



Using FPGAS to accelerate the training process of a Gaussian mixture model based spike sorting system  
by Yongming Zhu

A thesis submitted in partial fulfillment of the requirement for the degree of Master of Science in Electrical Engineering  
Montana State University  
© Copyright by Yongming Zhu (2003)

**Abstract:**

The detection and classification of the neural spike activity is an indispensable step in the analysis of extracellular signal recording. We introduce a spike sorting system based on the Gaussian Mixture Model (GMM) and show that it can be implemented in Field Programmable Gate Arrays (FPGA). The Expectation Maximization (EM) algorithm is used to estimate the parameters of the GMM. In order to handle the high dimensional inputs in our application, a log version of the EM algorithm was implemented. Since the training process of the EM algorithm is very computationally intensive and runs slowly on Personal Computers (PC) and even on parallel DSPs, we mapped the EM algorithm to a Field Programmable Gate Array (FPGA). It trained the models without a significant degradation of the model integrity when using 18 bit and 30 bit fixed point data. The FPGA implementation using a Xilinx Virtex II 3000 was 16.4 times faster than a 3.2 GHz Pentium 4. It was 42.5 times faster than a parallel floating point DSP implementation using 4 SHARC 21160 DSPs.

USING FPGAS TO ACCELERATE THE TRAINING PROCESS OF A GAUSSIAN  
MIXTURE MODEL BASED SPIKE SORTING SYSTEM

by

Yongming Zhu

A thesis submitted in partial fulfillment  
of the requirement for the degree

of

Master of Science

in

Electrical Engineering

MONTANA STATE UNIVERSITY  
Bozeman, Montana

November 2003

©COPYRIGHT  
by  
Yongming Zhu  
2003  
All Rights Reserved

N378  
Z619

APPROVAL

of a thesis submitted by

Yongming Zhu

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

Dr. Ross K. Snider

*Ross Snider*

(Signature)

*12-1-03*

Date

Approved for the Department of Electrical and Computer Engineering

Dr. James Peterson

*James N. Peterson*

(Signature)

*12-1-03*

Date

Approved for the College of Graduate Studies

Dr. Bruce McLeod

*Bruce R. McLeod*

(Signature)

*12-8-03*

Date

## STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature



Date

12/1/2003

## TABLE OF CONTENTS

1. INTRODUCTION .....	1
What is Neural Spike Sorting? .....	1
Spike Sorting System .....	3
Gaussian Mixture Model .....	6
Overview of the Thesis .....	7
2. THE GAUSSIAN MIXTURE MODEL AND THE EM ALGORITHM.....	9
Introduction .....	9
Mixture Model for Spike Sorting .....	9
The Gaussian Mixture Model (GMM) .....	9
GMMs for spike sorting .....	11
Spike sorting using the GMM .....	13
The advantage and limitation of the GMM spike sorting system .....	15
Maximum Likelihood and the Expectation Maximization Algorithm.....	16
Maximum Likelihood Estimation.....	17
The Expectation-Maximization (EM) algorithm.....	18
Expectation Maximization for the Gaussian Mixture Model .....	20
Implementation of the EM algorithm .....	23
3. IMPLEMENTATION IN MATLAB.....	25
Introduction .....	25
Neural Signal Data from a Cricket.....	26
Practical Problems Related to the EM algorithm.....	27
Model Number Selection.....	27
Initialization of the EM Algorithm.....	28
Evaluation of the Algorithm.....	30
Numeric Considerations in Matlab .....	30
The Log EM Algorithm.....	30
Using A Diagonal Covariance Matrix .....	34
Implementation Result in Matlab.....	36
Matlab Performance .....	38
4. PARALLEL DSP IMPLEMENTATION.....	40
Introduction .....	40
Hardware .....	40
SHARC ADSP-21160M DSP Processor.....	40
Bittware Hammerhead PCI board and VisualDSP++ .....	41
Analysis of the EM Algorithm.....	43
Parallel Transformation of the EM Algorithm.....	44
Performance on the Parallel DSP Platform.....	49

## TABLE OF CONTENTS - CONTINUED

5. FPGA IMPLEMENTATION .....	51
Introduction .....	51
The FPGA and Its Structure .....	52
Field Programmable Gate Arrays .....	52
Structure of Xilinx's Virtex II FPGA .....	53
AccelFPGA .....	56
The Fixed Point Representation of the EM Algorithm .....	58
Using Lookup Tables .....	59
Parallelism needed for FPGA implementation.....	65
Synthesis and Simulation .....	67
FPGA Result Evaluation .....	69
FPGA Performance .....	72
6. CONCLUSION AND FUTURE WORK .....	73
Conclusion.....	73
Future work .....	74
BIBLIOGRAPHY .....	76
APPENDICES .....	80
APPENDIX A: SIMPLYFYING THE EXPRESSION OF $Q(O, O^s)$ .....	81
APPENDIX B: FLOATING POINT MATLAB CODE.....	85
APPENDIX C: C CODE ON THE PARALLEL DSP PLATFORM .....	89
APPENDIX D: FIXED-POINT MATLAB CODE USING ACCELPGA .....	114

## LIST OF FIGURES

FIGURE	PAGE
1-1 This extracellular recording waveform shows different action potentials from an unknown number of local neurons. The data were recorded from an adult female cricket's cercal sensory system. (Data courtesy of Alex [7]).....	2
1-2 A typical neural signal recording, detecting and spike sorting system. ....	3
2-1 Non-Gaussian density estimation using a GMM .....	10
2-2 Illustration of the spike generating process.....	12
2-3 A multi-variant Gaussian model of the spike waveforms generated from a particular neuron. The solid line shows the mean vector of this model while the dashed line shows the three $\delta$ boundary according to the covariance matrix. All the waveforms in this model are shown in the background. ....	13
2-4 Block Diagram of the Bayesian's decision rule used for spike sorting system. The log probability for a spike being in a cluster is computed for each individual cluster and the highest scoring cluster is the identified neuron.....	15
2-5 Diagram of EM algorithm for GMM parameter estimation.....	23
3-1 Plot of all the 720 neural spike waveforms. ....	26
3-2 The increase in likelihood due to the increase of cluster numbers. The amount of the log likelihood increase is shown in the bar plot. The dashed line shows the threshold value. ....	28
3-3 Initialization of the EM algorithm.....	29
3-4 Diagram of the revised EM algorithm. The E-step implemented in the log domain is shown inside the dashed rectangle. ....	33
3-5 Results of three different approach. ....	35
3-6 Clustering result from Matlab. ....	37
3-7 Clustering result for the training data. Five clusters are labeled using different color.....	37

## LIST OF FIGURES - CONTINUED

FIGURE	PAGE
4-1 Block diagram of Hammerhead PCI system.....	42
4-2 The diagram shows most computational intensive parts in the EM algorithm and their execution percentage in the whole process. (a) Calculation of $p(x_i   o_l)$ , probability of each data point in each cluster based on the current parameters. (b) Update of means and covariance matrices. (c) Calculation of $p(o_l   x_i)$ and $\zeta(x_i   O)$ , probability of being in each cluster given individual input data point and likelihood of each data point in the current Gaussian Mixture Model. (d) Rest of the algorithm. ....	43
4-3 Diagram of multiple DSP implementation of EM algorithm.....	48
5-1 Virtex II architecture overview [31].....	54
5-2 Slice Structure of Virtex II FPGA [31]. ....	54
5-3 Top-down design process using AccelFPGA.....	57
5-4 Histogram of the input and output data range for the LUT table. (a) The input. (b) The output.....	62
5-5 Input and output curve of the exponential operation.....	63
5-6 Diagram of implementing the LUT in block memory .....	64
5-7 Diagram of the EM algorithm implementation in the FPGA.....	67
5-8 Floorplan of the EM algorithm implementation on a Virtex II FPGA.....	69
5-9 The mean vectors of the GMMs. (a) is the floating point version. (b) is the output of the FPGA implementation. ....	70
5-10 The clustering results of both floating point and fixed point implementations.....	71

## LIST OF TABLES

TABLE	PAGE
3-1 Performance of the spike sorting system on several PCs.....	38
4-1 List of semaphores and their functions. ....	46
4-2 Execution time for 8 iterations of EM algorithm on single and 4 DSP system.....	49
4-3 Performance of the EM algorithm on DSP system and PCs.....	49
5-1 Supported configuration of the block SelectRAM.....	55
5-2 The specifications of the Virtex II XC2V3000 FPGA.....	55
5-3 The error between the original floating point implementation and the LUT simulations. ....	63
5-4 List of directives can be used in AccelFPGA. ....	65
5-5 The toolchain used in the FPGA implementation. ....	65
5-6 Device utilization summary of the Xilinx Virtex II FPGA.....	68
5-7 The post-synthesis timing report. ....	68
5-8 The difference between the floating point output and FPGA output.....	70
5-9 Confusion matrix of fixed point spike sorting result for 720 neural spikes from 5 neurons. Overall correct rate is 97.25% comparing to the floating point result.....	71
5-10 Performance comparison between all the platforms. ....	72

