Gate Recurrent Unit Neural Networks for Hearing Instruments

Harshit Sharma
ha2442sh-s@student.lu.se
Pallavi Rajanna
pa6354ra-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Oskar Andersson and Adrian Mardari

Academic Supervisor: Joachim Rodrigues

Examiner: Pietro Andreani

June 25, 2020

© 2020
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Gated Recurrent Unit (GRU) neural networks have gained popularity for applications such as keyword spotting, speech recognition and other artificial intelligence applications. Typically for most applications training and inference is performed on cloud servers, and the result are transferred to the power constrained device, e.g., an hearing instrument (HI). This approach has disadvantages such as latency and connectivity, privacy concern, and high energy cost per bit for real-time data transfer. Therefore, there is a strong demand to move inference from cloud to power constraint devices. However, executing inference on HI introduces many challenges in terms of throughput, power budget, and memory footprint. This research investigate how efficient it is to execute inference on a dedicated hardware accelerator, rather than using an existing audio digital signal processor (xDSP in *Oticon's* HI).

The two approaches are compared in terms of area, power, energy dissipation and total clock cycles required to perform an inference. Straightforward implementation of nonlinear activation function is expensive in hardware, therefore, different methods of approximation are evaluated. Out of different approximation algorithms, fast sigmoid and fast tanh approaches were chosen. A pretrained keyword spotting (KWS) model was used. However, it exceeds the memory space available on xDSP. Instead, three small GRU networks were trained and executed on xDSP to approximate energy dissipation and clock cycle count if a bigger network was run on the xDSP.

Precision needed to store and compute data was reduced to minimize storage needed keeping detection accuracy in mind. By reducing wordlength from 32-bit to 8-bit for network parameters, memory space required was reduced by 4 times while accuracy decreased from 91% to 88%. The GRU inference runs on per layer basis, data flow was optimized to achieve significant reduction in area and power.

The xDSP needs around 2× more clock cycles to complete a full network inference for a benchmark keyword spotting neural network compared to dedicated hardware accelerator. The energy dissipation increased by around 10× while using Oticon's xDSP processor instead of a dedicated accelerator. The xDSP is capable of executing GRU network with upto 40 neurons per layer, but for bigger networks hardware accelerator is a better solution. All in all, the dedicated accelerator solution has the best performance from the explored solution and can be integrated in HI to compute neural networks.

Popular Science Summary

Artificial intelligence is becoming a huge part of our life, from being used in mobile phones, smart watchs, home entertainment systems etc. However, due to a large amount of computations need to be performed to execute a simple task, most of this processing is done in cloud servers. Although, there is a breakthrough in artificial intelligence, there has been serious limitation in terms of power and energy efficiency that needs to be addressed.

In 2019, artificial intelligence (AI) computer program known as AlphaStar built by Google's AI firm DeepMind played the science-fiction video game *StarCraft II* on European servers. The AI competed against 90,000 player and was placed within the top 0.15%. DeepMind, previously built world-leading AIs that play chess and Go. However, estimated power consumption of these AI is in order of megawatts, whereas human brain only consumes 20 watt. This means that AI needs to be more efficient before it can be completely integrated in daily our life.

AI has gained popularity in speech recognition technology, where AI can recognize spoken words, which can then be converted to text or used to perform tasks. A subset of speech recognition is keyword spotting, where a task is performed after identifying a keyword in the input voice signal. Companies such as Facebook, Amazon, Microsoft, Google and Apple have already integrated this feature on various devices through services like Google Home, Amazon Echo and Siri.

With this in mind, the goal of this thesis has been to select a pretrained keyword spotting model and propose a efficient dedicated hardware accelerator to perform this task. The spotting of spoken keyword has been performed using a GRU algorithm which is an advanced recurrent neural network. In order to compare the scalable and efficient hardware accelerator design, it was compared with an existing audio digital signal processor used in Oticon's hearing instruments. This research addresses the problem of high power consumption and large memory reference that restricts the use of large scale neural networks on power constrained devices. Research also addresses the issue of privacy, i.e., sharing of data with cloud servers.

The proposed dedicated hardware accelerator can be integrated in HI to compute neural networks.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

GRU	**G**ated **R**ecurrent **U**nit
GNNE	**G**RU **N**eural **N**etwork **E**ngine
ANN	**A**rtificial **N**eural **N**etwork
RNNs	**R**ecurrent **N**eural **N**etwork**s**
NN	**N**eural **N**etwork
NNs	**N**eural **N**etwork**s**
RTL	**R**egister **T**ransfer **L**evel
RAM	**R**andom-**a**ccess **M**emory
MAC	**M**ulitiply **A**nd **A**ccumulate
LUT	**L**ook **U**p **T**able**s**
KWS	**K**ey**w**ord **S**poting
FSM	**F**inite **S**tate **M**achine
FSMs	**F**inite **S**tate **M**achine**s**
HI	**H**earing **I**nstruments

Chapter 1
Introduction

Deep learning is an umbrella term for a set of machine learning algorithms that attempt to model high-level abstractions in data. Included in this machine learning branch are neural networks, which try to mimic the highly intricate way of the human brain in which it processes information. In large, due to increasingly powerful computing capabilities, these types of networks have, in the past years, produced remarkable advances in several fields.

1.1 Artificial Neural Networks

Neural Networks (NNs) were first proposed by Warren McCullough and Walter Pitts in 1944. Neural networks are multi-layer networks of neurons that are used to classify things, make predictions, etc. Artificial Neural Networks (ANNs) attempt to emulate the human brain, which is a collection of connected networks of neurons. An ANN is based on a network of connected units or nodes called artificial neurons where each connection can transmit a signal to other neurons. Different sections of the human brain are responsible for processing different pieces of information, and these parts of the brain are arranged hierarchically in layers. Therefore, as information enters the brain, it is processed in each layer of neurons and passed to the next one. It is this layered approach to processing information and making decisions that ANNs are trying to replicate. In the simplest form, an ANN has only three layers of neurons: the input layer (where the data enters the system), the hidden layer (where the information is processed), and the output layer (where the system decides what to do based on the data). However, ANNs can get much more complex than that, and include multiple hidden layers. When a Neural Network (NN) has more than one hidden layer it is referred to as a deep neural network. This distinction is illustrated in Figure 1.1. Whether it is three layers or more, information flows from one layer to another, just like in the human brain [1].

The basic structure of a NN consists of neurons whose underlining operation is *multiply-accumulate* (MAC). A neuron computes the weighted average of the input data and passes the information through a non-linear function, i.e., activation function, such as sigmoid function. Figure 1.2 illustrates operations at one neuron of a NN. For a given artificial neuron, let there be a $n+1$ inputs with signals a_1 through a_N and weights w_1 through w_N. The inputs (a_1 to a_N) are multiplied with their respective weights (w_1 to w_N), and the results are summed together. A

bias value (b) is added to this value, resulting in the final sum (z). The activation function is used to bring non-linearity in the output of the neuron.

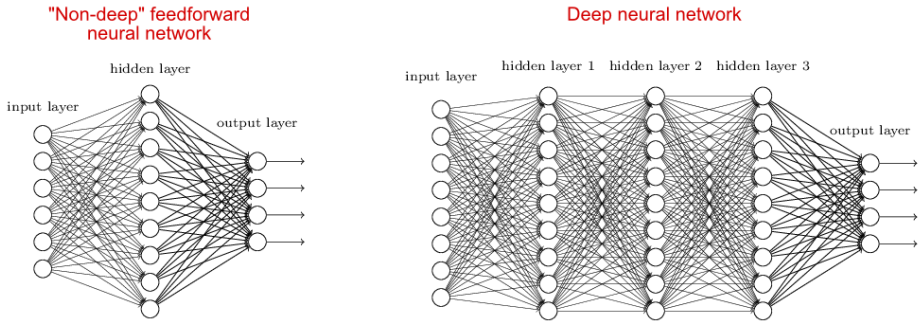

Figure 1.1: Neural network vs deep neural network. Edges represent weights and vertices/nodes represent activations [2].

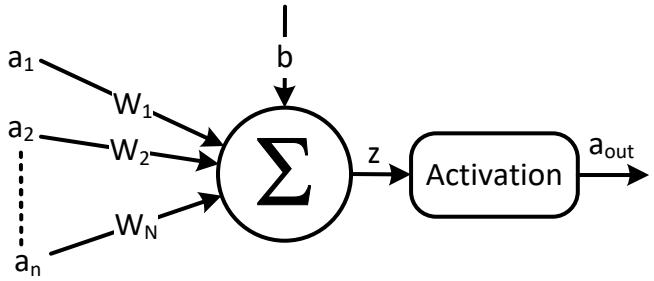

Figure 1.2: Operations at one neuron of a neural network.

Every activation function (or non-linearity) takes a number as input and performs a certain fixed mathematical operation on it. Figure 1.3, Figure 1.4 and Figure 1.5 illustrates three commonly used activation functions namely sigmoid, tanh, and rectified linear unit (ReLu).

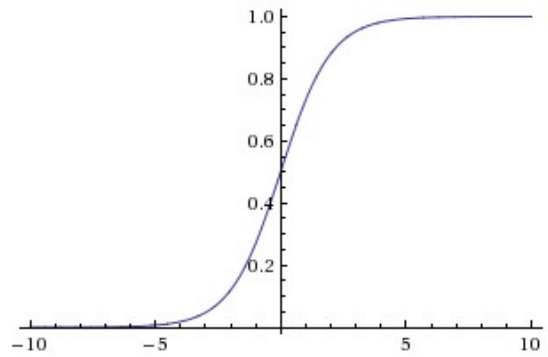

Figure 1.3: Sigmoid non-linearity squashes real numbers to range between [0,1].

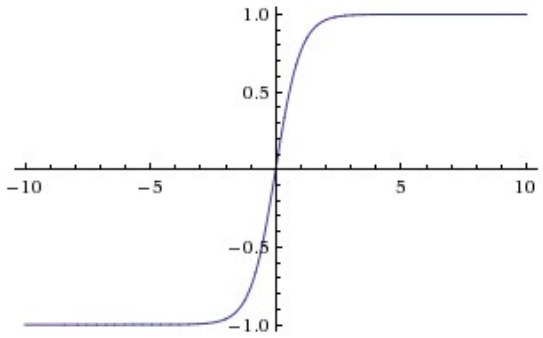

Figure 1.4: The tanh non-linearity squashes real numbers to range between [-1,1].

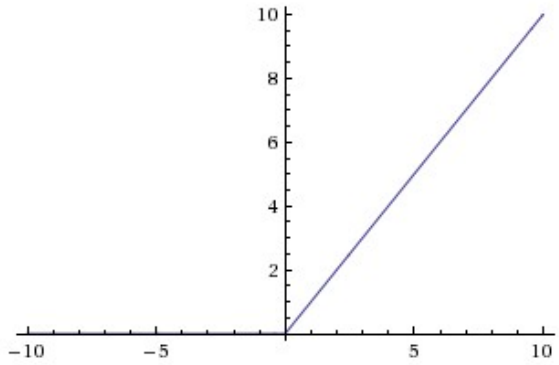

Figure 1.5: Rectified Linear Unit (ReLU) activation function, which is zero when $x < 0$ and then linear with slope 1 when $x > 0$.

Activation functions and the algorithms explored to approximate the sigmoid and tanh activation functions are explained later in Chapter 3.

The three most commonly used types of neural network architectures in artificial intelligence:

- Feedforward NNs - also referred to as multi-layer perceptrons, dense/fully connected networks, or ANNs. The flow of information through the network is unidirectional without any internal feedback connections, and are the essential deep learning models [3].
- Recurrent NNs (RNNs) [4]–[6] - these networks have an internal state (memory) to process sequences of inputs. Unlike feedforward NNs, RNNs have feedback loops which makes their output depended on the previous computations. However, conventional RNNs have a few limitations. They are difficult to train and have a very short-term memory, which limits their functionality. More advanced variants of RNNs models were developed, e.g., Gated Recurrent Unit (GRU) and Long Short Term Memory (LSTM), to overcome this problem.
- Convolutional neural network - has the ability to capture the spatial and temporal dependencies in an input through the application of relevant filters. It performs better on the image data set due to the reduction in the number of parameters involved and reusability of weights [7].

One of the neural network architecture paradigms that have driven breakthroughs in deep learning is RNNs. Recurrent neural networks, as their name states, use recurrent paths (loops) to insert previous outputs as inputs for the current iteration as shown in Figure 1.6.

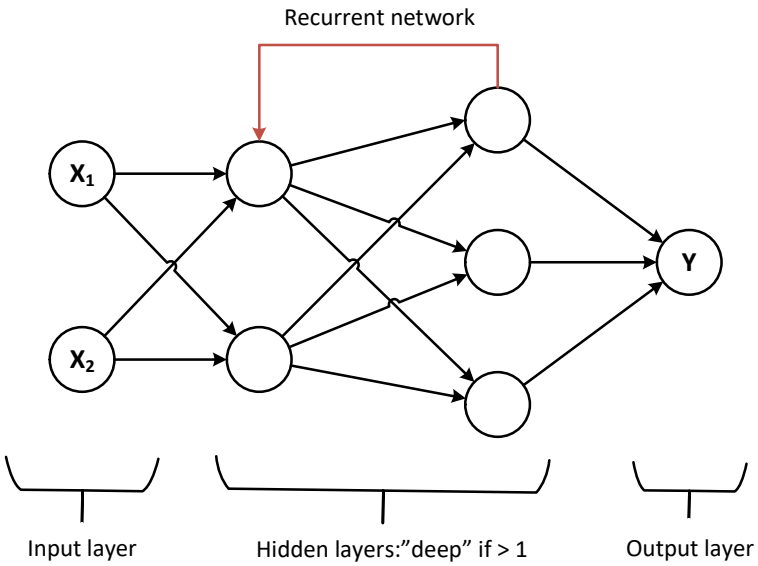

Figure 1.6: Recurrent Neural Network.

RNNs are a type of neural network that takes sequential input and produces sequential output by sharing parameters between time steps. Two identical data structures will not produce the same output if the previous inputs were not identical as well. Compared to the feed-forward neural networks which do not process previous results or states to make decisions. Feed forward computations require no feedback of previous outputs or any kind of short or long range context as shown in Figure 1.7. A feed-forward neural network is trained to match two identical inputs to the same output. This is very efficient in many cases where past events must not alter the networks perception of the current situation. A typical example is image recognition where two identical images must produce the same output.

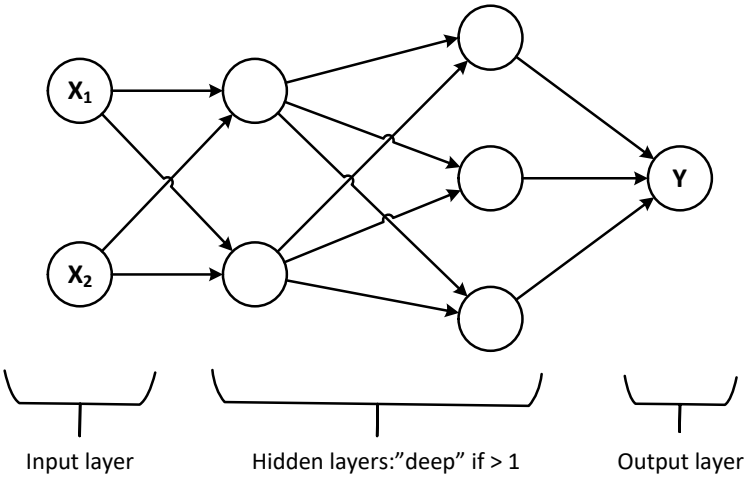

Figure 1.7: Feedforward Neural Network.

RNNs have led to breakthrough results in natural language processing [8], image captioning [9], and speech recognition [10]. Their limitation is that this mechanism provides access to the most recent events only. As stated in [10], traditional RNNs are incapable of dealing with long term dependencies.

1.2 Artificial NN at the edge

There are two main phases in the deep learning process: training and inference. Training refers to the preparation of a machine learning model to do a specific task. During training, each layer of data in the model assigned with random weights. After running forward pass through the data, the model predicts the scores and classify labels using the random weights assigned. After comparing classified labels against the actual labels, a loss function is used to compute the error. This error is then backpropagated through the network, and weights are updated accordingly via weight update algorithm such as Gradient Descent.

The inference is the actual execution of the task using the statistical model obtained during the training session. It comprises of a similar forward pass as

training to predict the values. Unlike training, it does not include a backward pass to compute the error and update weights. Therefore, the inference cannot happen without training.

Training and inference have different computational needs. Since deep learning algorithms are very costly in terms of energy dissipation, both the training of a NN and its inference is typically executed on Central Processing Unit (CPU) servers or Graphics Processing unit (GPU) in the cloud. Afterward, the results are transmitted to lower-complexity power-constrained (edge) devices such as hearing instruments. However, this setup has several disadvantages [11]

- Latency and connectivity issues, as it requires a wearable device always to be connected to the cloud, and low latency is critical for specific tasks.
- Privacy issues related to sharing raw data such as images, speech, video, locations with a remote system, which is not desirable for users.
- Wireless connections have a high energy cost per transferred bit for real-time data transfer on energy-constrained platforms.

Due to these reasons, there is a strong demand to move especially the inference from the cloud to the edge instead. This setup would mitigate latency and privacy issues as well as improve the battery life of the device [12] [3]. Designing efficient hardware architectures at the edge is therefore crucial; especially for low-power devices such as hearing instruments where area, memory footprint, power budget, and throughput, are limiting factors. These limiting factors of hearing instruments have the disadvantage that large NNs cannot fit on the edge device. Therefore, there should be a balance between which NNs that are deployed on the edge devices and which NNs that still are run in the cloud. Another motivation to design dedicated hardware accelerators for NNs is that conventional processors are not optimized for running NNs.

1.3 Recurrent Neural Networks

The feed forward neural networks make assumption that all inputs are independent on each other. This assumption is wrong in the case of sequential data, where data exhibits a dependency on past data.

RNN model include dependency on past data through a hidden state, or memory, that holds information of what has been processed so far. At any point in time, (1.1) shows that the value of the hidden state is a function of the value of the hidden state at the previous time step and the value of the input at the current time step.

$$h_t = \phi(h_{t-1}, x_t) \tag{1.1}$$

The term h_t and h_{t-1} are the values of hidden state at the time steps t and *t-1*, respectively, and x_t is the input value at time t. RNNs are network of nodes (neurons) organized into successive layers. Each node in a given layer is connected with a one-way connection to every other node in the next layer. Figure 1.8 illustrates the graphical representation of RNN cell on the left and unrolled RNN on the right. The RNNs, parameters are defined by three weight matrices U, V,

and W, corresponding to the input, output and hidden state respectively. At time t, the cell has an input x_t and an output y_t. A portion of output y_t (the hidden state h_t) is fed back into the cell as an input to next time step *t+1*.

Since, the same operation is applied on different inputs at each time step, weights matrices U, V, and W are shared across the time steps. Due to the sharing of weight vectors across all the time steps, the number of parameters that needs to learn reduces. The value of hidden vector is

$$h_t = \tanh(Wh_{t-1} + Ux_t), \tag{1.2}$$

where W is the hidden state weigh matrix, h_{t-1} is the previous hidden state, U is the input weight matrix and x_t the input at time t.

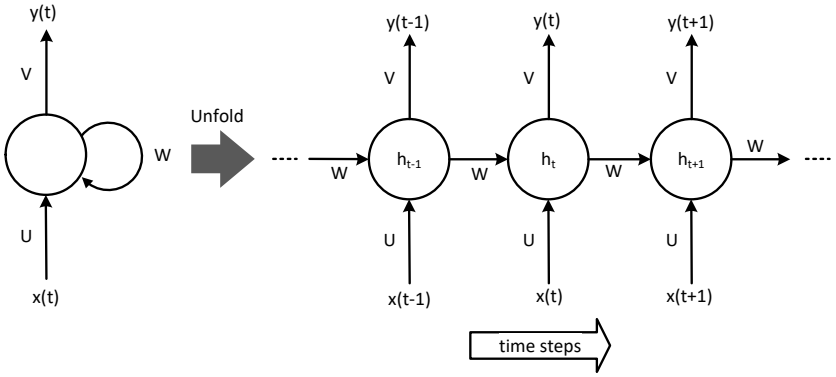

Figure 1.8: RNN cell.

Using (1.3), the output vector is computed where V is the weight matrix, h_t the hidden state at time t and softmax is the function that converts the raw predictions to the probability distributions of a list of potential outcomes the input sequence belongs.

$$y_t = softmax(Vh_t) \tag{1.3}$$

Tanh is chosen over other nonlinearities, to keep the gradients in the linear region of the activation function and prevent the vanishing gradient problem.

1.3.1 Vanishing and exploding gradient problem

Training of RNN involves back propagation. Since the parameters are shared by all the time steps, the gradient at each output not only depends on the current time step, but also on the previous time steps. This is known as backpropagation through time (BPTT).

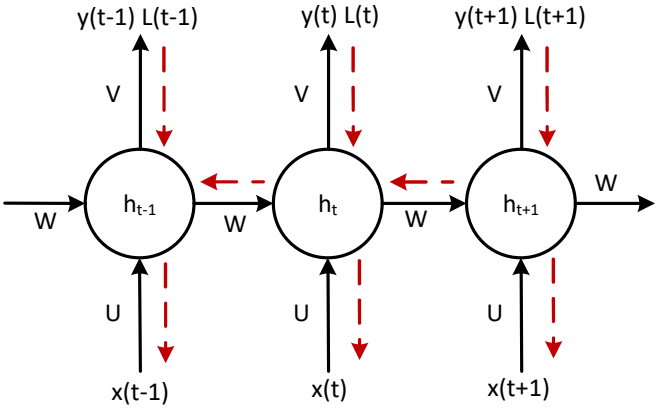

Figure 1.9: Back propagation in RNNs.

