // num_devices elements
33 cl_context context = NULL;
34 scoped_array<cl_command_queue> queue; // num_devices
       elements
35 cl_program program = NULL;
36 scoped_array<cl_kernel> kernel; // num_devices elements/
       num_kernels
37
38 scoped_array<cl_mem> weights_buf;
39 scoped_array<cl_mem> input_x_buf;
40 scoped_array<cl_mem> z1_buf;
41 scoped_array<cl_mem> z2_buf;
42 scoped_array<cl_mem> output_buf;
43
44 float* input_x[NUM_IMAGES];
45 float* output;
46 float* weights[NUM_KERNELS];
47
48 // Problem data
49 unsigned X_height = X_y;
50 unsigned X_width  = X_x;
51 unsigned Y_height = Y_y;
52 unsigned Y_width  = Y_x;
53
54 unsigned theta1_width = theta1_x;
55 unsigned theta2_width = theta2_x;
56 unsigned theta3_width = theta3_x;
57
58 // Open MNIST dataset file
59 std::ifstream file("../../mnist_dataset/coursera/data3",
       std::ios::in | std::ios::binary);
60
61 // Open time log file
62 std::ofstream log_file("logs/time_log_std_tmp.log");
63
64 // Tot time
65 cl_ulong tot_kernel_time_ns_h1 = 0;
66 cl_ulong tot_kernel_time_ns_h2 = 0;
67 cl_ulong tot_kernel_time_ns_output = 0;
68 double total_time;
69
70 // Function prototypes
71 void read_mnist();
```

```
72 bool init_opencl();
73 void run();
74 void verify();
75 void cleanup();
76
77 int main(void) {
78   // Read the MNIST data
79   read_mnist();
80
81   // Initialize OpenCL.
82   if (!init_opencl()) {
83     return -1;
84   }
85
86   // Run the kernel
87   run();
88
89   // Free the resources allocated
90   cleanup();
91
92   // Calculate accuracy
93   float accuracy = ((float)correct_count/NUM_IMAGES)*100;
94   printf("Accuracy: %f\n", accuracy);
95   return 0;
96 }
97
98 /////////// HELPER FUNCTIONS //////////
99
100 void read_mnist() {
101   for (int i = 0; i < NUM_IMAGES; i++) {
102     float buffer[IMG_SIZE];
103
104     if (file.is_open()){
105       file.read((char*)(buffer), sizeof(float)*IMG_SIZE);
           // values are floats here
106     } else {
107       printf("Could not read file\n");
108       file.close();
109     }
110
111     // 64 byte aligned malloc for DMA
112     input_x[i] = (float*)alignedMalloc(sizeof(float)*
          IMG_SIZE);
113     memcpy(input_x[i], buffer, sizeof(float)*IMG_SIZE);
114   }
```

```
115
116    // 64 byte aligned malloc for DMA
117    output = (float*)alignedMalloc(sizeof(float)*Y_y);
118
119    weights[KERNEL_H1_LAYER] = (float*)alignedMalloc(sizeof(
           float)*theta1_y*theta1_x);
120    weights[KERNEL_H2_LAYER] = (float*)alignedMalloc(sizeof(
           float)*theta2_y*theta2_x);
121    weights[KERNEL_OUTPUT_LAYER] = (float*)alignedMalloc(
           sizeof(float)*theta3_y*theta3_x);
122
123    memcpy(weights[KERNEL_H1_LAYER], Theta1, sizeof(float)*
           theta1_y*theta1_x);
124    memcpy(weights[KERNEL_H2_LAYER], Theta2, sizeof(float)*
           theta2_y*theta2_x);
125    memcpy(weights[KERNEL_OUTPUT_LAYER], Theta3, sizeof(float
           )*theta3_y*theta3_x);
126  }
127
128  // Initializes the OpenCL objects
129  bool init_opencl() {
130    cl_int status;
131
132    printf("Initializing OpenCL\n");
133
134    if (!setCwdToExeDir()) {
135      return false;
136    }
137
138    // Get the OpenCL platform
139    platform = findPlatform("Altera");
140    if (platform == NULL) {
141      printf("ERROR: Unable to find Altera OpenCL platform.\n
             ");
142    }
143
144    // Query the available OpenCL device.
145    device.reset(getDevices(platform, CL_DEVICE_TYPE_ALL, &
           num_devices));
146    printf("Platform: %s\n", getPlatformName(platform).c_str
           ());
147    printf("Using %d device(s)\n", num_devices);
148    for (unsigned i = 0; i < num_devices; ++i) {
149      printf("  %s\n", getDeviceName(device[i]).c_str());
150    }
```

```
151
152     // Create the context
153     context = clCreateContext(NULL, num_devices, device, &
            oclContextCallback, NULL, &status);
154     checkError(status, "Failed to create context");
155
156     // Create the program for all device. Use the first
            device as the
157     // representative device (assuming all device are of the
            same type)
158     std::string binary_file = getBoardBinaryFile("
            two_layers_nn_simd_max", device[0]);
159     printf("Using AOCX: %s\n", binary_file.c_str());
160     program = createProgramFromBinary(context, binary_file.
            c_str(), device , num_devices);
161
162     // Build the program that was just created
163     status = clBuildProgram(program, 0, NULL, "", NULL, NULL)
            ;
164     checkError(status, "Failed to build program");
165
166     // Create per-device objects
167     queue.reset(num_devices);
168     kernel.reset(NUM_KERNELS);
169     weights_buf.reset(num_devices*NUM_KERNELS);
170     input_x_buf.reset(num_devices*NUM_IMAGES);
171     z1_buf.reset(num_devices);
172     z2_buf.reset(num_devices);
173     output_buf.reset(num_devices*NUM_IMAGES);
174
175     const unsigned num_block_rows = Y_height; // Y ?? /
            BLOCK_SIZE;
176
177     // Command queue
178     queue[0] = clCreateCommandQueue(context, device[0],
            CL_QUEUE_PROFILING_ENABLE, &status);
179     checkError(status, "Failed to create command queue");
180
181
182     const char *kernel_name_100 = "hidden_layer_nn_100";
183     kernel[KERNEL_H1_LAYER] = clCreateKernel(program,
            kernel_name_100, &status);
184     checkError(status, "Failed to create kernel, layer: %d",
            KERNEL_H1_LAYER);
185
```

```
186    const char *kernel_name_25 = "hidden_layer_nn_25";
187    kernel[KERNEL_H2_LAYER] = clCreateKernel(program,
           kernel_name_25, &status);
188    checkError(status, "Failed to create kernel, layer: %d",
           KERNEL_H2_LAYER);
189
190    const char *kernel_name_10 = "hidden_layer_nn_10";
191    kernel[KERNEL_OUTPUT_LAYER] = clCreateKernel(program,
           kernel_name_10, &status);
192    checkError(status, "Failed to create kernel, layer: %d",
           KERNEL_OUTPUT_LAYER);
193
194    // Create weight buffers
195    weights_buf[KERNEL_H1_LAYER] = clCreateBuffer(context,
           CL_MEM_READ_ONLY | CL_MEM_BANK_1_ALTERA,
196         theta1_y * theta1_x * sizeof(float), NULL, &status);
197    checkError(status, "Failed to create buffer for weights
           Theta1");
198
199    weights_buf[KERNEL_H2_LAYER] = clCreateBuffer(context,
           CL_MEM_READ_ONLY | CL_MEM_BANK_1_ALTERA,
200         theta2_y * theta2_x * sizeof(float), NULL, &status);
201    checkError(status, "Failed to create buffer for weights
           Theta2");
202
203    weights_buf[KERNEL_OUTPUT_LAYER] = clCreateBuffer(context
           , CL_MEM_READ_ONLY | CL_MEM_BANK_1_ALTERA,
204         theta3_y * theta3_x * sizeof(float), NULL, &status);
205    checkError(status, "Failed to create buffer for weights
           Theta3");
206
207    // Write weight values to the weight buffers.
208    // Can be done here since the weight values does not
           change.
209    status = clEnqueueWriteBuffer(queue[0], weights_buf[
           KERNEL_H1_LAYER], CL_TRUE, // CL_FALSE
210      0, theta1_y * theta1_x * sizeof(float), weights[
             KERNEL_H1_LAYER], 0, NULL, NULL);
211    checkError(status, "Failed to transfer input X");
212
213    status = clEnqueueWriteBuffer(queue[0], weights_buf[
           KERNEL_H2_LAYER], CL_TRUE, // CL_FALSE
214      0, theta2_y * theta2_x * sizeof(float), weights[
             KERNEL_H2_LAYER], 0, NULL, NULL);
215    checkError(status, "Failed to transfer input X");
```