## ABSTRACT

The detection and classification of the neural spike activity is an indispensable step in the analysis of extracellular signal recording. We introduce a spike sorting system based on the Gaussian Mixture Model (GMM) and show that it can be implemented in Field Programmable Gate Arrays (FPGA). The Expectation Maximization (EM) algorithm is used to estimate the parameters of the GMM. In order to handle the high dimensional inputs in our application, a log version of the EM algorithm was implemented. Since the training process of the EM algorithm is very computationally intensive and runs slowly on Personal Computers (PC) and even on parallel DSPs, we mapped the EM algorithm to a Field Programmable Gate Array (FPGA). It trained the models without a significant degradation of the model integrity when using 18 bit and 30 bit fixed point data. The FPGA implementation using a Xilinx Virtex II 3000 was 16.4 times faster than a 3.2 GHz Pentium 4. It was 42.5 times faster than a parallel floating point DSP implementation using 4 SHARC 21160 DSPs.

## CHAPTER 1

## INTRODUCTION

What is Neural Spike Sorting?

Most neurons communicate with each other by means of short local perturbations in the electrical potential across the cell membrane, called action potentials or spikes [1]. By using extracellular glass pipettes, single etched (sharp) electrodes, or multiple-site probes, scientists have been able to record the electrical activity of neurons as they transmit and process information in the nervous system. It is widely accepted that information is coded in the firing frequency or firing time of action potentials [2,3]. It is further believed that information is also coded in the joint activities of large neural ensembles [4]. Therefore, to understand how a neural system transmits and processes information, it is critical to simultaneously record from a population of neuronal activity as well as to efficiently isolate action potentials arising from individual neurons.

Single nerve cell recording is possible by using intracellular electrodes. Glass pipettes and high-impedance sharp electrodes are used to penetrate a particular nerve cell and monitor the electrical activities directly. This has been used to understand the mechanism of action potentials and many other neuronal phenomena [5,6]. However, these electrodes are not practical in multi-neuron recording. For intact free-moving animals, a small movement of the intracellular electrodes will easily damage the nerve cell tissue. Furthermore, it is very difficult to isolate a large number of neurons in a small local region. In awake animals, the isolation sometimes lasts for a very short

period of time. Fortunately, since all that we need is the timing of action potential, it is possible to use extracellular electrodes to acquire this information. With larger tips than intracellular electrodes, extracellular electrodes can simultaneously record signals of a small number (3 to 5) of neurons from a local region. Figure 1 shows the waveform comprised of action potentials recorded by one extracellular microelectrode. Each voltage spike in the waveform is the result of an action potential from one or more neurons near the electrode. The process of identifying and distinguishing these spikes that arise from different neurons is called “spike sorting”.

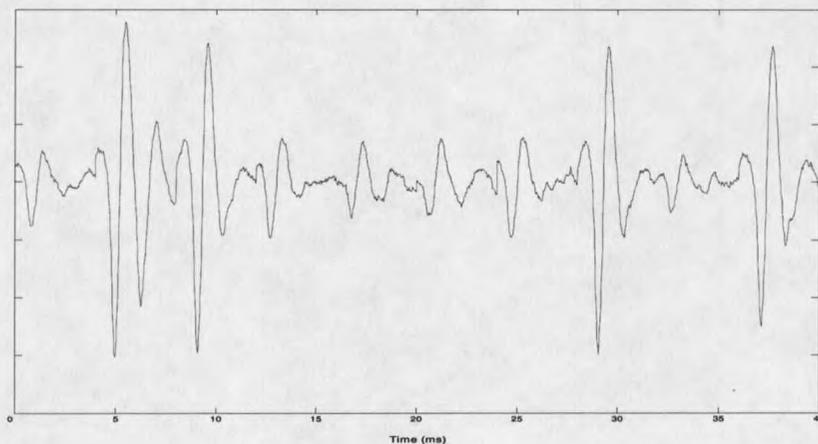


Figure 1-1 This extracellular recording waveform shows different action potentials from an unknown number of local neurons. The data were recorded from an adult female cricket's cercal sensory system. (Data courtesy of Alex [7])

Spike sorting provides an alternative to physical isolation for multi-neuron recording. In this approach, no effort is made to isolate a single cell; rather the spikes due to several cells are recorded simultaneously and sorted into groups according to their waveforms. Each group is presumed to represent a single cell since their waveforms change as a function of position relative to the electrode. The closer an

electrode is to a particular neuron, the greater the amplitude of the signal will be compared with other waveforms.

### Spike Sorting System

A typical system that measures and analyses extracellular neural signals is shown in Figure 1-2. There are four stages between the electrode and the identification of spike trains from individual neurons.

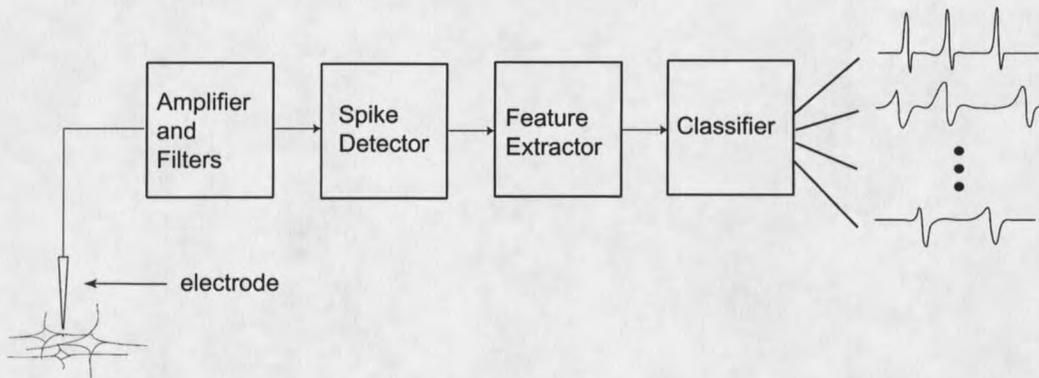


Figure 1-2 A typical neural signal recording, detecting and spike sorting system.

In the first stage neural signals are picked up by extracellular electrodes and amplified and then filtered. Low-pass filters are used to smooth the spike waveforms and provide an anti-aliasing filter before sampling. The frequency content of the waveform is typically less than 10 KHz. Other methods, such as wavelet denoising [8] can also be used to remove the recording noise. The second stage is spike detection. This is usually achieved by a simple threshold method, in which spikes are picked up when the maximum amplitude is bigger than a manually set threshold value. However, other methods including non-linear energy operators [9], wavelet-based detectors [10] and slope-shape detectors [8] have been used to improve the detection performance,

especially in the low SNR situations. A new approach, in which a noise model is first generated and then the distance from this noise model is computed to identify spikes [7] is used to obtain the spikes for this paper. However, even in this approach a threshold must be chosen.

Once the data has been collected, features must be extracted to be used in classification. The features can be simple user-defined features such as peak spike amplitude, spike width, slope of initial rise, etc. Another widely used method for feature extraction is Principle Components Analysis (PCA) [11]. PCA is used to find an ordered set of orthogonal basis vectors that can capture the directions in the data of largest variation. The first  $K$  principal components will describe the most variation of the data and can be used as features to classify the spikes. Wavelet decomposition has also been used to define spike features in a combined time-frequency domain. Under high background noise, another linear transforming method based on entropy maximization has been reported to generate good features for classification [12].

People use extracted features instead of the full sampled waveform because of the "Curse of Dimensionality", which means the computational costs and the amount of training data needed grow exponentially with the dimension of the problem. In high dimensional space, more data will be needed to estimate the model densities. Hence, with limited computational ability and training data, high dimensional problems are practically impossible to solve, especially in regards to statistical modeling. A person's inability to view high dimensional data is another reason to decrease the data dimension to two or three for manual clustering.

In this paper, we present a log version of the Expectation Maximization (EM) algorithm which solves the numerical problem arising from the high dimensional input based on the Gaussian Mixture Model. In this case, we can use the full sampled spike waveforms which preserve all the information of the neural spikes, as the classifier input.

