Efficient On-chip Voice Activity Detector Exploiting Temporal Sparsity

Master Thesis

Yuhang Jian

Delft University of Technology

TU Delft

Efficient On-chip Voice Activity Detector Exploiting Temporal Sparsity

by

Yuhang Jian

Student Name	Student Number
(Yuhang Jian)	5447151

Student number: 5447151
Project duration: Nov, 2022 - Dec, 2023

Thesis committee:

Prof. Dr. Ir. Rene Van Leuken	TU Delft, chairman
Dr. Ir. Chang Gao	TU Delft, supervisor
Dr. Ir. Charlotte Frenkel	TU Delft

TU Delft

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Chang Gao, for his guidance and support throughout this work. I feel lucky to have chosen him as my supervisor. He led my graduate study work step by step and provided much technical support and guidance to help me conduct better research. He taught me about the technical field and how to be a better engineer. This has benefited me a lot in my master's life.

I would also like to thank my two other master committee members, the chairman, Prof. Rene van Leuken, and the external member, Dr. Charlotte Frenkel. They provided valuable feedback on this work and provided many specific optimization methods for my future work. This support helps me make better plans for my future career.

Next, I would like to thank my colleagues in the research group office, Dr. Anil Kumaran and Dr. Gagan Singh. When I was overcoming difficult challenges in this work, they taught me many methods for conducting academic research, as well as some basic cross-field expertise, which greatly improved the efficiency of my research work.

Then, I definitely thank my good brother, Fanyuan Li, and my parents. Their concern for my daily life helps me to have the courage to face difficulties. Thank you for the encouragement when I was at a low point in my master's life.

Finally, I would like to thank myself. Thanks for my hard work as a graduate student at TU Delft. I will continue to study in the future and grow into an excellent engineer.

Yuhang Jian
Delft, December 2023

Abstract

Voice activity detection (VAD) is the prevailing approach to extracting meaningful speech information from the pervasive noise found in the physical environment. Presently, deep neural networks (DNN) are widely employed as the classifier component in Voice Activity Detection (VAD) systems. However, conventional deep neural networks, like fully connected (FC) deep neural networks, encounter the challenge of excessive computational complexity. This heightened complexity can result in diminished computing efficiency, unnecessary utilization of hardware resources, and redundant power consumption. To address the inefficiency issue from computational complexity, this study introduces a novel neural network architecture named DeltaFC. This architecture attains an operation time latency of less than 1 ms for each 30ms voice segment, resulting in a 54% reduction in latency compared to the baseline fully connected (FC) model. In software design, this study tackles the issue by compressing and encoding time-series information using the Delta algorithm, with the objective of introducing temporal sparsity. Based on the software results, the neural network surpasses both the baseline fully connected (FC) and LSTM [14] models in AUC (area under the curve), with accuracy at a lightweight parameter scale. In hardware design, this study reproduces the neural network software design into FPGA hardware RTL design, implementing a lightweight digital IP core. This digital IP core accelerates neural network operations in hardware by the deployment of Delta and CSR algorithms. Compared with not introducing temporal sparsity, the computing efficiency increases by approximately 85% with 0.5% loss in accuracy. This substantiates that within the domain of lightweight neural networks containing fewer than 30,000 parameters, the DeltaFC network proposed in this study is more suitable for Voice Activity Detection (VAD) when compared to fully connected (FC), LSTM [14], and other baseline network architectures.

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
VAD	Voice Activity Detection
STE	Short Time Energy
ZCR	Zero Crossing Rate
CNN	Convolutional Neural Networ
TD-CNN	Time-Domain Convolutional Neural Network
M-AECNN	Masked Auditory Encoder-based Convolutional Neural Network
FFT	Fast Fourier Transform
SNR	Signal Noise Ratio
GMM	Gaussian Mixed Model
LLR	Log Likelihood Ratio
FPGA	Field Programmable Gate Array
BNN	Binarized Neural Network
MSE	Mean Square Error
LSTM	Long Short Term Memory
CSR	Compressed Sparse Row
GCD	Greatest Common Divisor
DNN	Deep Neural Network
FC	Full Connection
RAW	Read After Write
RW	Read Write

List of Figures

List of Tables

1 Introduction

1.1. Motivation

In the real world, the surrounding environment is replete with various noises. Accurate interpretation of audio information, especially human sounds in a noisy environment, requires the use of Voice Activation Detection (VAD) to extract meaningful acoustic information from the ambient noise, as shown in Fig. 1.1. A typical VAD system consists of a feature extractor and a classifier. VAD classifier implementation methods encompass both conventional audio sensing classification techniques and deep learning methods based on deep neural networks (DNN). This work primarily delves into the DNN method.

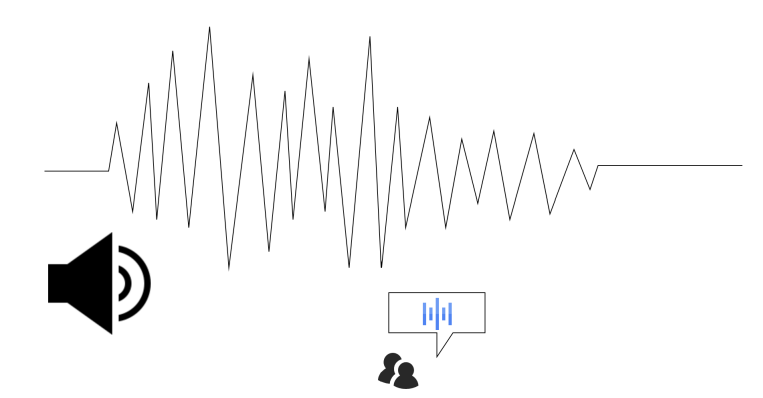


Figure 1.1: Human sounds in noisy environment

Conventional deep neural networks (DNN) face challenges of high computational complexity when dealing with large models, and their architectures are not well-suited for processing sequential information such as audio. Taking the example of a conventional fully connected (FC) neural network, the participation of numerous neurons and synapses in matrix multiplication and accumulation (MAC) operations elevates the computational complexity of the neural network [24], resulting in diminished computational efficiency, especially on hardware design. Therefore, a novel lightweight neural network named DeltaFC is proposed to address the issue of inefficient computational efficiency stemming from resource-intensive calculations in traditional neural networks, especially in hardware design.

1.2. Problem Description

In the domain of Voice Activity Detection (VAD), the output complexity is very low, characterized by a binary format (1 denotes "speech" and 0 signifies "noise," as shown in Fig. 1.2). Furthermore, after feature extraction, the input audio exhibits inherent temporal correlations along the time axis. Temporal correlation implies that the variations in input features at each time step

are insignificant. Inducing temporal sparsity through the intrinsic temporal correlations of the audio signal has become the central focus of acceleration in this work.

Following the induction of temporal sparsity, the input matrix exhibits sparsity, which can be accelerated through the application of partial convolution. The adoption of partial convolution in VAD speech recognition can be beneficial. Partial convolution selectively processes specific features by leveraging the temporal sparsity induced by the temporal correlations in the input audio signal. As a result, it is recommended to devise a novel neural network, such as DeltaFC, proposed in this work, which introduces more temporal sparsity in the input features. On the other hand, partial convolution can be implemented using the CSR algorithm to accelerate the calculation process.

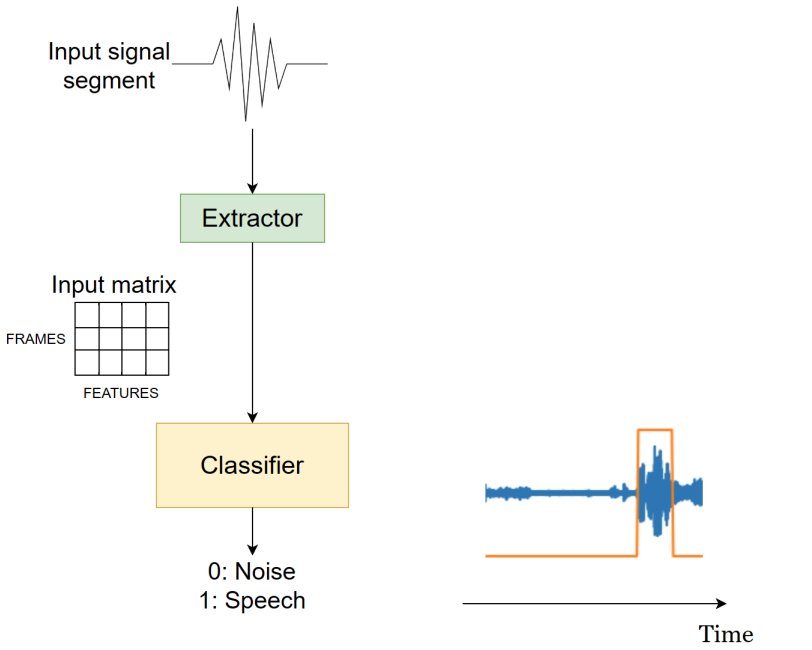


Figure 1.2: Always-on voice activity detection process

In software design, the concept of sparsity is introduced as a foundational principle. Sparsity refers to zero values and intermediate variables that can be converted to zero values in the model, bypassing unnecessary calculations involving zero-multiplied terms, as depicted in Fig. 1.3. Integrating sparsity enhances computational efficiency and resource utilization in sparsity-aware neural network accelerators [45]. Therefore, this study will use thresholds to filter features with insufficient delta values between the current and next audio frame, aiming to introduce more sparsity.

In hardware design, this work adopts a unique hardware architecture to implement the CSR and Delta algorithms [31], thereby implementing the DeltaFC digital IP core. In pursuit of a lightweight neural network, the strategy incorporates a multi-channel parallel DSP architecture for foundational matrix partitioning and MAC operations. Furthermore, to boost the computational speed of the neural network, this work employs a multi-stage pipeline architecture to deploy the CSR and Delta algorithms, directly accelerating neural network operations at the hardware design level.

1.3. Objective

The main objectives of this work are as follows:

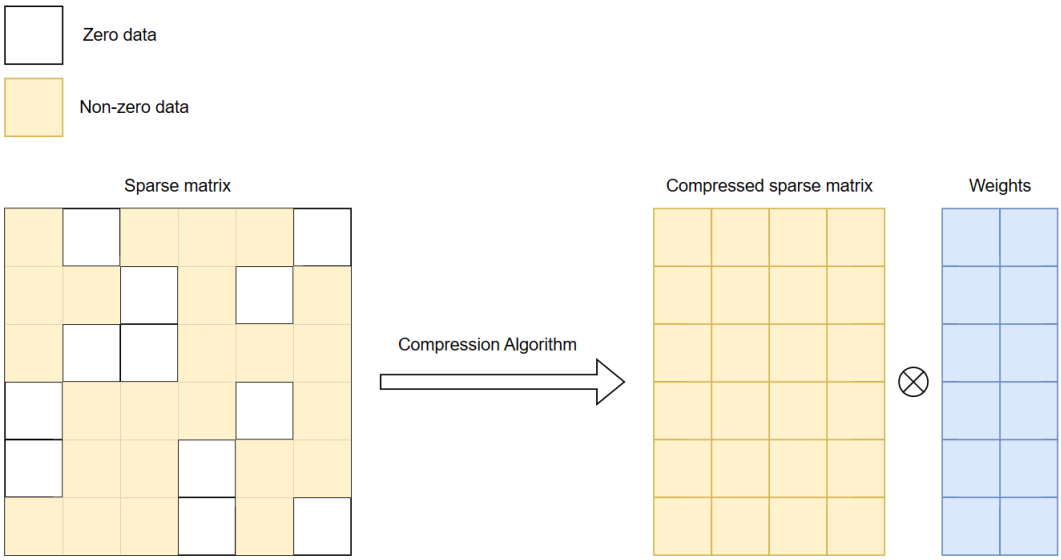


Figure 1.3: Structure of sparse matrix calculations.

- In software design, to train a lightweight (for example, the number of neural network parameters is less than 30k) DeltaFC DNN, which induces sufficiently high accuracy (Over 90%) and temporal sparsity (Over 50%).
- In software design, compared with the baseline neural network in terms of performance, the DeltaFC DNN shows advantages in AUC, accuracy, and parameter scale.
- In hardware design, to implement a lightweight digital IP that can deploy and accelerate the DeltaFC DNN, and it is supposed to be implemented by low resources consuming (below 50%) and run in high frequency (100MHz).
- In hardware design, the designed hardware module can achieve the expected acceleration, and the function verification is correct through synthesis, implementation, and routing.

1.4. Contributions

The main contributions of this work are as follows:

- Proposing a new neural network based on FC. This novel neural network named DeltaFC leverages the Delta algorithm to introduce temporal sparsity, enhancing the computational efficiency.
- Implementing the acceleration of the neural network on FPGA RTL design by the deployment of Delta and CSR algorithms. Deploying the Delta and CSR algorithms in FPGA RTL design is implemented through a multi-channel parallel architecture. This approach enables hardware acceleration, allowing for efficient processing of neural network computations on the hardware design.

1.5. Outline

The structure of this thesis is shown as follows:

- **Chapter I: Introduction**

  Brief introduction of the motivation for this work, the objectives to be achieved, and the contribution made.

- **Chapter II: Background**

  Introducing the feature extraction methods and components both in software and hardware design, encompassing elements like the neural network structure and optimization methods in software design, as well as the hardware architecture, resources, and FPGA units in hardware design.
- **Chapter III: Proposed Methodology**

  Outlining the implementation of Delta Deep Neural Network (DNN) in both software and hardware design. This encompasses the neural network architecture in software and hardware design and the deployment of CSR and the Delta algorithms.
- **Chapter IV: Results**

  Setup of the experimental framework for both software and hardware design. Subsequently, analyze the obtained experimental results and provide corresponding evaluations.
- **Chapter V: Conclusion**

  Summarize the objectives achieved in this work and list possible future works.

2
Background

The implementation of Voice Activity Detection (VAD) is a multifaceted process, comprising several distinct steps. The initial phase involves audio feature extraction, followed by audio VAD neural network classification. Subsequently, there's the implementation of audio VAD in hardware, culminating in the presentation of audio output results. Given the intricacy of this process, it is essential to delve into the background knowledge of each step. The upcoming chapter will provide a comprehensive introduction to each of these steps, offering a detailed understanding of the intricate VAD implementation process.

2.1. Feature Extraction

The initial phase in Voice Activity Detection (VAD) is feature extraction. As mentioned earlier, various methods are commonly employed for VAD feature extraction, including Short Time Energy (STE), Zero Crossing Rate (ZCR), Spectral Entropy, and Mel Frequency Cepstral Coefficients (MFCC), among others. In the subsequent sections, these feature extraction methods will be introduced in detail.

2.1.1. Short Time Energy (STE)

Since voice signals vary significantly over time, voice features can be distinguished by the short-term energy of each frame we defined. The formula of short-time energy is as follows [46]:

$$E(n) = \sum_{i=1}^{N} x_n(i)^2 \tag{2.1}$$

N is frame length, x_n(i) is the speech signal of frame n, and E(n) is the short-time energy of frame n. This is the simplest and most direct method, not only used for feature extraction but also for classification. However, STE is very sensitive to large amplitude signals and will likely exceed the threshold when the SNR is large. Therefore, STE is often used combined with other methods.

2.1.2. Zero Crossing Rate (ZCR)

As the name implies, ZCR means the frequency at which the signal statistically exceeds 0 or not. It can represent the existence and non-existence of discrete-time signals. The formula of ZCR calculation is as follows [46]:

$$Z_n = \sum_{m=-\infty}^{\infty} |sgn[s(m)] - sgn[s(m-1)]| \tag{2.2}$$

In the above equation, sgn(x) is the sgn function, which is a special form of the tanh function. And sgn(x) function is as follows.

$$|sgn(x)| = \begin{cases} 1 & , x \geq 0 \\ -1 & , x < 0 \end{cases} \tag{2.3}$$

s(m) is the speech signal of the frame. The ZCR is often combined with STE characteristics for endpoint detection. Following the detection of endpoints, a small sample of silence interval before the commencement of speech signal is taken, and STE and ZCR are calculated [28].

2.1.3. Spectral Entropy

Spectral entropy serves as an effective measure of signal confusion. In the case of white noise, characterized by a disordered spectrum, the spectral entropy is high. Conversely, speech signals exhibit a more organized spectrum, resulting in lower spectral entropy. This inherent difference allows spectral entropy to distinguish between noise and speech signals effectively. The fundamental concept behind utilizing spectral entropy in voice activity detection lies in the observation that the signal spectrum is less organized during non-speech intervals compared to speech intervals [34]. The formula for calculating spectral entropy is as follows:

$$H(n) = -\sum_{k=0}^{N/2} p_n(k) log_2 p_n(k) \tag{2.4}$$

P is the signal spectral line density. Through spectral entropy calculation, the signal can be extracted from the mixed signals by the entropy size.

To compare feature extraction results, Thein Htay Zaw1 [46] and Nu War raised different combinations of ZCR, STE, and LPE (Linear Prediction Error), and spectral entropy methods to evaluate the maximum and minimum accuracy. They kept the same parameters, such as sampling frequency for each combination, and finally, they obtained the table of comparison results, as shown in Tab. 2.1. In most cases, those feature extraction methods are used not independently but in combination.

Features	Minimum Accuracy
ZCR + Energy + LPE + Spectral Entropy (Proposed Algorithm)	90.35%
ZCR + Energy + Spectral Entropy	89.33%
ZCR + Energy + LPE	89.58%
Features	Maximum Accuracy
ZCR + Energy + LPE + Spectral Entropy(Proposed Algorithm)	97.22%
ZCR + Energy + Spectral Entropy	95.44%
ZCR + Energy + LPE	96.20%

Table 2.1: COMPARISON RESULTS OF THREE ALGORITHMS [46]

2.1.4. Mel Frequency Cepstral Coefficients (MFCC)

The traditional feature extraction method is the application of the speech signal in time domain parameters, such as ZCR, STE, and so on. Except for the 3 fundamental methods introduced previously, MFCC is one of the most effective features for speaker recognition, especially in recent years of research. MFCCs are based on the known variety of the human ear's critical bandwidths with frequency [7]. The perception of the human ear is sensitive to low frequency and rather ambiguous to high frequency. Therefore, Mel frequency simulates the hearing characteristics of the human ear, converts the spectrum to a non-linear spectrum based on

Mel frequency coordinates, and then converts it to the spectrum domain. Mel frequency and actual frequency can be transformed by the following formula [43]:

$$Mel(f) = 2595lg(1 + f/700) \tag{2.5}$$

The MFCC feature extraction process can be divided into steps as follows:

1. Pre-emphasis

   The purpose of pre-emphasis is to boost the high frequencies to flatten the signal spectrum, which can be regarded as the signals through the high-pass filter. This will facilitate the calculation of SNR.

$$H(z) = 1 - \mu z^{-1} \tag{2.6}$$

2. Frame

   Due to the non-stationary and short-term stationary characteristics of the speech signal, the speech signal is divided into frames.

3. Windowing

   Multiplying each frame by a Hamming window to increase the continuity of the left and right of the frame, as follows [43]:

$$S'(n) = S(n)W(n) \tag{2.7}$$

   where

$$W(n) = 0.54 - 0.4cos[2\pi n/(N-1)] \tag{2.8}$$

4. Fast Fourier Transform (FFT)

   To observe the signals' characteristics more conveniently, transform the time domain signal into frequency domain energy distribution. Then, the spectral energy of the speech signal is calculated.

5. Triangular window band-pass filter

   Passing the energy spectrum through a set of Mel-scale triangular window band-pass filters. Triangular window band-pass filter can smooth the spectrum and remove the effects of harmonics, emphasizing the formants of the original speech [43]. The filters are dense, and the threshold is low when frequencies are high, and filters are sparse, and the threshold is high when frequencies are low. This is also the auditory characteristic of the human ear.

The signal energy output by each band-pass filter is the basic feature of the signal, which can be used as the input feature of speech after further processing.

2.2. Traditional Classification Methods

The second step for VAD is classification. The commonly used classification methods include conventional methods such as threshold-based methods and Gaussian-based statistical model methods such as WebRTC.

2.2.1. Threshold-based classification

The energy spectrum of the signals can be obtained through feature extraction. The next step is classifying the noise and the non-noise based on the energy threshold. The threshold values are often preset; the values can be calculated in advance or pre-obtained according to the features of the training data set by neural network [22]. For scenes with varying noise, an adaptive threshold is required. Assume η is the adapted threshold energy, which varies from η_0 to η_1, and E_1 is the energy maximum, E_2 is the energy minimum. So η can be calculated as follows:

$$\eta = \begin{cases} \eta_0 & E < E_0 \\ \frac{\eta_0 - \eta_1}{E_0 - E_1}(E - E_0) + \eta_0 & E_0 \leq E \leq E_1 \\ \eta_1 & E > E1 \end{cases} \tag{2.9}$$

The advantage of this threshold preset is that it is simple and easy to understand. However, the false alarm rate will increase for short-time spiking signals or signals with low SNR because of the stubbornly preset threshold. Therefore, with time-varying, a more smoothy strategy can be used as follows:

$$\eta = \alpha\eta + (1 - \alpha)\eta_{new} \tag{2.10}$$

2.2.2. Gaussian-based statistical model WebRTC classification

Gaussian Mixed Model (GMM) is the statistical Model commonly used in speech algorithms, and WebRTC uses the Gaussian-based statistical model method combined with extractors such as STE and ZCR mentioned before. This method assumes that the audio signal has Gaussian model characteristics first and then uses the Gaussian model to fit two models of speech and noise to distinguish the two signals, as shown in Fig. 2.1.

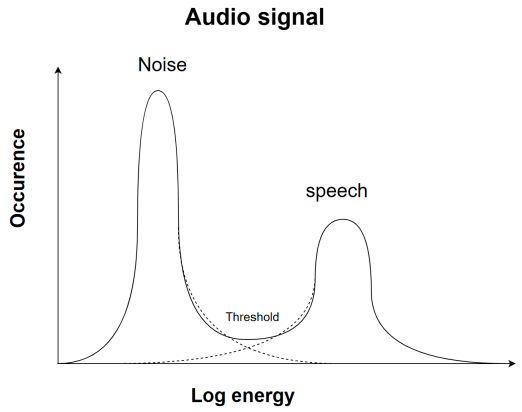


Figure 2.1: Gaussian model fitting to speech and noise

WebRTC uses the GMM statistical model to make VAD classification, divides 0 4kHz into several subbands, and uses these. The subband energy of the frequency band is used as GMM features. The WebRTC process is rough as follows [37]:

1. Initialization and configuration, including coefficients of the filters and parameters of VAD hangover;
2. Downsampling the speech signals to 8kHz and calculating the energy of each subband as a GMM feature (STE method for each subband);

3. Calculating the Gaussian probability corresponding to each subband and multiplying it with the weight of the subband as the final probability of speech/noise;
4. Calculating the LLR (log-likelihood ratio, LLR) of each subband and the LLR of each subband will be compared with the threshold as a local VAD decision. When one of the classification results has speech, it is determined that the current frame is a speech frame;
5. Smoothing the result;
6. Adaption, which means updating GMM parameters and iteration;

Gaussian-based statistical model WebRTC classification has significant improvements in accuracy and sensitivity. Its application scenarios are very wide, especially in conventional VAD methods. The VAD that Google developed for the WebRTC project is reportedly one of the best available, being fast, modern, and free [12]. Since it uses fixed point operations and is optimized for real-time use for web transmission [35], the Gaussian-based statistical model WebRTC is a good baseline for VAD implementation. In chapter 4, WebRTC will serve as a benchmark for the VAD classification of this work.

In Chapter 4, WebRTC will serve as the benchmark for the VAD classification in this work, providing a robust reference against which the outcomes of the proposed approach of this work can be measured.

2.3. Neural Network-Based Classification Methods

2.3.1. Novel method: Deep Neural Networks

Conventional VAD classification methods, such as the WebRTC model classification method, can distinguish noise and speech signals well in scenarios with high SNR and are also great in real-time response. However, for low SNR, the performance of conventional VAD classification methods is not well. However, deep learning classification performs excellently in speech recognition for a novel method. The commonly used structure of the Deep learning classification method is DNN. DNN-based VAD systems can fuse the advantages of multiple features much better than traditional VADs [40].

DNN structure and the training-testing process will be briefly introduced. Assume a feedforward FC DNN has multiple fully connected layers, each with multiple neurons. Features extracted from audio signals are input from the left fully connected layer of this DNN and will output from the right fully connected layer of this DNN. In DNN-based VAD classification, the output has only 1 judgment bit to distinguish whether the input signal is a speech signal or noise. In the training process, when the dataset is sufficiently rich, this model will be trained repeatedly to update parameters, including each layer's weights and biases. After dozens or even hundreds of training epochs, the trained-well model will be called by the final testing process. The results, such as the accuracy and loss of the model output, will be evaluated during the testing period.

Compared with conventional DNN classification, the classification thresholds of the DNN-based model classifier are not preset or calculated but are updated by the model's parameters under the massive dataset. With sufficiently rich and well-characterized datasets, the data output by the DNN model will fit the real-world acoustic data.

However, deep learning classification also has several drawbacks, but the solutions are given below:

1. Computing resources are usually more expensive than traditional methods;

   The problem of high consumption of computing resources can be optimized by changing the network model structure, model pruning, and compression [26];

2. The model's generalization ability is usually worse than that of traditional methods.

   The problem of the weak generalization ability of the model can be improved by using various regularization methods, expanding the training data set, and replacing the network input features.

2.3.2. Introduction of Neural network

In this work, the classification method adopts the neural network method. There are many types of neural networks, and based on the aforementioned type of connections, neural networks can be classified into two types: feedforward network and feedback network [48]. The feedforward neural network can be subdivided into a fully connected layer neural network and a convolutional neural network. The representative network of feedback neural networks is the Recurrent Neural Network(RNN). In the field of voice activity detection, the traditional method is to divide a piece of voice into several frames, extract the features of each frame, and input them into the traditional neural network for training. The most traditional form of neural network is a DNN composed of fully connected layers. It performs an inner product of each audio frame's features and network parameters. However, calculating such many parameters is inefficient in hardware implementation.

In later research on audio feature extraction, it was observed that each frame of the audio spectrum, post Mel-Frequency Cepstral Coefficients (MFCC), exhibits visual features similar to those found in images. This discovery led to applying a bottom-up local visual feature extraction mechanism to process audio spectrograms, treating them like image features. Abdel-Hamid et al. [1] applied their functionally extended Convolutional Neural Networks (CNNs) to sound spectrogram inputs. They showcased that their CNN architecture outperformed earlier basic forms of FC DNNs in tasks such as phone recognition and large vocabulary speech recognition. However, CNN, which has an advantage in speech recognition in audio processing based on the MFCC spectrum, is ineffective in voice activity detection. On the one hand, CNN will overkill and lose some key features in the pooling operation, and these lost features play a decisive role in identifying whether this audio is speech or noise (such as human voice in a noisy environment). On the other hand, the computational and storage load of convolutional neural networks is too large, which conflicts with the objective of this work, which requires lightweight networks.

To achieve neural network acceleration in hardware implementation and the processing of all keyframes in the audio, this study proposes a new fully connected (DeltaFC) DNN. In voice activity detection, the audio features after MFCC are coherent without instantaneous jumps. Leveraging this characteristic, the DeltaFC layer utilizes the feature difference (Δx) between adjacent frames as input features instead of the original features (x) of the audio. This section will introduce the baseline models of FC and Recurrent Neural Network (RNN). The subsequent chapter 3 will delve into the details of the DeltaFC layer, elucidating its innovative approach in enhancing hardware-accelerated neural network processing for voice activity detection.

2.3.3. Fully connected neural network(FC)

The FC neural network represents a classic application of the multilayer perceptron (MLP). This neural network type embodies the fundamental characteristics of a DNN, featuring an input layer, multiple hidden layers, and an output layer. It is called the FC because each network node is fully connected to all the nodes of the previous layer, which is used to integrate the features extracted earlier. Fig. 2.2 shows the FC neural network structure.

