KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Design, analysis and implementation of a low-power convolutional engine for speech recognition system

Huizhao Wang

Thesis submitted for the degree of
Master of Science in
Electrical Engineering, option
Electronics and Integrated Circuits

Thesis supervisor:
Prof.Dr.Ir. Marian Verhelst

Assessors:
Prof.Dr.Ir. Wim Dehaene
Prof.Dr.Ir. Luc Van Gool

Mentors:
Ir. Bert Moons
Ir. Bert De Brabandere

Academic year 2015 – 2016

Preface

I would like to thank everybody who let me learnt a lot the last year. Especially my promotor who gave me the chance to write this thesis and my supervisor who come up with such a good topic and helps me a lot while completing this thesis. I would also thank the jury for reading the text. My sincere gratitude also goes to my friends and my family.

Huizhao Wang

Contents

Abstract

Convolutional Neural Networks(CNN) combined with Hidden Markov Models(HMM) are widely used in today's Automatic Speech Recognition(ASR) system. But CNNs used in ASRs are normally with large-scale. To accelerate CNNs in ASRs, we introduce a low-power solution: using low bit width fixed point computation instead of floating point computation. Software simulation results using KALDI toolkit and TIMIT database shows that CNNs with bit width of 6 is enough to get accuracy very close to floating point results, which are 20.7% and 23.0% for two different CNN. Based on this results, we implement an convolutional engine, an ASIC implementation, that has a bit width of 6. This convolutional engine uses NanGate 45nm cell library, and works at 250MHz with supply voltage of 1.1V. It uses 32 MAC units working in parallel, and on-chip SRAMS to store part of parameters and a frame of inputs. This low-power engine can work at a very low power of 0.12W and 0.14W, and energy efficiency of 125GOPS/W and 110.5GOPs/W respectively.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

CNN	Convolutional Neural Network
ANN	Artificial Neural Network
DNN	Deep Neural Network
HMM	Hidden Markov Model
ASR	Automatic Speech Recognition
GMM	Gaussian Mixture Models
MFCC	Mel Frequency Cepstral Coefficients
FC	Fully Connected layers
FPGA	Field Programmable Gate Arrays
ASIC	Application Specific Integrated Circuits
LUT	Look Up Tables
PWL	Piece-wise Linear
TIMIT	Texas Instrument and Massachusetts Institute of Technology
SVM	Support Vector Machine

Chapter 1

Introduction

Machine learning is widely used to solve problems that are increasingly complex nowadays. One of the complex tasks, the state-of-art Automatic Speech Recognition(ASR), has combined neural networks with Hidden Markov Models(HMM).

Artificial Neural Network(ANN), known as one of traditional useful method in machine learning region, has taken place of Gaussian Mixture models(GMM) in GMM/HMM frame work. And it has been demonstrated great efficient in solving ASR problems. This will be introduced in detail in chapter 2.

In recent years, this hybrid ANN/HMM frame work has more interests in using more efficient types of neural network architecture. The Deep Neural Networks(DNN) combined with HMM[14] as a first try, showed big improvement for acoustic modeling. But unfortunately this frame work usually has large scale of parameters thus need long training time.

However in 2014, Ossama Abdel-Hamid[3] proposed that using full connected layers at the first several layers is simple but not efficient. If we use convolutional layer, which shares weights across nearby frequencies, instead of first several fully connected layers in DNNs, the results are demonstrated better. As a result, CNN-based systems with Mel Frequency Cepstral Coefficients(MFCC) features as input and with pooling layers in between adjacent convolutional layers emerged in 2014. The CNN/HMM[2][25][3] model shows further reduction of phone error rate during large vocabulary speech recognition tasks.

Even though CNN has shared parameters compared to DNN, it's implementation is not easy when considering of its efficiency and speed. Implementation of CNNs falls into two ways: Software implementation and hardware implementation. Software implementations offer good flexibility. However hardware implementations[22] can make full use of CNN's advantage of parallelism. Among these implementations, Specific-purpose (image classification, speech recognition, etc.) hardware implementations has provided high speed and compactness.

Currently, these workloads are mostly executed on GPUs[7], or on FPGAs[5]. Others have proposed ASIC implementations[22][6] with high computational ability but somehow ignores power consumption or memory transfers. What's more, most of the works focused on vision system. Making use of these results, our thesis aims at designing low-power hardware accelerated CNNs for real time speech recognition system. However, as accuracy is a primary requirement, there is a trade-off between accuracy and energy consumption.

1.1 Contributions and reasons

The main contributions of this thesis are:

- To simulate a low-precision CNNs in Kaldi.

  One way to implement low-power hardware is to use low-precision hardware. M. Courbariaux[8] has already showed that very low precision multipliers are sufficient for training DNN. As accuracy is a primary requirement, this part is to simulate low-precision CNN architecture in Kaldi toolkit, and to see whether CNN-based speech recognition can still produce good results even when using low-precision CNN. Further more, to determine which bit width to use in this fixed-point CNN.

  In this part, the low-precision CNN architecture is evaluated by a speaker independent speech recognition task using of the standard TIMIT data sets[10].

- To implement a low-precision hardware of convolutional engine, which is core part of a CNN.

  After chosen the optimum bit width, we implement a convolutional engine using Synopsis Design Compiler and synthesized it. The library is NanGate freedpk45 Open Cell Library.

- To make assumptions of energy based on above implementations.

  Energy assumptions are made using Synopsis Design Vision based on both operations and architecture. And results are evaluated at the end. A convolutional engine is optimized in terms of parallelism when considering energy consumption.

1.2 Thesis outline

This thesis include 5 chapters:

Chapter 2 gives theoretical backgrounds, including ANN, CNN and introduction of CNN/HMM speech recognition system.

Chapter 3 gives details of software simulations and results.

Chapter 4 gives a simple energy assumption model and an implementation of synapse used in CNNs. What's more, gives an trends of optimization.

Chapter 5 gives a brief conclusion.

Chapter 2

Theory and Backgrounds

This chapter provides background on Convolution Neural Networks (CNNs) and their performance in speech recognition field. Then briefly describe how CNN train and decode audio signals. Existing hardware implementation of CNN are summarized. This chapter also introduced low-precision arithmetic, which is used in low-power computing.

2.1 CNN/HMM based ASR System

Speech recognition, which is also called Automatic Speech Recognition (ASR), is to accurately and efficiently convert a piece of speech signal into a text message or phoneme sequence.

Speech recognition has a very long history. As early as 1932, researchers at Bell Labs have investigated the science of speech perception.[30] Since then, speech recognition today has become more complex and accuracy.

A speech recognizer system today consists of three part: acoustic model, the lexicon and language model. The basic block diagram of ASR is shown in Figure 2.1. An acoustic model is to present relationship between an audio signal and the phonemes. A Language model is a probability model probability distribution over sequences of words, aims at match phonemes with word sequences, which is not covered in this work. For example, a given sequence is assigned a probability $P(w_1, w_2, \ldots, w_n)$ to the whole length. It is used to mimic the probability to predict which words will occur next.

For this task, we focused on phoneme recognition, so that acoustic model is more important and will be discussed in detail. As shown in Figure 2.1, the raw audio signal from each records can be transformed by applying the mel-frequency cepstrum, which as a typical method of feature extraction, result as a set of mel frequency cepstral coefficients (MFCC). These set of features , sometimes along with other features, are used as an input to the acoustic model. Hidden Markov models (HMMs) as a typical acoustic model are used for at least 2 decades. Recently CNNs has made big improvements. Some proposal claimed that CNNs can acheive relatively 10%[3]

reduced error rate. So that in our work, we use CNN/HMM hybrid frame work as acoustic model. This process will be discussed in detail as follow.

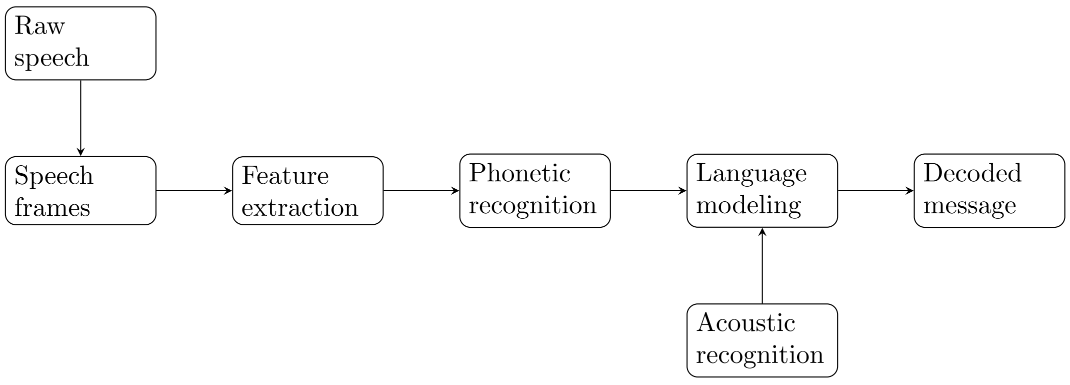


Figure 2.1: Block diagram of speech recognition.

2.1.1 Phoneme

Phoneme is the smallest basic unit of a speech. When doing the words recognition, phoneme recognition may first comes out. Even though speakers or words may different form each other, a same phoneme must have the same acoustic characteristics. Based on this, we can use speech recognition system to convert pieces of sounds to phoneme text.

Basic phoneme utterances can be sorted into several groups, including utterance vowel, diphthong, nasal, stop consonant and unvoiced classes. A given language will use only a small subset of the many possible sounds that the human speech organs can produce. Normally, the English language uses a set of 39 phonemes.

2.1.2 Feature extraction

Raw speech is generally sampled at a high frequency, most commonly 16kHz or 8kHz. This produces a large number of data points and normally stored as specific format.

Feature extraction, as the first step of speech recognition, aims to converting an initial set of raw measured data to features that contains more non-redundant informations. And this features has better computer interpretations. There are many methods for feature extraction: Mel frequency cepstral coefficients (MFCC), Linear Predictive Coding (LPC) and Short-Time Fourier Transform (STFT). In which MFCC is the most widely used because of its adaption for human hearing and its steps are summarized in Figure 2.2. MFCC coefficients[18] are generated as follows:

1) For MFCC method, a speech signal may first be divided into frames. Each frames has a length of 10 to 25ms. Adjacent frames may have a certain length overlap. This step usually apply a Hamming window function at fixed intervals to remove edge effects.

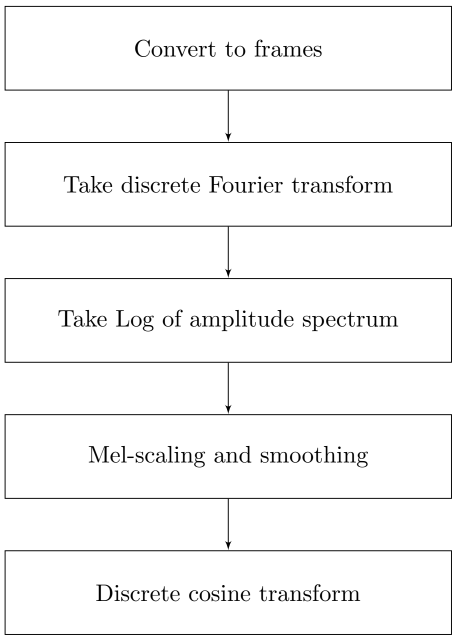

Convert to frames
Take discrete Fourier transform
Take Log of amplitude spectrum
Mel-scaling and smoothing
Discrete cosine transform

Figure 2.2: Process to create Mel features.

2) After frames are got, we need to know the energy distribution on each frequency interval. The Discrete Fourier Transform is taken in which inputs are signal sequence and outputs are a sequence of complex numbers that presents magnitude and phase spectrum of signals. An alternative way is STFT.

3) Only the amplitude information are put into the logarithm because perceptual studies shows that magnitude of spectrum is much more important than phase.

4) Next step is to smooth the spectrum and emphasize perceptually meaningful frequencies. This step can reduce and smooth the spectral components. For example, reduce 256 spectral component into 40. For speech, it has been found that lower frequencies components contains more important informations than higher frequencies components. We can use Mel scales to divide frequency axis into variable length frequency bins. For instance, we convert this bins to Hz to determine start and end point of frequency bins. As a consequence, using the Mel Scale, the lower frequency coefficients become more important. Mel scale smoothing as shown in figure 2.6 makes use of triangle filter banks to compute energy at each bins.

5) The last step of MFCC feature extraction is using the Discrete Cosine Transform (DCT) to obtain cepsral features. This step decorrelates the Mel-spectral coefficients.

6) For some systems, pitch features can add to the MFCC features. Considering the correlations between frames, we need to compute first-order time derivative and

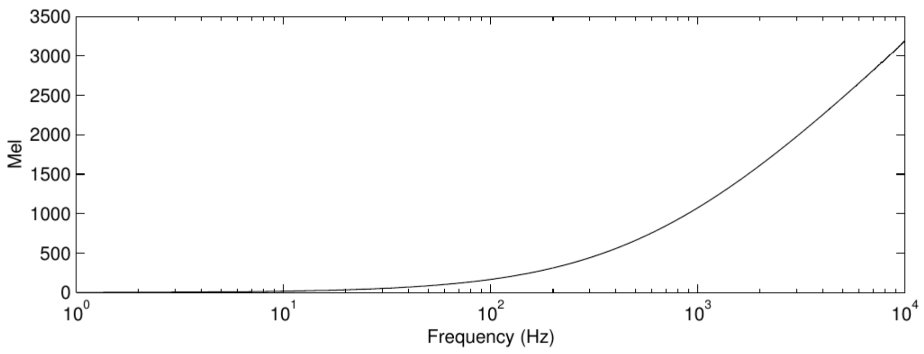


Figure 2.3: Mel scale.

sometimes second-order time derivative, and attaching these derivatives on to our general feature set. They are also known as delta and double-delta components.

2.1.3 Convolutional Neural Networks

Convolutional neural networks are neural networks with a specific structure. To describe CNNs, Artificial Neural Networks(ANNs) may first be introduced. The current section describes how Convolution Neural Networks develops from Artificial Neural Networks.

Artificial Neural Networks

Neural networks are biologically inspired classifiers. They are widely used for many machine learning applications due to their ability to learn non-linear functions.

1) Architecture

A complete neural network consists of a set of nodes, connections, a feed-forward computing process and back-forward training procedure. An artificial neuron is a basic computing unit similar to neurons in our brain. Figure 2.4 shows the architecture of a single artificial neuron. A neuron consists of a synapse node and an activation part.

A neuron takes $x\{1, \dots, n\}$ as input, and its output can be calculated by the function $h = f(x, W; b)$, in which f is activation function. The input vectors are first calculated by function:

$$y = \sum_{i=0}^{i=n} x_i \times w_i + b \tag{2.1}$$

In which, w_i is called weights and b is bias. y is also called the output of a synapse. Then the results y goes into activation node.