The last stage is the clustering of spikes. In this stage, spikes with similar features are grouped into clusters. Each cluster represents the spikes that arise from a single neuron. In most laboratories, this stage is done manually with the help of visualization software. All the spikes are plotted in some feature space and the user simply draws ellipses or polygons around sets of data which assigns data to each cluster. This process is extremely time-consuming and is affected by human bias and inconsistencies.

The development of microtechnology enables multi-tip electrode recording, such as stereotrode [13] or tetrode [14] recording . By adding spatial information of action potentials, multi-electrode recording can improve the accuracy of spike sorting [15]. Obviously, the amount of data taken from multi-channel recording can be too overwhelming to deal with manually. To solve this problem, some type of automated clustering method is required. During the past three decades, various unsupervised classification methods have been applied to the spike sorting issue. The applications of general unsupervised clustering methods such as K-means, fuzzy C-means, and neural network based classification schemes have achieved some success. Methods such as Template Matching and Independent Component Analysis have also been used. A complete review can be seen in [16].

In most cases, the classification is done off-line. There are two reasons: 1. Most clustering methods need user involvement. 2. Most automated clustering methods are very computational intensive and general hardware, such as personal computers or work stations, can't meet the real-time requirement of the data streams. However, online spike sorting is needed for many applications of computational neuroscience [8]. Implementing a classification method on high-speed hardware will reduce the training time needed to develop sophisticated models, which will allow more time to be devoted to data collection. This is important in electrophysiology where limited recording times are the norm.

We implement a Gaussian Mixture Model based spike sorting system on both a DSP system and a Field Programmable Gate Array (FPGA) platform. The FPGA implementation speeds up the system dramatically which will allow an online spike sorting system to be implemented.

#### Gaussian Mixture Model

If we view the measured spike signal as the combination of the original action potentials with the addition of random background noise, we can use a statistical process to model the spike signal generating process. Clustering can then be viewed as a model of the statistical distribution of the observed data. The whole data set can then be modeled with a mixture model with each cluster being a multivariate Gaussian distribution.

Gaussian mixture models have been found very useful in many signal processing applications, such as image processing, speech signal processing and pattern recognition [17,18]. For the spike sorting problem, a number of studies, based on the

Gaussian Mixture Model, have also been tried to provide statistically plausible, complete solutions. These studies have shown that better performance can be obtained by using a GMM than other general clustering methods, such as K-means [19], fuzzy c-means [20] and neural-network based unsupervised classification schemes [21]. With some modifications, the GMM based approaches also show promise in solving the overlap and bursting problems of spike sorting [22].

The Expectation Maximization (EM) algorithm is normally used to estimate the parameters of a Gaussian Mixture Model. EM is an iterative algorithm which updates mean vectors and covariance matrices of each cluster on each stage. The algorithm is very computational intensive and runs slowly on PCs or workstations.

In this paper, we present a log version of the EM algorithm which can handle high dimensional inputs. We also implement this revised EM algorithm on three different platforms, which include the PC, Parallel DSP and FPGA platforms. By comparing the different implementations, we found that, in terms of speed, the FPGA implementation gives the best performance.

#### Overview of the Thesis

The thesis will mainly focus on the following three points: the Gaussian Mixture Model, the Expectation Maximum algorithm, and the hardware implementation of the EM algorithm. Chapter 2 introduces the Gaussian Mixture Model and how to use the EM algorithm. Chapter 3 introduces the log version of the EM algorithm to estimate the mixture parameters. In Chapter 3, a log version of the EM algorithm and its performance on a PC is presented. The details of the implementation of the EM algorithm on a parallel DSP system are described in Chapter 4. Chapter 5 describes the parallel implementation of the EM algorithm on a

Xilinx Virtex II FPGA and compares the performance of FPGA with the previous implementations. Finally, Chapter 6 summarizes the work of this thesis and suggests possible future research directions.

## CHAPTER 2

## THE GAUSSIAN MIXTURE MODEL AND THE EM ALGORITHM

Introduction

This chapter introduces the Gaussian Mixture Model (GMM) and a maximum likelihood procedure, called the Expectation Maximization (EM) algorithm, which is used to estimate the GMM parameters. The discussion in this chapter will focus on the motivation, advantage and limitation of using this statistical approach. Several considerations for implementing the EM algorithm on actual hardware will be discussed at the end of this chapter.

The chapter is organized as follows: Section 2.2 serves to describe the form of the GMM and the motivations to use the GMM in clustering the action potentials from different neurons. In section 2.2, first a description of the Gaussian Mixture Model is presented. Then several assumptions and limitations of using the GMM in spike sorting are discussed. Section 2.3 introduces the clustering of the action potentials based on Bayesian decision boundaries. Finally, the EM algorithm and derivation of the GMM parameters estimation procedure are presented.

Mixture Model for Spike SortingThe Gaussian Mixture Model (GMM)

A Gaussian mixture density is a weighted sum of a number of component densities. The Gaussian Mixture Model has the ability to form smooth approximations to arbitrarily shaped densities. Figure 2-1 shows a one-dimensional example of the

GMM modeling capabilities. An arbitrary non-Gaussian distribution (shown in solid line) is approximated by a sum of three individual Gaussian components (shown by dashed lines).

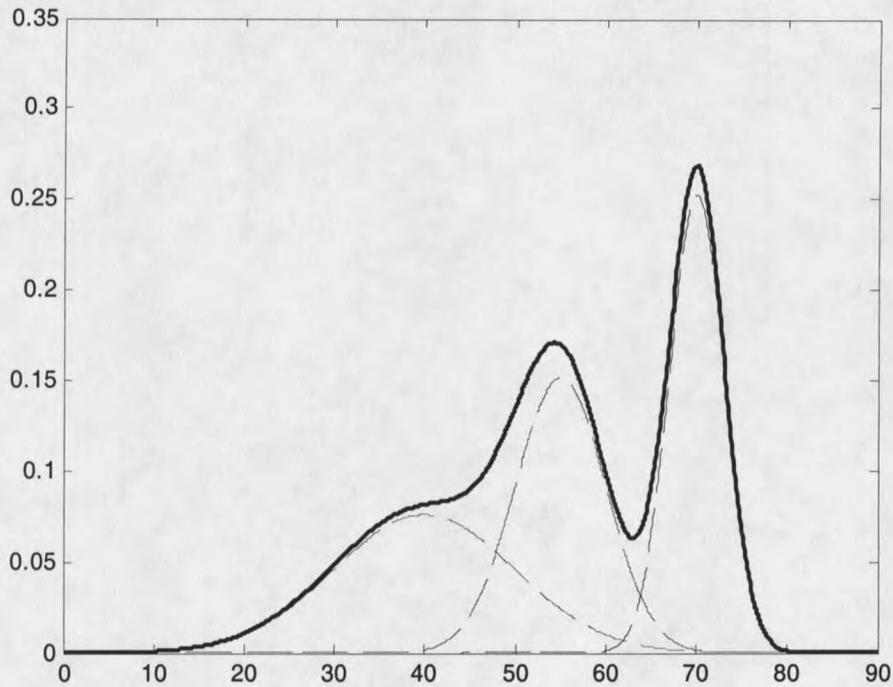


Figure 2-1 Non-Gaussian density estimation using a GMM

The complete Gaussian mixture density is parameterized by the mean vectors ( $\mu$ ), covariance matrices ( $\Sigma$ ) and weights ( $\alpha$ ) from all component densities which are represented by the notations  $O = \{\alpha, \mu, \Sigma\}$ . The model is given by Eq. 2-1,

$$p(x|O) = \sum_{l=1}^M \alpha_l p(x|o_l) \quad \text{Eq. 2-1}$$

where  $p(x|O)$  is the probability of  $x$  given model  $O$ ,  $x$  is a  $D$ -dimensional random vector,  $p(x|o_l)$  ( $l = 1, \dots, M$ ) are the component densities and  $\alpha_l$  ( $l = 1, \dots, M$ ) are the

mixture weights. Each component density is a D-variant Gaussian function given by Eq. 2-2,

$$p(x | o_l) = \frac{1}{(2\pi)^{R/2} |\Sigma_l|^{1/2}} e^{-\frac{1}{2}(x-\mu_l)^T \Sigma_l^{-1} (x-\mu_l)} \quad \text{Eq. 2-2}$$

where  $\mu_l$  is the mean vector and  $\Sigma_l$  is the covariance matrix of each individual Gaussian model  $o_l$ .

