HARDWARE ACCELERATION OF GMM BASED ACOUSTIC SCORING IN AUTOMATIC SPEECH RECOGNITION

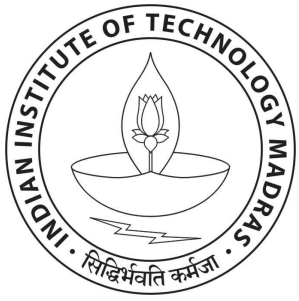
A Project Report

submitted by

ARJUN S B

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY



DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

MAY 2019

THESIS CERTIFICATE

This is to certify that the thesis titled **HARDWARE ACCELERATION OF GMM BASED ACOUSTIC SCORING IN AUTOMATIC SPEECH RECOGNITION**, submitted by **ARJUN S B**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bona fide record of the research work done by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Nitin Chandrachoodan
Project Guide
Associate Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 5th May 2019

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to the following people without whom my project work at IIT Madras, would not have been possible.

First and foremost, I express my sincere gratitude to Dr. Nitin Chandrachoodan for his valuable guidance, constant support, and encouragement for completing the project. I would also like to thank the Department of Electrical Engineering, IIT Madras for providing the facilities required for the completion of the project.

I would like to extend my gratitude to Prithvi Raj, Akhil Reddy and Radhika for providing support throughout my project related activities. I thank all my lab mates for their guidance and support.

Last, but most importantly I thank my family for their encouragement and motivation without whom it would not have been possible for me to finish my M-Tech Course.

ABSTRACT

Automatic Speech Recognition(ASR) has become a useful tool in speech interfaces that governs several man-machine interfaces. The demand for accurate, real-time ASR systems always push the researchers to find an efficient way to implement the ASR. The most critical part of an ASR system is acoustic scoring. Over the years several methods have been used for acoustic scoring. This project uses the Gaussian Mixture Model(GMM) based acoustic scoring which calculates the score based on multidimensional Gaussian distributions.

This project aims at implementing a hardware accelerator for the GMM based acoustic scoring component of an ASR system. The implementation targets at methods to speed up the GMM function so as to produce real-time speech recognition. The proposed accelerator is implemented and tested on ZedBoard and ZCU102 boards. The main challenge for GMM hardware implementation is the bulk data transfer from the external memory to the hardware function. To analyze the efficient way to transfer the data, a memory bandwidth test for the target FPGA boards is also done. Apart from optimizing the hardware function, this project also describes the efficient way of caching the score which eventually reduces the total GMM evaluation time for a particular audio wave.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ASR	Automatic Speech Recognition
GMM	Gaussian Mixture Models
HMM	Hidden Markov Model
FPGA	Field Programmable Gate Array
SoC	System On Chip
PS	Processing System
PL	Programmable Logic
RTL	Register Transfer Level
HLS	High Level Synthesis
IP	Intellectual Property
MFCC	Mel Frequency Cepstral Coefficients
CMVN	Compute Mean and Variance Normalization
LDA	Linear Discriminant Analysis
FST	Finite State Transducer
WFST	Weighted Finite State Transducer
PDF	Probability Density Function
CPU	Central Processing Unit
DFT	Discrete Fourier Transform
FIFO	First In First Out
FSM	Finite State Machine
FFT	Fast Fourier Transform
RAM	Random Access Memory
API	Application Programming Interface
AXI	Advanced eXtensible Interface
WER	Word Error Rate
DMA	Direct Memory Access
DAC	Digital to Analog Converter
ADC	Analog to Digital Converter

BRAM	Block RAM
II	Initiation Interval
LUT	Look Up Table
HDL	Hardware Description Language
SDK	Software Development Kit
HDF	Hardware Description File
ILA	Integrated Logic Analyzer

CHAPTER 1

INTRODUCTION

1.1 Introduction

Automatic Speech Recognition(ASR) is being widely used in speech interfaces that are now used for a variety of human-machine interactions. Voice-based user interfaces require more accurate, large vocabulary, real-time ASR.ASR is a compute-intensive process and therefore supporting large vocabulary with appreciable accuracy comes at a high computational cost. The challenge also involves meeting the energy constraints in platforms such as smart-phones, tablets or wearable devices.

Cloud-based (software) implementations of speech recognition which are put into applications in Internet-connected devices ensures real-time decoding with high accuracy. But the reliability of such devices to the Internet availability restricts its use in local applications. So hardware speech recognition becomes useful when the Internet connection is slow or for local operations that do not require Internet capabilities. Also, the devices with limited battery capacity which cannot afford the overhead of using cloud-based decoder can make use of hardware speech recognition.

The major compute intense part of any ASR system is the acoustic scoring. In this project, we use Kaldi toolkit for decoding. In the proposed ASR system, the input audio signal coming out from an ADC is split into frames. For each frame of speech, the acoustic scoring computes the logarithm of likelihood, referred to as 'acoustic_cost', that the frame is part of a particular phoneme, for all potential phonemes in the language. The version of Kaldi decoder used in this project uses Gaussian Mixture Models(GMM) for acoustic scoring. Considering both memory footprint and accuracy, GMM based acoustic scoring is best suited for ultra-low power, low-cost devices.

1.2 Motivation

Real-time ASR decoding is computationally demanding. The computational requirements often increase as researchers identify improved modeling techniques. Hardware acceleration is an efficient approach to achieve high performance, high speed, and low power ASR. Hardware accelerators are optimized functional blocks designed to offload specific tasks from general purpose CPUs.Due to their optimized and dedicated architecture these blocks can perform faster than analogous software running in CPU.

Field Programmable Gate Arrays(FPGA) are very suitable to host hardware accelerators since they can perform highly optimized functions at the gate level. On Integrating the processor and FPGA systems, the processor can be offloaded by accelerating practically anything in FPGA logic-from calculating a cyclic-redundancy-check (CRC) to offloading the entire TCP/IP stack. When the FPGA-based accelerator produces a new result, the data needs to be passed back to the processor as quickly as possible, so that the processor can update the data. Transferring the data quickly and coherently is the key to realizing this performance boost. The integration of an ARM processor and FPGA logic with high speed, on-chip interconnect buses for performance, along with an Accelerator Coherency Port for coherency, makes this possible in the SoC FPGA-based systems of today.

1.3 Objective and Scope

In the proposed system, ASR transcribes an audio stream into a sequence of words. The input audio signal, fed to the system, coming out of ADC is a 16 KHz, 16-bit signal. The signal is split into a set of frames. Each frame is typical of 25ms (400 samples) duration and has a frameshift of 10ms. This the undergo a sequence of signal processing and finally gives out a 40-dimensional vector called the 'feature' vector. The whole process of taking the input from ADC to giving out the feature vector constitutes the front end part of ASR. The Back end of ASR is usually expressed as a Hidden Markov model (HMM) inference problem, where the hidden variables are states within a Markov process modeling speech production, and the observed variables are acoustic features. Decoding an utterance is equivalent to maximum a posteriori (MAP) inference

over the HMM, for which we use the Viterbi algorithm. Statistical models are used to evaluate the transition probabilities between states and the observation likelihoods over features. Over the years, people have developed and refined the ability to train the necessary models from large corpora of transcribed audio files. This training task can be performed offline using a computer cluster. The objective of this project is to accelerate the decoding task to make the system run in real-time.

The project exclusively aims at speeding up the evaluation of acoustic score for reducing the decoding time. In the thesis forward, the task of computing acoustic score is referred to as 'GMM Log Likelihood' computation to match with the function name used in the working code. As a baseline, the current working code with minimum word error rate indicates that GMM based acoustic score computation consumes around 65 percent of the total time taken for decoding the entire wave file. Apart from being the most compute intensive part in the decoding process, the main challenge in putting the GMM log likelihood module into hardware is the associated data transfer. The model file used for GMM log likelihood computation contains the values of a number of mixtures, inverse variance, mean inverse variance, etc. associated with a particular probability distribution which is pre-computed and stored. With the current model file used, each function call to the GMM module requires a transfer of around 2 Kilo-Bytes of data. So the primary focus of the project is to bring up techniques to speed up the data flow between the processor and FPGA's programmable region.

The FPGA used for implementing the hardware function belongs to **Zynq-7000** family which is based on Xilinx SoC architecture. These products integrate a feature-rich dual-core or single-core ARM® Cortex™-A9 based processing system (PS) and 28 nm Xilinx programmable logic (PL) in a single device. A major part of the project uses SDSoC™ Development environment for compiling the application code written in C language into fully functional Zynq-SoC. SDSoC™ makes it easier to implement the hardware as it combines the functions of Xilinx Vivado HLS, Vivado IP integrator, and SDK into its environment. Vivado HLS does the generates the RTL file for the code written in C/C++/openCV and does the synthesis. It also creates the IP for the block which is synthesized. Vivado IP integrator part integrates the IP thus created with the processor which in this project is a zynq processor. Xilinx SDK links the software part of the system running on the processor with hardware part implemented in the Programmable Logic(PL) region of the FPGA. So the SDK code regulates the data

transfer between Processing System(PS) and PL. By combining these three process into a single environment SDSoC™ provides an easy way to implement systems in hardware and this project makes good use of the same.

1.4 Organization of the thesis

The remaining chapters in this thesis are organized as follows:

Chapter 2 gives an insight into the concept and working of an ASR system from front-end feature extraction to back-end decoding. The details about various models used for speech recognition are also discussed briefly. The chapter also introduces the primary equation governing the GMM based acoustic scoring.

The specifications of the target FPGA is given in chapter 3. This chapter also gives details about the software development environment and the optimization methods that are used for constructing the GMM code. The performance results of the memory bandwidth test conducted to analyze the data transfer rate from external memory to hardware function are also mentioned in this chapter.

Chapter 4 includes the actual implementation methods of the GMM function. The chapter initially describes the architecture for the hardware function. This chapter also explains the optimization used in the code and the improvement in latency corresponding to each optimization.

Chapter 6 discuss the implementation of the hardware function using HLS and SDK with different interfaces. This chapter also compares the efficiency of hardware implementation between SDSoC and HLS-SDK.

Chapter 6 concludes the thesis by suggesting future improvements that can be added to this system.