The weight matrix in the FC neural network is a two-dimensional matrix comprising weights and an additional line for bias. Initially, FC utilizes random numbers to determine the connec-

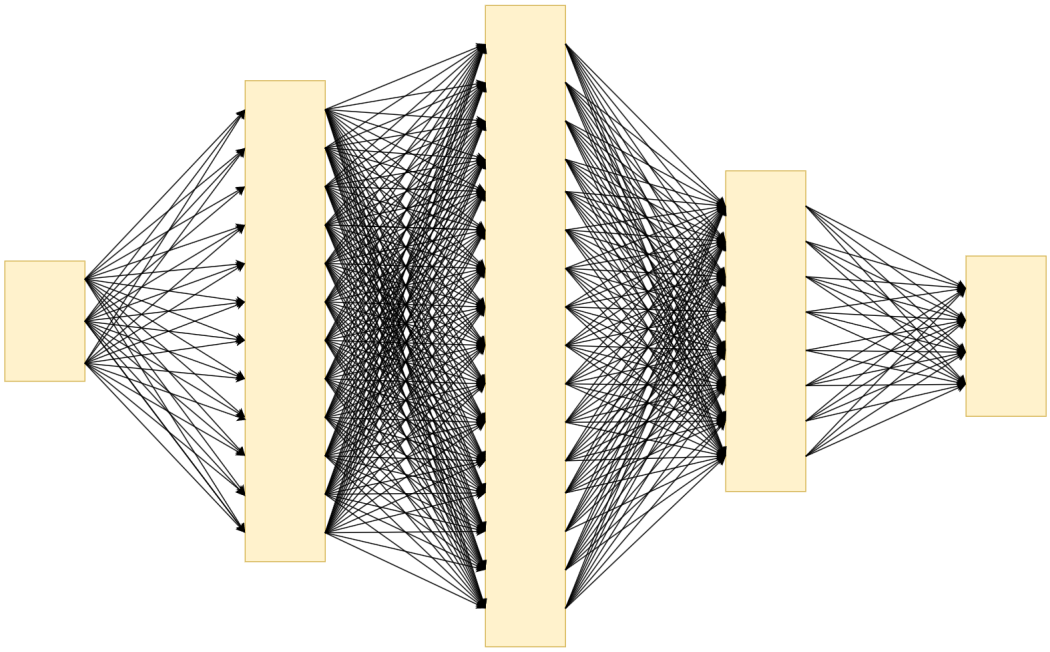


Figure 2.2: Basic structure of 5-layer DNN.

tion weights between the input and hidden layers. Subsequently, error correction learning techniques, such as the backpropagation (BP) algorithm [47], are employed to iteratively adjust the connection weights between the hidden layer and the output layer. This iterative learning process enables the network to fine-tune its parameters based on the observed errors, enhancing its ability to capture underlying patterns in the data accurately. Assuming an FC has input nodes of 3 and output nodes of 3, then input can be assumed as x_1, x_2, x_3, while output as a_1, a_2, a_3. And weights can be assumed as $W_{i,j}$(i, j represents the row and column of this FC matrix, as 3 and 3 in this FC), then bias as b_1, b_2, b_3. The output can be calculated as Eq. 2.2.

$$\begin{aligned} a_1 &= W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1 \\ a_2 &= W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2 \\ a_3 &= W_{31}x_1 + W_{32}x_2 + W_{33}x_3 + b_3 \end{aligned} \tag{2.11}$$

Eq. 2.11 can be transformed into matrix calculations, as Eq. 2.12.

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} & W_{13} & b_1 \\ W_{21} & W_{22} & W_{23} & b_2 \\ W_{31} & W_{32} & W_{33} & b_3 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{bmatrix} \tag{2.12}$$

The FC operation calculations are evidently straightforward, primarily involving matrix operations. This simplicity renders the modification of the FC flexible, and the computational process remains linear and robust. The hardware design used in this work is suitable for this mode of operation, which means the large-scale matrix operation adopts the multi-channel pipeline parallel structure design. However, the amount calculation operations of the FC is complex, especially when the network's input and output reach the size of 64, 128, which poses a challenge to the lightweight hardware design of this work. Therefore, this work improves based on FC and proposes the DeltaFC neural network, which will be introduced in detail in Chapter 3.

2.3.4. Recurrent neural network(RNN)

In contrast to the feedforward neural network represented by FC, the Recurrent Neural Network (RNN) stands out as a representative of the feedback neural network. The typical structure of the RNN unit is shown in Fig. 2.3.

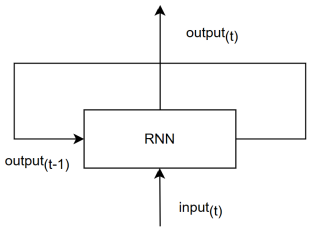


Figure 2.3: typical structure of RNN unit

It's obvious to see that RNN has a strong temporal correlation since the output of each current state is related to the output of the previous state. Here is the reference of calculation process derivation of RNN by Robin M. Schmidt [36] from Eberhard-Karls-University Tübingen. Assuming time, t denotes the current state, while t-1 denotes the previous moment. X represents input features, and H represents the hidden layer. The hidden layer computation in RNN is similar to the MAC operation in FC. However, due to the temporal correlation in RNN, each hidden layer output equals the sum of the hidden output from the current layer and the hidden output from the previous layer, as shown in the Eq. 2.13 and Eq. 2.14.

$$H_t = \phi_h(X_t W_{xh} + H_{t-1} W_{hh} + b_h) \tag{2.13}$$

$$O_t = \phi_o(H_t W_{ho} + b_o) \tag{2.14}$$

From this derivation process, it can be clearly observed that RNN uses the output of the previous state as a part of the current state's input (usually as bias). If the input is input frame by frame in a sequential manner, the RNN unfold calculation structure is shown in Fig. 2.4.

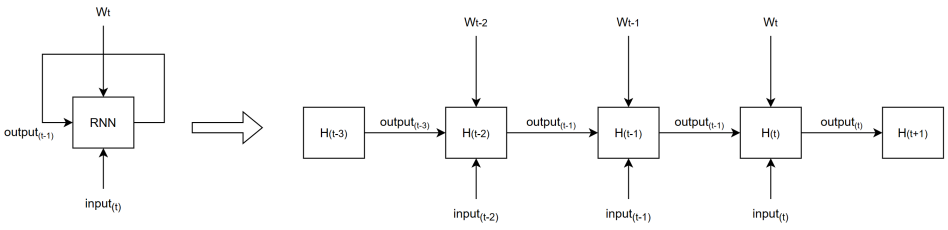


Figure 2.4: The standard RNN and unfolded RNN.

Due to the special structure of RNN processing information from the time axis, RNN is very suitable for processing time series information, such as audio, video, .etc. Therefore, RNNs find applications in natural language processing, speech recognition, computer vision, and Video processing [36]. However, vanishing or exploding gradients are a key problem of RNNs. Since the input of each current temporal layer is associated with the output of the previous temporal layer, if the weight of the matrix in the operation of the previous layer is small (for example, less than 1), then this will affect the gradient of the next layer which will decrease if in continuous. The gradient will approach 0 (gradients vanishing) under the iteration of multiple small weight frames. On the contrary, it will cause a gradient explosion. These issues prompted

the adoption of Long Short-Term Memory units (LSTMs), specifically designed to address the gradient vanishing and explosion problems [36].

Compared with RNN, LSTM introduces control gates to manage the stream information from the data selection $C_{(t-1)}$ and $C_{(t)}$, the input $Input_{(t)}$, the output of the previous time step $H_{(t-1)}$, and the output$H_{(t)}$, as shown in Fig. 2.5. These control gates dynamically regulate data selection $C_{(t)}$ within the network. For instance, the forget gate decides whether to retain the output of the previous layer $H_{(t-1)}$. Each gate has its own corresponding weights and biases because of the control gate selection. By selectively choosing data through control gates, gradient disappearance, and explosion issues can be mitigated as much as possible.

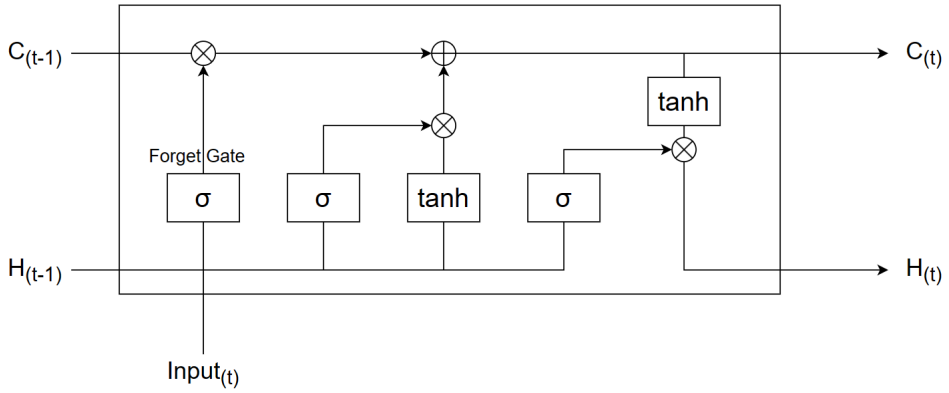


Figure 2.5: The structure of LSTM

It is worth noting that because of the introduction of control gates, the computational complexity of LSTM is very high. Furthermore, the iterative structure of LSTM determines that it is unsuitable for lightweight parallel processing. This leads to the fact that the hardware implementation of LSTM will consume a lot of resources, such as a large number of FIFOs and BRAMs used to cache the memory, which is contrary to the objective of the lightweight design of this work. However, the original RNN also faces the dilemma of acceleration induced by temporal sparsity disabled. Therefore, in response to this problem, the DeltaFC proposed in this work uses FC as the baseline and introduces a iterative structure similar to RNN, and Delta algorithm to implement lightweight operations based on memory processing of time series information. Delta-Fully-Connected (DeltaFC) will be introduced in Chapter 3.

2.4. Neural Network Optimization Methods

This work not only realizes the classification processing of the neural network but also accelerates its operation. In this section, two acceleration strategies for neural networks will be introduced. One is the neural network quantization that simplifies data structure, and the other is the sparsity strategy.

2.4.1. Quantization

In mathematics and digital signal processing, quantization is the process of mapping input values from a large set (usually a continuous set) into a smaller set (usually with a finite number of elements). In optimizing neural networks, this method converts floating-point numbers into fixed-point integer values. In the realm of neural network quantization, the weights and activation tensors are stored at lower bit precision compared to their standard training precision of 16 or 32 bits. Transitioning from 32 to 8 bits substantially reduces memory overhead, decreasing storage requirements by a factor of 4. Simultaneously, the computational cost

for matrix multiplication experiences a quadratic reduction, diminishing by a factor of 16, as highlighted in [29]. The quantization of the neural network will slightly impact the accuracy, but it will save a lot of hardware cache resources.

At the hardware design, quantization can save resource consumption of computing units to accelerate neural networks. In neural network calculations, the most basic operation is Multiply-Accumulate (MAC), as shown in Eq. 2.15.

$$y_n = W_n x_n + y_{n-1} \tag{2.15}$$

In this process, if the neural network is not quantized and the operation results of the previous layer are passed to the next layer without loss, then W, X, and bias must support floating-point logic. This means, on the one hand, not only more bits need to be stored in the data cache, but the bit width of the multiplier must also be expanded. On the other hand, since the number of bits of floating-point numbers is variable, the storage bit width is fixed. This leads to the fact that in the caching strategy, data with a short bit width often has vacant bits in the storage of data with a long bit width, which also causes a waste of storage resources. The computational cost of digital arithmetic is generally scaled either linearly or quadratically with the number of bits utilized. Consequently, fixed-point calculations are deemed more efficient than their floating-point counterparts [15]. In summary, unquantized neural networks are contrary to this work's objective of lightweight design. The implementation of specific floating-point data to fixed-point data transformation will be elaborated in Chapter 3.

2.4.2. Sparsity

Sparsity refers to the characteristic that the processed data presents several 0 values. Sparsity can be divided from the time and space dimensions into temporal and spatial sparsity. However, for some signals that do not contain sparsity, some operations can be used to make them sparse, that is, to create activation sparsity. This work will introduce the commonly used neural network sparsity applications, including activation, spatial, and temporal sparsity.

Activation sparsity

In natural language processing, the features of the audio signal contain inherent temporal correlation. However, activation sparsity makes these features lose information spontaneously by temporal correlation to achieve the activation of sparsity. So that is to say, sparse activation causes information loss; in the information encoding, more elements are zero or tend to zero. For example, the Sigmoid and ReLu functions are representative methods for increasing activation sparsity in neural networks.

Improving activation sparsity significantly improves the computational efficiency of neural networks. However, directly increasing the activation sparsity of audio leads to losing key features of the audio signal. Therefore, the increase in activation sparsity in the audio field often adopts indirect methods, such as introducing a memory layer to perform Delta operations on the features of the front and rear frames or to increase or reduce the dimensions of the features. Park et al. [32] also suggested that in algorithms where the kernel exhibits sparsity, the Compressed Sparse Row (CSR) format can be employed to pre-compress the sparse kernel before inference. This approach enhances activation sparsity without incurring any performance overhead.

Activation sparsity focuses more on operations that perform activations on sparsity. Therefore, making more features tend to 0 indirectly is the key to increasing activation sparsity. In this sense, the temporal sparsity and spatial sparsity mentioned later can also be classified as subsets of activation sparsity. For easier distinguishing, the sparsity used in this work is always expressed as temporal sparsity.

Spatial sparsity

Spatial sparsity is the statistical probability of 0 value after compressing the input data or network parameters in space (such as a two-dimensional matrix) as a whole or locally. Its input features lack temporal correlations between different frames, only maintaining the traditional topological relationships in regular space. Based on this characteristic, spatial sparsity is commonly applied to Convolutional Neural Networks (CNNs) that are well-suited for image processing. In Convolutional Neural Networks (CNNs), spatial activation sparsity is defined as the ratio of zero activations within a feature map to the total number of activation neurons in that feature map.

Spatial sparsity is applied in CNN to explain the acceleration principle. When an all-zero feature is input into each hidden layer of the CNN, the bias item in the matrix prevents the matrix from being an all-zero array, and these non-zero elements are considered to be at the ground state of spatial sparsity. In cases where the input array is sparse, only calculations related to the values of hidden layers that deviate from their ground state are processed, as described in [13]. This process is shown in Fig. 2.6, showcasing how active spatial locations evolve across the layers.

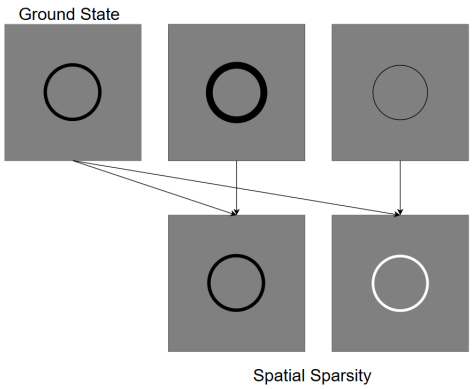


Figure 2.6: Spatial sparsity involves calculations only for hidden layer values that differ from their ground state

As the name suggests, the main field of application of spatial sparsity lies in the spatial sparse algorithmic acceleration. However, in the field of time-sequential audio processing, implementing spatial sparsity is unsuitable since it only accelerates the matrix operation in the traditional regular space. It is not sensitive to the time axis; therefore, temporal sparsity fits this work better.

Temporal Sparsity and Delta algorithm

Unlike spatial sparsity in static signal processing (such as image processing), temporal sparsity focuses more on streaming signal processing (such as audio-video processing).

Neural networks, including Recurrent Neural Networks (RNNs) and the DeltaFC model introduced in this study, encounter the challenge of heightened computational demands when processing time-series information. This is due to the network's computation of features at each time step, leading to a potential increase in computational complexity. Nevertheless, a temporal correlation exists within time-series information, manifesting as subtle changes in features between previous and current frames, as shown in Fig. 2.7. Therefore, if the temporal correlations in time-series information can be indirectly induced to generate temporal sparsity, it is possible to reduce computational complexity.

Temporal sparsity can be introduced by utilizing the Delta algorithm, which specifically focuses on the amplitude difference, denoted as Δy, between consecutive frames rather than

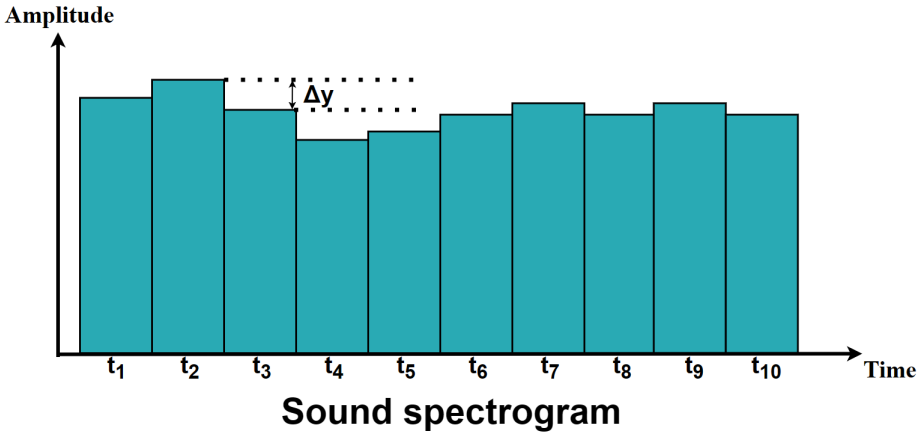


Figure 2.7: Sound spectrogram of voice segment features

processing the entire amplitude. When the amplitude difference Δy drops below a predefined threshold, it is designated as 0. In the time-series information, numerous such 'Δy' values exist, providing an opportunity to skip calculations involving these zero values [42], which can introduce temporal sparsity, as shown in Fig. 2.8. This process effectively reduces unnecessary operations, thereby boosting the speed of neural network operations. The detailed process of the delta algorithm will be discussed in Chapter 3. Temporal sparsity denotes the proportion of sparse input features along the time dimension. This is quantified as the ratio of the count of zero values to the count of input features, as shown in Eq. 2.16.

$$Temporal\_Sparsity = \frac{count\_of\_0\_values}{count\_of\_input\_features} \tag{2.16}$$

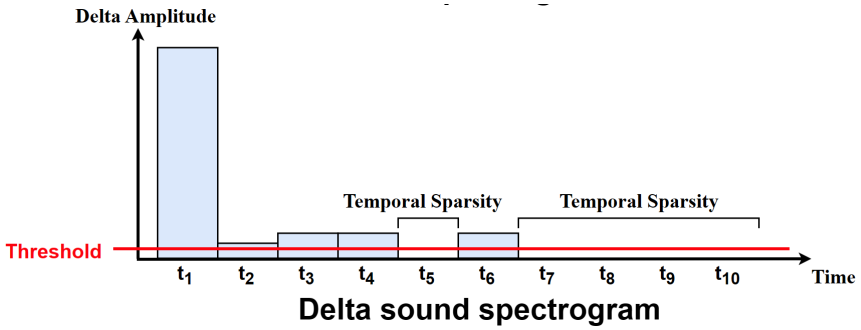


Figure 2.8: After Delta algorithm conversion, the sound spectrogram of voice segment features shows a certain temporal sparsity.

However, a single time-sparse application will face the problem of high resource consumption due to high spatial complexity. For example, in video processing, the features of each frame contain a complete stack of several images. A single temporal sparsity application cannot handle complex visual features on the time axis (dimensions out of bounds), which differs from the single-dimensional audio representation on the time axis. Huixiang Chen [5] introduced a 3D CNN structure incorporating a novel approach. This architecture leverages differential convolution along the temporal dimension, specifically operating on the temporal delta of Integrated Maps (imaps) for each layer. The computation is performed bit-serially, focusing solely on the significant bits of the temporal delta. This methodology is extended across all CNN convolution kernels in the network. Additionally, Chen proposed a control mechanism capable of dynamically switching between spatial delta dataflow and temporal

delta dataflow. This dynamic switching mechanism facilitates resource balancing between the network's temporal and spatial sparsity.

The integration of temporal and spatial sparsity is geared toward handling complex streaming media tasks, such as video processing. However, in this project's audio processing context, the focus only lies on temporal sparsity. The detailed analysis of this temporal sparsity application will be provided in Chapter 3.

2.4.3. Compressed Sparse Row (CSR) Algorithm

The resulting features matrix exhibits temporal sparsity after applying the Delta algorithm for sparsification. The sparse matrix needs to undergo matrix compression to accelerate neural networks, for which the CSR algorithm is employed. The CSR algorithm can effectively compress the sparse values of the matrix, making the matrix operation more compact [3].

The Compressed Sparse Row (CSR) algorithm offers an efficient approach for storing sparse matrices in a memory-efficient manner [3]. It uses three arrays to store the parameters of the compression matrix: data, indices, and indptr.

- data: Non-zero elements of each row of the matrix in row order;
- indices: The column index value of each non-zero element;
- indptr: The index value of the first non-zero element in each row in the data array

In this work, the concept of 'indptr' is not involved, so it will not go into detail. Here is a simple example of how the CSR algorithm stores a 4 × 4 sparsed matrix, as Fig. 2.9 shows.

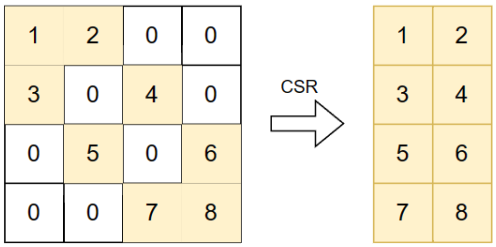


Figure 2.9: CSR algorithm conversion

From this conversion, it is easy to get the arrays of data and indices,

- data: [1,2,3,4,5,6,7,8];
- indices: [0,1,0,2,1,3,2,3]

Indeed, models in real-world applications are much larger, often extending to several thousand rows and columns. Given the substantial size of these models, the Compressed Sparse Row (CSR) algorithm becomes particularly crucial for handling large-dimensional sparse matrices. It is significant to compress matrix elements at scale through temporal sparsity, thereby accelerating data processing. In this work, the CSR algorithm holds particular importance. It plays an important role in boosting Multiply-Accumulate (MAC) operations involving sparse matrices in the hardware design of neural networks. Detailed insights into this utilization will be provided in Chapter 3.

2.5. Neural Network Hardware Architecture

The primary objective of this project is to implement neural networks in hardware design, which needs a focus on hardware architecture design as the initial step.

2.5.1. Multistage pipeline

The multistage pipeline is the basic structure of modern computer instruction execution. This work's neural network hardware IP also applies this classic modern computer architecture. Since the neural network hardware IP is integrated into the FPGA and has the characteristics of multi-channel parallel execution, when executing a series of instructions in a certain order, this sequential series of instructions can be dispersed in different clock cycles in the form of a pipeline. For example, if this hardware IP is supposed to execute 4 instructions named A, B, C, and D sequentially and loop for 2 rounds, then 8 clock cycles will be wasted if these 4 instructions are executed serially. However, if the pipeline is executed parallel, it will only consume 5 clock cycles, as shown in Fig. 2.10.

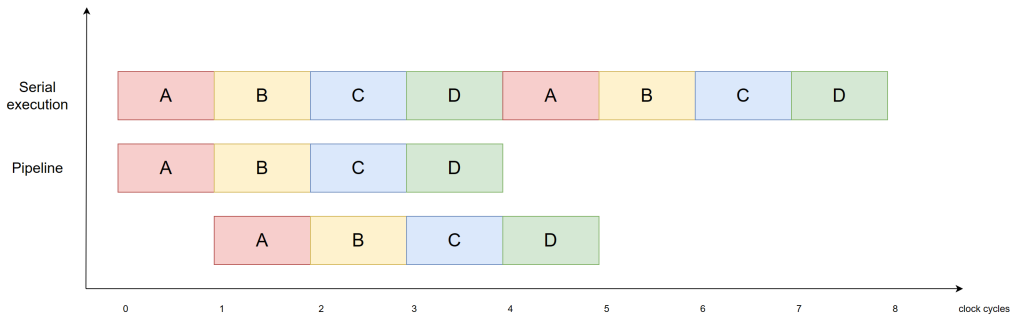


Figure 2.10: Multistage pipeline

The CPU design under modern computer architecture will sequentially execute the Instruction Fetch (IF) stage, Instruction Decode (ID) stage, Instruction Execute (EX) stage, Memory Access (MEM) stage, and Write Back (WB) stage in the form of a pipeline [6]. However, in this work, although the neural network hardware IP implemented by FPGA does not have a specific instruction set (such as X86, RISC-V, etc.), the multi-instruction pipeline structure is also used in specific task execution, which will reduce the hardware execution time and increase the overall throughput of the hardware IP at high frequencies (above 100MHz). The pipeline hardware structure also encounters some pipeline hazards, which will be discussed in detail in Chapter 3.

2.5.2. Von Neumann architecture

In the classic von Neumann architecture, the arithmetic unit and control unit are separated from each other in the main frame of the computer system, and instructions and data co-exist in the memory. The basic architecture of this work also adopts the von Neumann structure, which demonstrates that the hardware IP of this work separates the neural network memory (cache) and control (digital logic). After the stream data input, the IP can read the cache inside the IP to the processing element (PE) to perform MAC operations as Fig. 2.11.

Before each MAC operation, read and write operations of the memory unit are required. If faced with a high-frequency large-model computing scenario, the limited total bandwidth between the computing core and memory directly limits the data transmission speed, leading to the "Memory Wall". Therefore, an integrated storage and computing architecture(Memory Computing Architecture) that breaks the original traditional von Neumann architecture is currently proposed for high-performance hardware computing architecture of large models. It integrates the computing unit PE with storage and directly completes operations in the storage unit to avoid memory access delays, as shown in Fig. 2.12. For high-performance computing chips, the cutting-edge technology in recent years has focused on memristor storage and

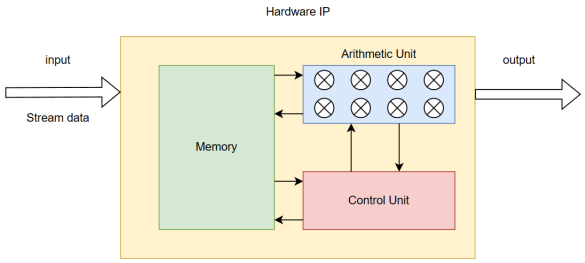


Figure 2.11: Von Neumann Architecture

computing integrated chips [21].

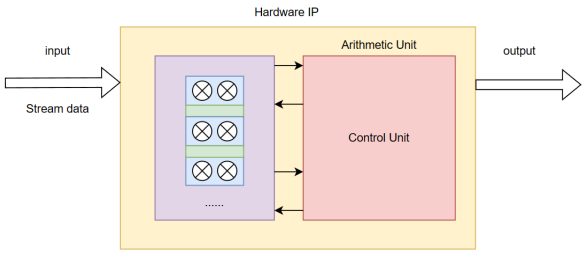


Figure 2.12: In Memory Computing Architecture

However, this project focuses on implementing on-chip lightweight neural networks. While there are demands for computing power with multi-channel parallel requirements, these needs can be adequately addressed through the use of multistage pipelines and Von Neumann architecture designs. Furthermore, the Von Neumann architecture is a well-established and mature design that seamlessly integrates hardware acceleration functions.

2.6. Neural Network Hardware Deployment

This work's VAD audio detection process must meet fast response requirements, high clock frequency, and flexible circuit design. The field programmable gate array (FPGA) can achieve high parallelism due to its programmable hardware flexibility to guarantee fast calculations to meet fast response requirements. In practical applications, since AI algorithms require multiple optimization iterations, FPGA is also often used as a prototype verification platform for ASIC in AI hardware deployment.

On the Xilinx Zynq 7000 series FPGA development board, numerous hardware resources are integrated to facilitate the implementation of various programmable gate arrays. This work mainly uses hardware resources such as LUT, FF, BRAM, and DSP. These hardware resources can be implemented as corresponding hardware units, such as FIFO, SRAM multipliers, etc.

2.6.1. Hardware Resources

LUT

LUT (Look-Up Table) was proposed to solve the reconfigurable problem of traditional combinational logic circuits during wiring and gates. In FPGAs, LUTs are commonly used to replace gate circuits to implement combinational logic synthesis.

LUT uses a truth table to determine the outputs according to the inputs. Therefore, LUT can adapt to different Boolean algebraic logics. Fig. 2.13 shows the difference between a 6-input AND gates chain synthesized logic circuit and the corresponding LUT6 (6-input LUT)

synthesized logic circuit. As Fig. 2.13 shows, the adapted synthesis of LUT is more flexible than combinational gates.

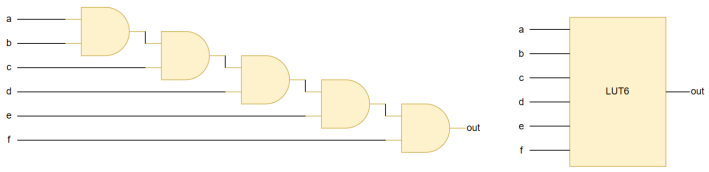


Figure 2.13: AND chain vs. LUT6 synthesis

FF

FF (Flip-Flop) is a memory unit that can store one bit of binary code. Several FFs can store multi-bitwidth data. For ASIC and FPGA design, sequential logic synthesis usually uses DFF. However, timing violations may occur in DFF if the circuit is improperly designed after synthesis or implementation. The most common cases are setup time violations and hold time violations.