The mixture weights  $\alpha_l$  satisfy the constraint that  $\sum_{l=1}^M \alpha_l = 1$  ( $\alpha_l \geq 0$ ), which ensures the mixture is a true probability density function. The  $\alpha_l$ 's can be viewed as the probability of each component.

In spike sorting, action potentials generated by each neuron are represented by each component in a Gaussian Mixture Model. Details about using GMMs to model the spike sorting process will be described in the next section.

### GMMs for spike sorting

Since action potentials are observed with no labeling from neurons they arise from, spike generating can be considered to be a hidden process. To see how the hidden process leads itself to modeling the spike waveforms, consider the generative process in Figure 2-2.

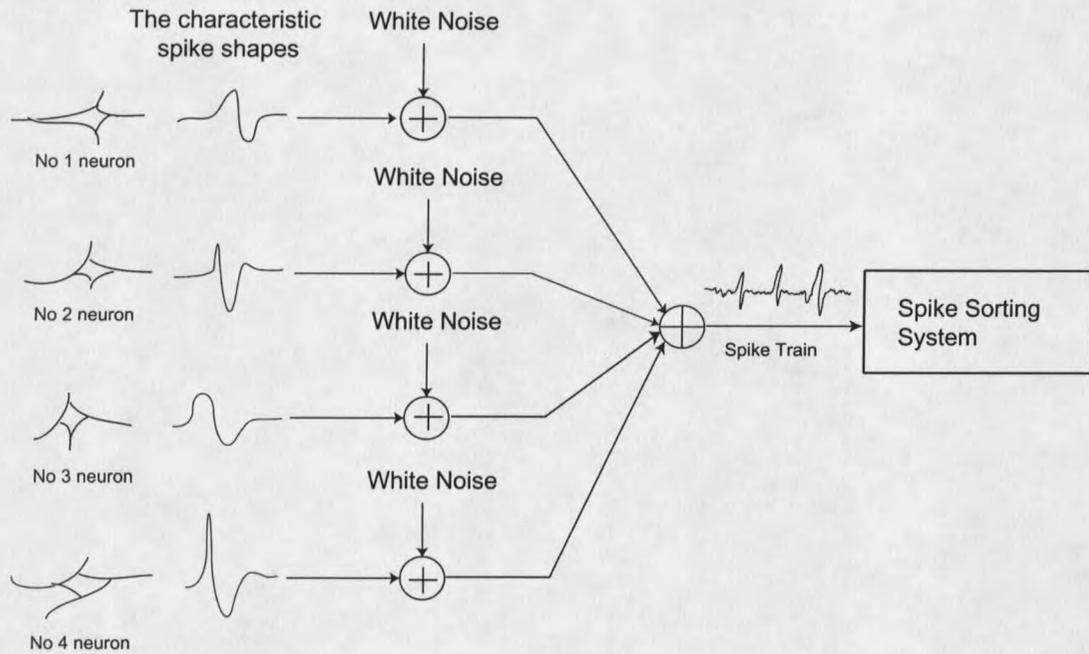


Figure 2-2 Illustration of the spike generating process.

The neural spike train is considered to be generated from a hidden stochastic process. Each spike is generated from one of the neurons according to an *a priori* discrete probability distribution  $\{\alpha_i\}$  for selecting neuron  $i$ . If we assume that the action potentials from each neuron has its own characteristic waveform shape (mean value) and addition of white noise with different variance (covariance), observations from each neuron can be modeled by a multi-variable Gaussian model as seen in Figure 2-3. The observation sequence thus consists of feature vectors drawn from different statistical populations (each neuron) in a hidden manner.

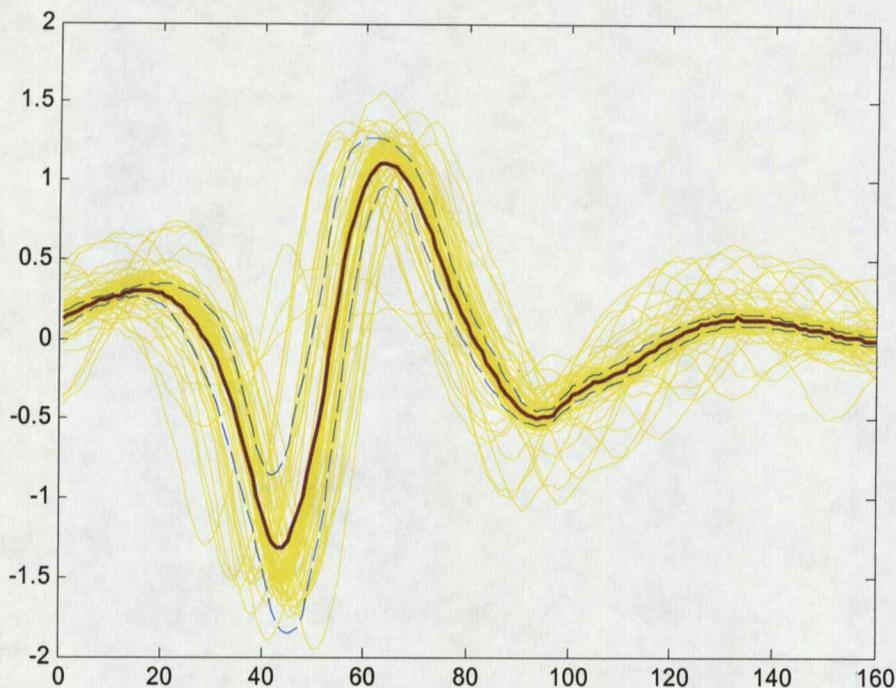


Figure 2-3 A multi-variant Gaussian model of the spike waveforms generated from a particular neuron. The solid line shows the mean vector of this model while the dashed line shows the three  $\delta$  boundary according to the covariance matrix. All the waveforms in this model are shown in the background.

Furthermore, if we assume that action potentials from different neurons are independent and identically distributed [16], the observation vector distribution can be given by a Gaussian Mixture Model (Eq. 2-1). Thus, the GMM results directly from a probabilistic modeling of the underlying spike generating process.

#### Spike sorting using the GMM

In the last section, we describe how to use the GMM to model the spike generating process. From the perspective of spike sorting, the goal is to estimate the parameters of the GMM and classify each action potential with respect to each neuron

class in which the action potential is generated from. To do this, the first step is to estimate all the parameters, including all the priors' probabilities  $\{\alpha_i\}$  together with the means and covariance of each Gaussian component. Maximum likelihood (ML) parameter estimation is the method we will use to find these parameters with a set of training data. It is based on finding the model parameters that are most likely to produce the set of observed samples. The maximum likelihood GMM parameter estimates can be found via an iterative parameter estimation procedure, namely the Expectation Maximization (EM) algorithm. EM algorithms have been widely used in estimating the parameters of a stochastic process from a set of observed samples. Details about ML and EM algorithm will be discussed in the next section.

After estimation of the model parameters, classification is performed by the Bayesian decision rule. The probability that each observation ( $x$ ) belongs to a particular classe ( $o_i$ ) is calculated by using Bayes' rule (Eq. 2-3),

$$p(o_i | x) = \frac{p(x | o_i) p(o_i)}{\sum_{k=1}^M p(x | o_k) p(o_k)} \quad \text{Eq. 2-3}$$

Each individual density  $p(o_i | x)$  is determined by the means and covariance of each component density using Eq. 2-2. The prior probability  $p(o_i)$  is identical to the weights  $\alpha_i$  in Eq. 2-1.

Since the cluster membership is probabilistic, the cluster boundaries can be computed as a function of a confidence level. The spikes that don't fall into any clusters will be considered as noise, or overlapping spikes that can be dealt with later.

Mathematically, it can be shown that if the model is accurate, the Bayesian decision boundaries will be optimal, resulting in the fewest number of misclassifications [23].