CHAPTER 2

ASR BACKGROUND

2.1 Introduction

This chapter discusses the ASR system in detail and the relevant functions used in the C code for implementing the same. As mentioned in the previous chapter, ASR system has 3 main sections: *(1) front end feature extraction*, *(2) Acoustic scoring*, *(3) Decoding based on Viterbi search*. Figure 1.1 shows the conceptual illustration of an ASR system. The code for ASR is a simplified version of the open source Kaldi speech recognition tool kit. Kaldi provides an ASR system based on finite-state transducers (using the open source OpenFst), together with detailed documentation and scripts for building complete speech recognition systems. Kaldi is written in C++, and the core library supports modeling of arbitrary phonetic-context sizes, acoustic modeling with standard Gaussian mixture models as well as standard subspace Gaussian mixture models (SGMM).

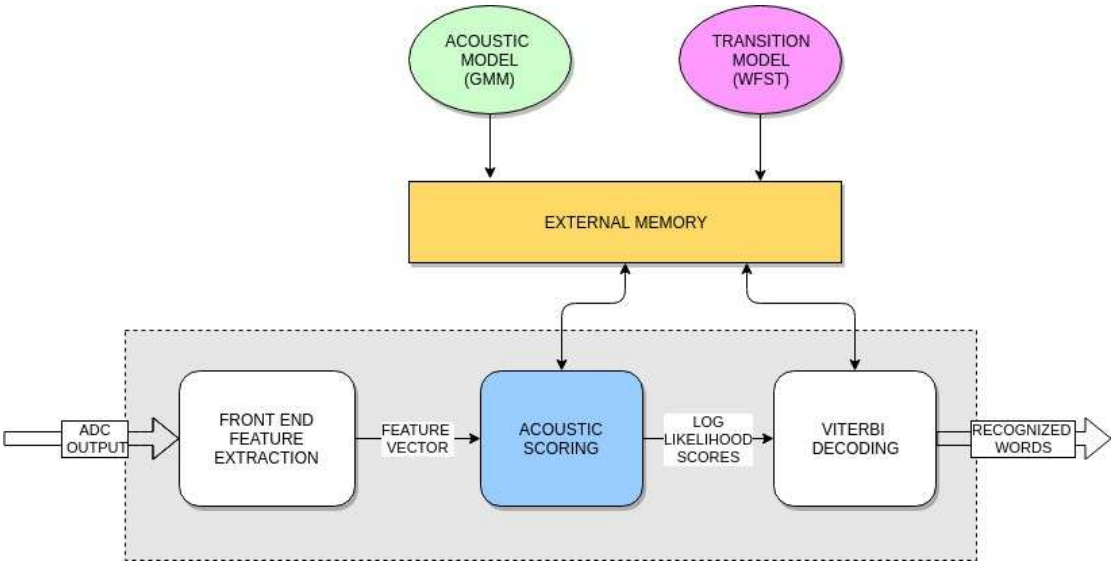


Figure 2.1: Conceptual Illustration of Speech Recognizer

2.2 Probability theory of ASR

The primary goal of an ASR system is to hypothesize the most likely discrete symbol sequence, out of all valid sequences in the language model(L) used, from the given acoustic input (F). As stated above, the input is treated as a set of discrete observations, such that:

$$F = f_1, f_2, f_3, ..., f_t \tag{2.1}$$

where f_t corresponds to feature vector representing a particular frame.
Similarly, the symbol sequence to be recognized is defined as:

$$W = w_1, w_2, w_3, ..., w_n \tag{2.2}$$

The fundamental ASR system goal can then be expressed as:

$$\widehat{W} = argmax P(W|F) \quad for \quad W \in L \tag{2.3}$$

This equation implies that for a given sequence W and acoustic input sequence F,the probability P(W|F) needs to be determined.
Using Bayes' theorem

$$P(W|F) = P(F|W)P(W)/P(F) \tag{2.4}$$

P(W) is defined as the prior probability for the sequence itself which is calculated based on the occurrence of a word sequence. So P(W) is determined by the Language model. Since the P(F) is the same for each sentence W, P(F) = 1
Thus equation 2.3 can be simplified as

$$\widehat{W} = argmax P(F|W)P(W) \tag{2.5}$$

The probability P(F|W) is the acoustic score.

2.3 Front-end feature extraction

The goal of the feature extraction step is to compute a sequence of feature vectors to clearly represent the given input signal. The 16 KHz audio signal coming from the ADC undergoes a series of signal processing steps to finally give out a 40-dimensional feature vector. The feature thus created should allow an automatic system to discriminate between different through similar sounding speech sounds, they should allow for the automatic creation of acoustic models for these sounds without the need for an excessive amount of training data, and they should exhibit statistics which are largely invariant across speakers and speaking environment. There are many feature representations in use, but the most common is the Mel -frequency cepstral coefficient (MFCC) feature set. The MFCC feature extraction process has many steps which are elaborated below.

As mentioned previously, the first stage of front end feature extraction part is framing. The signal is split into frames of 25 ms duration. This constitutes 400 samples of the input. The next framing starts after 10 ms i.e, after 160 samples which is referred to as the frameshift. A pre-emphasis stage can be added prior to framing so as to amplify energy in the high frequencies of the input signal and thereby allowing information in these regions to be more recognizable. The windowing stage splits the input signal into discrete time segments. A suitable windowing function is used to prevent edge effects. After windowing, DFT is applied to the windowed signal. Suitable FFT algorithm is used to speed up the feature extraction as well as using minimum hardware resource in order to save it for the highly resource demanding back-end.

Even though the resulting spectrum of the DFT contains information in each frequency, human hearing is less sensitive at frequencies above 1000 Hz. This has an impact on the accuracy and efficiency of the ASR system. Therefore, the spectrum is warped using a logarithmic Mel scale by passing the output of FFT block through the Mel Filter Bank(MFP). MFP is a bank of triangular filters constructed with filters distributed equally below 1000 Hz and spaced logarithmically above 1000 Hz. The output of filtering the DFT signal by each Mel filter is known as the Mel spectrum. A Mel frequency can be computed using equation 2.6. Taking the logarithm of this provides Mel spectrum coefficients. The final step in obtaining MFCC is performing a discrete cosine transform(DCT) on the Mel spectrum co-coefficients. The output of DCT is

Mel-cepstral coefficients of 13th order.

$$mel(f) = 1125ln(1 + \frac{f}{700}) \tag{2.6}$$

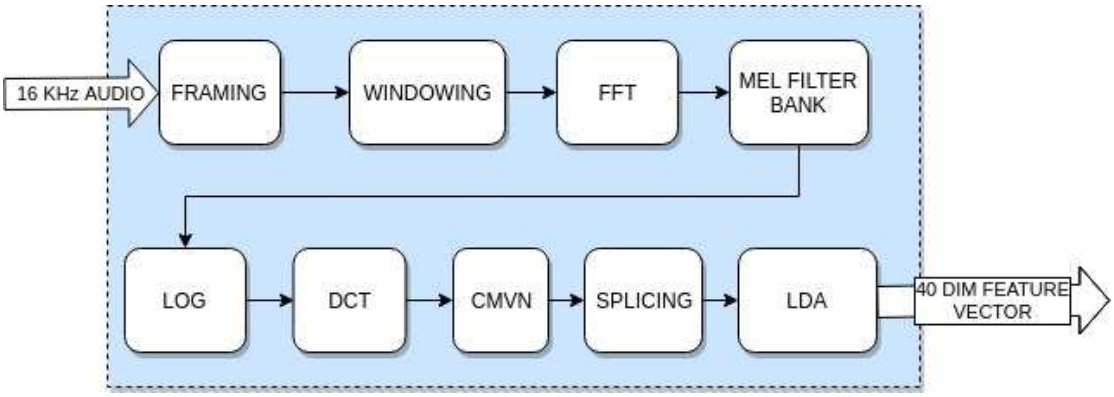


Figure 2.2: Block diagram of front-end

The three stages following the DCT is used to improve the feature vector so as to increase the efficiency of ASR. The Mel-cepstral coefficients are passed on to Compute Mean and Variance Normalisation(CMVN) stage. In this stage, the mean and variance of the cepstral coefficients are calculated and the feature vector is updated based on equation 2.7

$$feat\_out[i] = (feat\_in[i] - mean[i])/sqrt(var[i]) \tag{2.7}$$

To derive more general features for speech recognition linear discriminant analysis (LDA) is used. With LDA, using a hidden Markov model (HMM) states as classes have been shown to give improved recognition performance LDA combines features in several time frames into one reasonable size feature vector. Prior to LDA, the coefficients coming out of CMVN block are pushed to the splicing stage. Splicing stage combines the feature coefficients of 9 frames and gives out a 117-dimensional vector. The final stage does an appropriate matrix multiplication to compress this 117-dimensional vector into a 40-dimensional vector. Acoustic score computation takes this feature vector as one of its inputs along with the corresponding mixture parameters.

2.4 HMM-GMM scoring

Acoustic scoring is one among the two main sections in ASR back-end. In the proposed ASR system, Hidden Markov Model(HMM) framework is used to construct the decoding graph. The HMM framework requires modeling the dependencies between variables, specifically the transition model and emission model. The transition model incorporates information about the language, including its vocabulary and grammatical constraints. The emission model describes how observations vary depending on the unobserved state. In speech recognition, this is called the acoustic model. The observations in the acoustic model is the feature vector. An illustration of the HMM framework is shown in figure 2.3. The transition model is represented as $p(x_{t+1}|x_t)$ as it indicate the probability of reaching a state from a particular previous state. The acoustic model is represented as $p(f_t|x_t)$ and it indicates the probability of feature vector being emitted by a particular state.

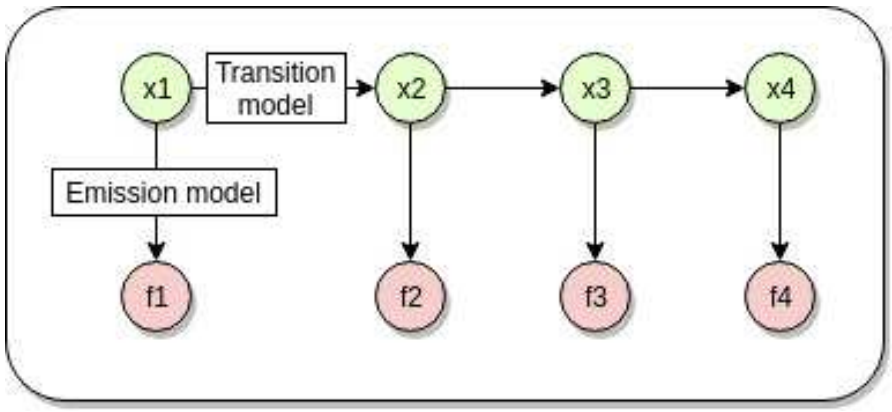


Figure 2.3: Illustration of HMM framework for ASR system

2.4.1 Transition model:WFST

In this system, Weighted Finite-State Transducers(WFST) are used to represent the transition from one state to another in the HMM framework. Similar to the Finite State Machine(FSM), WFST shows the path to the next state from a current state. The path from one state to another is called an arc. WFST arc contains information of input label, weights, and output label. Input label is the transition id from one state to another, weights indicate the cost associated with the arc and output label gives the phoneme given by the transition path.