Figure 1.9 represents the state of an RNN at different instances of time t. During the forward propagation (denoted by solid black lines), the network produces predictions that are compared with the labels (ground truth values) to compute the loss L_t at each time step. During backpropagation (shown by solid red lines), the gradients of the loss with respect to the weight matrices U, V, and W are computed at each time step and the parameters are updated by adding the gradients. At any point of time, the gradient is the product of all the gradients up to that point.

When the gradients of the hidden state is less than one, brackpropagation across multiple time steps leads to smaller and smaller product of the gradients, leading to vanishing gradient problem. Similarly, if the gradients are larger than one, the product gets larger and larger, leading to the exploding gradient problem.

The side effect of vanishing gradients is that the gradients of distant time steps do not contribute to the learning process, so the RNN do not get trained for long term dependencies. Exploding gradients are easy to identify, as the gradients becomes very large, the training process will crash.

As stated in [10], traditional RNNs are incapable of dealing with long term dependencies. Over the years more advanced alternatives such as the Gated Recurrent Unit (GRU) and Long Short Term Memory (LSTM) have evolved. Both the architectures have been designed to deal with vanishing gradient problem and learn long term dependencies. The GRU is a simpler form of LSTM, using fewer gates that modulate the flow of information inside the unit and has no separate memory cell. The GRU architecture is the focus of this thesis and is explained in the chapter 2 in detail.

1.4 Keyword Spotting System

The goal of keyword spotting is to detect the presence of specific spoken words in unconstrained speech. The majority of keyword spotting systems are based on generative hidden Markov models and lack discriminative capabilities. However, discriminative keyword spotting systems are currently based on frame-level posterior probabilities of sub-word units [13]. This project presents a discriminative keyword

spotting system based on GRU NN, that uses information from long time spans to estimate word-level posterior probabilities. KWS application has highly constrained power budget and typically runs on tiny microcontrollers with limited memory and compute capability [14].

An open-source KWS project developed by ARM [14] is used in our experiments. The project provide pretrained models for various neural network architectures such as DNN, CNN, LSTM, GRU, etc.

The parameters are extracted from the pretrained GRU model using python and are used for inference in both high-level (Python and Matlab) and low level (xDSP and hardware) implementation. The pretrained GRU model has three layer (input, hidden and output), having 154 neurons in input and hidden layer and 12 neurons in output layer. The input to the GRU NN is the flattened featured matrix, Mel-frequency cepstral coefficients (MFCC) is one of the commonly used for feature extraction. Feature extraction using MFCC involves translating the time-domain speech signal into a set of frequency-domain spectral coefficients, which enables dimensionality compression of the input signal.

The input data represents features from a one-second long audio recordings which are obtained from the Google speech commands dataset [14] created by TensorFlow and AIY teams. The dataset has more than 65,000 one-second long utterances of 30 short words, by thousands of different people. The files are in 16-bit little-endian PCM-encoded WAVE format. They are divided into three sets, namely training, validation, and test set in 80:10:10 ratio that undergo further selection. Raw data was extracted from the recordings using Python in a test set, which was used as an input for the GRU NN.

Figure 1.10 illustrates a feature extractor and a NN based classifier. The input signal of length L is framed into overlapping frames with window size of l and window stride s. The total frames are given as

$$T = \frac{L - l}{s} + 1.$$

From each frame F MFCC features are extracted, producing total of TF features for the input audio signal of length L. For GRU NN, 10 MFCC features (F) are extracted from a window size of 40ms (l), with a stride of 40ms (s), which gives 250 features for 1 second of audio input. A GRU NN processes 250 features sequentially, ten per frame.

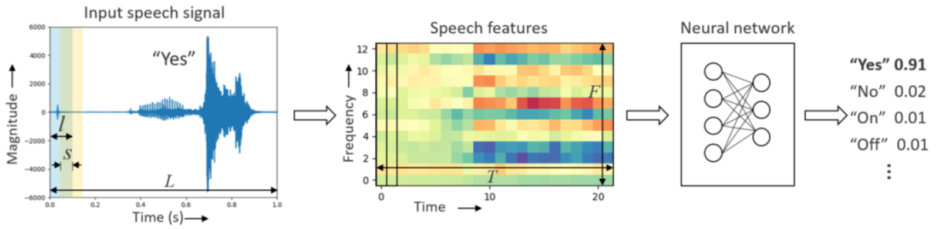

Figure 1.10: KWS system [14].

The output layer has 12 neurons, each representing on keyword/category. The first neuron correspond to *"silence"* (no speech is present in the recording) and

"unknown" (meaning that GRU NN is not able to classify the word into one of the twelve keywords). The remaining ten neurons represents the following keywords respectively : *"yes", "no","up", "down", "left", "right", "on", "off", "stop", "go"*.

To meet the requirement of implemented feature extractor, there is a real time requirement of 40ms to classify the features using GRU NN.

1.5 Thesis Goal

The objective of this master thesis is to examine the GRU algorithm and propose an efficient ASIC architecture capable of functioning as an GRU NN and can be implemented at the edge device. To achieve the mentioned goals, the thesis project is split into five tasks:

1. **Behavioral model of GRU Network using Python and Matlab** - The first task is to implement a behavioral model of GRU network using numpy scientific library, and parameters from pretrained KWS model in Python and then construct a fixed-point model without using any predefined functions. Both the numpy and fixed-point models are functionally the same and are used as reference models.

2. **Basic implementation of neural network on the xDSP** - Implement GRU NN inference on Oticon's audio DSP processor referred to as the xDSP. This implementation is carried out to obtain important metrics such as cycle count and energy dissipation, which are further used to compare performance of dedicated GRU engine (GRUE).

3. **Wordlength exploration and hardware optimization** - Neural network accelerators with low latency and low energy dissipation are desirable for edge computing. Due to resource constraints, specifically the memory requirements, model size plays an important role. Model size refers to the network parameters (inputs, weights and biases). A common approach to reduce the model size is by reducing the precision of the parameters. However, reducing precision leads to trade-off between network accuracy and computation efficiency. Different ways of realising activation functions such as sigmoid and tanh are also explored to have efficient hardware resource use. Therefore, the purpose of this task is to investigate efficient algorithms to implement the activation functions and identify the appropriate wordlength for representing the NN parameters.

4. **Data reuse exploration** - For RNNs, the fundamental operation unit is MAC which requires multiple memory access operations. This has an impact on both throughput and energy efficiency. Hence, we will exploit different data reuse techniques, to enhance the parallel execution of the algorithm, and minimize the data movement.

5. **A dedicated neural network engine** - Implementation of a dedicated accelerator and investigate the improvements in latency and energy dissipation.

Chapter 2

Gated Recurrent Unit - GRU

GRU is a variant of RNN that is capable of learning long term dependencies. GRUs have an internal mechanism called gates that regulate the flow of information and avoid the vanishing gradient problem. Two such mechanisms are update gate z_t and reset gate r_t. During training GRU learns the parameters for these gates. This chapter will explain how the GRU mechanism work and discuss how the output of a GRU cell can be used to classify the given input sequence.

2.1 A GRU cell

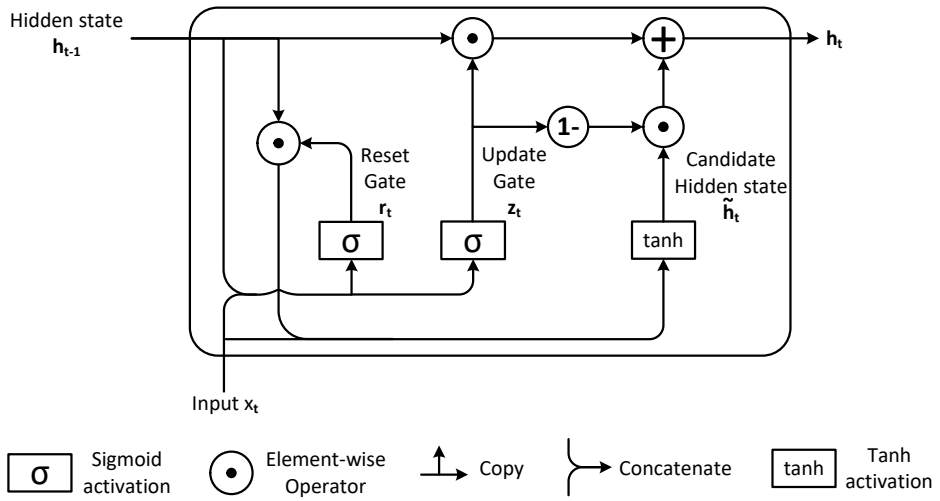

Figure 2.1: Gated Recurrent Unit (GRU).

The architecture of a GRU cell used is depicted in Figure 2.1 and encapsulated in the following equations:

$$r_t = \sigma(x_t U_r + h_{t-1} W_r + b_r) \tag{2.1}$$

$$z_t = \sigma(x_t U_z + h_{t-1} W_z + b_z) \tag{2.2}$$

$$\tilde{h_t} = \tanh(x_t U_h + (r_t \odot h_{t-1}) W_h + b_h) \tag{2.3}$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h_t}. \tag{2.4}$$

Each element of (2.1), (2.2), (2.3), (2.4) is summarized as follows:

- r_t is the reset gate.
- z_t is the update gate.
- $\tilde{h_t}$ is the candidate hidden state.
- U and W are weight matrix, and subscripts r, z, and h indicate that weight matrix belongs to reset gate, update gate and candidate hidden state, respectively.
- b denotes the bias vector and subscripts r, z, and h indicate that bias vector belongs to reset gate, update gate and candidate hidden state, respectively.
- σ denotes the sigmoid activation and tanh signify tangent hyperbolic activation.
- $\odot$ stands for element-wise vector multiplication also known as Hadamard product, and subscripts t denote the time step associated with the gate or unit.

2.1.1 Reset gate and Update gate

Through the use of reset r_t and update z_t gates, a GRU network can learn when the incoming information is important and when it should be integrated into the internal state of the cell (hidden state).

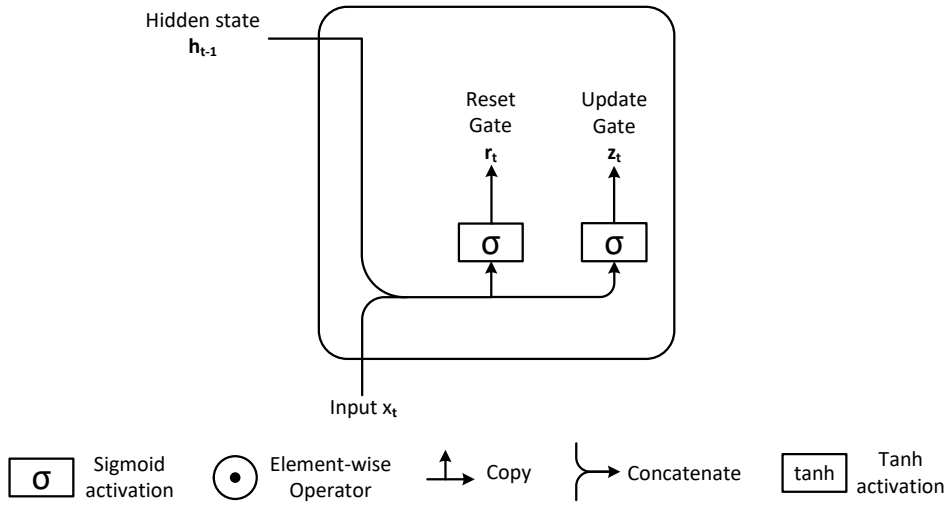

Figure 2.2: Reset and update gate in a GRU unit.

The reset gate r_t defines how to combine the new input x_t with the previous memory h_{t-1}, i.e., how much past information to forget. The update gate z_t allow

us to control how much hidden state h_t is a copy of the old state. The sigmoid function modulates the outputs of the reset and update gate to the interval [0,1].

Figure 2.2 illustrates the inputs for both the reset and update gate in a GRU unit, for current time step input x_t and the hidden state of previous time step h_{t-1}. The reset gate r_t and update gate z_t are computed as (2.1) and (2.2), respectively.

2.1.2 Candidate hidden state

The candidate hidden state (h_{t-1}) is computed as (2.3). The value of the reset gate r_t decides how much information from previous hidden state h_{t-1} to forget, denoted by $r_t \odot h_{t-1}$. Setting the value of r_t closer to one makes the influence of the previous hidden state h_{t-1} influence less and vice versa. Figure 2.3 illustrates the computational flow after including the candidate hidden state. The symbol $\odot$ indicates the pointwise multiplication between the tensors.

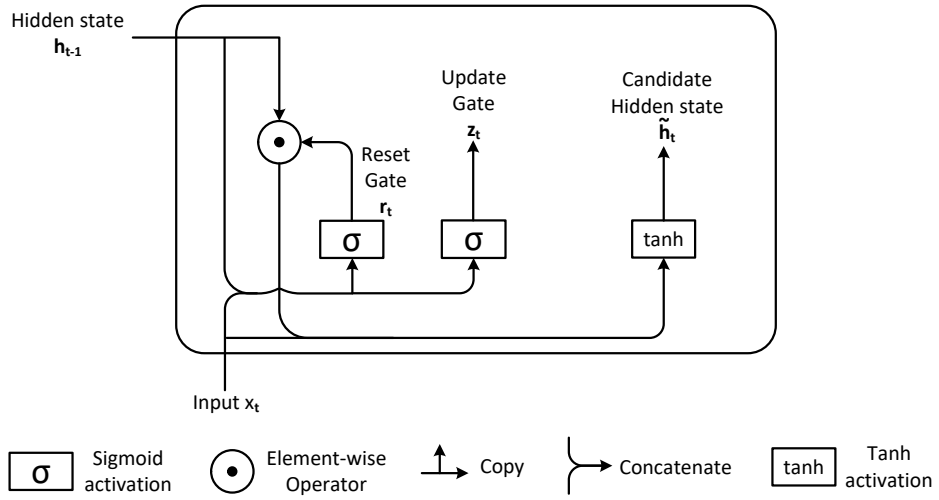

Figure 2.3: Candidate hidden state computation in a GRU unit.

2.1.3 Hidden state

The hidden state, h_t, is computed as (2.4). The hidden state stores the information of the current GRU cell and passes this information down the network. The update gate z_t is used to decide how much information to gather from previous hidden state h_{t-1}, given by $z_t \odot h_{t-1}$ and current memory content $\tilde{h_t}$, given by $(1-z_t) \odot \tilde{h_t}$. When z_t is set close to one, $1 - z_t$ will be close to zero, the model retains the previous information h_{t-1}, ignoring the information from x_t and skipping the timestep t in the dependency chain. On other hand, when z_t is close to zero, the new hidden state h_t approaches the candidate hidden state $\tilde{h_t}$.This helps to cope with the vanishing gradient problem and capture long-term dependencies better.

2.1.4 Output layer

A fully connected layer that takes the hidden state output h_t as input and delivers the number of classes in the network as output y_t. The output layer is computed by

$$y_t = h_t W_y + b_y. \tag{2.5}$$

After the output layer, softmax activation is performed that converts the logits into probabilities that sum to one. Softmax activation given by (2.6), outputs a vector that represents the probability distributions of a list of potential outcomes the input sequence belongs. Logits are the raw prediction output by the output layer of GRU NN, before softmax activation takes place.

$$softmax(y_i) = \frac{e^{y_i}}{\sum_{j=1}^{n} e^{y_j}} \tag{2.6}$$

Figure 2.4 shows the complete structure of an GRU cell connected with a fully connected layer and a softmax function to deliver the expected probability results.

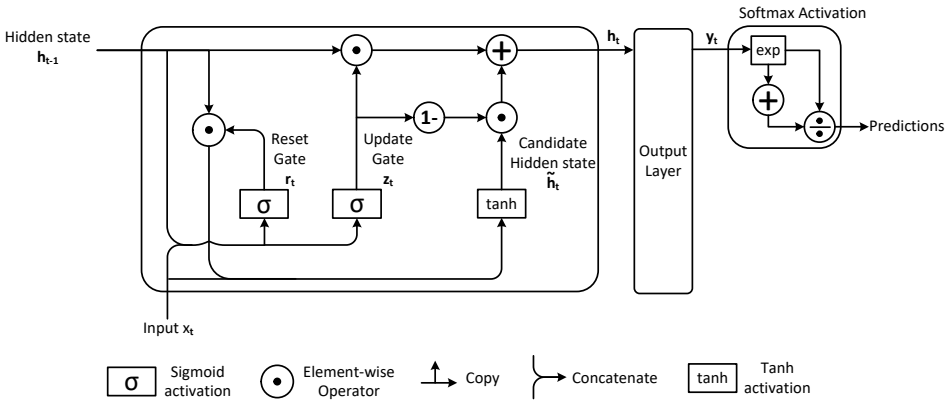

Figure 2.4: Structure of an GRU unit followed by a output layer and a softmax activation function.

2.2 Arithmetic Operations

In order to design an architecture that can efficiently function as an GRU neural network it is important to understand the sequence of arithmetic operations, their number and dependencies. This section goes through the arithmetic operations that formulate the entire algorithm.

Matrix to vector multiplications dominate the number of computations and parameters. For each matrix to vector multiplication, the input vector x_t of size m and the hidden state output vector h_{t-1} of size n are multiplied with weight matrices of size (m + n)×n. That requires (m + n) times n MAC operations, which is equivalent to (m + n) times n multiplications and additions. Since this computation is repeated three times within the GRU computation for reset, update

and hidden state, these numbers are multiplied by three to obtain the total number of MAC operations for an GRU. However this calculation can be broken into smaller operations which can prove valuable when it comes to designing the control of the arithmetic units and exploring design space parallelism.

Every matrix to vector multiplication operation in the algorithm is followed by addition of bias. This requires n more additions.The result of bias addition is saturated by the sigmoid function for reset and update gates. However, for the candidate hidden state tanh function is used.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.7}$$

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.8}$$

Using (2.7) and (2.8), the sigmoid and tanh function implementation requires division and an exponential function which would cost a lot in terms of area, energy and efficiency. In order to avoid this and also reduce execution time, these function can be approximated.

The hidden state h_t computes as (2.4), requires element-wise matrix multiplication. This operation requires additional n adders. Finally, the output layer also requires a matrix to vector multiplication operation increasing the number of such operations to four in the entire computation.More details on this will be discussed in the following chapters.

2.3 Data dependencies

It is crucial to determine data dependencies throughout the entire algorithm. Identifying data dependencies allows parallelization of operations in an ASIC implementation. Figure 2.5 shows data dependencies throughout arithmetic operations in the GRU algorithm. It can be observed that the gate operations r_t, z_t operations can be computed in parallel, independent of each other. However the c_t, h_t computations require the output of gate operations.

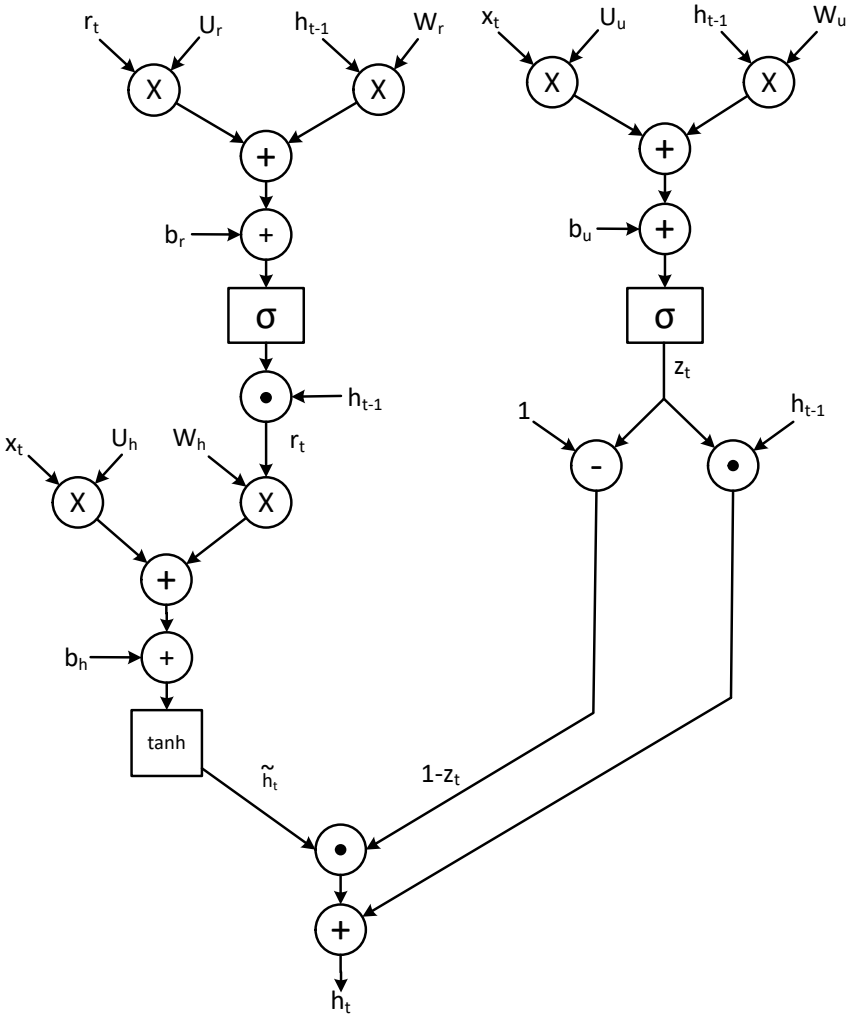

Figure 2.5: Data dependency graph of the GRU algorithm

Chapter 3
Wordlength and Hardware Optimization

GRUs have been successfully developed in broad range of applications including speech recognition [6], [15]. However, storage and computational complexity of these models have forced the majority of computations to be performed on high-end computation platforms or in the cloud. Storing of data such as weights and biases becomes an issue with growing size of the network. Our initial representation of GRU uses 32 bits for representing weights, biases and inputs. Such precision costs in terms of computation and storage complexity, model size and memory access operations such as load and stores.