```
216
217   status = clEnqueueWriteBuffer(queue[0], weights_buf[
        KERNEL_OUTPUT_LAYER], CL_TRUE, // CL_FALSE
218     0, theta3_y * theta3_x * sizeof(float), weights[
          KERNEL_OUTPUT_LAYER], 0, NULL, NULL);
219   checkError(status, "Failed to transfer input X");
220
221   for (unsigned i = 0; i < NUM_IMAGES; i++) {
222     // Create input and output buffers
223     input_x_buf[i] = clCreateBuffer(context,
          CL_MEM_READ_ONLY | CL_MEM_BANK_2_ALTERA,
224        X_height * X_width * sizeof(float), NULL, &status);
225     checkError(status, "Failed to create buffer for input")
          ;
226
227     output_buf[i] = clCreateBuffer(context,
          CL_MEM_WRITE_ONLY | CL_MEM_BANK_1_ALTERA,
228        Y_height * Y_width * sizeof(float), NULL, &status);
229     checkError(status, "Failed to create buffer for output"
          );
230
231     // Fill input with all input images
232     status = clEnqueueWriteBuffer(queue[0], input_x_buf[i],
           CL_TRUE, // CL_FALSE
233       0, X_width * X_height * sizeof(float), input_x[i], 0,
            NULL, NULL);
234     checkError(status, "Failed to transfer input X");
235   }
236
237   // Create temporary buffers
238   z1_buf[0] = clCreateBuffer(context, CL_MEM_READ_WRITE |
        CL_MEM_BANK_2_ALTERA,
239      theta1_y * sizeof(float), NULL, &status);
240   checkError(status, "Failed to create buffer for z");
241
242   z2_buf[0] = clCreateBuffer(context, CL_MEM_READ_WRITE |
        CL_MEM_BANK_2_ALTERA,
243      theta2_y * sizeof(float), NULL, &status);
244   checkError(status, "Failed to create buffer for z");
245
246   return true;
247 }
248
249 void run() {
250   cl_int status;
```

```
251
252    // Launch kernels.
253    // This is the portion of time that we'll be measuring
         for throughput
254    // benchmarking.
255    scoped_array<cl_event> kernel_h1_event(num_devices);
256    scoped_array<cl_event> kernel_h2_event(num_devices);
257    scoped_array<cl_event> kernel_output_event(num_devices);
258
259    const double start_time = getCurrentTimestamp();
260
261    // Set kernel arguments.
262    // Hidden layer 1 kernel
263    unsigned argi = 0;
264
265    status = clSetKernelArg(kernel[KERNEL_H1_LAYER], argi++,
         sizeof(cl_mem), &z1_buf[0]); // output
266    checkError(status, "Failed to set argument %d", argi - 1)
         ;
267
268    status = clSetKernelArg(kernel[KERNEL_H1_LAYER], argi++,
         sizeof(cl_mem), &weights_buf[0]); //
269    checkError(status, "Failed to set argument %d", argi - 1)
         ;
270
271    argi++; // input argument set in the execution loop below
272
273    status = clSetKernelArg(kernel[KERNEL_H1_LAYER], argi++,
         sizeof(theta1_width), &theta1_width);
274    checkError(status, "Failed to set argument %d", argi - 1)
         ;
275
276    // Hidden layer 2 kernel
277    argi = 0;
278
279    status = clSetKernelArg(kernel[KERNEL_H2_LAYER], argi++,
         sizeof(cl_mem), &z2_buf[0]); // output
280    checkError(status, "Failed to set argument %d", argi - 1)
         ;
281
282    status = clSetKernelArg(kernel[KERNEL_H2_LAYER], argi++,
         sizeof(cl_mem), &weights_buf[1]); //
283    checkError(status, "Failed to set argument %d", argi - 1)
         ;
284
```

```
285 status = clSetKernelArg(kernel[KERNEL_H2_LAYER], argi++,
        sizeof(cl_mem), &z1_buf[0]); // input
286 checkError(status, "Failed to set argument %d", argi - 1)
        ;
287
288 status = clSetKernelArg(kernel[KERNEL_H2_LAYER], argi++,
        sizeof(theta2_width), &theta2_width);
289 checkError(status, "Failed to set argument %d", argi - 1)
        ;
290
291 // Output layer kernel
292 argi = 0;
293
294 argi++; // output argument set in the execution loop
        below;
295
296 status = clSetKernelArg(kernel[KERNEL_OUTPUT_LAYER], argi
        ++, sizeof(cl_mem), &weights_buf[2]); //
297 checkError(status, "Failed to set argument %d", argi - 1)
        ;
298
299 status = clSetKernelArg(kernel[KERNEL_OUTPUT_LAYER], argi
        ++, sizeof(cl_mem), &z2_buf[0]); // input
300 checkError(status, "Failed to set argument %d", argi - 1)
        ;
301
302 status = clSetKernelArg(kernel[KERNEL_OUTPUT_LAYER], argi
        ++, sizeof(theta3_width), &theta3_width);
303 checkError(status, "Failed to set argument %d", argi - 1)
        ;
304
305
306 // Enqueue kernel.
307 // Use a global work size corresponding to the size of
        the output matrix.
308 // Each work-item computes the result for one value of
        the output matrix,
309 // so the global work size has the same dimensions as the
         output matrix.
310 //
311 // Events are used to ensure that the kernel is not
        launched until
312 // the writes to the input buffers have completed.
313 const size_t global_work_size_h1[1] = {theta1_y}; // 100
314 const size_t local_work_size_h1[1]  = {theta1_y};
```

```
315
316    const size_t global_work_size_h2[1] = {theta2_y}; // 25
317    const size_t local_work_size_h2[1]  = {theta2_y};
318
319    const size_t global_work_size_output[1] = {theta3_y}; //
         10
320    const size_t local_work_size_output[1]  = {theta3_y};
321
322    printf("Launching for device %d (global sizes:\nh1: %zd)\
         nh2: %zd)\noutput: %zd)\n",
323            0, global_work_size_h1[0], global_work_size_h2
                [0], global_work_size_output[0]);
324
325    for(unsigned i = 0; i < NUM_IMAGES; i++) {
326      // Set remaining arguments
327      status = clSetKernelArg(kernel[KERNEL_H1_LAYER], 2,
           sizeof(cl_mem), &input_x_buf[i]); // input
328      checkError(status, "Failed to set argument %d", 2);
329
330      status = clSetKernelArg(kernel[KERNEL_OUTPUT_LAYER], 0,
            sizeof(cl_mem), &output_buf[i]); // output
331      checkError(status, "Failed to set argument %d", 0);
332
333      /****************** Hidden layer 1 ********************
           */
334      status = clEnqueueNDRangeKernel(queue[0], kernel[
           KERNEL_H1_LAYER], 1, NULL,
335            global_work_size_h1, local_work_size_h1, 0, NULL, &
               kernel_h1_event[0]);
336      checkError(status, "Failed to launch kernel");
337
338      /****************** Hidden layer 2 ********************
           */
339      status = clEnqueueNDRangeKernel(queue[0], kernel[
           KERNEL_H2_LAYER], 1, NULL,
340            global_work_size_h2, local_work_size_h2, 0, NULL, &
               kernel_h2_event[0]);
341      checkError(status, "Failed to launch kernel");
342
343
344      /****************** Output layer ********************
           */
345      status = clEnqueueNDRangeKernel(queue[0], kernel[
           KERNEL_OUTPUT_LAYER], 1, NULL,
346            global_work_size_output, local_work_size_output, 0,
```