The transition model used in the proposed ASR system is a composition of 4 WFSTs collectively known as **H C L G** FST.

- **H** FST contains the HMM information about the transition ids and its mapping to context-dependent phones given by the output label of the FST arcs.
- **C** FST represent the context dependency. It contains information about mapping context dependent phones to mono-phones.
- **L** FST is also called Lexical FST. This maps the phones to words.
- **G** FST is the grammar FST. This contains information related to the grammatical correction of words.

WFST model used in our system is a speaker independent model and it has 237925 states and 580605 arcs together giving out a vocabulary of around 7000 words. The WFST model is stored in the external memory and can be replaced with a much efficient model in the future.

2.4.2 Emission model: GMM

This section gives the theoretical basis for the function whose hardware implementation is being focused in the project. As mentioned earlier, the acoustic likelihood gives the likelihood of a feature being emitted by a state given the hypothesis for the state. The acoustic likelihood in this system is calculated based GMM. GMMs used in ASR are typically limited to diagonal covariance matrices to diagonal covariance matrices to reduce the number of parameters. Diagonal GMMs can be efficiently evaluated in the log domain, using a dot product for each component followed by a log-sum across components.

$$p(f_t|x_t) = \sum_{m=1}^{N} \frac{W_m}{(det(\Sigma_m))^{0.5}(2\pi)^{D/2}} \times exp(-\frac{1}{2}(f_t - \mu_m)\Sigma_m^{-1}(f_t - \mu_m)^T) \quad (2.8)$$

Where N indicate the number of mixture components corresponding to a particular probability density function(PDF) and D indicate the dimension of input feature vector which here is 40. Σ_m is the diagonal co-variance matrix.
The equation 2.8 can be rearranged as:

$$p(f_t|x_t) = \sum_{m=1}^{N} A_m exp(\sum_{d=1}^{D} -\frac{1}{2}\frac{(f_{t,d} - \mu_{m,d})^2}{\sigma_{c,k}^2}) \quad (2.9)$$

$$where\ A_m = \frac{W_m}{(det(\Sigma_m))^{0.5}(2\pi)^{D/2}} \tag{2.10}$$

This can be further simplified as:

$$p(f_t|x_t) = \sum_{m=1}^{N} exp(g_m \sum_{d=1}^{D} -\frac{1}{2}\frac{(f_{t,d} - \mu_{m,d})^2}{\sigma_{m,k}^2}) \tag{2.11}$$

where g_m = Log(A_m) is called gconst value of a particular mixture
To compress the parameters to be stored, we take the log of the acoustic likelihood. So the final equation of the function which is targeted on hardware becomes:

$$Log(p(f_t|x_t)) = \sum_{m=1}^{N} g_m + \sum_{d=1}^{D}[\frac{(f_{t,d} \times \mu_{m,d})}{\sigma_{m,k}^2} - \frac{f_{t,d}^2}{2\sigma_{m,k}^2}] \tag{2.12}$$

The parameters gconst , weights (W_m) , inverse variance ($\frac{1}{\sigma_{m,k}^2}$) and mean inverse variance($\frac{\mu_{m,d}}{\sigma_{m,k}^2}$) corresponding to a particular mixture in a PDF is pre-computed and stored in external memory. The GMM log likelihood function when called requires the transfer these associated data from external memory to the PL and this makes the GMM module most compute intensive as the function is called thousands of times for a particular frame.

Generally, if we have 'X' PDF distributions, an average of 'Y' mixture components per distribution, and feature vectors with dimension D, the total number of GMM parameters will be:

$$XY(1 + 2D) \tag{2.13}$$

In the acoustic model used for this project, X = 1532, Y = 6,and D = 40. With single-precision floating point values, the model has a size of 8.8 MB. Applying various test wave files, it is found that, there are around 2000 GMM function calls on an average per frame which requires a transfer of around 4 MB (as per equation 2.13) of data from external memory to PL. Since the frameshift is 10 ms, the rate of GMM evaluation is 100 frames per second. This constitutes a memory bandwidth of 400 MB/s. Suitable FPGA with proper optimization technique is used to speed up the data transfer.

2.5 Viterbi Decoding

The decoding used in ASR is based on the Viterbi algorithm. The search has both forward pass and backward pass. In the forward pass, we construct the possible states depending on previous states and the acoustic model. The backward pass does the traversing from the final state to the initial state through the most likely path and gives out the words uttered in the particular speech. The search begins with an empty hypothesis and uses the information of the from the feature vector and HMM framework to develop a set of active hypothesis. The hypothesis is represented by the states in HMM and contains information about the input label, output label, cost associated with the hypothesis and the previous state. When a new hypothesis with sufficiently high likelihood is found, it will be saved. As shown in figure 2.4, the forward pass has four stages: *(1),Hypothesis fetch*, *(2),Arc fetch*, *(3),GMM log likelihood evaluation*, *(4),Pruning*.

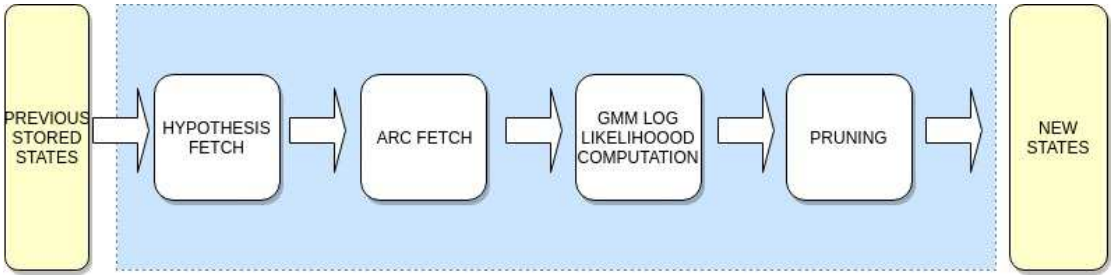


Figure 2.4: Forward pass of Viterbi decoding

The first stage of decoding search is the hypothesis fetch. To start with the decoding for a particular frame, the data that we have as input is the set of active hypothesis stored after the decoding of the previous frame and the feature vector corresponding to the current frame. The hypothesis is represented by the state id. So we fetch the states one by one. For each state fetched, we fetch the arcs going out of the particular state from the WFST model. This makes the second stage. For each arc, we retrieve all the parameters which include input label, weights and output label and pass it to the GMM log likelihood stage for the computation of the final acoustic score of the new hypothesis.

The first three stages produce a large number of hypothesis out of which most of them are very unlikely. Storing all the hypothesis would require much higher memory and also increases the energy cost. In order to limit the number of hypotheses, we add a final pruning stage to the decoding process. In the pruning stage a suitable beam of

the score for the hypothesis is set and those hypotheses with score beyond the beam are neglected. The remaining hypotheses are pushed into active hypothesis list which will be fetched in the search for the next frame.

The backward pass is the trace-back function. From the most likely hypothesis after the decoding, the trace-back function traverses through the respective parent hypothesis until it reaches the initial hypothesis. During trace-back, the output label of arcs in the most likely path gets stored. The sequence of output labels is then mapped to get the appropriate words based on the model file and vocabulary. In our system, initially, the trace-back function was run when the last frame was decoded. But in the later stage of the project trace-back was done after a set of frames gets decoded and not waiting for the entire decoding to complete. This was done to limit the storage and to give out words in pace with the decoding process.

2.6 Decoding code structure

As mentioned in the previous section, the decoding process constructs a set of active hypothesis. In the software implementation of the same, we use a structure to represent each hypothesis and its referred as Token.

```
struct Token {
    int state_;
    float cost_;
    int olabel_;
    Token *prev_;
};
```

The Decoder function used in the code is shown below:

```
int Decoder_Decode(Decoder * Dec, float* features)
{
    Token_ClearToks(prev_toks);
    (cur_toks).swap(prev_toks);
    Decoder_ProcessEmitting(Dec, features);
    Decoder_ProcessNonemitting(Dec);
    Decoder_Prune(Dec, cur_toks);
    return (cur_toks).size();
}
```

In the code, cur_toks and prev_toks are used to store the list of pointers to Tokens. prev_toks has the address of Tokens available after decoding the previous frame. cur_toks store the address of Tokens which are newly generated. In the initial stage of the project, both cur_toks, and prev_toks were implemented as C++ Map data structure. Later on, these were changed to arrays for hardware implementation. The size of these maps is limited to 10000 for reducing the memory footprint.

Decoder_ProcessEmitting(), Decoder_ProcessNonemitting() and Decoder_Prune() are the 3 main sub-functions of decoder function. In this Decoder_ProcessEmitting() handles the first three stages of the forward pass. It fetches Tokens from prev_toks, gets the arc related to each Token, calls GMM log-likelihood function to compute the acoustic score for all the arcs whose input label is non zero and push new Token into cur_toks. Decoder_ProcessNonemitting() is similar to Decoder_ProcessEmitting() but it fetches Token from cur_toks and it evaluates only arcs with input label zero. This function is used to merge the states whose transition between them does not give out a phoneme. Since Decoder_ProcessNonemitting() does not intend to give out phonemes, it does not call the GMM log-likelihood function.

Decoder_Prune() handles the pruning stage in the Viterbi search flow. The Tokens in cur_toks after Decoder_ProcessNonemitting() is searched and those Tokens which are above a certain cutoff is eliminated. The cutoff is set based on the minimum cost of all Tokens in the list plus a beam width. Beam width is adjusted based on accuracy and memory requirement. In this project, the beam width is set to 12. The final filtered Tokens are then stored back to cur_toks. This is then swapped to the prev_toks list for the next decoding.