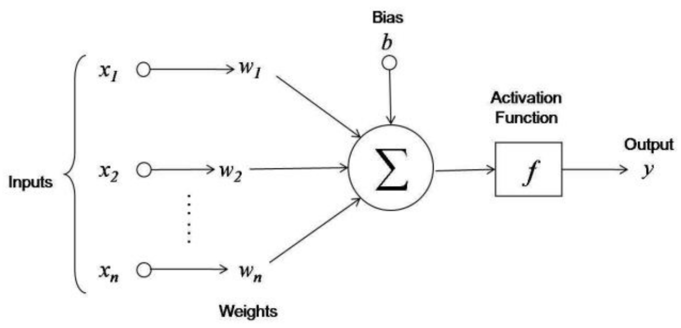


Figure 2.4: A single neuron

Common choices for activation functions are sigmoid, hyperbolic tangent or ReLU. The weights $w\{1, \dots, n\}$ and the bias b, are parameters which is necessary to find the optimum values. We use the sigmoid function:

$$F(z) = \frac{1}{1 + e^{-y}} \tag{2.2}$$

Normally the output of a neuron is directly connected to the input of the neuron in the next layer, which result as a multilayer neural network as shown in Figure 2.5. All the units operate simultaneously, allowing for algorithms to be implemented efficiently on specific purpose hardware which supports parallel processing. With a large amounts of labeled training data, a neural network can train its weights by propagating the error backwards and update its weights every iteration till the result doesn't change much.

2) Training Algorithm

Gradient Descent is a classical optimization algorithm. Given a set of parameters W, we seek to optimize a loss function,$E(W)$. Here W presents the parameters which is necessary to be optimized. Gradient descent algorithm works by taking a step in the negative direction at a certain position. In other words, we can optimize the W that leads to minimized $E(W)$ by computing $\frac{\partial E(W)}{\partial W}$, and then updating the W vector to

$$W(t+1) = W(t) + \mu \frac{\partial E(W(t))}{\partial W(t)} \tag{2.3}$$

Where μ is the learning rate, and t donates the iteration index.

In order to implement this procedure through multiple layer neural network, we need to approximate the gradient of the error with respect to all the weights vectors. It is not easy to do when the error respect to weights connect to hidden neurons instead of connecting directly to output layer. Now we need a back-propagation algorithm that takes advantage of chain rule:

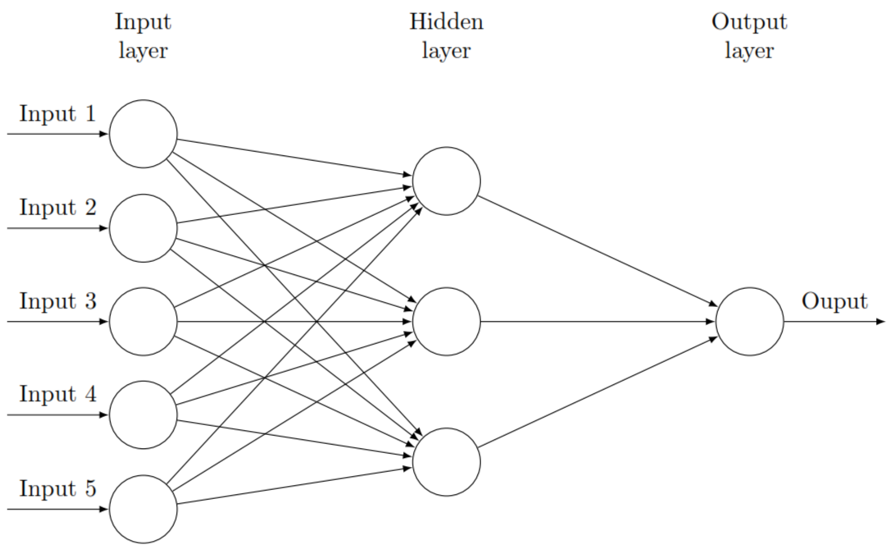


Figure 2.5: A multilayer neuron network

$$\frac{\partial E}{\partial W_n} = \frac{\partial F_n}{\partial W}(X_n, W_{n-1})\frac{\partial E}{\partial X_n} \tag{2.4}$$

$$\frac{\partial E}{\partial X_{n-1}} = \frac{\partial F_n}{\partial X}(X_n, W_{n-1})\frac{\partial E}{\partial X_n} \tag{2.5}$$

where X_n is the output for layer n. W_n is the set of weights used in layer n. $F_n(X_n, W_{n-1})$ is applied to use X_{n-1} to produce X_n.

The back-propagation algorithm can be implemented as shown in Algorithm 1.

Once the gradient is computed with back propagation, the gradient are used to perform a parameter update. There are several approaches for performing the update. One way is called batch training, which compute the error on the entire set of training examples, taking the average and then update the W. Batch training is widely used in Time-delay Neural Networks (TDNN)[29]. An alternative approach is called Stochastic Gradient Descent (SGD)[16], which is popularized by LeCun. Since evaluating the sum of gradients is very expensive, SGD is usually much faster than batch training and often results in better solutions. SGD introduces a great deal of noise in to training by using a subset of entire examples as an estimate for the overall error. It can prevent falling into local minimums and makes the system converge much faster. Depending on this big advantage, This is very effective in large-scale machine learning task and becomes much more popular.

Moreover, to minimize the error of all inputs, the back propagation is repeated several times. We also need to use an annealing algorithm to reduce learning

Algorithm 1 Back-propagation algorithm

```
 1: procedure Back-propagation
 2:     Divide the training data set into evenly-sized patterns.
 3:     for each input patern x do
 4:         Forward propagation
 5:         compute error for each output neuron
 6:         for $i = K$ to 1 do
 7:             // K is the index for hidden layers
 8:             Back propagate error on each hidden layer
 9:         end for
10:              Update weights for output neurons
11:         for $i = K$ to 1 do
12:             Update weights for each hidden layer
13:         end for
14:     end for
15: end procedure
```

rate when there is little progress. Because with a high learning rate, the system usually unable to settle down into deeper, but narrower parts of the loss function. However, with a low learning rate, it usually takes too much time to convergence and sometimes may miss the best position.

Convolutional Neural Networks

CNN is a type of feed-forward multilayer artificial neural network. It is widely used in classification tasks in which the goal is to sort one object into a known class. The input is a feature vector or matrix that describes some observation. For example, in image prediction task based on CIFAR-10 dataset, we build the CNN as a model in which the inputs are pixel arrays, and the outputs are probabilities for each classes.

A CNN consists of one or more convolution layers and then followed by several full connected layers. Here is an example of CNN for image recognition shown in Figure 2.6. The inputs of a CNN are usually designed as a 3D structure. To transform this input volume, we use the following four types of layers to build a typical CNN architecture.

1. Architecture

    a) Convolutional Layer

    The convolutional layer is the core building block of CNNs. A convolutional layer arrange its neurons in three dimensions (width, height and depth). Each neuron in convolutional layer focus on a very small local area of input volume. Using a shared weights between each neurons in the same slice can dramatically decrease the amounts of required weights compared to full connected layers. In other words, the convolutional layer is making use of a small filter that convolves across the width and height of the input volume to reduce number of parameters. By using of a group of filters, the convolutional layer extends its depth. Obviously, each filter represents a certain feature for the classification task. So that all these neurons can learn to activate for different features.

b) Pooling Layer

Between two convolutional layers, there is typically a pooling layer. Pooling layer is a form of non-linear down-sampling layer and can reduce computing complexity. There are several types of pooling method, the most common used is max-pooing. For example, the max pooling layer has a filter size of 2*2 and generates the maximum one on that location as the output. The function of this layer is progressively reduce the spatial size of previous layer without losing important feature information. It is built based on the assumption that once a feature has been found, its rough location relative to other features played important role instead of its exact location.

c) Fully Connected Layer

After several alternated convolutional layers and pooling layers, several fully connected layers occurs at the end of the entire network. Neurons in full connected layer have fully connections to all activations in previous layer. This layer can convert the activation volume into vectorised output. Particularly, it is the last layer in the decoding process.

d) Loss Layer

A loss layer, which is normally the last layer of a CNN, is to produce the deviation between the predicted and true labels. Various loss functions can be used here, in which the softmax loss is the most popular one. Since the softmax function normalizes outputs to the range [0,1] and ensures that it sums to 1, the outputs can be treated as posterior probabilities. For each output k, the score function is defined using softmax function as:

$$P(Y = k|X = x_i) = \frac{e^{h_k}}{\sum_j e^{h_j}} \tag{2.6}$$

where $h = f(x_i, W; b)$.

The loss function is nomally the negative log likelihood of the correct class:

$$L_i = -logP(Y = y_i|X = x_i) = -log(\frac{e^{h_{y_i}}}{\sum_j e^{h_j}}) \tag{2.7}$$

where y_i is the correct class.

2. Training algorithm

A CNN is trained using the back-propagation algorithm already discussed in the Artificial neural networks. All the parameters of all the filters in all layers are updated simultaneously by the training procedure. Using SGD, we train the

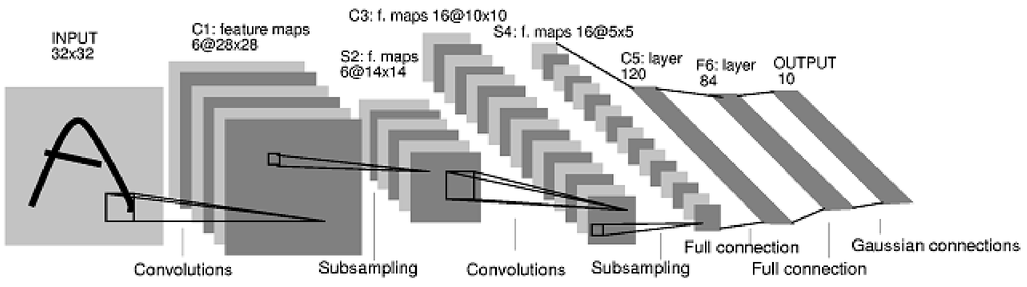


Figure 2.6: Architecture of LeNet-5[16].

CNNs to minimize a loss function. Possible loss functions include cross-entropy and mean squared error. Updating the shared weights of convolution kernels can be done by treat the shared weights as independent, the loss for each kernel can then be summed up and averaged to update the shared weights. Specifically the CNN has pooling layer, we need to updates the weights accordingly as shown in algorithm 2.

Algorithm 2 CNN training algorithm

```
    procedure CNN BACK-PROPAGATION
 2:     Divide the training data set into evenly-sized patterns.
        for each input pattern x do
 4:         Forward propagation
            compute error for each output neuron
 6:         for $i = K$ to 1 do
                Back-propagate the error for each layer $i$
 8:         end for
                Update weights for output neurons
10:         for $i = K$ to 1 do
                if layer i is not a sub-sampling layer then
12:                 Update weights for layer i
                end if
14:             set the derivatives as 0 to non-max source formax-pooling layer
                update threshold for other type of sub-sampling layer
16:         end for
        end for
18: end procedure
```

2.1.4 Hidden Markov Model

Hidden Markov models (HMMs) is a statistical Markov model with hidden states is widely used in many systems due to its ability to model the temporal transitions between states. It is well known that HMMs have been successful in temporal pattern recognition of speech signals. For a simple Markov model, the states are directly visible, the state transition probabilities are known for observer. For HMMs, each of these hidden states is associated with a probability distribution of observations.

To understand Markov Model, consider that a system in which we take all the previous observations into consideration is obviously impractical. For many applications, the most recent observations are much more important than the observations at very early time. To present this, we use a model with edges representing transition probabilities between states.

For basic Markov model, the states are known for observer, which is not enough to be applicable to many problems since we often dont know which state we are. Hidden markov models solves these problems. In HMMs, there exists transition probabilities and emission probabilities. A diagram of an HMM is shown in Figure 2.7. Where, the O_t is observed feature generated form probabilities $b_j(O_t)$, which is the emission probabilities, and also the transition probabilities denoted by a_{ij}.

Mathematically an HMM model, $\gamma = (A, B, \pi)$. Where A is the set of all a_{ij}, the transition probabilities from state i to j. B is the set of all $b_j(O_t)$. π is the initial state distribution vector, in which π_i present the probability that initial state is i. For a given γ model, a sequence of utterance can be generated.

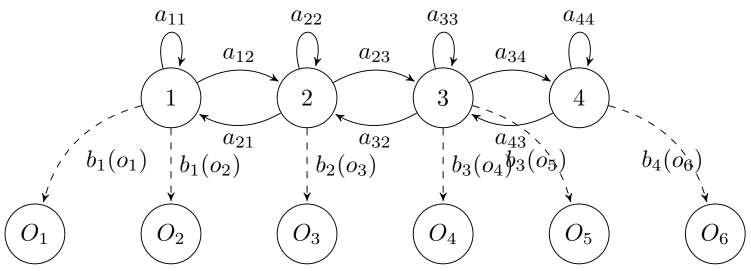


Figure 2.7: Architecture of Hidden Markov Models.

HMM is convenient to construct from smaller determined models. For instance, we can construct a phoneme model for a mono-phone, then concatenate mono-phone models to form a large vocabulary model. The hierarchial models of them are shown in Figure 2.8.

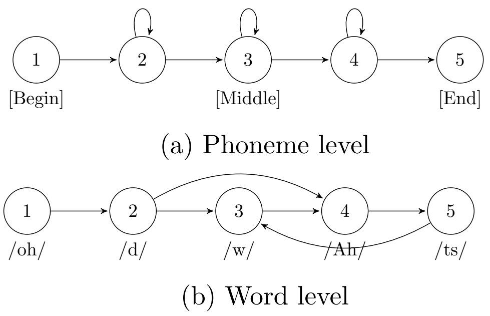


Figure 2.8: Hierarchial models of HMMs.

For a given sequence of observations, and a given Hidden Markov model γ, we want to generate a probability of most likely sequence of states:

$$S = argmaxP(s_t|O) \tag{2.8}$$

which can not be generated directly. However, Bayes rules helps a lot:

$$P(s_i|O) = \frac{P(O|s_i)P(s_i)}{P(O)} \tag{2.9}$$

Here, we convert problem to maximize the $P(O|s_i)$. HMMs assume the observations are independent, thus we can say the probability of the observation given the model and the sequence s is related to the individual observations at time t by:

$$P(O|s,\gamma) = \prod_{T}^{t=1} P(O_t|s_t,\gamma) \tag{2.10}$$

Summing up all the possible sequence of s, we can obtain $P(O|\gamma)$. Obviously, there is still many difficulties on calculating the $P(O|\gamma)$ directly. Fortunately, there is a recursive algorithm for the so-called forward procedure.

$$P(O|\gamma) = \sum_{N}^{i=1} \alpha_t(i) \tag{2.11}$$

in which, $\alpha_t(i) = P(O_1, O_2, \ldots, O_t, s_t = i|\gamma)$, which gives the probability of sequence of observations $O_1, O_2, \ldots, O_t$ and ending up in state i at time t. Thus we can obtain the value $P(O|\gamma)$. Another important algorithm used in decoding the continuous speech task is Viterbi algorithm. The Viterbi algorithm attempts to find the best sequence of states with a given model leads to best describe the observation sequence. In addition, We can use the Baum-Welch algorithm[4]to train HMMs, which will not introduce any detail here.