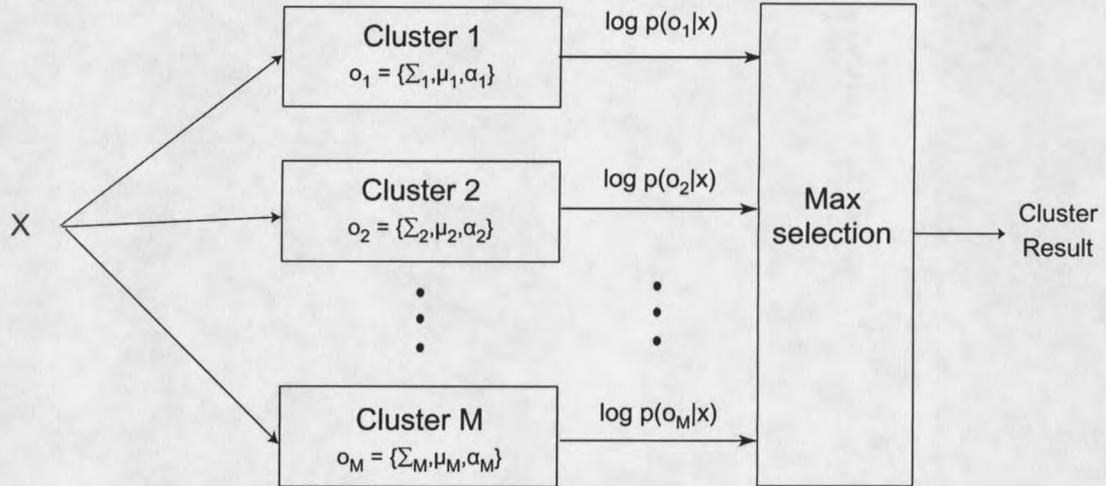


Figure 2-4 Block Diagram of the Bayesian's decision rule used for spike sorting system. The log probability for a spike being in a cluster is computed for each individual cluster and the highest scoring cluster is the identified neuron.

Figure 2-4 shows a block diagram of the Bayesian's decision rule applied to spike sorting using GMM. The input to the system is a vector representing an action potential. The probability that an action potential belongs to a class is calculated using Eq. 2-3 over all the classes. The vector is classified into a certain class whenever the vector falls into the class with a high confidence level. If the waveform doesn't belong to any cluster with high probability, the vector will be considered as noise and will be thrown away. It is also possible that the waveform results from overlapping spikes and could be stored for further investigation.

#### The advantage and limitation of the GMM spike sorting system

GMM is a statistical approach for the spike sorting problem. It defines a probabilistic model of the spike waveforms. A big advantage of this framework is that

it is possible to quantify the certainty of the classification. By using Bayesian probability theory, the probability of both the spike form and number of spike shapes can be quantified. This is used in spike sorting because it allows experimenters to make decisions about the isolation of spikes. Also by monitoring the classification probabilities, experimenters will be able to tell possible changes in experiment conditions. For example, a drop in probability may indicate electrode drift or a rise in noise levels.

In spite of the advantages of the GMM approach, there are several inherent limitations of this statistical model for spike sorting. The Gaussian mixture model makes the assumption that the noise process is Gaussian and invariant. These two cases are not necessarily true in spike sorting. For example, with the presence of burst-firing and overlap spikes, the noise may show larger variation at the tails. Another assumption made is that all neurons fire independently and all the noise is uncorrelated. It is likely that in a local area, neurons have some correlation to each other. We could use a more complete statistic approach which includes the correlation between neurons to model the spike generating process [24]. However, this is outside of the scope of this thesis.

#### Maximum likelihood and the Expectation Maximization Algorithm

This section describes an unsupervised maximum likelihood procedure for estimating the parameters of a Gaussian mixture density from a set of unlabeled observations. A maximum likelihood parameter estimation procedure is developed for the GMM. First, a brief discussion of maximum likelihood estimation and the use of the EM algorithm for GMM parameter estimation is given. This is followed by the

derivation of the mixture weight, mean, and variance estimation equations. Finally, a brief discussion of the complete iterative estimation procedure is given.

### Maximum Likelihood Estimation

Maximum likelihood parameter estimation is a general and powerful method for estimating the parameters of a stochastic process from a set of observed samples. We have a density function  $p(x|O)$  that is governed by the set of parameters  $O = \{\alpha, \mu, \Sigma\}$ . We also have a data set of size  $N$ , supposedly drawn from this distribution, i.e.  $X = \{x_1, x_2, x_3, \dots, x_N\}$ . We assume that these data vectors are independent and identically distributed (i.i.d.) with distribution  $p(x|O)$ . Therefore, the resulting probability for all the samples is

$$p(X|O) = \prod_{i=1}^N p(x_i|O) = \zeta(O|X) \quad \text{Eq. 2-4}$$

This function  $\zeta(O|X)$  is called the likelihood of the parameters given the data, or the likelihood function. The likelihood is treated as a function of the parameters  $O$  where the data  $X$  is fixed. For maximum likelihood parameters estimation, our goal is to find the  $O$  that maximizes  $\zeta(O|X)$ . That is, we wish to find  $O^*$  where

$$O^* = \arg \max_O \zeta(O|X) \quad \text{Eq. 2-5}$$

If  $p(x|O)$  is simply a single Gaussian distribution and  $O = (\mu, \sigma^2)$ . Then we can set the derivative of the  $\zeta$  function to zero and solve the likelihood Eq. 2-6 directly for  $\mu$  and  $\sigma^2$ .

$$\frac{\partial p(X|O)}{\partial O} = 0 \quad \text{Eq. 2-6}$$

However, in our case using the GMM  $p(x|O)$  is a weighted sum of several single Gaussian distributions. Attempting to solve Eq. 2-6 directly for the GMM parameters,  $O = \{\alpha, \mu, \Sigma\}$ ,  $i = 1, \dots, M$ , does not yield a closed form solution [25]. Thus we must resort to more elaborate techniques.

### The Expectation-Maximization (EM) algorithm

The maximum likelihood GMM parameters estimates can be found via an iterative parameter estimation procedure, which is called the Expectation-Maximization (EM) algorithm [25]. The EM algorithm is a general method of finding the maximum-likelihood estimate of the parameters of an underlying distribution from a given data set when the data is incomplete or has missing values. There are two main applications of the EM algorithm. The first use of the EM algorithm occurs when the data indeed has missing values. The second application occurs when optimizing the likelihood function becomes analytically intractable. This latter application is used for the GMM case where the analytical solution for the likelihood equation can not be found.

The idea behind the second application of the EM algorithm is to simplify the likelihood function  $\zeta$  by assuming the existence of additional but hidden parameters. We assume that data  $X$  is observed and is generated by some distribution. We also assume that there is another hidden data set  $Y$ , which will be defined in section 2.3.3. We call  $Z = \{X, Y\}$  the complete data set. The density function for the complete data set is:

$$p(Z|O) = p(X, Y|O) = p(Y|X, O)p(X|O) \quad \text{Eq. 2-7}$$

With this new density function, we can define a new likelihood function,

$$\zeta(O|Z) = \zeta(O|X, Y) = p(X, Y|O) \quad \text{Eq. 2-8}$$

which is called the complete-data likelihood. Note that since  $Y$  is a hidden variable that is unknown and presumably governed by an underlying distribution, the likelihood function is also a random variable. The EM algorithm first finds the expected value of the complete-data log-likelihood with respect to the unknown data  $Y$  given the observed data  $X$  and the current parameter estimates. That is, we define:

$$Q(O|O^s) = E[\log p(X, Y|O) | X, O^s] \quad \text{Eq. 2-9}$$

where  $Q(O|O^s)$  is the expected value of the log-likelihood with respect to the unknown data  $Y$ ,  $O^s$  are the current parameters estimates and  $O$  are the new parameters that we optimize to increase  $Q$ . Note that in this equation,  $X$  and  $O^s$  are constants, and  $O$  is a normal variable that we wish to adjust. The evaluation of this expectation is called the E-step of the EM algorithm. The second step is the M step. This step is to maximize the expectation we computed in the first step. That is we find:

$$\hat{O}^s = \arg \max_O \zeta(O, O^s) \quad \text{Eq. 2-10}$$

These two steps are repeated as often as necessary. The maximum likelihood parameters estimates have some desirable properties such as asymptotic consistency and efficiency [26]. This means that, given a large enough sample of training data, the model estimates will converge to the true model parameters with probability one. So each step of iteration in EM algorithm is guaranteed to increase the log-likelihood and the algorithm is guaranteed to converge to a local maximum of the likelihood function [26].