Therefore, to cope with the computational and storage complexity, solutions such as weight and bias quantization are evaluated in this section along with the impact of reduced wordlength on performance, energy and accuracy. This step includes both high-level simulation, e.g., Python/Matlab, of GRU NN's and evaluation of their accuracy, as well as evaluating different algorithms to implement activation functions (sigmoid and tanh).

3.1 Data representation

An important consideration for implementing an NN is the arithmetic representation format. The difficulty is to achieve a balance between the need for numeric precision, which is important for network accuracy and speed, and the cost of hardware logic associated with increasing precision. While standard floating point representation (i.e 32 and 64 IEEE floating point formats) offer adequate precision, they require more resources than other area efficient arithmetic representations, such as less precise floating and fixed point formats.

Recent studies have empirically shown that 16-bit precision is sufficient to train DNNs without effecting model accuracy [16], [17], and [18]. Therefore, state-of-the-art hardware platform for training are now moving towards 16-bit floating point precision from traditional 32-bit floating point. This is due to reduced storage requirements associated with the reduced precision and high energy efficiency [19], [20]. Furthermore, research on using floating point 8-bit for training are also carried out, demonstrating two to four times speedup without compromising in accuracy [21].

3.1.1 Floating-Point format

A floating point number is represented as $\pm d_0.d_1d_2...d_{p-1} * \beta^e$. Where β is called the base and e is called the exponent and p is the precision. For example, if $\beta = 10$ and p = 4 then the number 0.2 is represented as $2.000x10^{-1}$. The exponent is said to have *biased* representation when the value of the exponent is

$$e = k - (\beta^{m-1} - 1), \tag{3.1}$$

where k is the value of the exponent bits interpreted as an unsigned integer and m is the number of bits in the exponent. The Floating-Point number is said to be normalized number when d_0 is '1'.

One of the most common floating point formats is the IEEE floating point format (IEEE 754-1985 format [22]). In this format, for 32-bit precision numbers, $\beta = 2$, p = 24, m = 8, and e = k - 127. The value of floating point number can be obtained as

$$\pm(d_0 + d_1\beta^{-1} + ... + d_{p-1}\beta^{-(p-1)})\beta^e, (0 <= d_i < \beta). \tag{3.2}$$

The bit representation of this format is illustrated in Figure 3.1.

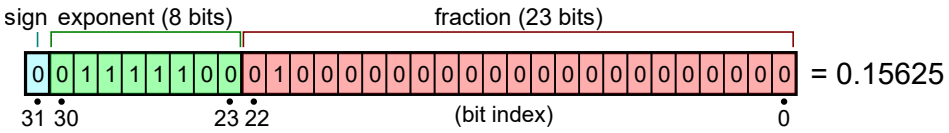

Figure 3.1: Example layout of 32-bit floating point

3.1.2 Fixed-Point format

Figure 3.2 illustrates a Fixed-point representation. It contains two parts, one is the integer part which is b_{ws-1} to b_w and the other part is fractional part which is b_3 to b_0. If the base of the Fixed-Point number is β and it is a positive number, the decimal equivalent value can be calculated by

$$b_{ws_1}\beta^{ws-5} + ... + b_4 + b_3\beta^{-1} + b_2\beta^{-2} + b_3\beta^{-3} + b_4\beta^{-4}. \tag{3.3}$$

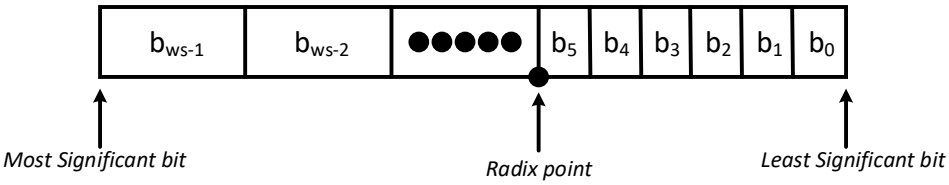

Figure 3.2: Format of a Fixed-Point Number

If the base of Fixed-Point number is 2, the value is determined by which number representation that is used.

- Sign bit representation ranges from $-(2^{(k-1)}-1)$ to $(2^{(k-1)}-1)$, for K bits.
- Signed representation ranges from $-(2^{(k-1)})$ to $(2^{(k-1)}-1)$, for K bits.

Generally 2's complement representation is preferred because of unambiguous property and more convenient arithmetic operations.

Wordlength and number representation of GRU model parameters such as weights, biases, inputs, has a big impact on the overall performance of the GRU inference. A larger wordlength increases the model size, and effects the efficient use of HW resources. By reducing the parameters, word length, more parameters can be read through the same memory interface in a single cycle, thus, decreasing the memory wordlength requirements. Another motivation is that reduced wordlength of the parameters can enable a higher number of operations for almost same hardware costs, since lower precision multipliers require less silicon area and power. The drawback of using a smaller wordlength is potential accuracy decrease of the implementation, explained in following sections.

A reduced floating-point representation could provide lower wordlength parameters with close to no impact on accuracy, since it has a greater dynamic range and precision than fixed-point. However, the drawback of using an equivalent floating-point is that the energy cost increases up to 10 times [23]. Floating-point can provide a higher range for the same wordlength. There is significant amount of research that suggests that for majority of applications, 8-bit fixed-point or lower precision is sufficient, specifically for inference purposes [23].

Another advantage of using fixed-point representation is that the adders and multipliers occupy less area and are more power-efficient since the same adders and multipliers as for integer operations can be reused. Quantizing the fixed-point by powers of two is also considerably cheaper to do than the other touched upon representations, since it only requires arithmetic shift operations.

Our experiments are focused on 16 bit fixed point implementation, described in detail in the following section in terms of higher-level simulations using Matlab and Python.

3.2 Role of the Activation Function

Neural network activation functions are a crucial component of deep learning. Activation functions determine the output of a deep learning model, its accuracy, and also the computational efficiency of training a model. Activation functions are mathematical equations that determine the output of a neural network. The function is attached to each neuron in the network, and determines whether it should be activated ("fired") or not, based on whether each neuron's input is relevant for the model's prediction[24]. Besides restricting outputs to a certain range, activation functions break the linearity of a neural network, allowing it to learn more complex functions than linear regression.The weight multiplication at each layer is a linear transformation. Adding the bias vector makes it an affine transformation. Thus, a many layer neural network would reduce to a single layer network in the absence of non-linear activation functions [25], [26].

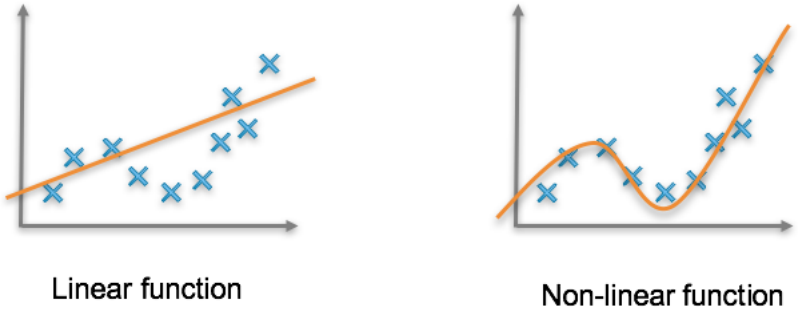

Figure 3.3: Feature selection Linear vs Non linear [27].

A linear equation is simple to resolve, but it is restricted in its complexity and has less power to learn complex functional mappings from data. Thus with an non linear activation function, neural network will be able to learn and model other complicated kinds of data such as images, videos, audio, speech, etc. Also, another important feature of an activation function is that it should be differentiable to perform backpropagation optimization strategy during training [19].

3.3 Activation Function exploration

GRU network uses two activation functions, sigmoid function and tanh function . Sigmoid outputs range from 0 to 1, and are often interpreted as probabilities, as shown in Figure 3.4 and is given by (3.4).

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.4}$$

The tanh function is simply a scaled and shifted version of the sigmoid function, such that its outputs range from -1 to 1, as shown in Figure 3.5. The relationship between sigmoid and tanh is given by (3.5).

$$\tanh(x) = 2 \cdot \sigma(2 \cdot x) - 1 = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{3.5}$$

Due to its symmetric property it's enough to calculate the activation function only for positive arguments x. For negative values of x it can be calculated using

$$F(-x) = 1 - F(x). \tag{3.6}$$

Gated Neural network algorithm uses nonlinear activation functions sigmoid and the hyperbolic tangent (tanh) function to flatten the vectors within range [0,1] and range [-1,1]. Since the reset and update in GRU neural network outputs a value between 0 and 1, it can either let no flow or complete flow of information. The sigmoid function is used as a gating function for the two gates (reset and update gate).

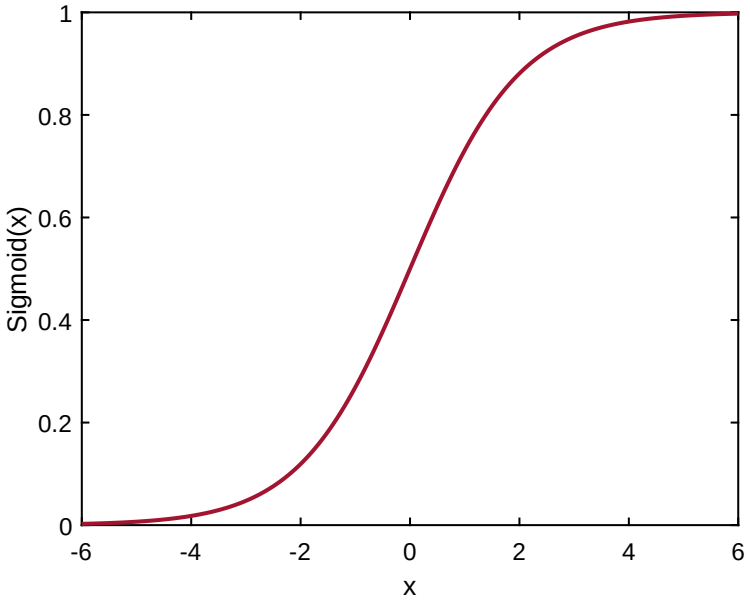

Figure 3.4: Sigmoid function.

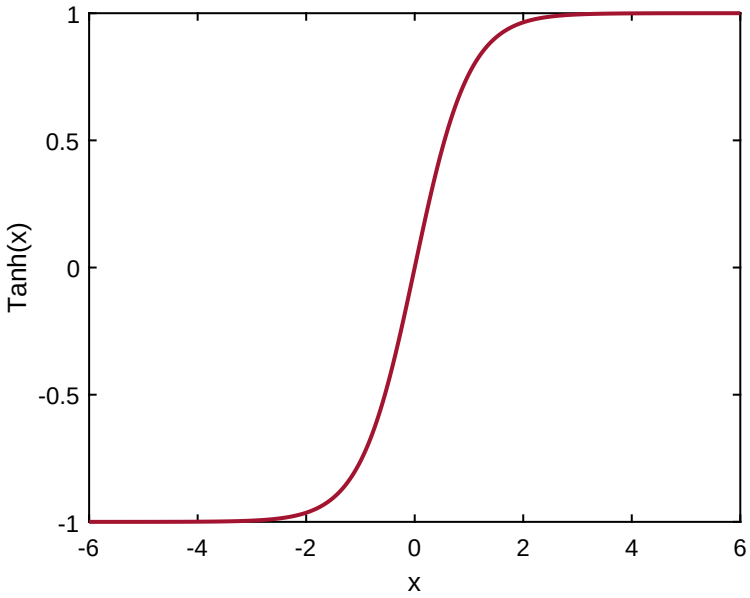

Figure 3.5: The hyperbolic tangent function (tanh).

To overcome the vanishing gradient problem, a function whose second derivative can sustain for a long-range before going to zero is required. Since tanh has the above property, it is used to calculate the candidate hidden state value [28].

Straightforward implementation of the nonlinear activation functions in hardware is very expensive in terms of hardware, as these equations require division and exponent function [29]. Different methods of approximation are used for the hardware implementation of nonlinear activation functions: look up table (LUT), Taylor transformation, piecewise linear approximation, fast sigmoid and tanh approximation.

When decomposing a taylor series, many multiplications and additions need to be performed, therefore, this method is not well suitable for implementation in hardware [30]. LUT is the most commonly used approach to implement the sigmoid and tanh functions. In this approach, the function curve is divided into different segments and the corresponding output values for each input segment are stored in a table. Only one memory access time is required to get the function outputs, thus, LUT is the fastest implementation as compared to other techniques. However, in order to get better accuracy, large storage space are required for LUT implementation [29].

3.3.1 Piecewise linear approximation of the sigmoid function (PLAN approximation)

In this method, the approximation is performed using

$$f(x) = \begin{cases} 1 & x \geq 5 \\ 0.03125 * x + 0.84375 & 2.375 \leq x < 5 \\ 0.125 * x + 0.625 & 1 \leq x < 2.375 \\ 0.25 * x + 0.5 & 0 \leq x < 1. \end{cases} \tag{3.7}$$

Calculations need to performed only for positive input data x. For negative input data x, the sigmoid function is calculated using (3.6). For implementation in hardware the real numbers are converted to integers using multiplication by 2^{10} as

$$f(x) = \begin{cases} 1024 & |x| \geq 5120 \\ 2^{-5} \cdot |x| + 864 & 2432 \leq |x| < 5120 \\ 2^{-3} \cdot |x| + 640 & 1024 \leq |x| < 2432 \\ 2^{-2} \cdot |x| + 512 & 0 \leq |x| < 1024. \end{cases} \tag{3.8}$$

This expression can be implemented in hardware with only shifters and adders, with no multipliers needed. Figure 3.6 and Figure 3.7 illustrates the comparison of sigmoid and tanh approximation, respectively.

The functions are tested by passing inputs ranging from -6 to 6 in steps of 0.001. The sigmoid PLAN approximation has relative error of 0.032 and tanh has relative error of 0.151. Since PLAN approximation algorithm used is optimised for sigmoid approximation, approximating of tanh using (3.5) introduces error (spikes) as shown in Figure 3.7.

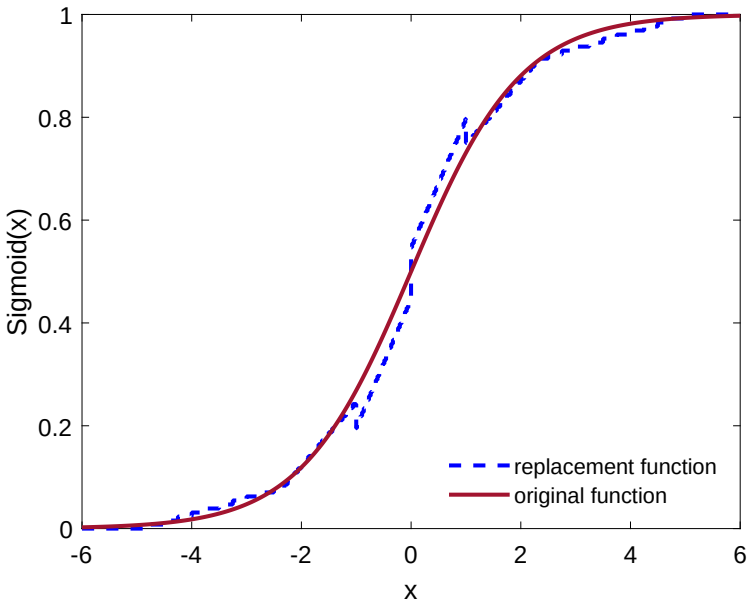

Figure 3.6: PLAN approximation of sigmoid function.

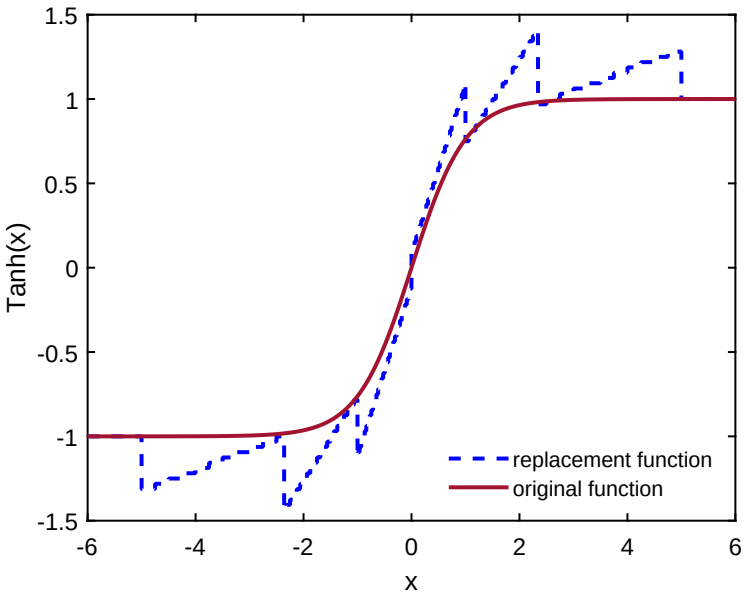

Figure 3.7: PLAN approximation of hyperbolic tangent function (tanh).

3.3.2 Taylor Series Implementation

The Taylor series expansion of the sigmoid and hyperbolic tangent function are computed as

$$\sigma(x) = \frac{1}{2} + \frac{x}{4} - \frac{x^3}{48} + \frac{x^5}{480} - \cdots \tag{3.9}$$

$$\tanh(x) = x - \frac{x^3}{3} + \frac{2x^5}{15} - \frac{17x^7}{315} + \cdots \tag{3.10}$$

Implementing these polynomials is not feasible for hardware as it is complex and costly in terms of area. Therefore, this approach was not evaluated.

3.3.3 Second order curve approximation

In this method, the approximation is performed using

$$f(x) = \begin{cases} 1024, & x \geq 4096 \\ -2^{-15} \cdot x^2 + 2^{-2} \cdot x + 512 & 0 \leq x < 4096. \end{cases} \tag{3.11}$$

Calculations must be performed only for positive input data x. For negative input data x, the sigmoid function is calculated using (3.6). For implementation in hardware the real numbers are converted to integers using multiplication by 2^{10}.

$$f(x) = \begin{cases} 1, & x \geq 4 \\ -0.03125 \cdot x^2 + 0.25 \cdot x + 0.5 & 0 \leq x < 4. \end{cases} \tag{3.12}$$

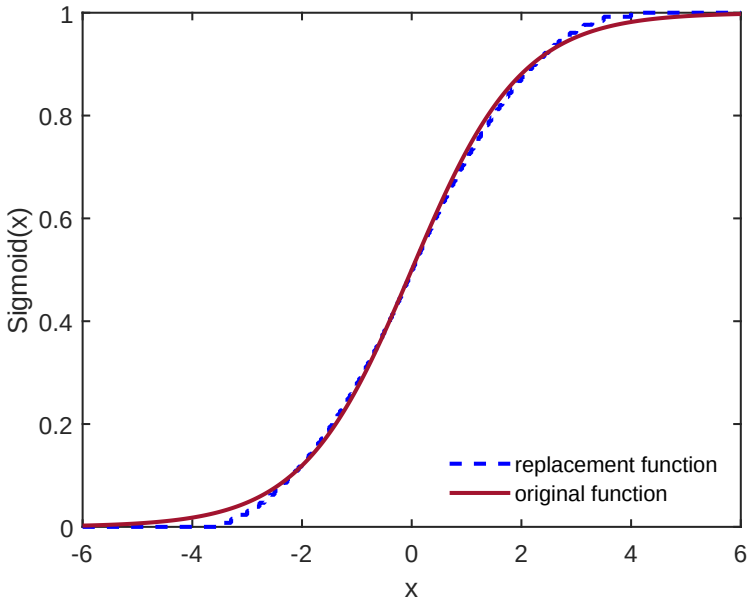

Figure 3.8: Second Order approximation of sigmoid function

Inputs in the range (-6, 6) are divided in steps of 0.01. In this range the maximum and average relative error of approximation of sigmoid function were 0.029 and 0.020, respectively. The hyperbolic tangent function is a scaled and shifted version of sigmoid function, its implemented using (3.5).

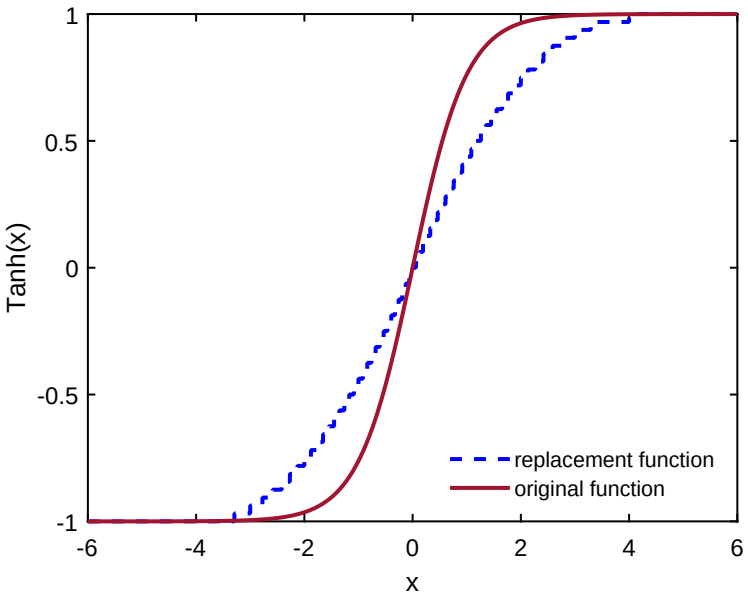

Figure 3.9: Second Order approximation of hyperbolic tangent function (tanh).

The maximum error and average relative error of approximation of tanh function were 0.357 and 0.121, respectively. Implementation of second-order polynomial on HW requires one multiplier, shift registers, and adders.