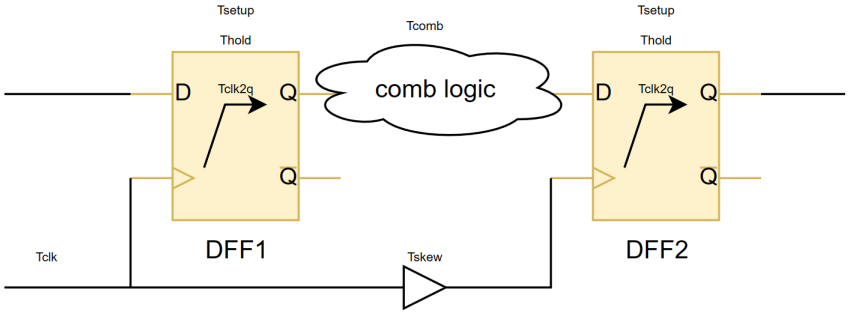


Figure 2.14: Cascaded DFFs STA (Static Timing Analysis)

As shown in Fig. 2.14, this is a circuit of Cascaded DFFs. Tclk2q represents the delay from clock to Q, and Tcomb represents a delay of combinational logic (including routing delay) between 2 DFFs, while Tclk and Tskew represent clock cycle and clock skew, respectively. To ensure the requirements of setup time and hold time, setup time and hold time should satisfy the relationship as Eq. 2.17 and Eq. 2.18:

$$Tclk2q + Tcomb + Tsetup <= Tclk + Tskew \tag{2.17}$$

$$Tclk2q + Tcomb >= Thold + Tskew \tag{2.18}$$

In FPGA, the fundamental piece, internal DFF devices, cannot be replaced, and the clock uses the global clock (or BUFG). Therefore, if setup hold timing violations are encountered, in addition to the constraints on each clock path or fan-in and fan-out of the DFF, the most common way is to optimize the combinational logic. Specific timing logic optimization based on STA will be discussed in Chapter 3.

BRAM

BRAM (Block Random Access Memory) is the RAM composed of embedded memory resources (RAMB36E1 and RAMB18E1) inside the FPGA. Xilinx 7000 series FPGA boards mainly include two types of memory resources, one of which is distributed RAM composed of LUT units, and

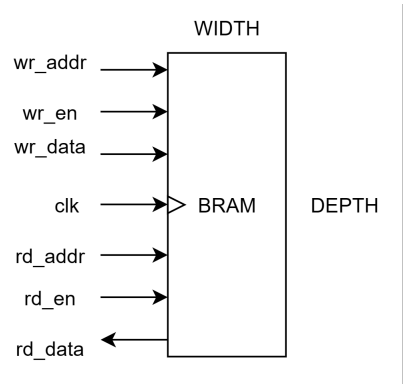


Figure 2.15: Simple BRAM Configuration

the other is BRAM. BRAM can realize the cache, FIFO, pipeline, and other functions of FPGA hardware. BRAMS are fundamental memory resources in FPGA.

Fig. 2.15 shows a simple BRAM configuration diagram. This BRAM contains a read data channel 'rd_data', a write data channel 'wr_data', and a corresponding read address channel 'rd_addr' and a write address channel 'wr_addr'. This BRAM implements a function similar to a single-port SRAM. It is driven by the input clock 'clk'. When wr_en is high, data is written to the BRAM, and when rd_en is high, then data is read from the BRAM. The bit-width of data is 'WIDTH', and the depth of BRAM is 'DEPTH', so this BRAM storage capacity can be calculated as Eq. 2.19. This case shows a simple BRAM implementation. Multiple BRAMs can be combined to realize RAM of larger capacity and more complex functions, which will be discussed in the next section.

$$Capacity = WIDTH * DEPTH/8 \tag{2.19}$$

DSP

Although many LUT resources are integrated into the FPGA that can be used for MAC calculations, especially high-width fixed-point (or floating-point) operations, those will consume LUT resources that are not worth the gain. Therefore, FPGA handles large-scale data operations through DSP. The 7000 series FPGA board used in this work integrates 12 DSP hard cores dedicated to digital MAC operations, the DSP48E1 slice.

Fig. 2.16 shows a simple DSP48E1 structure. The DSP's main computing resources include multipliers and adders for fixed-point (or floating-point operations) and logic gates for logic operations. For example, this DSP48E1 can perform the simplest operations, 'E = (B+C) * A' and 'E = E $\vee$ D'. In actual scenarios, DSP48E1 integrates many computing units internally, which can support multiplication at most signed 18-bit with 25-bit numbers [20]. DSP48E1 can decompose complex arithmetic operations into different small pieces of operations and execute them in parallel, which means that it can be dynamically configured. Moreover, DSP48E1 can operate at the FPGA's highest clock frequency (100MHz in this work). Therefore, the DSP48E1 is well-suited for handling computationally intensive operations on FPGA.

2.6.2. Hardware implementation unit

The basic resources inside the FPGA can implement various hardware units, such as SRAM series, FIFO, multipliers, etc.

FIFO

FIFO (First-in-Frist-out) is often used as a cache unit in FPGA. In high-speed communication, FIFO acts as a reservoir to cache data. FIFO also often uses high-speed protocols, such as the

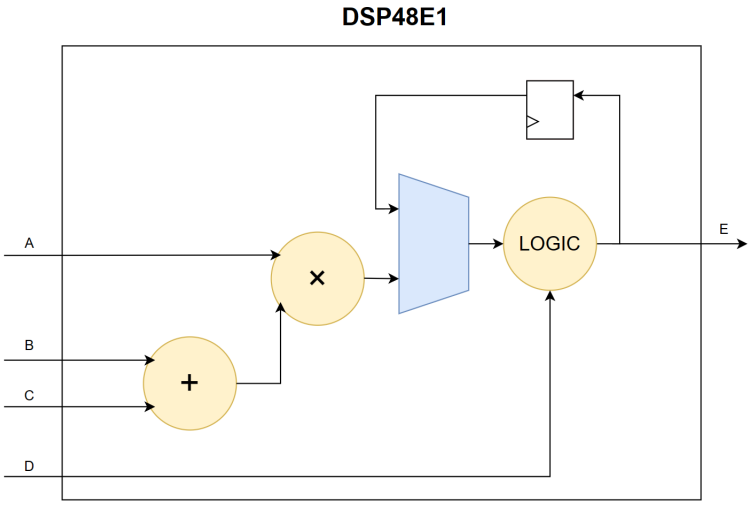


Figure 2.16: Simple DSP48E1

AXI-4 protocol, when transmitting data with other units. FIFO is implemented by synthesized BRAM in FPGA.

Fig. 2.17 shows a FIFO structure. Obviously, FIFO has empty and full signal judgments, but it does not have ‘addr’ addressing compared to BRAM. Because the data caching rule of FIFO is first-in-first-out, the address line comprises a self-increasing pointer inside the FIFO. FIFO can be divided into synchronous FIFO and asynchronous FIFO depending on the input read and write clock. Since FIFO is implemented by SDPRAM (Simple Dual Port RAM), the read and write clock domains of FIFO are separated. If the sources of ’clk_wr’ and ’clk_rd’ are the same, it is a synchronous FIFO. However, when it comes to clock domain crossing (CDC) transmission, if the read and write clock sources are different, it is an asynchronous FIFO. Although the read and write clock domains are independent, in CDC transmission, the pointer transmission through read and write domains needs to use Gray code to avoid metastability.

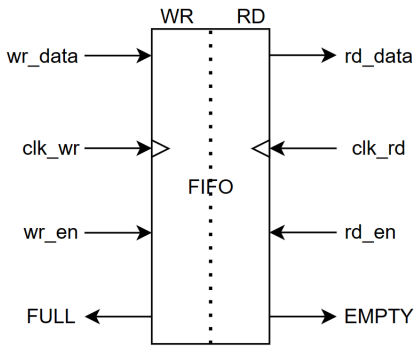


Figure 2.17: FIFO Configuration

FIFO is often used for high-speed communication cache because it has a simple structure, no addressing line, and bubbles in transmission. However, in this work, due to the acceleration of neural network operations, data at specific locations will be addressed during the operation, so the cache in this work uses SRAM instead of FIFO, which will be introduced in detail in Chapter 3.

SRAM

SRAM refers to static random access memory. When SRAM is powered on, the stored data will not be lost and does not need to be refreshed. Because SRAM is fast and consumes less power than other RAMs, it is often used for cache or on-chip memory. However, due to the high cost, the BRAM-synthesised SRAM resources in the FPGA are also limited, which need to be reused in digital logic design. There are three types of commonly used SRAM in FPGA: SPRAM (Single Port RAM), SDPRAM (Simple Dual Port RAM), and TDPRAM (True Dual Port RAM). Fig. 2.18 shows the structure of three SRAMs.

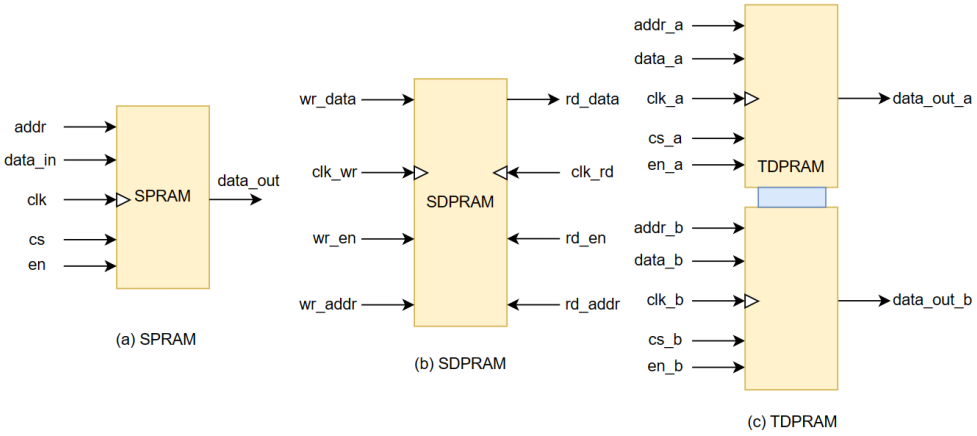


Figure 2.18: SRAM structure. (a)SPRAM (b) SDPRAM (c)TDPRAM

- SPRAM

  SPRAM is the simplest form of SRAM. Its structure is similar to BRAM. SPRAM has only one channel and one clock for reading and writing, meaning it cannot read while writing or vice versa. Due to its simple structure and function, its applications are limited.
- SDPRAM

  SDPRAM is a standard SRAM application that is widely used in FPGA. It has two channels and two clocks for reading and writing, respectively. However, although it can read and write in two channels simultaneously because the read and write channels of SDPRAM are fixed, neither channel can read or write simultaneously.
- TDPRAM

  TDPRAM is a high-end application of SRAM. TDPRAM can be regarded as a combination of the RAM of two SPRAMs. It has two independent sets of channels and clocks, which can read and write data simultaneously without interfering with each other. This means that TDPRAM can read or write from two channels simultaneously, or it can read and write respectively simultaneously. TDPRAM is flexible enough to get reconfigured.

In this work's actual approach, many combinations of various SRAMs will be flexibly configured to implement caching. However, both SDPRAM and TDPRAM will encounter a risk: RAM read and write conflicts (when reading and writing the same address simultaneously). The solution will be discussed in detail in Chapter 3.

Multiplier

The multiplier mentioned here specifically refers to the DSP synthesized multiplier that can perform MAC operations. As mentioned earlier, one single DSP48E1 supports signed multiplication operations of up to 25 bits variable and 18 bits variable. However, in neural network

hardware implementation, neural network accelerators often use 8-bit quantization to perform convolution operations. This will definitely waste a lot of bit width when only operating the multiplication of 8-bit and 8-bit. To improve DSP utilization rate and data transmission throughput, multiple multipliers can be combined and concatenated together to form a multi-channel DSP multiplier.

Fig. 2.19 shows two unsigned multiplications of A, B, and C that have been 8-bit fixed-point quantized using a dual-channel DSP multiplier and a single-channel DSP multiplier. It is assumed that there is no overflow in multiplication under 8-bit fixed-point quantization, and the DSP supports up to 25-bit and 18-bit unsigned multiplication. It can be obviously seen that after A and B are concatenated together, within the same clock cycle, (a) can complete two multiplications of A×C and B×C, but (b) can only complete A×C. A dual-channel DSP multiplier can bring 50% multiplication calculation efficiency compared with a single-channel DSP multiplier.

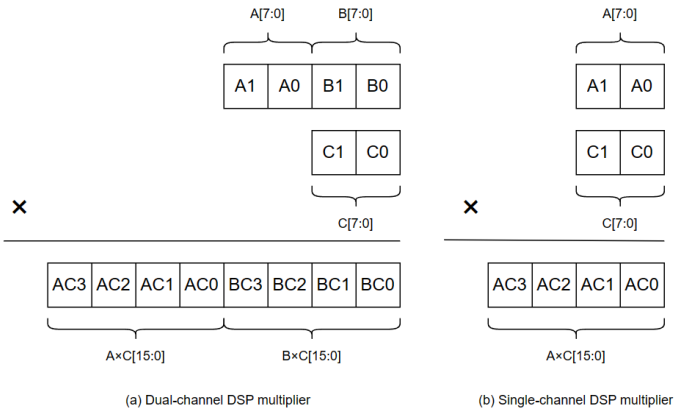


Figure 2.19: Multiplier example. (a)Dual-channel DSP multiplier (b) Single-channel DSP multiplier

Regarding multiplier hardware implementation, if a single DSP is supposed to implement dual channels, the two multiplicands can be concatenated together directly. It is worth mentioning that without adding the $'set\_property'$ constraint instruction, it will be defaultly synthesized into 2 DSPs. This depends on the resource requirements of the project.

2.7. Related Work

The following is a summary of some recent cutting-edge VAD works. It is worth mentioning that some VAD works focus on optimizing energy consumption and resource utilization, and some VAD works focus on improving accuracy, while some VAD works focus on optimizing the neural network training process.

Time-Domain CNN (TD-CNN) VAD [4] [33] [16]:

Inspired by recent advances in the raw-data convolutional neural network (CNN), Feifei Chen and Ka-FaiUn proposed a switched-capacitor time-domain CNN (TD-CNN) analog feature extractor (AFE). The always-on full-bandwidth analog-to-digital converter (ADC) and digital feature extractor can consume a significant power of >20 μW for conventional VAD with an analog filter bank. The TD-CNN with sparsity-aware computation (SAC) and sparsified quantization (SQ) is the first layer for extraction, and then the extracted features are then 1 bit quantized by a comparator to evaluate the binarized neural network (BNN). Unlike the conventional AFE, all the sampling units execute in parallel, and the sampling capacitors' execution is determined by sampling cycles and kernels. A signal sampling of each kernel

occurs in a different time domain; this reasonable allocation will significantly reduce the hardware resources and power consumption. This article used the TIMIT dataset as their test dataset in the final experiment. Benchmarking with the prior art, their VAD in 28-nm CMOS scores a 90% (94%) speech (non-speech) hit rate on the TIMIT dataset with small power (108 nW) and area (0.8 mm2), and the KWS accuracy is 94.3%.

Masked auditory encoder based CNN (M-AECNN) VAD [22]:

Encoders of the network will become not robust if they are designed for using energy levels or Mel-frequency cepstral coefficients (MFCCs). Motivated by the human acoustic system, Masashi Unoki and Nan Li proposed a masked auditory encoder-based CNN (M-AECNN) to improve the robustness of VAD. It used an auditory encoder (AE) for signal feature extraction and masked auditory encoder-based CNN (M-AECNN) for classification. AE used a gamma chirp auditory filter bank (GAFB) as the feature extraction step's feature extractor. This filter bank can reproduce the psychophysiological estimation of human auditory filters over a wide range of central frequencies, which simulates the human-ear effect of MFCC but does not consume much power like MFCC. In the classification step, to simulate the masking effect of human ears on different frequencies, they propose an estimation of masking weights block to learn a group of masking values for AE. The masking weights block is executed iteratively until MSE loss tends to 0; the classification accuracy can increase significantly. In evaluating the final results, the proposed approach average achieves an absolute improvement in AUC of about 10.5% over the CNN-MFCC-based approach; this improvement is more obvious in the -5 dB and -10 dB low-SNR noisy environments. The results of this study demonstrate the potential significance of using human auditory properties for robust VAD.

CNN-LSTM-DNN-based VAD [44] [9]:

L. Deng and J.C.Platt proved that CNNs are good at extracting features, and LSTMs (Long Short Term Memory) are good at processing sequence data. However, the input of CLDNN (convolutional long short-term memory DNNs) must be sequence data because of the LSTM, while CNN is located in the bottom layer of CLDNN. Therefore, Tianjiao Xu and Hao Li proposed a two-stage training strategy based on the VAD data training step. In the first stage, the CNN is trained on frame-level data to get high-level feature expressions individually. The network targets the VAD label of each frame. CLDNN applies a CNN at the bottom to reduce frequency variation in the input, which is then passed to an LSTM to perform temporal modeling. In the second stage, the LSTM receives the high-level feature expression and is trained with sequence data. The two-stage learning strategy improves data utilization, which uses the frame-level features in the first stage and sequence data in the second. The final results show that the proposed method achieves over 2.89% relative improvement over the original CLDNN on noise-matched conditions and over 1.07% on unmatched conditions. It represents that the proposed method has obvious advantages in discriminative ability and generalization ability. They can obtain an accurate VAD system trained with very limited training data using the proposed method.

Reconfigurable DNN accelerator for VAD [24] [41] [25]:

IoT speech recognition systems, high energy efficiency, and extremely low power consumption are particularly important for DNN implementation. Bo Liu proposed a reconfigurable DNN accelerator architecture for voice activity detection(VAD) to reduce power consumption and achieve high energy efficiency. It contains two optimization techniques: one is a processing element that supports digital-analog hybrid approximate calculation, and the other is a dynamically configurable approximate calculation unit. The optimized DNN digital-analog hybrid architecture for the processing element accumulates and quantizes pulse signals with different delay widths at the analog TDC output. For the calculation unit, this work proposed a

multi-step quantized multiplication unit that adds a MUX to dynamically select the number of quantization bits to achieve the purpose of dynamic configuration. From the results, compared with Thinker, this work improves the energy efficiency with proposed approximate computing units and achieves over 6.5X better energy efficiency. Compared with EERA-ASR, this work can achieve up to 10X better in energy efficiency. The results show that, implemented and simulated with TSMC 28 nm HPC + process technology, the estimated power of this DNN accelerator is 6 $\sim$ 12 μW, and the energy efficiency achieves 33.33 $\sim$ 66.67 TOPS/W, which is over 6.5X better than state-of-the-art architectures.

3
Proposed Methodology

This chapter will illustrate the software-hardware co-design of this work from the software implementation and hardware implementation of VAD. In software design, this work innovatively proposes a new type of neural network, DeltaFC, whose architecture is based on FC, and introduces the optimization mechanism of the network on temporal sparsity and quantization. In hardware design, this work proposes an innovative FPGA architecture to deploy the Delta algorithm and CSR algorithm and implements the calculation acceleration of neural networks directly in hardware design.

3.1. Software implementation

The software design of this work is aimed at the algorithm of this new neural network, DeltaFC DNN. The operation for one frame in the DeltaFC layer is shown in Fig. 3.1.

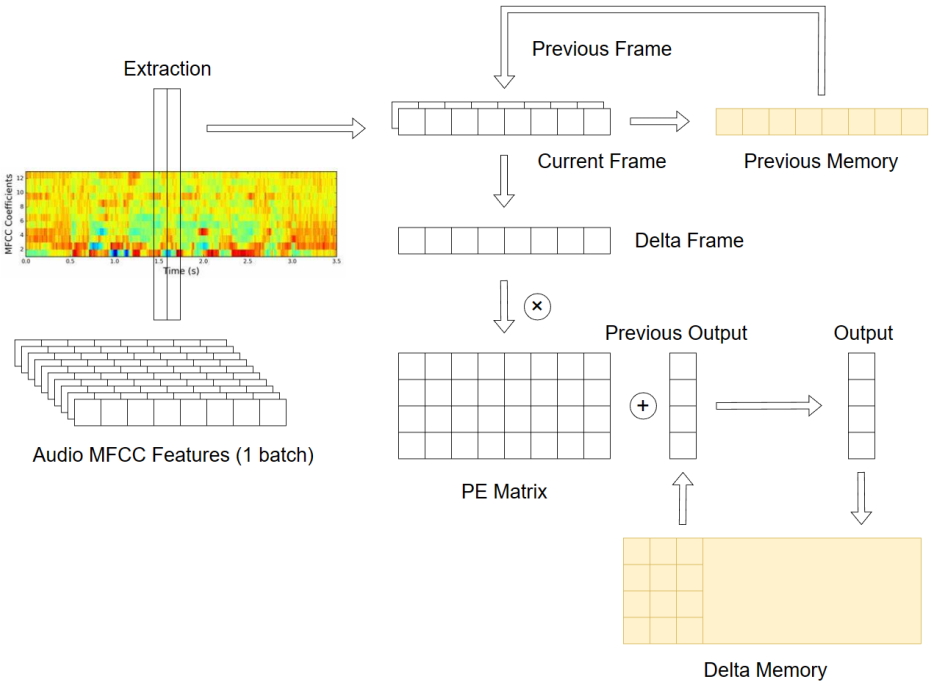


Figure 3.1: DeltaFC calculation of one frame

The multi-layer DeltaFCs and FCs are combined to create the DeltaFC DNN, as shown in Fig. 3.2.

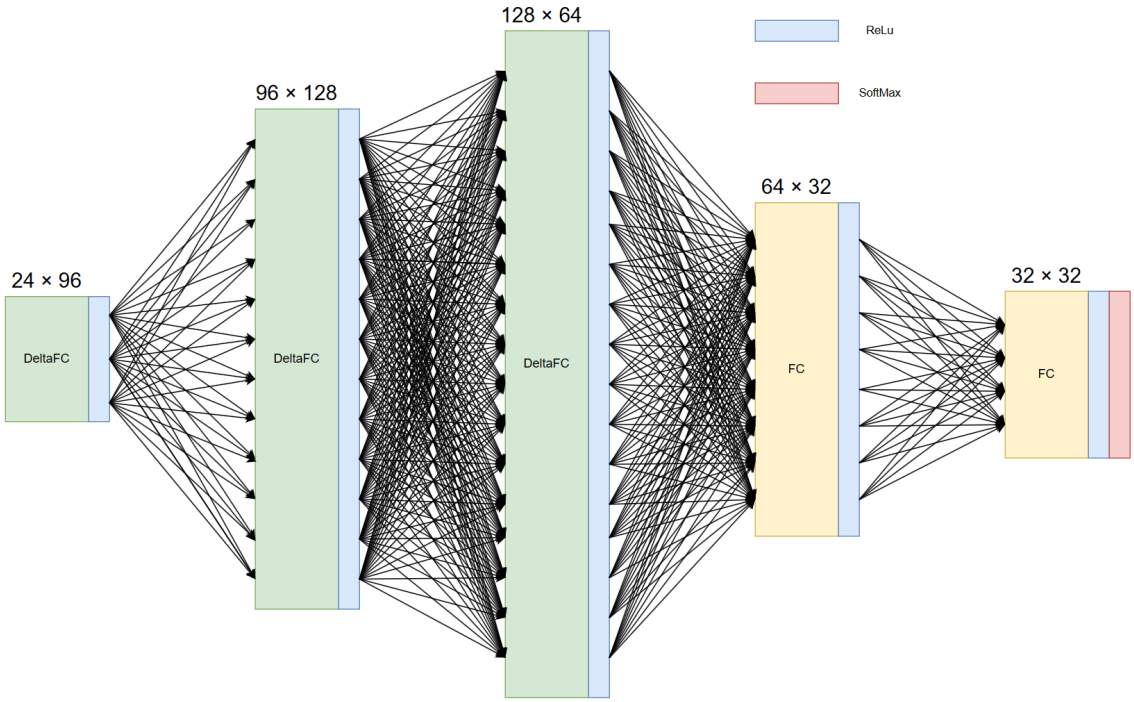


Figure 3.2: DeltaFC DNN structure

3.1.1. Delta Algorithm

The delta algorithm is the core of this work. The Delta algorithm has a wide range of applications, such as real-time data analysis in quantitative finance [18], data preprocessing in digital communications, and deep learning algorithm compression involved in this article. Its purpose is to compute the increment of feature data for each frame rather than the feature data of each frame itself. The underlying principle of the Delta algorithm is similar to differential calculus.

Assume that the $f(x)$ is a basic neural network operation. And assume that the input feature at time t is x_t, then the input feature of the previous frame is x_{t-1}. Then, the corresponding $f(x)$ in differential expansion with the feature itself in the regular space and differential expansion with the time axis as shown in Eq. 3.1.

$$\begin{aligned} f(x) = wx + b => f'(x) = w\Delta x \\ => f'(x) = w(x_t - x_{t-1}) \end{aligned} \tag{3.1}$$

Δx in differential calculus is infinitesimal, corresponding to the time frame $(x_t - x_{t-1})$ being as small as possible. After Delta algorithm processing, $f(x)$ essentially differentiate to $f'(x)$, so to integrate it to $f(x)$ itself, bias $'b'$ need to be equal to its value in the previous frame, as shown in Eq. 3.2.

$$\begin{aligned} \int f'(x)dx = \int w(x_t - x_{t-1})dx = f(x) \\ f(x_t) = w(x_t - x_{t-1}) + f(x_{t-1}) \end{aligned} \tag{3.2}$$

Due to the small feature variation along the time axis in the MFCC-processed audio stream data used in this work, the Delta algorithm is well-suited for compressing feature information,

significantly reducing data redundancy. Furthermore, the combination of the Delta algorithm and the threshold filtering introduced later results in considerable sparsity in the original data. These sparsity characteristics greatly accelerate the operation speed of the neural network.

3.1.2. DeltaFC DNN layer

DeltaFC is a neural network based on FC and introduced RNN recurrent memory layer, as shown in Fig. 3.3.

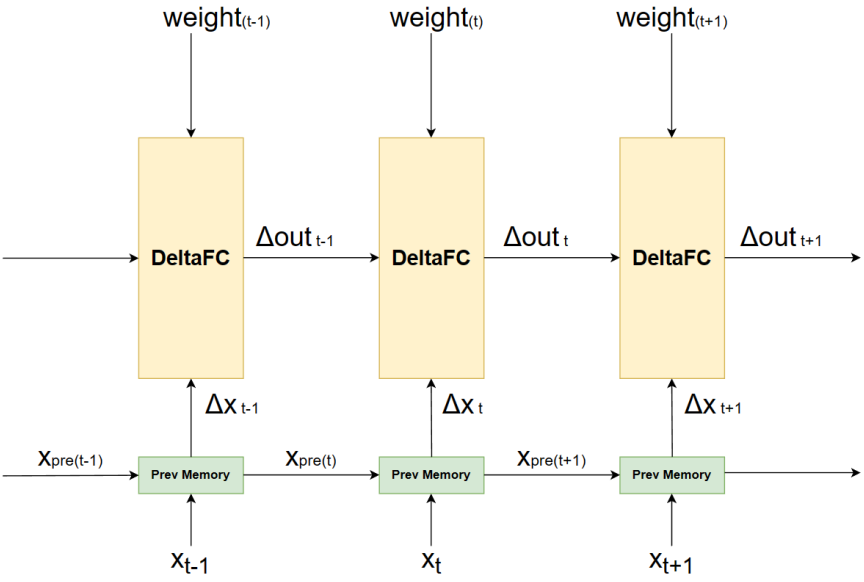


Figure 3.3: DeltaFC Neural network structure

Delta Features

The time-series information is input frame by frame with a fixed time step. Firstly, assume the current state of the input audio frame is the 'input current frame', and the adjacent previous input frame is regarded as the 'input previous frame'. In DeltaFC, the input feature is the absolute value difference of the features of the two adjacent frames of current x_t and previous x_{t-1} (read from prev memory, so actually is $x_{pre(t)}$, unless otherwise specified, it is expressed as x_{t-1}), rather than the input feature itself, as shown in Eq. 3.3.

$$\begin{aligned} \Delta x_t &= x_t - x_{pre(t)} \\ &= x_t - x_{t-1} \end{aligned} \tag{3.3}$$

Here is an example: assuming that there are two audio frames and each contains 8 features, the features after MFCC of current and previous frames are shown in Fig. 3.4. Because time series information exhibits temporal correlation, the variation between frames will not be obvious.