### Expectation Maximization for the Gaussian Mixture Model

In the spike sorting case, the GMM model consists of  $M$  classes, each class represents a neuron that has been recorded. Let  $X = \{x_1, x_2, x_3, \dots, x_N\}$  be the sequence of observation vectors, i.e. the measured waveforms. In this case, we assume the following probabilistic model:

$$p(x | O) = \sum_{l=1}^M \alpha_l p(x | o_l) \quad \text{Eq. 2-11}$$

where the parameters are  $O = \{\alpha_1, \alpha_2, \dots, \alpha_M, o_1, o_2, \dots, o_M\}$  such that  $\sum_{l=1}^M \alpha_l = 1$  and each  $o_l$  is a single Gaussian density function. Then the log-likelihood expression for this density from the data set  $X = \{x_1, x_2, x_3, \dots, x_i\}$  is given by:

$$\log(\zeta(O | X)) = \log \prod_{i=1}^N p(x_i | O) = \sum_{i=1}^N \log \left( \sum_{l=1}^M \alpha_l p(x_i | o_l) \right) \quad \text{Eq. 2-12}$$

which is difficult to optimize because it contains the log summation. Now, we introduce a new set of data  $Y$ .  $Y$  is a sequence of data  $Y = \{y_1, y_2, y_3, \dots, y_N\}$  whose values inform us which neuron generated the observation vector  $x_i$ . This variable  $Y$  is considered to be a hidden variable. If we assume that we know the values of  $Y$ , the likelihood becomes

$$\log(\zeta(O | X, Y)) = \sum_{i=1}^N \log(P(x_i | o_{y_i})P(y_i)) = \sum_{i=1}^N \log(\alpha_{y_i} p_{y_i}(x_i | o_{y_i})) \quad \text{Eq. 2-13}$$

Since the values of  $Y$  are unknown to us, we don't actually know which neuron generated each spike. However, if we assume  $Y$  is a random vector which follows a known distribution, we can still proceed.

The problem now is to find the distribution of the hidden variable  $Y$ . First we guess that parameters  $O^s$  are appropriate for the likelihood  $\zeta(O^s)$ . Given  $O^s$ , we can easily

compute  $p_{y_i}(x_i | O^g)$  for each data vector  $x_i$  for each component density  $y_i$ . In addition, the mixing parameters,  $\alpha_j$  can be thought of as prior probabilities of each mixture component, that is  $\alpha_j = p(o_j)$ . Therefore, using Bayes's rule, we can compute:

$$p(y_i | x_i, O^g) = \frac{\alpha_{y_i} p_{y_i}(x_i | o_{y_i}^g)}{\sum_{y_i=1}^M \alpha_{y_i} p_{y_i}(x_i | o_{y_i}^g)} \quad \text{Eq. 2-14}$$

$$p(Y | X, O^g) = \prod_{i=1}^N p(y_i | x_i, O^g) \quad \text{Eq. 2-15}$$

Here we assume that each neuron fires independently from each other. Now we have obtained the desired marginal density by assuming the existence of the hidden variables and making a guess at the initial parameters of their distribution. We can now start the E step in the EM algorithm and using this marginal density function to calculate the expected value of the log-likelihood of the whole data. The expected value is found to be (The derivation can found in Appendix A):

$$Q(O, O^g) = \sum_{l=1}^M \sum_{i=1}^N \log(\alpha_l) p(l | x_i, O^g) + \sum_{l=1}^M \sum_{i=1}^N \log(p_l(x_i | o_l)) p(l | x_i, O^g) \quad \text{Eq. 2-16}$$

With the expression for the expected value of the likelihood and we can go to the next M step to maximize this expected likelihood with respect to the parameters  $O = \{\alpha_l, \mu_l, \Sigma_l\}$ . We can maximize the term containing  $\alpha_l$  and the term containing  $o_l$  independently since they are not related.

To find the expression for  $\alpha_l$ , we have to solve Eq. 2-17 with the constraint of Eq 2-18:

$$\frac{\partial(\sum_{l=1}^M \sum_{i=1}^N \log(\alpha_l) p(l | x_i, \mathbf{O}^g))}{\partial(\alpha_l)} = 0 \quad \text{Eq. 2-17}$$

$$\sum_{l=1}^M \alpha_l = 1 \quad \text{Eq. 2-18}$$

We introduce the Lagrange multiplier  $\lambda$ , and solve the following equation:

$$\frac{\partial(\sum_{l=1}^M \sum_{i=1}^N \log(\alpha_l) p(l | x_i, \mathbf{O}^g) + \lambda(\sum_{l=1}^M \alpha_l - 1))}{\partial(\alpha_l)} = 0 \quad \text{Eq. 2-19}$$

and we get the expression

$$\alpha_l = \frac{1}{N} \sum_{i=1}^N p(l | x_i, \mathbf{O}^g) \quad \text{Eq. 2-20}$$

To get the expression of  $\mu_l$  and  $\Sigma_l$ , we have to go through a similar procedure, but the computation is much more complicated. To calculate these two parameters for the Gaussian distribution function, we'll need to take derivatives of a function of matrices. The detailed procedure can be found in [25], and here we just give the results.

$$\mu_l = \frac{\sum_{i=1}^N x_i p(l | x_i, \mathbf{O}^g)}{\sum_{i=1}^N p(l | x_i, \mathbf{O}^g)} \quad \text{Eq. 2-21}$$

$$\Sigma_l = \frac{\sum_{i=1}^N p(l | x_i, \mathbf{O}^g) (x_i - \mu_l^{new})(x_i - \mu_l^{new})^T}{\sum_{i=1}^N p(l | x_i, \mathbf{O}^g)} \quad \text{Eq. 2-22}$$

Collectively, Eq. (2-20), (2-21) and (2-22) form the basis of the EM algorithm for iteratively estimating the parameters of a GMM. The detailed steps to implement these equations will be discussed in the next section.

### Implementation of the EM algorithm

As shown in Figure 2-5, the procedure to implement the iterative EM algorithm consists of the following steps:

- 1) Initialization: Initialize the model parameters  $O(0)$
- 2) E-step: Calculate the expected likelihood of the given data based on the current parameters  $O(i)$ . Calculate the increase of likelihood. If the difference is below a certain threshold, stop the iterative process.
- 3) M-step: Estimate new model parameters  $O(i+1)$  using current model parameters  $O(i)$  via estimation equations. Replace current model parameters with new model parameters, which are the new parameters for the next E-step.

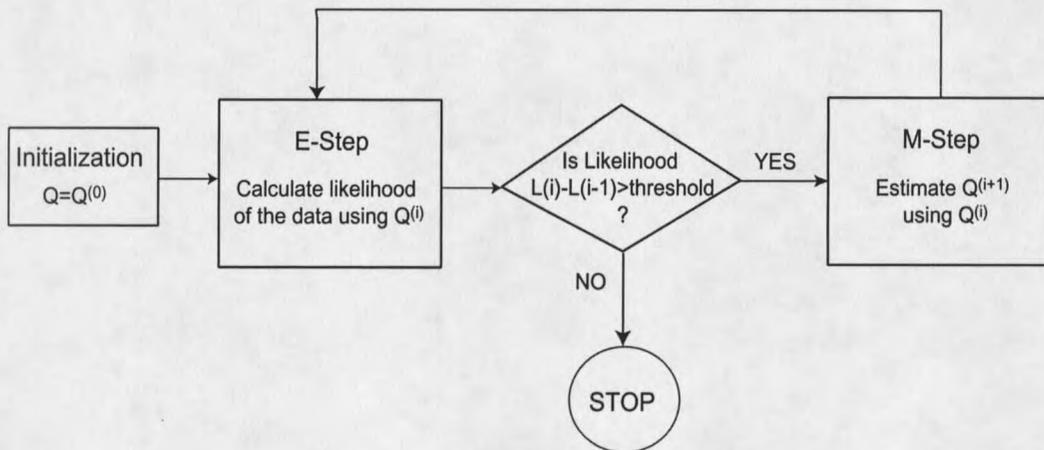


Figure 2-5 Diagram of EM algorithm for GMM parameter estimation.

Since the new model parameters obtained from the estimation equations guarantee a monotonically increasing likelihood function, the algorithm is guaranteed to converge to a stationary point of the likelihood function. It has been argued that the convergence of the EM algorithm to the optimal parameters of the mixture is

slow if the component distributions overlap considerably [25]. But if the clusters are reasonably well separated, which is typically the case in spike sorting, the convergence of the likelihood of the mixture model is rapid and the mixture density approximates the true density.

## CHAPTER 3

## IMPLEMENTATION IN MATLAB

Introduction

We have shown how to use the EM algorithm to estimate the parameters of Gaussian Mixture Model. In this chapter, we will find that a number of practical issues have to be examined closely before we can achieve robust parameter estimates. Moreover, since we will use the full-sampled waveforms as input, the high dimensional training data will cause numeric problems such as underflow due to limitations in precision. We will present a revised version of EM algorithm to solve these problems which we implement in software and hardware.