2.1.5 CNN/HMM hibrid model

As Early as 1990s, the Artificial neural networks has been used in ASR combined with HMMs. But constrained by poor large parameters training ability at that time, the hybrid model didn't perform well.

Over last few years, the advantages of DNNs and new learning methods has been taken. In the hybrid context-dependent DNN/HMM framework, as shown in Figure 2.9, DNN replaced the Gaussian Mixture Models (GMMs) to compute HMM state-posterior probabilities. The difference is that DNN use large dimention of outputs layer to accommodate large number tied-states of tri-phone HMMs. The HMM is supposed to handle the speech temporal variability.

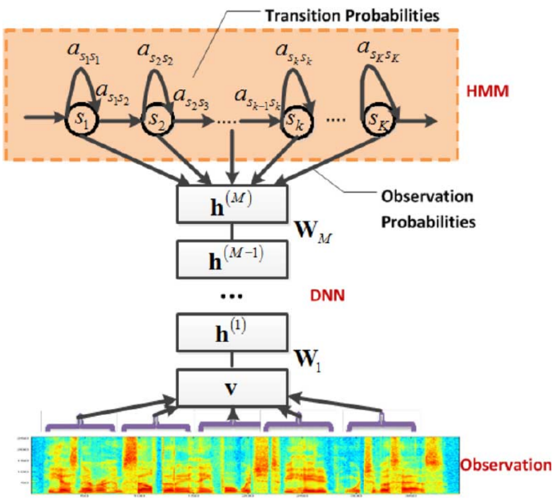


Figure 2.9: Diagram of DNN/HMM.[9]

More recently, Abdel-Hamid et al[3] showed that CNN can futher improve the hybrid model performance on TIMIT[10] phoneme recognition task. In CNN/HMM hybrid framework,as shown in Figure 2.10, convolutional layers of 1D CNN are applied to frequency axis. 2-D CNNs are the ones whose features not only presented by frequency but also by time(derivatives). 2-D CNNs are widely used in image recognition system now. The inputs of CNN are consecutive feature frames with a center frame at time t. Moreover, some systems use generative model named Restricted Boltzmann Machines RBM for unsupervised coarse pre-training[15], which can reduce the required labeled data for fine supervised training.

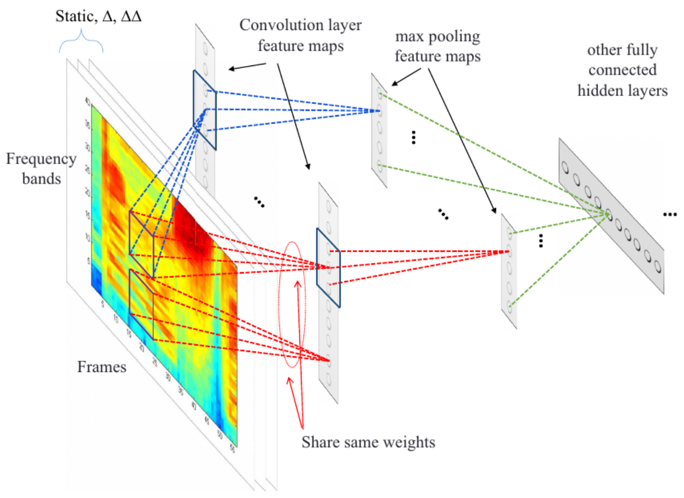


Figure 2.10: A CNN applied in ASR task.[1]

2.2 Embedded Energy Efficient CNNs

Hardware implementations of CNNs can be evaluated by several dependent specifications: Throughput, speed, power and footprint. These performances are results from where the designs put their emphasis on.

2.2.1 Existing Embedded CNNs

As indicated above, there are several existing hardware implementations for feed-forwarding or training CNNs at high speed and high efficiency. For example, Diannao[6], which is proposed by ICT, China, can achieve 452 GOPS in a small footprint of 3.02 mm^2 at 485 mW. And Neuflow[22], which is a SoC design proposed by IBM achieve 160 GOPS, at 570 mW with footprint of 12.5 mm^2 to accelerate neural networks. Some optimized FPGA implementation of CNN[31] can achieve 61.62 GFLOPS under 100MHz working frequency.

To implement low-power hardware, the easiest way is using low-precision arithmetic. However, these designs are either for bit width of 16 or for floating-point numbers. In this thesis, bit-width can go to 6 based on software simulation without loosing too much accuracy. So the power can be much lower than existing implementations.

2.2.2 Low-precision method

Different number format indicates different hardware implementations, which may result in different performance. Here is an FPGA example that use same bit-width but different number format will cause different power consumption[13], shown in Table 2.1.

Table 2.1: A comparison of multiplication unit with different number format.

	32 bit Fixed-point	32 bit Floating-point
Power(mW)	136.3	164.7
Frequency	100MHz	100MHz

Floating-point numbers is in most cases so advantageous over fixed-point number representation based on its very small resolution. But Fixed-point numbers are widely used in hardware implementation because of its simpler implementation and better power efficiency, like DSPs.

Floating-point format

Floating-point representation normally used for approximation of the real value. It allows a large dynamic range to be represented. The term floating-point refers to the fact that a number's radix point can float. A floating-point is represented as:

$$x = (-1)^s \times 1 \times m \times 2^{e-b} \tag{2.12}$$

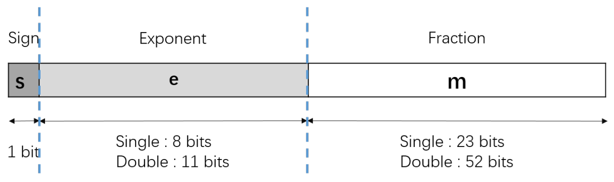
where, s represents the sign bit, m is a fraction number in $(1, 2)$. And its IEEE 754 Format (single precision) is shown in Figure 2.11.

Fixed-point format

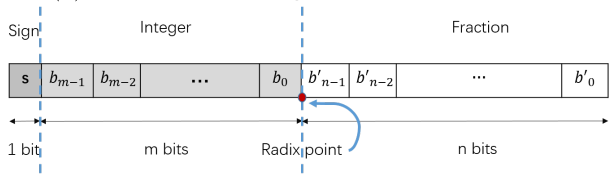
A fixed-point representation represents the value in binary. Fixed point formats consist of a signed bit and a known place of decimal point, which can mimic a shared factor among all numbers. There are a lot of format to represent fixed-point numbers. For this work, we choose $Q_{m,n}$ format which m stand for the number of bits for integer part and n stand for bits for fraction part. For a $Q_{m,n}$ number, the total bits is $m + n + 1$, including the signed bit. A fixed-point is represented as:

$$\begin{aligned} x = (-2^m) \times s + b_{m-1} \times 2^{m-1} + b_{m-2} \times 2^{m-2} + \cdots + b_0 \times 2^0 \\ +b'_{n-1} \times 2^{-1} + b'_{n-2} \times 2^{-2} + \cdots + b'_0 \times 2^{-n} \end{aligned} \tag{2.13}$$

And its format is shown in Figure 2.11.



(a) Format of floating-point number



(b) Format of fixed-point number

Figure 2.11: Format of floating-point and fixed-point numbers.

Converting Floating-point to Fixed-point

Converting algorithm also called quantization. This method intrinsically lowed the precision of a number. The method is shown as Algorithm 3.

Algorithm 3 Converting algorithm

```
    procedure Converting Floating-point to Fixed-point
        Given the total number of bits N = m+n+1, and range R.
3:      Calculate the m.
        for each input floating x do
            Calculate $y = x * 2^n$
6:          if x exceed the range R then set it to R.
            end if
            Round the value of y.
9:          Convert the y from decimal to integer representation.
        end for
    end procedure
```

2.3 Summary

An entire CNN/HMM hybrid ASR system consists of a feature extraction module, a CNN network and a HMM module. Among this CNN module consumes the largest part of computing energy because of its complexity.

To design an energy efficient ASR decoding system, an practical way is to use a hardware implemented CNN in feed-forward process. To further reduce power, fixed point format arithmetic take place of floating point arithmetic in CNN module.

Chapter 3

Software Simulation of Low-precision CNNs

This chapter describes two CNNs and their environment used in KALDI ASR toolkit[23]. And gives the performances of ASRs when realizing fixed point arithmetic on these CNNs.

3.1 Toolkit and Corpus

In this thesis, we used the CNN prototypes implemented by Karel Vesely, also named nnet1 framework, in KALDI toolkit. Kaldi consists of a C^{++} based library and training scripts for acoustic models. The KALDI toolkit architecture is shown in Figure 3.1. Because of its various executable tools, it has many alternatives for different types fo speech recognition tasks. For our experiment, we treat the KALDI nnet1[28] as the state-of-art baseline.

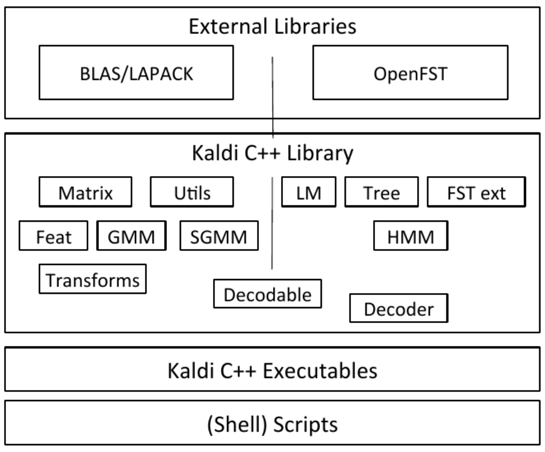


Figure 3.1: KALDI toolkit architecture

The TIMIT (Texas Instruments (TI) and Massachusetts Institute of Technology (MIT)) acoustic-phonetic continuous speech corpus, is a database of phonetically and lexically transcribed speech from American English speakers of different sexes and dialects. It contains a total of 6300 sentences from 630 independent speakers from the United States. All sentences were segmented at the phone level and were sampled at frequency of 16KHz.

TIMIT is a widely used database for phone segmentation, phone classification and phone recognition tasks. For phone recognition tasks,the Phone Error Rate (PER) stays roughly 21%, which is seemed as the baseline in our work.

For KALDI TIMIT recipe, training is done on 48 phone classes and in scoring process we map them to 39 phonemes.

3.2 Feature extraction and transformation

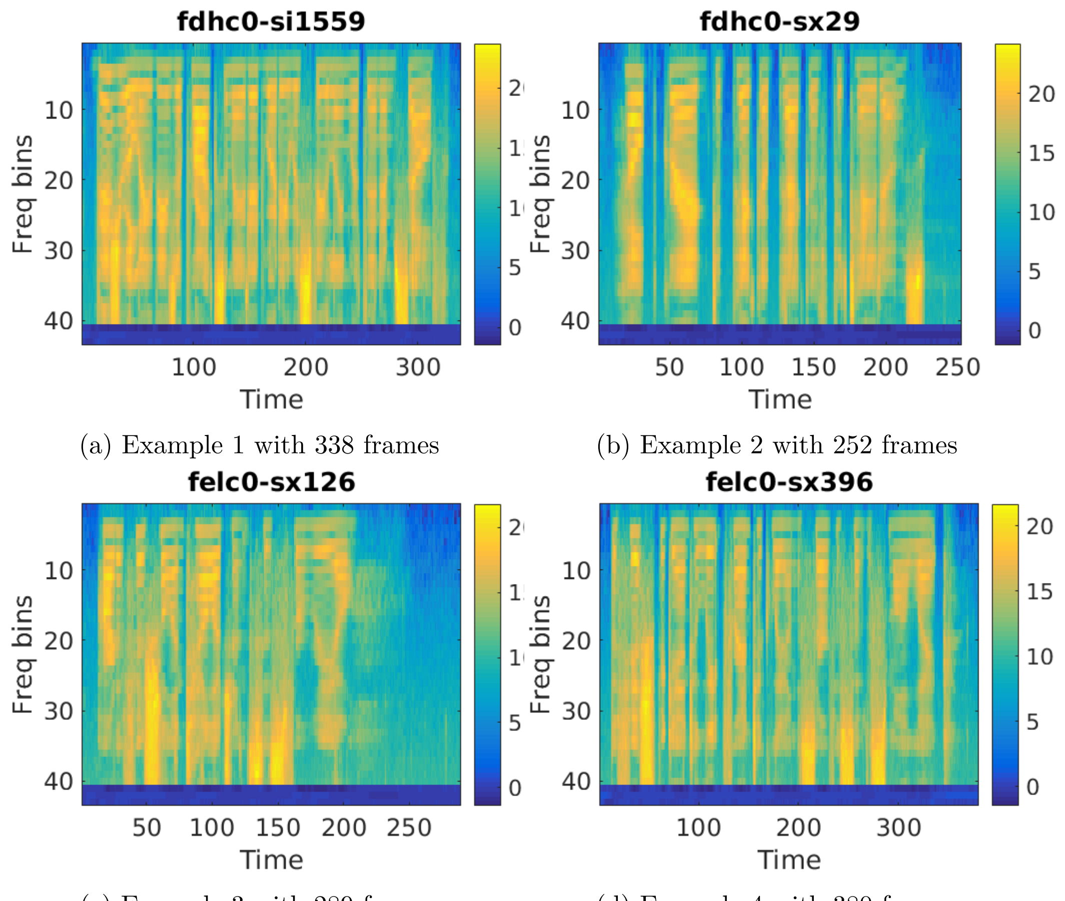
KALDI provides verious ways to transform speech signals from raw measured data to informative features. This process is so called feature extraction.

MFCC and Perceptual Linear Predictive (PLP) features are most often used features in ASR systems. Both these features are derived from Mel-scaled log filter bank features. For example, a 13-dimensional MFCC features may extracted from a 24-dimensional fbank features, using a DCT module. Although both MFCC and PLP features are more invariant than filter bank (fbank) features, some useful information may loss during the transformation process. Some systems use the fbank features directly in CNN/HMM hybrid systems.

In our experiment, we use the default setup of feature extraction. First, make the fbank features (40 mel-bins) together with 3-dimensional KALDI pitch features[11]. Pitch features in KALDI consists of Probability of Voicing (POV) feature, pitch feature and delta pitch feature. These pitch features are from a modified RAPT algorithm[27], which is provided by KALDI and will not be discussed here. The 3-dimensional pitch features are used for better performance on both tonal and non-tonal language recognitions.

Second, delta and double-delta (d + dd) features are calculated to capture further time dynamic information. This step ends up a 129 dimensional features for each frame.

Figure 3.2 shows several feature extraction examples for 4 independent utterance. The utterance index are : fdhc0-si1559, fdhc0-sx29, felc0-sx126 and felc0-sx396. And their frame numbers are 338, 252, 289 and 380 respectively. They are the outputs of feature extraction process without delta and delta-delta features. The Y-axes stand for frequency bins and pitch features, X-axes stand for index of frames along time. The brighter the pixel, the larger the values at that point, which stand for higher energy at that frequency bin.