CHAPTER 3

HARDWARE AND MEMORY BANDWIDTH

3.1 Introduction

From the theory of ASR given in the previous chapter, it is clear that the GMM log-likelihood function is a compute-intensive function as it evaluates the acoustic score based on equation 2.12. So to put this function in hardware turns out to be the obvious option for speeding up the decoding process. This chapter discusses about the hardware being used to implement the GMM log-likelihood function. There are several optimization techniques that help to speed up the computation and data transfer. Both SDSoC™and HLS has its own pragmas which bring out those optimizations. This chapter discusses those optimization techniques which are used specifically for this project. Apart from being compute-intensive, the major challenge in putting the GMM log-likelihood function in hardware is the associated data transfer. To get an estimate of the memory bandwidth required to successfully implement GMM function, a memory bandwidth test is conducted on the target hardware. The performance result of this test is analyzed and used as a reference for the actual function.

3.2 FPGA specifications

The target FPGA used for implementing the GMM function is ZebBoard™. ZebBoard™is a complete development kit whose architecture is based on Xilinx Zynq®-7000 all programmable SoC. The board contains all the necessary interfaces and supporting functions to enable a wide range of applications.Basic features of ZebBoard™ are shown in table 3.1.

The bigger goal of doing this project is to put the entire ASR decoding into hardware. This will require more resources than that is available in ZedBoard. So GMM function is also tested on ZCU102 board. ZCU102 architecture is based on Zynq UltraScale+ XCZU9EG-2FFVB1156E MPSoC. ZCU102 has PS clock frequency of 1200

MHz for first grade systems. The PL section features of ZCU102 as shown in table 3.2.

Section	Features	Description /count
	Part number	XC7Z020-CLG484-1
Processing system	Processor core	Dual-core ARM Cortex-A9
	Clock Frequency	667 MHz
	On-chip memory	256 KB
	External memory	512 MB DDR3
		256 Mb Quad-SPI Flash
		External SD card support
Programmable Logic	Clock	100 MHz oscillator
	Flip-flops	106400
	LUTs	53200
	BRAMs	140 (36 Kb blocks)
	DSP slices	220

Table 3.1: Basic features of ZedBoard

Features	Description /count
clock	Oscillator frequency min=100 MHz max=600 MHz
Flip-flops	548160
LUTs	274080
BRAMs	912 (36 Kb blocks)
DSP slices	2520

Table 3.2: PL section features of ZCU102

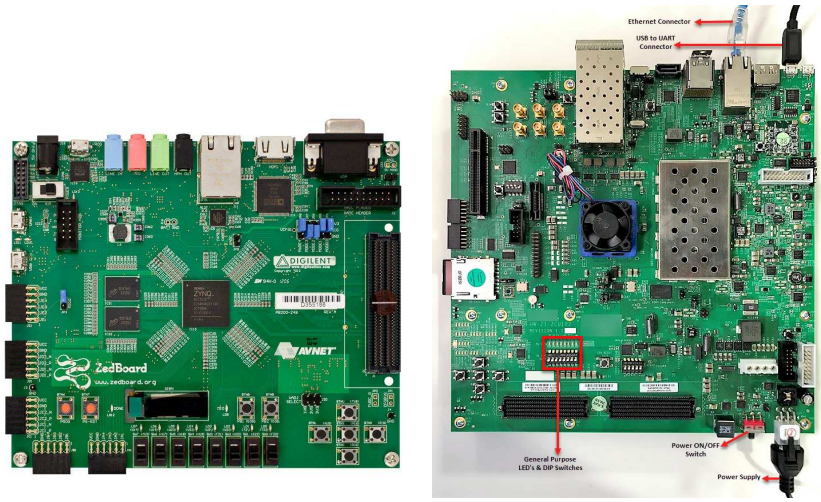


Figure 3.1: (a)Zed Board and (b) ZCU102

3.3 Optimizations in SDSoC and Vivado HLS

As mentioned in the Introduction chapter, the code for implementing in hardware is written in C++ and SDSoC does the high-level synthesis to get the RTL code for the hardware. Optimizations in hardware are generally made with proper construction of RTL code. Since we use HLS, the method to make necessary changes in RTL for hardware optimization is by specifying Pragmas. Vivado HLS specifies a set of $\#pragma$ directives that are used within the hardware function. Since we use SDSoC™ environment to compile and build the code, some of the HLS pragmas cannot be compiled. These include pragmas associated with function argument interfaces. To substitute those pragmas SDSoC has its own pragams which does the corresponding optimization. All the HLS and SDx(referring SDSoC) pragmas and their use are understood based on the reference papers (4), (5), (6), (7). Some of the pragmas which are relevant to this project will be detailed in the next section of this chapter as well as in the next chapter. The General approach to optimize and accelerate a hardware function is :

- Use HLS pragmas within the hardware function to accelerate the execution.
- Use SDx pragmas to accelerate the data transfer between PS and PL.

3.4 Estimation Methodology

For a function accelerated in hardware, SDSoC allows estimation of 2 major parameters: Latency and Resource Allocation.The $Estimate\ Performance$ option on the project home page can be used to estimate the resource allocated as well as latency for the execution within the hardware function. The reports generated shows the percentage utilization of resources and latency of computational loops in the hardware function. In certain cases, where the number of times a computational loop gets executed in hardware is determined during run time, SDSoC cannot estimate latency and $Estimate\,Performance$ fails to generate reports. In such cases, we can use the option to launch the successfully built project into Vivado HLS where it shows the performance estimation reports. Even in this case, the latency is not estimated if the loop count is not determined during compile time but an approximate estimation of latency

can be done using $pragma\ HLS\ LOOP\_TRIPCOUNT$ where it allows to specify minimum, maximum and average number of times the particular computational loop is executed.

The latency estimation reports does not show the time taken for the transfer of data from PS to PL and vice versa. One way to estimate the data transfer time is to analyze the data motion reports generated while building the project. Data motion reports gives information about the interfaces associated with each function argument and the number of Arm CPU cycles taken for the data transfer. But a more accurate way to measure the execution time of functions is to use special SDSoC API calls that measure activity based on the free running clock of the Arm processor. These functions can be used to log the start and end times of a function. In this project we make use of a class defined in $sds\_lib.h$ and having those API functions to estimate the latency. The code segment for the same is shown below:

```
class perf_counter
{
public:
uint64_t tot, cnt, calls;
perf_counter() : tot(0), cnt(0), calls(0) {};
inline void reset() { tot = cnt = calls = 0; }
inline void start() { cnt = sds_clock_counter(); calls++; };
inline void stop() { tot += (sds_clock_counter() - cnt); };
inline uint64_t avg_cpu_cycles() { return ((tot+(calls>>1))/calls);};
};
```

The function whose latency is to be estimated is placed between the start() and stop() function calls and the sds_clock_counter() counts the number of CPU clock cycles. The function avg_cpu_cycles() taken by the function in each call.

3.5 Memory Bandwidth test

Theoretically estimating that the data transfer in GMM log-likelihood function will be the major bottle-neck compared to computation, a sample hardware function is built on hardware to test the latency. The results and analysis from this section used for optimizing the GMM log-likelihood function. Detailing about to the data movement

and the relevant pragmas used are also discussed in this section.

A set of test wave files were used for testing the entire ASR code. In order to reduce the decoding time, a 5-second waveform named $5-secwave$ was used for testing the GMM log-likelihood function in an isolated code. Before the final optimizations in the actual implementation of GMM log-likelihood function, the number of function calls to GMM function was estimated to be around 1 million for decoding $5-secwave$. So a test code reading the equivalent amount of data from PS and being called 1 million times is implemented for memory bandwidth test. The initial test code is shown below:

```
#pragma SDS data zero_copy(ar1[0:500])
#pragma SDS data mem_attribute(ar1:PHYSICAL_CONTIGUOUS)
void func_hw(float ar1[500],float* le1){
    float read_array1[500];
    for(int i=0;i<500;i=i++){
#pragma HLS PIPELINE
        read_array1[i]=ar1[i];
    }
    *le1 =read_array1[499];
}
```

The function $func\_hw$ reads 500 words(32 bit data) from PS. The data size is fixed in accordance with the actual GMM function data transfer. The latency estimation results of a single call to $func\_hw$ are shown below:

Code Description	CPU cycles	HW cycles in ZedBoard
Function Overhead	11000	1650
Without pipelining the loop	17000	2550
Pipelining the loop	14500	2175

Table 3.3: Performance estimate for single function call of func_hw

Hardware cycles corresponding to the CPU cycles is computed by multiplying the ratio of (PL clock frequency)/(PS clock frequency) to the CPU cycles. For ZedBoard with PL frequency 100 MHz, the multiplicating factor is 0.15

To estimate the number of cycles taken for 32-bit transfer, the function overhead should be subtracted from the actual number of cycles shown. The code above has

implemented case 3 of table 3.3. The data transfer cycle estimate is as follows:

$$500 \times 4\ Byte \longrightarrow (2175 - 1650)\ cycles$$

$$4\ Byte \longrightarrow 1.05\ cycles$$

Therefore with the optimizations used in the code almost 1 word per cycle transaction occurs from PS to PL. This is derived neglecting function overhead.

3.5.1 Function Overhead

When a C++ program executes the function call instruction, the CPU stores the memory address of the instruction following the function call and it copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of the function is less than the switching time from the caller function to called function. In SDSoC, if the main function is calling a hardware function, this function overhead is observed to be more profound as it deals with both PL and PS addresses. A function overhead estimation test was done to estimate the time consumed for function overhead while calling a hardware function. The results of the same are shown in table 3.4. It is also observed that the function overhead

Number of function calls	**Average CPU cycles**
1	10000
100	3000
500	1000

Table 3.4: Overhead for calling function with 1 scalar argument

for the same number of function calls increase if the number of arguments increase. Also for large number of function calls having single scalar argument, the function call overhead is observed to saturate at around 1000 CPU cycles. Function call overhead is also based on internal caching and related cache flushing. So to accurately estimate overhead is challenging.