3.3.4 Fast Sigmoid and Hyperbolic Tangent function

Sigmoid function can be approximated as

$$f(x) = \begin{cases} 1 & x \geq 4 \\ (0.5 * x)/(1 + x) + 0.5 & 0 \leq x < 4 \\ 1 - ((0.5 * x)/(1 + x) + 0.5) & -4 \leq x < 0 \\ 0 & x \leq -4. \end{cases} \tag{3.13}$$

From (3.5), hyperbolic tangent function can be written as

$$f(x) = \begin{cases} 1 & x \geq 4 \\ 2 * ((0.5 * x)/(1 + x) + 0.5) - 1 & 0 \leq x < 4 \\ 2 * (1 - ((0.5 * x)/(1 + x) + 0.5)) - 1 & -4 \leq x < 0 \\ 0 & x \leq -4. \end{cases} \tag{3.14}$$

$$f(x) = \begin{cases} 1 & x \geq 4 \\ 2 * ((0.5 * x)/(1 + x) + 0.5) - 1 & 0 \leq x < 4 \\ 2 * (1 - ((0.5 * x)/(1 + x) + 0.5)) - 1 & -4 \leq x < 0 \\ 0 & x \leq -4. \end{cases} \tag{3.15}$$

Figure 3.10 and Figure 3.11 illustrates the comparison of original and replacement functions for sigmoid and tanh activation, respectively. The average relative error of sigmoid and tanh function were 0.07 and 0.17, respectively.

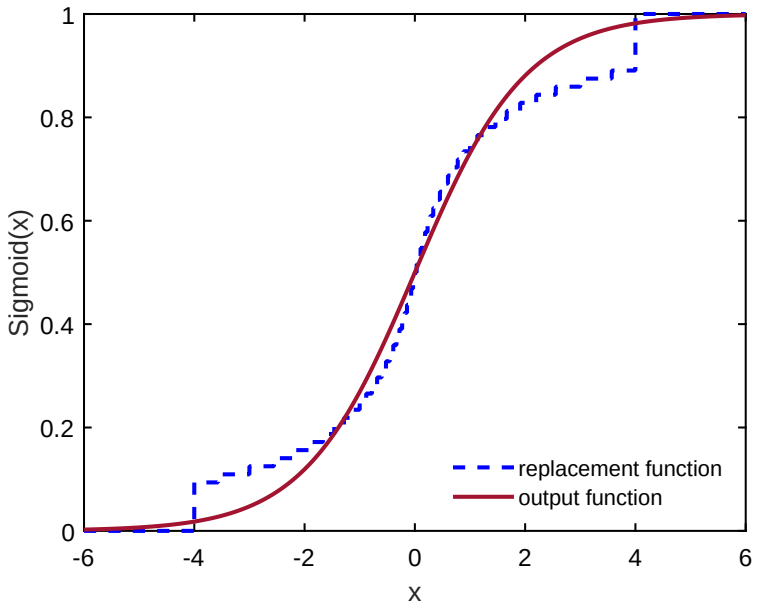

Figure 3.10: Fast sigmoid function approximation

The fast sigmoid, and fast tanh function were implemented using adders, shifters, and division operation can be performed using CORDIC division algorithm . A CORDIC division algorithm is based on re-writing the equation $z = x/y$ into the form $x - yz = 0$ [31]. The expanded series representation of z is expressed as (3.16), which can be simplified to (3.17).

$$x - y * \sum_{i=1}^{B} a(i)2^{-i} = 0 \tag{3.16}$$

$$x - \sum_{i=1}^{B} a(i)(y2^{-i}) = 0 \tag{3.17}$$

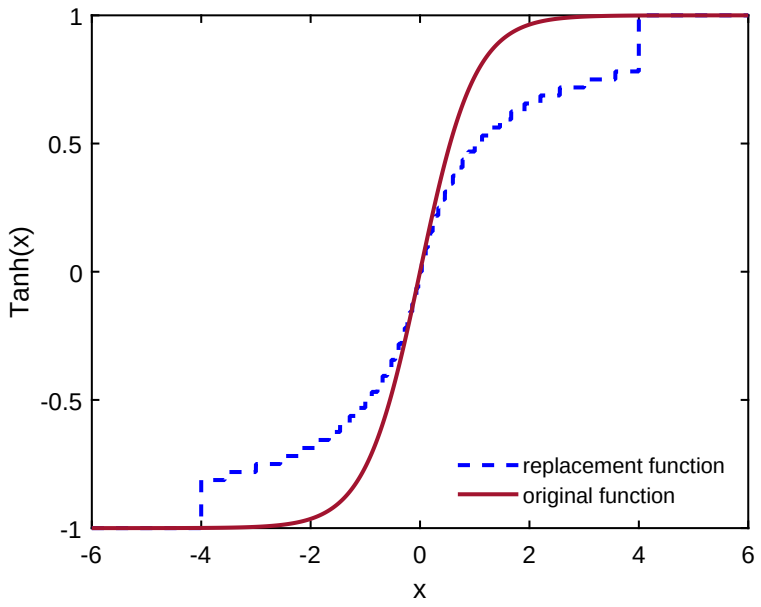

Figure 3.11: The Fast hyperbolic tangent function approximation

Using (3.17), the quotient z is estimated one bit at a time by driving x to zero using right shifted versions of y. If the current residual is positive, the i^{th} bit in z is set. Likewise if the residual is negative, the i^{th} bit in z is cleared.

```
divide(x,y){
   for (i=1; i=<B; i++){
      if (x > 0)
         x = x - y*2^(-i);
         z = z + 2^(-i);
      else
         x = x + y*2^(-i);
         z = z - 2^(-i);
   }
   return(z)
}
```

Listing 3.1: Cordic algorithm 1 [31].

While x may be either positive or negative, y is always assumed to be positive. As a result, the division algorithm is only valid in two quadrants. Also, if the initial value for y is less than the initial value for x it will be impossible to drive the residual to zero. This means that initial y value must always be greater than x, resulting in domain of $0 < z < 1$. The algorithm may be modified as follows for four quadrant division with $-1 < z < 1$.

```
divide_4q(x,y){
   for (i=1; i=<B; i++){
      if (x > 0)
        if (y > 0)
```

```
            x = x - y*2^(-i);
            z = z + 2^(-i);
         else
            x = x + y*2^(-i);
            z = z - 2^(-i);
      else
         if (y > 0)
            x = x + y*2^(-i);
            z = z - 2^(-i);
         else
            x = x - y*2^(-i);
            z = z + 2^(-i);
   }
   return(z)
}
```

Listing 3.2: Cordic algorithm 2 [31].

Since multiplications are all powers of 2, they are realized using shift operations; no actual multiplier is needed. Hence, it is well suited to hardware implementation, simplifying overall hardware complexity.

Due to the nature of fast sigmoid given by (3.14), the value of y is never less than the initial value of x. Therefore, CORDIC algorithm mentioned in Listing 3.1 was used. The fast sigmoid and fast tanh approximation technique has been chosen for implementation for this thesis.

3.4 Quantization Experiments

The motivation behind quantization and wordlength optimization is that NNs are typically trained with floating-point weights and activations. In various cases, quantizing a model trained with a 32-bit floating-point into eight-bit fixed-point, without any re-training, can result in a moderately low accuracy loss, which can be improved by further fine-tuning [32].

Quantizing a model without re-training is referred to as post-training quantization. However, publications have shown that there are cases where post-training quantization to eight-bit fixed-point effect accuracy [33]. Smaller models such as MobileNet does not respond well to post-training quantization, likely due to their smaller representational capacity. As discussed above, fixed-point weights are sufficient to run NNs without any significant accuracy loss. Edge devices have memory constraints that motivate the need for quantization of 32-bit floating-point weights and biases to eight-bit fixed-point weights and biases. Also, using an eight-bit fixed-point consumes four times less overall bandwidth and the model size compared to 32-bit floating-point. Furthermore, fixed-point integer operations are more energy efficient than floating-point operations as shown in Table 3.1.

Table 3.1: Energy and area comparison [34].

INT8 Operation	Energy Saving vs FP32	Area Saving vs FP32
Add	30x	116x
Multiply	18.5x	27x

A basic quantization of fixed-point numbers is shifting the values until the MSBs fit within the specified word length. The range of fixed-point is determined by -2^{i-1} to $2^{i-1} - 2^{-f}$. For instance, if the wordlength is 16 bits and the number of both integer i and the fractional f bits is eight, its range is -128 to 127. If the number lies out side this range, it is truncated by a specific number of positions. If the number is very small, e.g., 0.00034 and cannot be represented by the available fractional bits in fixed-point representation, number get eradicated to zero. Therefore, the chosen number of integer and fractional bits significantly influences the outcome, and it should be adjusted to fit the specified dataset. For values of weights $w \epsilon [0, 1]$, reserving most of the fractional bits during quantization gives the highest precision.

3.4.1 High-level model (Accuracy evaluation)

Implementing pretrained KWS GRU NN with 32-bit parameters is not efficient approach. To compress the neural network model, the number of bits required per parameter (weights and biases) was reduced by truncating the fraction bits. The MATLAB model of GRU fixed point implementation was used to perform experiments. The accuracy of the fixed-point model remained constant untill 28-bit representation, dropped by one percentage unit at 24-bit representation, and two percentage unit at 16-bit representation as illustrated in Figure 3.12. This section will focus on 16-bit representation and below. The accuracy of the fixed-point model is compared with accuracy of 32-bit full-precision floating-point implementation having a accuracy of 91%. Within 16-bit fixed point representation, the most accurate results were achieved with the Q10.6 format. Since, input ranges from -247 to +90, minimum nine integer bits are required.

Table 3.2: Accuracy of GRU network with different precision.

GRU network precision	Accuracy
32-bit floating point	91.1%
32-bit fixed-point	90%
28-bit fixed-point	90%
24-bit fixed-point	89%
16-bit fixed-point	87.7%
14-bit fixed-point	80%

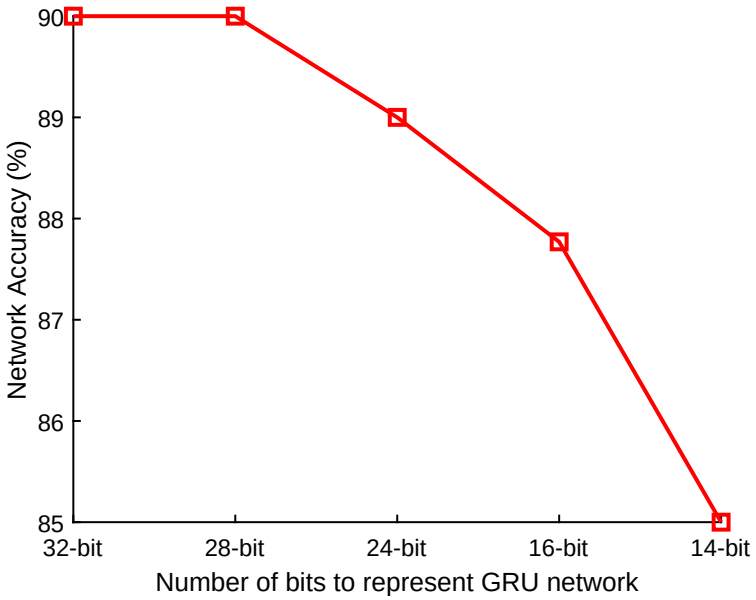

Figure 3.12: Network accuracy for GRU network with different precision bits to represent the network.

After selecting a 16-bit fixed-point representation for GRU network, a second experiment was conducted where network parameter (weights and biases) bits were truncated from 16-bit to 6-bit representation. The accuracy of the network remained constant until 10-bit representation, a shoulder was noticed at 8-bit representation as illustrated in Figure 3.13. Also, fluctuation in the network accuracy was observed just before the shoulder. To reduce the precision of the network, fractional bits are truncated. Table 3.3 shows the accuracy of GRU NN with varying network parameter precision.

Table 3.3: Accuracy of 16-bit GRU network with different precision bits for parameters.

Network parameter precision	Accuracy
10-bit fixed-point	87.7%
9-bit fixed-point	86.6%
8-bit fixed-point	87.7%
7-bit fixed-point	85.5%
6-bit fixed-point	83.3%

Since the accuracy was identical with eight bits as network parameter precision, further experiments were conducted using 16-bit GRU network with 8-bit parameter precision.

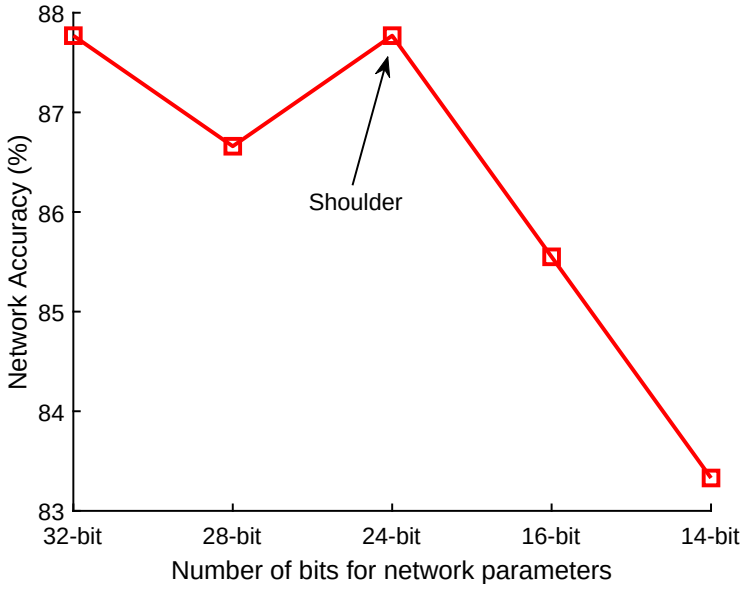

Figure 3.13: Network accuracy for different network parameter precision for fixed-point implementation of 154x154x154x12 topology.

Figure 3.14 depicts the accuracy plot for different input precision. Figure 3.15 illustrates the accuracy plot when the MAC output precision is kept same as input precision, shoulder was noticed at 13-bit representation. MAC output precision here refers to the wordlength used for truncating the output of the MAC unit. Table 3.4 and Table 3.5 show the accuracy of various combinations of input and MAC output precision used.

The shoulder in the graph refers to a point, after which there is a sudden drop in the accuracy.

Table 3.4: Classification accuracy for different input data precision.

Input precision	Accuracy
16-bit fixed-point	87.7%
15-bit fixed-point	87.7%
14-bit fixed-point	87.7%
13-bit fixed-point	87.7%
12-bit fixed-point	85.5%
11-bit fixed-point	80%

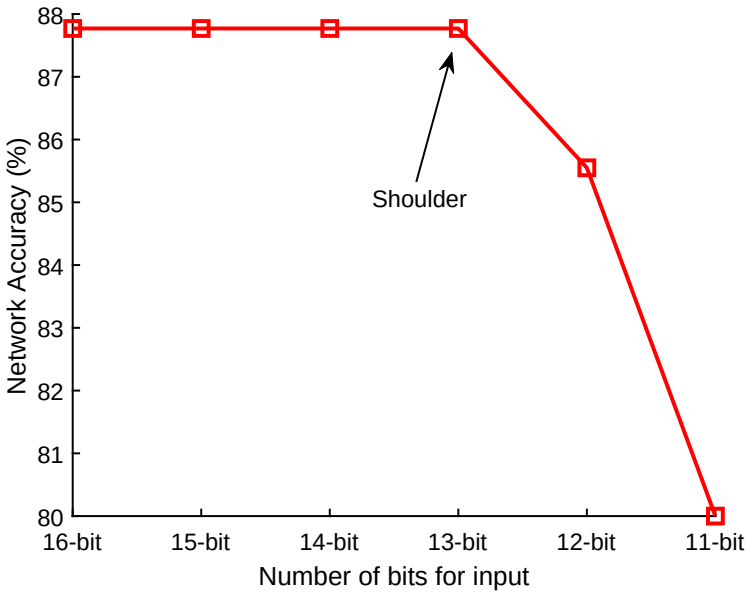

Figure 3.14: The figure shows the accuracy for different input data precision.

When the MAC output precision was truncated with the same precision as input, shoulder was observed at 14-bit representation. From the above experiments, it can be concluded that reducing the MAC output precision leads to accuracy loss, and using 16-bit fixed point network with 8-bit weights and biases produced the best accuracy at slightly higher hardware cost.

Table 3.5: Classification accuracy for different input precision based KWS Tensorflow dataset.

Network precision	Accuracy
16-bit fixed-point	87.7%
15-bit fixed-point	85.5%
14-bit fixed-point	81.1%
13-bit fixed-point	70.0%
12-bit fixed-point	45.5%

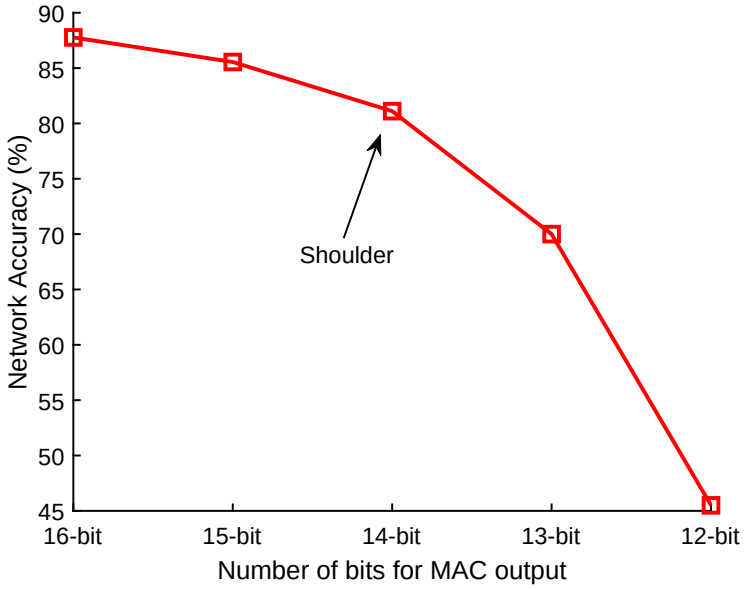

Figure 3.15: The figure shows the accuracy for different input data precision and MAC output precision at Q10.6 format.

Figure 3.16 illustrates the raw predictions with the chosen wordlength, i.e., 16-bit for representing GRU fixed-point model and eight-bit for representing weights and biases. All the experiments are carried out without retraining the network after performing the truncation. Findings might be different if the network is re-trained after performing the truncation of network parameters, which might recover some of the accuracy loss introduced due to truncation.

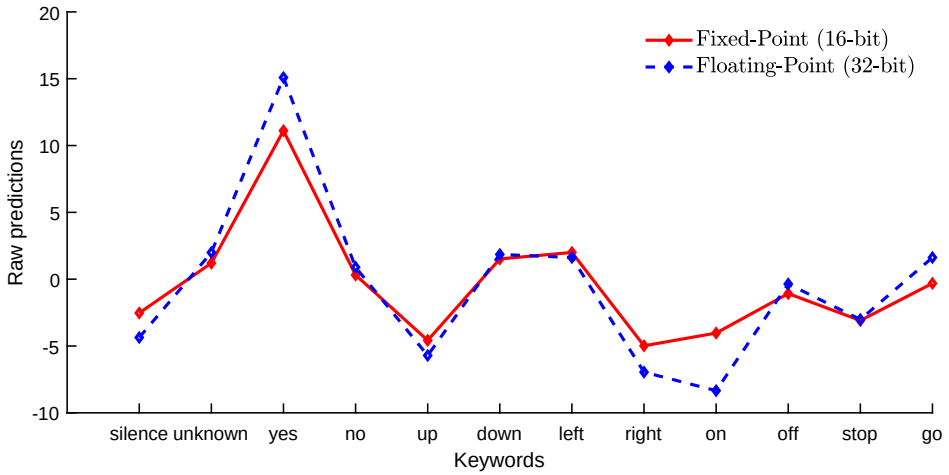

Figure 3.16: Raw prediction of the fixed-point implementation (red line) against the 32-bit floating-point implementation (blue line).

Chapter 4

xDSP Implementation

The xDSP is a 3-stage pipeline processor optimized for DSP applications with multiple data-paths, register files, parallel instructions and custom functional units. The sections below provide a high-level overview of the architecture of the xDSP processor.

4.1 Register Files

The XDSP contains three register files: one for the integer datapath (R), one for the scalar datapath (S), and one for the vector datapath (V). To allow a greater flexibility, the scalar datapath also has access to the R register file, and the vector datapath has access to the R and S register files.

Table 4.1: Register files.

Register file	Data type	Integer size	Fixed point format
R	Integer	16 bits	N/A
S	Scalar	24 bits	{5.19} bits
V	Vector	4 x 24 bits	4 x {5.19} bits

4.2 Datapath

The XDSP contains the following datapaths:

1. Integer datapath - supports arithmetic, logic, shift, and compare instructions. Also handles the program control flow. This datapath can only access the 16-bits R register file.
2. Scalar datapath - is for data processing in both fixed point and integer format. This datapath also supports more complex operations specifically targeted to DSP applications and allows access to both R and S register files.
3. Vector datapath - can process four scalars in parallel. This datapath can access all the register files.

4. Multicycle datapath - for operations that cannot be completed in a single clock cycle.

The main bottleneck for processing NN is the memory access. Each MAC operation requires three memory reads (input, weight and bias). Hence to reduce the memory access operations and increase the speed of computation the vector datapath is used.

4.3 Memory interfaces

The xDSP utilizes a 24-bit data bus for scalar/integer datatypes and 96 bit data bus for vector data. The memory can be accessed with a 16-bit address, which are stored as sign extended 24 bit variables. All of the vector data memory accesses are performed as single cycle operations, while the scalar memory accesses depend on the Network-on-Chip (NoC) arbitration delay.

The program memory consists of variable length instructions (8/16/24/32 bit) and is packed in 8-bit blocks. The processor uses a 16-bit address format. Hence it can address a maximum 64 kB of program memory. The program memory interface is a 64-bit data bus and uses a 128-bit program memory cache for instruction fetching.

4.4 Design considerations

The XDSP uses 24 bits to represent fixed point data. To implement KWS GRU model, 230KB of memory storage is required to store the network parameters, inputs, weights and biases.