(a) Example 1 with 338 frames (b) Example 2 with 252 frames

(c) Example 3 with 289 frames (d) Example 4 with 380 frames

Figure 3.2: Examples of feature extraction

Before the input features goes into CNN, there is normally an feature transformation process, which includes splicing current frame with previous and next frames, add bias vectors and rescale to normalize input features. Because Speech signals are always continuous signals, it is obviously reasonable to take previous frames and next frames both into account when processing.

The default transformation step in KALDI is splicing previous 5 frames and next 5 frames together with current frame. As a result of splicing, each frame end up with a 129 $\times$ 11, which is also 1419 dimensional vectorized features. Assume that each piece of utterance consists of n frames. It ends up with a 1419 $\times$ n dimensional feature matrix. Each column of the input matrix stand for one frame.

3.3 CNN implementation

Tranformed input features need to be feed into the CNNs. There is two types of CNNs already implemented in KALDI TIMIT recipe. In our work, both the two CNNs are used in simulations for both accuracy and power efficiency consideration.

3.3.1 1-D CNN implementation

For the 1-D CNN implementation, an optimized architecture proposed by Tara N. Sainath[25] was used, shown in Figure 3.3. However, as the 9 pitch features, including delta and double-delta components, are added to each single frame before splicing. After splicing, each frame has 99 pitch features.

In nne1 framework, these pitch features are handled by parallel components which are different from convolutional layers.These parallel components are parts of neural networks above the red dash line in Figure 3.3. Different from convolutional layers, they deal with the 99 pitch features, and consists of two fully connected layers. Convolutional layers handle the rest 1320 features for each input frame.

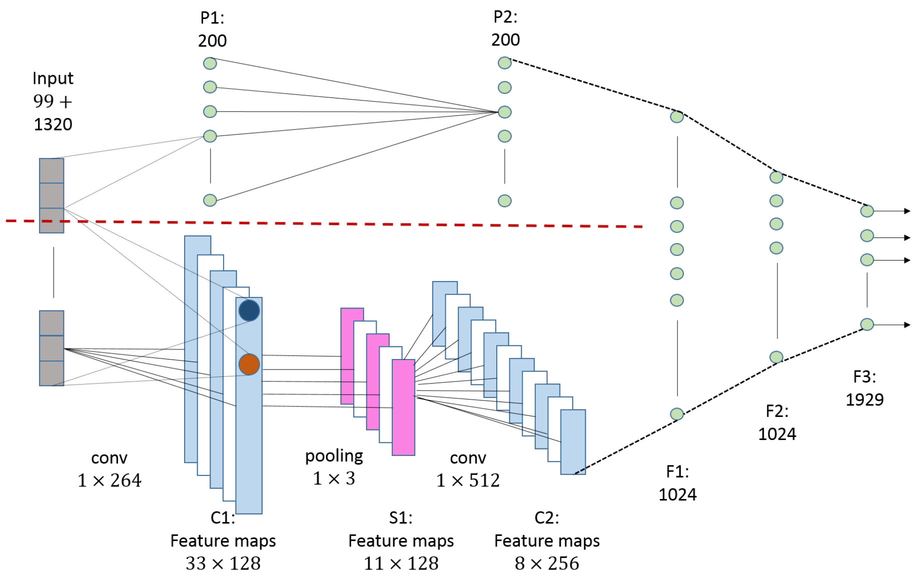


Figure 3.3: 1-D CNN architecture

For the main path which used convolutional layers, the input has a dimension of 1320 and is presented as a vector, first feeds into a convolution layer. The first convolutional layer has 128 convolution filters. Each filter of it has a dimension of 264, which means one filter can mimic a synapse as described in chapter 2:

$$y = \sum_{i=1}^{264} x_i w_i \tag{3.1}$$

To perform 1-D CNN, input vector are divided into 33 patches and each patch has a size of 40. The convolutional kernel put 8 features of each patch simultaneously and combine them as input features, so that input dimension is 264 and fits the convolution filter.If the input features are re-organized as Figure 3.4, in which each row stand for a patch of 40 features.At one time, 264 features go into a synapse. Step here is 1. Each convolutional kernel with same weight matrix generate outputs node on same splice in feature maps. Outputs generated by different kernel lies on different slice.

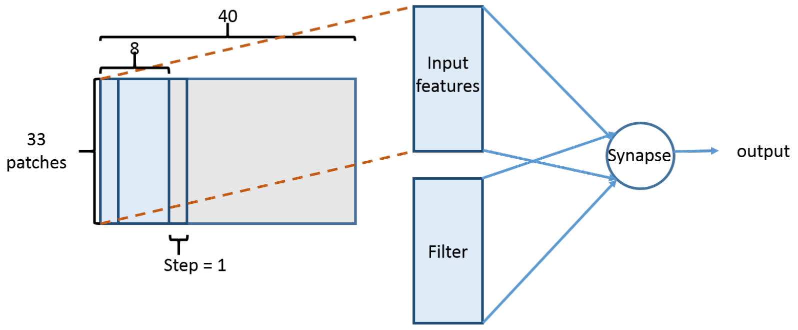


Figure 3.4: A typical architecture of first convolutional layer in 1-D CNN

The pooling layer use the simplest max-pooling method. This process is shown in Figure 3.5. The pooling size is 3, and step is 3. Each successive 3 inputs compared with each other, then output the maximum.

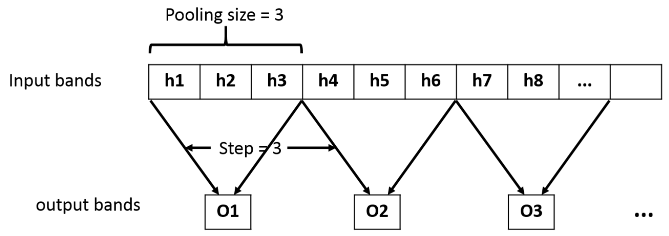


Figure 3.5: A typical architecture of max-pooling layer in 1-D CNN

Before the next convolutional layer, there are rescale, addshift and sigmoid layers that are not shown in the architecture. Addshift layer is to add bias to the whole inputs to make the mean of inputs to be zero. Rescale layer is to make each input multiply a predefined weights to make the variance of inputs to be one. This two steps always come together for normalization.

Sigmoid layer, as already explained in chapter 2, used as activation layer. The dimension of these three layers are the same with outputs of their previous layers, and they never change the dimension during the propagate.

The second convolutional layer has 256 filters, with each filter has a size of 512. Patch size is also 512 and patch step is 128. The output size of second convolutional layer is 2048. Succesively comes another rescale, addshift and sigmoid layers.

The parallel path used 2-layer fully connected neuron network, each fully connected layer has a size of 200.

Before the 3 last fully connected layer, the outputs from two path are combined. The size of them are 1024, 1024 and 1929 respectively. In our feed-forward neural network, the last layer is fully connected layer instead of softmax layer in feed-forward-feedback networks.

The filter size, input dimension and output dimension of each layer are summarized in Table 3.1, in which *P* stands for parallel component and FC stands for Fully Connected layers.

Table 3.1: list of layers in 1-D CNN (P stands for parallel component)

Component	Filter Size	Input Dim	Output Dim
1 Convolution	264	1320	33×128
2 Max-pooling	3	4224	1408
3 Rescale	-	1408	1408
4 Addshift	-	1408	1408
5 Sigmoid	-	1408	1408
6 Convolution	512	1408	8×256
7 Rescale	-	2048	2048
8 Addshift	-	2048	2048
9 Sigmoid	-	2048	2048
(P1) FC	99	99	200
(P2) Sigmoid	-	200	200
(P3) FC	200	200	200
(P4) Sigmoid	-	200	200
10 FC	2248	2248	1024
11 Sigmoid	-	1024	1024
12 FC	1024	1024	1024
13 Sigmoid	-	1024	1024
14 FC	1024	1024	1929

3.3.2 2-D CNN implementation

The 2-D CNN implementation is similar to 1-D CNN implementation. We also use the implementation of nne1. In order to compare the performance with 1D CNN implementation, we use the same setup for feature extraction and the same set of test and training data. That's results for the same dimension of input features used in both 1-D and 2-D CNNs.

2-D CNN also use two convolutional layers with a max-pooling layer in between and followed by 3 fully connected layer. Due to nne1's setup, we don't use parallel components for pitch features this time. So the input features for first convolutional layer has a dimension of 1419 each frame. The architecture is shown in Figure 3.6.

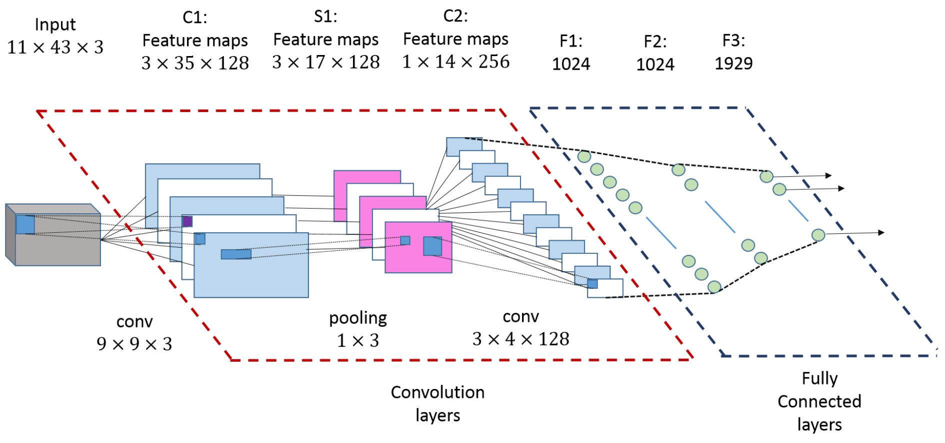


Figure 3.6: 2-D CNN architecture

Input features are first arranged as a cube, also called feature maps, shown in Figure 3.7. The depth of the cube is 3, which represents fbank features, delta features and double delta features.

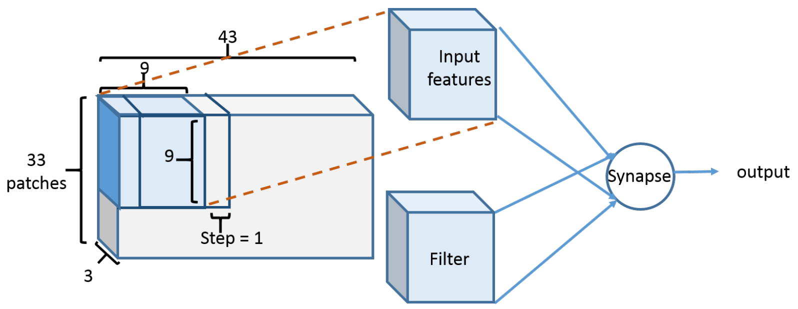


Figure 3.7: A typical architecture of convolutional in 2D CNN

In the first convolutional layer, a cubic convolutional kernel of size $9 \times 9 \times 3$ looks into each position on input feature maps. Each weight matrix for a convolutional kernel is called a filter here. As same as 1-D CNN architecture, 128 filters for first convolutional layer are used. Step for each filter along the x-axes and y-axes of input feature maps are both 1.

The size of the pool is 1×3. It is more complex then 1-D CNN, because it use different steps along x-axes and y-axes, which are 2 and 1 respectively,as shown in Figure 3.8. The output of pooling layer is a $3 \times 17 \times 128 = 1408$ dimensional array.

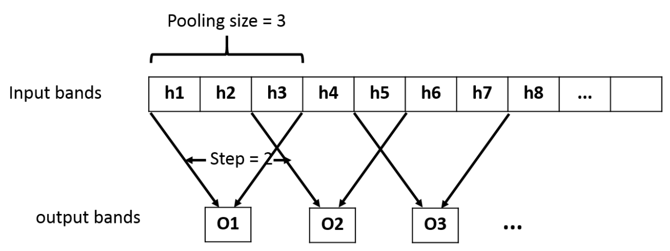


Figure 3.8: A typical architecture of max-pooling layer in 2D CNN

In the second convolutional layer, there are 256 different filters and each filter with a size of $3 \times 4 \times 128$. The output has a dimension of $1 \times 14 \times 256 = 3584$.

Then comes 3 fully connected layers. The final layer has the same output dimension as 1-D CNN, which is 1929 output nodes. The output dimension is the same with 1-D CNN.

The input dimension, filter size and output dimension for each layer used in 2-D CNN are shown in Table 3.2.

Table 3.2: Layers for 2-D CNN

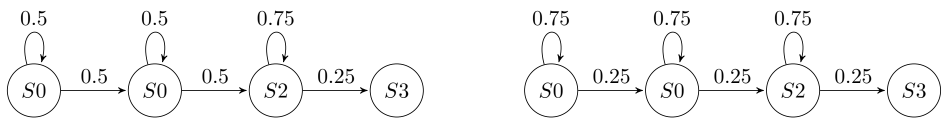
Component	Filter Size	Input Dim	Output Dim
1 Convolution	$9 \times 9 \times 3$	$11 \times 43 \times 3$	$3 \times 35 \times 128$
2 Max-pooling	1×3	$3 \times 35 \times 128$	$3 \times 17 \times 128$
3 Rescale	-	6528	6528
4 Addshift	-	6528	6528
5 Sigmoid	-	6528	6528
6 Convolution	$3 \times 4 \times 128$	$3 \times 17 \times 128$	14×256
7 Rescale	-	3584	3584
8 Addshift	-	3584	3584
9 Sigmoid	-	3584	3584
10 FC	3584	3584	1024
11 Sigmoid	-	1024	1024
12 FC	1024	1024	1024
13 Sigmoid	-	1024	1024
14 FC	1024	1024	1929

3.4 Hidden Markov Model

In KALDI, we use 3-state Bakis model HMM topology. Each phone out of the 48 context-independent phones consists of 4 states: 3 emitting state and 1 non-emitting state. This topology is a left-right HMM. Each of the 3 emitting state has transition to the left state and to themselves. The non-emitting state is the end state which can not transition to any other states. Figure 3.9 shows the initial probabilities for each HMMs, whose transition probabilities will be changed during training process. In the initial state, probabilities for phone 2 to phone 48 are the same.

KALDI does not train the transition probabilities of the non-emitting probability and instead it uses the probabilities given in the HmmTopology object. This property simplifies our training mechanism.

After that, we use 1929 pdf-ids which relates to the 1929 outputs node of CNN to tie the 48 context-independent phones. There is also an decision tree that for each HMM-state of each monophone, it automatic ask questions about any phone and HMM states in the context window. This decision tree is automatically generated and won't be discussed in our work.



(a) initial state for phone 1. (b) initial state for phone 2-48.

Figure 3.9: HMM topology for phones.

In the CNN/HMM arcitecture, each output of the CNN output layer represent the posterior probability for each state in HMM. Combining these with the trained transition probabilities, we can use Viterbi algorithm to decode out a sequence of phones based on input frames.

3.5 Simulation Results

We use 10% of the corpus as test data set, rest of the corpus as train data set. 10% of train data used as cross validation set.

As accuracy is a very important specification used for evaluating the performance of an ASR system. Our low-power CNN need to take accuracy into consideration. These software simulations are based on both floating point implementations and fixed point implementations. The results are used for analysis of relationship between accuracy and bit width.