3.5.2 Data zero_copy and Pipelining

Referring to the optimizations mentioned in section 3.3, 2 optimization methods are implemented in the test code for memory bandwidth. SDS Data zero_copy pragma for data transfer and HLS pipeline pragma for the loop inside the hardware function. By default, SDSoC assigns data movers for function arguments by its own. These data movers transfer data between PS and PL and among PL. Usually, for scalar argument, SDSoC assigns AXI_LITE data mover. For Array arguments, there are different data movers available based on the size of data being transferred. The list of data movers and their features is mentioned in reference (4). As the size of the data transferred by an array argument gets larger, the conventional data transfer which happens with an address fetch followed by data fetch will result in large latency. All the data movers which transfer data from PS to PL has an associated address fetched from the external memory for each data and thus does not pose an efficient approach to transfer data in large volume. The best solution to overcome this to use Data zero_copy data mover.

```
1 #pragma SDS data zero_copy(arg[0:SIZE])
2 #pragma SDS data mem_attribute(arg:PHYSICAL_CONTIGUOUS)
```

Data zero_copy generates an AXI master interface through which data gets transferred directly from external memory to PL. AXI master interface supports burst read and writes into and from the PL to external DDR memory. While reading an array from the DDR to PL, the AXI master sends the starting address of the array location in DDR along with the burst information. Therefore, the address transfer happens only once and data is transferred continuously based on the burst limit. AXI3 master interface supports a maximum burst of 16 words and AX14 interface supports a maximum burst of 256 words. By default, the zynq processing system has AXI3 master interface. But the IP integrator within SDSoC places the required AXI interconnect between the accelerator hardware IP and the Zyng processor IP so as to make it work on AXI4 standards.

For data zero_copy to work, the argument passed from the main function should have contiguous memory location in DDR. Since the conventional dynamic allocation using $malloc$ does not allocate contiguous memory, we use dynamic allocation using sds_alloc defined in sds_lib.h for array declaration. Apart from allocating contiguous memory location using sds_alloc, the SDS data mem_attribute pragma should also be added to specify the compiler about the nature of array location.

The data transfer is optimized using the methods discussed above. The accelerate the execution of the loop within the hardware function HLS pipeline pragma is used. As the name suggests, it pipelines the operation within the loop reduces the initiation interval f by allowing the concurrent execution of operations. Initiation interval(II) is the number of clock cycles after which new input can be processed. The ideally for achieving maximum speed up, II targeted is 1. But depending on the complexity of operations II increases and the code should be restructured to reduce the II. In the test code, pipeline reduces the latency to almost half the value(neglecting overhead). Depending on the code, there are other pragmas that can be used for optimization, reference (6).

3.5.3 Modified test

Since the data transfer rate is shown above(1 word per cycle) is obtained neglecting the function overhead, the actual time taken may further increase. Since the bigger goal of this project is to push the entire ASR decoding into hardware, the actual hardware top function being called from the main function is $Decoder\_decode()$ mentioned in section 2.4. For the test wave $5 - secwave$ of 5-second duration, $Decoder\_decode()$ is being called 500 times from the main function and each top function calls GMM log-likelihood function 2000 times(this count is before the final optimization) on an average. Since the function overhead appears only for the top function call, the overhead for calling GMM function is negligible. To simulate the memory bandwidth test based on this kind of GMM function call, the actual code which reads data from external memory is being called from another hardware function. The top hardware function is called 500 times from the main function and the function reading a large volume of data is called 2000 times from the top function to make it 1 million calls to match with actual GMM function call. The performance results are shown in table 3.5.

From table 3.5, it can be observed that the same amount of data when reading into a single array is taking double the time compared to when the data is read into 2 arrays of half the size in the same loop. This optimization came as a result of pipelining. The pipelining limits this optimization advantage up to 2 simultaneous reading as reading into 4 arrays of one fourth the size is observed to give the same result as reading into 2 arrays.

Number of calls to inner hw function	Method of reading data(2KB)	Average hw cycles (ZedBoard)	Time taken (ZedBoard)
500×2000	Read into single array of size 500 using for loop	1075200	5.44 sec
500×2000	Read into 2 arrays of size 250 using for loop	615327	3.12 sec
500×2000	Read into 4 arrays of size 125 using for loop	615116	3.09 sec
500×800	Read into 2 arrays of size 250 using for loop	246406	1.25 sec
500×800	Read into 2 arrays of size 250 using memcpy	477450	2.41 sec

Table 3.5: Performance results of modified memory bandwidth test

```
    float read_array1[5][50];
    float read_array2[5][50];
    for(int i=0;i<5;i++){
        for(int j=0;j<50;j++){
#pragma HLS PIPELINE
            read_array1[i][j]=ar1[i*50+j];
            read_array2[i][j]=ar2[i*50+j];
        }
    }
```

Since reading 2KB data into 2 arrays prove to be the most optimized method, the above code segment is used in the modified memory bandwidth test. The data transfer rate calculated based on this optimization is as follows:

$$2000 \times 500 \times 4\ Byte \longrightarrow 615327\ cycles$$

$$4\ Byte \longrightarrow 0.61\ cycles$$

So pipelining and reading simultaneously into 2 arrays gives a data transfer rate of around **2 words per cycle**.

The last 2 observations in table 3.5 are done based on the data in accordance with the actual GMM log likelihood calls. With efficient caching, the average GMM function calls per frame(for the $5 - secwave$ test wave)can be limited to 800 instead of 2000 for a particular frame. Memory bandwidth test is also done using $memcpy()$ function. Since pipelining optimization cannot be applied in $memcpy()$, the data transfer rate is restricted to around 1 word per cycle rather than 2.

CHAPTER 4

IMPLEMENTATION OF ACOUSTIC SCORING

4.1 Introduction

The hardware implementation of the actual GMM log-likelihood function and associated challenges are being discussed in this chapter. The memory bandwidth test results have put an insight into the approach that should be taken to overcome the data transfer bottleneck. Some of the basic optimization methods within the hardware function are also discussed in this chapter and the improvement in latency achieved by applying relevant optimization is listed out. The architecture of the GMM log likelihood computation block is described in section 4.2 and the base code is constructed based on the specified architecture. The table below shows the transition model and the GMM model details that are used in this project.

Data	**count**
Number of transition-ids	11582
Number of PDF-ids	1532
Maximum number of mixtures for a PDF-id	18
Minimum number of mixtures for a PDF-id	5
Average number of mixtures for a PDF-id	6

Table 4.1: Transition model and GMM model features

4.2 GMM Log-Likelihood function architetcure

GMM log-likelihood function is constructed based on equation 2.12. The input to module includes input feature vector and the mixture values corresponding to the particular PDF. The mixture data includes: *1, Number of mixtures*,*2, Inverse Variance*, *3,Mean Inverse Variance* and *4, Gconsts*. These mixture data are pre-computed and stored in the model file based on equations in section 2.4.2. As mentioned in table 4.1, the maximum number of mixtures for a particular PDF is 18. So the maximum size of

the gconsts array is 18 and inverse variance, mean inverse variance array is 18 X 40. Since the average number of mixtures is 6, around 1 KB (6 X 40 X 4 Byte) data is being transferred by both inverse variance and mean inverse variance fetch which constitute a 2 KB data fetch. The architecture for implementing the GMM function is shown in figure 4.1.

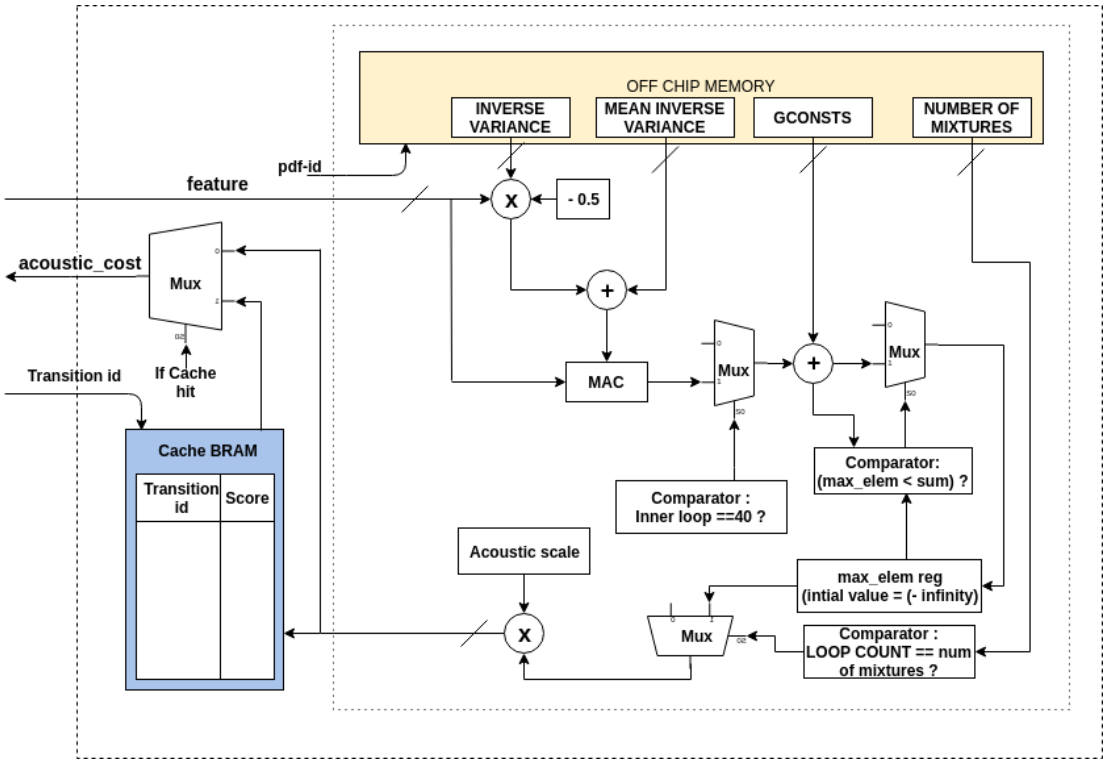


Figure 4.1: GMM Log-Likelihood function architetcure

4.2.1 GMM score caching

The main function sends in feature vector and transition-id(FST arc input label) to the GMM log likelihood module. Since for a particular frame, different arcs can have the same input label and the PDF associated with a particular arc input label is always same, we do not need to compute the GMM score every time. Instead, the score is cached if the transition-id is hit for the first time and later when the same transition-id is hit, the value from the cache BRAM is sent back. This reduces the average GMM evaluation time. For the test wave(5-sec wave) GMM computation without caching occurs 3 million times whereas with caching it reduces to around 1 million. This clearly indicates around 33% reduction in GMM evaluation time. Since the caching happens for a particular frame, the frame number is also an input to the GMM module.