If the two frames are directly input into the neural network for parallel matrix operations, the neural network needs to use an 8-way multiplier (in hardware implementation, it is mostly a DSP), which is a huge computational load on neural network operations. However, the current and previous frames in this example are features after the audio MFCC and have a timing

86.3124	79.6325	99.1254	86.9875	69.4521	88.6734	77.1210	82.4568

(a) Current frame

85.0123	78.6320	98.0254	86.9475	66.4521	88.1734	77.1210	81.4568

(b) Previous frame

Figure 3.4: Current(a) and previous(b) frames

1.3001	1.0005	1.1000	0.0400	3.0000	0.5000	0.0000	1.0000

Figure 3.5: Delta absolute frame

correlation. If the feature is the difference between the current and previous features, then the Delta absolute frame can be calculated as shown in Fig. 3.5.

The Delta difference is hundreds of times less than the original calculated value. However, if the calculated difference without quantization is less than 1 but contains too many decimal bits, this also imposes a computational load on the neural network. These decimals less than 1 have little impact on the final result, so these decimals can also be regarded as sparse values. Therefore, the threshold is applied to limit the range of delta input features.

Introducing a threshold in the Delta input feature can help improve temporal sparsity. The threshold specifies the minimum range of feature propagation, similar to a filter. If the Delta input feature is higher than the threshold, it can continue propagation to the next frame, or it will be regarded as 0, as shown in Eq. 3.4. In addition to threshold, correspondingly, prev memory will compare the feature x_t of the current frame with the x_{t-1} of the previous frame. If the input Δx_t is higher than the threshold, the prev memory will retain the x_t of the current frame, and if it is less than the threshold, it will retain the x_{t-1} of the previous frame, as shown in Eq. 3.5.

$$\Delta(x) = \begin{cases} \Delta(x) & , |\Delta(x)| \geq Threshold \\ 0 & , |\Delta(x)| < Threshold \end{cases} \tag{3.4}$$

$$x_{pre(t+1)} = \begin{cases} x_t & , |\Delta(x)| \geq Threshold \\ x_{t-1} & , |\Delta(x)| < Threshold \end{cases} \tag{3.5}$$

For example, when the Delta input feature obtained the value shown in Fig. 3.5, assuming the threshold is set to 2.0000, the Delta input feature filtered as shown in Fig. 3.6.

0.0000	0.0000	0.0000	0.0000	3.0000	0.0000	0.0000	0.0000

Figure 3.6: Delta absolute frame after threshold filter

Only one feature, '3.0000', is left in the Delta input frame after threshold filtering, and the rest are filtered out. Through delta transformation and threshold, the calculation of the eight features of each frame is simplified to the calculation of only one feature. The threshold is dozens of times smaller than the original feature, so the filtered result has little effect on the final recognition result, especially for this work whose output is a binary value of 0(noise) or 1(speech). During the operation of the neural network, due to the 87.5% temporal sparsity, the operation can save 7 operation cycles. In this process, there are many such 0 values in the neural network operations, and the neural network directly bypasses the 0 values, directly accelerating the operation of the neural network.

Processing Element(PE)
The structure of DeltaFC is based on FC, which means that the FC module undertakes the work of the processing element (PE). It can be seen from chapter 2 that the calculation of FC output is as Eq. 3.6.

$$Z_n = \sum_m (W_{n,m} x_m) + bias_n \tag{3.6}$$

where Z_n is the output of FC, and W is weights while $bias$ is the bias. If it is specific to the FC of one frame, the operation is shown in Eq. 3.7.

$$C_m = W_m x_m + bias \tag{3.7}$$

After the previous delta features transformation, since the input is the difference between the two frames current and previous, x is replaced by Δx, and bias is replaced by the output of the previous frame C_{m-1}, then the above formula can be transformed into Eq. 3.8.

$$\begin{aligned} C_m &= W_m \Delta x_m + C_{m-1} \\ &= W_m (x_m - x_{m-1}) + C_{m-1} \end{aligned} \tag{3.8}$$

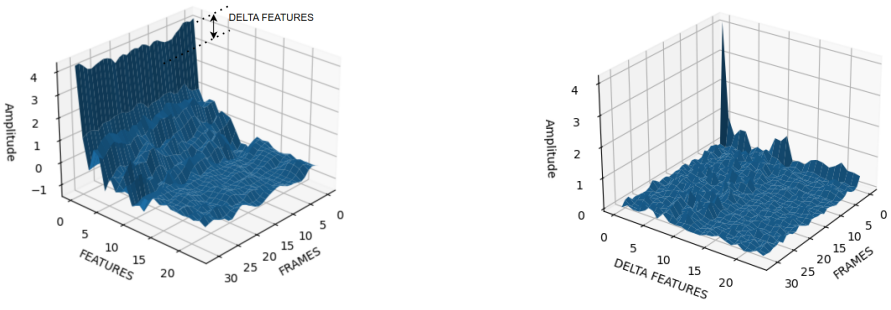
Where C_m is the output of frame m, and Δx_m will perform the threshold filtering process. Such a loop structure is similar to the memory unit of RNN. Now, as the input features are considered timing correlated, the expectation is that Δx and Z_n are also temporally sparse. In essence, the temporal sparsity between consecutive features is cast on the temporal sparsity of the delta features that are propagated.

This is reflected in the large amount of 0 values in each frame in the matrix operation of PE, and the input of the previous layer will be calculated as the output of the next layer, which implements the temporal sparsity propagation. However, after the features of a frame pass through PE, they need to be cached as the input of the next frame instead of being dropped. This is embodied as a caching strategy on the neural network of the hardware, usually implemented by FIFO or BRAM.

3.1.3. Temporal Sparsity

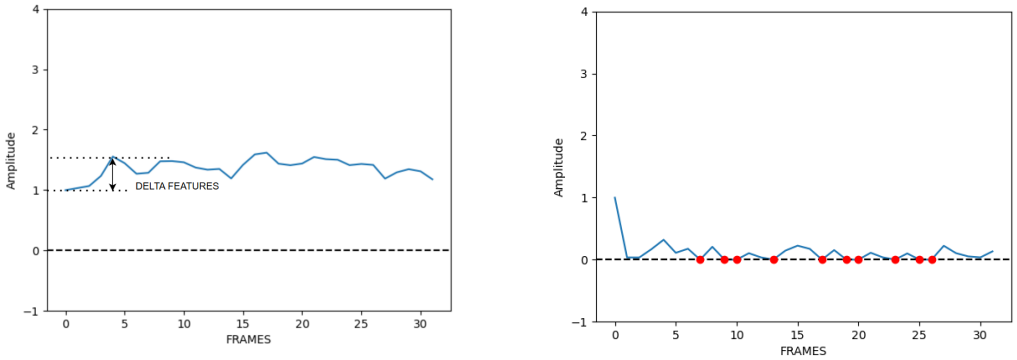
As discussed in Chapter 2, streaming signals like voice and video exhibit robust temporal correlation—meaning that their features experience small changes over time. These characteristics imply that the stream signals can induce significant temporal sparsity after the Delta algorithm transformation.

Temporal sparsity can be achieved through the iterative Delta algorithm. Fig. 3.7 shows the input audio signal features after MFCC in the first frame of the first batch. This frame of audio contains 24 Features and 32 Frames (more subdivided features and frames). It contains information on a brief 30-ms slice of an audio segment.



(a) Input features before delta algorithm **(b)** Input features after delta algorithm

Figure 3.7: Input features use the Delta algorithm to increase temporal sparsity



(a) Input features before delta algorithm **(b)** Input features after delta algorithm

Figure 3.8: Input features use the Delta algorithm to increase temporal sparsity

The 'FRAMES' on the x-axis can be regarded as the time axis, and the y-axis represents the feature axis of the input neural network, while the z-axis represents the amplitude of this audio slice after MFCC. It's obvious that along the time axis 'FRAMES', the change of the MFCC value of the feature axis is not significant. This exhibits the temporal correlation of the audio signal. Furthermore, if observing the feature changes of audio slices after MFCC on a fixed feature axis (such as 2nd FEATURE), then Fig. 3.8 can be observed.

Fig. 3.8(a) input feature that has not been transformed by iterative Delta algorithm. If ordinary hardware neural networks are used to process this stream data, even if it has been fixed-point quantized, it will cause useless hardware resources due to bit width limitations. When applying the Delta algorithm introduced in this study and utilizing a threshold of 0.03 (not optimized), the streaming data transforms, resulting in the representation shown in Fig. 3.8(b). It can be seen that the input features of Fig. 3.8(a) are dense and do not have temporal sparsity, but the input features of Fig. 3.8(b) have temporal sparsity increased to 31.25% after the Delta algorithm iteration. As mentioned previously, neural network hardware operations will skip these 0 values, so this is reflected in a 45.5% improvement in hardware operation efficiency. How temporal sparsity accelerates neural network hardware operations will be discussed in detail in the next section.

3.1.4. Quantization

As mentioned in the Chapter 2, this work applies 8-bit and 16-bit fixed-point quantization. Firstly, it's necessary to compare floating-point and fixed-point quantization. This section mainly focuses on single-precision floating-point quantization and 8-bit fixed-point floating-point quantization.

In the software design, this work uses floating-point numbers after normalization. However, during the hardware design, the input dataset utilizes the dataset after fixed-point quantization, which is achieved through a Python conversion script in the software.

Floating-point Quantization

According to the IEEE-754 standard [17], a 32-bit single-precision floating-point number can be represented as shown in Fig. 3.9.

Figure 3.9: 32-bit Single-Precision Floating-Point Number

Where S represents the sign bit to indicate the positive or negative of the number, while E represents the exponent indicating that the absolute value range of the number is within the 2^{E+1}, and F represents the decimal mantissa after normalization. If a number -3.75 is given, the process of expressing -3.75 as a 32-bit single-precision floating-point number is as in Eq. 3.9.

$$\begin{aligned} -3.75 &= -11.11 = -1.111 * 2^1 \\ &= (-1)^1 * (1 + 0.11100000000000000000000) * 2^{128-127} \end{aligned} \tag{3.9}$$

According to the IEEE-754 standard [17], a 32-bit single-precision floating-point number can be represented as Eq. 3.10.

$$X = (-1)^S * (1 + F) * 2^{E-127} \tag{3.10}$$

Then -3.75 can be represented as 1 1000 0000 1110 0000 0000 0000 0000 0000 in 32-bit single-precision floating-point number form and 1 80E0 0000(C 0700 0000) in hexadecimal form. Regarding the floating-point form, the number -3.75 contains a lot of sparsity.

However, the sparsity will be greatly reduced if the number is not divisible, such as -3.967 (C 07DE 3540). Due to the uncertainty associated with the decimal point in floating-point numbers and the fixed bit-width of hardware memory, it is imperative to standardize the input bit width for both the operation and storage units. Insufficient bit count can result in a wasteful utilization of the bit width in hardware implementation, as shown in Fig. 3.10. Although FPGA contains many floating-point units(FPUs), these applications increase the power consumption of the FPGA and do not meet the lightweight objective of this work. Indeed, floating-point numbers are notably intricate and, in general, may not be well-suited for hardware implementation [19].

Fixed-point Quantization

As the name suggests, Fixed-point numbers are numbers with a fixed decimal point. But in most cases, the decimal point of the fixed-point number is fixed to the last digit, which

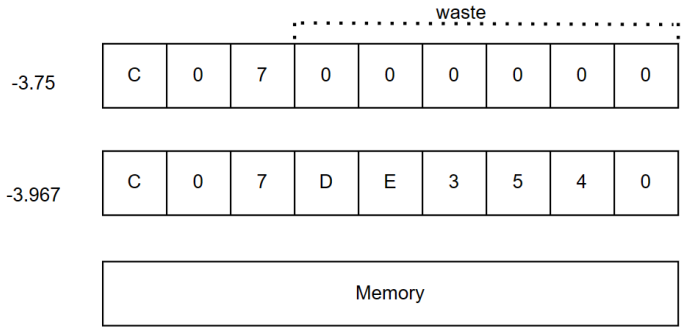


Figure 3.10: Low bits number with high bits memory will cause the waste of memory

means that the fixed-point number usually represents a pure integer. If the original decimal is converted into this form, the original number needs to be enlarged to the Nth power of 2, shifted to the left by N bits. Here is an example of how 4.5 transformed to a 16-bit fixed-point number, as shown in Eq. 3.11.

$$\begin{aligned} 4.527 &=> 4 + 0.527 \\ 4 &=> 8'h04 \\ 0.527 &=> 0.527 * 2^8 = 134.912 \simeq 135 = 8'h87 \\ 4.527 &=> \{8'h04, 8'h87\} = 16'h0487 \end{aligned} \tag{3.11}$$

However, when the hexadecimal value 16'h0487 is converted to decimal, it yields 4.52734375 instead of the exact value of 4.5. This implies that the conversion to a fixed-point number may result in a loss of precision, but its impact on the binary output of this study is minimal. Consequently, all features are normalized to a consistent number of bits, effectively mitigating resource wastage in hardware implementation.

Quantization can save resources, and, more importantly, implementing quantization in FPGA can accelerate neural network operations. This work applies the form of eight integers and eight decimals for the FPGA quantization. The first bit of the integer part is the sign bit, as shown in Fig. 3.11, which means that the fixed-point number range of the work is between -32768 and 32767. The original audio feature signal will be normalized first at the software design to control all audio feature ranges within this range, ensuring that most calculation cases will not overflow.

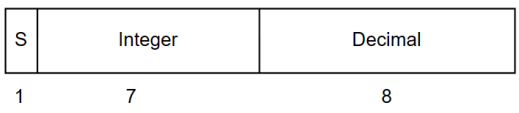


Figure 3.11: 16-bit Fixed-Point Number

Here is a simple example. This operates in the above form to calculate the product of two decimals, such as 2.853*4.212. If expressed as fixed-point arithmetic, the two numbers can be expressed as Eq. 3.12:

$$\begin{aligned} 2.853 &=> 16'h02DA \\ 4.212 &=> 16'h0436 \\ 2.853 * 4.212 &=> (16'h02DA * 16'h0436) >> 8 = (32'h000C01FC) >> 8 \\ &= 16'h0C01 <=> 12.00390625 \end{aligned} \tag{3.12}$$

It is evident that, given the fixed bit representation, each transformed number is allocated and processed with a consistent 4-bit storage and calculation. Moreover, 2.853*4.212 uses 4 shift multiplication operations, while 16'h02DA * 16'h0436 only uses 3 shift multiplication operations and 1 shift operation. The hardware consumption of the shift operation is less, and the reduced two-shift operations are reflected in a 25% reduction in calculation time in the DSP hardware circuit. For results, the true result 12.016836 deviates from the quantized result 12.00390625 by less than 0.1%, and precision is not significantly lost in 16-bit fixed-point quantization.

In summary, this study employs a 16-bit fixed-point quantization method for quantifying input features in the hardware implementation. This approach substantially accelerates the neural network's operations within the hardware framework.

3.2. Hardware Implementation

The hardware design objective of this work is to deploy the DeltaFC DNN and the acceleration algorithm on an FPGA. However, hardware algorithm deployment necessitates careful consideration of hardware architecture and timing.

3.2.1. DeltaFC Architecture on FPGA

In the initial stage of hardware deployment, a key focus is the hardware architecture. Hardware deployment includes the neural network itself and the neural network accelerator module. The core of this section is to utilize hardware resources to implement hardware functions effectively.

Module Reuse Architecture

The initial step in neural network hardware design involves establishing the top-level architecture. Intuitively, if the neural network software design shown in Fig. 3.2 is directly translated into a hardware design, the resulting representation would resemble what is shown in Fig. 3.12.

The blue parts in Fig. 3.12 are the PEs, and the green parts are the cache memory units. Such conversion seems to be very straightforward and effective and is very efficient in prototype verification of small models. However, directly deploying the algorithm on hardware may lead to insufficient FPGA hardware resources, primarily due to the limited resources of the FPGA on the Zynq board used in this lightweight study. Secondly, in this design, resource redundancy is serious. For example, as shown in Fig. 3.12, during the operation of the first neural network layer, the remaining four layers of neural networks are idle, but their power consumption continues. This does not meet the requirements for rational utilization of hardware resources. Therefore, this work proposes a module reuse architecture, as shown in Fig. 3.13.

As shown in Fig. 3.13, when the module reuse architecture is adopted, only two pieces of RAM serve as caches, which store data alternately. For example, the first layer uses $Data\,Cache\,1$ for reading and data $Data\,Cache\,2$ for writing, but the second layer uses $Data\,Cache\,2$ for reading and $Data\,Cache\,1$ for writing. In addition, the PEs of the hardware is composed of a specific module, a multiple-channel parallel DSP. The module reuse architecture can alleviate the pressure on hardware resources, significantly reduce power consumption, and improve module utilization efficiency. This architecture will serve as the basic architecture for this work.

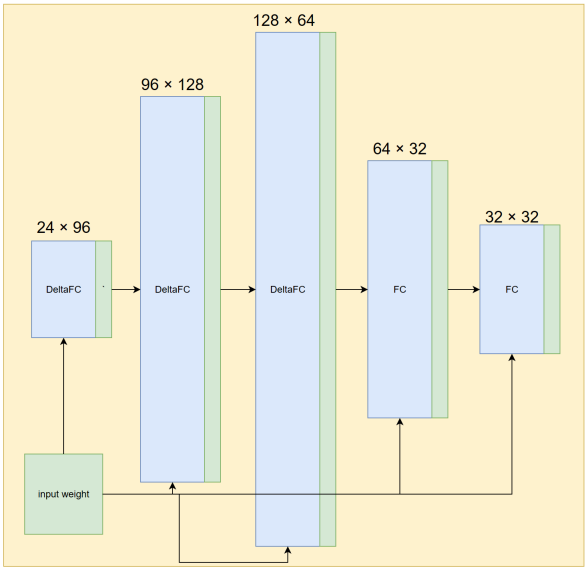


Figure 3.12: DeltaFC DNN hardware design conversion based on software design

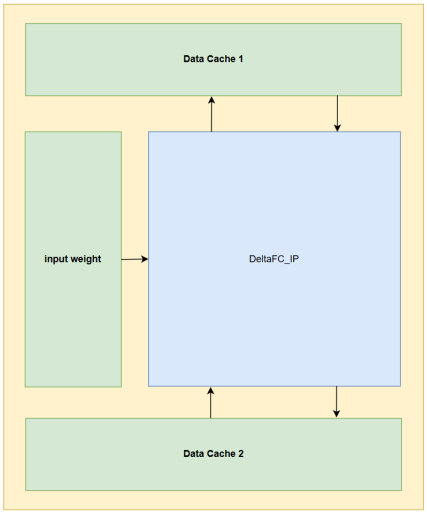


Figure 3.13: DeltaFC DNN hardware design of module reuse architecture

CSR Buffer RAM For CSR Acceleration

After the previous introduction, the simple hardware implementation of DeltaFC with module reuse architecture is shown in Fig. 3.14.

As Fig. 3.14 shows, this is a simple FPGA hardware implementation of DeltaFC neural network architecture. It caches the x_t and x_{t-1} in each FRAME through $Data\,Cache\,1\&2$ FIFO and $Prev$ FIFO, and then performs matrix MAC operations after the Delta algorithm through PE implemented by DSP. However, this simple architecture can only perform the basic Delta algorithm but cannot perform the CSR acceleration based on temporal sparsity.

The reason is obvious, as discussed in Chapter 2. This is due to the absence of an address line in the FIFO, making it impossible to address cache data at a specific location. The acceleration of the CSR algorithm necessitates the extraction of CSR indices for each row in the sparse matrix. In other words, it involves retrieving the addresses of non-zero elements in each row of the sparse matrix. Therefore, for non-zero element caches that require sparse

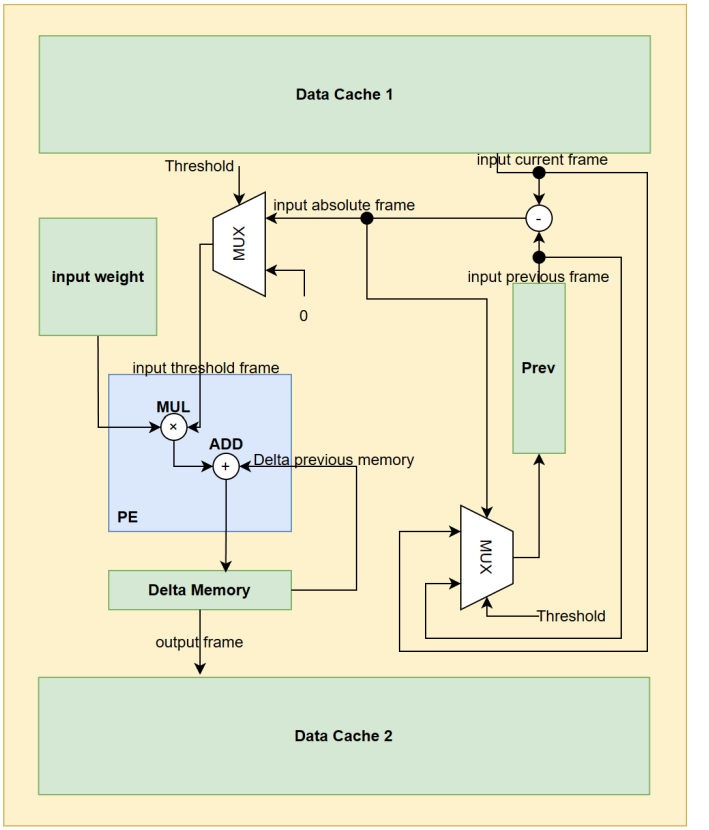


Figure 3.14: DeltaFC simple architecture on FPGA

acceleration, the hardware resources need to be configured as SRAM instead of FIFO, as shown in Fig. 3.15.

The $CSRBuffer$ RAM can both write the CSR indices of a sparse matrix and read them when required. Subsequently, these CSR indices can be used to address the $Input, weight$, and $Data, Cache, 1\&2$ SRAMs, facilitating data retrieval from specific addresses. Unlike $Data\,Cache\,1\&2$, $CSRBuffer$ RAM essentially stores a series of addresses rather than the data itself. From the perspective of modern computer architecture, $CSRBuffer$ RAM can also be regarded as a high-speed cache for addressing instructions. $CSRBuffer$ RAM is an indispensable memory unit for CSR algorithm acceleration.

3.2.2. Multiple Channel Parallel DSPs

The PEs utilized for the matrix MAC operations in the neural network are synthesized and implemented through multi-channel parallel DSPs in the FPGA hardware. However, there is a difference between the matrix MAC operation process on hardware and the execution process on software. The hardware design must account for the parallel architecture and incorporate matrix partitioning strategies to optimize parallel operations efficiently, avoiding potential timing conflicts in STA.

Matrix Multiplication Process In Hardware Design

The most important part of the DNN is the MAC operation of the matrix. However, limited by the hardware architecture, the matrix multiplication process differs between the software and hardware levels. The distinction lies in the parallelism of hardware matrix multiplication.

Fig. 3.16(a) shows the matrix multiplication process in software design. Matrix multiplication in software design is similar in structure to scalar space multiplication. The condition for

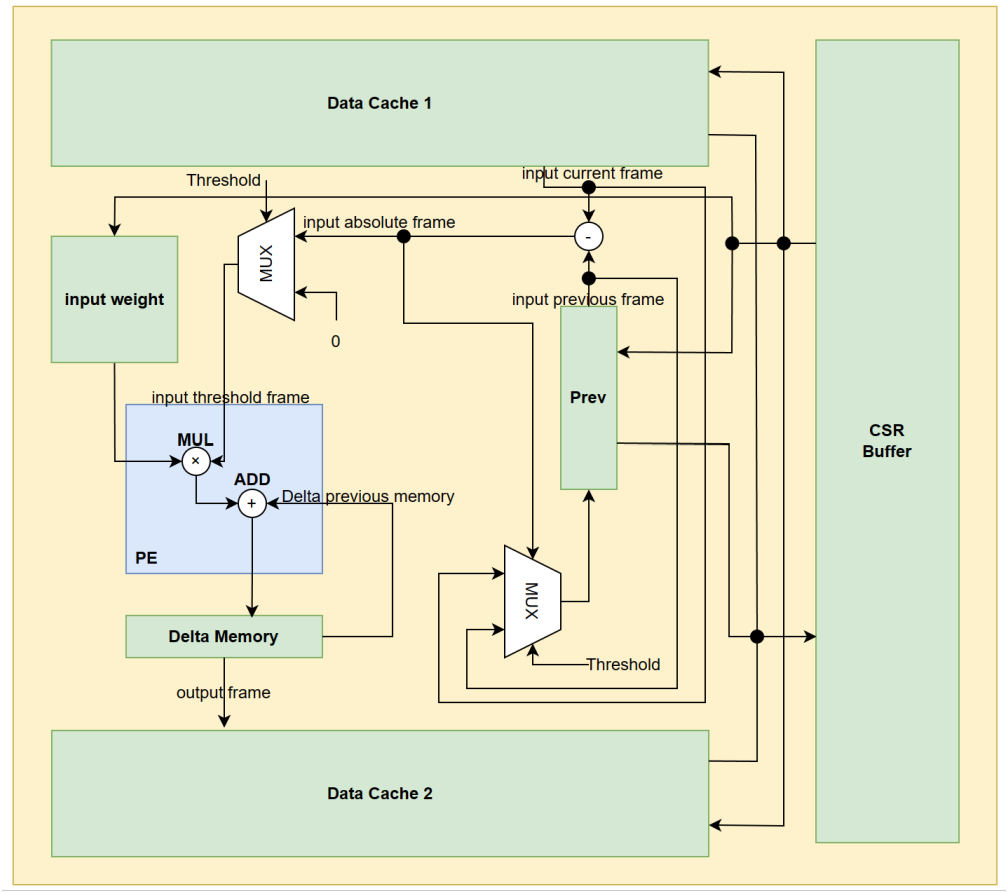


Figure 3.15: DeltaFC accelerator architecture on FPGA

performing matrix multiplication is that the number of columns of the first matrix must equal the number of rows of the second matrix.

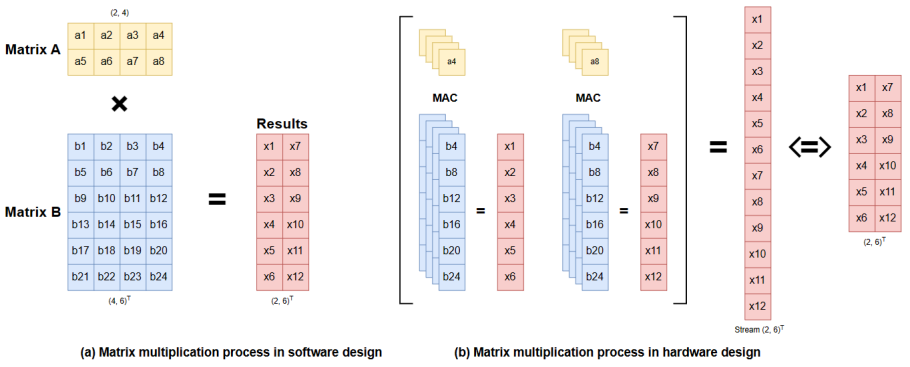


Figure 3.16: Matrix multiplication process in software and hardware design

However, as shown in Fig. 3.16(b), the matrix multiplication process in hardware design is parallel. Unlike software design, which multiplies matrix A and matrix B by each element sequentially, hardware matrix multiplication is a parallel multiplication of matrix A and a column element of matrix B (Transposed) and then performs MAC operations. Moreover, since each matrix element in the hardware is input individually with the clock cycle (like a stream), the hardware matrix can be regarded as a one-dimensional array without rows and columns.

In summary, due to the parallel architecture of the hardware, the matrix operation of the software cannot be directly transplanted to the hardware, but more hardware deployment issues, such as parallelism, need to be considered. But precisely because of the parallel architecture of the hardware, the hardware computing efficiency is strong enough to process large-scale data operations. As shown in Fig. 3.16, if the hardware design adopts software architecture, it requires 48 clock cycles to process whole multiplication. But if the hardware parallel architecture is adopted in hardware design, that is, a 6-channel multiplier for matrix multiplication, only 8 clock cycles are needed, and efficiency increases by 500%.