3.5.1 Floating point experimental results

As already said above, both 1-D CNN and 2-D CNN architectures use the same train and test data sets. And their input and output dimensions are the same. The Phone Error Rate(PER) for 2 implementations are shown in Table 3.3. PERs for 1-D CNN implementation and 2-D CNN implementation are 20.7% and 23.0% respectively.

For comparison, Table 3.3 also shows some results of other published results. This results are all based on TIMIT database.

Table 3.3: Comparison of our CNNs with other published results

Method	PER
GMMs trained as SVMs[26]	30.1%
SVM [17]	19.7%
NN with 3 hidden layers[3]	22.95%
Deep Belief Networks(DBNs)[20]	23.0%
1-D CNN (this work)	**20.7%**
2-D CNN (this work)	**23.0%**

3.5.2 Low-precision CNNs

As our work focuses on low power feed-forward CNN for phone recognition task. The coefficients in CNNs are supposed already fully trained. Also, from the software experiments, the range of input can be well estimated. At least, a saturation method can be used to constrain the input range. With all the ranges of all coefficients are well known, the coefficients can be pre-computed and stored in memory device. And all the decimal point for each hardware module can be predicted.

In low-precision CNNs, which are also fixed point CNNs, all the inputs and outputs of all operations should be in format of fixed point numbers. Under this circumstance, after the CNNs are well trained, we quantize all the input features coefficients, including weights and bias to mimic fixed point CNNs.

Moreover, the the biggest difference between fixed-point arithmetic and floating-point arithmetic is that some of fixed-point arithmetic, like multiplication and addition will cause dimension increasing. To deal with this , one way is to use truncation method which will further reduce the precision.

In our N bits CNN, N sweeps from 2 to 12, because accuracy doesn't change too much when N is larger then 12, and is very close to accuracy of floating point system. N includes 1 sign bit.

As we can observe, input range of sigmoid layer is roughly from -64 to 64. However, the output range of sigmoid function is from 0 to 1. It is obvious that higher the bit width used in sigmoid module, the better the accuracy. In N bits CNN, we use N bits multiplier, $2 \times N$ bits adder and $2 \times N$ bits sigmoid in order to achieve better accuracy.

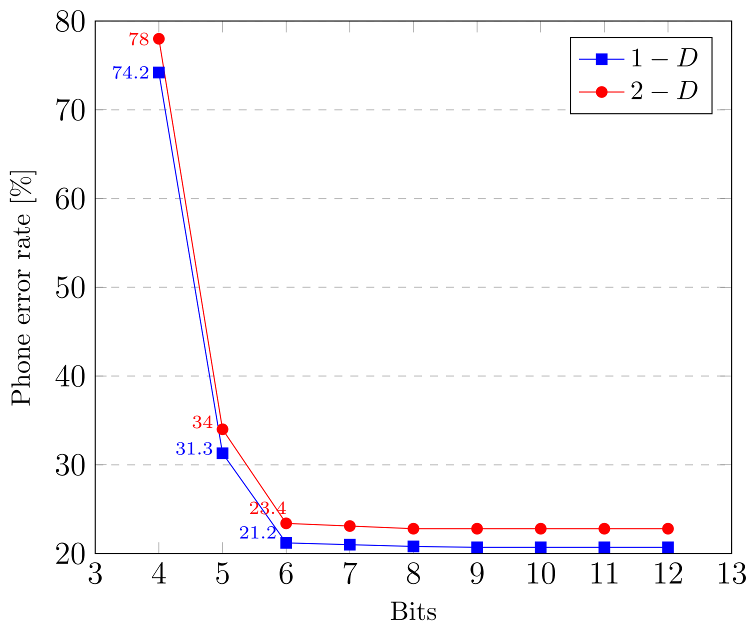


Figure 3.10: PER based on bit width.

The final decoding results presented as PER are shown in Figure 3.10. The two systems are exactly the same except the CNN part.

From the figure, we can see that PER increases exponentially as bit width decrease. When the bit width decreased to 6, the PERs start different from floating point experimental results.

3.6 Conclusion

This chapter introduced in detail about CNNs in KALDI toolkit. And simulation results of software low-precision CNN's implementations. From Figure 3.10 and Table 3.3 we can see that bit width from 5 to 8 is reasonable for implementation of low-power hardware of CNN.

Chapter 4

Hardware Implementation of Low-power convolutional engine

To demonstrate that low precision algorithm can reduce power for CNN implementation on hardware. First gives a simple energy estimation making use of categories of operations. It ignores the power consumes on communication with memories, and simply sum up all the power consumed by each independent operation as total power consumption. The second estimation model are based on implementation for convolutional layer. It includes the access to memory.

All implementations presented here used the Verilog language that target both ASIC synthesis and programmable hardware like FPGAs. And the implementation is application specific integrated chip (ASIC)-based implementations.

4.1 Implementation Tools

For power consumption estimation, we used Synopsys PrimeTime to estimate power from gate-level simulations. PrimeTime takes a placed and routed design and a simulation waveform as input, then produces several gate-level power analysis reports as output. Before this estimation process, we need to prepare another software called Synopsys Design-vision and several files:

- Gate Level Netlist (.v) generated by Design-vision from Synthesizable Verilog HDL description of our design.

- Waveform (.vcd) file that generated by verification process using testbanch (.v), testvectors (.DM) and Gate Level Netlist (.v).

- Activity file (.saif) converted from waveform (.vcd)

- Standard Cell library (.db) and Constraints files (.scd).

The flow can be summarized in Figure 4.1.

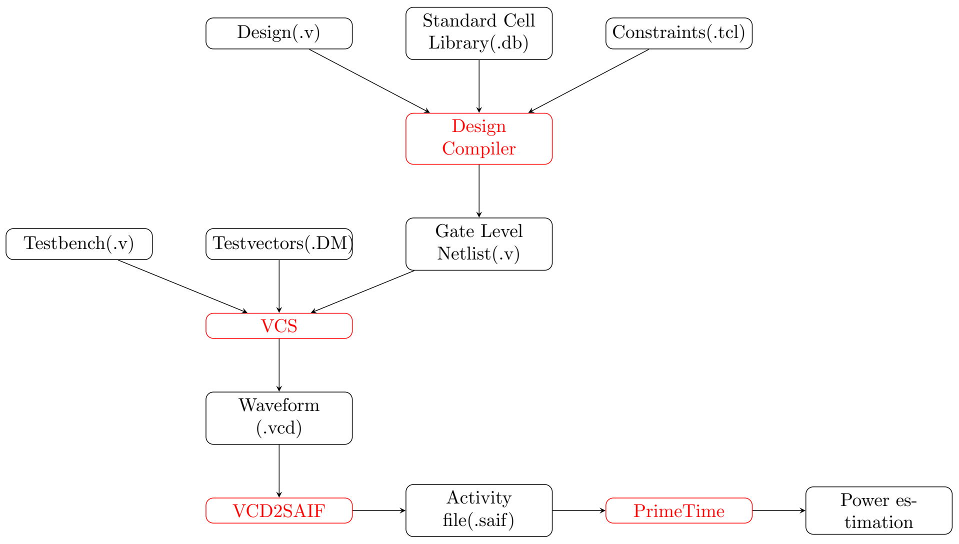


Figure 4.1: Power estimation flow.

4.1.1 General Overview

Efficiency of power is described by total power. Total power is consist of static power and dynamic power. The equation is shown below.

$$P_{tot} = P_{stat} + P_{dym} = I_{leak}V_{DD} + \alpha f_c C_L V_{DD}^2 \tag{4.1}$$

Static Power

When a gate is turned off, it may also has leakage current. And leakage current times supply voltage is the static power.

Dynamic Power

Dynamic power is caused by charge and discharge the load and internal capacitance. According to the capacitance it works on, it can be further divided into two categories: internal power and switching power.

Internal Power

Internal power is the power consumed when charging and discharging internal cell capacitance, it is the power caused by gate itself. Short-circuit power is also included in this catagory.

Switching Power

Switching power is the power when charging and discharging the output load capacitance. The load capacitance which consists of interconnect (net) capacitance and gate capacitance the net is connected to. Switching power is depends on the switching activity of the cell, and switching activity is related to operating frequency.

4.2 A Simple Energy Model

The simple energy model is based on categories of operations needed in 1-D CNN and 2-D CNN. Due to software simulations, all the operations can be divided into 4 categories : addition, multiplication, sigmoid and comparison.

4.2.1 Numbers of different operations

The sum of each operation category is calculated in Table 4.1 and Table 4.2 for 1-D CNN and 2-D CNN respectively. Operations for 1-D CNN is much less than for 2-D CNN.

Table 4.1: Operation numbers for 1-D CNN

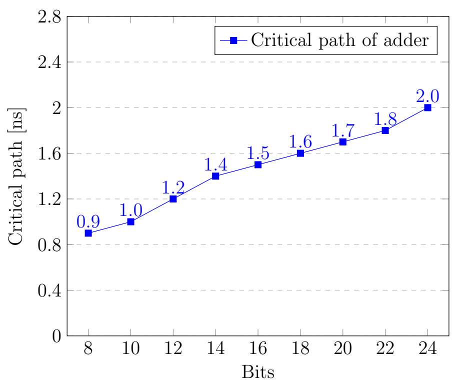
	Operations			
Component	Addition	Multiplication	Sigmoid	Comparison
1 Convolution	$264 \times 33 \times 128$	$264 \times 33 \times 128$	-	-
2 Max-pooling	-	-	-	1408
3 Rescale	-	1408	-	-
4 Addshift	1408	-	-	-
5 Sigmoid	-	-	1408	-
6 Convolution	$512 \times 8 \times 256$	$512 \times 8 \times 256$	-	-
7 Rescale	-	2048	-	-
8 Addshift	2048	-	-	-
9 Sigmoid	-	-	2048	-
(P1) FC	99×200	99×200	-	-
(P2) Sigmoid	-	-	200	-
(P3) FC	200×200	200×200	-	-
(P4) Sigmoid	-	-	200	-
10 FC	2248×1024	2248×1024	-	-
11 Sigmoid	-	-	1024	-
12 FC	1024×1024	1024×1024	-	-
13 Sigmoid	-	-	1024	-
14 FC	1024×1929	1024×1929	-	-
Sum	7,533,291	7,533,291	5904	6528

Table 4.2: Operation numbers for 2-D CNN

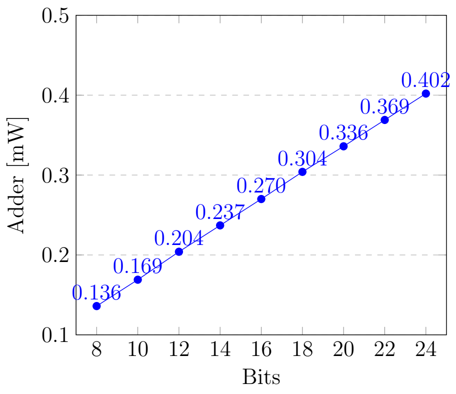
	Operations			
Component	Addition	Multiplication	Sigmoid	Comparison
1 Convolution	$243 \times 105 \times 128$	$243 \times 105 \times 128$	-	-
2 Max-pooling	-	-	-	6528
3 Rescale	-	6528	-	-
4 Addshift	6528	-	-	-
5 Sigmoid	-	-	6528	-
6 Convolution	$1536 \times 14 \times 256$	$1536 \times 14 \times 256$	-	-
7 Rescale	-	3584	-	-
8 Addshift	3584	-	-	-
9 Sigmoid	-	-	3584	-
10 AffineTransform	3584×1024	3584×1024	-	-
11 Sigmoid	-	-	1024	-
12 AffineTransform	1024×1024	1024×1024	-	-
13 Sigmoid	-	-	1024	-
14 AffineTransform	1024×1929	1024×1929	-	-
Sum	15,468,416	15,468,416	12160	6528

4.2.2 Adder

The adder we used here is a N-bit signed adder automatically generated from IP blocks in Synopsis IC compiler. Input dimension is N and output dimension is $N + 1$. Critical path for adders and power consumptions are shown in Figure 4.2. In order to compare the parameters at same clock frequency, we need to give enough margin. As a results, 400MHz is given to do power consumption simulation.



(a) Critical path of adder.

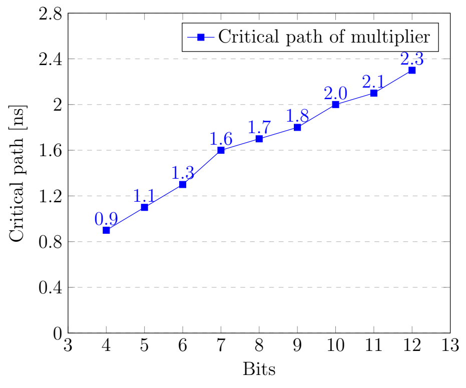


(b) Power of adder @ freq = 400MHz

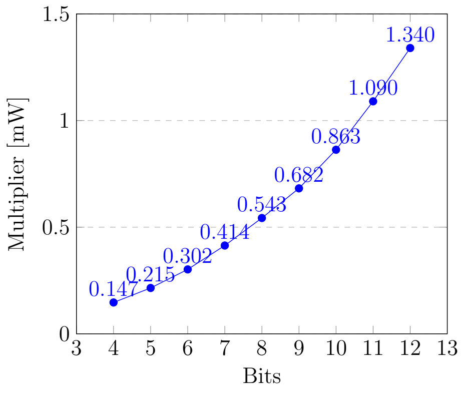
Figure 4.2: Critical path length and average power consumption of adders.

4.2.3 Multiplier

The multiplier here is also automatically generated. The input dimension is N and output dimension is $2 \times N$. Critical path for adders and multipliers are shown in Figure 4.4.



(a) Critical path of multiplier.

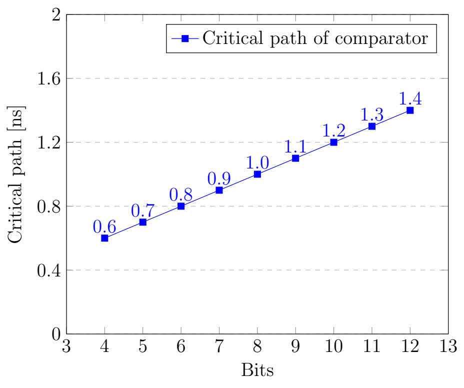


(b) Power of mult @ freq = 400MHz

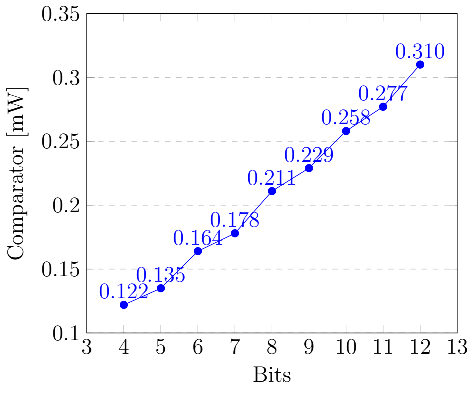
Figure 4.3: Critical path length and average power consumption of multipliers.