```
                NULL, &kernel_output_event[0]);
347     checkError(status, "Failed to launch kernel");
348 
349   }
350 
351   // Wait for all kernels to finish.
352   clWaitForEvents(num_devices, kernel_output_event);
353 
354   cl_ulong time_ns_h1 = getStartEndTime(kernel_h1_event[0])
        ;
355   //printf("Hidden layer kernel time (device %d): %0.3f ms\
        n", 0, double(time_ns_h1) * 1e-6);
356   tot_kernel_time_ns_h1 += time_ns_h1;
357   cl_ulong time_ns_h2 = getStartEndTime(kernel_h2_event[0])
        ;
358   //printf("Hidden layer kernel time (device %d): %0.3f ms\
        n", 0, double(time_ns_h2) * 1e-6);
359   tot_kernel_time_ns_h2 += time_ns_h2;
360   cl_ulong time_ns_output = getStartEndTime(
        kernel_output_event[0]);
361   //printf("Output layer kernel time (device %d): %0.3f ms\
        n", 0, double(time_ns_output) * 1e-6);
362   tot_kernel_time_ns_output += time_ns_output;
363 
364   const double end_time = getCurrentTimestamp();
365   const double total_time = end_time - start_time;
366   cl_ulong tot_kernel_time_ns = tot_kernel_time_ns_h1 +
        tot_kernel_time_ns_h2 + tot_kernel_time_ns_output;
367 
368   // Wall-clock time taken.
369   printf("\nTime: %0.3f ms\n", total_time * 1e3);
370 
371   // Write to timing log file
372   log_file << "Total time: " << std::setprecision(3) <<
        total_time * 1e3 << "ms\n";
373   log_file << "\nTotal kernel time: " << std::setprecision
        (3) <<  tot_kernel_time_ns*NUM_IMAGES * 1e-6 << "ms\n"
        ;
374   log_file << "\nKernel time: " << std::setprecision(3) <<
         tot_kernel_time_ns * 1e-6 << "ms\n";
375   log_file << "\nHidden layer 1 kernel time: " << std::
        setprecision(3) <<  tot_kernel_time_ns_h1 * 1e-6 << "
        ms\n";
376   log_file << "\nHidden layer 2 kernel time: " << std::
        setprecision(3) <<  tot_kernel_time_ns_h2 * 1e-6 << "
```

```
      ms\n";
377   log_file << "\nOutput layer kernel time: " << std::
        setprecision(3) <<  tot_kernel_time_ns_output * 1e-6
        << "ms\n";
378   log_file.close();
379
380   // Compute the throughput (GFLOPS).
381   // There are Y_width * Y_height output values, with each
        value
382   // computed using W_width multiplies and adds.
383   const float flops = (float)(2.0f * Y_width * Y_height *
        X_height * NUM_IMAGES/ total_time); // not correct
384   printf("\nThroughput: %0.2f GFLOPS\n\n", flops * 1e-9);
385
386   // Release kernel events.
387   clReleaseEvent(kernel_h1_event[0]);
388   clReleaseEvent(kernel_h2_event[0]);
389   clReleaseEvent(kernel_output_event[0]);
390
391   // Verify results.
392   verify();
393
394   file.close();
395   if (!file.is_open())
396     printf("Successfully closed file\n");
397 }
398
399 void verify() {
400   printf("Verifying\n");
401
402   for (unsigned i = 0; i < NUM_IMAGES; i++) {
403     int max_idx = 0;
404     // Read the result.
405     cl_int status = clEnqueueReadBuffer(queue[0],
          output_buf[i], CL_TRUE,
406           0, Y_height * Y_width * sizeof(float), output, 0,
              NULL, NULL);
407     checkError(status, "Failed to read output matrix");
408
409     for (int j=1; j < Y_y; j++){
410         float tmp;
411         if (output[max_idx] < output[j])
412           max_idx = j;
413     }
414
```

```
415     if (max_idx == TEST_NUM-1){
416       correct_count++;
417       //printf("The linear classifier reads CORRECTLY a %d\
             n", TEST_NUM);
418     } else if (max_idx == 9) { // idx 9 is 0 here
419       correct_count++;
420       //printf("The linear classifier reads CORRECTLY a %d\
             n", TEST_NUM);
421     } else {
422       printf("The linear classifier reads it WRONG, it read
              a %d\n", max_idx+1);
423     }
424   }
425 }
426 
427 // Free the resources allocated during initialization
428 void cleanup() {
429   for(unsigned i = 0; i < NUM_KERNELS; ++i) {
430     if(kernel && kernel[i]) {
431       clReleaseKernel(kernel[i]);
432     }
433   }
434   if(queue && queue[0]) {
435     clReleaseCommandQueue(queue[0]);
436   }
437   if(weights_buf[KERNEL_H1_LAYER]) {
438     clReleaseMemObject(weights_buf[KERNEL_H1_LAYER]);
439   }
440   if(weights_buf[KERNEL_H2_LAYER]) {
441     clReleaseMemObject(weights_buf[KERNEL_H2_LAYER]);
442   }
443   if(weights_buf[KERNEL_OUTPUT_LAYER]) {
444     clReleaseMemObject(weights_buf[KERNEL_OUTPUT_LAYER]);
445   }
446   if(z1_buf[0]) {
447     clReleaseMemObject(z1_buf[0]);
448   }
449   if(z2_buf[0]) {
450     clReleaseMemObject(z2_buf[0]);
451   }
452   for (unsigned i = 0; i < NUM_IMAGES; i++) {
453     if(input_x_buf[i]) {
454       clReleaseMemObject(input_x_buf[i]);
455     }
456     if(output_buf) {
```

```
457         clReleaseMemObject(output_buf[i]);
458       }
459     }
460 
461     if(program) {
462       clReleaseProgram(program);
463     }
464     if(context) {
465       clReleaseContext(context);
466     }
467 
468     for (int i = 0; i < NUM_IMAGES; i++) {
469       alignedFree(input_x[i]);
470     }
471 }
```

Listing 7: main.cpp

A.1.4 CNN

```
1  #ifndef CNN_H
2  #define CNN_H
3  
4  #define INPUT_W      40
5  #define INPUT_H      90
6  #define INPUT_SIZE   INPUT_W*INPUT_H
7  
8  #define OUTPUT_SIZE 2
9  
10 // Layer output sizes, uncomment to test other layers
11 //#define OUTPUT_SIZE 6800 // 16*25*17
12 //#define OUTPUT_SIZE 1344 // 32*6*7
13 //#define OUTPUT_SIZE 1280 // 64*4*5
14 //#define OUTPUT_SIZE 32
15 //#define OUTPUT_SIZE 128
16 //#define OUTPUT_SIZE 2
17 
18 #endif // CNN_H
```

Listing 8: cnn.h

```
1  
2  const unsigned int num_layers = 6; // mby use #define
3  
4  unsigned int layer_config[][15] = {
5                { // conv1
```

```
6                 0, // "0" -> conv, "1" -> fc (only conv,
                     conv with pool, fc, ??)
7                 1, // type_num for conv1 -> 1 for conv2 ->
                     2
8                 90, 40, // h, w input data
9                 16, 1, 16, 8, //16 // n, c, h, w, b -
                     weights and bias
10                25, 17, 3, 2, // h, w, s_h, s_w - Conv
                     parameters
11                1, // relu_on
12                0, // read from data "0"-> input_buf  "1"->
                      output_buf  "2"->"tmp_1_buffer"  "3"->"
                     tmp_2_buffer"
13                2 // output buffer   "0"-> input_buf  "1"->
                      output_buf  "2"->"tmp_1_buffer"  "3"->"
                     tmp_2_buffer"
14              },
15              { // conv2
16                0,
17                2,
18                25, 17,
19                32, 16, 8, 4,
20                6, 7, 3, 2,
21                1,
22                2,
23                3
24              },
25              { // conv3
26                0,
27                3,
28                6, 7,
29                64, 32, 3, 3,
30                4, 5, 1, 1,
31                1,
32                3,
33                2
34              },
35              { // fc1/ip1
36                1,
37                1,
38                1280, 1,
39                1, 1, 32, 1280, // 64*4*5 = 1280
40                1, 32, 1, 1,
41                0, // relu should not be on in this layer
42                2,
```

```
43                3,
44              },
45              { // fc2/ip2
46                1,
47                2,
48                32, 1,
49                1, 1, 128, 32,
50                1, 128, 1, 1,
51                1,
52                3,
53                2
54              },
55              { //fc3/ip3
56                1,
57                3,
58                128, 1,
59                1, 1, 2, 128,
60                1, 2, 1, 1,
61                0,
62                2,
63                1
64              }
65 };
```

Listing 9: layer_config.h

```
1  #include "../host/inc/cnn.h"
2
3  #ifndef SIMD_WORK_ITEMS
4  #define SIMD_WORK_ITEMS 2
5  #endif
6
7  #ifndef NUM_COMPUTE_UNITS
8  #define NUM_COMPUTE_UNITS 4
9  #endif
10
11 #define SIGMOID(x) (1.0f / (1 + exp(-x)))
12 #define RELU(x) (x > 0 ? x : 0)
13
14 __kernel
15 __attribute__((task))
16 void taskconv(
17       // Params Ports
18       unsigned in_h,
19       unsigned in_w,
20       unsigned out_c,
```