Multiple Parallel DSPs And Matrix Partitioning

As mentioned in Chapter 2, hardware operation efficiency can be improved by adopting multiple parallel DSPs. If multiple parallel DSPs are used, the multiplicands must be concatenated together so that the synthesis tool can synthesize them into multiple parallel DSPs or a single multi-channel DSP. The example of 6 data concatenation for 6 parallel DSPs synthesize is shown in Fig. 3.17.

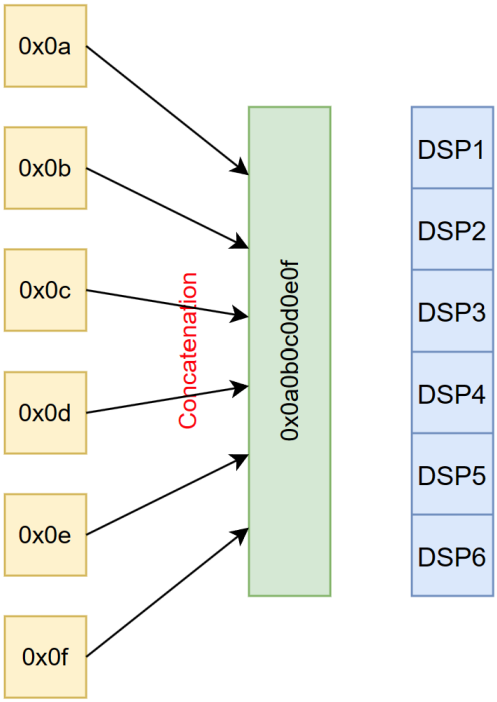


Figure 3.17: 6 data concatenation for 6 parallel DSPs synthesize

As shown in Fig. 3.16(b), this hardware design concatenated 6 matrix B elements together so that the synthesis tool can automatically synthesize 6 parallel DSPs. However, for some large models, such as when the dimensions of the matrix reach thousands, it is impossible to concatenate thousands of elements together. Not only will this cause timing violations for the STA, but the hardware resources do not have enough DSPs. Assuming that the hardware design in Fig. 3.16 only has 3 DSPs available, then the matrix multiplication needs to be partitioned into several parts, as shown in Fig. 3.18 and Fig. 3.19.

It's obvious to see that, after matrix partitioning, the elements of each column are partitioned into 2 parts. And the original result is combined with all parts concatenated together in order. However, after partitioning the original matrix into 3 channels instead of 6 channels parallel DSPs architecture, this will also cause an increase in timing, as shown in Fig. 3.20. Where '1',

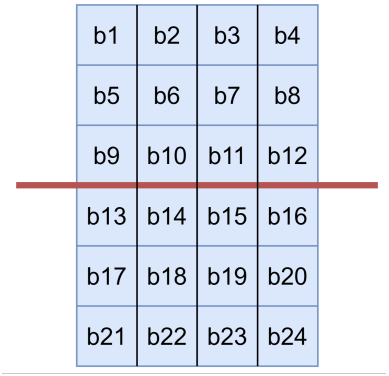


Figure 3.18: Matrix partitioning

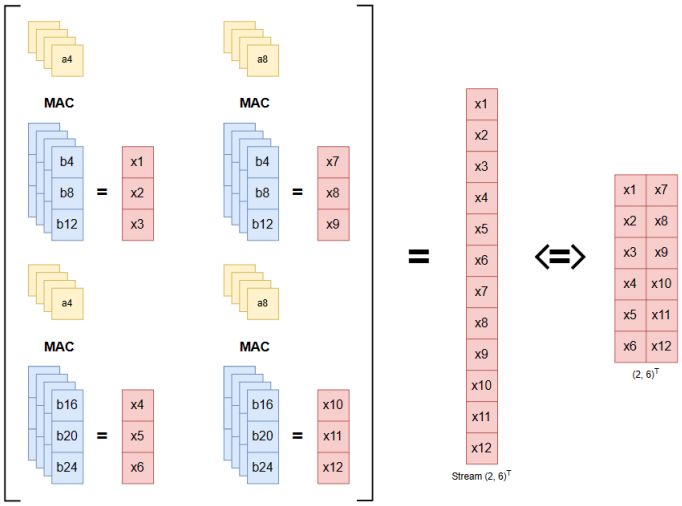


Figure 3.19: Matrix multiplication process in 3 parallel DSPs hardware architecture

'2', '3', etc. represent the 'a1', 'a2', 'a3', etc. in Matrix A shown in Fig. 3.16, and are similar in Matrix B. While '16', '13', etc. represent concatenated data from 'x1' to 'x6', 'x1' to 'x3' in results shown in Fig. 3.16.

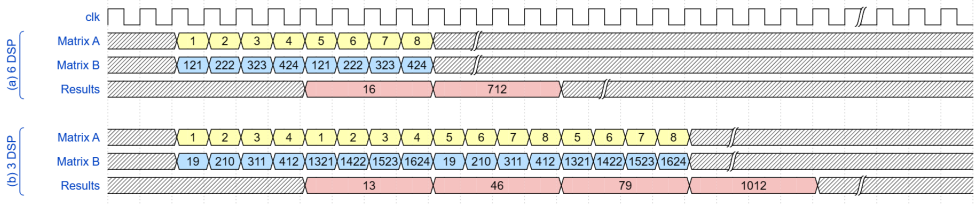


Figure 3.20: Timing diagram of 6 DSPs design and 3 DSPs design

The timing of multiple DSP operations is related to the nodes of matrix A and matrix B, matrix A's frames (number of rows), and DSP channels. Assume that the nodes of matrix A are $'NodesA'$, while the nodes of matrix B are $'NodesB'$, then the frames of the matrix are $'FRAMES'$, and the number of channels is $'CHANNELS'$, then the timing required for multiple DSPs operation matrix can be Calculated as Eq. 3.13.

$$Timing = \frac{Nodes\,A * Nodes\,B}{CHANNELS} * FRAMES \tag{3.13}$$

Although matrix partitioning will increase the timing, it can make the timing in a single clock cycle of the DSPs less stressful so that more combinational logic can be executed within one clock cycle, such as the Delta operation used in this work. In addition, matrix partitioning can also make multiple DSPs can accommodate models of different sizes, such as the models in this work. In this work, the nodes of the DeltaFC DNN are 24, 96, 128, 64, and 32, respectively. The number of channels needs to be selected so that each channel has as little redundancy as possible. This means these nodes' greatest common divisor (GCD) can be chosen as the number of channels. Obviously, the 8-channel DSP architecture is the best choice.

3.2.3. CSR Algorithm Implementation On FPGA

CSR algorithm is important in accelerating DeltaFC neural network within hardware design. Extracting the indices of non-zero elements at high frequency is crucial for FPGA hardware. However, the software algorithm deployment hardware is not executed sequentially on the Pytorch platform but in parallel in FPGA. Hardware algorithm deployment needs to take into account computer hardware architecture issues.

Read After Write (RAW) Hazard

FPGAs generally do not have an instruction set (such as X86, RISC-V) like a CPU or MCU. However, if indices read and write the CSR algorithm as instructions, then RAW hazard is the biggest obstacle to CSR algorithm deployment.

Assume a sparse matrix has 12 features and 3 frames, as shown in Fig. 3.21. Then, the cached 8-bit data transmission of the first frame (each frame has 12 features) is shown in Fig. 3.22 (a). Each 8-bit fixed-point quantized data enters hardware execution on the clock positive edge. It's a sparse matrix data. Then, the data transmission after using the CSR algorithm is shown in Fig. 3.22 (b). The data transmission first writes the CSR indices into '$CSRBuffer$ RAM', and then it can read the indices from '$CSRBuffer$ RAM' to perform the CSR algorithm. But in this way, the operation of reading the indices will always be later than writing the indices, known as the Read After Write (RAW) hazard.

FEATURES = 12

FRAMES = 3												
	1	0	0	a	0	12	0	5	a	12	0	21
	0	2	0	9	0	11	1	4	c	9	0	19
	0	0	0	b	1	10	0	5	b	c	0	1c

Figure 3.21: sparse matrix example of 12 features and 3 frames

Here is an example explaining how the RAW occurs. As shown in Fig. 3.22, 'csr_addr' can be regarded as the address operand of the indices instruction. According to the hardware pipeline structure in this figure, the CSR indices of each feature are written and read one by one with each positive edge clock. In the second feature after CSR algorithm acceleration in Fig. 3.22 (b), this IP should be supposed to enter the data '0a' under 'csr_addr' as '$0x04$'. However, only the second '$addr$' as '$0x02$' can be read from $CSRBuffer$ RAM at this moment, but at the same time, the fourth '$addr$' as '$0x04$' is supposed to be written to $CSRBuffer$ RAM for CSR acceleration. This results in reading the indices '$0x04$' will be later than writing the indices '$0x02$', which is known as RAW conflict.

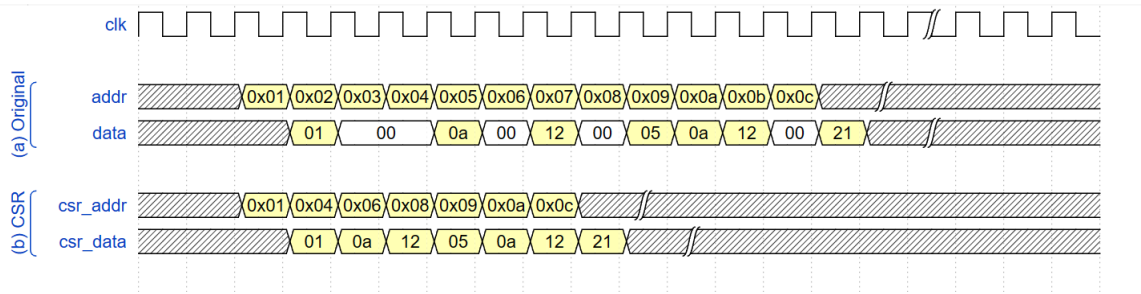


Figure 3.22: (a) sparse data transmission and (b) CSR data transmission

There are three main methods to solve data hazards: RAW, interlock, bypassing, and speculation [39].

- **Interlock**

  Interlock refers to locking the upstream transmitter and downstream receiver after the pipeline data transmission encounters a RAW data hazard and waits for the hazard to be resolved to continue transmission, as shown in Fig. 3.23. Interlock can solve RAW hazards, but neural network acceleration requires data transmission without bubbles. Interlocked waiting cycles do not contribute to the efficiency of neural network acceleration. However, it is worth mentioning that since there is no data switching in SRAM during the waiting time, this method can reduce the dynamic power consumption of FPGA.

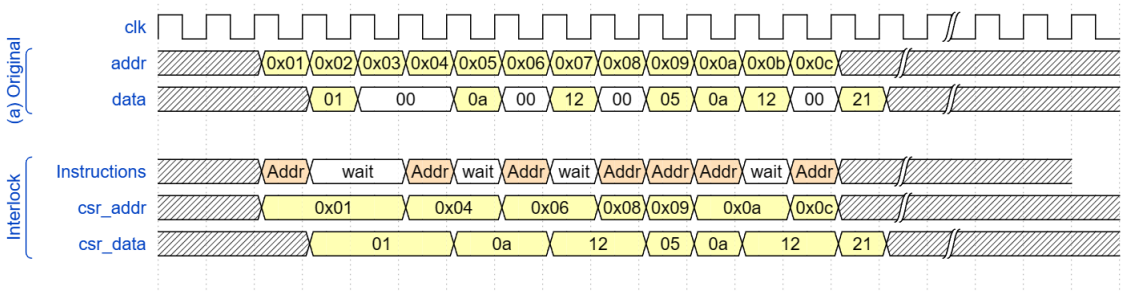


Figure 3.23: Interlock data transmission

- **Bypassing**

  Bypassing refers to data bypassing the read and write RAM operations and directly participating in the CSR data transmission. In FPGA, bypassing is usually done by combinational logic, as shown in Fig. 3.24.

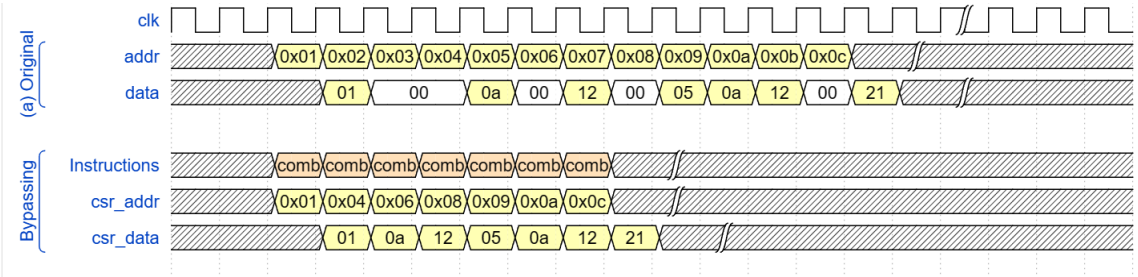


Figure 3.24: Bypassing data transmission

This combinational logic chain is a complex process with several steps. For example, assuming $DATA\ 4$ is the non-zero data in those 4 data, the process is shown in Fig. 3.25.

In step (a), 4 cached data are read from the cache by burst transmission, and then in step (b), it performs non-zero value addressing on the cached data one by one. If it is a non-zero value, finally, in step (c), the indices are transmitted directly without write-back to $CSRBuffer$ RAM.

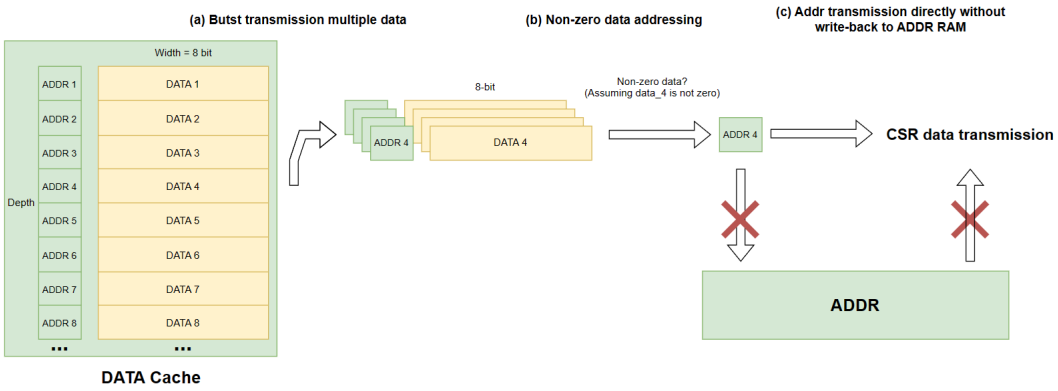


Figure 3.25: Bypassing data transmission process

Since it's a combinational logic chain, this process executes within one clock cycle. This means that such a process can easily cause timing violations in STA due to long combinational logic, especially in high frequency (100M in this work). This method was the strategy at the beginning of this work. However, the worst native slack (WNS) under pipeline and bit-width optimization has reached positive. Since burst transmissions require too many FFs and BRAMs, this has redundancy on resources and power consumption. Therefore, bypassing strategies on CSR transfers is not recommended.

- **Speculation**

The solution employed in this work to tackle RAW hazards is speculation, also known as prediction. Speculation involves predicting the CSR indices of non-zero data at time $t+1$ in the next frame while the data is currently being transmitted at time t in the current frame. Due to the hardware's parallel pipeline structure, the current data operation and next data speculation processes can be executed simultaneously in parallel. Through speculation, the write operation is always completed in the previous frame, and the read operation is always completed in the current frame. The current frame operation can always read the updated write data of the previous frame, so there is no RAW hazard. It is similar to the dynamic branch prediction of the CPU.

The sparse matrix example is shown in Fig. 3.22. Firstly, assuming this matrix is multiplied by another matrix B, the ratio of matrix B nodes to the number of parallel channels is 2. Therefore, from Eq. 3.13, each frame of the original sparse matrix requires 24 clock cycles before the CSR algorithm, as shown in Fig. 3.22 (a). Then, since the value of the transmitted sparse matrix on each frame is reflected in the features of each time step, the first frame can be regarded as the first time step, assuming $Current\,t$, and the second frame can be assumed as $Next\,t+1$. In the speculation method, this IP will predict the non-zero value indices of the frame at $Next\,t+1$ while the hardware transmits the frame of $Current\,t$, as shown in the Fig. 3.26. Specifically, while transmitting the first frame, due to the parallel architecture of the hardware, this IP will simultaneously predict the CSR indices of the second frame and store them in $CSRBuffer$ RAM. Then, at the start of transmitting the second frame, $CSRBuffer$ RAM will read out the stored CSR indices, which have been written in the first frame, to implement the CSR algorithm. At

the same time, it will also predict the CSR indices of the third frame and store them in $CSRBuffer$ RAM. This cycle continues until the entire sparse matrix is processed.

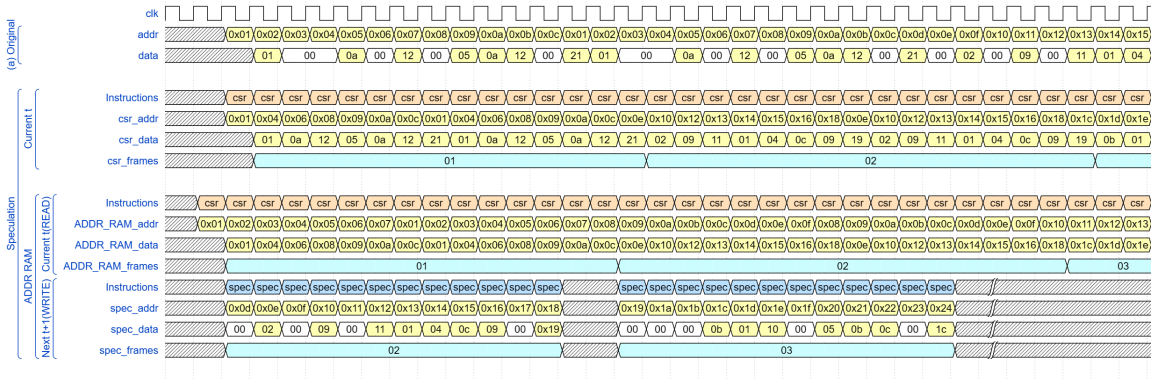


Figure 3.26: Speculation data transmission process

As Fig. 3.26 shows, the CSR algorithm can be fully deployed, and the entire hardware pipeline transmission has no bubbles through the speculation method. Moreover, unlike the method of bypassing, the entire speculation process is advanced based on the hardware pipeline, so there are no timing violations in the combinational logic. And it can achieve the purpose of high-frequency transmission. But it is worth noting that, to ensure pipeline transmission, the timing of the next frame $Next\,t+1$ speculation must be shorter than the timing of the current frame $Current\,t$ operation. The timing of the next frame $Next\,t+1$ speculation is equal to the sparse matrix nodes $Nodes\,A$, but the timing of the current frame $Current\,t$ operation is uncertain, which is related to the CSR indices of the previous frame. This rule can be summarized as Eq. 3.14.

$$\begin{aligned} Speculation\,Timing = Nodes\,A &< \frac{Nodes\,A * Nodes\,B}{CHANNELS} * FRAMES * (1 - Sparsity) \\ 1 - Sparsity &> \frac{CHANNELS}{Nodes\,B * FRAMES} \end{aligned} \tag{3.14}$$

If the number of channels and frames is substituted into 8 and 32 of this work, Eq. 3.14 can be simplified to Eq. 3.15.

$$1 - Sparsity > \frac{1}{NodesB * 4} \tag{3.15}$$

$Nodes\,B$ refers to the dimension of the multiplicand matrix (weights), which is also the nodes of the next layer of the current layer network. The sparsity after the delta algorithm is low, generally only about 50%, but $Nodes\,B$ is relatively large, with more than 32 nodes. Fortunately, it has been verified that the speculation method-based CSR algorithm of this work accelerates without bottlenecks in 98.5% of cases, but the speculation method also has many limitations, which will be discussed in Chapter 4.

Read-Write (RW) Conflict

In hardware design, read-write (RW) conflicts arise from simultaneous read-and-write operations on the same address of the same dual-port SRAM. In this work, RW conflicts mainly occur in the reading and writing of $CSRBuffer$ RAM. To avoid redundancy of BRAM resources,

$CSRBuffer$ RAM will only store the CSR indices of a single frame. Therefore, $CSRBuffer$ RAM needs to be updated with the operation of each frame. This implies that the $CSRBuffer$ RAM should write the data in the same clock cycle as it reads the data, ensuring a continuous and uninterrupted data transmission without introducing bubbles in the process. As a result, the reading and writing of data at the same address is uncertain.

RW conflicts often occur when SRAM reads and writes to the same address simultaneously, as shown in Fig. 3.27 for example. Assume that within a single clock cycle, when the $CSRBuffer$ RAM (assumed the depth is 8 and the width is 4-bit) reads out the required CSR indices (assumed to be $Address\,1\,old$), the DeltaFC IP provides an updated CSR indices for the next frame (assumed to be $Address, 1, new$), which needs to be promptly written to the $CSRBuffer$ RAM. However, since the read and write indices of the previous frame and the next frame are the same, the reading and writing of $CSRBuffer$ RAM are directed to the same address (assumed to be $CSRBuffer\,1$). Read operations and write operations are actually in sequence. This sequence depends on effects such as implementation routing delay and clock skew, so the sequence is uncertain. In other words, $CSRBuffer$ RAM can read first and then write, or write first and then read. Therefore, the address $Address\,1$ stored in $CSRBuffer\,1$ of $CSRBuffer$ RAM is also uncertain.

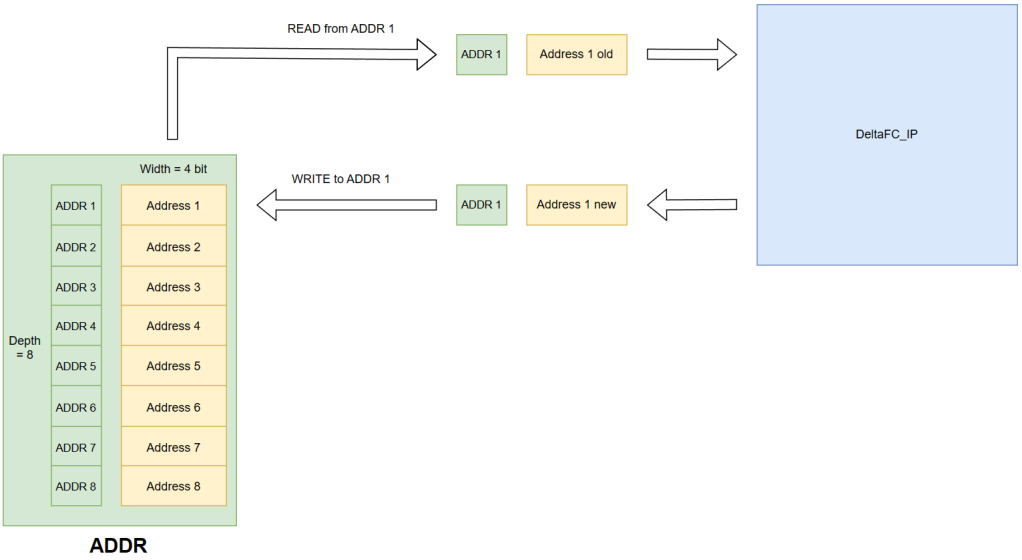


Figure 3.27: RW conflicts often occur when SRAM reads and writes to the same address simultaneously

Unlike RAW hazards, RW conflicts' read and write operations occur simultaneously, which cannot be separated by time. Therefore, the solution to avoid RW conflicts is to separate the space of read and write operations, that is, to operate on different addresses. Therefore, after reading and writing space separation, the optimized $CSRBuffer$ RAM is shown in Fig. 3.28. In the optimized $CSRBuffer$ RAM, the read operation of this frame will be read from the address from $CSRBuffer\,1$ to $CSRBuffer\,8$, while the write operation will be written from the address from $CSRBuffer\,9$ to $CSRBuffer\,16$, and vice versa for the next frame. Address separation optimization can ensure that only one address is operated in one clock cycle, thereby avoiding RW conflicts.

It's noteworthy to mention that the optimization involving address separation leads to a doubling of the original size of the BRAM area. Nevertheless, upon evaluation, the increase in area was found to be insignificant, and the resource utilization are tolerable.

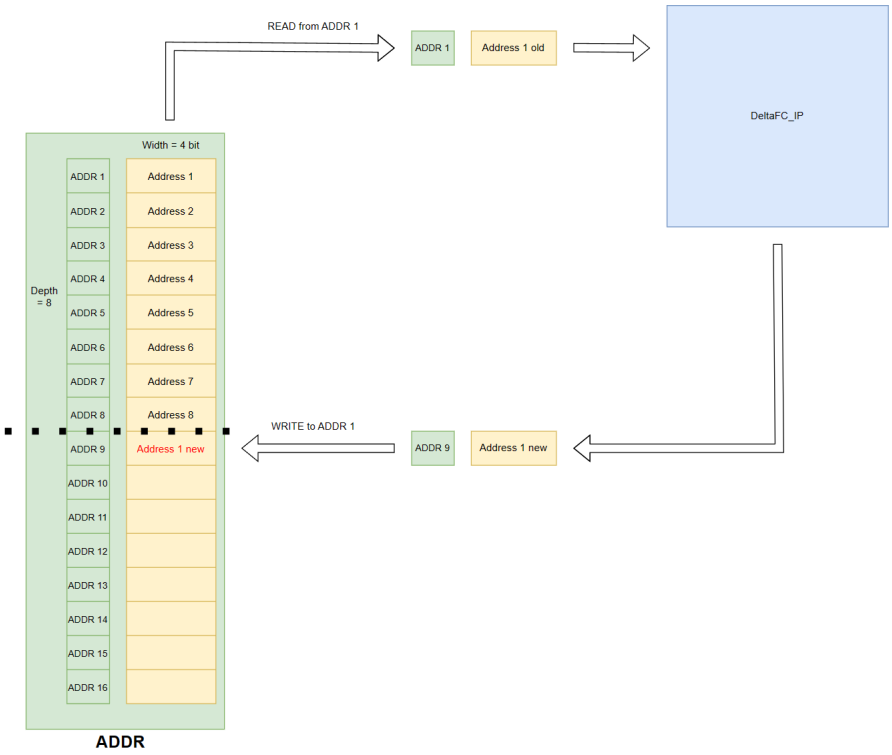


Figure 3.28: Optimized CSR Buffer RAM can solve RW conflicts

CSR Algorithm Acceleration

This section will discuss how the CSR algorithm accelerates the operation of neural networks. The CSR algorithm extracts the indices of the sparse matrix in each frame, but it only compresses the sparse matrix. Since the MAC operation of the neural network can be regarded as mainly the matrix multiplication of sparse matrices and weights, the CSR indices also need to address the weights correspondingly.

Still taking the matrix in Fig. 3.21 as an example. Assume that this sparse matrix is multiplied by the weight matrix with dimensions of 3×12, shown in Fig. 3.29 (a). Fig. 3.29 (b) shows a simplified diagram of (a), where 125 represents the 3 channels parallel form of columns $w1$ to $w25$, and so on. It can be seen that the weight matrix of this example is not sparse, and the weights of the DeltaFC trained in this work are also non-sparse matrices. For easier comparison, the nodes and the number of parallel channels ratio of the weights example here is set to 1.

w1	w2	w3	w4	w5	w6	w7	w8	w9	w10	w11	w12
w13	w14	w15	w16	w17	w18	w19	w20	w21	w22	w23	w24
w25	w26	w27	w28	w29	w30	w31	w32	w33	w34	w35	w36

(a) Weights example

125	226	327	428	529	630	731	832	933	1034	1135	1236

(b) Weights simplified example

Figure 3.29: Weights example, which has dimensions of 3×12

The process of multiplication under CSR acceleration of the sparse matrix (shown in Fig. 3.21) and the example of the weight (shown in Fig. 3.29) is shown in Fig. 3.30. First, step (a) performs the CSR algorithm on the sparse matrix alone to extract CSR indices. Then, in step (b), it is worth noting that the sparse matrix and the weights matrix need to be CSR addressed. Then, in step (c), the non-zero elements of the two matrices addressed by CSR indices will perform MAC operations in one-to-one correspondence. Not only are the 0 values of the sparse matrix skipped, but also the weight matrices. The final step (d) is to merge the results of these three frames.