4.3 Implementation and Results

From the architecture shown in the previous section, a baseline C++ code which runs with maximum speed is constructed. This code is modified in different steps to implement it efficiently in hardware with maximum possible speed up to have real-time speech decoding and ensure no reduction in word error rate(WER). Since there are a set of mixtures associated with a particular PDF, each PDF is represented as a structure in C++ and the entire 1532 PDFs were stored as an array of structs in the external DDR memory. Entire ASR code which was initially written based on map data structure was changed to arrays considering hardware compatibility. The table below shows the time taken by the entire ASR code run on ZedBoard and ZCU102 Arm processor(software) to decode 5-sec wave and the time consumed in GMM computation.

Description	**Time taken(ZedBoard)**	**Time taken(ZCU102)**
Entire decoding	107 sec	41 sec
GMM evaluation alone	24 sec	9 sec

Table 4.2: Decoding time for 5-sec wave in Arm software

The Arm frequency of ZCU102 is almost double the frequency as that of ZedBoard. So it takes almost half the time taken by ZedBoard. When arrays where used, the process of storing the best hypothesis were done by linear search and it consumes a bigger part of the decoding time. This search mechanism is finally implemented in an efficient using binary search tree and as a result, GMM evaluation was observed to be the most time-consuming section with 60-65 percent of total time. So the final target of this project is to implement GMM so as to get a computation time of around 65 percent of total wave duration. For the test wave 5-sec wave, this time is 3.25 sec.

Initially, when the GMM function was put into hardware without any optimization, it took around **22 minutes** for an entire evaluation. In this case, SDSoC allocated default data movers which do not do burst read and thus resulted in such large time. referring to the optimizations mentioned in chapter 3, modifications are made in the implementation giving out betterment in performance.

4.3.1 Data transfer through AXI Master

From the results obtained from the memory bandwidth test, it can be inferred that the best way to transfer a large volume of data is through the AXI master interface. Since hardware function cannot access globally declared variables or arrays, the coding was done such that along with feature vector and transition id, the corresponding PDF data is also passed from the main function to the hardware function. All data was stored in contiguous locations using sds_alloc. Even though contiguous locations were used to store the PDF structure data, the compiler does not identify it as physically contiguous locations. To overcome this problem, each mixture- data was stored in large contiguous arrays and was accessed based on offsets which are determined by PDF ids. Data zero_copy pragma is applied to each argument so that the data gets transferred through AXI master interface and support burst read. Proper pipelining techniques are used within the hardware function to achieve minimum II for the computational loop. The performance results based on this optimization when the hardware is implemented in ZedBoard is shown in table 4.3.

Performance description	**Results**
Number of GMM function calls	3 million
Average Hardware cycles for execution	1320
Average Function overhead hardware cycles	750
Total execution time	39.6 sec
Total Function overhead time	22.5 sec

Table 4.3: Performance results for 5-sec wave after first optimization

4.3.2 Caching outside the hardware function

From table 4.3 it is clear that the major part of the execution time is being consumed by the function overhead. From the architecture shown in the previous section, it is clear that the GMM computation is done only around 30-35 percent of the actual number of functions. The remaining call gets the GMM score from the cache memory. But the overhead for calling the hardware function is same whether the final score is obtained using computation or from the cache BRAM. So to avoid this unnecessary function overhead that occurs when caching, we pull the caching block outside the actual hardware function. In the ASR code, $Decoder\_ProcessEmitting()$ calls GMM function. So we place the caching inside this function. So the architecture gets confined to just

the computation block. Even at this point transition id is used to cache the GMM score. The performance results based on this optimization is shown in table 4.4.

Performance description	Results
Number of GMM function calls	1 million
Average Hardware cycles for execution	2385
Average Function overhead hardware cycles	633
Total execution time	23.85 sec
Total Function overhead time	6.33 sec

Table 4.4: Performance results for 5-sec wave after second optimization

Since each function call gets the score by actual computation, the average hardware cycles are observed to increase. But the number of calls has now reduced to 1 million and therefore the reduction in total execution time is justified.

4.3.3 Burst reading into BRAMs

Apart from allocating contiguous locations for the arguments(feature and mixtures) and using Data zero_copy, the data transfer gets enhanced if the inputs are first read into on-chip memory. Since the size of the input is more than 128 bytes, the input gets stored in on-chip BRAMs. BRAMs are 36 Kb memory blocks and serve as the data storage space for the hardware. As mentioned earlier, the number of 140 BRAMs in ZedBoard which constitute around 5 MB of storage. Reading into this BRAMs is similar to the code shown in the memory bandwidth test. Even though this method is expected to increase the BRAM utilization and take additional loops for just reading the input data, the actual data transfer time is found to decrease. The performance results are shown in table 4.5.

Performance description	Results
Number of GMM function calls	1 million
Average Hardware cycles for execution	1653
Average Function overhead hardware cycles	633
Total execution time	16.53 sec
Total Function overhead time	6.33 sec

Table 4.5: Performance results for 5-sec wave after third optimization

With function overhead remaining as in the previous optimization, the actual execution time is found to reduce by 6 seconds when the data is initially read into BRAMs before using it for computation.

4.3.4 Calling from another hardware function

Even when the caching is pulled out of actual hardware function and burst reading method is used for data transfer, the function overhead is still found to take around 40% of total time. One way to overcome this problem is to limit the number of calls from the main function without actually affecting the output. In ASR code, the GMM function is called by $Decoder\_ProcessEmitting()$ which is called once per frame. So the actual top function call is around 500 for 5-sec wave and average GMM calls per frame are 2000. So we use this kind of coding method to isolate and test GMM function. To make 1 million calls to the hardware function, we call a top function once per frame from the main function and this top function calls the GMM function. Since the number of calls from the main function is much less compared to actual GMM calls, the effect of function overhead is negligible. Also, the caching code segment can be put in the top hardware function so in actual ASR code, caching is done within $Decoder\_ProcessEmitting()$. The performance results are shown in table 4.6.

Performance description	**Results**
Number of Top function calls	498
Average GMM calls inside top function	2000
Total execution time	11.57 sec
Total Function overhead time	10ms

Table 4.6: Performance results for 5-sec wave after fourth optimization

4.3.5 Caching using PDF-id

To further optimize the GMM evaluation time, the caching method was improved. Initially, the GMM score caching was done based on transition ids for a particular frame. But the actual GMM computation is done using mixture data associated with a particular PDF. From the model file, as there are only 1532 PDF ids for 11582 transition ids, it is evident that different transition id can have the same PDF and thus for a particular frame, different transition id which maps into the same PDF gives out the same GMM score. So the basis for caching can be changed to checking whether the PDF id is hit. Comparing the number of transition ids and PDF ids present in the model, it seems to give a reduction in GMM calls and thus total evaluation time reduces. Another advantage of this method is the reduction in BRAM utilization. Since the caching is done in

top function, cache data is stored in BRAMs. Since the size of the array used for cache data decreases from 11582 to 1532, BRAM count is observed to decrease significantly. Performance results are shown in table 4.7.

Performance description	Results
Number of Top function calls	498
Average GMM calls inside top function	800
Total execution time	4.3 sec
Total execution time(ZCU102:400 MHz)	1.8 sec
Total Function overhead time	$\tilde{1}$0ms

Table 4.7: Performance results for 5-sec wave after fifth optimization

4.3.6 Other Optimizations

Apart from the major optimizations discussed in previous sections, some minor optimizations are also used in the implementation to improve efficiency. Feature vector caching is one among them. For a particular frame, the feature vector used for computation of acoustic score is the same. Since the synthesized hardware module accepts feature vector in every call, it reads feature data every time. This puts additional unnecessary data access time though it is much lesser compared to the actual data transfer. So in order to stop this redundant read, the following code segment is used:

```
1     static float features_[40];
2     static int fr_no;
3     if(fr_no!=frame){
4     for(int_ i=0;i<40;i++){
5 #pragma HLS PIPELINE
6           features_[i]=features[i];
7     }
8     fr_no=frame;
9     }
```

The frame number is also passed to the hardware function which uses a local variable to account for the change in frame number. As per the code, the input feature vector is read only when the frame number changes, i.e, only at the start of the new frame. This saves the time for transfer of 40 X 4-byte data in every call.

To reduce the computation time, loop unrolling technique is used. Since the Gaussian mixture values have 2-dimensional data, the computations are implemented in nested loops. One way to optimize this is to unroll the loop to parallelize the operation rather than carrying out a single set of operation. This gives better latency but at the cost of more resource utilization. The code segment used for the following is shown below:

```
  for (int i=0; i<num_mix; i++) {
#pragma HLS loop_tripcount min=5 max=18
      .......
#pragma HLS PIPELINE
      for (int j=0; j<40;j++) {
        .........
      }
      .....
  }
```

HLS has its own pragma for unrolling, but it is also observed that when the pipeline pragma is used outside the inner loop, the inner loop gets unrolled completely creating parallel blocks. As a result of loop unrolling, the computation time is observed to be less than 50% of the time taken without unrolling. Figure 4.2 shows the hardware acceleration in ZedBoard obtained using the optimization methods described in previous sections for 5-sec wave

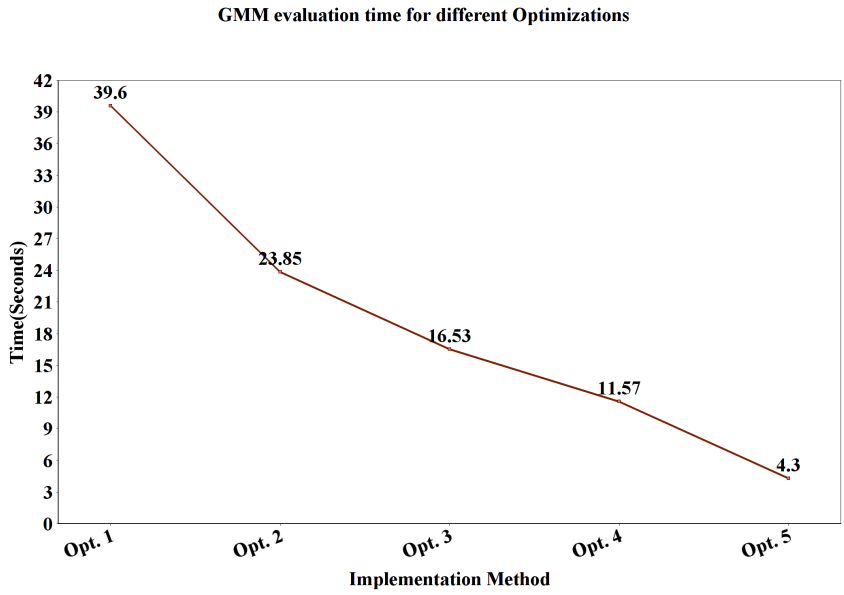


Figure 4.2: Hardware accelaration of GMM function for 5-sec wave

The above timings were obtained keeping the active tokens' count in ASR decoding as 10000. In the later stages of the project, Binary search tree and then binary heap techniques were used to get the best possible new tokens. Since the best tokens are arranged to the starting of the new active tokens list, this technique allows limiting the active tokens being processed without much degradation in WER. As the active tokens are limited, the actual GMM hardware calls are also expected to reduce. Performance results based on a different number of active tokens used are shown in table 4.8