The data memory size available on XDSP is limited, which constraints the size of NNs that can be run on the XDSP. Thus, three small GRU NN with 4, 20, and 40 neurons in each layer were chosen for implementation on XDSP. These network were trained using the Python script provided with the KWS model. The trained smaller models occupy 1.5kB, 7KB, and 21KB of the data storage available on XDSP, respectively. Table 4.2 shows the number of entries required by GRU NN parameters (inputs, weights and biases).

Table 4.2: Number of entries for different parameters in GRU NN.

Network parameter	Entries			
	4 neurons	20 neurons	40 neurons	154 neurons
Weights	180	2040	6480	77616
Biases	15	72	132	474
Inputs	10	10	10	10
Total	205	2122	6622	78100

The xDSP has a separate datapath for vectorized data, see section 4.2. To use this feature, weights and initial inputs are loaded into the memory as matrices in the vectorized format such that four multiplications are executed in parallel.

4.5 Implementation

This section introduces basic concepts that are needed in order to understand design decisions, wordlength used, and bias alignment in registers.

4.5.1 Multiply-accumulate operation

The Vector double precision MAC functional unit available on XDSP is used for implementing matrix multiplication operation performed in different layers of GRU network. This function executes in two clock cycles.

The MAC operation multiplies two input vectors, adds the results together and then stores the sum in an accumulator. The accumulator uses a vector type and is interpreted as two custom fixed-point values, *high* and *low*, stored in either the high (accu[3:2]) or low (accu[1:0]) part of the vector. Each part holds the accumulated values in extended precision of 48 bits (Q18.30). The accumulated value has 8 guard bits, which also means that the lowest 8 bits are discarded after multiplication.

The *hi* and *lo* in Figure 4.1, return data in Q18.6 format, and Q0.24 bit format, respectively. Together, they represent a number in Q18.30 format. The result of the MAC operation in extended precision (Q18.30) is then scaled back to the Q5.19 format.

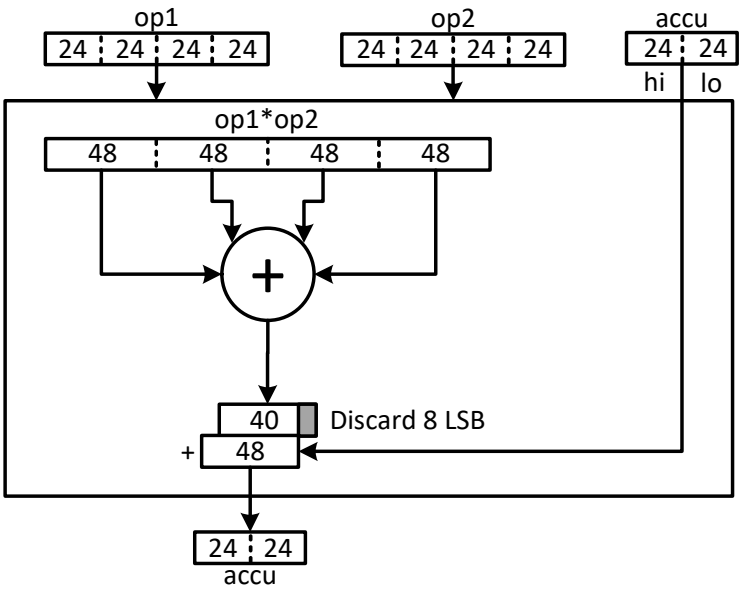

Figure 4.1: MAC unit.

With this MAC unit, only one neuron output can be computed at a time as the four multiplication results (and the subsequent ones) are added together.

4.5.2 Bias alignment

As mentioned in the above section since computations are executed for one neuron at a time, bias values are loaded from the memory as regular scalar values in Q5.19 format unlike weights and initial inputs. Hence, loading the bias cannot exploit the advantages of the vectorized data format.

However, before the MAC operation is performed, the accumulator is initialized with the bias value to save one addition operation that would otherwise have to be performed after completing all MAC operations per neuron. Loading the bias into the accumulator requires an alignment step since the accumulator stores results in an extended double-precision Q18.30 format as mentioned in section 4.5.1, while the bias value is in Q5.19 format.

In Figure 4.2 and Figure 4.3, accu[1] interprets data in Q18.6 format, therefore bias values are by default shifted 13 positions right and loaded into the accu[1]. This step ensures, that all the integer values and first six fractional values are stored correctly. Then the bias value is shifted by eleven positions left in order to extract the remaining 13 fractional bits that have to be loaded into the accu[0], which interprets data in Q0.24 format.

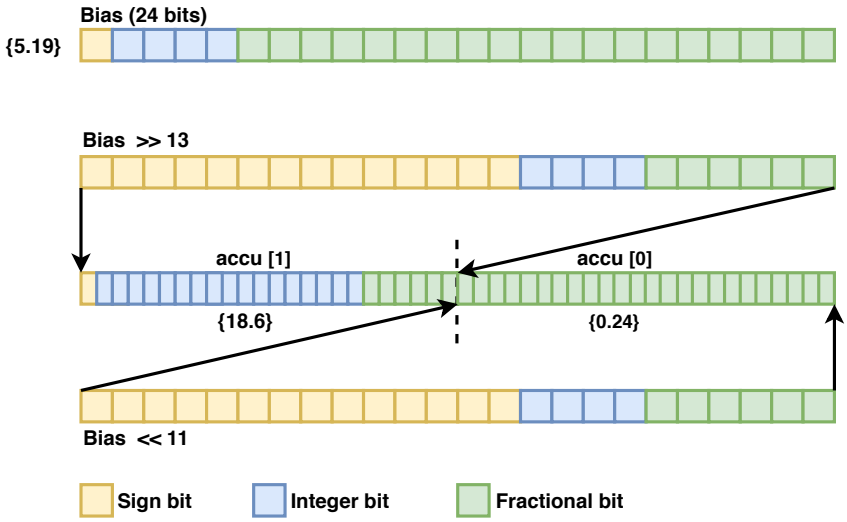

Figure 4.2: Illustration of steps executed to load the bias value into accumulator.

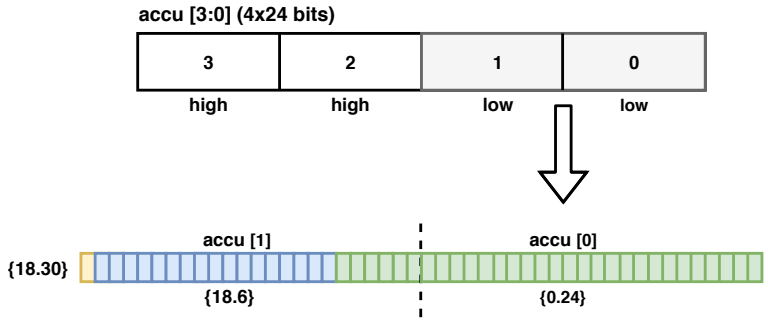

Figure 4.3: 96-bit accumulator divided into low and high part.

4.6 Result

The main purpose of the XDSP implementation is to extract metrics for establishing a baseline energy cost for inference, the memory footprint and latency, which is later used for comparing with ASIC implementation. After the network with 4 neurons per layer and 40 neurons per layer was implemented, all the instructions belonging to the GRU inference are extracted from the execution flow of the program. This was accomplished by adding two breakpoints instructions to indicate start and stop of the inference. Instructions between the breakpoints are then responsible for the inference related operations. This helps in identifying instructions strictly involved with the GRU computations. The total cycle count of the inference was calculated by adding the number of times each instruction was called at runtime.

The data memory size available in xDSP as mentioned in section constrains the size of the GRU NNs run on the xDSP. Therefore, the biggest Network size chosen for inference on xDSP was with 40 neurons per layer. The memory footprint of this network is 20.1 kB, with weights occupying major portion of the memory space as shown in Table 4.2. The memory footprint of the smaller network with four neurons per layer chosen for inference is 1.4kB. The metrics results of the two networks are used for obtaining a scaling factor which is later used for approximating energy dissipation for a network with 154 neurons per layer.

Table 4.3 shows the number of instruction calls for the MAC and load/store instructions. LD_v, LD_s and LD_i are vector load, scalar load and integer load instructions respectively. The LD prefix and ST prefix stands for load instructions and store instructions, respectively.

Table 4.3: Instruction calls per inference for the network with 4, 20, and 40 neurons per layer.

Network	Neurons per layer		
	4	20	40
Instruction	Calls		
LD_v	4963	40871	126616
LD_s	2007	7027	17452
LD_i	2645	8420	18087
LD Total	9615	56318	162155
ST_v	2137	15258	44075
ST_s	1533	6030	15351
ST_i	767	3849	7724
ST Total	4437	25137	67150
MAC	1204	12060	39210
Others	30984	129495	331190
Total	46239	223010	560495
Total cycles	**40898**	**195098**	**492150**

The MAC unit has four 24-bit multipliers and four 48-bit adders. The memory accesses instructions (Load/Store operations) are for 16, 24 or 96 bit words as described in section 4.1. The "Other" instructions relate to branches, jumps, NOPs, shifts, etc. It is important to note here that the Table 4.3 do not include the instruction fetch and instruction decode stages, but only the execution stage.

The GRU NN grows both in x and y dimension, where inputs and the number of neurons contributes to x dimension and only neurons contribute to y dimension. From Table 4.2, it can be observed that weights occupies the major portion of the entries and grows quadratically with number of neurons. By plotting a second order extrapolation, a scaling factor of ten was estimated when network size row from 40 neurons per layer to 154 neurons per layer. The the clock cycles for the 154 neurons per layer network was also estimated from second order extrapolation plot.

Figure 4.4 illustrate required clock cycles to compute an inference. The vertical bar shows the amount of clock cycles required and the horizontal line depicts the maximum clock cycles to comply with real time requirement of 40 ms.

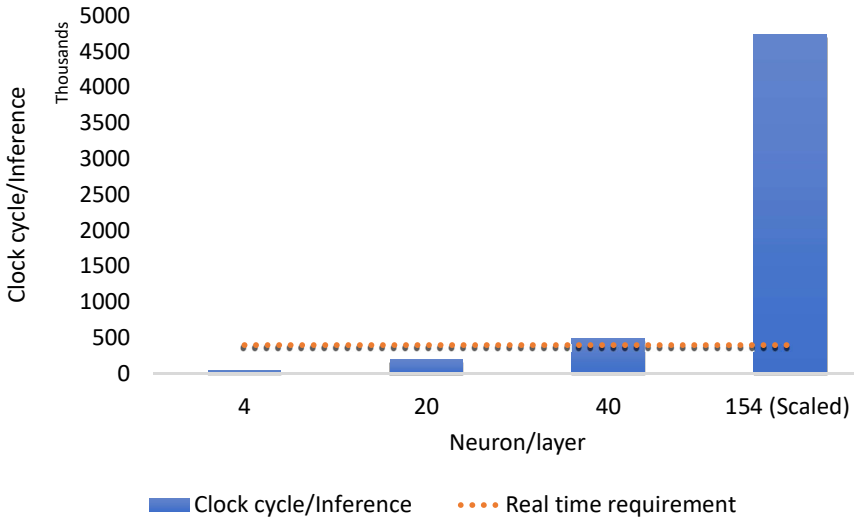

Figure 4.4: Clock cycles required per inference of GRU NN.

4.7 Discussion

There is a real time requirement of 40 ms to do computation as mentioned in section 1.4. From Figure 4.4, it can be observed that GRU NN with upto 40 neurons per layer can be computed on xDSP. However, xDSP is not designed to compute large-scale GRU NN.

The full implementation, i.e., executing GRU network with 154 neurons per layer was not possible due to memory limitation. There is a possibility of executing full implementation by packing the network parameters, which will reduce the number of bits required per parameter to half. Unpacking of the data is required while execution. However, packing/unpacking of the data adds extra instructions in the program memory, which can exceed the program memory space available.

Chapter 5
Dedicated Neural Network Engine

This chapter covers the different aspects related to the design of a dedicated GRU neural network engine (GNNE) and its register-transfer level (RTL) implementation using SystemVerilog. The motivation for developing a dedicated GNNE is to improve the energy efficiency required to perform inference and explore optimization possibilities that can not be exploited using audio digital signal processor (DSP).

Due to the complexity of the entire design, it will be broken down in separate sections following the order they were designed during the development stage. The following sections provide a detailed description of the proposed architecture and implementation of GNNE, followed by behavioural and RTL simulations using KWS application input data set. The GRU algorithmic flow is also discussed in order to understand the optimizations performed.

5.1 Design Considerations

There is a number of things to consider before designing an ASIC implementation for computational units of GRUE.

- Matrix multiplications and element-wise multiplication operation require a great number of multiplications and additions. Hence it is necessary to provide sufficient amount of multipliers and adders that operate in parallel depending on throughput requirement, and reduce the time it takes to produce a final result.
- The final design needs to be flexible and capable to adapt for different network sizes.
- Reduce the number of memory read operations while streaming the data from memory to computational units.

To address the first two considerations, the architecture need to provide the possibility of parameterized instantiation which allows larger execution blocks that provide more throughput, such that it can be changed to fit the needs of the user. The default value is set to eight for adders and multipliers, which is used throughout this thesis. The data feed to the multiplier and adder unit must be structured such that improves latency. As the entire multiplier and adder unit is working with vectors to increase throughput and parallelize the design. Instead of

sending and receiving one element of a vector at a time the memory is designed to send and receive vectors of size eight. Hence, memories used in the design are required to store a vector of eight elements at a address to reduce the memory read operations which also increase throughput. Furthermore, the data flow in entire design is considered in vector form.

5.2 Overview

The architecture is parameterized with number of integer and fractional bits. The default is set to 16-bit fixed-point representation (Q10.6 format) with some exceptions which will be explained in sections below.

NN parameters such as weights and biases contributes to the biggest footprint in NN applications. Therefore, data movement far from engine will be more costly in terms of energy and computations. Due to this reason GNNE is implemented with two dedicated memory types, one is SRAM and another is a scratch pad memory. Scratch pad memory is used because memory storage required for intermediate results is very small, where SRAM would be area inefficient. Furthermore, simultaneous read and write operation are performed, which will would require more instances of SRAM or a dual port SRAM. Initially, the main memory (SRAM) needs to be populated with data (weights and biases) before the inference begins. A standard-cell based scratch pad memory is used to store the temporary data which changes at every time step. The GNNE specific memory layout is further descried in section 5.7. The input data loading operation is performed by the xDSP processor or direct memory access (DMA). There are four interfaces to communicate with the engine:

1. **APB_Interface** - The Advance Peripheral Bus interface is used for initial setup of configuration parameters such as start/stop addresses for weights/biases, different segments of scratch pad memory, and iteration for CORDIC module.
2. **Clkpmrst_Interface** - The Clock/Reset interface is used for driving the clock and reset signals for all the modules in the GRUE.
3. **Mem_Interface** - The memory interface is used for reading the data from the memory.
4. **SCMem_Interface** - The scratch memory interface is used to read/write data to the scratch pad memory.

The next sections will go through the design steps, optimizations and describe the data flow to give basic understanding of the design.

5.3 Top module

The modules defined in the previous sections are instantiated in the *nn_gru_top* module. The top module forwards the data received via the three interfaces, i.e., apb_interface, clkpmrst_interface, mem_interface to all the instantiated submodules. Besides functioning as a wrapper module, the top module performs

logic necessary for producing the output prediction of keyword spotted along with the max value. Since the KWS GRU pre-trained model is trained for 12 keywords, see section 1.4, output layer produces 12 raw predictions. These 12 values are compared with each other and the maximum value is stored along with its index. The index of the maximum value is the prediction (label of the keyword detected).

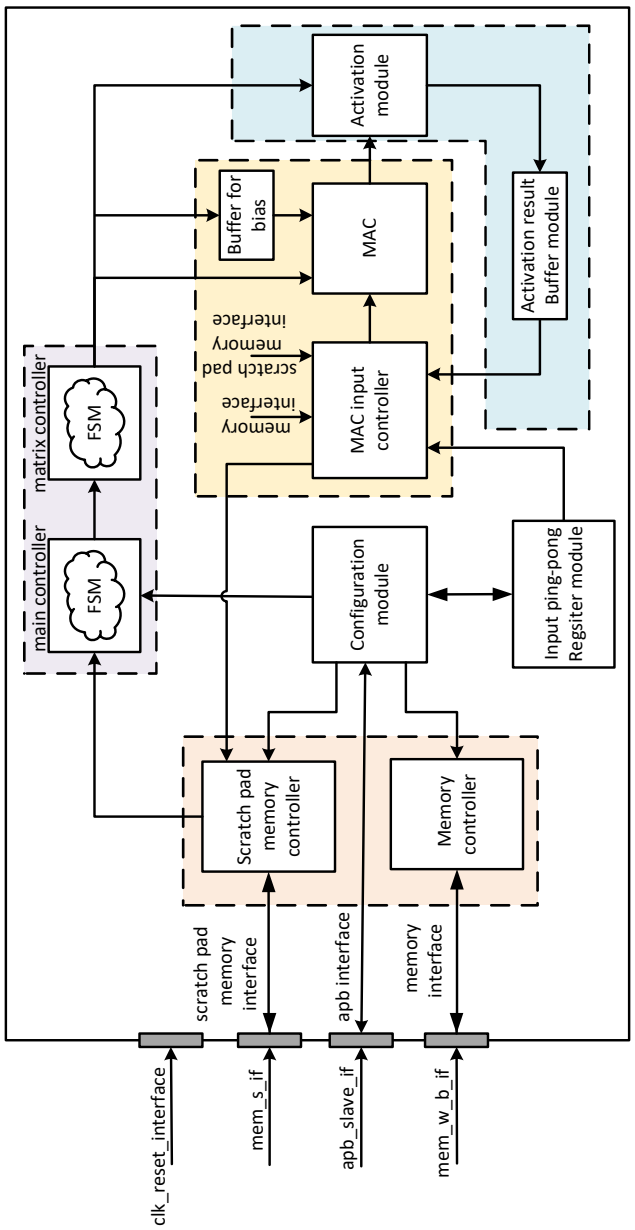

Figure 5.1: Top level architecture of the design.

5.3.1 Data Flow

Without optimizing the architecture a straightforward implementation of GRU algorithm would require multiple multiplier banks and adder banks to perform matrix multiplication, element-wise multiplication and addition operations across different layers shown in Figure 5.2.

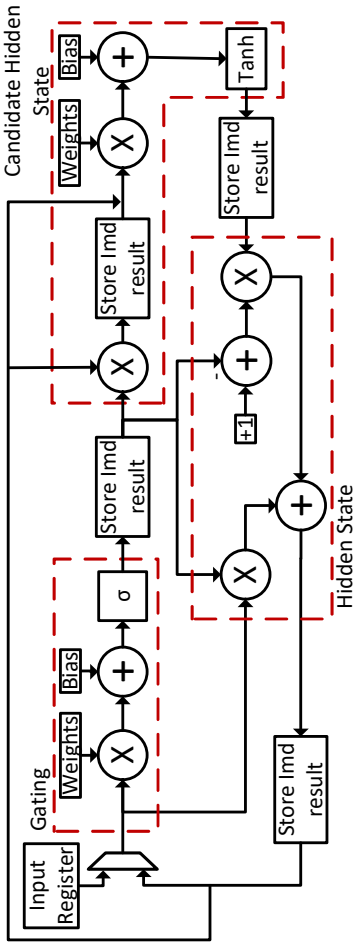

Figure 5.2: Data Flow of GRUE without optimizations.

However, inference runs on a per layer basis, i.e., the first layer output acts as input to the second layer and so forth. At a particular time only one multiplier and adder bank is used for computation, this provides the opportunity to reuse the hardware to perform multiplication/addition across different layers. Thus, by reusing the hardware it is possible to achieve significant reduction in area and power which is crucial for hardware implementations in power-constrained systems.

An implementation as described in Figure 5.2 requires multiple scratch pad memories to store the intermediate data computed. Upon completion of a layer, the

intermediate data (output of previous layer) is passed to the computational units in the next layer, thus, repeating the process until all layers have been evaluated. As the network size grow larger, the amount of intermediate data that must be streamed between the compute units increases, in turn increasing the memory capacity needed.

The algorithmic flow can be understood using Matlab pseudo code shown in Listing 5.1.

```
%% Gating
concatenate1 = [input prev_h];
gating = sigmoid(concatenate1 * gate_weights + gate_bias);
%% Split
reset_gate = gating(1:154);
update_gate = gating(155:end);
%% Candidate hidden state
r_state = reset_gate .* prev_h;
concatenate2 = [input r_state];
candidate_hidden = tanh(concatenate2 * candidate_weights +
    candidate_bias);
%% Hidden state
new_h = update_gate .* prev_h + (1 - update_gate) .*
    candidate_hidden;
prev_h = new_h;
```

Listing 5.1: Matlab pseudo code for GRU NN

The gating stage illustrated in Figure 5.2 computes reset and update gate given by (2.1) and (2.2). The reset gate result, r_t, is multiplied with h_{t-1}, denoted r_state in Listing 5.1. Input x_t is concatenated with r_state and multiplied with weights W_h. After adding bias b_h to the previous result, tanh activation function is applied. This step is given by (2.3) and represented as candidate hidden state in Figure 5.2.

In the algorithmic data flow, each layer is processed to completion before proceeding to the next layer. By restructuring the data flow, it is possible to reduce the need to store and retrieve the intermediate data after each layer. The results of the previous layer should be processed as soon as it is ready to reduce the memories required to hold the intermediate results. Figure 5.3 illustrates the proposed architecture of the GRUE after optimizing the data flow. Furthermore, network parameters (weights and biases) are structured in a way that biases required for the computation are stored first, followed by the weights illustrated in Figure 5.4, this reduces the hardware complexity (logic required to control the memory) and improves energy dissipation.

Since only one layer is computed at a time, one MAC computational unit is used for performing all computations. The multiplier bank and adder bank has eight multipliers and adders, respectively, see section 5.5.3.