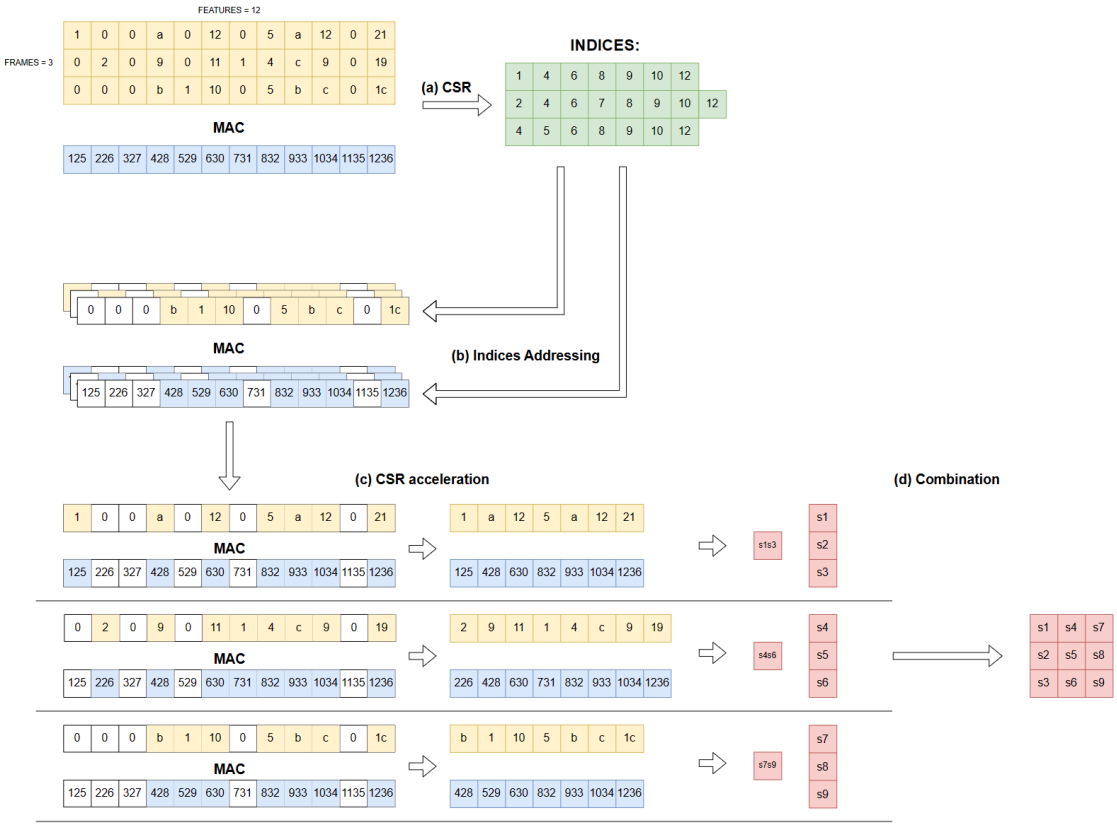


Figure 3.30: Process of CSR Acceleration

The CSR algorithm acceleration timing diagram is shown in Fig. 3.31. It can be observed that the acceleration effect of the CSR algorithm is very obvious. In the original matrix MAC operation without CSR acceleration, it takes 36 clock cycles to complete, but after CSR acceleration, it only takes 22 clocks, reducing the MAC timing by 30.6% and increasing the operation efficiency by 63.6%. The timing and efficiency of CSR acceleration are related to temporal sparsity, according to the definition of temporal sparsity introduced in Chapter 2, which refers to the ratio of the count of 0 values to the number of input features in input signals. Therefore, the relationship between the timing and efficiency improvement of CSR acceleration and temporal sparsity can be expressed as Eq. 3.16 and Eq. 3.17.

$$Timing = Original\,Timing * (1 - Temporal\,Sparsity) \tag{3.16}$$

$$Efficiency\,Improvement = \frac{Temporal\,Sparsity}{1 - Temporal\,Sparsity} \tag{3.17}$$

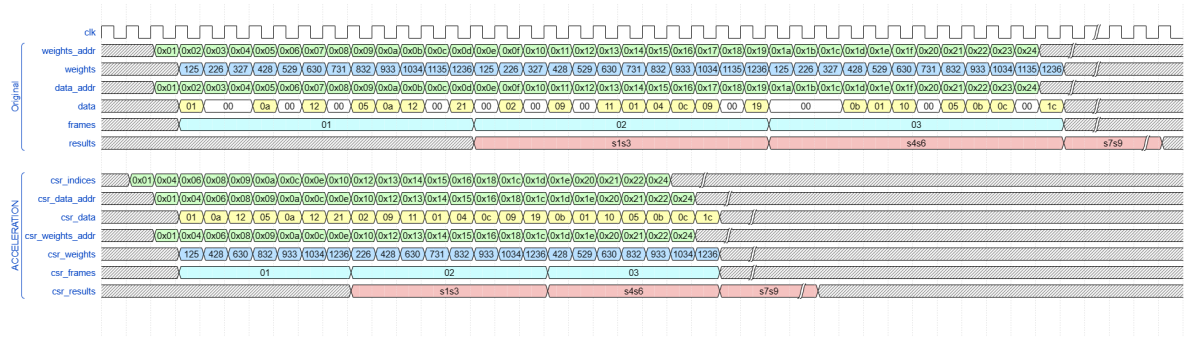


Figure 3.31: Timing diagram of CSR algorithm acceleration

The CSR algorithm plays a very important role in the field of sparse matrix acceleration. It accelerates the MAC operations by compressing sparse matrix. In addition, it is worth noting that because the CSR algorithm can reduce the scale of data transmission, it can also save memory resources in proportion to temporal sparsity.

3.2.4. Delta Algorithm Implementation On FPGA

The Delta algorithm is fundamental for transforming an original temporal-related matrix into a sparse matrix. However, when deploying the Delta algorithm in hardware, it is essential to consider the hardware architecture and the memory resource configuration of the FPGA hardware.

Multistage Pipeline

In software design, the input matrix undergoes Delta conversion frame by frame sequentially. However, in hardware, the input matrix undergoes Delta conversion feature by feature sequentially in the pipeline. The Delta algorithm-based PE operation of a single element in detail is shown in Fig. 3.32. Fig. 3.31 omits all write-back operations for updates to simplify the process.

- **1. Addressing**

  The $CSRBuffer$ RAM introduced previously stores the CSR indices. These indices will address the original elements of the $Data\,cache$, $Prev$ RAM, and $input\,weight$ RAM that the Delta algorithm has not processed. Assume that the output by the $Data\,Cache$ RAM is $curr\_part$ and the output by $Prev$ RAM is $prev\_part$.

- **2. Differentiation**

  The $curr\_part$ and $prev\_part$ processed in step 1 execute the Delta algorithm. This will go through three steps. The first step is to take the absolute difference ($Delta\_value$) between the $curr\_part$ and the $prev\_part$. The second step is to perform threshold filtering on the $Delta\_value$. The third step is to write back the updated $prev\_part$ to $Prev$ RAM based on the threshold filtering result.

- **3. MAC**

  The Delta value processed in step 2 participates in the MAC operation. Due to the speculation in the previous frame, all Delta Values here are non-sparse.

- **4. Integration**

  The delta memory output by the $Delta, Memory$ RAM will be combined with the MAC product processed in step 3 and then subjected to the ReLU activation function. Subsequently, the updated delta memory will be written back to the $Delta, Memory$ RAM.

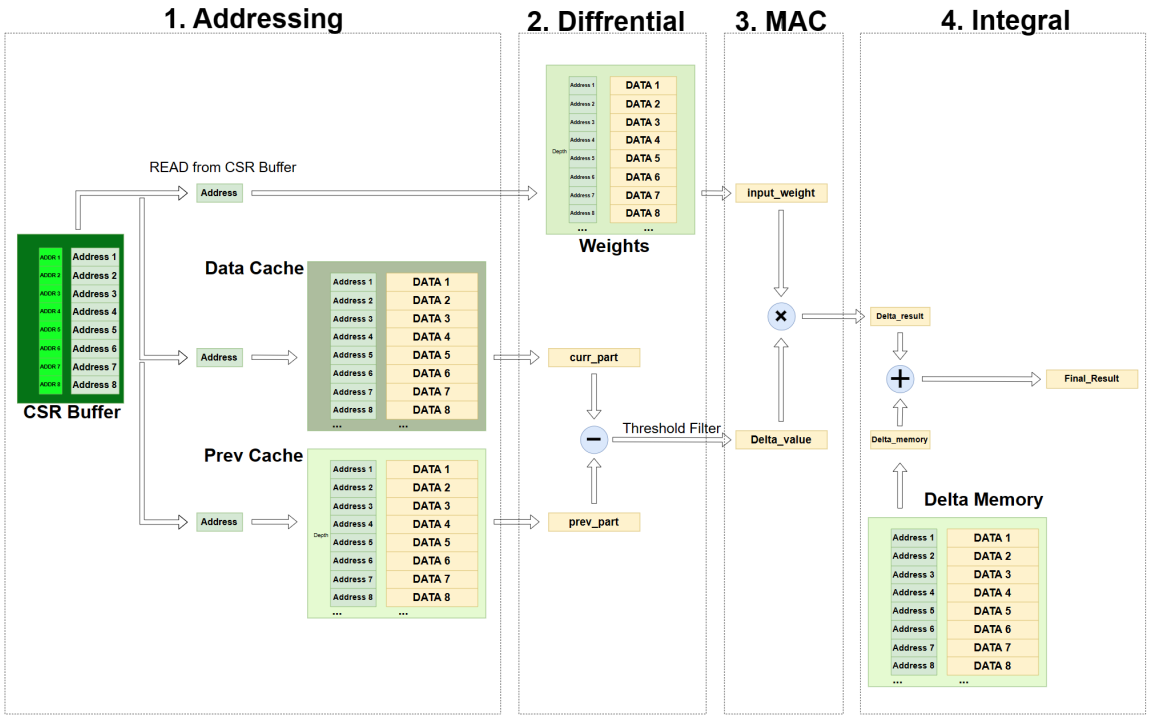


Figure 3.32: Process of Delta Algorithm Implementation on FPGA (omits all write-back operations for updates)

The Delta algorithm is a complex process, so a 4-stage pipeline architecture is used to deploy the Delta algorithm on hardware, as the timing diagram shown in Fig. 3.33. Although the multi-stage pipeline architecture requires more area and resources for cache or logic segmentation, it allows the timing of the FPGA hardware to be stable and run faster. This architecture prioritizes performance and speed optimization at the expense of increased area utilization.

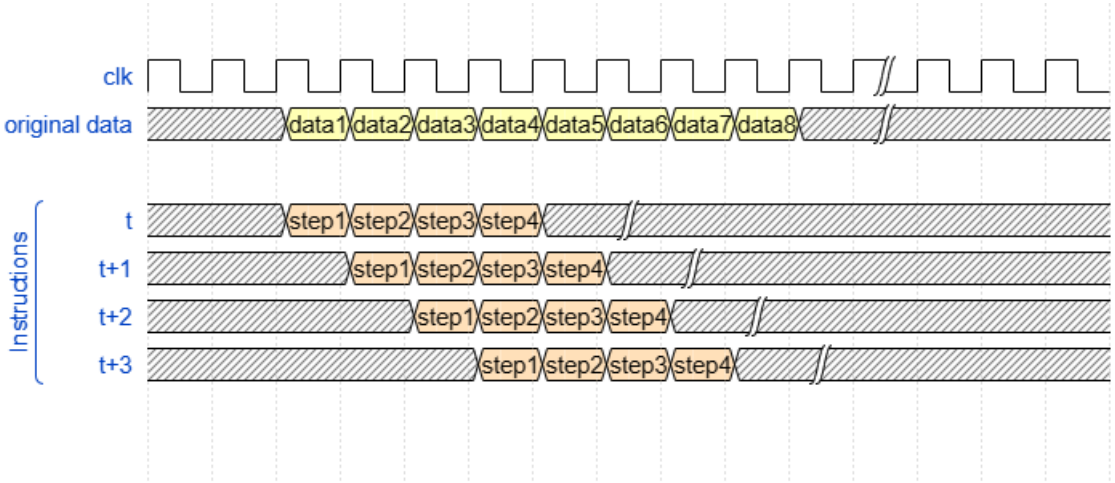


Figure 3.33: Timing diagram of a 4-stage pipeline of Delta algorithm deployment

Dual-port SRAM Application

As previously introduced, from the speculation perspective, it is known that the CSR algorithm must process the current frame and simultaneously speculate on the next frame. The pre-

algorithm of the CSR algorithm is the Delta algorithm, which can transform the original non-sparse dataset into a sparse dataset. The delta algorithm operation must also be performed during the speculation process on the current and next frames. Since the Delta algorithm involves many cache tasks, accurately reading and writing the memory of the $Previous$ $Current$ and $Next$ frame is the key to deploying the delta algorithm.

To cache various data used for Delta algorithms effectively, this work flexibly configures a variety of SRAMs, as shown below.

- Weights_RAM

  This SRAM is used to read and write weights for DeltaFC IP processing. Since there is only initial writing and PE operation, reading and writing processes do not occur simultaneously. Therefore, this SRAM is configured as SPRAM.
- Prev_RAM

  This SRAM is used to read and write data of the previous frame for the Delta differentiation algorithm. However, as mentioned previously, $Prev$ RAM needs to update the cache of the previous frame, which means $Prev$ RAM has to read and write simultaneously. To avoid RW conflicts, reading and writing need to be separated. This SRAM is configured as SDPRAM.
- Prev_ACC_RAM

  This SRAM is used to read and write data of the next frame for the Delta algorithm before speculation. Since the cache of the next frame also needs to be updated as frames. In the same way, to avoid RW conflicts, this SRAM is configured as SDPRAM.
- CSR Buffer_RAM

  This SRAM is used to read and write CSR indices for the CSR algorithm. Since the CSR indices also need to be updated as frames. In the same way, to avoid RW conflicts, this SRAM is configured as SDPRAM.
- Delta_Memory

  This SRAM is used to read and write the delta memory for the Delta integration algorithm. Since the delta memory also needs to be updated as frames. In the same way, to avoid RW conflicts, this SRAM is configured as SDPRAM.
- Data_Cache_1 & Data_Cache_2

  These SRAMs are used to read and write the original dataset for DeltaFC IP processing. However, since the processing of cache data is flexible, not only do the Delta or CSR algorithms need to read and write to update the cache simultaneously, but both ports must also read or write simultaneously during the speculation process. This means that the two ports of SRAM are not fixed for reading and writing but are free for reading and writing. The only SRAM that can be freely configured with two ports is TDPSRAM. So, this SRAM is configured as TDPRAM.

Obviously, to deploy the Delta algorithm and CSR algorithm, a large amount of SRAM resources must be consumed for caching. This is also a common problem with timing hardware algorithms and neural network IP. But in any case, all BRAM resource consumption is within the controllable range. For example, the BRAM consumption in this work does not exceed 50% of the FPGA hardware resources.

3.2.5. Interface Design

In essence, the neural network accelerator in the hardware design of this work is a digital IP that serves as the PL side of the ZYNQ board. Interface design is necessary for data communication between PL and PS. This includes the AXI4-stream interface design and the IO ports design of the digital IP of the neural network digital IP.

AXI4-stream

The AXI4-stream protocol is a high-speed communication protocol dedicated to streaming data transmission in digital systems [38]. Streaming data mainly includes video, audio, etc. The AXI4-stream protocol has the advantages of low latency, simple structure, and easy integration, so it is suitable for streaming data transmission with strict real-time and flexible reconfiguration requirements. This work uses the AXI4-stream interface protocol for communication between the PL and the PS.

AXI4-stream significantly differs from other AXI4 series protocols, mainly reflected in the different interface definitions. First, unlike the AXI4-full and AXI4-lite protocols, the AXI4-stream protocol does not have address lines, meaning the channel is not memory-mapped. Secondly, since AXI4-stream focuses on streaming data transmission, there is no write response channel to achieve high-throughput intensive data communication. In addition, since AXI4-stream is suitable for one-way transmission of stream data, there are no write and read channels but a one-way channel composed of the master end and the slave end.

Because this work aims to implement a lightweight neural network on hardware design, the interface definition should also be simplified as much as possible to meet basic data communication needs. In this work, a single DeltaFC digital IP as the slave has the following AXI interfaces:

- 1. TDATA

  The TDATA channel is the most important channel of the AXI4-stream protocol, which contains valid data for stream data transmission. In this work, the TDATA channel includes $s\_axis\_input\_tdata$ and $s\_axis\_weights\_tdata$.
- 2. TVALID

  The TVALID channel indicates whether the master end is valid for the data transmission. In this work, the TVALID channel includes $s\_axis\_input\_tvalid$ and $s\_axis\_weights\_tvalid$.
- 3. TREADY

  The TREADY channel indicates whether the slave end is ready for the data transmission. In this work, the TREADY channel includes $s\_axis\_input\_tready$ and $s\_axis\_weights\_tready$.
- 4. TLAST

  The TLAST channel flags the last package of streaming data. In this work, the TLAST channel includes $s\_axis\_input\_tlast$ and $s\_axis\_weights\_tlast$.

IO ports design

The AXI4-stream interface protocol mentioned in the previous subsection is for the purpose of high-speed streaming data communication. In this section, the data transmitted by IO ports will be discussed.

It can be seen from the previous hardware design the fundamental inputs of the neural network mainly include data and weights, and the outputs of the neural network include neural network calculation results. But in terms of hardware implementation, except for AXI4-stream interfaces, adding a synchronized clock and reset input on the hardware design is necessary. On the other hand, the hardware neural network deployed on the Zynq board also includes

SDK development, which involves communication between the PL and the PS side. More straightforwardly, the ARM core of the PS is supposed to accept the feedback signal from the FPGA of the PL to interrupt, reset, etc. Therefore, in PL hardware development, designing the feedback signal from FPGA to PS, such as the start signal 'START' and the end signal 'END_FLAG' of the recognition task in this work, is also necessary. This IP interface diagram is shown in Fig. 3.34.

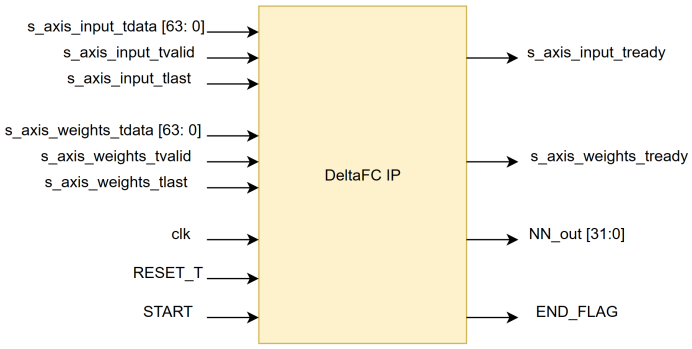


Figure 3.34: DeltaFC digital IP diagram

4
Results

4.1. Experiment Setup

This section primarily focuses on establishing the testing environment platform, including preparing software and hardware datasets and a general overview of the evaluation indicators.

4.1.1. Software Design Setup

Dataset Preparation

The dataset used in the training process of this study is divided into two types of datasets: speech and noise. On the one hand, the speech dataset uses Libri-Speech, a corpus of approximately 1000 hours of 16kHz read English speech, prepared by Vassil Panayotov with the assistance of Daniel Povey. The data is derived from reading audiobooks from the LibriVox project [23]. On the other hand, the noise dataset uses QUT-noise. The QUT-NOISE-TIMIT corpus consists of 600 hours of noisy speech sequences designed to enable a thorough evaluation of voice activity detection (VAD) algorithms across various common background noise scenarios [8].

They must undergo several processing steps to transform these datasets into input features for DeltaFC DNN.

- 1. Sampling the source audio

  The first step is to traverse all the audio files in the data sets, set the sampling parameters to sample those source audios, and then save the normalized array of sampled audio files in the 'raw' dictionary of the data class and shuffle. The sampling parameters are shown in Tab. 4.1.

Sample Rate	16000 Hz
Sample Width	2
Sample Channels	1

Table 4.1: Dataset sampling parameters

- 2. Conversion from audio files to fixed frames

  This step is to convert all the files with different frame rates of the source audio files 'raw' into fixed frame rates and store these converted frames in the 'frames' dictionary of the data class. The size of each frame is 30ms.

- 3. Labeling the sampled frames

  This step aims to recognize and label the converted frames of the second step. The WebRTC vad library developed by Google [12] is used as the benchmark label of the data set itself.

- 4. Extracting the dataset features

  This step aims to extract the audio features through MFCC. The MFCC parameters are shown in Tab. 4.2. After MFCC, the original one-dimensional audio waveform array can be converted into a two-dimensional matrix containing spectral feature vectors over time, which will be used as the neural network input.

Sample Rate	16000 Hz
Window Step	30 ms
Window Length	120 ms
NFFT	2048

Table 4.2: The MFCC parameter settings for each 30ms audio segment.

- 5. Partitioning the dataset

  This step divides the train, test, and validation sets into the 0.8, 0.1, and 0.1 ratios. These divided datasets are used for neural network training and testing and neural network validation.

To use the GPU to accelerate the CUDA training of neural networks more efficiently, this work uses 2048 batches of parallel training. The previous operations divided the data set into 24 features, 32 frames, and 2048 batches as the basic training, testing, and validation data package.

Receiver Operating Characteristic (ROC)
ROC curves are commonly utilized to visually represent the trade-off between clinical sensitivity and specificity for various possible cut-off values in a test or a combination of tests [10]. ROC was originally the data of the positive diagnosis rate analysis in medicine, but this visualization tool is also applicable in the field of neural network classification. The ROC curve reflects the response of the neural network model to stimuli at different points on the curve (also known as sensitivity). The model's probability of accurately recognizing different points on the curve is reflected in voice activity detection. The accurate recognition probability here can be expressed by FRR (False Rejection Rate) and FAR (False Acceptance Rate).

In the domain of voice activity detection, where the model output consists solely of noise and speech (a two-category model), False Rejection Rate (FRR) and False Acceptance Rate (FAR) can be expressed as shown in Eq. 4.1.

$$\begin{aligned} FRR &= \frac{FN}{FN+TP} \\ FAR &= \frac{FP}{FP+TN} \end{aligned} \tag{4.1}$$

FN (False Negative), FP (False Positive), TN(True Negative), and TP(True Positive) are the predicted results obtained from the neural network confusion matrix. Where FN, FP, TN, and TP represent results as follows:

- FN: predicted as noise actually speech
- FP: predicted as speech actually noise
- TN: predicted as noise actually noise
- TP: predicted as speech actually speech

FRR represents the proportion of actual noise in all speech predicted by the model, and FAR represents the proportion of actual speech in all noise predicted by the model. Both the lower and the better; this means the model has a clear classification for speech and noise. The ROC curve uses FAR and FRR as coordinate axes, reflecting the model's sensitivity to both speech and noise.

The TN, TP, FN, and TP obtained from the neural network confusion matrix can also be used to calculate accuracy, as shown in Eq. 4.2. As the definition suggests, accuracy reflects the proportion of correctly recognized speech/noise samples in the entire sample set in audio classification.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.2}$$

ROC is a probability curve, and AUC represents the degree or measure of separability. It tells how much the model is capable of distinguishing between classes. An excellent model typically exhibits an AUC close to 1, indicating a high level of separability [30].

4.1.2. Hardware Design Setup

Dataset Quantization

Since floating point calculations cannot be performed directly in the hardware without a dedicated IP, the FPGA hardware input dataset must be transformed through fixed-point quantization. The hardware dataset input here includes the dataset itself and the neural network weights.

To reproduce the neural network results of the software to the greatest extent, the input dataset quantization of the hardware adopts a 16-bit fixed-point quantization. However, since the weight of the software shows a certain normalization after neural network training, the weights can adopt 8-bit fixed-point quantization.

FPGA Board

The ZYNQ board model used in this work is MiniZed (xc7z007sclg225-1). It includes 512 MB DDR3L memory and various interfaces (such as HDMI, USB, and Ethernet). Besides, it features an ARM Cortex-A9 dual-core processor embedded for PS development [2]. However, although MiniZed's integrated units are diverse, their physical size is small, there are few on-board resources, and the layout is compact. Due to its lightweight design, it is easy to carry. Therefore, it is suitable for small-scale hardware FPGA design, such as the lightweight neural network of this work. The on-board resources parameters of MiniZed are as shown in the Tab. 4.3.

Slice LUTs	Slice Registers	Slice	LUT as Logic	Block RAM Tile	DSPs
14400	28800	4400	14400	50	66

Table 4.3: On-board resources parameters of MiniZed

PPA(Performance, Power, Area)

FPGA's PPA (Performance, Power, Area) are the most important indicators to evaluate the quality of FPGA design. PPA involves evaluating FPGA hardware design from three aspects:

- 1. Performance

  Performance indicates how well hardware implements functions. In this work, hardware performance includes the following evaluation indicators: STA slacks, accuracy, and acceleration efficiency.

- 2. Power

  Power consumption refers to the power consumption of hardware to perform tasks per frame.
- 3. Area

  The area represents the resource utilization of hardware. In this work, this includes resource utilization.

4.2. Experiment Results

4.2.1. Software Design Results

This work will first test the trained DeltaFC DNN on software and evaluate the performance of the DeltaFC. The performance of DeltaFC DNN in software design is mainly evaluated through the following aspects:

- 1. DeltaFC DNN basic parameters

  This mainly discussed the performance comparison of DeltaFC and other neural networks, including accuracy AUC, FAR & FRR, and other indicators.
- 2. DeltaFC DNN temporal sparsity

  This mainly includes evaluating the impact of temporal sparsity induced by DeltaFC on accuracy.
- 3. DeltaFC DNN testing actual results

  This mainly includes the test results of DeltaFC on a test dataset audio (train set unseen), compared with the test results of Google's WebRTC, which serves as the baseline for comparison.

Results of comparison between DeltaFC and others

In this subsection, this study uses 4 different types of neural networks for horizontal comparisons across various metrics such as AUC, Accuracy, and FAR/FRR. Among them, the LSTM network is referenced from Niklas Hansen [14], and DeltaGRU is referenced from Chang Gao [11]. To verify the DeltaFC's performance from the classification perspective, this work uses other 3 neural networks for comparison.

To ensure a fair comparison, the parameters for all neural networks are set as follows:

- 1. All network parameters are controlled below 40K to reach the demand for lightweight.
- 2. All networks use the same datasets, Librispeech for speech and QUT-noise for noise.
- 3. All network training processes adopt the ADAM optimizer, using the same momentum (0.9) and learning rate (0.01).
- 4. All network training processes employ early stopping to prevent overfitting and exploding gradients.

As a result, the parameters of those 4 networks are shown in the Tab. 4.4. In this part of the work, to balance with the optimization of other networks, the threshold of DeltaFC is chosen as 0.

- **ROC(Receiver Operating Characteristic) and AUC(Area Under Curve)**

  In the field of VAD, AUC reflects the model's classification performance for noise and speech. The AUC results are shown in Fig. 4.1.

  It can be seen that the classification performance of FC and DeltaFC for speech and noise is the highest in this work of lightweight networks with parameters less than 40K,

Network type	Parameters	Early stopping	Stopping epoch
FC	27202	True	70
DeltaFC	27202	True	53
LSTM [14]	34128	True	118
DeltaGRU [11]	38608	True	99

Table 4.4: The parameters of 4 different basic neural networks tested in this work. The threshold of DeltaFC is 0, which is equivalent to FC mathematically

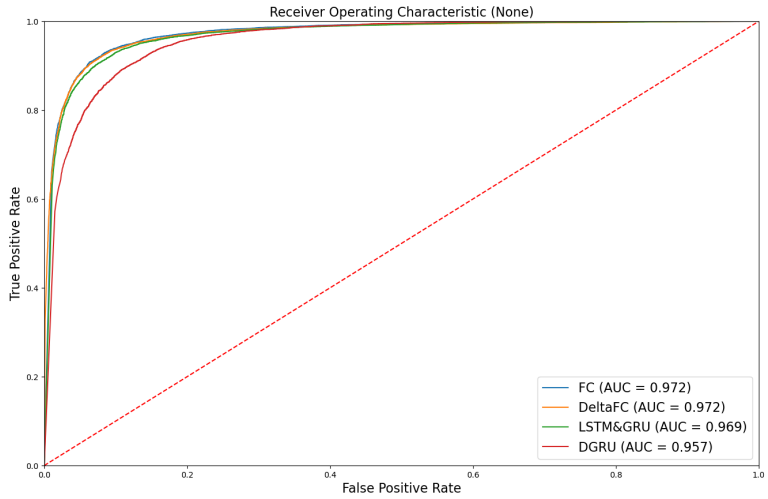


Figure 4.1: AUC results of 4 networks. The threshold of DeltaFC is 0, which is equivalent to FC mathematically

reaching 0.972. This shows that introducing the FC structure has slightly improved classification performance.