The chapter is organized as follows: First, the training data we used in this paper will be introduced. The practical problems related to the EM and applications on GMM spike sorting system are discussed in the next section. These problems include initialization of the GMM model, cluster number selection and algorithm evaluation. In the next section, the numeric underflow problem caused by high dimensional input data will be addressed. We compare three different approaches to handle this problem and present a log version of the EM algorithm with diagonal covariance matrices. This new EM algorithm gave the best result among the three methods. At the end of this chapter, we will show the resulting model trained by this log EM algorithm and its performance on a PC.

### Neural Signal Data from a Cricket

The signals we use for this thesis were recorded from the cricket cercal sensory system [27]. Data was recorded using an NPI SEC-05L amplifier and sampled at 50 KHz rate with a digital acquisition system running on a Windows 2000 platform. The stimuli for these experiments were produced by a specially-designed device which generated laminar air currents across the specimen's bodies. Details about the experiments can be found in [27]. After raw data was collected, neural spikes were detected by first identifying the segments of having no spikes. A noise model was generated by assuming that the noise was Gaussian. Then, the spikes were detected if the distance between the recorded data and the noise model was bigger than a manually set threshold [7]. After the threshold was set, we got rid of any waveforms that were obviously not action potentials and left 720 spikes for training of the EM algorithm. The 720 spikes are show in Figure 3-1. Each spike lasts for 3.2ms and has 160 sampling points. This set of 720 160-dimensional vectors was used in the EM algorithm.

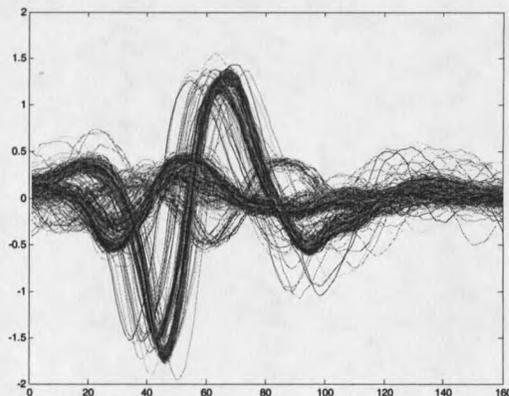


Figure 3-1 Plot of all the 720 neural spike waveforms.

## Practical Problems Related to the EM algorithm

### Model Number Selection

The number of Gaussians for the mixture model must be selected before using the EM method. However, in spike sorting, we do not have *a priori* information of how many neurons are producing action potentials. So we have to estimate the model number using the whole data set.

Based on the probabilistic foundation of GMM, we can view the choice of the optimal cluster number as a problem of fitting the data by the best model. This problem can be solved by applying the maximum likelihood estimate. However, the likelihood is a monotonically increasing function of the number of parameters. In other words, the likelihood will increase as the model number becomes larger. A number of methods have been proposed to determine the model number using maximum likelihood criteria by adding some penalty functions which can compensate for the monotonically increasing of the log-likelihood function of the model number [24]. However, it is hard to define the suitable penalty term and not very efficient to implement.

To determine the model number in this paper, we use the method presented in [12]. We calculate the likelihood of the data set from clusters ranging in size from 1 to 7. As shown in Figure 3-3, the initial increment of likelihood is large but then becomes small. This is because as the number of cluster approaches the true number of the model, the increment of likelihood becomes small. So we picked the point when the increase ( $\delta$ ) in likelihood fell below a certain threshold for 2 continuous points. Eq. 3-1 shows the actual calculation. The threshold  $\delta$  was set to 4.

$$\log(\zeta(t+2)) - \log(\zeta(t+1)) < \delta \quad \text{and} \quad \log(\zeta(t+1)) - \log(\zeta(t)) < \delta \quad \text{Eq. 3-1}$$

From Figure 3-2, we can see that 5 models is a reasonable choice for the model number. Also, keep in mind that you can always merge or split clusters using more sophisticated methods [12,28].

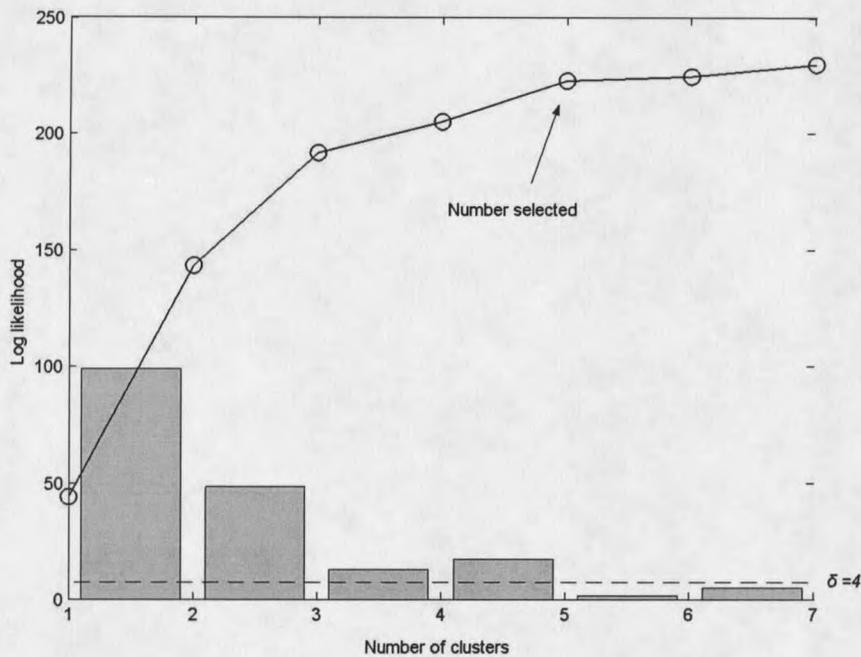


Figure 3-2 The increase in likelihood due to the increase of cluster numbers. The amount of the log likelihood increase is shown in the bar plot. The dashed line shows the threshold value.

### Initialization of the EM Algorithm

The selection of initial model parameters will affect the final results. This is because the likelihood surface associated with GMM tends to exhibit multiple local maxima. Thus, different initialization points will lead to different local maximum results. A generally used method is to apply the K-means clustering algorithm first and then use its output as the initial value of the mixture model's parameters [16].

However, the K-means method itself is also sensitive to initialization and does not guarantee to get the right initialization for the mixture model. Furthermore, K-means algorithm is computationally expensive to implement.

Another more straight forward method is to randomly select initialization from the data set. It has been shown that this method gives the same performance as the K-means method [24]. In this paper, we initialize the means to randomly chosen data points and pick a single covariance matrix for all the clusters. We set values of the initialized covariance matrix to be reasonably large so that each cluster can see all the data at the beginning of the training. We randomly picked 100 starting points from the data set and chose the result that provided the largest likelihood. Figure 3-3 shows the means for the 5 initialization clusters. The covariance matrix  $\{ \Sigma_i \}$  and prior probabilities  $\{ \alpha_i \}$  of all the clusters are set to be equal. All the EM training processes in this paper start from this same initialization so that all the results are comparable.

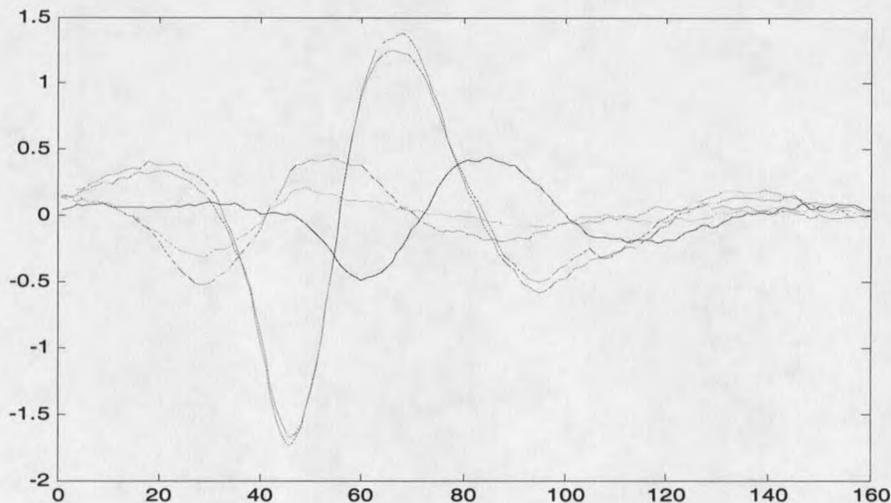


Figure 3-3 Initialization of the EM algorithm

### Evaluation of the Algorithm

One hard problem of spike sorting is that it is hard to evaluate the clustering results. Signals from different cells may be very similar and difficult to distinguish even for trained neurophysiologists. It is nearly impossible to create reliable expert-based validation data sets. Hence, simulated neural data are often used for evaluating the spike sorting algorithm. Recently, simultaneously intro- and extra-cellular recording technology has provided the possibility to evaluate the algorithm on real data.