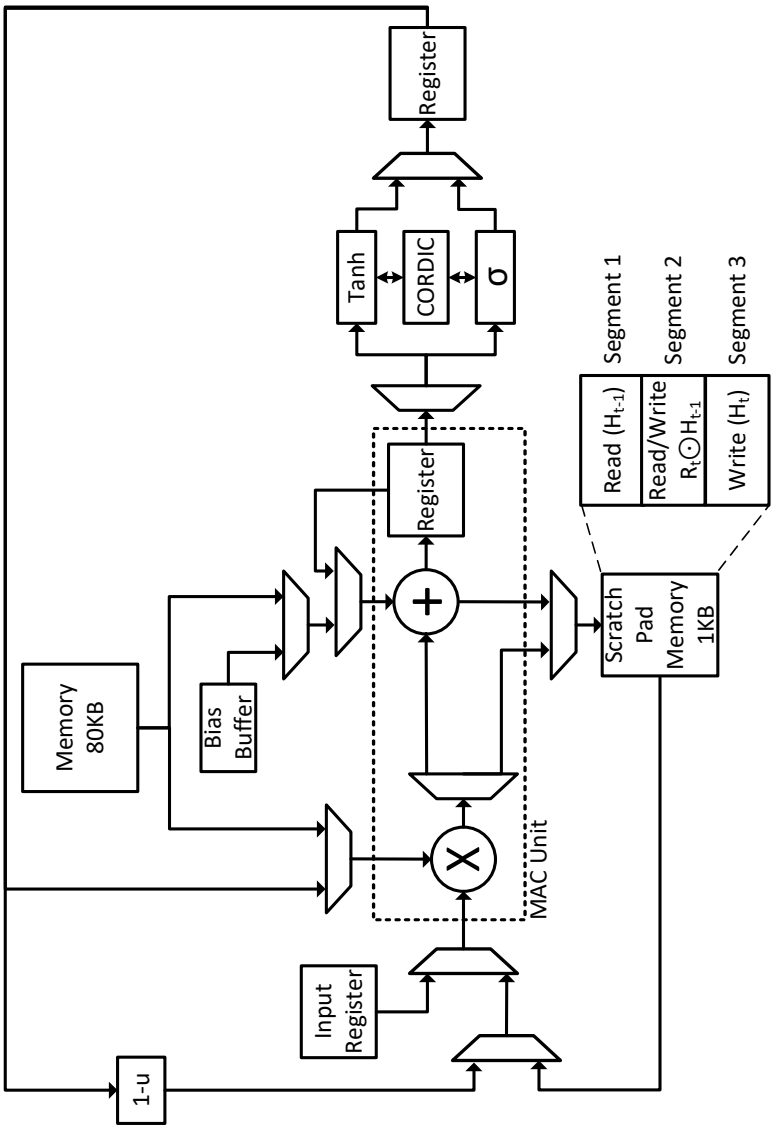

Figure 5.3: The proposed architecture of GRUE.

Instead of computing reset gate (2.1) and update gate (2.2) at once at the gating state, only computations of reset gate are performed. Since update gate results are not utilized until the hidden state (2.4), denoted as *new_h* in Listing 5.1, is computed. Furthermore, only eight values of reset gate (r_t) are computed at a time and stored in a buffer, these values are then multiplied with h_{t-1} (*prev_h*) using the same MAC unit, and the results (*r_state*) are stored in scratch pad memory. This process is repeated until all the values of *r_state* denoted as $r_t \odot h_{t-1}$ in (2.4) are computed and stored in scratch pad memory.

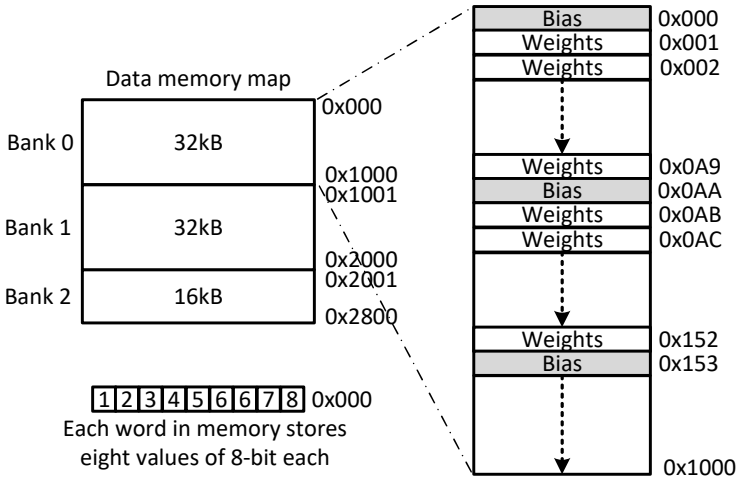

Figure 5.4: Structure of data memory representing the way biases and weights are arranged.

Computation of hidden state (new_h) requires candidate hidden state ($\tilde{h_t}$) and update gate (z_t) to be processed, refer (2.4). If the candidate hidden state is computed completely before computing an update gate, additional scratch pad memory will be required to store the values. The requirement for extra intermediate storage is evaded by computing four values of candidate hidden state ($\tilde{h_t}$) along with four values of the update gate (z_t). Using these eight values four hidden state values are computed. Due to this optimization, scratch pad memory was introduced with a write bit enable functionality. Following sections will go through the design steps and explain how architecture is structured.

5.4 Input ping-pong registers

The registers in the ping-pong group store the input data for hardware layers. There are two register groups; each group has ten registers of 16-bit each. The ping-pong registers are implemented to reduce latency. Figure 5.5 illustrates the structure of the ping-pong register group. After reset, both group one and group two are in idle state, and the consumer is set to the group first. The consumer register is to determine which ping-pong group to select for the datapath.

A producer, e.g., an CPU programs the input data for the first time step into register group one. After write completes, the CPU sets the enable bit in group_one_enable register. Each registered group has a status register that is set when the GRUE is using the data. When register group one is in use, CPU reads the status register of group two to ensure that it is idle. CPU begins programming the second register group and sets the second group enable bit when done. When the enable bit is set, any write to the register of the group that has enabled bit set will be dropped until the GRUE completes execution.

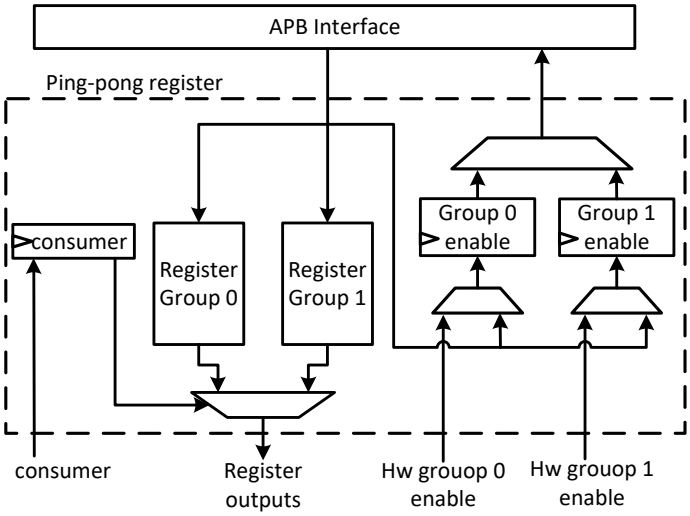

Figure 5.5: Structure of ping-pong registers.

When the GRUE has consumed the data in the first register group, GRUE will clear the enable bit of the first register group. GRUE will switch to a second register group if the enable bit is already has been asserted. This process repeats, with the second register group as an active group and the first register group inactive, which CPU can write. This allows the GRUE to switch between groups and keep executing without delay. Additionally, CPU does not need to enable an interrupt, nor poll the finish signal.

5.5 Arithmetic Functional Units

The following sections will discuss the mathematical complexity of the required computations in GRU NN. This section will introduce a proposed structure and discuss the complications that were considered while designing.

5.5.1 Matrix Multiplication

A vector of m elements multiplied with a matrix of m rows and n columns requires m x n multipliers and m x (n-1) additions. As mentioned in 5.1 the number of multipliers in the multiplication unit and the number of adders in the addition unit are chosen to be eight. Hence, the matrix multiplication is broken down into smaller chunks of computations. Row by Column multiplication approach is implemented where each element of vector is multiplied with the corresponding element of column in the matrix

$$P = A \times B \tag{5.1}$$

$$\begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_m \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & \dots & a_m \end{bmatrix} \times \begin{bmatrix} a_1 \times b_{1,1} + a_2 \times b_{2,1} + \dots + a_m \times b_{m,1} \\ a_1 \times b_{1,2} + a_2 \times b_{2,2} + \dots + a_m \times b_{m,2} \\ a_1 \times b_{1,3} + a_2 \times b_{2,3} + \dots + a_m \times b_{m,3} \\ \vdots \\ a_1 \times b_{1,n} + a_2 \times b_{2,n} + \dots + a_m \times b_{m,n} \end{bmatrix} \tag{5.2}$$

Since the inputs are limited to vectors of eight elements, this computation is further broken down to the following form.

$$\begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_m \end{bmatrix} = \begin{bmatrix} [a_1 \times b_{1,1} + \dots + a_8 \times b_{8,1}] + \dots + [a_{k+8+1} \times b_{(k*8+1),1} + \dots + a_m \times b_{m,1}] \\ [a_1 \times b_{1,2} + \dots + a_8 \times b_{8,2}] + \dots + [a_{k+8+1} \times b_{(k*8+1),2} + \dots + a_m \times b_{m,2}] \\ \vdots \\ [a_1 \times b_{1,n} + \dots + a_8 \times b_{8,n}] + \dots + [a_{k+8+1} \times b_{(k*8+1),n} + \dots + a_m \times b_{m,n}] \end{bmatrix} \tag{5.3}$$

where,

$$k = \left\lfloor \frac{m}{8} \right\rfloor - 1 \tag{5.4}$$

This method will produce eight results in parallel every cycle that will eventually accumulate to a eight values of final result vector P. By repeating the same process multiple times the entire result vector P is calculated. After m cycles all elements of the result vector P have been calculated. This approach makes it possible to acquire an eight value vector as a result and further apply computations on it while the next eight values of the result vector P is computed.

5.5.2 Element-wise Multiplication

The element wise multiplication also known as Hadamard product is computed in many layers of GRU algorithm. Element-wise vector multiplication can be represented by equation 5.5, where A and B are m-sized vectors.

$$P = A \odot B \tag{5.5}$$

$$\begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_m \end{bmatrix} = \begin{bmatrix} a_1 \times b_1 \\ a_2 \times b_{2,n} \\ \vdots \\ a_m \times b_m \end{bmatrix} \tag{5.6}$$

$$\begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_m \end{bmatrix} = \begin{bmatrix} [a_1 \times b_1 + \dots + a_8 \times b_{8,1}] + \dots + [a_{k+8+1} \times b_{(k*8+1),1} + \dots + a_m \times b_{m,1}] \\ [a_1 \times b_{1,2} + \dots + a_8 \times b_{8,2}] + \dots + [a_{k+8+1} \times b_{(k*8+1),2} + \dots + a_m \times b_{m,2}] \\ \vdots \\ [a_1 \times b_{1,n} + \dots + a_8 \times b_{8,n}] + \dots + [a_{k+8+1} \times b_{(k*8+1),n} + \dots + a_m \times b_{m,n}] \end{bmatrix} \tag{5.7}$$

This operations can be computed using the same multipliers used for matrix multiplication as these operations are not computed simultaneously. Thus, reducing the need of a separate unit for element-wise multiplication and reducing energy dissipation as the number of multipliers in the design reduces. In order to use the same multiplication unit the element-wise multiplication must also be broken down to chunks of eight elements.

5.5.3 Multiply-accumulate (MAC)

Figure 5.6 shows the structure of the MAC unit, which consists of eight 16x8-bit multipliers, four 24-bit, two 25-bit, one 26-bit and one 27-bit adder, and a 27-bit accumulator register to store partial sums. The accumulator is designed to be 27-bit to avoid overflow. The accumulator width was obtained through Matlab simulations, considering the worst-case scenario of NN configuration as KWS GRU pre-trained model of size 154x154x154x12 [14]. An additional overflow check is introduced to check the truncated MAC output for overflows.

The bias value is preloaded in the accumulator before MAC operation begins; this acts as a reset for accumulator and saves one addition operation. Different layers in GRU NN have different fractional and integer parts for representing biases, e.g. input layer has Q2.6 format, whereas the hidden layer has Q1.7 format. Therefore, MAC has a number of shifts (# shifts) as input parameters to properly align bias in an accumulator.

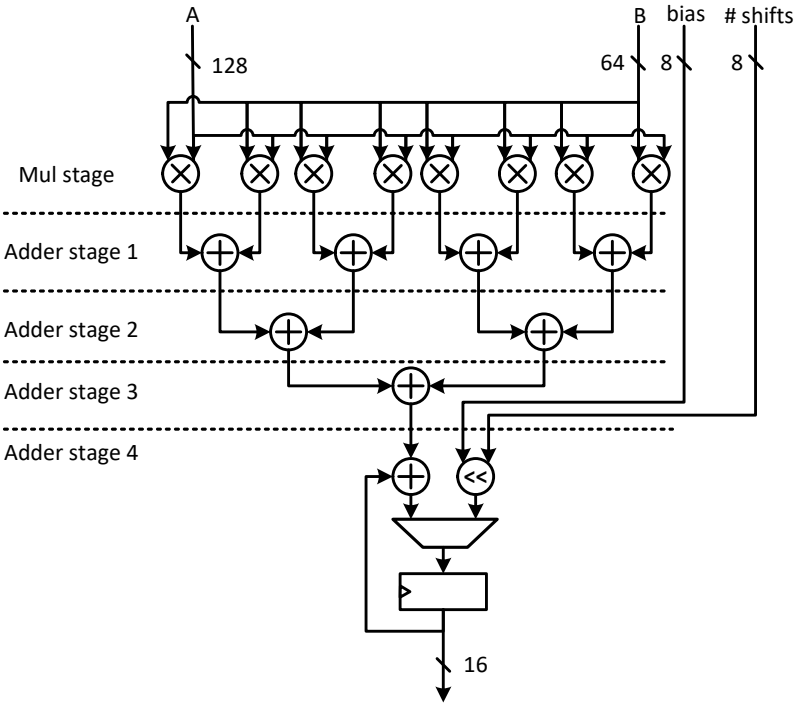

Figure 5.6: Architecture of MAC unit.

Due to the limited range of the chosen fixed-point format Q10.6, using the same word length for the accumulators and weights and activation would quickly cause overflows, leading to significant accuracy loss. Therefore, higher wordlength is preferred for accumulators. Multiplying two n-bit integers will result in a $2n$-bit number. To avoid overflow, the accumulator should be $2n + K$ bits wide, where K represents the number of bits required to avoid overflow. The extra bits required

to avoid overflow depends on the network size and the range of the values of the parameters.

MAC unit is used for point-wise multiplication, vector-matrix multiplication and addition due to hardware optimization performed in section 5.3.1. It is designed such that point-wise multiplication and addition results are obtained after multiplier stage and adder stage 1, respectively.

5.6 Configuration Module

The configuration module contains configuration registers that are needed for performing inference for multiple time steps. The word length of the configuration registers is based on pretrained KWS GRU model. The module was designed to configure the parameters of the NNs. The configuration registers are

- start - 1-bit register to trigger the inference.
- cordic_itr - Represents the number of cordic iteration. A five-bit register is used to represent CORDIC iteration for up to 32-bit operands.
- sc_blk_0_start_addr, sc_blk_1_start_addr, sc_blk_2_start_addr - Registers contain the start address of three segments of scratch pad memory. The word length necessary to address the memory space depends on the size of the scratch pad memory. A six-bit register is used for the implemented KWS GRU NN.
- sc_blk_0_end_addr, sc_blk_1_end_addr, sc_blk_2_end_addr - Registers contain the end address of three segments of the scratch pad memory.
- mem_start_addr - Register contains the start address of SRAM that store weights and biases. A 14-bit register is used to address the SRAM memory space.
- mem_end_addr - A 14-bit wide register contains end address of SRAM.
- input_sample_0_0-4 - The APB interface bus width is 32-bits wide. However to store 12 inputs of 16-bit word length, 192-bit register will be required. Hence, four 32-bit wide registers are defined to store group 0 input values.
- input_sample_1_0-4 - As mentioned above to accommodate for the 32-bit wide APB bus, four registers of 32-bit are defined to store group 1 input values.
- input_consumer - A one-bit wide register to indicate which ping-pong register group the functional units are using.
- input_sample0_en - A one-bit wide register which is set high when ping-pong register group 0 input values are ready. The GRUE set this bit low when the input are consumed and no longer valid.
- input_sample0_stat - A one-bit wide register which is set high when group 0 register input values are used for the GRU network computation.

- input_sample1_en - 1-bit register which is set high when the interface has finished loading the group 1 input values.
- input_sample1_stat - 1-bit register which is set when the group 1 register input values are used for GRU network computation.
- prediction - Register to store the final prediction of the GRUE. The final prediction is a number between 1-12, four-bits are required to represent the prediction value.
- finish - A one-bit register that indicate computation of a batch of input is done.

These registers are accessed over APB interface.

5.7 Memory

The GRUE has two memory modules, one SRAM bank consist of three single-port SRAM memory that stores 64-bit words. A second is a scratch pad memory which is a two port memory, consisting of standard-cells, that store 128-bit words. Based on the requirements of KWS model, the number of words (vectors) required to store weights and biases of GRU NN inference is 10001 64-bit vectors (∼78.132KB). The RAM bank consist of three SRAM, two SRAMs of size 32KB and one of 16 KB giving total storage of 80KB as shown in Figure 5.7. Reading from smaller SRAMs costs less energy when compared to reading from bigger SRAM instance.

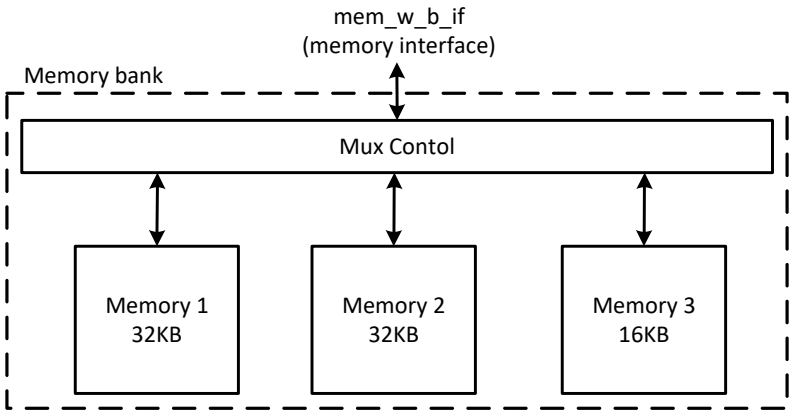

Figure 5.7: Structure of RAM bank.

Due to data flow optimizations explained in section 5.3.1, there is a need of a memory that allows read and write in the same clock cycle. Scratch pad memory is used instead of dual port memory for this purpose due to advantages such as low latency and less area as dual port memories have an area overhead for small capacities. A scratch pad memory have 60 words of 128-bit each (∼0.9375kB).

Another approach is to use three single port memories (SRAMs), each of 0.3125kB (20 words x 128-bits). However, this approach also has a higher area than

standard-cell memories due to the area overhead SRAMs have for small capacities. Also for applications that requires several small capacity memories scratch pad memories are a suitable choice [35].

The scratch pad memory is divided into three segments where each word stores eight elements of 8-bit each, illustrated in Figure 5.8. After each new time step the memory segment one and three are swapped, i.e. the memory segment used for reading the inputs in one time step will become a segment for storing new results in next time step and vice-versa, as shown in Figure 5.9.

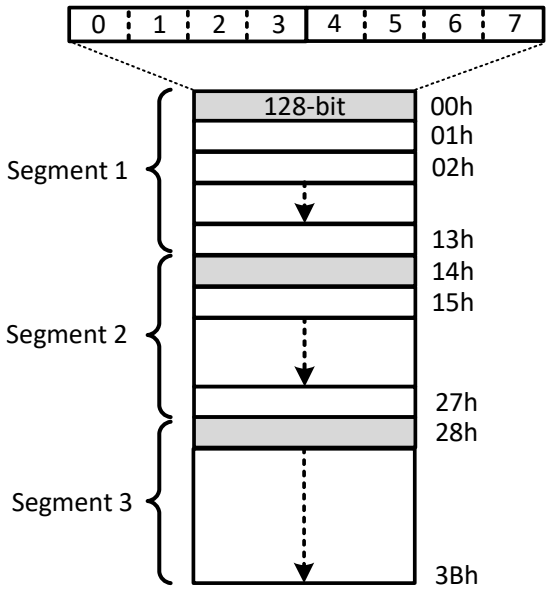

Figure 5.8: Structure of scratch pad memory constructed using standard cells.

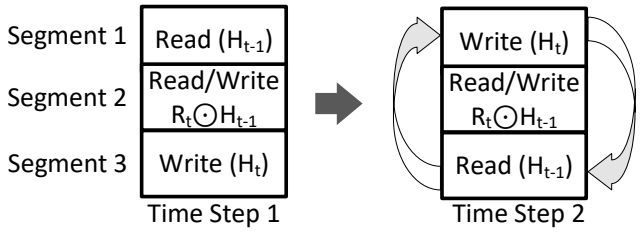

Figure 5.9: Figure showing the flipping of scratch memory segments after each time step.

5.8 FSMs

The two FSMs illustrated in Figure 5.10 and Figure 5.11 control the data flow in the different units. State machines control the counters related to the number of inputs, time steps, number of elements computed, CORDIC iteration count, etc. These counters are used to change the states based on the comparison between the counter value and the parameters. During reset both state machine are in *idle* state.

The main controller is responsible for controlling the GRU algorithmic flow by commanding the matrix controller and transferring the correct data to all the modules. The matrix controller is responsible for controlling the MAC unit and activation functions. The communication between the main controller and matrix controller is limited to two signals, *matrix trigger* and *eight done*.