4.2.4 Comparator

As max-pooling layers in both 1-D CNN and 2-D CNN use pool size of 3 . So that we use 3 inputs 1 output comparators. We use an input buffer of size of 3. The state machine and critical path for this comparator are shown below. The input are put in series. For 1-D CNN comparator, this module gives one output every 3 clock cycles. For 2-D CNN comparator, the module gives one output every 2 clock cycles.



(a) Critical path of comparator.



(b) Power of comparator @ freq = 400MHz

Figure 4.4: Critical path length and average power consumption of comparators.

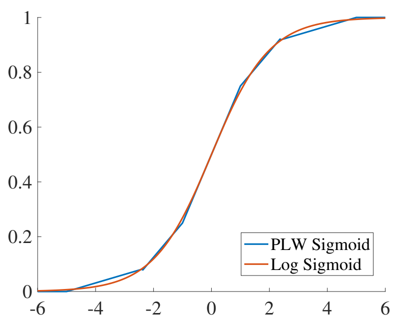
4.2.5 Sigmoid

There are many ways to implement sigmoid function, such as using CORDIC function, Look Up tables (LUTs) and Piece-wise Linear (PWL) or quadratic approximations. However, each of these methods has its defects. A CORDIC function gains accuracy at the cost of latency. Piece-wise approximation has limited accuracy. Since our work focus more on power and accuracy, implementation that use Look Up Tables (LUTs) seems a good choice here. But, as our design need to sweep the data width from 2 bits to more than 10 bits, the LUT becomes a big RAM and will definitely consume huge static power. According to software simulations, the input range of sigmoid module is fixed and normally used is from -64 to 64, which also means 6 bits for integer part. And the output of sigmoid module must be in the range from 0 to 1. With a determined format of fixed-point numbers, an easy way to implement sigmoid function is to use PWL.

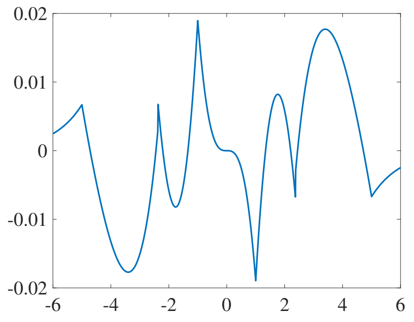
The PWL approximation of Sigmoid activation function was proposed by Manish Panicker[21] and is shown in Table 4.3, where X stand for input and Y stand for output.

Table 4.3: Input range for each layer in 2-D CNN

condition	operation
$X \geq 5$	$Y = 1$
$2.375 \leq X < 5$	$Y = 0.03125 \times X + 0.84375$
$1 \leq X < 2.375$	$Y = 0.125 \times X + 0.625$
$-1 \leq X < 1$	$Y = 0.25 \times X + 0.5$
$-2.375 \leq X < -1$	$Y = 0.125 \times X + 0.375$
$-5 \leq X < -2.375$	$Y = 0.03125 \times X + 0.15625$
$X < -5$	$Y = 0$



(a) Sigmoid function and PWL approximation



(b) Error

Figure 4.5: PWL approximation of sigmoid function

Figure 4.5 shows the comparison between the PWL approximation of sigmoid function and real log sigmoid function:

$$F(z) = \frac{1}{1 + e^{-z}} \tag{4.2}$$

It shows that the maximum error is 0.02. As the input and output of sigmoid module will be further quantized in our implementation, this error is acceptable compared to the quantization error when we use small bit width.

Figure 4.6 shows the block diagram of our implementation. Parameters a and b are predifined inside the block. The critical path of sigmoid module is shown in Figure 4.7.

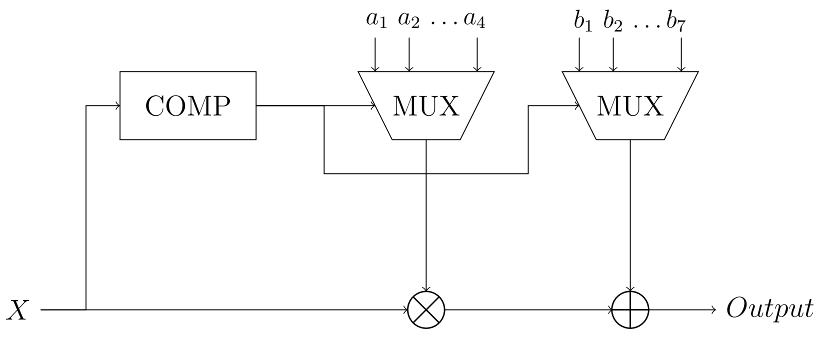
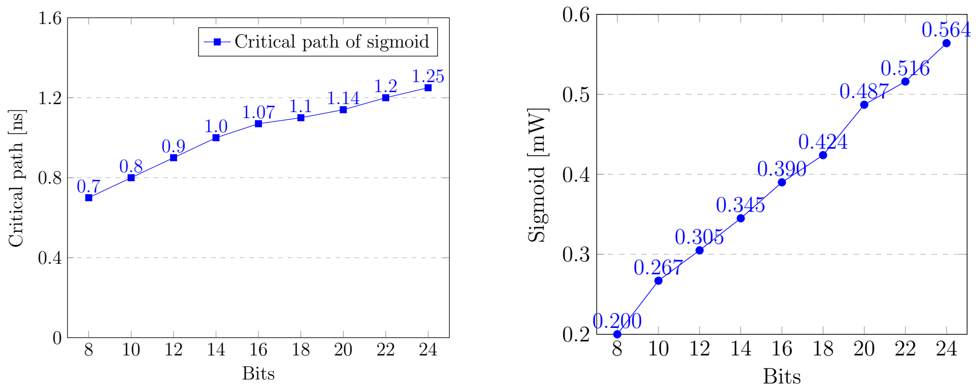


Figure 4.6: Block diagrams for Sigmoid module.



(a) Critical path of sigmoid.

(b) Power of sigmoid @ freq = 400MHz

Figure 4.7: Critical path length and average power consumption of sigmoid.

4.2.6 Energy model of CNNs

According to the result in software experiments, we did a sweep of bit width from 2 to 12. when discuss with power consumption, we need to determine the clock frequency first. According to critical path of each basic modules above, the maximum frequency can be used is shown in Figure 4.8.

From the figure, the maximum clock frequency can be used is 434.78 MHz. So we choose a frequency of 400MHz as a reference for energy estimation.

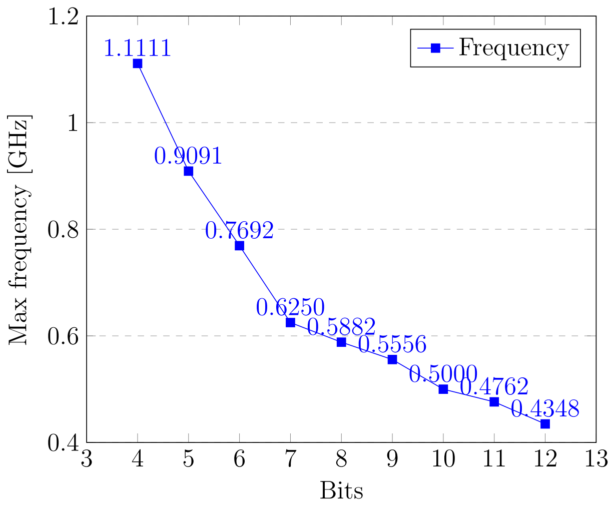


Figure 4.8: maximum frequency for CNNs based on basic modules.

The simple estimation model is given by equation:

$$Energy = \frac{P_a \times NUM_a + P_m \times NUM_m + P_c \times NUM_c + P_s \times NUM_s}{clock_{freq}} \tag{4.3}$$

In which NUM represents number of operations, P stands for power, a for adder, m for multiplier, c for comparator and s for sigmoid.

The average power for each basic module, are simulated based on a random generated number set. The clock frequency is 400 MHz and supply voltage is 1.1 V. And library used here is NanGate 45 nm Standard Cell Library.

The estimated energy for 1-D CNN and 2-D CNN are shown in Figure 4.9.

For analysis of relationship between accuracy and energy consumption, combine Figure 4.9 with Figure 3.10, we may roughly know the relation of accuracy and energy consumption as shown in Figure 4.10.

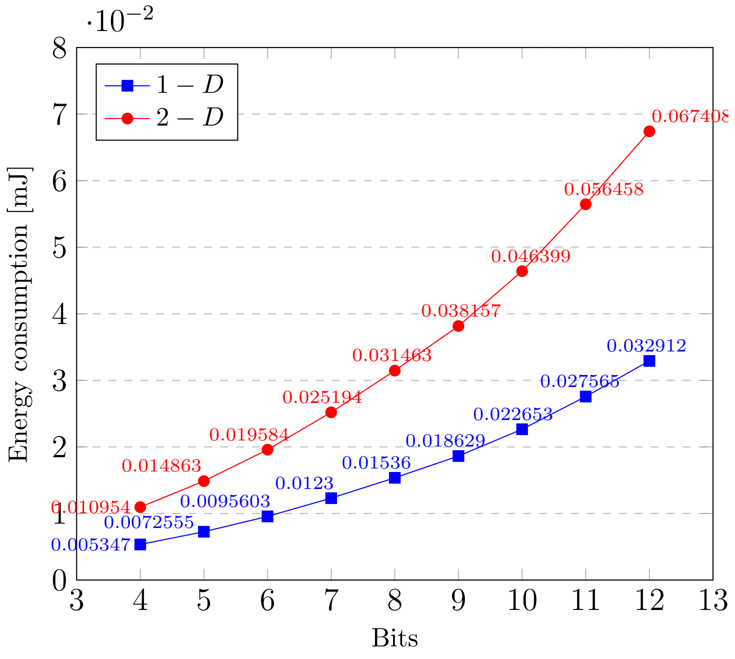


Figure 4.9: Energy estimation

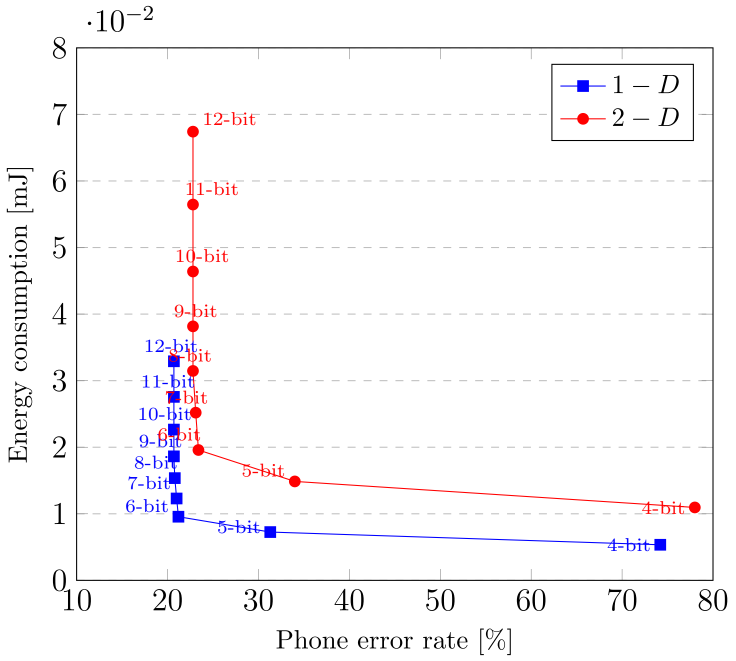


Figure 4.10: Accuracy Vs Energy.

4.3 Convolutional Layer Implementation

A higher level estimation is based on architecture level. Although there are many layers in whole CNN, the convolutional layer is the core part of a CNN. Our work is to implement a convolutional engine that can be used for first layer of 1-D and 2-D CNNs and make some optimization.

The block diagram of convolutional convolutional is shown in Figure 4.11. First, we need a CNN with already trained parameters. Then all the parameters and input test features are stored in external memory as fixed-point numbers. This external memory absolutely has constraints but are ignored here. Because the performance of a accelerator processor is limited by memory bandwidth and computing ability. We make an assumption that memory bandwidth is large enough.

Based on this design, an energy estimation focused on convolutional layer can be built. According to energy efficiency, an optimization can be made based on different degrees of parallelism.

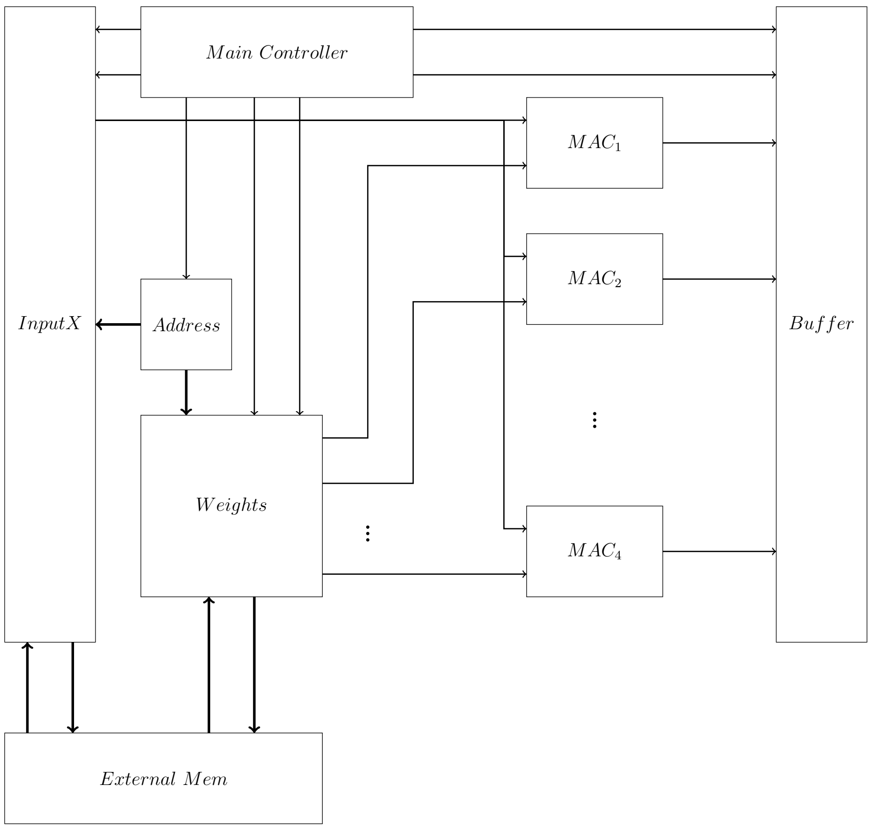


Figure 4.11: Block diagrams for convolutional layer.

In our design, there are n modules implemented, without the output buffer. We can choose which degree of parallelism to implement by choosing how many MACs working in parallel. When there are n MACs working simutaneously, we call the degree is np, and we need n independent weights caches.

4.3.1 Multiply-accumulate unit

The most important module in convolutional layer is the Multiply-accumulate unit(MAC), which can perform a filter process. The block diagram is shown below.