```
21        unsigned in_c,
22        unsigned K_h,
23        unsigned K_w,
24        unsigned out_h,
25        unsigned out_w,
26        unsigned S_h,
27        unsigned S_w,
28        unsigned relu_on,
29
30        // Data Ports
31        __global float *restrict output, // top
32        __global float *restrict input, //bottom,
33        __global float *restrict weights,
34        __global float *restrict bias
35        )
36  {
37    unsigned M = out_c;
38    unsigned N = in_c;
39
40    unsigned filter_size_2d = K_h * K_w;
41    unsigned filter_size_3d = filter_size_2d*N;
42
43    unsigned ifm_size = in_h*in_w;
44    unsigned ofm_size = out_h*out_w; // input argument?
45
46    float running_sum = 0.0f;
47
48    printf("IN_H: %d, IN_W: %d, R: %d, C: %d, M: %d, N: %d,
         K_h: %d, K_w: %d, S_h: %d, S_w: %d, ifm_size: %d \n",
         in_h, in_w, out_h, out_w, M, N, K_h, K_w, S_h, S_w,
         ifm_size);
49
50    for (unsigned row = 0; row < out_h; row++) { // output
         rows
51      for (unsigned col = 0; col < out_w; col++) { // output
           cols
52        for (unsigned to = 0; to < M; to++) { // M output
             feature maps
53          for (unsigned ti = 0; ti < N; ti++) { // N input
               feature maps
54            for (unsigned i = 0; i < K_h; i++) { // filter
                 height
55              for (unsigned j = 0; j < K_w; j++) { // filter
                   width
56                running_sum += weights[to*filter_size_3d + ti
```

```
                *filter_size_2d + i*K_w + j] * input[ti*
                ifm_size + (S_h*row + i)*in_w + (S_w*col +
                 j)];
57            } // j
58          } // i
59        } // ti
60        running_sum += bias[to];
61        if (relu_on)
62          output[to*ofm_size + row*out_w + col] = RELU(
              running_sum);
63        else
64          output[to*ofm_size + row*out_w + col] =
              running_sum;
65        running_sum = 0.0f;
66      } // to
67    } // col
68  } // row
69 }
70
71 __kernel
72 //__attribute((reqd_work_group_size(100,1,1))) // May need
     to compile several times for different work group sizez
73 //__attribute((num_simd_work_items(SIMD_WORK_ITEMS))) //
     This is not supported when using channels
74 __attribute((num_compute_units(NUM_COMPUTE_UNITS)))
75 void ndrconv(// Params Ports
76          unsigned in_h,
77          unsigned in_w,
78          unsigned out_c,
79          unsigned in_c,
80          unsigned K_h,
81          unsigned K_w,
82          unsigned out_h,
83          unsigned out_w,
84          unsigned S_h,
85          unsigned S_w,
86          unsigned relu_on,
87
88          // Data Ports
89          __global float *restrict output, // top
90          __global float *restrict input, //bottom,
91          __global float *restrict weights,
92          __global float *restrict bias
93          )
94 {
```

```
95      // Local ID index (offset within a block)
96      int local_id_x = get_local_id(0);
97      int local_id_y = get_local_id(1);
98      int local_id_z = get_local_id(2);
99
100     // Global ID index (offset within the NDRange)
101     int global_id_x = get_global_id(0);
102     int global_id_y = get_global_id(1);
103     int global_id_z = get_global_id(2);
104
105     unsigned weight_size_2d = K_h*K_w;
106     unsigned weight_size_3d = weight_size_2d*in_c;
107     unsigned ifm_size = in_h*in_w;
108     unsigned ofm_size = out_h*out_w;
109
110     unsigned ifm_idx = 0;
111     unsigned row_idx = 0;
112     unsigned cnt_ifm = 1;
113     unsigned cnt_row = 1;
114
115     float running_sum = 0.0f;
116
117     for (int k = 0; k < weight_size_3d; ++k) {
118         running_sum += weights[weight_size_3d*global_id_z +
                k] * input[ifm_size*ifm_idx + (S_h*local_id_y +
                row_idx)*in_w + (S_w*local_id_x + cnt_row-1)];
119
120         if (cnt_ifm == weight_size_2d) {
121           row_idx = 0;
122           ifm_idx++;
123           cnt_ifm = 1;
124           cnt_row = 1;
125         }
126         else if (cnt_row == K_w) {
127           row_idx++;
128           cnt_row = 1;
129           cnt_ifm++;
130         }
131         else {
132           cnt_row++;
133           cnt_ifm++;
134         }
135     }
136
137     running_sum += bias[global_id_z];
```

```
138
139     // Store result in output
140     if (relu_on)
141       output[ofm_size*global_id_z + local_id_y*out_w +
            local_id_x] = RELU(running_sum);
142     else
143       output[ofm_size*global_id_z + local_id_y*out_w +
            local_id_x] = running_sum;
144 }
145
146 __kernel
147 //__attribute((reqd_work_group_size(100,1,1))) // May need
       to compile several times for different work group sizez
148 //__attribute((num_simd_work_items(SIMD_WORK_ITEMS))) //
       This is not supported when using channels
149 void fully_connected( // Input and output matrices
150                 // Weight weight
151                 unsigned W_width, // mby save locally
152                 unsigned relu_on,
153
154                 // Data ports
155                  __global float *restrict output,
156                  __global float *restrict input,
157                  __global float *restrict weights,
158                  __global float *restrict bias
159                )
160 {
161     // Local storage for a block of input matrices W and X
162     //__local float W_local[W_y][W_x];  //[W_y][W_x];
163     //__local float X_local[X_y];
164
165     // Block index
166     int block_id = get_group_id(0);
167     //int block_y = get_group_id(1);
168
169     // Local ID index (offset within a block)
170     int local_id = get_local_id(0);
171     int global_id = get_global_id(0);
172     //printf("LID: %d GID: %d\n", local_id, global_id);
173     //int local_y = get_local_id(1);
174
175     float running_sum = 0.0f;
176
177     //running_sum += weights[local_id*W_width];
178
```

```
179     //#pragma unroll
180     for (int k = 0; k < W_width; ++k) // fix bias -> add
           bias
181     {
182         //float t1_k = Theta1[local_id][k]; // W[local_id*
               W_x + k];
183         //float x_k = X[k];
184         running_sum += weights[local_id*W_width + k] *
               input[k];
185         //printf("LID: %d running_sum = %f\n", local_id,
               running_sum);
186     }
187 
188     running_sum += bias[local_id];
189 
190     barrier(CLK_LOCAL_MEM_FENCE);
191     // Store result in matrix C
192     if (relu_on)
193       output[get_global_id(0)] = RELU(running_sum);
194     else
195       output[get_global_id(0)] = running_sum;
196 }
```

Listing 10: cnn.cl

```
1  #include <string.h>
2  #include <sys/stat.h>
3  #include <iostream>
4  #include <fstream>
5  #include <iomanip>
6  #include <math.h>
7  #include "CL/opencl.h"
8  #include "AOCLUtils/aocl_utils.h"
9  #include "wav_utils/record_voice.h"
10 #include "wav_utils/write_wav.h"
11 #include "cnn.h"
12 #include "cnpy.h"
13 
14 #include "layer_config.h"
15 
16 using namespace aocl_utils;
17 
18 #define NUM_TESTS   500
19 #define LAYER_NUM   6
20 #define MAX_LAYER_NUM   16
21 #define TMP_BUF_SIZE 6800 // should at least be the maximum
```

```
    buffer size needed
22
23 // Configuration file instructions
24 enum config_item{
25 layer_type, // "0" -> conv, "1" -> fc
26
27 type_num, // for conv1 -> 1 for conv2 -> 2 etc.
28
29 data_h, data_w,
30
31 weight_n, weight_c, weight_h, weight_w,
32
33 conv_h, conv_w, conv_stride_h, conv_stride_w, //Conv
    Parameters
34
35 relu_on,
36
37 memrd_src, //"0"-> input_buf  "1"-> output_buf  "2"->"
    tmp_1_buffer"  "3"->"tmp_2_buffer"
38
39 memwr_dst//"0"-> input_buf  "1"-> output_buf  "2"->"
    tmp_1_buffer"  "3"->"tmp_2_buffer"
40
41 };
42
43 // Define the kernel names used
44 const char *knl_name_mem_rd = "mem_read";
45 const char *knl_name_conv  = "ndrconv"; // "taskconv"
46 const char *knl_name_pool  = "max_pool";
47 const char *knl_name_mem_wr = "mem_write";
48 const char *knl_name_fc = "fully_connected";
49
50 // OpenCL runtime configuration
51 unsigned num_devices = 0;
52 cl_platform_id platform = NULL;
53 cl_context context = NULL;
54 cl_program program = NULL;
55 scoped_array<cl_device_id> device; // num_devices elements
56
57 scoped_array<cl_kernel> knl_conv;
58 scoped_array<cl_kernel> knl_fc;
59
60 scoped_array<cl_command_queue> queue_conv;
61 scoped_array<cl_command_queue> queue_fc;
62
```