Number of Active Tokens	WER	Total GMM evaluations	Hw calls after caching	Per frame avg. hw cycles	Time (sec)
8192	15.26%	2933742	401808	861020	4.28
4096	15.48%	2324695	397677	855300	4.25
2048	15.66%	1684930	370847	797490	3.97
1024	15.76%	997933	308454	662685	3.3

Table 4.8: Performance results after limiting the active tokens for 5-sec wave

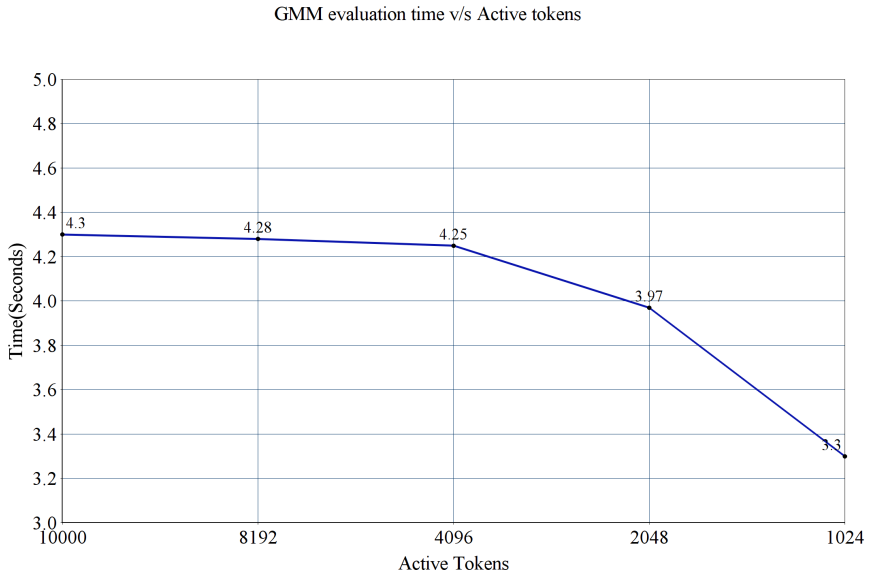
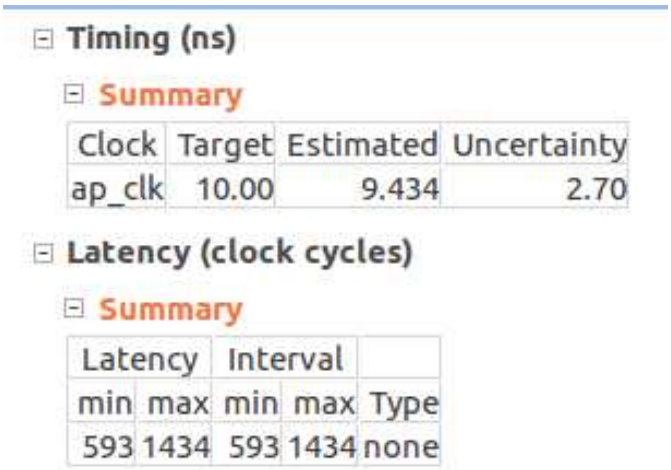


Figure 4.3: Change in GMM evaluation time with reduction in active tokens

When active tokens were limited to 8192, the results were similar to that obtained for 10000 tokens. When 4096 tokens were used, actual GMM evaluation count has reduced by 3 lakh but the number of hardware calls after caching is almost the same which resulted in comparable latency with 8192 tokens. The latency was found to improve as the tokens were further limited to 2048 and 1024 tokens. With a 0.5% reduction in WER, around 25% reduction in GMM evaluation time was obtained.

4.4 HLS Synthesis report

Figure 4.4 shows the latency details. The minimum latency corresponds to the GMM evaluation when the number of mixtures for the particular PDF is 5 and the maximum corresponds to that of the maximum number of mixtures,i.e 18. As per the reports, the computation part takes half of the hardware cycles and the remaining time indicates data read. The actual time taken by the computation part was found to higher than that estimated from HLS reports and it is estimated using SDK which is discussed in the next chapter. Since the average number of mixtures in the model file is 6, the average GMM evaluation cycles are close to the minimum latency figures shown in the report.



Timing (ns)

Summary

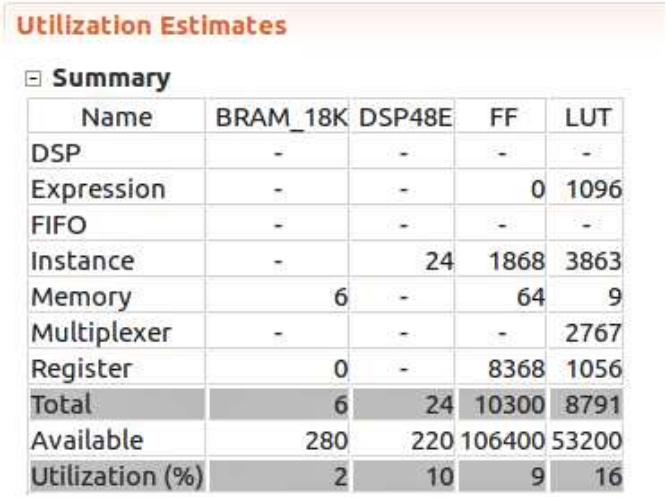
Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.434	2.70

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
593	1434	593	1434	none

Figure 4.4: Latency estimation with final optimization



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1096
FIFO	-	-	-	-
Instance	-	24	1868	3863
Memory	6	-	64	9
Multiplexer	-	-	-	2767
Register	0	-	8368	1056
Total	6	24	10300	8791
Available	280	220	106400	53200
Utilization (%)	2	10	9	16

Figure 4.5: Resource utilization with the final optimization

The resource utilization for the final implementation shown in figure 4.5 shows the actual count of BRAMs, DSPs and other resources utilized for this hardware implementation. The report shows the number of total BRAM count to be 280(18Kbits blocks) whereas ZedBoard actually has 140(36Kbits blocks). So the number of BRAMs(memory) actually used within the hardware is 3 and it is used by the local arrays used to store feature vector, inverse variance and mean inverse variance data. Apart from this, each argument to the top hardware function which is of AXI master port is allocated a BRAM(instance). So actual BRAM count for GMM function is 7. Due to unrolling techniques, the number of DSPs, LUTs, and FFs have increased but it eventually results in better latency.

4.5 Conclusion

The GMM log likelihood module is successfully implemented in hardware. The GMM function was tested in isolation from the entire ASR code and 5-sec wave was the test wave used to test and analyze the latency. The code was compiled and built in SDSoC.The primary target hardware was ZedBoard and the final implementation was done in ZCU102 at 400 MHZ also. The initial implementation in ZedBoard without any optimization technique resulted in latency of around 22 minutes. This time was finally brought down to 3.3 using different optimization methods.

CHAPTER 5

IMPLEMENTATION IN HLS-SDK

5.1 Introduction

The previous chapter describes the implementation of GMM function in hardware and the methods used to get minimum latency. The C++ code was compiled and built using SDSoC™ environment. One of the initial problems that were faced while implementing the hardware was the function overhead. This problem was overcome by calling a top hardware function once per frame from the main function and calling the actual GMM hardware function within the top function. The flexibility to create such kind of a system was provided by the nature of ASR decoding as it calls the entire decoding function once per frame. In situations where the hardware function is to be called always from the main function, the function overhead still poses the main challenge. Since SDSoC does the entire task from the high-level synthesis of the code to develop the software for calling the hardware, there are chances that this process ignores some optimizations which are achieved if the HLS synthesis and software development is separately done. This stands as a motivation to test and implement the GMM hardware function using Vivado HLS and Xilinx SDK. Vivado HLS synthesize the C++ code and create the IP for the hardware function. This IP is then integrated with the Zynq processor using Vivado block design tool and finally the software test bench which runs on the processor and calls the hardware function is created using Xilinx SDK.

The GMM function hardware implementation is first tried using AXI_LITE interfaces for the arguments. In this implementation, the hardware function is directly called from the main function and caching based on PDF id is used to match with the actual function calls. In the later part, implementation using AXI_MASTER interface is tried and the challenges faced are listed out.

5.2 Implementation using AXI_LITE interface

The first step in the hardware implementation of any function is the high-level synthesis of the C++ code which is done here by Vivado HLS. As mentioned in previous chapters, Vivado HLS first creates the RTL code corresponding to C++ code. In the RTL code, the input and output to the module must be performed through a port in the design interface and it is typically operated using a specific I/O protocol. In order to manage these interfaces, HLS describes an interface pragma which specifies how RTL ports are generated from the function definition during interface synthesis. More details about interface pragma can be obtained from reference (6). Since the GMM function puts up the need for efficient data transfer, the function arguments should be assigned with AXI ports for high speed and performance. Choosing $s\_axilite$ mode in interface pragma implements all input-output ports as an AXI4-Lite interface. After successful synthesis and C/RTL co-simulation, the IP corresponding to the hardware is created and exported in Vivado HLS. In the Export IP process, Vivado HLS produces an associated set of C driver files to implement AXI4-Lite ports. The block is then integrated with the processor IP in Vivado design tool. The automatic connection tool in Vivado adds the required AXI interconnect for the ports. Also, the Address editor window specifies the range and starting address of the address space dedicated to the hardware module. Figure 5.1 shows the IP block design created using Vivado.