After reset, main controller is in *idle* state, *start* signal is generated by a producer (e.g., a CPU) and transmitted to GRUE through APB interface. The start signal is to let main controller know, input data is available in ping-pong registers and computation can be started.

5.8.1 Main Controller FSM

The main controller is an FSM that control multiple time steps of the GRU algorithm to produce final prediction against the input data set. When valid input data is presented through the APB bus, it is stored in the input ping pong registers. The start signal is asserted and the state changes from *idle* to *reset gate* state. In the reset gate state, another FSM called matrix controller is triggered, which controls bias loading, matrix multiplication, and activation functions. Whenever matrix controller is triggered, one element is computed and stored in a temporary register of size 8x16-bit. Once eight such values are computed and stored in a register, *eight_ done* signal is set high and FSM switch to *R_ h* state.

The *R_ h* state computes $r_t \odot h_{t-1}$, which is a part of (2.3), and stores the result in segment two of scratch pad memory via multiplier intermediate results input port, shown in Figure 5.16. The FSM remains in this state for one clock cycle. If the scratch memory segment 2 is filled, *sc_ blk_ 2_full* flag is asserted and the main controller moves to *candidate and update gate* state. If the scratch memory segment two is not full, the main controller moves back to the reset gate state, and this process continues until scratch memory segment two is full (i.e., 154 values are stored).

In *candidate and update gate* state, matrix controller is triggered which is used to calculate four values of candidate hidden state ($\tilde{h_t}$) given by (2.2) and four values of update gate (z_t) given by (2.1). These eight values are stored in the same temporary register, which was used to store the values of the reset gate. Once the register is full *eight_ done* signal is asserted and main controller moves to *new_ h* state.

The *New_ h state*, computes h_t given by (2.4). This state computes four values at a time and stores them in segment 1 or in segment 3 of scratch memory, depending on which time step is being executed as explained in section 5.7. After computing four values, main controller moves back to *candidate and update gate*

state until *mat_comp_done* is asserted. When *mat_comp_done* is set high and *time_step* equals 25, main controller transit to *output_layer* state. Otherwise, when *time_step* is less than 25 and *cont_comp* is high main controller moves to *reset gate* state.

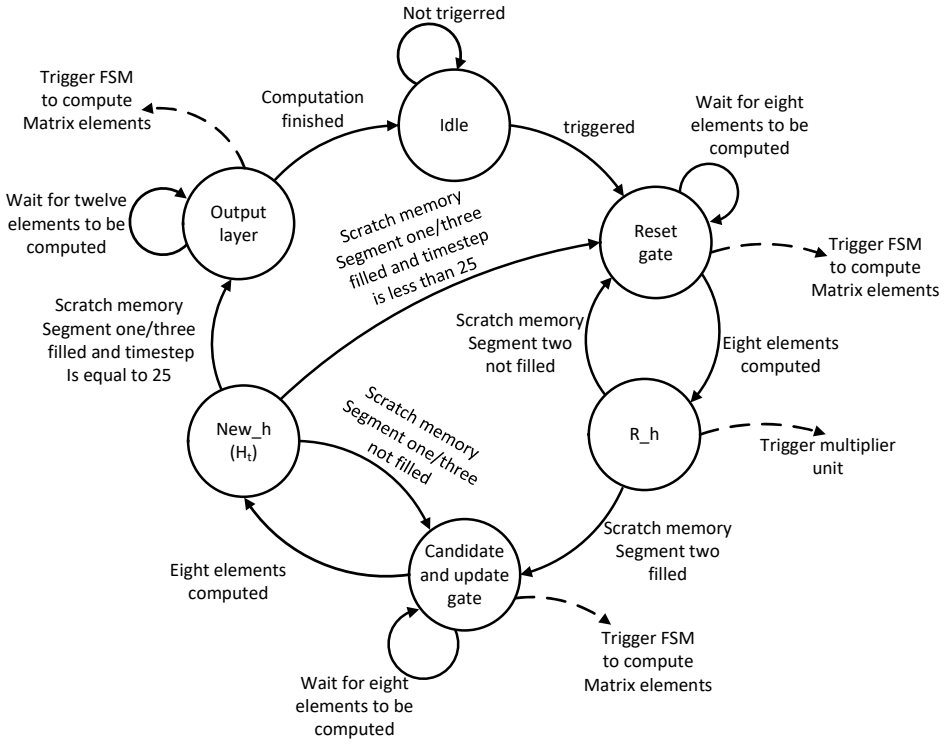

Figure 5.10: Main controller FSM.

The *Output layer* state computes the final twelve outputs by triggering matrix controller FSM. Once all the values are computed finish signal is set high, and the main controller moves to the *Idle* state.

5.8.2 Matrix controller FSM

The FSM in Figure 5.11 controls the matrix multiplication in the GRUE. The main controller triggers this state machine explained in section 5.8.1. During reset, the FSM is in *Idle* state, and it remains in the idle state until the main controller triggers the FSM.

In *Load bias* state, the bias value is loaded into the bias buffer and the accumulator register from memory as shown in Figure 5.6. In this state *ld_bias* signal is asserted, which is sent to the bias buffer module, explained in section 5.9. The FSM stays in load bias state for a single cycle, after which it changes to *compute matrix* state.

The *Compute matrix* state is responsible for computing an element of a matrix

using MAC operation followed by an activation function. To achieve this several steps needs to be done: the memory address controller has to increment the weights address; the MAC input selector module provides the input data from the input register and segment one/three of the scratch memory depending on time step as explained in section 5.7 and the MAC module has to be enabled with relevant weights and input. This is accomplished by setting the *mem_addr_cnt_en* and *mac_en* flags. Once an element is computed *ele_calc_done_upcnt_en* flag is set to increment the counter responsible for track of number of elements computed. Afterwards, sigmoid or tanh activation is performed depending on the layer being executed by setting *act_funct_en* and *act_funct_sel* flags.

Eight biases are fetched at a time from memory, out of which seven are stored in a buffer, as explained in section 5.9. The *rd_bias* flag is used to read the bias from the buffer and load it to the accumulator register. The rd_bias signal is used for loading the biases corresponding to the computation of the element second to eight. The FSM stays in this state, and the above process is repeated until eight elements are computed, this is denoted by "still computing" in Figure 5.11. When eight values are computed, *eight_done* flag is set and the state moves to *activation wait*.

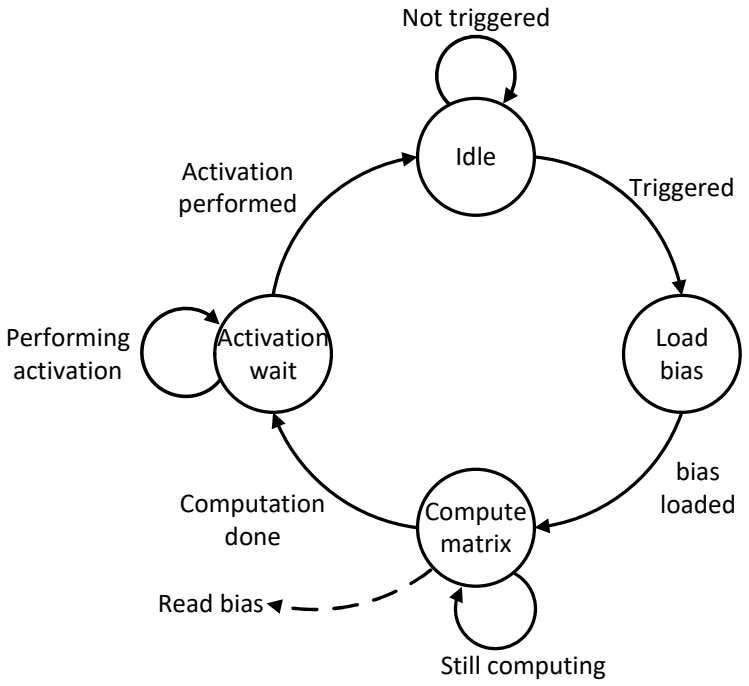

Figure 5.11: Matrix controller FSM.

In the *Activation wait* state, the FSM waits for the activation function to complete the computation. Since activation function is implemented using fast sigmoid and fast tanh approximations using the cordic algorithm as mentioned in section 3.3.4, it takes up to 16 clock cycles to perform activation. This state is required

for the last element because activation functions are performed in parallel with MAC operation. Once the MAC output is valid, it is sent to activation module, at the same time MAC unit gets new input and start second element computation as shown in Figure 5.12. On completion, *eight_ done* flag is asserted and FSM switches to *Idle* state.

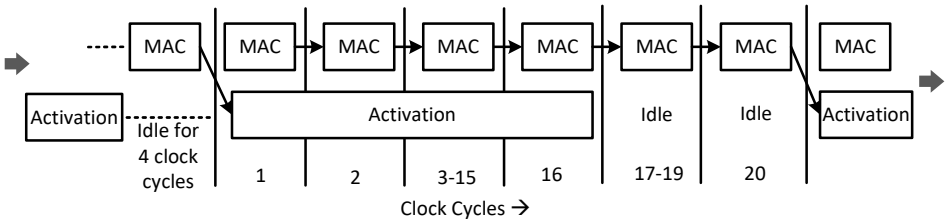

Figure 5.12: Activation computation in parallel with the MAC operation.

This state is required for the last element because activation functions are performed in parallel with MAC operation. Once the MAC output is valid, it is sent to activation module, at the same time MAC unit gets new input and start second element computation as shown in Figure 5.12. On completion, *eight_ done* flag is asserted and FSM switches to idle state.

5.9 Buffer for bias

The buffer module consists of seven registers, where each entry in the register is represented as an eight-bit signed value. Since the design is implemented using eight multipliers, as explained in section 5.5.3. The weights and biases are arranged in a memory such that eight values are stored at each word, see section 5.3.1. Due to this implementation, eight values of the bias are fetched from memory at a time. Out of these eight biases, one bias at MSB position is loaded to the accumulator register, and remaining seven values are stored in a buffer. The buffer keeps track of the values that have been used using a counter.

Two signals, namely *ld_ bias* and *rd_ bias* are used to control the loading and reading of the biases to and from the buffer, respectively as shown in Figure 5.13. Initially, at the start of computation, *ld_ bias* signal is set, and biases are fetched from the main memory and loaded into the buffer. Afterward, *rd_ bias* signal is set when a new value of the bias is required to be loaded in the accumulator register. Once all the values in buffer are utilized i.e., eight values are computed, and the temporary register explained in section 5.8.1 is filled; the buffer is loaded with new biases by asserting the *ld_ bias* signal and the process repeats.

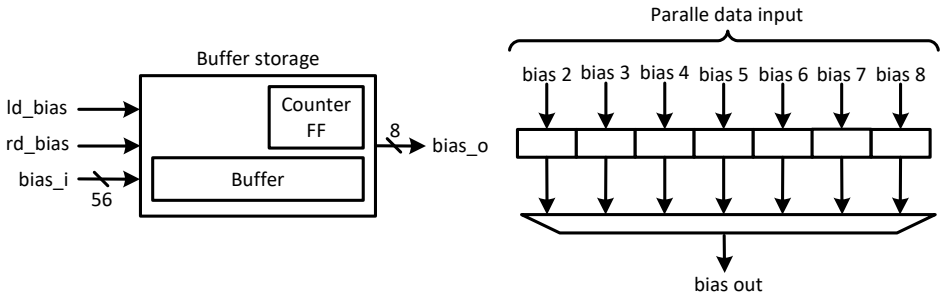

Figure 5.13: Block diagram of bias buffer.

5.10 MAC input selector module

This module is responsible for providing weights and inputs to the MAC unit. Main memory, scratch pad memory, the activation result buffer, and input ping-pong registers are four inputs to the MAC input module as shown in Figure 5.14. Based on the value of the *decider* signal from the main controller, the MAC input module determines which combination of the input source should be selected. When performing matrix multiplication in the GRUE, 21 iterations of MAC operation are required to get one element of a resultant matrix (P).

$$\begin{bmatrix} p_1 & p_2 & p_3 & \dots p_{154} \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & a_3 & \dots & a_{164} \end{bmatrix} \times \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \dots w_{1,154} \\ w_{2,1} & w_{2,2} & w_{2,3} \dots w_{2,154} \\ w_{3,1} & w_{3,2} & w_{3,3} \dots w_{3,154} \\ \vdots & & \\ w_{164,1} & w_{164,2} & w_{164,3} \dots w_{164,154} \end{bmatrix} \tag{5.8}$$

Consider an example of reset gate computation where (2.1) can be simplified to form

$$r_t = \sigma([x_t, h_{t-1}]W_r + b_r), \tag{5.9}$$

where $[x_t, h_{t-1}]$ represents input x_t concatenated with h_{t-1} (which is a vector of zeros in the first time step).

In first iteration eight values fetched from the input ping-pong register along with the corresponding weights from the main memory, these are sent to MAC input A and B, respectively. In second iteration two values are fetched from the input ping-pong register and concatenated with six values of h_{t-1} from the scratch pad memory and sent to A along with the corresponding weights from main memory to B. From iteration three to twenty-one, values for scratch pad memory with corresponding weights from main memory are passed to the MAC unit. This process is carried out for matrix multiplication multiplication in (2.1), (2.2) and (2.3).

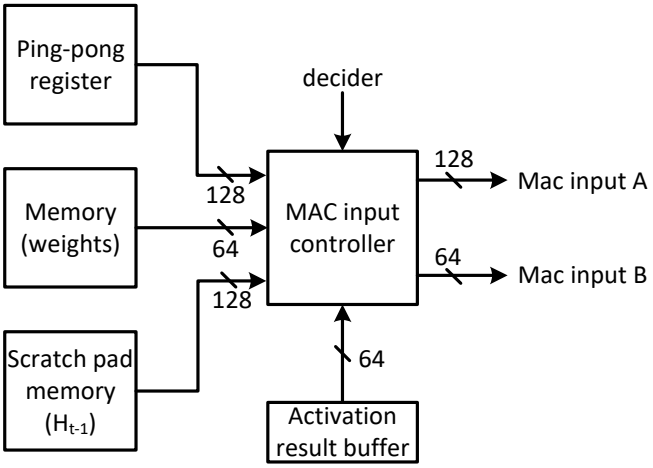

Figure 5.14: Data flow of MAC input controller.

For point-wise multiplication $(r_t \odot h_{t-1})$ in (2.3), activation results r_t are fetched from activation result buffer, see section 5.11, and eight h_{t-1} values are fetched from scratch pad memory.

MAC input controller module handles the input distribution for (2.4) in a slightly different manner. After computation of eight values in the *candidate and update gate* state, see section 5.8.1, activation result buffer has four values of tanh and four values of sigmoid. MAC input controller uses four values of sigmoid from the buffer and performs $1 - z_t$ operation in (2.4). Append $1 - z_t$ result with h_{t-1} and send it to MAC input A. The *Activation result buffer* data is sent to MAC input B.

5.11 Activation result buffer module

The activation functions, i.e., sigmoid and tanh, are computed one at a time. As seen in (2.3), (2.4) the output of the activation functions are further fed as input to the MAC unit.

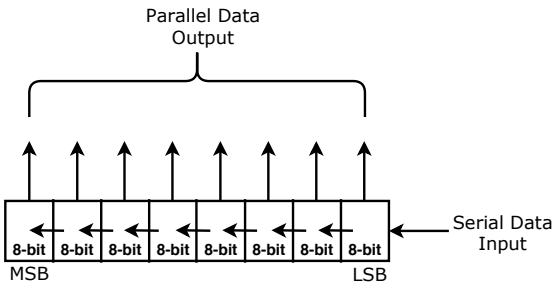

Figure 5.15: A shift register with parallel output to buffer activation results.

However the MAC unit operates on vectors of eight elements as input. This requires the output of the activation function to be buffered until eight values have been computed. Thus, a shift-buffer with parallel output is implemented which stores 8 x 8 bits, a vector of 8 elements. Based on the *act_compt_done* signal from the nn_gru_act_wrap_rtl module and *decider* signal from the main controller ,the output from the activation function in nn_gru_act_wrap_rtl module is stored.

5.12 Scratch pad memory controller module

This module controls the flow of data to/from the scratch pad memory. On reset the registers storing start/end addresses for the three segments of scratch pad memory are initialized zero. When *start* signal is asserted, registers are loaded with appropriate start/end addresses by configuration module. Depending on the *decider* signal from main controller, read/write operations are performed to the different segments of scratch pad memory.

When the main controller is in R_h state, i.e., computing $r_t \odot h_{t-1}$, see section 5.8.1, scratch pad memory controller keep track of the number of values computed. Once scratch pad memory segment one is full, it asserts the *sc_blk_2_full* signal which is used by main controller to switch between states.

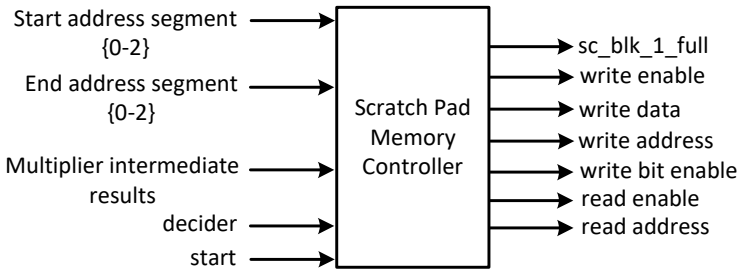

Figure 5.16: Block diagram of scratch pad memory controller.

5.13 Functionality verification

On reset, the data (weights and biases) are loaded in the *nn_gru_memory* from three files namely *wght_bias_0.bin*, *wght_bias_1.bin*, and *wght_bias_2.bin*. Three binary files corresponds to three memories instantiated in memory bank, see section 5.7. The binary files are generated using a Matlab script and it specifies the weight and biase values. Testbench uses SystemVerilog's *$loadmemh* built-in function to load binary files in memory, illustrated in Listing 5.2.

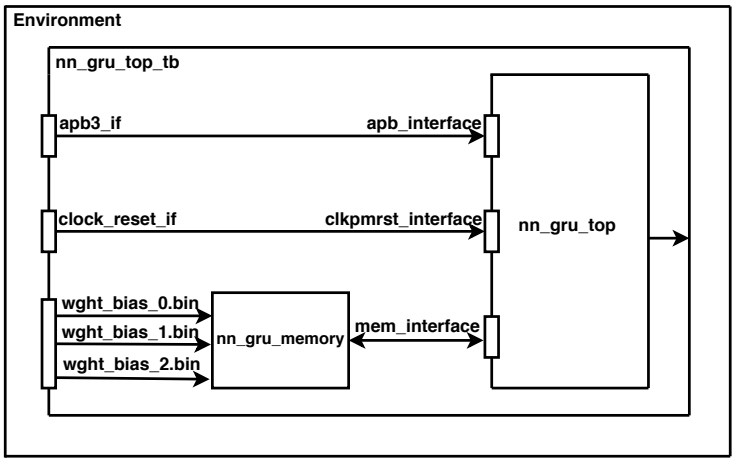

Figure 5.17: Overview of the testbench setup

The *input_batch.bin* file generated using Matlab was used to load the input values in the ping-pong input register, see section 5.4.

```
1 `testbench.memory_block_inst.inst0.loadmem("wght_bias_0.bin");
2 `testbench.memory_block_inst.inst1.loadmem("wght_bias_1.bin");
3 `testbench.memory_block_inst.inst2.loadmem("wght_bias_2.bin");
```

Listing 5.2: SystemVerilog code for loading SRAM.

The testbench executes a sequence of write operations for populating the configuration registers. All the registers in the configuration module are set to their reset values when *reset_sys_b* is asserted, the registers in the *nn_gru_cfg* module are loaded with data necessary for executing the inference using the *cfg_data_in* input. This is done by utilizing the *write_reg* task that takes register address and data as arguments. A single *write_reg* operation require two clock cycles. After the single GRU inference complete, *read_reg* task is used to read the max value and the prediction value from the register.

Chapter 6
Synthesis Results

The Synopsys Design Vision tool was used to perform synthesis and estimate power for the proposed design using standard cell library in a 28 nm technology. The switching activity data generated from the gate-level simulations is dumped to switching activity interchange format (SAIF) file, which is used for power analysis. The SAIF file contains toggle counts on the signals of the design.

6.1 xDSP

The xDSP was synthesised at 9.6MHz frequency. The power number obtained by running a benchmark on the smaller network of 4 neuron per layer are shown in Table 6.1. As xDSP has a limited amount of memory, leakage of additional memory instances that would be needed for a full network is added.

Table 6.1: Power in μW for the small and estimated GRU network implementation on xDSP. It includes power numbers with and without power management (PM).

Module	Power (μW)	
	Up to 40 neurons	Estimated for 154 neurons
xDSP	44.90	44.90
Memory	36.01	36.01
Additional memory leakage	0	58.50
Total power	80.89	139.41

The total power during computation of one inference of GRU is 139.41μW, where the leakage of additional memories that are required to store the parameters of

the GRU network is also added. Table 6.2 illustrates the energy dissipation per inference for the 4, 20, and 40 neuron per layer network.

Table 6.2: Energy dissipation in μJ for the GRU network implementation on xDSP.

GRU network	Energy (μJ)
4 neuron per layer	0.40
20 neuron per layer	1.93
40 neuron per layer	4.87
154 neuron per layer (estimated)	48.71

Due to the limited memory on xDSP processor, energy for the GRU network with 154 neurons in each layer was estimated by finding the scaling factor, see section 4.6. The energy dissipation of the network with 154 neuron per layer is 48.71μJ, estimated by multiplying energy dissipation of 40 neuron per layer network by a scaling factor of 10.

6.2 Dedicated GRUE

Table 6.3 shows the power numbers for the GRUE with three memory instances, two instances of size 4096x64 and one instance of size 2046x64. The dedicated GRUE with three memory instances have high dynamic power cost of 121μW. Due to this another approach was also evaluated, where ten small memory instances of 8KB each was used, shown in Table 6.3.

Table 6.3: Power in μW for the dedicated GRUE with three and ten memory instances.