```
63  scoped_array<cl_mem> input_buf; // or data_buf?
64  scoped_array<cl_mem> output_buf;
65  scoped_array<cl_mem> weights_buf;
66  scoped_array<cl_mem> bias_buf;
67  scoped_array<cl_mem> tmp_1_buf;
68  scoped_array<cl_mem> tmp_2_buf;
69
70  float* input;
71  float* weights[num_layers];
72  float* bias[num_layers];
73  float* output;
74  float* golden_ref;
75
76  int weight_sizes[num_layers];
77  int bias_sizes[num_layers];
78
79  // Tot time
80  cl_ulong tot_kernel_time = 0;
81  double start_time;
82  double end_time;
83  double total_time;
84
85  // Weight files info
86  const char* weights_path = "../fpga/weights/";
87
88  // Input file name
89  const char* input_file_name = "../input/test_input_1"; //
        voice_rec_live";
90
91  // Open time log file
92  std::ofstream log_file("logs/time_log_cu4.log");
93
94  // Function prototypes
95  void read_new_input();
96  void prepare();
97  bool init_opencl();
98  void run();
99  void verify();
100 void cleanup();
101
102 int main(void) {
103   // Prepare data
104   prepare();
105
106   // Initialize OpenCL.
```

```
107   if (!init_opencl()) {
108     return -1;
109   }
110
111   // Run the kernel
112   run();
113
114   // Free the resources allocated
115   cleanup();
116
117 }
118
119 /////////// HELPER FUNCTIONS //////////
120
121 // Read/record new input
122 void read_new_input() {
123   // record new input
124   int num_bytes = NUM_SECONDS * SAMPLE_RATE * NUM_CHANNELS
        * sizeof(float);
125   float* recorded_samples = (float*)malloc(num_bytes);
126   record_voice(recorded_samples);
127   printf("num_bytes = %d\n", num_bytes);
128
129   // Set start time before preprocessing, after voice done
        recording
130   start_time = getCurrentTimestamp();
131
132   // Write the recorded samples to a wave file
133   write_wav("../input/voice_rec_live.wav", recorded_samples
        , num_bytes, NUM_CHANNELS, SAMPLE_RATE, 32);
134
135   // Execute preprocess program
136   system("./../input/mfcc_preprocess --input ../input/
        voice_rec_live.wav --output ../input/voice_rec_live");
137
138   // Read mfcc preprocessed input file
139   std::ifstream input_file(input_file_name, std::ios::in |
        std::ios::binary);
140   if (input_file.is_open()){
141     input_file.read((char*)(input),  sizeof(float)*
          INPUT_SIZE); // sizeof() should return IMG_SIZE*4
          bytes
142     input_file.close();
143   }
144   else{
```

```
145     printf("Could not read input_file\n");
146     input_file.close();
147   }
148 
149   free(recorded_samples);
150 }
151 
152 // Prepare weights and biases
153 void prepare(){
154   // could mby have these as only one buffer
155   char conv_filename[sizeof(weights_path)+11]; // should do
        this another way
156   char conv_bias_filename[sizeof(weights_path)+11];
157   char fc_filename[sizeof(weights_path)+9];
158   char fc_bias_filename[sizeof(weights_path)+9];
159 
160   for (int i = 0; i < num_layers; i++) {
161     // NpyArray objects
162     cnpy::NpyArray weight_npy;
163     cnpy::NpyArray bias_npy;
164 
165     // Load weight or bias into NpyArray
166     if (layer_config[i][layer_type] == 0) { // conv layer
167       sprintf(conv_filename, "../fpga/weights/W_conv%d.npy"
           , layer_config[i][type_num]);
168       sprintf(conv_bias_filename, "../fpga/weights/b_conv%d
           .npy", layer_config[i][type_num]);
169       weight_npy = cnpy::npy_load(conv_filename);
170       bias_npy = cnpy::npy_load(conv_bias_filename);
171     } else if (layer_config[i][layer_type] == 1) { // fc
         layer
172       sprintf(fc_filename, "../fpga/weights/W_ip%d.npy",
           layer_config[i][type_num]);
173       sprintf(fc_bias_filename, "../fpga/weights/b_ip%d.npy
           ", layer_config[i][type_num]);
174       weight_npy = cnpy::npy_load(fc_filename);
175       bias_npy = cnpy::npy_load(fc_bias_filename);
176     }
177 
178     // Store sizes for later use
179     bias_sizes[i] = bias_npy.shape[0];
180     weight_sizes[i] = 1;
181     for (int j = 0; j < weight_npy.shape.size(); j++)
182       weight_sizes[i] *= weight_npy.shape[j];
183 
```

```
184       // 64 byte aligned malloc for DMA
185       weights[i] = (float*)alignedMalloc(sizeof(float)*
              weight_sizes[i]);
186       bias[i] = (float*)alignedMalloc(sizeof(float)*
              bias_sizes[i]);
187
188       memcpy(weights[i], weight_npy.data, sizeof(float)*
              weight_sizes[i]);
189       memcpy(bias[i], bias_npy.data, sizeof(float)*bias_sizes
              [i]);
190
191       // Destroy the NpyArray objects
192       weight_npy.destruct();
193       bias_npy.destruct();
194     }
195
196     //cnpy::NpyArray input_npy = cnpy::npy_load("../fpga/
            per_layer/unit_tests/input_1_data.npy"); // Test input
197     cnpy::NpyArray golden_ref_npy = cnpy::npy_load("../fpga/
            per_layer/unit_tests/input_1_ip3.npy");
198
199     // Allocate aligned memory for DMA transfer
200     input       = (float*)alignedMalloc(sizeof(float)*
            INPUT_SIZE); // input
201     output      = (float*)alignedMalloc(sizeof(float)*
            OUTPUT_SIZE); // output_c1
202     golden_ref  = (float*)alignedMalloc(sizeof(float)*
            OUTPUT_SIZE); // golden_ref
203
204     //memcpy(input, input_npy.data, sizeof(float)*INPUT_SIZE)
            ; // Test input
205     memcpy(golden_ref, golden_ref_npy.data, sizeof(float)*
            OUTPUT_SIZE);
206
207     // Destruct
208     //input_npy.destruct(); // Test input
209     golden_ref_npy.destruct();
210   }
211
212   // Initializes the OpenCL objects
213   bool init_opencl() {
214
215     cl_int status;
216
217     printf("Initializing OpenCL\n");
```

```
218
219   if (!setCwdToExeDir()) {
220     return false;
221   }
222
223   // Get the OpenCL platform
224   platform = findPlatform("Altera");
225   if (platform == NULL) {
226     printf("ERROR: Unable to find Altera OpenCL platform.\n
        ");
227   }
228
229   // Query the available OpenCL device.
230   device.reset(getDevices(platform, CL_DEVICE_TYPE_ALL, &
        num_devices));
231   printf("Platform: %s\n", getPlatformName(platform).c_str
        ());
232   printf("Using %d device(s)\n", num_devices);
233   for (unsigned i = 0; i < num_devices; ++i) {
234     printf("  %s\n", getDeviceName(device[i]).c_str());
235   }
236
237   // Create the context
238   context = clCreateContext(NULL, num_devices, device, &
        oclContextCallback, NULL, &status);
239   checkError(status, "Failed to create context");
240   // Create the program for all device. Use the first
        device as the
241   // representative device (assuming all device are of the
        same type)
242   std::string binary_file = getBoardBinaryFile("
        cnn_ndrconv_cu4", device[0]); // "cnn_ndrconv_cu4"
243   printf("Using AOCX: %s\n", binary_file.c_str());
244   program = createProgramFromBinary(context, binary_file.
        c_str(), device , num_devices);
245
246   // Build the program that was just created
247   status = clBuildProgram(program, 0, NULL, "", NULL, NULL)
        ;
248   checkError(status, "Failed to build program");
249
250   // Create per-device objects
251   queue_conv.reset(num_devices);
252   queue_fc.reset(num_devices);
253   knl_conv.reset(num_devices);
```