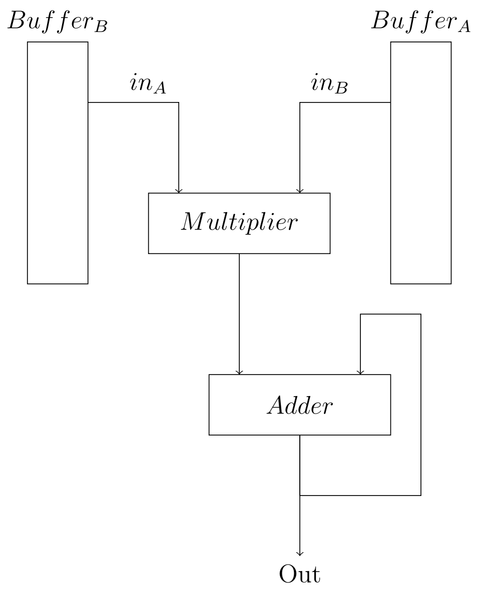


Figure 4.12: Block diagrams for MAC module.

MAC module is controlled by main controller(MC) as well as address controller(AC). AC gives *start_sig* and *out_sig* to MAC to decide when to start accumulate and when to give outputs. There are 4 states in a MAC module as shown in Figure 4.13.

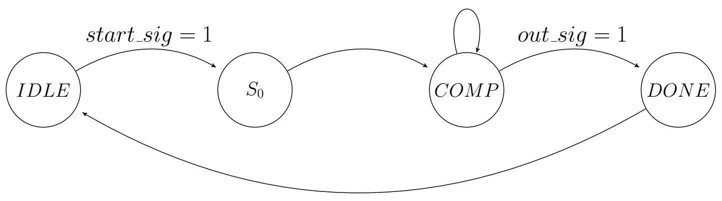


Figure 4.13: State machine of MAC.

Behavior of each state are shown in table Table 4.4.

Table 4.4: Behavior of states in MAC

State	Description
IDLE	Stay at initial state till *start_sig* are set.
S0	Start compute with $o = w \times x$, where w and x are inputs of MAC.
COMPUTE	Accumulate with $o = o + w \times x$.
OUT	Update output reg with $out = o + b$, in which b is bias.

4.3.2 Main controller

This module is to control the rest modules in the block: Address controller, inputs/weights caches and MACs. This module checks the state of rest modules and give control signals to them.

The most important control signals are *D_read*, *M_write*, *restart* and *D_done* signals. *M_write* is set to 1 when weights cache module need to load parameters from external memory and other devices need to wait. After caches are ready to be read, *D_read* is set to 1 and *M_write* is set to 0. Then address controller and MACs can start to compute using data stored in caches and input buffers. When one round of convolution on input feature maps has done,it means parameters in weights caches should be update to another filter. Then the main controller will receive a *ready* signal and jump to RESTART state to set *restart* signal to 1.

The state machine is shown in Figure 4.14. Behavior of each state are shown in table Table 4.7.

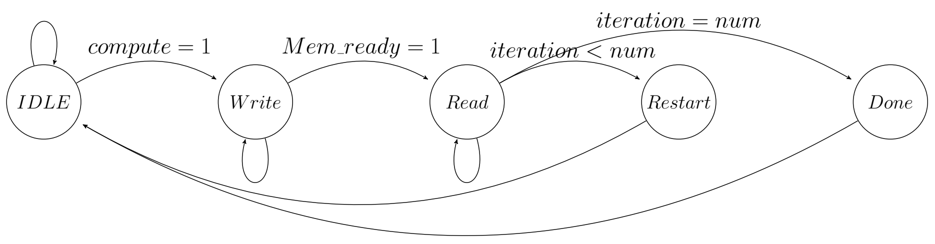


Figure 4.14: State machine of main controller.

Table 4.5: Behavior of states in Main controller

State	Description
IDLE	Set all signals to 0 untill *compute* = 1 and jump to WRITE state.
WRITE	Set $M\_write = 1$ and $D\_read = 0$. When caches are ready to be read, jump to READ state.
READ	Set $M\_write = 0$ and $D\_read = 1$. When one round of convolution is done, jump to RESTART state or DONE state.
RESTART	Set *restart* signal to 1, and jump to IDLE.
DONE	Set *D_done* signal to 1, and jump to IDLE.

4.3.3 Input/weights cache

In order to reduce power of reading from external memory and speed up the chip, one method is using on-chip SRAMs, which in our work are called input cache and weights caches. We use a group of registers to mimic SRAMs.

The input/weights caches is not caches that can be used in CPUs. They are simpler than the latter. These caches only has two functional phases: reading mode and writing mode. At writing mode, when $M\_write = 1$ and $D\_read = 0$, caches load parameters (weights and bias) from external memory, which are not implemented here, and other modules have no access to this cache. At reading mode, address controller and MACs read parameters from these caches, which will save much energy compared to the ones that read parameters directily from external memory.

The state machine is shown in Figure 4.15.

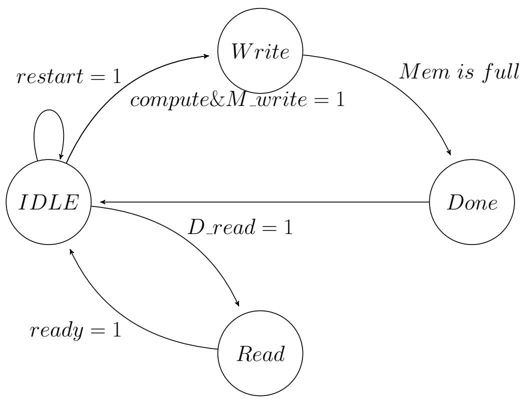


Figure 4.15: State machine of cache.

And behavior of each state are shown in table Table 4.6.

Table 4.6: Behavior of states in Input/weights cache

State	Description
IDLE	Initial $M\_address$, which are used to read parameter from external memory.
WRITE	Let $M\_address = M\_address + 1$. Write $data\_in$ to relative register. When registers are full, jump to IDLE and wait for control signals.
READ	Let $data\_out = mem[address]$, where mem is registers.
DONE	Set MEM_ready signal to 1, which is input of main control unit.

4.3.4 Address controller

Address controller is the most complex module in this block. It's function is to decide which address to be read during reading mode of input/weights caches. The input features for 2-D CNN are shown in Figure 4.17. We assume that they are stored in input cache row by row as a vector.

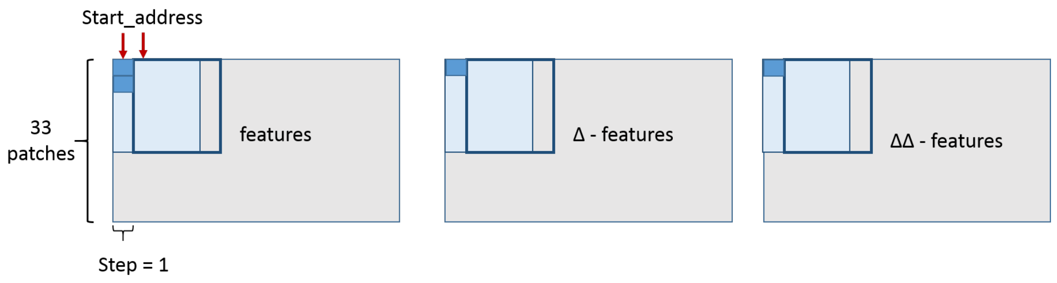


Figure 4.16: Input features of 2-D CNN.

In computing mode, address for input($addr\_x$) should add a fixed feature map length or add a shift value in every clock cycle. After it goes through one local block , at a size of $9 \times 9 \times 3$ in 2-D CNN, a synapse process is done and an output should be generate. Then the $addr\_x$ should change its start address based on its convolution step. To realize this convolution function, we use large amounts of states, shown in Figure 4.17.

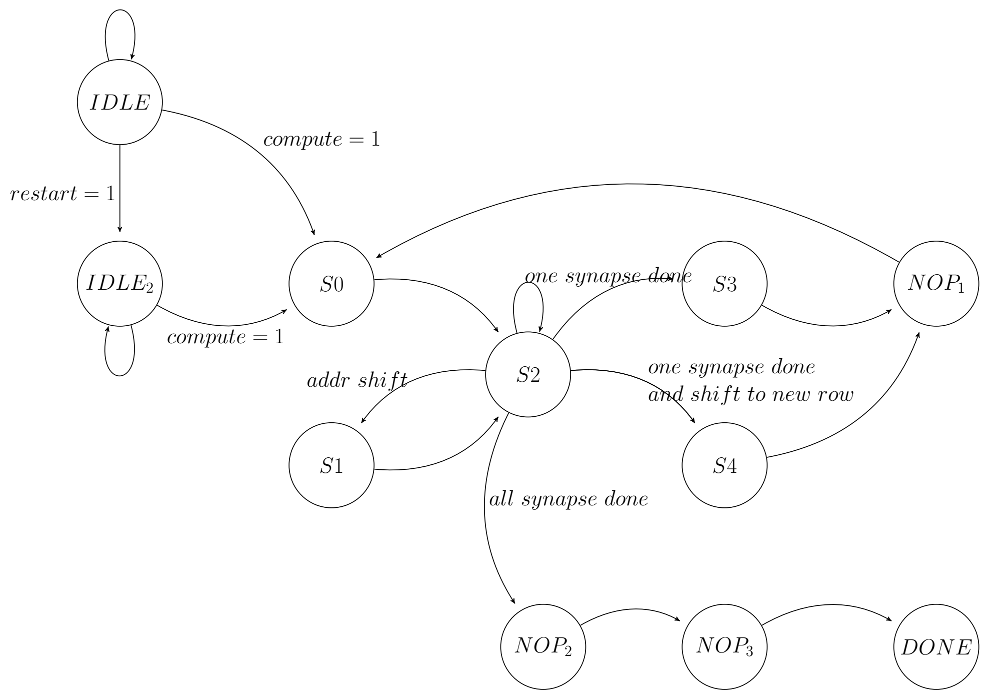


Figure 4.17: Input features of 2-D CNN.

And behavior of each state are shown in table Table 4.6.

Table 4.7: Behavior of states in input/weights cache

State	Description
IDLE	Initial $M\_address$, which are used to read parameter from external memory.
WRITE	Let $M\_address = M\_address + 1$. Write data in to relative register. When registers are full, jump to IDLE and wait for control signals.
READ	Let $data\_out = mem[address]$, where mem is register array.
DONE	Set MEM_ready signal to 1, which is input of main control unit.

4.3.5 Results and optimization

Using the building blocks above, we can do some experiments to mimic a convolutional layer. There is a counter named *iteration* to count how many group of filters are used.

In this work, we implement the first convolutional layer in two CNNs. So there is totally 128 trained filters stored in external memory and waiting to be load. If n MACs and n weights caches works in parallel, we need to reload the caches for

$(\frac{128}{n} - 1)$ times. The working timing is shown in Figure 4.18. It is implicit that writing phase and reading phase occurs alternatively.

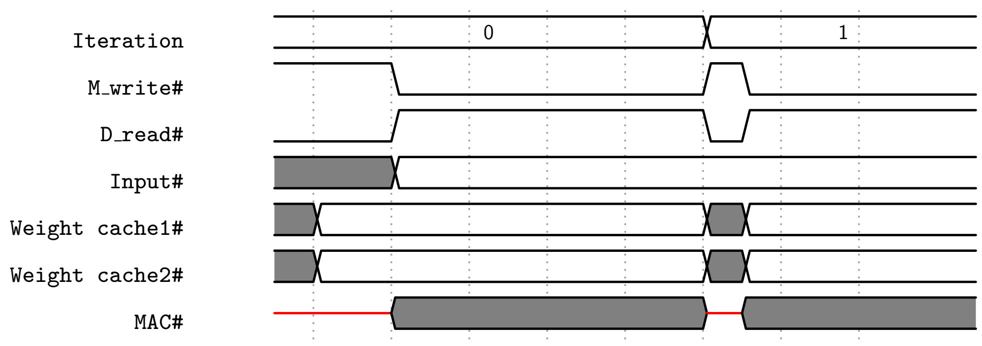
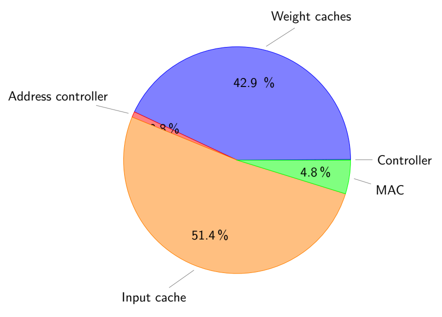


Figure 4.18: Scheduling of synapse.

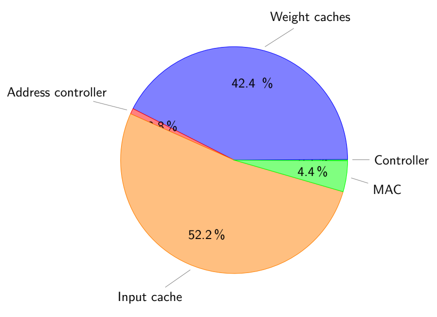
Simulation is done using Design Compiler. The critical path of 4 MACs implementation is 2.47ns. To implement higher level of parallelism in this convolutional engine, and make comparison, we set frequency at 250MHz. It means that in this work, clock frequency can be in a range of 250MHz to 400MHz.

Partition of total power is presented by pie gragh shown in Figure 4.19. Because when we use parallel MACs and weights caches, the input cache is shared. But the power of input cache doesn't go very high when using higher parallelism. It only increase less than 1% when the number of MACs working in parallel doubled, due to the increase of load capacitance. Thus, in pie graphs, we can see that as parallelism goes higher, just weights caches become more dominant relatively.

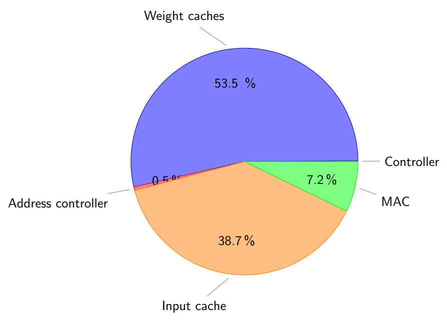
From the scheduling above, we can calculate MAC efficient, shown in Figure 4.20, from accumulating the time that MAC works over the time needed for whole layer. From the figure we can see that MAC efficient decreased when parallelism goes higher. Because we the first time we load input frames from external memory we need approximately 5 times long time. And as parallelism goes higher, the *iterations* we need to reload weights decrease. This lead to relatively lower MAC efficiency when parallelism goes higher.



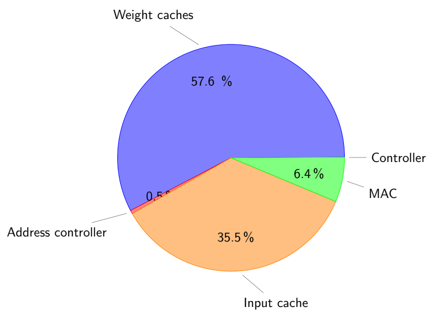
(a) 4 MACs in parallelism in 1-D CNN.