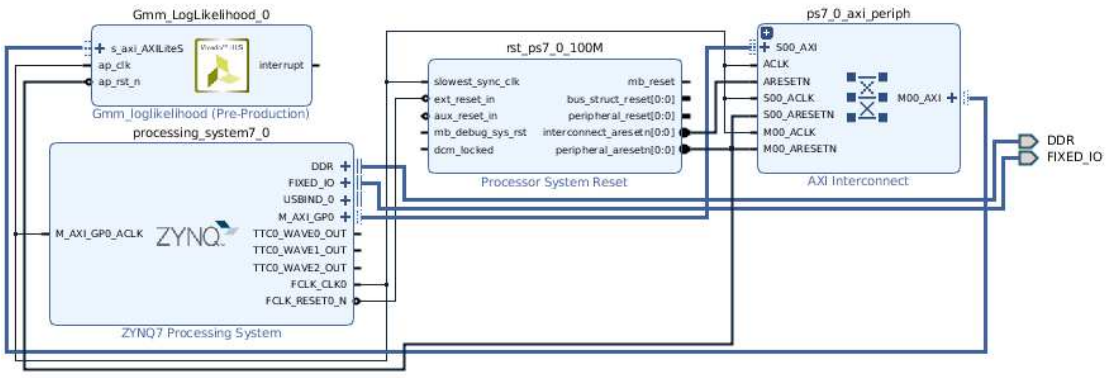


Figure 5.1: Block design using Vivado with AXI_LITE interface

Once the block design is created, the bitstream file for the hardware is generated. Finally, the hardware file is exported and launched in the Xilinx SDK for creating the software part. In this process, a Hardware Description File(HDF) of the GMM function is sent to SDK. This file contains the details of the starting address of each argument and for array arguments, all registers are in contiguous locations. In SDK, an application project is created which compiles and build the software part. Application projects can be created either in standalone or Linux mode.

In Linux mode, we can map a kernel address space to a user address space. This eliminates the overhead of copying user space information into the kernel space and vice versa. To do this, a function is created which does an $open()$ on the $/dev/mem$ file which gives access to the physical memory and then use the $mmap()$ to map a portion of that memory into user space, depending on the starting address obtained from Vivado design, for the software program to be able to access. This will allow the program to write to physical memory and control the hardware device without any overhead.

When memory mapping is done, the data to be transferred is written to these mapped registers. After writing the data, the hardware module is started by giving a start signal. Writing 0x01 to the memory mapped base address starts the module. Finally, when the execution is done, the second bit of the base address is internally set high. On checking this condition, the output data can be read back from the corresponding registers. Since the code does its own optimizations, checking for the done signal is found to give an error. To avoid this we use the volatile keyword as shown below:

```
1 #define BASE_ADDR 0x43c00000
2 volatile unsigned fptr = setup_devmem(BASE_ADDR);
```

In the code above 0x43c00000 is the base address obtained from Vivado design and setup_devmem does the memory mapping to the volatile unsigned int variable fptr. The latency estimation was done using the standard functions defined in time.h header file. In this method, the time taken for writing the data into the hardware, computation and reading the output can be separately estimated. Performance results for 5-sec wave obtained using this implementation are shown in table 5.1.

Total GMM evaluation time	45532 ms
Total Write time	42584 ms
Total computation time	2278 ms
Total read time	187 ms

Table 5.1: Latency report for Implementation using AXI4-Lite Interface

The above timing is obtained with data corresponding to 10000 active tokens used. To compare the timing with SDSoC, the same code is implemented in SDSoC. Unlike the implementations mentioned in the previous chapter, the Data zero_copy pragma is removed for a fair comparison. Since SDSoC does not allow HLS interface pragma, the default data mover allocated by SDSoC which uses AXI interface is used for the arguments. The performance result for the SDSoC implementation and its comparison with SDK implementation is shown in table 5.2.

Average hardware cycles	42005
Total GMM calls	400K
Total evaluation time	168 sec
Increase in time compared to SDK Implementation	122.5 sec

Table 5.2: Timing results in SDSoC implementation and its Comparison with SDK

From the results, it is clear that the function overhead is negligible in SDK implementation. Even though the implementation method shown in the previous chapter tackles the overhead issue for GMM function, in general, SDK implementation is observed to have much-improved latency compared to SDSoC.

5.3 Challenges in AXI Master interface

The results in the previous section motivate to test and compare the latency in SDK and SDSoC implementation for the AXI master interface. Since the memory mapping method is not explicit as in AXI-Lite interface, the implementation of the AXI master interface was tested on a sample code rather than testing on the actual GMM code. The sample code includes a hardware function which reads an array from the main function and returns a scalar output after a minor computation. AXI master interface is set for the array argument and the output argument is set to AXI-Lite using HLS pragma. To facilitate burst reading, the array is initially read into BRAM using memcpy(). The implementation process is similar to that mentioned in the previous section except in

Vivado design suite, we need to add an extra HPO slave interface at the processor IP to run the auto connection. The IP block design is shown in figure 5.2

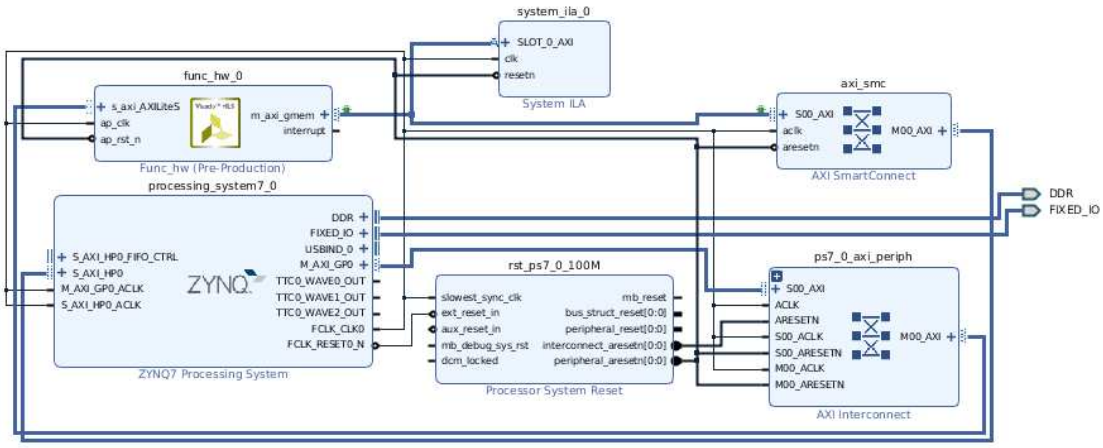


Figure 5.2: Block design using Vivado with AXI_MASTER interface

Since the code has both AXI Lite and Master interface, 2 separate interconnects are generated in the auto connection. The AXI smart-connect IP connects the AXI master port of the hardware module to the processor's HPO slave port which directs it to the external memory. An ILA(Integrated Logic Analyzer) IP is also added manually to debug the system.

The main challenge faced in this implementation is to initiate the data transfer through AXI master. When a port is defined to have the AXI master interface, its corresponding slave is the external DDR registers. Since the data transfer is supposed to happen directly from DDR to the hardware at much higher speed, the control signal issued by the processor is observed to work asynchronously with the hardware. The HDF file indicates a single register corresponding to the master port. The attempt was made to initiate the write transaction by writing the starting address location(of the data to be read into the hardware) into the register described in the HDF file. But no data beats were observed in ILA. To restrict undesirable optimizations being made by the compiler, the arguments were also declared as volatile which still failed to initiate the write transaction. With no successful attempt(both in Linux mode and standalone mode) to implement data transfer through AXI master using SDK, this stands as a potential future work as SDK implementation results with AXI-Lite interface is observed to be better than SDSoC implementation.

CHAPTER 6

CONCLUSION AND FUTURE SCOPE

6.1 Conclusion

A hardware accelerator for GMM log-likelihood function which computes the acoustic score for an ASR system is successfully implemented. The function was implemented in ZedBoard and ZCU102. The hardware function was tested with an audio wave of 5 seconds duration for different optimizing methods. The final implementation showed around 5X improvement over the initial Arm software implementation. In ZedBoard, the time taken for evaluation of 5-sec wave was brought down from 22 minutes in the initial implementation to 3.3 seconds using the final optimization. This improvement in latency is achieved by optimizing the data transfer, optimizing the computation and implementing an efficient caching system.

6.2 Future Work

The improvement in latency mentioned above is with respect to the model file used. The GMM evaluation time can also be reduced by compressing the parameters. The mixture data can be quantized efficiently so that each transfer can send more data. This compression of parameters comes with a compromise in the accuracy,i.e, increase in WER. So an efficient quantization method is to be implemented. The present implementation used floating point values. This can be changed into a fixed point with properly allocating the integer and fractional bits. Further, as the implementation in HLS and creating the software part in SDK proved to be better than using SDSoC, the possibility of data transfer through AXI master interface in SDK implementation can also be worked out.

REFERENCES

[1] Michael Price, James Glass and Anantha P. Chandrakasan, *A 6 mW, 5,000-Word Real-Time Speech Recognizer Using WFST Models*. IEEE JOURNAL OF SOLID-STATE CIRCUITS, VOL. 50, NO. 1, 2015.

[2] Michael Price, James Glass, and Anantha P. Chandrakasan, *A Low-Power Speech Recognizer and Voice Activity Detector Using Deep Neural Networks*. IEEE JOURNAL OF SOLID-STATE CIRCUITS, VOL. 53, NO. 1, 2018.

[3] Lawrence R. Rabiner, *A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition*. PROCEEDINGS OF THE IEEE, VOL. 77, NO. 2, 1989.

[4] Xilinx. *SDSoC Profiling and Optimization Methodology Guide(UG1235)*. Xilinx(v2018.3) User Guide, 2019.

[5] Xilinx. *SDSoC Programmers Guide(UG1278)*. Xilinx(v2018.2) User Guide, 2018.

[6] Xilinx. *SDx Pragma Reference Guide(UG1253)*. Xilinx(v2018.2) User Guide, 2018.

[7] Xilinx. *Vivado HLS Optimization Methodology Guide(UG1270)*. Xilinx(v2018.1) User Guide, 2018.

[8] Xilinx. *Zynq-7000 SoC Technical Reference Manual(UG585)*. Xilinx(v1.12.2) User Guide, 2018.

[9] Xilinx. *Vivado Design Suite AXI Reference Guide(UG1037)*. Xilinx(v4.0) User Guide, 2017.