```
254   knl_fc.reset(num_devices);
255   weights_buf.reset(num_devices*num_layers);
256   bias_buf.reset(num_devices*num_layers);
257   input_buf.reset(num_devices);
258   output_buf.reset(num_devices);
259   tmp_1_buf.reset(num_devices);
260   tmp_2_buf.reset(num_devices);
261
262   // Command queue
263   queue_conv[0] = clCreateCommandQueue(context, device[0],
          CL_QUEUE_PROFILING_ENABLE, &status);
264   checkError(status, "Failed to create command queue conv")
          ;
265   queue_fc[0] = clCreateCommandQueue(context, device[0],
          CL_QUEUE_PROFILING_ENABLE, &status);
266   checkError(status, "Failed to create command queue fc");
267
268   knl_conv[0] = clCreateKernel(program, knl_name_conv, &
          status);
269   checkError(status, "Failed to create conv kernel");
270
271   knl_fc[0] = clCreateKernel(program, knl_name_fc, &status)
          ;
272   checkError(status, "Failed to create fc kernel");
273
274   // Buffers.
275   for (int i = 0; i < num_layers; i++){
276     weights_buf[i] = clCreateBuffer(context,
            CL_MEM_READ_ONLY | CL_MEM_BANK_1_ALTERA,
277         weight_sizes[i] * sizeof(float), NULL, &status);
278     checkError(status, "Failed to create buffer for weights
            layer = %d", i);
279
280     bias_buf[i] = clCreateBuffer(context, CL_MEM_READ_ONLY
            | CL_MEM_BANK_1_ALTERA,
281         bias_sizes[i] * sizeof(float), NULL, &status);
282     checkError(status, "Failed to create buffer for bias");
283   }
284
285   input_buf[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
          CL_MEM_BANK_2_ALTERA,
286       INPUT_SIZE * sizeof(float), NULL, &status);
287   checkError(status, "Failed to create buffer for input");
288
289   output_buf[0] = clCreateBuffer(context, CL_MEM_WRITE_ONLY
```

```
        | CL_MEM_BANK_1_ALTERA,
290       OUTPUT_SIZE * sizeof(float), NULL, &status);
291   checkError(status, "Failed to create buffer for output");
292
293   // Allocate fc buffers
294   tmp_1_buf[0] = clCreateBuffer(context,  CL_MEM_READ_WRITE
        ,
295     TMP_BUF_SIZE * sizeof(float), NULL, &status);
296   checkError(status, "Failed to create buffer tmp 1");
297
298   tmp_2_buf[0] = clCreateBuffer(context,  CL_MEM_READ_WRITE
        ,
299     TMP_BUF_SIZE * sizeof(float), NULL, &status);
300   checkError(status, "Failed to create buffer tmp 2");
301
302   return true;
303 }
304
305 void run() {
306   printf("Running\n");
307
308   cl_int status;
309
310   // Launch kernels.
311   // This is the portion of time that we'll be measuring
        for throughput
312   // benchmarking.
313   scoped_array<cl_event> conv_event(num_devices);
314   scoped_array<cl_event> fc_event(num_devices);
315
316   // Set kernel arguments.
317   // Hidden layer 1 kernel
318   while(1) {
319     tot_kernel_time = 0; // reset tot kernel time
320     read_new_input();
321     for (int i = 0; i < num_layers; i++) {
322       if (layer_config[i][layer_type] == 0) { // conv layer
323         int argi;
324         for (argi = data_h; argi < relu_on+1; argi++) {
325           status = clSetKernelArg(knl_conv[0], argi-2,
                sizeof(unsigned),  &layer_config[i][argi]);//
                &out_h);
326           checkError(status, "Failed to set argument %d",
                argi - 2);
327         }
```

```
328
329         argi = argi - 2;
330         // Data ports
331
332         // Output port
333         if (layer_config[i][memwr_dst] == 1) {
334           status = clSetKernelArg(knl_conv[0], argi++,
                  sizeof(cl_mem), &output_buf[0]);
335           checkError(status, "Failed to set argument %d",
                  argi - 1);
336         }
337         else if (layer_config[i][memwr_dst] == 2) {
338           status = clSetKernelArg(knl_conv[0], argi++,
                  sizeof(cl_mem), &tmp_1_buf[0]);
339           checkError(status, "Failed to set argument %d",
                  argi - 1);
340         }
341         else if (layer_config[i][memwr_dst] == 3) { // else
                ?
342           status = clSetKernelArg(knl_conv[0], argi++,
                  sizeof(cl_mem), &tmp_2_buf[0]);
343           checkError(status, "Failed to set argument %d",
                  argi - 1);
344         }
345
346         // Input port
347         if (layer_config[i][memrd_src] == 0) {
348           status = clSetKernelArg(knl_conv[0], argi++,
                  sizeof(cl_mem), &input_buf[0]);
349           checkError(status, "Failed to set argument %d",
                  argi - 1);
350         }
351         else if (layer_config[i][memrd_src] == 2) {
352           status = clSetKernelArg(knl_conv[0], argi++,
                  sizeof(cl_mem), &tmp_1_buf[0]);
353           checkError(status, "Failed to set argument %d",
                  argi - 1);
354         }
355         else if (layer_config[i][memrd_src] == 3) { // else
                ?
356           status = clSetKernelArg(knl_conv[0], argi++,
                  sizeof(cl_mem), &tmp_2_buf[0]);
357           checkError(status, "Failed to set argument %d",
                  argi - 1);
358         }
```

```
359
360
361         status = clSetKernelArg(knl_conv[0], argi++, sizeof
                (cl_mem), &weights_buf[i]);
362         checkError(status, "Failed to set argument %d",
                argi - 1);
363
364         status = clSetKernelArg(knl_conv[0], argi++, sizeof
                (cl_mem), &bias_buf[i]);
365         checkError(status, "Failed to set argument %d",
                argi - 1);
366
367       } else if (layer_config[i][layer_type] == 1) { // fc
            layer
368         int argi = 0;
369
370         status = clSetKernelArg(knl_fc[0], argi++, sizeof(
                unsigned), &layer_config[i][weight_w]);
371         checkError(status, "Failed to set argument %d",
                argi - 1);
372
373         status = clSetKernelArg(knl_fc[0], argi++, sizeof(
                unsigned), &layer_config[i][relu_on]);
374         checkError(status, "Failed to set argument %d",
                argi - 1);
375
376         // Data ports
377
378         // Output port
379         if (layer_config[i][memwr_dst] == 1) {
380           status = clSetKernelArg(knl_fc[0], argi++, sizeof
                  (cl_mem), &output_buf[0]);
381           checkError(status, "Failed to set argument %d",
                  argi - 1);
382         }
383         else if (layer_config[i][memwr_dst] == 2) {
384           status = clSetKernelArg(knl_fc[0], argi++, sizeof
                  (cl_mem), &tmp_1_buf[0]);
385           checkError(status, "Failed to set argument %d",
                  argi - 1);
386         }
387         else if (layer_config[i][memwr_dst] == 3) {  //
                else?
388           status = clSetKernelArg(knl_fc[0], argi++, sizeof
                  (cl_mem), &tmp_2_buf[0]);
```

```
389           checkError(status, "Failed to set argument %d",
                  argi - 1);
390         }
391 
392         // Input port
393         if (layer_config[i][memrd_src] == 0) {
394           status = clSetKernelArg(knl_fc[0], argi++, sizeof
                  (cl_mem), &input_buf[0]);
395           checkError(status, "Failed to set argument %d",
                  argi - 1);
396         }
397         else if (layer_config[i][memrd_src] == 2) {
398           status = clSetKernelArg(knl_fc[0], argi++, sizeof
                  (cl_mem), &tmp_1_buf[0]);
399           checkError(status, "Failed to set argument %d",
                  argi - 1);
400         }
401         else if (layer_config[i][memrd_src] == 3) { // else
               ?
402           status = clSetKernelArg(knl_fc[0], argi++, sizeof
                  (cl_mem), &tmp_2_buf[0]);
403           checkError(status, "Failed to set argument %d",
                  argi - 1);
404         }
405 
406         status = clSetKernelArg(knl_fc[0], argi++, sizeof(
               cl_mem), &weights_buf[i]);
407         checkError(status, "Failed to set argument %d",
               argi - 1);
408 
409         status = clSetKernelArg(knl_fc[0], argi++, sizeof(
               cl_mem), &bias_buf[i]);
410         checkError(status, "Failed to set argument %d",
               argi - 1);
411       }
412 
413       // Enqueue kernel.
414       // Use a global work size corresponding to the size
             of the output matrix.
415       // Each work-item computes the result for one value
             of the output matrix,
416       // so the global work size has the same dimensions as
              the output matrix.
417       //
418       // Events are used to ensure that the kernel is not
```