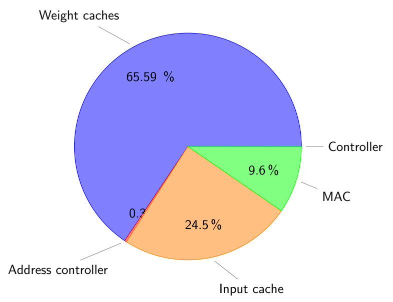
(b) 4 MACs in parallelism in 2-D CNN.



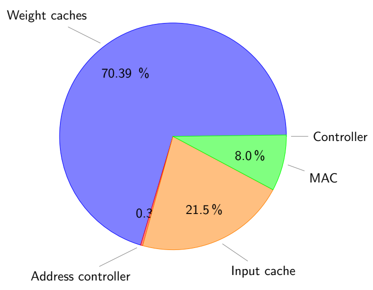
(c) 8 MACs in parallelism in 1-D CNN.



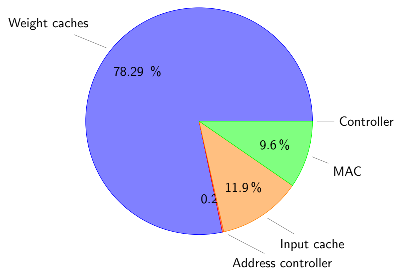
(d) 8 MACs in parallelism in 2-D CNN.



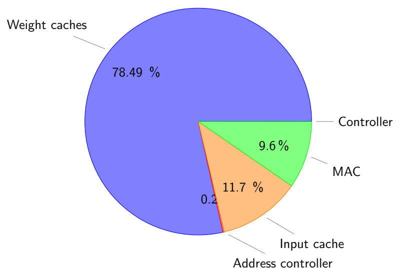
(e) 16 MACs in parallelism in 1-D CNN.



(f) 16 MACs in parallelism in 2-D CNN.



(g) 32 MACs in parallelism in 1-D CNN.



(h) 32 MACs in parallelism in 2-D CNN.

Figure 4.19: Partition of energy consumption of each modules in CNNs.

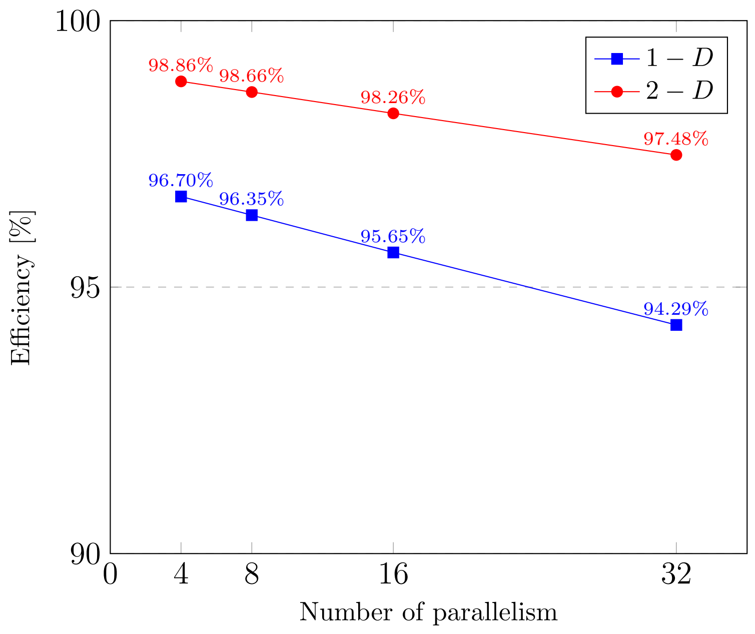


Figure 4.20: Efficiency of MACs

The total energy consumption of the first convolutional layer can be calculated based on an assumption of power of access to external memory. We can assume that:

$$P_{write} = 1.5 \times P_{mult} \tag{4.4}$$

where P_{write} is the power consumed when input/weights caches loading parameters from external memory.P_{mult} is power of multiplier, here we use $P_{mult} = 0.302(mW)$ as a reference. Power of each implementation are listed in Table 4.8.

	1-D CNN		2-D CNN		
Parallelism	Power(W)	Time(ms)	Power(W)	Time(ms)	Peak P_write(W)
4	0.0285	1.153	0.0341	3.3	2.42
8	0.0399	0.579	0.0463	1.653	4.35
16	0.063	0.291	0.0764	0.830	8.21
32	0.120	0.148	0.140	0.422	15.94

Table 4.8: Average power and time for first convolutional layers in CNNs @ freq = 250MHz.

So that energy consumption of first convolutional layer in CNNs are estimated in Figure 4.21.

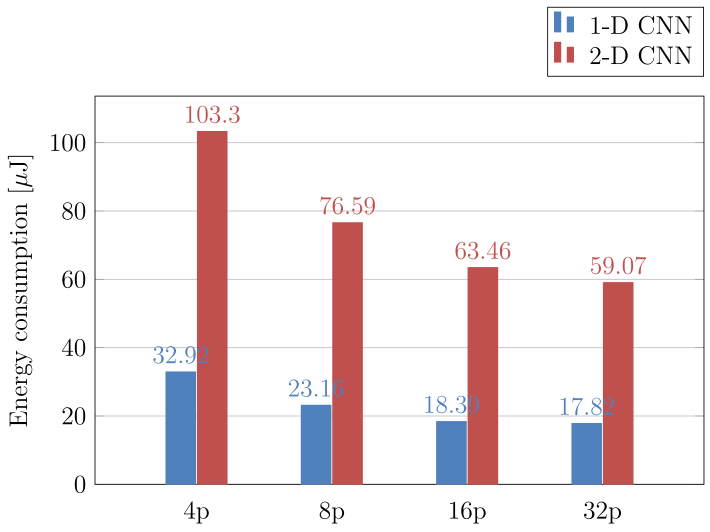


Figure 4.21: Energy of first convolutional layers. p stand for number of MACs working in parallel.

From this figure, we can conclude that in terms of energy consumption, we'd better use more weights caches and MACs work simultaneously. But from phase of MAC efficiency and area consideration, less parallelism is better.

4.3.6 Conclusion

This chapter first estimate energy consumption based on numbers of operations. This first order estimation gives a sight that an implementation of CNNs at bit width of 6 is optimum for both accuracy and energy consideration. Second, we gives an implementation of first layer in 1-D CNN and 2-D CNN, and gives an optimization trend. For our aims of energy efficient CNN implementation, parallelism of 32 MACs seems fits the goal. A comparison of performance of our work and other existing works are shown in Table 4.9.

Table 4.9: Comparisons of performance.

Name	Tech	Performance	Energy efficiency
NeuFlow[22]	IBM 45nm	320GOPS	230GOPs/W
nn-X[12]	Xilinx Zynq	240GOPS	25 GOPS/W
Qadeer et al.[24]	IBM 45nm	410GOPS	236 GOPS/W
DianNao[6]	TSMC 65nm	452GOPS	931GOPS/W
IBM TrueNorth[19]	Samsung 28nm	-	46 GOPS/W
Our work(1-D CNN)	NanGate 45nm	15 GOPS	125.6 GOPS/W
Our work(2-D CNN)	NanGate 45nm	15.5 GOPS	110.5 GOPS/W

Chapter 5

Conclusion and Future Work

In this thesis, we design a convolutional engine that can be used in CNN/HMM hybrid ASR system based on KALDI nnet1 software implementation. What we have done are shown below:

1. We have implemented several neural networks in KALDI nnet1 framework using default setups. The data set used here is TIMIT corpus. Input features are MFCC features with a dimension of 43×3 for each frame and frame length is 25ms. After splicing step, dimension for each frame extands to 1419. In KALDI, we use 3-state Bakis model HMM topology. There are two topology of CNNs implemented in nnet1 framework: 1-D CNN and 2-D CNN. 1-D CNN uses 2 concolutional layers, 3 fully connected layers and 2 small fully connected layers as parallel component. 2-D CNN used 2 convolutional layers and 3 fully connected layers. The floating point results of these 2 architectures are 20.7% and 23.0% of phone error rate respectively. Compare to the state-of-art experiments based on TIMIT data set, shown in Table 3.3, the results are acceptable.

2. Based on simulation results, we find out that low bit width fixed point computing is enough for accuracy consideration in ASR system. After CNNs are fully trained, we quantize all the parameters and test input features. Using this parameters and inputs, we can mimic a fixed point network. With different degrees of quantization, the results are given in Figure 3.10.

3. We figure out the optimum bit width that saves energy without loosing much accuracy. Using the operations needed in two CNNs and power for each single operation, we estimate the energy for each CNN. The sigmoid module are implemented using PWL method. After that, we draw a relation of accuracy and energy consumption. In consideration of accuracy and energy, CNNs of bit width of 6 is Pareto optimum point.

4. We implement a convolutional engine at a bit width of 6 that can be used in first convolutional layers in CNNs, and evaluate its performance. Degrees of parallelism is considered as a parameter to optimize. It is obviously from

results that the higher the parallelism, the higher the power, but with larger throughput and lower energy consumption.

From the performance and energy efficiency shown in Table 4.9, advantage of our work is low power and focus on ASR system. Disadvantage is low throughput. To optimize it, we can go much higher parallelism. But fan out of input cache is much higher, and bandwidth of external memory will be a limitation.

There is still large space to do research on. We can fully implement these CNNs and optimize its architecture. Or we can try other architectures of convolutional engine to realize higher parallelism, like use bufferlines for input features.

Bibliography

[1] O. Abdel-Hamid, L. Deng, and D. Yu. Exploring convolutional neural network structures and optimization techniques for speech recognition. In *Interspeech*, pages 3366–3370, 2013.

[2] O. Abdel-Hamid, A.-R. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on audio, speech, and language processing*, 22(10):1533–1545, 2014.

[3] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, and G. Penn. Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition. In *2012 IEEE international conference on Acoustics, speech and signal processing (ICASSP)*, pages 4277–4280. IEEE, 2012.

[4] P. M. Baggenstoss. A modified baum-welch algorithm for hidden markov models with multiple observation spaces. *IEEE Transactions on speech and audio processing*, 9(4):411–416, 2001.

[5] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 247–257. ACM, 2010.

[6] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, volume 49, pages 269–284. ACM, 2014.

[7] A. Coates, B. Huval, T. Wang, D. J. Wu, and A. Y. Ng. Deep learning with cots hpc systems. *International Conference on Machine Learning*, 2013.

[8] M. Courbariaux, Y. Bengio, and J.-P. David. Low precision arithmetic for deep learning. *arXiv preprint arXiv:1412.7024*, 2014.

[9] G. E. Dahl, D. Yu, L. Deng, and A. Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(1):30–42, 2012.

[10] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, D. S. Pallett, N. L. Dahlgren, and V. Zue. Timit acoustic-phonetic continuous speech corpus. *Linguistic data consortium, Philadelphia*, 33, 1993.

[11] P. Ghahremani, B. BabaAli, D. Povey, K. Riedhammer, J. Trmal, and S. Khudanpur. A pitch extraction algorithm tuned for automatic speech recognition. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2494–2498. IEEE, 2014.

[12] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello. A 240 g-ops/s mobile coprocessor for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 682–687, 2014.

[13] G. Govindu, L. Zhuo, S. Choi, P. Gundala, and V. K. Prasanna. Area, and power performance analysis of a floating-point based application on fpgas. Technical report, DTIC Document, 2003.

[14] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[15] H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin. Exploring strategies for training deep neural networks. *Journal of Machine Learning Research*, 10(Jan):1–40, 2009.

[16] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[17] H. Lee, P. Pham, Y. Largman, and A. Y. Ng. Unsupervised feature learning for audio classification using convolutional deep belief networks. In *Advances in neural information processing systems*, pages 1096–1104, 2009.

[18] B. Logan et al. Mel frequency cepstral coefficients for music modeling. In *ISMIR*, 2000.

[19] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

[20] A.-r. Mohamed, G. Dahl, and G. Hinton. Deep belief networks for phone recognition. In *Nips workshop on deep learning for speech recognition and related applications*, volume 1, page 39, 2009.

[21] M. Panicker and C. Babu. Efficient fpga implementation of sigmoid and bipolar sigmoid activation functions for multilayer perceptrons. *IOSR Journal of Engineering (IOSRJEN)*, pages 1352–1356, 2012.

[22] P.-H. Pham, D. Jelaca, C. Farabet, B. Martini, Y. LeCun, and E. Culurciello. Neuflow: Dataflow vision processing system-on-a-chip. In *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1044–1047. IEEE, 2012.

[23] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, et al. The kaldi speech recognition toolkit. In *IEEE 2011 workshop on automatic speech recognition and understanding*, number EPFL-CONF-192584. IEEE Signal Processing Society, 2011.

[24] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz. Convolution engine: balancing efficiency & flexibility in specialized computing. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 24–35. ACM, 2013.

[25] T. N. Sainath, A.-r. Mohamed, B. Kingsbury, and B. Ramabhadran. Deep convolutional neural networks for lvcsr. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8614–8618. IEEE, 2013.

[26] F. Sha and L. K. Saul. Large margin gaussian mixture modeling for phonetic classification and recognition. In *2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings*, volume 1, pages I–I. IEEE, 2006.

[27] D. Talkin. A robust algorithm for pitch tracking (rapt). *Speech coding and synthesis*, 495:518, 1995.

[28] K. Vesel`y, A. Ghoshal, L. Burget, and D. Povey. Sequence-discriminative training of deep neural networks. In *INTERSPEECH*, pages 2345–2349, 2013.

[29] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. J. Lang. Phoneme recognition using time-delay neural networks. *IEEE transactions on acoustics, speech, and signal processing*, 37(3):328–339, 1989.

[30] Wikipedia. Speech recognition. URL: `https://en.wikipedia.org/wiki/Speech_recognition`, last checked on 2016-08-13.

[31] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

Master thesis filing card

Student: Huizhao Wang

Title: Design, analysis and implementation of a low-power convolutional engine for speech recognition system

UDC: 621.3

Abstract:

Convolutional Neural Networks(CNN) combined with Hidden Markov Models(HMM) are widely used in today's Automatic Speech Recognition(ASR) system. But CNNs used in ASRs are normally with large-scale. To accelerate CNNs in ASRs, we introduce a low-power solution: using low bit width fixed point computation instead of floating point computation. Software simulation results using KALDI toolkit and TIMIT database shows that CNNs with bit width of 6 is enough to get accuracy very close to floating point results, which are 20.7% and 23.0% for two different CNN. Based on this results, we implement an convolutional engine, an ASIC implementation, that has a bit width of 6. This convolutional engine uses NanGate 45nm cell library, and works at 250MHz with supply voltage of 1.1V. It uses 32 MAC units working in parallel, and on-chip SRAMS to store part of parameters and a frame of inputs. This low-power engine can work at a very low power of 0.12W and 0.14W, and energy efficiency of 125GOPS/W and 110.5GOPs/W respectively.

Thesis submitted for the degree of Master of Science in Electrical Engineering, option Electronics and Integrated Circuits

Thesis supervisor: Prof.Dr.Ir. Marian Verhelst

Assessors: Prof.Dr.Ir. Wim Dehaene
Prof.Dr.Ir. Luc Van Gool

Mentors: Ir. Bert Moons
Ir. Bert De Brabandere