- **Accuracy and FAR&FRR**

  The accuracy and corresponding FAR&FRR of the four networks are shown in Tab. 4.5. FAR and FRR are controlled within the range of 8% $\pm$ 1 % as much as possible to facilitate comparison.

Network type	Accuracy	FAR	FRR	Fixed FAR	Fixed FRR
FC	91.26%	12.24%	5.05%	8.65%	6.99%
DeltaFC	91.85%	10.67%	5.53%	7.23%	8.45%
LSTM [14]	91.48%	9.05%	8.13%	9.07%	8.06%
DeltaGRU [11]	88.94%	10.84%	10.91%	14.09%	8.02%

Table 4.5: Accuracy and FAR&FRR results of 4 networks. The threshold of DeltaFC is 0, which is equivalent to FC mathematically

It can be seen that among all networks, the classification accuracy performance of sequential networks such as DeltaFC is significantly higher than that of the other three. And the accuracy of DeltaFC is the highest, reaching 91.85%. But what is interesting is that the structure of DeltaFC is similar to FC, but the accuracy is higher than FC. The difference is due to the stochasticity during the neural network training process. Meanwhile, DeltaFC has the lowest FAR but the highest FRR. The main reason for this result is that DeltaFC is more sensitive to noise on the one hand, and it is related

to the selection of data sets on the other hand. But in summary, the classification performance of DeltaFC shows advantages compared with the other 3 basic neural networks, especially in aspects of parameter scale, AUC, and accuracy.

Results of DeltaFC temporal sparsity

As mentioned previously in Chapter 3, when the feature difference ΔX is sufficiently small, the operation can be skipped if the delta input falls below a specified threshold. This approach is implemented to enhance temporal sparsity in the data processing. However, a high threshold will cause the accuracy rate to collapse since too many key features are filtered out. Choosing a reasonable threshold will help accelerate the model without sacrificing accuracy, which requires a balance between temporal sparsity and accuracy. This section will explore the relationship between threshold, accuracy, and temporal sparsity in software design.

In this section, the discussion revolves around selecting a threshold without quantization. From the input features extraction after MFCC, the audio features of this work are generally concentrated between 0 and 1, especially the absolute value of features in the hidden layer of the DeltaFC are concentrated between 0 and 0.5. This demonstrates that the threshold selection from the range of 0 to 0.5 can filter most small features.

Firstly, keep the early stopping epoch of DeltaFC as 78, and then change the thresholds of all hidden layers in the range of 0 to 0.5, then can get relationships between accuracy & temporal sparsity and threshold curves, as shown in Fig. 4.2.

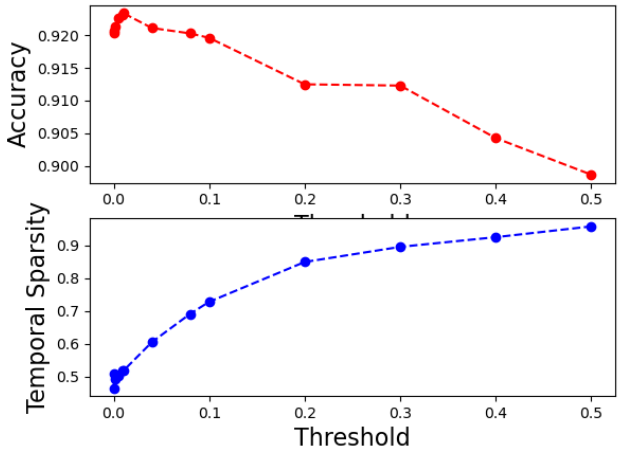


Figure 4.2: Relationships between accuracy & temporal sparsity and threshold from 0 to 0.5

Then, make a relationship curve where the threshold changes from 0 to 0.1, as shown in Fig. 4.3. It can be seen that, on the one hand, for accuracy, the accuracy gradually increases in the range of threshold changes from 0 to 0.01, reaches a peak at 0.01, and then gradually decreases in subsequent changes. When the threshold reaches 0.5, the accuracy collapses below 90% directly. On the other hand, for temporal sparsity, this curve is more similar to a power function whose power is less than 1 with threshold-related. This is because when the threshold is high, the occupancy of the features becomes more and more stable, and the temporal sparsity converges accordingly. The next section will discuss more detailed analyses of accuracy and temporal sparsity.

From the perspective of accelerating the operation speed of DeltaFC, it is hoped that the higher the sparsity of the model, the better to generate more 0s to realize the zero-jump operation to accelerate the neural network calculation. However, since the network is for voice

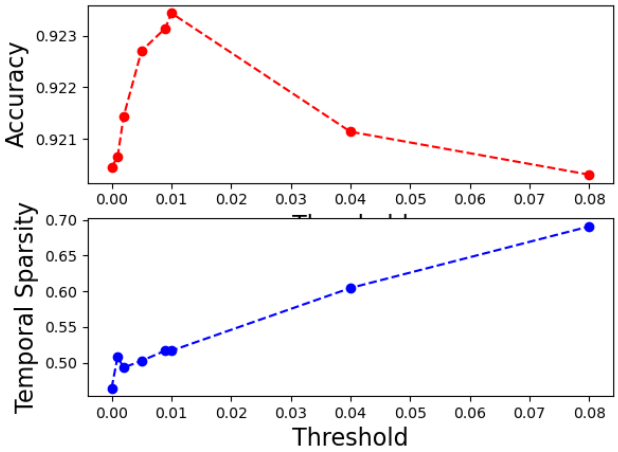


Figure 4.3: Relationships between accuracy & temporal sparsity and threshold from 0 to 0.1

activity detection, it becomes imperative to fulfill the requirements for computational speed and prioritize high accuracy. The selection of the threshold depends on the objective of the task.

Results of tested audio

The tested audio also uses part of the original (unseen training dataset). For the label of the tested audio, the WebRTC_vad [12] developed by Google for the WebRTC project is used as the benchmark, which can classify a piece of audio data as being voiced or unvoiced. The original audio labels classified by WebRTC_vad are shown in Fig. 4.4.

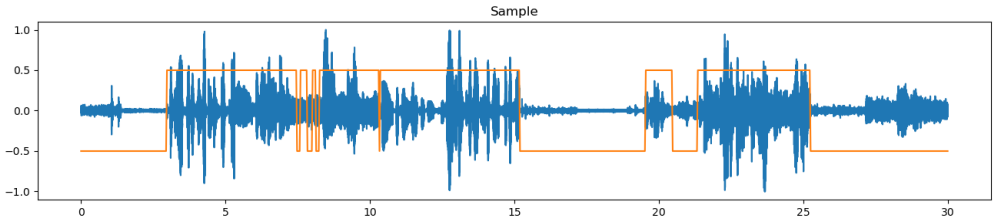


Figure 4.4: Audio part and voice label developed by WebRTC

The original audio labels predicted by DeltaFC in the threshold of 0.01 are shown in Fig. 4.5.

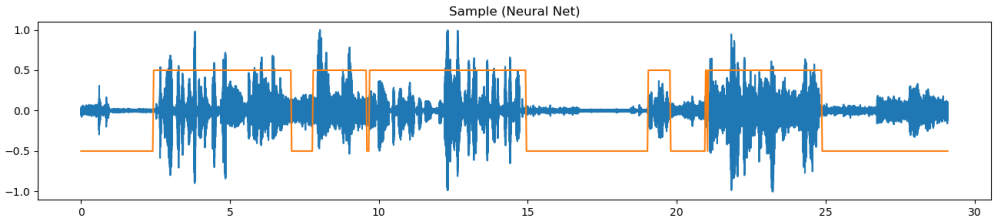


Figure 4.5: Audio part and voice label developed by DeltaFC

The noise and speech classification accuracy results predicted by the DeltaFC network are 92.3% compared with WebRTC_vad. This proves that the neural network has good detection accuracy in lightweight networks.

4.2.2. Hardware Design Results

This work will then deploy the DeltaFC DNN software design on FPGA hardware and implement hardware acceleration. The device diagram of this work is shown in Fig. 4.6. The evaluation of hardware design will be carried out from the following aspects of PPA:

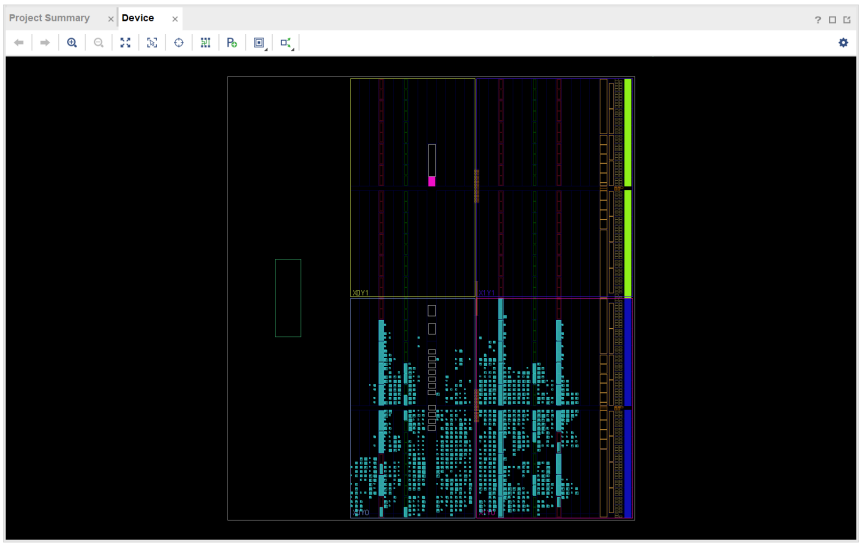


Figure 4.6: The device diagram of DeltaFC IP core

- 1. DeltaFC hardware area and utilization

  This mainly includes the area and utilization of digital IP after implementation.
- 2. DeltaFC hardware power consumption

  This mainly involves the power consumption evaluation of various modules in the DeltaFC hardware design.
- 3. DeltaFC hardware STA slacks

  This mainly involves the STA of DeltaFC hardware design, including setup and hold slacks evaluations.
- 4. DeltaFC hardware threshold variation on accuracy

  This mainly involves examining how adjusting the threshold from 0 to 0.9 affects the accuracy of DeltaFC on the hardware.
- 5. DeltaFC hardware threshold variation on throughput

  This mainly involves examining how adjusting the threshold from 0 to 0.1 affects the throughput of DeltaFC on the hardware.
- 6. DeltaFC hardware acceleration

  This mainly involves the impact of DeltaFC acceleration on timing and accuracy.

Results of Area And Utilization

This work first conducted out-of-context (OOC) synthesis and implementation for the Delta digital IP core. In the context, synthesis adopted strategy $'Flow_Perf_Optimized\_high * (VivadoSynthesis2022'$, and implementation adopted strategy $'Performance\_Auto\_1'$. After synthesis and implementation, the area and utilization of the DeltaFC IP core are shown in Fig. 4.7.

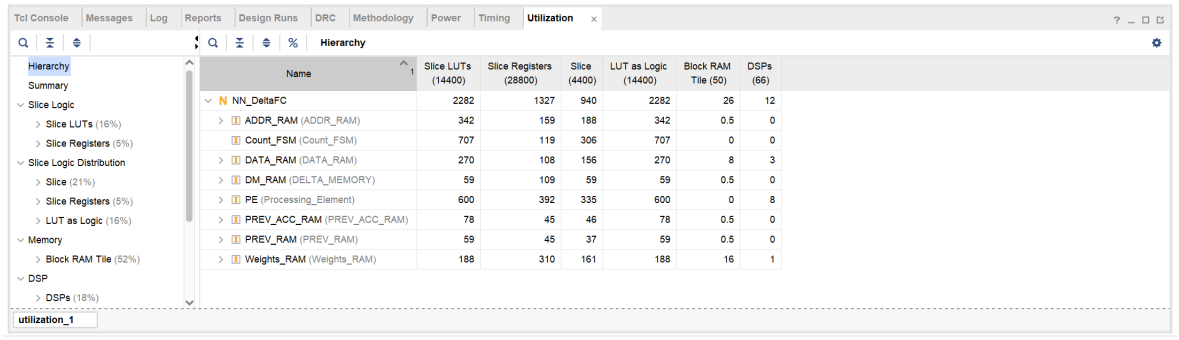


Figure 4.7: The area and utilization of DeltaFC IP core

It can be observed that the resource utilization of the DeltaFC IP, after implementation, is around 25% for all components except BRAM. However, the BRAM utilization reaches 52%. This is primarily due to the significant consumption of BRAM by the FPGA hardware for caching weight values and activation values in neural networks, which is inevitable. But overall, the FPGA hardware design has met the expected standards.

Results of Power Consumption

After synthesis and implementation, the power consumption of the DeltaFC IP core is shown in Fig. 4.8.

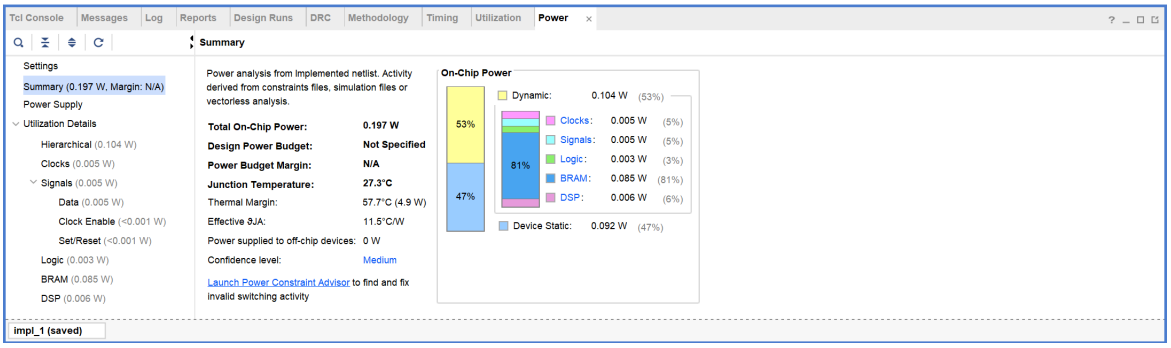


Figure 4.8: The power consumption of DeltaFC IP core

It can be observed that the power consumption of the FPGA neural network is primarily concentrated on BRAM instead of DSP, which reaches 81%. This is mainly because the SRAMs of the FPGA hardware for neural networks require frequent read and write operations of weights and activation values.

Results of STA slacks

After synthesis and implementation, the STA slacks of the DeltaFC numeric IP core are shown in Fig. 4.9.

It can be observed that, at a synchronous clock frequency of 100 MHz, there are no timing violations in the DeltaFC IP core. Both synthesis and implementation are free of critical warnings and latch designs. This is attributed to the strict adoption of a pipeline structure in all RTL modules of this work. Additionally, logic reuse and multi-register synchronization designs are implemented in multiple modules, ensuring that the possibility of combinational logic timing violations in the hardware design is minimized.

Figure 4.9: The STA slacks of DeltaFC IP core

Results of DeltaFC threshold variation on accuracy.
Accuracy is a benchmark metric for the DeltaFC hardware design. The accuracy evaluation is primarily conducted through quantized test datasets and weights input into the DeltaFC IP. To replicate software results and provide a more intuitive diagram of the impact of thresholds on accuracy, this work uses multiple threshold points ranging from 0 to 0.9 in the accuracy testing during this section. The testing process includes both hardware testing and software testing with corresponding thresholds. Since the quantized thresholds in hardware are supposed to be integers, for example, 0.1 in software design will be quantized to 2.56 but is reflected as 3 in the hardware. If not specified, all instances of thresholds in the following sections are quantized thresholds. The testing results are shown in Fig. 4.10.

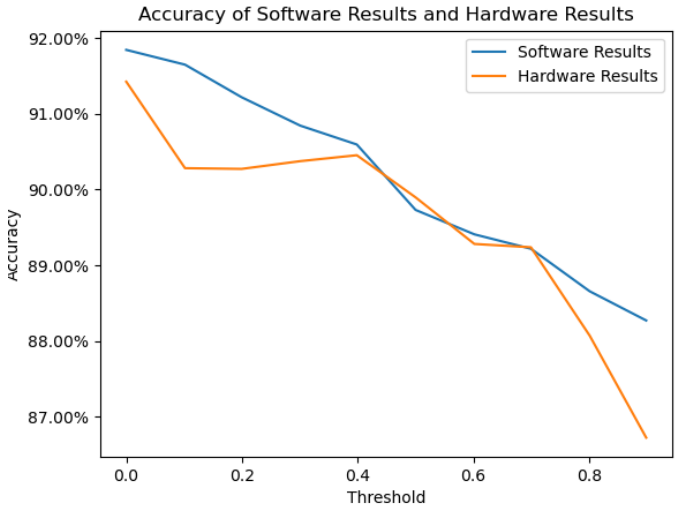


Figure 4.10: The threshold variation on accuracy of software design and hardware design

It's evident that, as the threshold uniformly varies from 0 to 0.9, the accuracy of the software design gradually decreases. However, although the overall accuracy of the hardware design is decreasing, the loss gap between hardware accuracy and software accuracy diminishes gradually before the threshold of 0.4. Between the thresholds 0.4 and 0.7, the accuracy of software design is almost the same as that of hardware design. Beyond the threshold of 0.7, the loss gap between hardware and software accuracy increases again. This indicates that between the threshold values of 0.4 and 0.7, the DeltaFC hardware accuracy demonstrates strong robustness.

Results of DeltaFC threshold variation on Throughput.
More comprehensively, increasing the threshold enhances temporal sparsity and accelerates the neural network's throughput.

First, for this work, the total operation load is composed of adding MAC operations at each layer. Each clock will perform 2 operations for MAC operation, one multiplication and one addition. Therefore, the operation load of each layer can be calculated as Eq. 4.3.

$$Operation\_Load\_of\_Each\_Layer = (Nodes\,A + 1) * Nodes\,B * FRAMES * 2 \tag{4.3}$$

Where $Nodes\,A$ represents the previous layer nodes, and the $Nodes\,B$ represents the next layer nodes. Therefore, after substituting the nodes of each layer of DeltaFC, 24, 96, 128, 64, 32, the total operation load can be calculated as Eq. 4.4.

$$\begin{aligned} Total\_Operation\_Load &= (24+1)*96*32*2 \\ &+ (96+1)*128*32*2 \\ &+ (128+1)*64*32*2 \\ &+ (64+1)*32*32*2 \\ &+ (32*32+1)*2 \\ &= 1611778 \end{aligned} \tag{4.4}$$

Next, this work uses an 8-channel parallel DSP architecture to perform MAC operations, and the operating frequency is 100MHz. Therefore, the theoretical maximum throughput of this work can be calculated as 1.6 Gop/s, as shown in Eq. 4.5.

$$Throughput = 100MHz * 8 * 2 = 1.6Gop/s \tag{4.5}$$

Finally, the throughput can also be calculated as Eq. 4.6. Therefore, obtaining the latency corresponding to each threshold can deduce the throughput corresponding to each threshold. Assuming that the threshold varies from 0 to 0.5. The temporal sparsity, accuracy, latency, and throughput corresponding to each software and hardware design threshold are shown in Tab. 4.6. And the curve corresponding to this Tab. 4.6 is shown in the Fig. 4.11.

$$Throughput = \frac{Operation\_Load}{Latency} \tag{4.6}$$

It can be seen that, except for the threshold of 0.5, the changing trends of throughput and temporal sparsity are the same. This not only ensures that the threshold can significantly increase the throughput of the hardware but also underscores a tight correlation between throughput and temporal sparsity. However, high temporal sparsity will lead to error throughput, which will be analyzed in the next section.

Results of DeltaFC acceleration
This work's hardware acceleration significantly improves computational efficiency. Here is a representative case. Assuming the threshold for this network is set to 0.01, the timing diagram for the DeltaFC IP without acceleration (corresponding to '$ACCELERATION\_MODE$' to 0 in Figure) is shown in Fig. 4.12.

As the Fig. 4.12 shows, the result is '$fffccb1a$'. The calculation time displayed in the Tcl console is 2133000 ns. After incorporating hardware acceleration (corresponding to '$ACCELERATION\_MODE$' to 1 in Figure), the timing diagram for DeltaFC IP is shown in Fig. 4.13.

	Software Design		Hardware Design			
Threshold	Temporal Sparsity	Accuracy	Throughput(Gop/s)	Efficiency	Latency(ms)	Accuracy
0	50.7%	91.84%	1.423	88.9%	1133	91.43%
0.004	54.0%	91.77%	1.560	97.5%	1033	91.34%
0.012	56.1%	91.81%	1.640	102.5%	983	91.42%
0.02	60.5%	91.85%	1.935	120.9%	833	91.37
0.028	59.1%	91.56%	1.825	114.1%	883	91.3%
0.036	61.9%	91.68%	2.058	128.6%	783	90.88%
0.1	73.2%	91.65%	3.024	189.0%	533	90.28%
0.2	84.5%	91.22%	4.842	302.6%	333	90.27%
0.3	89.5%	90.84%	6.912	432.0%	233	90.38%
0.5	94.1%	89.7%	0.452	28.3%	3567	89.8%

Table 4.6: The relationships between threshold and temporal sparsity, latency, and throughput. The lightweight DeltaFC architecture parameter scale used in this work is 27202, and the maximum theoretical throughput is 1.6Gop/s

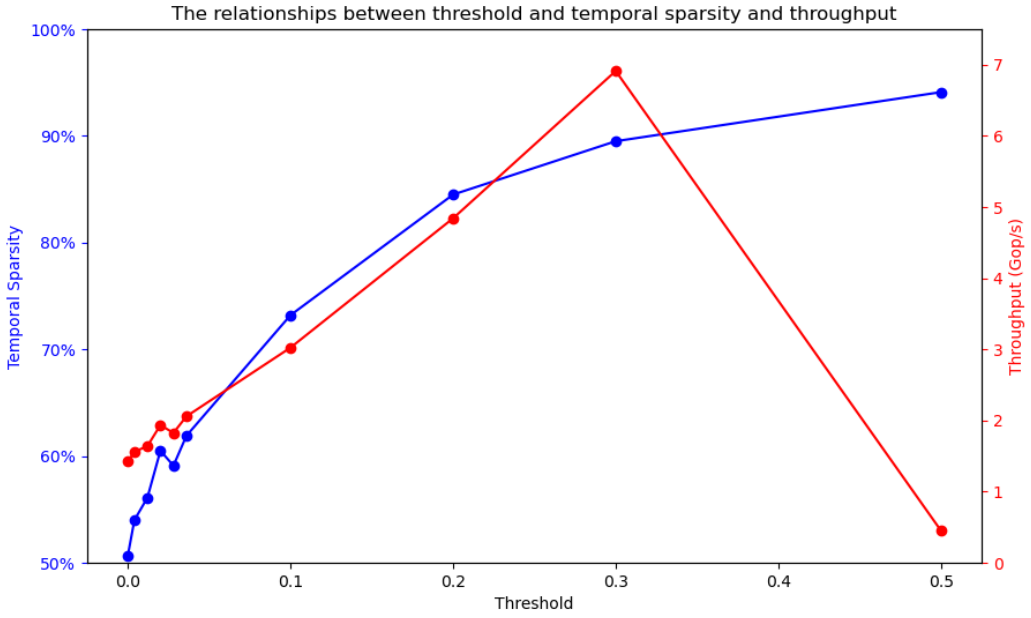


Figure 4.11: The relationships between threshold and temporal sparsity and throughput

As the Fig. 4.12 shows, the result is still '$fffccb1a$'. However, the calculation time displayed in the Tcl console is only 983000 ns. Moreover, the latency and throughput of DeltaFC before and after induced temporal sparsity acceleration are shown in Tab. 4.7. The results show that the computation time has reduced by 54%, leading to an 85% improvement in computational efficiency.

	Latency(ms)	Effective Throughput (GOp/s)
Without temporal sparsity	2.133	0.7556
With temporal sparsity	0.983	1.640

Table 4.7: Comparison of latency and effective throughput of DeltaFC before and after acceleration.

Acceleration, in the majority of cases, significantly enhances computational efficiency. This

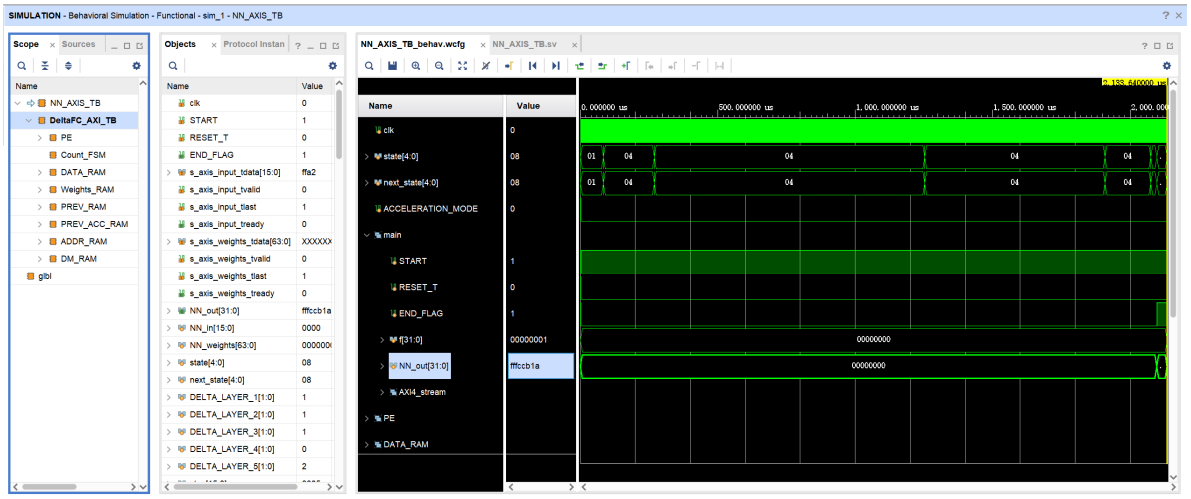


Figure 4.12: The timing diagram for the DeltaFC IP without acceleration

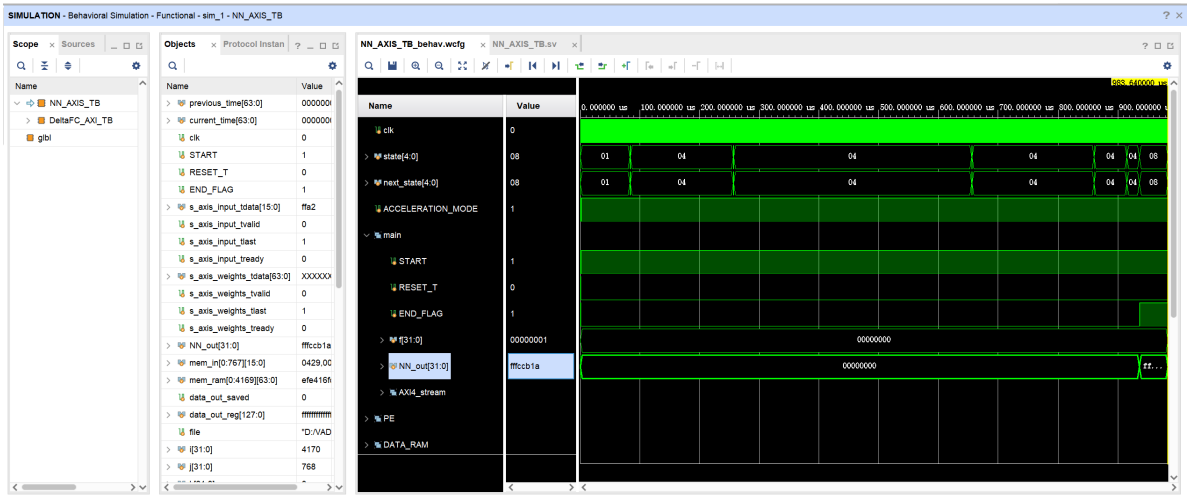


Figure 4.13: The timing diagram for the DeltaFC IP with acceleration

indicates that through the combined deployment of the Delta algorithm and CSR algorithm, sparse acceleration induced by the temporal sparsity of DeltaFC can be implemented. However, in exceptionally rare instances, acceleration can lead to a loss in accuracy. If the input features are expanded to the entire test dataset of 20,480 batches instead of a single batch, the accuracy results compared to the original non-accelerated results are shown in the Tab. 4.8:

	Accuracy
Without temporal sparsity	91.42%
With temporal sparsity	90.96%

Table 4.8: Comparison of the accelerated hardware accuracy with the non-accelerated hardware accuracy for the entire test dataset

As shown in the figure, this indicates that acceleration induced by temporal sparsity has a 0.5% probability of causing accuracy loss. Theoretically, acceleration only compresses the sparse matrix, and sparse matrices do not contribute to the results. The accuracy loss caused by acceleration will be discussed in the next section.