```
           launched until
419      // the writes to the input buffers have completed.
420      const size_t global_work_size_conv[3] = {layer_config
            [i][conv_w], layer_config[i][conv_h], layer_config
            [i][weight_n]};
421      const size_t local_work_size_conv[3]  = {layer_config
            [i][conv_w], layer_config[i][conv_h], 1};
422
423      const size_t global_work_size_fc[1] = {layer_config[i
            ][weight_h]};
424      const size_t local_work_size_fc[1]  = {layer_config[i
            ][weight_h]};
425
426      if (i == 0) { // first layer
427        status = clEnqueueWriteBuffer(queue_conv[0],
              input_buf[0], CL_FALSE,
428          0, INPUT_SIZE * sizeof(float), input, 0, NULL,
               NULL);
429        checkError(status, "Failed to transfer input");
430      }
431
432      if (layer_config[i][layer_type] == 0) { // conv layer
433        status = clEnqueueWriteBuffer(queue_conv[0],
              weights_buf[i], CL_FALSE,
434          0, weight_sizes[i] * sizeof(float), weights[i],
               0, NULL, NULL);
435        checkError(status, "Failed to transfer weights");
436
437        status = clEnqueueWriteBuffer(queue_conv[0],
              bias_buf[i], CL_FALSE,
438          0, bias_sizes[i] * sizeof(float), bias[i], 0,
               NULL, NULL);
439        checkError(status, "Failed to transfer bias");
440
441        if (clFinish(queue_conv[0]) == CL_SUCCESS)
442          printf("cl_finish == success\n");
443
444        //// kernel taskconv
445        //status = clEnqueueTask(queue_conv[0], knl_conv
              [0], 0, NULL, &conv_event[0]);
446        //checkError(status, "Failed to launch simple_conv
              kernel");
447
448        // kernel ndrconv
449        status = clEnqueueNDRangeKernel(queue_conv[0],
```

```
        knl_conv[0], 3, NULL,
450     global_work_size_conv, local_work_size_conv, 0,
        NULL, &conv_event[0]);
451     checkError(status, "Failed to launch kernel");
452
453     // Wait for all kernels to finish.
454     clWaitForEvents(num_devices, conv_event);
455
456     // Get kernel times using the OpenCL event
        profiling API.
457     cl_ulong conv_time_ns = getStartEndTime(conv_event
        [0]);
458     printf("Conv layer kernel time (device %d): %0.3f
        ms\n", 0, double(conv_time_ns) * 1e-6);
459     log_file << "Conv layer " << i << " kernel time: "
        << std::setprecision(3) <<  conv_time_ns * 1e-6
        << "ms\n";
460
461     tot_kernel_time += conv_time_ns;
462   }
463   else if (layer_config[i][layer_type] == 1) { // fc
      layer
464     status = clEnqueueWriteBuffer(queue_fc[0],
        weights_buf[i], CL_FALSE, // use blocking
        CL_TRUE?
465       0, weight_sizes[i] * sizeof(float), weights[i],
          0, NULL, NULL);
466     checkError(status, "Failed to transfer weights");
467
468     status = clEnqueueWriteBuffer(queue_fc[0], bias_buf
        [i], CL_FALSE, // use blocking CL_TRUE?
469       0, bias_sizes[i] * sizeof(float), bias[i], 0,
          NULL, NULL);
470     checkError(status, "Failed to transfer bias");
471
472     if (clFinish(queue_fc[0]) == CL_SUCCESS) //test
473       printf("cl_finish == success\n");
474
475     status = clEnqueueNDRangeKernel(queue_fc[0], knl_fc
        [0], 1, NULL,
476       global_work_size_fc, local_work_size_fc, 0, NULL,
            &fc_event[0]);
477     checkError(status, "Failed to launch kernel");
478
479     // Wait for all kernels to finish.
```

```
480         clWaitForEvents(num_devices, fc_event);
481
482         cl_ulong fc_time_ns = getStartEndTime(fc_event[0]);
483         printf("FC layer kernel time (device %d): %0.3f ms\
            n", 0, double(fc_time_ns) * 1e-6);
484         log_file << "FC layer " << i << " kernel time: " <<
             std::setprecision(3) <<  fc_time_ns * 1e-6 << "
            ms\n";
485
486         tot_kernel_time += fc_time_ns;
487       }
488     } // end layer iteration
489
490     // Read the result.
491     if (layer_config[num_layers-1][layer_type] == 0) { //
         conv layer
492       status = clEnqueueReadBuffer(queue_conv[0],
           output_buf[0], CL_TRUE,
493         0, OUTPUT_SIZE * sizeof(float), output, 0, NULL,
            NULL);
494       checkError(status, "Failed to read output matrix");
495     }
496     else if (layer_config[num_layers-1][layer_type] == 1) {
          // fc layer
497       status = clEnqueueReadBuffer(queue_fc[0], output_buf
           [0], CL_TRUE,
498         0, OUTPUT_SIZE * sizeof(float), output, 0, NULL,
            NULL);
499       checkError(status, "Failed to read output matrix");
500     }
501
502     if (output[1] > output[0]) {
503       printf("\nYou said HEY SPARK!\n\n");
504       printf(" %f < %f\n\n", output[0], output[1]);
505       break;
506     }
507     else {
508       printf("\nWrong word\n");
509       printf(" %f > %f\n\n", output[0], output[1]);
510     }
511   } // end while
512
513   //const double end_time = getCurrentTimestamp();
514   end_time = getCurrentTimestamp();
515   total_time = end_time - start_time;
```

```
516
517   // Wall-clock time taken.
518   printf("\nTotal time: %0.3f ms\n", total_time * 1e3);
519   printf("\nTotal kernel time: %0.3f ms\n", tot_kernel_time
          * 1e-6);
520
521   // Total time log
522   log_file << "\nTotal time: " << std::setprecision(3) <<
         total_time * 1e3 << "ms\n";
523   log_file << "\nTotal kernel time: " << std::setprecision
         (3) <<  tot_kernel_time * 1e-6 << "ms\n";
524   log_file.close();
525
526   // Release kernel events.
527   clReleaseEvent(conv_event[0]);
528   clReleaseEvent(fc_event[0]);
529
530   // Read the result.
531   if (layer_config[num_layers-1][layer_type] == 0) { //
         conv layer
532     status = clEnqueueReadBuffer(queue_conv[0], output_buf
           [0], CL_TRUE,
533       0, OUTPUT_SIZE * sizeof(float), output, 0, NULL, NULL
             );
534     checkError(status, "Failed to read output matrix");
535   }
536   else if (layer_config[num_layers-1][layer_type] == 1) {
         // fc layer
537     status = clEnqueueReadBuffer(queue_fc[0], output_buf
           [0], CL_TRUE,
538       0, OUTPUT_SIZE * sizeof(float), output, 0, NULL, NULL
             );
539     checkError(status, "Failed to read output matrix");
540   }
541
542   // Verify results.
543   verify();
544 }
545
546
547 void verify() { // need to account for some rounding
       differences
548   printf("Verifying\n");
549
550   bool success = 1; // mby err_cnt aswell
```

```
551
552   for (unsigned i = 0; i < OUTPUT_SIZE; i++) { // mby also
         check difference
553     printf("output[%d]      = %f\ngolden_ref[%d] = %f\n\n",
           i, output[i], i, golden_ref[i]);
554     if (output[i] != golden_ref[i])
555       success = 0;
556   }
557
558   if (success)
559     printf("The convolution is CORRECT\n");
560   else
561     printf("The convolution is  WRONG\n");
562 }
563
564 // Free the resources allocated during initialization
565 void cleanup() {
566
567   if(knl_conv && knl_conv[0]) {
568     clReleaseKernel(knl_conv[0]);
569   }
570   if(knl_fc && knl_fc[0]) {
571   clReleaseKernel(knl_fc[0]);
572   }
573   if(queue_conv && queue_conv[0]) {
574     clReleaseCommandQueue(queue_conv[0]);
575   }
576   if(queue_fc && queue_fc[0]) {
577     clReleaseCommandQueue(queue_fc[0]);
578   }
579   if(input_buf && input_buf[0]) {
580     clReleaseMemObject(input_buf[0]);
581   }
582   if(output_buf && output_buf[0]) {
583     clReleaseMemObject(output_buf[0]);
584   }
585   if(weights_buf && weights_buf[0]) {
586     clReleaseMemObject(weights_buf[0]);
587   }
588   if(bias_buf && bias_buf[0]) {
589     clReleaseMemObject(bias_buf[0]);
590   }
591   if(tmp_1_buf && tmp_1_buf[0]) {
592     clReleaseMemObject(tmp_1_buf[0]);
593   }
```

```
594   if(tmp_2_buf && tmp_2_buf[0]) {
595     clReleaseMemObject(tmp_2_buf[0]);
596   }
597   if(program) {
598     clReleaseProgram(program);
599   }
600   if(context) {
601     clReleaseContext(context);
602   }
603
604   for (int i = 0; i < num_layers; i++) {
605     alignedFree(weights[i]);
606     alignedFree(bias[i]);
607   }
608   alignedFree(input);
609   alignedFree(output);
610   alignedFree(golden_ref);
611 }
```

Listing 11: main.cpp

A.1.5 Live setup code