Module	Power (μW)	
	Three memory instances	Ten memory instances
Memory bank	135.50	90.06
Scratch pad memory	7.91	5.99
Combinational	55.82	41.15
Clock network	1.42	1.42
Register	4.73	4.61
Total Power	205.38	143.23

It can be observed from Table 6.3, that memory consumes the majority of the power, ∼66%. The Figure 6.1 illustrates the area comparison of the SRAMs, scratch pad memory, and the top module. The SRAMs occupy 87% of the area and the *top module* occupy merely 3% of the area.

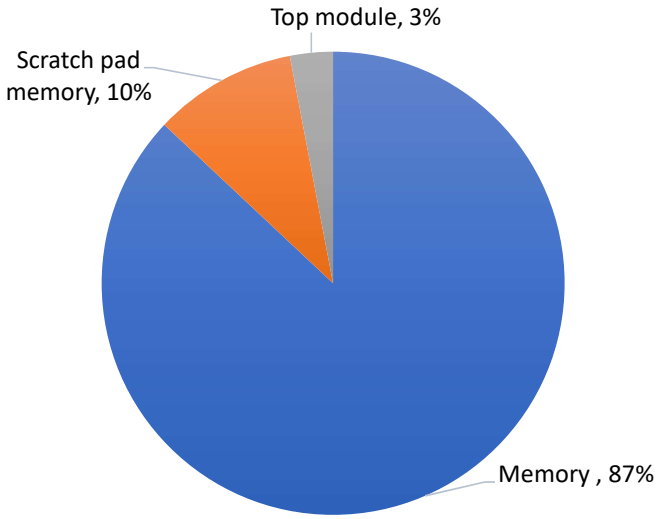

Figure 6.1: Area percentage occupied by modules.

Figure 6.2 illustrates the change in power cost of the memory when two different memory configurations were used. The dynamic power, and the total power of the memory reduced by 47% and 31%, respectively, when ten small memory instances were used. However, leakage power increased by 76%.

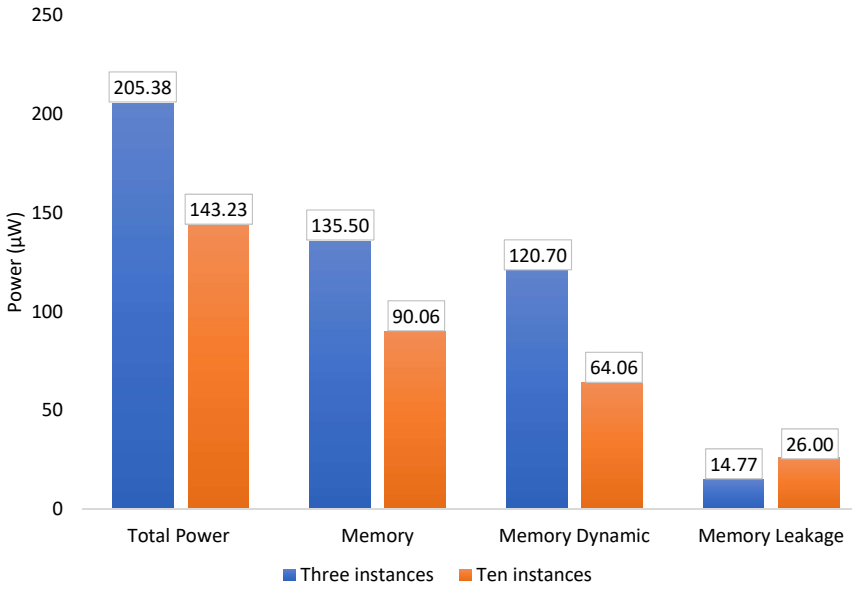

Figure 6.2: Power cost comparison of GRUE with three and ten memory instances.

Table 6.4 shows the energy cost of dedicated GRUE with three memory instances and ten memory instances.

Table 6.4: Energy cost for the dedicated GRUE.

Implementation	Energy/CC (pJ)	Energy/Inference (μJ)
Three memory instances	20.53	5.52
Ten memory instances	14.32	3.85

Although, significant decrease in power cost was observed when ten small memory instances was used, there is considerable increase in area, illustrated in Figure 6.3. The total area increased by 39%, whereas area occupied by memories increased by 45%.

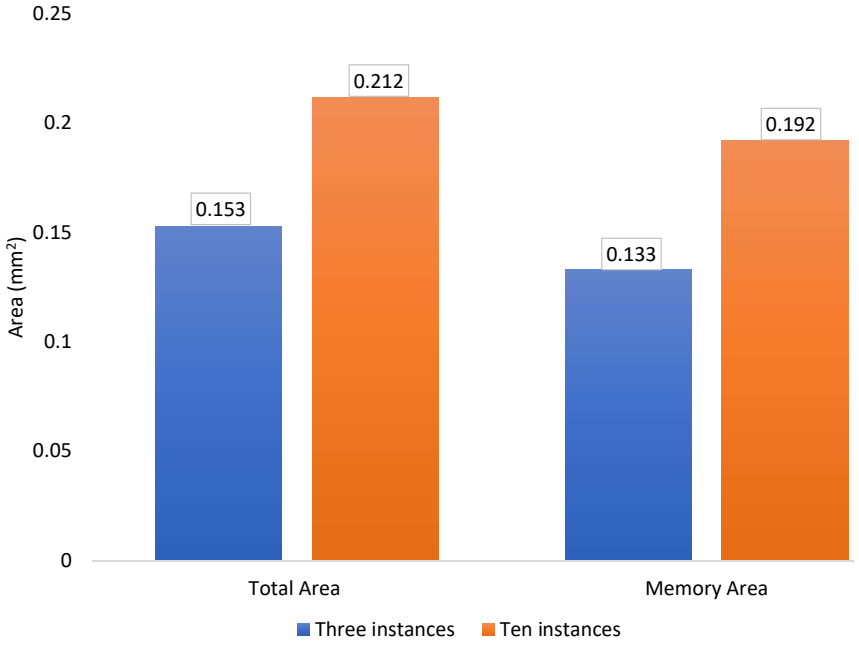

Figure 6.3: Area comparison of GRUE with three and ten memory instances.

Figure 6.4 illustrates the energy dissipation comparison between xDSP and GRUE. The xDSP with scaled GRU network dissipate more energy by a factor of ∼13 compared to GRUE with three memory instances. Moreover, GRUE completes the inference in 268854 clock cycles, which meets the real time requirement of 40 ms to do the computation.

Figure 6.5 shows the area comparison between xDSP and GRUE implementation. The area of the additional memory instances that are required for storing the network parameters are added in the xDSP area. The area of the xDSP without considering additional memories is 0.3 mm^2, which is comparable to area of dedicated GRUE with ten memory instances.

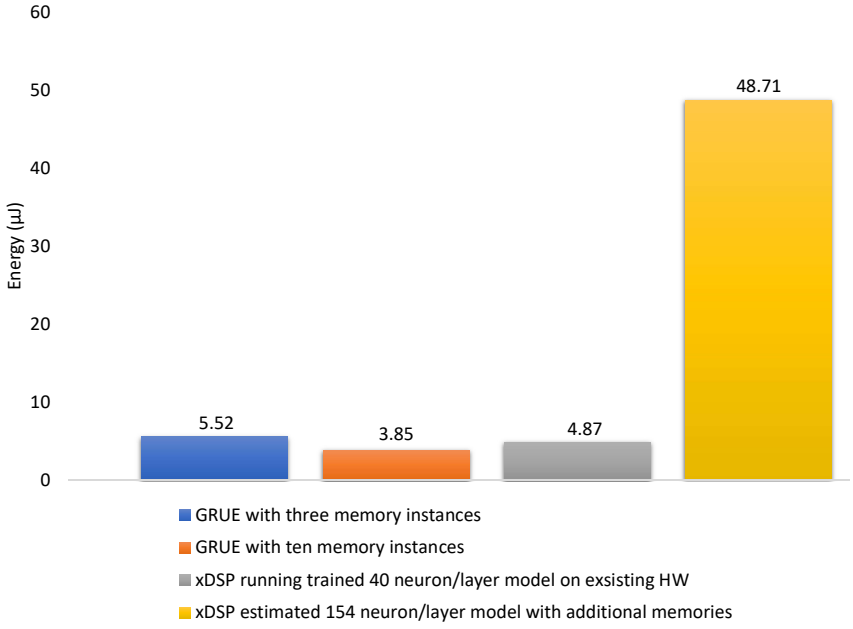

Figure 6.4: Energy cost comparison of GRUE with xDSP.

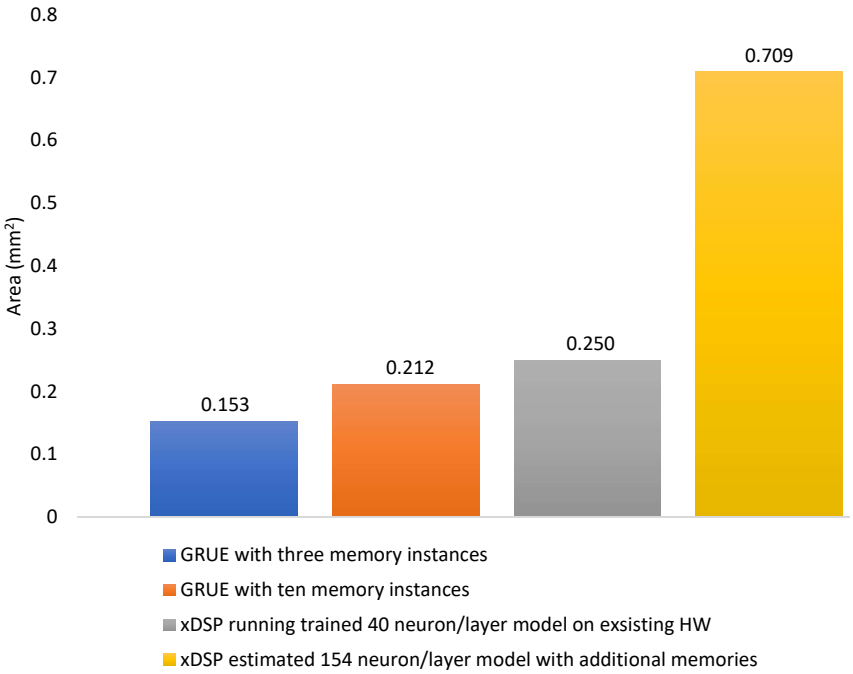

Figure 6.5: Area comparison of GRUE with xDSP.

6.3 Discussion

The GRUE operates independent of software in general. The loading of input parameters into the memory, loading of the configuration registers with the appropriate network parameters and memory addresses are required to be performed in software. Adding the power management logic for memories can yield considerable improvements in terms of leakage power and reduce the energy cost per clock cycle.

In terms of latency, GRUE performs close to minimum number of cycles considering a single 64-bit memory interface and 8-bit parameters. This was achieved by data flow optimizations, mentioned in Section 5.3.1. The GRU performs additional MAC operations, caused by the fact that weights do not fit exactly in the vector representation, i.e., the last vector is padded with four zero-valued weights, which are also computed. Most of the energy cost is associated with memory access and to less extent MAC operations. To achieve significant improvements on an GRUE in terms of latency and energy dissipation, intermediate results are stored in the scratch pad memory constructed using standard cells. Also, LUT was not investigated at the implementation of HW.

Chapter 7
Conclusion

The purpose of thesis was to explore and evaluate the impact of GRU NN processing on low-power hearing instruments (HI). Oticon's HI use xDSP processor for processing the audio data. Therefore, the xDSP processor was used for implementation of GRU algorithm and for comparison with the dedicated GRUE.

The implementation of GRU inference on the xDSP processor was used to get baseline metrics that were used to measure the improvements of the optimized GRUE. It was observed that memory access and MAC operations are the most costly in terms of energy cost. Also, most of the instructions shown in Table 4.3 did not correspond to MAC or memory access operations. This was probably caused due to un-optimized software implementation and the functional units in xDSP are designed for a different purpose. Since, energy dissipation and power budget is a major concern in lower-power devices, a fixed-point or integer number representation is ideal. The loss of accuracy caused due to reduced wordlength can be overcome by retraining the network.

The GRUE performs inference in significantly less number of clock cycles. The number of clock cycles are close to the number of vectorized parameters that need to be retrieved from memory. In contrast to xDSP, GRUE takes significantly less clock cycles to perform inference and meets the real time computation requirement of 40 ms. For an audio sample of one second as input, xDSP takes around 473 ms where as GRUE takes 26 ms. xDSP in not suitable for computing large-scale GRU NNs. The dedicated GRUE solution has the best performance from the explored solution and can be integrated with xDSP to compute NNs.

7.1 Future works

As future work, LUT can be implemented for computing activation functions. Which will reduce the clock cycles (CC) required for computing sigmoid or tanh activation from 16 CC to 2 CC. Furthermore, synthesizing GRUE with LUT provide more accurate area and energy estimations for comparison with current implementation.

Retrain the GRU NN with quantized weights, biases to recover some accuracy loss. Inputs can also be quantized to eight-bit along with weights and biases, followed by retraining, to further reduce the network size. Moreover, physical implementation by place and route can be carried out to obtain more accurate area

and timing results.

7.1.1 Look Up Table (LUT)

A LUT approach is also evaluated for approximating the sigmoid and tanh activation. MATLAB was used to generate a LUT and a decoding logic. The generated LUT comprises of 50 entries of eight-bit each. A verilog file was also generated using MATLAB. The LUT are hard-wired after synthesis, whose purpose is specific to a function. Therefore, sigmoid and tanh require separate LUT. The area of a single LUT is comparable to fast sigmoid and fast tanh implementation. Since, separate LUT is required for sigmoid and tanh activation, the resulting area of LUT implementation will be more than fast sigmoid and tanh approach for approximation. The average relative error of sigmoid replacement function is 0.0017 and for tanh is 0.0038. It is observed that LUT implementation of activation functions has less error as compared to other approximations evaluated.

Bibliography

[1] B. Marr, "Deep learning vs neural networks - what's the difference?," 2020. [Online]. Available: https://bernardmarr.com/default.asp?contentID=1789.

[2] amoeba (https://stats.stackexchange.com/users/28666/amoeba), *What is the difference between a neural network and a deep neural network, and why do the deep ones work better?* [Online]. Available: https://stats.stackexchange.com/q/184921.

[3] B. Moons, D. Bankman, and M. Verhelst, *Embedded Deep Learning: Algorithms, Architectures and Circuits for Always-on Neural Network Processing*, 1st. Springer Publishing Company, Incorporated, 2018, ISBN: 3319992228.

[4] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. P. Kuksa, "Natural language processing (almost) from scratch," *CoRR*, vol. abs/1103.0398, 2011. arXiv: 1103.0398. [Online]. Available: http://arxiv.org/abs/1103.0398.

[5] L. Deng, J. Li, J. Huang, K. Yao, D. Yu, F. Seide, M. L. Seltzer, G. Zweig, X. He, J. D. Williams, Y. Gong, and A. Acero, "Recent advances in deep learning for speech research at microsoft," in *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*, IEEE, 2013, pp. 8604–8608. DOI: 10.1109/ICASSP.2013.6639345. [Online]. Available: https://doi.org/10.1109/ICASSP.2013.6639345.

[6] "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, 2012. DOI: 10.1109/MSP.2012.2205597. [Online]. Available: https://doi.org/10.1109/MSP.2012.2205597.

[7] S. Y. K. G. B. L. p. i. L. E. Z. J. Y. T. T. N. X. S. S. Y. W. H. Y. Jiantao Qiu Jie Wang, "Going deeper with embedded fpga platform for convolutional neural network," in *FPGA '16: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 26–35. DOI: 10.1145/2847263.2847265. [Online]. Available: https://doi.org/10.1145/2847263.2847265.

[8] R. Socher, C. C. Lin, A. Y. Ng, and C. D. Manning, "Parsing natural scenes and natural language with recursive neural networks," in *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*, L. Getoor and T. Scheffer, Eds., Omnipress, 2011, pp. 129–136. [Online]. Available: https://icml.cc/2011/papers/125_icmlpaper.pdf.

[9] J. Mao, W. Xu, Y. Yang, J. Wang, and A. L. Yuille, "Deep captioning with multimodal recurrent neural networks (m-rnn)," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6632.

[10] A. Graves, A. Mohamed, and G. E. Hinton, "Speech recognition with deep recurrent neural networks," *CoRR*, vol. abs/1303.5778, 2013. arXiv: 1303.5778. [Online]. Available: http://arxiv.org/abs/1303.5778.

[11] W. Wahby, T. Sarvey, H. Sharma, H. Esmaeilzadeh, and M. S. Bakir, "The impact of 3d stacking on gpu-accelerated deep neural networks: An experimental study," pp. 1–4, 2016.

[12] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *CoRR*, vol. abs/1703.09039, 2017. arXiv: 1703.09039. [Online]. Available: http://arxiv.org/abs/1703.09039.

[13] S. Fernández, A. Graves, and J. Schmidhuber, "An application of recurrent neural networks to discriminative keyword spotting," in *Artificial Neural Networks - ICANN 2007, 17th International Conference, Porto, Portugal, September 9-13, 2007, Proceedings, Part II*, J. M. de Sá, L. A. Alexandre, W. Duch, and D. P. Mandic, Eds., ser. Lecture Notes in Computer Science, vol. 4669, Springer, 2007, pp. 220–229. DOI: 10.1007/978-3-540-74695-9_23. [Online]. Available: https://doi.org/10.1007/978-3-540-74695-9_23.

[14] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," *CoRR*, vol. abs/1711.07128, 2017. arXiv: 1711.07128. [Online]. Available: http://arxiv.org/abs/1711.07128.

[15] L. Deng, J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams, Y. Gong, and A. Acero, "Recent advances in deep learning for speech research at microsoft," IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 2013. [Online]. Available: https://www.microsoft.com/en-us/research/publication/recent-advances-in-deep-learning-for-speech-research-at-microsoft/.

[16] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, F. R. Bach and D. M. Blei, Eds., ser. JMLR Workshop and Conference Proceedings, vol. 37, JMLR.org, 2015, pp. 1737–1746. [Online]. Available: http://proceedings.mlr.press/v37/gupta15.html.

[17] D. Das, N. Mellempudi, D. Mudigere, D. D. Kalamkar, S. Avancha, K. Banerjee, S. Sridharan, K. Vaidyanathan, B. Kaul, E. Georganas, A. Heinecke, P. Dubey, J. Corbal, N. Shustrov, R. Dubtsov, E. Fomenko, and V. O. Pirogov, "Mixed precision training of convolutional neural networks using integer operations," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=H135uzZ0-.

[18] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=r1gs9JgRZ.

[19] B. Fleischer, S. Shukla, M. Ziegler, J. Silberman, J. Oh, V. Srinivasan, J. Choi, S. Mueller, A. Agrawal, T. Babinsky, N. Cao, C. Chen, P. Chuang, T. Fox, G. Gristede, M. Guillorn, H. Haynie, M. Klaiber, D. Lee, S. Lo, G. Maier, M. Scheuermann, S. Venkataramani, C. Vezyrtzis, N. Wang, F. Yee, C. Zhou, P. Lu, B. Curran, L. Chang, and K. Gopalakrishnan, "A scalable multi- teraops deep learning processor core for ai trainina and inference," in *2018 IEEE Symposium on VLSI Circuits*, 2018, pp. 35–36.

[20] M. Harris, *Mixed-precision programming with cuda 8*. [Online]. Available: https://devblogs.nvidia.com/mixed-precision-programming-cuda-8/.

[21] N. Wang, J. Choi, D. Brand, C. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," *CoRR*, vol. abs/1812.08011, 2018. arXiv: 1812.08011. [Online]. Available: http://arxiv.org/abs/1812.08011.

[22] "Ieee standard for binary floating-point arithmetic," *ANSI/IEEE Std 754-1985*, pp. 1–20, 1985.

[23] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[24] *7 types of neural network activation functions: How to choose?* [Online]. Available: https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/.

[25] h. Avinash Sharma V, *Understanding activation functions in neural networks*. [Online]. Available: https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0.

[26] V. Vaghela, *Activation functions and it's types-which is better?* [Online]. Available: https://www.neuronactivator.com/blog/journey-of-an-ai-enthusiast-second-phase?c=1.

[27] *Deep learning: Overview of neurons and activation functions*. [Online]. Available: https://mc.ai/deep-learning-overview-of-neurons-and-activation-functions/.

[28] h.-a. Wasi Ahmad, *What is the intuition of using tanh in lstm*. [Online]. Available: https://stackoverflow.com/questions/40761185/what-is-the-intuition-of-using-tanh-in-lstm.

[29] T. Yang, Y. Wei, Z. Tu, H. Zeng, M. A. Kinsy, N. Zheng, and P. Ren, "Design space exploration of neural network activation function circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 10, pp. 1974–1978, 2019.

[30] M. Paschou, "Asic implementation of lstm neural network algorithm," Master's thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2018, p. 79.

[31] M. Pascale, *Microcontrollers cordic methods*. [Online]. Available: https://www.drdobbs.com/microcontrollers-cordic-methods/184404244.

[32] P. Gysel, J. J. Pimentel, M. Motamedi, and S. Ghiasi, "Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks," *IEEE Trans. Neural Networks Learn. Syst.*, vol. 29, no. 11, pp. 5784–5789, 2018. DOI: 10.1109/TNNLS.2018.2808319. [Online]. Available: https://doi.org/10.1109/TNNLS.2018.2808319.

[33] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *CoRR*, vol. abs/1806.08342, 2018. arXiv: 1806.08342. [Online]. Available: http://arxiv.org/abs/1806.08342.

[34] W. Dally, *High-performance hardware for machine learning.* [Online]. Available: https://media.nips.cc/Conferences/2015/tutorialslides/Dally-NIPS-Tutorial-2015.pdf.

[35] O. Andersson, B. Mohammadi, P. Meinerzhagen, A. Burg, and J. N. Rodrigues, "Ultra low voltage synthesizable memories: A trade-off discussion in 65 nm cmos," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 63, no. 6, pp. 806–817, 2016.