4.3. Experiment Analysis

The last section analyzes the experiment results from software design and hardware design. On the one hand, the software design subsection mainly analyses how the threshold affects temporal sparsity and accuracy. On the other hand, the hardware design subsection mainly analyzes the impact of thresholds on hardware acceleration.

4.3.1. Software Design Analysis

Impacts of threshold on temporal sparsity and accuracy

Threshold is used to filter low amplitude delta input features. This means that the threshold can increase the temporal sparsity by filtering more low features, but an excessively high threshold will reduce the accuracy of the neural network because of the loss of the key feature. So, it is very important to choose the appropriate threshold.

Fig. 4.14 shows a classic example of this work to show how threshold increases sparsity and decreases accuracy. This is the last frame's last 8 features of the 2048th batch in the first speech package after being transformed by the Delta algorithm, as shown in Fig. 4.14. When the threshold changes from 0.05 to 0.25, the temporal sparsity also increases from 12.5% to 100%. However, when the threshold is too high, such as reaching 0.25, all 8 features shown in Fig. 4.14 are filtered out, meaning all information is lost. These feature losses will cause accuracy loss in the final recognition result. Therefore, as shown in Fig. 4.2 and Fig. 4.3, temporal sparsity will gradually increase as the threshold increases, and accuracy will show a downward trend as the threshold increases.

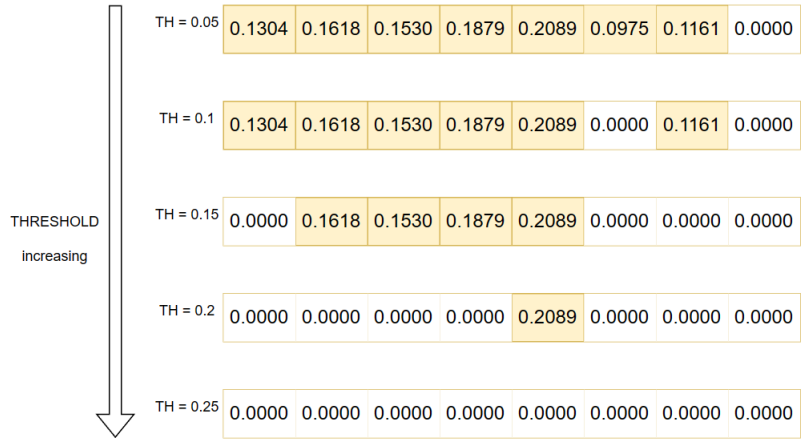


Figure 4.14: Temporal sparsity will increase as threshold increases

However, it is worth noting that, as shown in Fig. 4.2 and Fig. 4.3, a slightly increased threshold boosts a little accuracy. This is a very interesting phenomenon, which means that the threshold can help filter out some of the feature glitches that negatively affect the recognition results (similar to a high-pass filter).

The curve of temporal sparsity and threshold

Fig. 4.2 shows the resulting relationship curve between accuracy and threshold. It is a power function with an exponent between 0 and 1. The shape of such a curve is related to the distribution of input features.

Fig. 4.15 shows a classic example of this work to show why the temporal sparsity and threshold curve exhibit power function characteristics. This is the 2048th batch's features before the Delta algorithm in the first speech package, as shown in Fig. 4.15. It can be seen that this

curve has the characteristics of a Gaussian distribution. This is because the dataset has been normalized before input. Among them, the data included in the red circle are excluded outliers of input features, and the red line represents the fitting curve of the Gaussian distribution. From the perspective of input features, it generally has normal distribution characteristics.

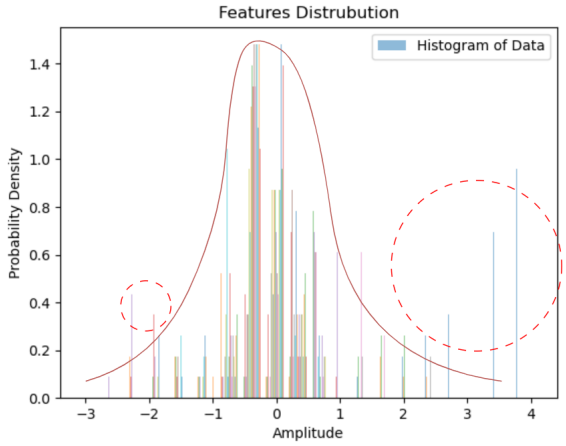


Figure 4.15: Data distribution of 2048th batch's features before Delta algorithm

This is the 2048th batch's features after the Delta algorithm in the first speech package, as shown in Fig. 4.16. It can be seen that after the Delta algorithm transformation, the absolute input features curve becomes half of the Gaussian distribution curve.

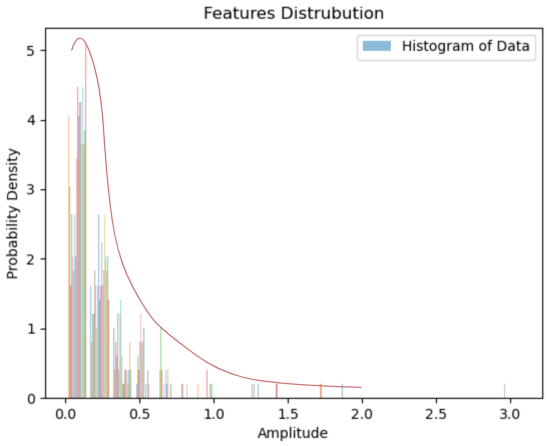


Figure 4.16: Data distribution of 2048th batch's features after Delta algorithm

The integration of the Gaussian distribution represents the probability that the data is distributed in a certain interval. If it is assumed that the threshold varies from 0 to a specific amplitude (assumed to be 0.5), then the area enclosed between 0 and 0.5 represents the distribution probability of data between 0 and 0.5, as shown in Fig. 4.17. According to the definition of temporal sparsity, as shown in Eq. 2.16, these features below the threshold (between 0 and the threshold) will be filtered to 0, so the data as shown in the Fig. 4.17 will eventually become sparse values. The probability of these filtered sparse values is exactly the value of temporal sparsity.

Therefore, the integral Gaussian distribution represented in Fig. 4.17 is the temporal sparsity, as shown in Eq. 4.7. The integration of the Gaussian distribution exhibits the characteristics of

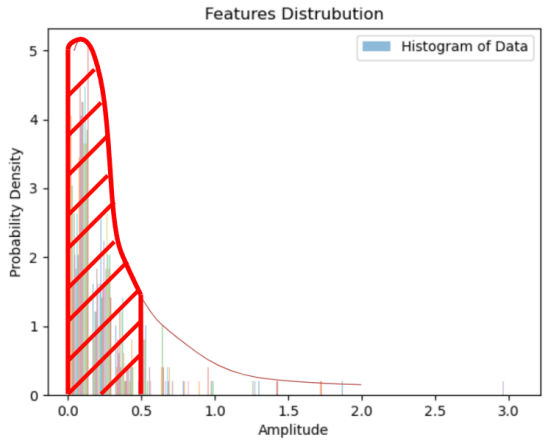


Figure 4.17: Integration of Features Distribution curve

a power function [27], so the curve of temporal sparsity also exhibits the characteristics of a power function.

$$\int_0^{Threshold} Distrubution\_Probability \ d(Amplitude) = Temporal\_Sparsity \tag{4.7}$$

4.3.2. Hardware Design Analysis

Impacts of Quantization On Accuracy

It can be seen from the results in the figure that the accuracy of hardware design is 0.5% averagely less than that of software design. This is mainly because the hardware design uses fixed-point quantization, which will cause a certain information loss after quantification.

It was mentioned in Chapter 2 that the input features of the hardware design have been 16-bit quantized; however, the weights of the hardware design model itself have only been 8-bit quantized. This is because DeltaFC, similar to other RNNs, also has the characteristics of weight sharing. In the iterative reasoning process, the weights already have certain regularization characteristics. Therefore, to save hardware resources, only 8-bit quantization is used. But precisely, only 8-bit quantization of weights means that the weights fitting is weakened, and the model may not be able to capture some subtle key features, thus affecting the accuracy of the neural network.

There are several ways to solve the decrease in accuracy caused by quantization.

- 1. 16-bit Fixed-point Quantization of Weights

  The high bit-width quantization of the model itself will lead to increased hardware resource consumption and computational complexity. This work's lightweight neural network implementation does not recommend higher precision quantification of weights.

- 2. Introducing Quantization-Aware Training Techniques

  Fixed-point quantization can be performed not at the end of the training process but between training processes, which is known as quantization-aware training. This allows the weights to fit the model as much as possible with a low-precision bit width after each epoch ends, thereby reducing the information loss caused by iterative quantization. This process is completed in software design.

Impacts of Threshold on Throughput

Thresholds can significantly influence the throughput, closely correlating to temporal sparsity. The throughput can be calculated as Eq. 4.8.

$$Throughput = \frac{Timing\_Cost * Baseline\_Throughput}{Latency} \tag{4.8}$$

The operation load refers to the total executed operations, which is the product of timing cost and a constant of baseline throughput. However, this IP will bypass all sparse value operations after hardware acceleration. This highlights that the latency associated with each voice segment exclusively involves non-sparse value operations. Therefore, the latency can be deduced as Eq. 4.9.

$$Latency = Timing\_Cost * (1 - Temporal\_Sparsity) \tag{4.9}$$

If substituting Eq. 4.9 into Eq. 4.8, then the throughput can be deduced as Eq. 4.10.

$$Throughput = \frac{Baseline\_Throughput}{1 - Temporal\_Sparsity} \tag{4.10}$$

Since temporal sparsity is always in the range of 0 to 1, throughput is positively related to temporal sparsity. Therefore, increasing the threshold not only results in heightened temporal sparsity but also contributes to an increased throughput. The observed trends in throughput and temporal sparsity exhibit complete alignment and consistency.

However, as shown in Fig. 4.11, when the sparsity is too high, it can lead to incorrect throughput. The reason for this problem is the memory bottleneck, which will be discussed in detail in the next section.

Impacts of Acceleration on Throughput and Accuracy

Fig. 4.11 shows that if the sparsity is too high, it can lead to crashed throughput. Fig. 4.10 shows that the accuracy of the results after hardware acceleration is slightly lower than the results before acceleration. But in fact, the induced temporal sparsity acceleration only skips sparse value operations, and theoretically, there should not be throughput loss and accuracy loss. The main reason for the throughput and accuracy loss is that a memory bottleneck of insufficient RAM ports can occur during speculation.

Here is an example of how memory bottlenecks occur. Assume a sparse matrix A has 12 features and 3 frames, as shown in Fig. 3.21. Then, assuming this matrix is multiplied by another matrix B, and the ratio of matrix B nodes to the number of parallel channels is only 1 instead of 2 mentioned in Chapter 3. In this way, the timing diagram of the speculation data transmission process is shown in Fig. 4.18. It can be seen that since the rule shown in Eq. 3.14 is not satisfied, the first frame calculation time is not enough to cover the speculation time.

On the one hand, for a wrong throughput issue, if high sparsity causes the calculation time to be lower than the speculation time since there is no enabled signal response, the hardware will always wait for the CSR indices speculated in the previous frame. This will lead to a transmission deadlock and an abnormal pipeline structure. This issue could have been avoided with more cache units. However, due to the architecture design of this work, there is no more memory to cache the CSR indices of the previous layer, and the deadlock caused by the memory bottleneck will continue. Therefore, the memory bottlenecks will occur if the threshold is high enough, such as beyond 0.5.

On the other hand, for accuracy loss issues, when the threshold is low, while the $CSRBuffer$ RAM reads the CSR indices of the second frame, the $CSRBuffer$ RAM input write port not only needs to write the unspeculative data of the second frame but also needs to write the

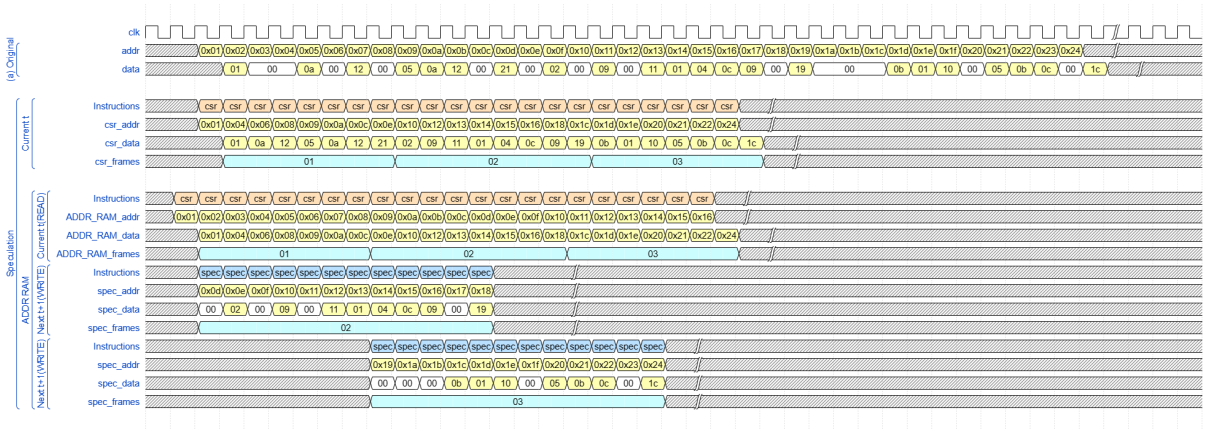


Figure 4.18: Timing diagram of insufficient RAM ports of bottleneck

speculative data of the third frame simultaneously. However, $CSRBuffer$ RAM is synthesized based on BRAM, which has only one write port, which leads to a memory bottleneck caused by insufficient memory ports. Therefore, due to the memory bottleneck, the input CSR indices are uncertain so accuracy loss will occur under the forward propagation of the hierarchical DNN. According to the results in Tab. 4.8, when the threshold is below 0.3, the probability of memory bottleneck occurring is approximately 0.5%.

The memory capacity can be increased to solve the memory bottleneck to avoid crashed throughput and accuracy loss. In other words, adding a parallel $CSRBuffer$ RAM can assist in data writing. Data can be written simultaneously in two parallel $CSRBuffer$ RAMs during the reading process. While writing data, one of the $CSRBuffer$ RAMs can write the CSR indices of the current frame, and simultaneously, the other $CSRBuffer$ RAM can write the CSR indices of the next frame. This avoids the memory bottlenecks. However, adding a parallel memory will double the $CSRBuffer$ RAM resources used for acceleration and double the power consumption. Although such overhead is manageable in this work, it is intolerable for non-lightweight models. For other large models, the memory bottleneck issue can be optimized through the following strategies:

- Optimizing memory access patterns
- Adopting more efficient data structures
- Using more efficient compression algorithms

5
Conclusions and Future works

5.1. Conclusion

After the experimental results and analysis in Chapter 4, the following conclusions can be drawn:

- **1. In software design, DeltaFC DNN can achieve 92.3% accuracy, 27202 parameters, and 56% temporal sparsity with a threshold of 0.01.**

  Both temporal sparsity and accuracy are related to thresholds. When the threshold continues to increase, the accuracy will decrease by about 0.5%, but the temporal sparsity will be improved significantly, which can accelerate the operation of the neural network by 85%.
- **2. In software design, DeltaFC DNN has higher accuracy, higher AUC, and lower parameters than baseline neural networks such as FC, LSTM, etc.**

  In more detail, DeltaFC DNN can achieve the highest accuracy (91.8%) with the smallest parameter (27202). This means that DeltaFC DNN is suitable for processing temporal information such as audio and can maintain the advantage of being lightweight. The software design results are as expected.
- **3. In hardware design, DeltaFC digital IP is deployed on FPGA RTL design with low resource utilization and successfully achieves high-frequency operation (100MHz)**

  DeltaFC digital IP implements the deployment of the Delta algorithm and CSR algorithm on FPGA through RTL design. Through reasonable logic design, the IP implements DeltaFC DNN itself and supports acceleration algorithms with extremely low resource utilization.
- **4. In hardware design, DeltaFC digital IP could implement all expected functions without functional verification and timing verification errors.**

  This digital IP can reproduce all results in the software design and has passed STA, pre-simulation post-simulation, etc. This proves that the hardware design can be used as an independent IP for block design in future work.

5.2. Future work

After the previous analysis, the following future work can be proposed for reference:

- **1. Block design and ARM core SDK development**

  This work is only the FPGA (PL) development of the Zynq board. But to truly realize VAD testing on board, SDK development (PS) for the ARM core of the Zynq board is also necessary.

- **2. In software design, training-aware quantization can be used to reduce accuracy loss.**

  Training-aware quantization can increase the fit of weights to the model. In addition, this method also involves the configuration of neural network training parameters in software design, such as step size, optimizer selection, etc.

- **3. In hardware design, the memory bottlenecks can be optimized.**

  Accuracy will be slightly reduced due to memory bottlenecks caused by hardware acceleration. Therefore, the memory access pattern of the acceleration unit and the memory architecture can be optimized to avoid memory bottlenecks as much as possible.

References

[1] Ossama Abdel-Hamid and Hui Jiang. “Rapid and Effective Speaker Adaptation of Convolutional Neural Network Based Models for Speech Recognition”. In: Aug. 2013. DOI: 10.21437/Interspeech.2013-336.

[2] Avnet. *MiniZed Hardware User Guide*. English. Version 2.0. Avnet. 40 pp. published.

[3] Urban Borštnik et al. “Sparse matrix multiplication: The distributed block-compressed sparse row library”. In: *Parallel Computing* 40.5 (2014), pp. 47–58. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2014.03.012. URL: https://www.sciencedirect.com/science/article/pii/S0167819114000428.

[4] Feifei Chen et al. “A 108-nW 0.8-mm2 Analog Voice Activity Detector Featuring a Time-Domain CNN With Sparsity-Aware Computation and Sparsified Quantization in 28-nm CMOS”. In: *IEEE Journal of Solid-State Circuits* 57.11 (2022), pp. 3288–3297. DOI: 10.1109/JSSC.2022.3191008.

[5] Huixiang Chen et al. “3D-based video recognition acceleration by leveraging temporal locality”. In: (June 2019), pp. 79–90. DOI: 10.1145/3307650.3322260.

[6] Lin Chen, Xiao Ma, and Xiang Ji. “FPGA-based LoongArch Five-stage Pipeline CPU”. In: *Journal of Physics: Conference Series* 2450 (Mar. 2023), p. 012058. DOI: 10.1088/1742-6596/2450/1/012058.

[7] Khalid A. Darabkh, Laila Haddad, and Saadeh Sweidan. “A Modified Speech Recognition Algorithm for People with Physical Disabilities”. In: *2017 European Conference on Electrical Engineering and Computer Science (EECS)*. 2017, pp. 23–27. DOI: 10.1109/EECS.2017.13.

[8] David Dean et al. “The QUT-NOISE-TIMIT corpus for evaluation of voice activity detection algorithms”. In: *Proceedings of the 11th Annual Conference of the International Speech Communication Association*. Ed. by K Hirose, S Nakamura, and T Kaboyashi. CD Rom: International Speech Communication Association, 2010, pp. 3110–3113. URL: https://eprints.qut.edu.au/38144/.

[9] Li Deng and John Platt. “Ensemble Deep Learning for Speech Recognition”. In: *Proc. Interspeech*. Sept. 2014. URL: https://www.microsoft.com/en-us/research/publication/ensemble-deep-learning-for-speech-recognition/.

[10] Suzanne MSc Ekelund. “ROC Curves—What are They and How are They Used?” In: *Point of Care: The Journal of Near-Patient Testing Technology* 11.1 (2012), pp. 16–21. DOI: 10.1097/POC.0b013e318246a642.

[11] C. Gao et al. “Edgedrnn: recurrent neural network accelerator for edge inference”. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 10 (4 2020), pp. 419–432. DOI: 10.1109/jetcas.2020.3040300.

[12] Google. *’WebRTC’*. 2020. URL: https://webrtc.org/.

[13] Benjamin Graham. “Spatially-sparse convolutional neural networks”. In: (Sept. 2014).

[14] Niklas Hansen and Simom Holst Albrechtsen. “Voice activity detection in noisy environments”. In: DEEP LEARNING, DTU COMPUTE, Dec. 2018, p. 02456. URL: https://github.com/nicklashansen/voice-activity-detection.

[15] Mark Horowitz. “1.1 Computing’s energy problem (and what we can do about it)”. In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.

[16] Itay Hubara et al. “Binarized Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by D. Lee et al. Vol. 29. Curran Associates, Inc., 2016. URL: https://proceedings.neurips.cc/paper/2016/file/d8330f857a17c53d217014ee776bfd50-Paper.pdf.

[17] “IEEE Standard for Floating-Point Arithmetic”. In: *IEEE Std 754-2008* (2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[18] Sofie Reyners Jan De Spiegeleer Dilip B. Madan and Wim Schoutens. “Machine learning for quantitative finance: fast derivative pricing, hedging and fitting”. In: *Quantitative Finance* 18.10 (2018), pp. 1635–1643. DOI: 10.1080/14697688.2018.1495335. eprint: https://doi.org/10.1080/14697688.2018.1495335. URL: https://doi.org/10.1080/14697688.2018.1495335.

[19] Ž Jovanović and V Milutinović. “FPGA accelerator for floating-point matrix multiplication”. In: *IET Computers & Digital Techniques* 6.4 (2012), pp. 249–256.

[20] Emre Karabulut and Aydin Aysu. “A Hardware-Software Co-Design for the Discrete Gaussian Sampling of FALCON Digital Signature”. In: (2023). https://eprint.iacr.org/2023/908. URL: https://eprint.iacr.org/2023/908.

[21] Zheng Lei et al. “Research on the Application of Integrated Storage and Computing Chip Based on Image Recognition”. In: *2023 IEEE International Conference on Sensors, Electronics and Computer Engineering (ICSECE)*. 2023, pp. 605–609. DOI: 10.1109/ICSECE58870.2023.10263456.

[22] Nan Li et al. “Robust Voice Activity Detection Using a Masked Auditory Encoder Based Convolutional Neural Network”. In: *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2021, pp. 6828–6832. DOI: 10.1109/ICASSP39728.2021.9415045.

[23] “Librispeech: An ASR corpus based on public domain audio books”. In: IEEE, 2015, pp. 5206–5210. DOI: 10.1109/ICASSP.2015.7178964.

[24] Bo Liu et al. “An energy-efficient voice activity detector using deep neural networks and approximate computing”. In: *Microelectronics Journal* 87 (2019), pp. 12–21. ISSN: 0026-2692. DOI: https://doi.org/10.1016/j.mejo.2019.03.009. URL: https://www.sciencedirect.com/science/article/pii/S0026269218307043.

[25] Bo Liu et al. “EERA-ASR: An Energy-Efficient Reconfigurable Architecture for Automatic Speech Recognition With Hybrid DNN and Approximate Computing”. In: *IEEE Access* 6 (2018), pp. 52227–52237. URL: https://api.semanticscholar.org/CorpusID:52966121.

[26] Jing Liu et al. “Discrimination-Aware Network Pruning for Deep Model Compression”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.8 (2022), pp. 4035–4051. DOI: 10.1109/TPAMI.2021.3066410.

[27] Dmitri Martila and Stefan Groote. “Evaluation of the Gauss Integral”. In: *Stats* 5.2 (2022), pp. 538–545. ISSN: 2571-905X. DOI: 10.3390/stats5020032. URL: https://www.mdpi.com/2571-905X/5/2/32.

[28] Sameeraj Meduri and Rufus Ananth. *A Survey and Evaluation of Voice Activity Detection Algorithms: Speech Processing Module*. Koln, DEU: LAP Lambert Academic Publishing, 2012. ISBN: 3659172049.

[29] Markus Nagel et al. “A White Paper on Neural Network Quantization”. In: (June 2021).

[30] Sarang Narkhede. “Understanding AUC - ROC Curve”. In: *Towards Data Science* (2018). URL: https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5.

[31] Daniel Neil et al. “Delta Networks for Optimized Recurrent Network Computation”. In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, June 2017, pp. 2584–2593. URL: https://proceedings.mlr.press/v70/neil17a.html.

[32] Jongsoo Park et al. “Faster CNNs with Direct Sparse Convolutions and Guided Pruning”. In: *arXiv e-prints*, arXiv:1608.01409 (Aug. 2016), arXiv:1608.01409. DOI: 10.48550/arXiv.1608.01409. arXiv: 1608.01409 [cs.CV].

[33] Michael Price, James Glass, and Anantha P. Chandrakasan. “A Low-Power Speech Recognizer and Voice Activity Detector Using Deep Neural Networks”. In: *IEEE Journal of Solid-State Circuits* 53.1 (2018), pp. 66–75. DOI: 10.1109/JSSC.2017.2752838.

[34] Philippe Renevey and Andrzej Drygajlo. “Entropy based voice activity detection in very noisy conditions”. In: Sept. 2001, pp. 1887–1890. DOI: 10.21437/Eurospeech.2001-446.

[35] Abhipray Sahoo. “Voice activity detection for low-resource settings -”. In: *CS230 deep learning* (). URL: http://cs230.stanford.edu/projects_winter_2020/reports/32224732.pdf.

[36] Robin Schmidt. “Recurrent Neural Networks (RNNs): A gentle Introduction and Overview”. In: (Nov. 2019).

[37] Jongseo Sohn, Nam Soo Kim, and Wonyong Sung. “A statistical model-based voice activity detection”. In: *IEEE Signal Processing Letters* 6.1 (1999), pp. 1–3. DOI: 10.1109/97.736233.

[38] AXI4 stream. “AMBA Axi4 interface protocol”. In: *AMD* (). URL: https://www.xilinx.com/products/intellectual-property/axi.html.

[39] Wenyu Tang. “Automatic Functional Datapath Optimization”. In: (2014).

[40] Sibo Tong, Hao Gu, and Kai Yu. “A comparative study of robustness of deep learning approaches for VAD”. In: *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2016, pp. 5695–5699. DOI: 10.1109/ICASSP.2016.7472768.

[41] Fengbin Tu et al. “Deep Convolutional Neural Network Architecture With Reconfigurable Computation Patterns”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.8 (2017), pp. 2220–2233. DOI: 10.1109/TVLSI.2017.2688340.

[42] Preetha Vijayan. and T.G.R.M. van Leuken. “Temporal Delta Layer: Training Towards Brain Inspired Temporal Sparsity”. In: *Delft University of Technology* (2021). DOI: https://repository.tudelft.nl/islandora/object/uuid:0806241d-9037-4094-a197-6e65d6482f2b.

[43] Hongzhi Wang, Yuchao Xu, and Meijing Li. “Study on the MFCC similarity-based voice activity detection algorithm”. In: *2011 2nd International Conference on Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC)*. 2011, pp. 4391–4394. DOI: 10.1109/AIMSEC.2011.6009945.

[44] Tianjiao Xu et al. “Improve Data Utilization with Two-stage Learning in CNN-LSTM-based Voice Activity Detection”. In: *2019 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*. 2019, pp. 1185–1189. DOI: 10.1109/APSIPAASC47483.2019.9023306.

[45] Amirreza Yousefzadeh and Manolis Sifalakis. “Training for temporal sparsity in deep neural networks, application in video processing”. In: ArXiv, July 2021. URL: /abs/2107.07305..

[46] Thein Htay Zaw and Nu War. “The combination of spectral entropy, zero crossing rate, short time energy and linear prediction error for voice activity detection”. In: *2017 20th International Conference of Computer and Information Technology (ICCIT)*. 2017, pp. 1–5. DOI: 10.1109/ICCITECHN.2017.8281794.

[47] Jinghua Zhang, Chen Li, and Marcin Grzegorzek. “Applications of Artificial Neural Networks in Microorganism Image Analysis: A Comprehensive Review from Conventional Multilayer Perceptron to Popular Convolutional Neural Network and Potential Visual Transformer”. In: (July 2021).

[48] Jinming Zou, Yi Han, and Sung-Sau So. “Overview of Artificial Neural Networks”. In: *Artificial Neural Networks: Methods and Applications*. Ed. by David J. Livingstone. Totowa, NJ: Humana Press, 2009, pp. 14–22. ISBN: 978-1-60327-101-1. DOI: 10.1007/978-1-60327-101-1_2. URL: https://doi.org/10.1007/978-1-60327-101-1_2.