```
1  #ifndef RECORD_VOICE_H
2  #define RECORD_VOICE_H
3
4  #define SAMPLE_RATE 48000
5  #define FRAMES_PER_BUFFER 1024
6  #define NUM_SECONDS 0.9
7  #define NUM_CHANNELS 1
8
9  #define PA_SAMPLE_TYPE paFloat32
10 #define SAMPLE_SILENCE 0.0f
11
12 void record_voice(float* recorded_samples);
13
14
15 #endif // RECORD_VOICE_H
```

Listing 12: record_voice.h

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "portaudio.h"
4  #include "wav_utils/record_voice.h"
5
6  void record_voice(float* recorded_samples) {
```

```
7
8    PaStreamParameters input_parameters;
9    PaStream* stream;
10   PaError err;
11   //float* recorded_samples;
12   int total_frames;
13   int num_samples;
14   int num_bytes;
15   float max, average, val;
16
17   total_frames = NUM_SECONDS * SAMPLE_RATE;
18   num_samples = total_frames * NUM_CHANNELS;
19   num_bytes = num_samples * sizeof(float);
20
21   //recorded_samples = (float*)malloc(num_bytes);
22   if (recorded_samples == NULL){
23     printf("Could not allocate record array.\n");
24     exit(1);
25   }
26
27   for (int i = 0; i < num_samples; i++)
28     recorded_samples[i] = 0;
29
30   err = Pa_Initialize();
31   if (err != paNoError)
32     goto error;
33
34   input_parameters.device = Pa_GetDefaultInputDevice();
35   if (input_parameters.device == paNoDevice) {
36     fprintf(stderr, "Error: No default input device.\n");
37     goto error;
38   }
39   input_parameters.channelCount = NUM_CHANNELS;
40   input_parameters.sampleFormat = PA_SAMPLE_TYPE;
41   input_parameters.suggestedLatency = Pa_GetDeviceInfo(
         input_parameters.device)->defaultLowInputLatency;
42   input_parameters.hostApiSpecificStreamInfo = NULL;
43
44   err = Pa_OpenStream(&stream,
45                       &input_parameters,
46                       NULL,                 // &
                             output_parameters
47                       SAMPLE_RATE,
48                       FRAMES_PER_BUFFER,
49                       paClipOff,            // we won't
```

```
                                 output out of range samples so don
                                 't bother clipping them
50                          NULL,                // no callback,
                                 use blocking api
51                          NULL);               // no callback,
                                 so no callback userData
52    if (err != paNoError)
53      goto error;
54
55    err = Pa_StartStream(stream);
56    if (err != paNoError)
57      goto error;
58    printf("\n------NOW RECORDING!!------\n\n"); // fflush(
         stdout);
59
60    err = Pa_ReadStream(stream, recorded_samples,
         total_frames);
61    if (err != paNoError)
62      goto error;
63
64    err = Pa_CloseStream(stream);
65    if (err != paNoError)
66      goto error;
67
68    //free(recorded_samples);
69
70    Pa_Terminate();
71
72    return;
73
74  error:
75    Pa_Terminate();
76    fprintf(stderr, "An error occured while using the
         portaudio stream\n");
77    fprintf(stderr, "Error number: %d\n", err);
78    fprintf(stderr, "Error message: %s\n", Pa_GetErrorText(
         err));
79    return;
80  }
```

Listing 13: record_voice.cpp

```
1  #ifndef WRITE_WAV_H
2  #define WRITE_WAV_H
3
4  typedef struct wav_header_t {
```

```
5        // RIFF wave header
6        char   chunk_id[4];
7        int    chunk_size;
8        char   format[4];
9
10       // Format subchunk
11       char   subchunk1_id[4];
12       int    subchunk1_size;
13       short  audio_format; // short int
14       short  num_channels;
15       int    sample_rate;
16       int    byte_rate;
17       short  block_align;
18       short  bits_per_sample;
19
20       // Data subchunk
21       char   subchunk2_id[4];
22       int    subchunk2_size;
23   } wav_header_t;
24
25   void write_wav(const char* file_name, float* audio_data,
         int num_bytes, short num_channels, int sample_rate,
         short bits_per_sample);
26
27   #endif // WRITE_WAV_H
```

Listing 14: write_wav.h

```
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <fstream>
4
5    #include "wav_utils/write_wav.h"
6
7
8
9    void write_wav(const char* file_name, float* audio_data,
         int num_bytes, short num_channels, int sample_rate,
         short bits_per_sample) {
10     wav_header_t header;
11     // RIFF wave header
12     header.chunk_id[0] = 'R';
13     header.chunk_id[1] = 'I';
14     header.chunk_id[2] = 'F';
15     header.chunk_id[3] = 'F';
16     header.chunk_size = 36 + num_bytes;
```

```
17   header.format[0] = 'W';
18   header.format[1] = 'A';
19   header.format[2] = 'V';
20   header.format[3] = 'E';
21
22   // Format subchunk
23   header.subchunk1_id[0] = 'f';
24   header.subchunk1_id[1] = 'm';
25   header.subchunk1_id[2] = 't';
26   header.subchunk1_id[3] = ' ';
27   header.subchunk1_size = 16; // 16 for PCM, size for rest
        of subchunk
28   header.audio_format = 3;    // 1 for PCM, 3 for float it
        seems
29   header.num_channels = num_channels;
30   header.sample_rate = sample_rate;
31   header.bits_per_sample = bits_per_sample;
32   header.byte_rate = header.sample_rate * header.
        num_channels * header.bits_per_sample/8;
33   header.block_align = header.num_channels * header.
        bits_per_sample/8;
34
35   // Data subchunk
36   header.subchunk2_id[0] = 'd';
37   header.subchunk2_id[1] = 'a';
38   header.subchunk2_id[2] = 't';
39   header.subchunk2_id[3] = 'a';
40   header.subchunk2_size = num_bytes;
41
42   // Write
43   std::ofstream file(file_name, std::ios::binary); // std::
        ios::out
44   file.write((char*)&header, sizeof(header));
45   file.write((char*)audio_data, num_bytes);
46   file.close();
47
48 }
```

Listing 15: write_wav.cpp

A.2 Reports

All the reports are included included in the attachements. In this section only the top.fit.summary report and base.fit.summary report for the best performing CNN is included.

```
1  Fitter Status : Successful - Sat Jun  3 01:48:39 2017
2  Quartus Prime Version : 16.0.2 Build 222 07/20/2016 SJ Pro
      Edition
3  Revision Name : top
4  Top-level Entity Name : top
5  Family : Arria 10
6  Device : 10AX115S3F45E2SGE3
7  Timing Models : Preliminary
8  Logic utilization (in ALMs) : 66,204 / 427,200 ( 15 % )
9  Total registers : 119130
10 Total pins : 173 / 960 ( 18 % )
11 Total virtual pins : 0
12 Total block memory bits : 11,677,962 / 55,562,240 ( 21 % )
13 Total RAM Blocks : 832 / 2,713 ( 31 % )
14 Total DSP Blocks : 97 / 1,518 ( 6 % )
15 Total HSSI RX channels : 8 / 72 ( 11 % )
16 Total HSSI TX channels : 8 / 72 ( 11 % )
17 Total PLLs : 14 / 144 ( 10 % )
```

Listing 16: CNN NDRconv 4 CUs top.fit.summary

```
1  Fitter Status : Successful - Sat Jun  3 00:32:48 2017
2  Quartus Prime Version : 16.0.2 Build 222 07/20/2016 SJ Pro
      Edition
3  Revision Name : base
4  Top-level Entity Name : top
5  Family : Arria 10
6  Device : 10AX115S3F45E2SGE3
7  Timing Models : Preliminary
8  Logic utilization (in ALMs) : 57,385 / 427,200 ( 13 % )
9  Total registers : 87212
10 Total pins : 173 / 960 ( 18 % )
11 Total virtual pins : 0
12 Total block memory bits : 2,145,162 / 55,562,240 ( 4 % )
13 Total RAM Blocks : 308 / 2,713 ( 11 % )
14 Total DSP Blocks : 2 / 1,518 ( < 1 % )
15 Total HSSI RX channels : 8 / 72 ( 11 % )
16 Total HSSI TX channels : 8 / 72 ( 11 % )
17 Total PLLs : 14 / 144 ( 10 % )
```

Listing 17: CNN NDRconv 4 CUs base.fit.summary