In this paper, we focus on how to implement the GMM based spike sorting algorithm on hardware. Since evaluations for both simulated and real data show that the GMM approach gives very good results [16,29], we just concentrated on the software and hardware implementations. When targeting FPGAs, the use of fixed point data types is important because of the increase in speed that can be attained. Since using fixed point data will lose a certain amount of precision as compared to floating point data, we evaluated the FPGA implementation with respect to the floating point implementations.

### Numeric Considerations in Matlab

#### The Log EM Algorithm

To reserve all the original information in the spike waveforms, we used the full-sampled spikes as the clustering input. The implementation of the training process of EM algorithm should be able to handle the 160 dimensional inputs. However, if we look at the EM algorithm closely, we will find many places where a number of small numbers (between 0 and 1) are multiplied together. This easily leads to a numerical

zero in Matlab, which is defined to be 2.2251e-308 or less for double precision data. The calculation of the likelihood of the whole data set and the determination of covariance matrix are two places where a numeric zero problem occurs most often. To compensate for this, we have to transform these operations into the log domain so that the multiplications turn into additions that prevent the underflow problem. Here we present a log version of the EM algorithm which solves this problem.

Given the current model, we calculate the probability of each data point in each cluster. The original equations are listed as follows:

$$p(x_i | o_l) = \frac{1}{(2\pi)^{M/2} |\Sigma_l|^{1/2}} e^{-\frac{1}{2}(x_i - \mu_l)^T \Sigma_l^{-1} (x_i - \mu_l)} \quad \text{Eq. 3-2}$$

$$p(x_i | O) = \sum_{l=1}^M \alpha_l p(x_i | o_l) \quad \text{Eq. 3-3}$$

$$\zeta(O | X) = \prod_{i=1}^N p(x_i | O) \quad \text{Eq. 3-4}$$

Notice that in Eq. 3-2 and Eq. 3-4, there are product operations that will cause underflow. By changing them into the log domain, the equations are:

$$\ln(p(x_i | o_l)) = -\frac{1}{2}(x_i - \mu_l)^T \Sigma_l^{-1} (x_i - \mu_l) - \frac{M}{2} \ln(2\pi) - \frac{1}{2} \ln(|\Sigma_l|) \quad \text{Eq. 3-5}$$

$$\ln(p(x_i | O)) = \ln\left(\sum_{l=1}^M \alpha_l e^{\ln(p(x_i | o_l))}\right) \quad \text{Eq. 3-6}$$

$$\ln(\zeta(O | X)) = \sum_{i=1}^N \ln(p(x_i | O)) \quad \text{Eq. 3-7}$$

After changing into the log domain, the products in Eq. 3-2 and Eq. 3-4 are turned into additions and subtractions. Exponential operations are also eliminated

except in Eq. 3-6. Since all the  $e^{\ln(p(x_i|o_l))}$  terms are very small, this can still cause underflow problems ( $\ln(0)$ ). To handle this, we need to add another normalization process to calculate Eq. 3-5. We do a normalization across all the clusters by subtracting all the  $\ln(p(x_i | o_l))$  terms by the maximum value across the clusters. We then store the  $\max(\ln(p(x_i | o_l)))$  for future use. After subtraction, the exponential value  $e^{\ln(p(x_i|o_l))}$  will be guaranteed between  $[0, 1]$ . We then performed the multiplications and additions. The log version of the middle results can be obtained by applying  $\ln$  to it. The last step is to add the  $\max(\ln(p(x_i | o_l)))$  back to the middle result. This way, without changing the final result, we go around the numerical zero problems and get the same results.

$$\ln(p(x_i | O)) = \ln\left(\sum_{l=1}^M \alpha_l e^{\ln(p(x_i|o_l)) - \beta}\right) + \beta \quad \text{Eq. 3-8}$$

$$\beta = \max_{o_l} p(x_i | o_l) \quad \text{Eq. 3-9}$$

After getting the likelihood of the whole data and before updating the parameters, the probability of being in one cluster given input  $x_i$  has to be calculated. This probability can be calculated by Baye's rule in Eq. 3-10.

$$p(o_l | x_i) = \frac{p(x_i | o_l) p(o_l)}{\sum_{k=1}^M \alpha_k p(x_i | o_k)} = \frac{p(x_i | o_l) p(o_l)}{p(x_i | O)} \quad \text{Eq. 3-10}$$

However, this equation can also exhibit underflow problems if all the  $p(x | o_k)$ 's are very small, which will occur if a spike doesn't belong to any clusters. So we still need to convert Eq. 3-10 into the log domain as follows:

$$\ln p(o_i | x_i) = \ln p(x_i | o_i) + \ln p(o_i) - \ln p(x_i | O) \quad \text{Eq. 3-11}$$

Since we have already completed  $\ln p(x_i | O)$  from Eq. 3-8, it is fairly easy to get the result of Eq. 3-11. We can use exponentiation to get  $p(o_i | x_i)$  back into the normal domain and begin the parameter updates. All the parameter updates are done in the normal domain since no underflow problems will occur in this process. The new parameters will be used to calculate the likelihood for the whole data set again in the log domain to determine whether the iterative process should stop. The whole process can be seen in the Figure 3-4.

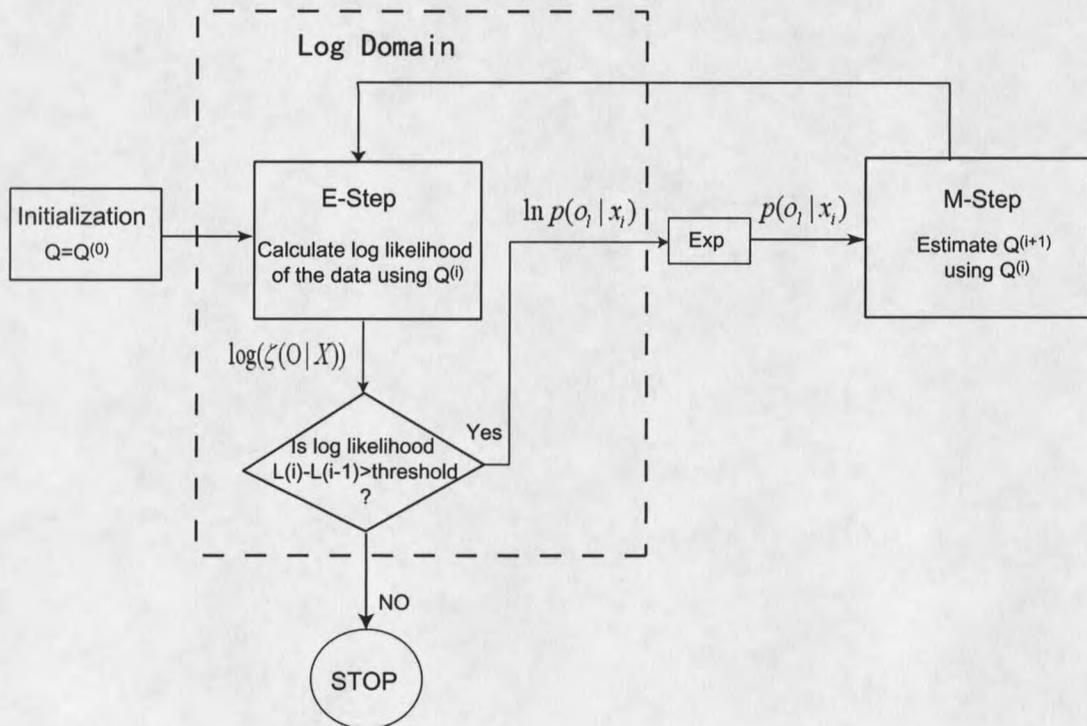


Figure 3-4 Diagram of the revised EM algorithm. The E-step implemented in the log domain is shown inside the dashed rectangle.

### Using A Diagonal Covariance Matrix

While computing Eq. 3-2, we need the determinant of the covariance matrices  $\{\Sigma_i^{-1}\}$ . However, it is difficult to get the determinant of the full covariance matrix since the data dimension is large. All the values of covariance matrices are approximately  $10^{-2}$ , so during the training process the matrices will easily become singular because of limited precision.

We tried three different approaches to solve this problem. The first method was to recondition the covariance matrices whenever a singular matrix appears. Every time when there was a numeric zero in the covariance matrix, we added a very small number ( $10^{-4}$ ) to prevent it from being zero. This way, the algorithm could go on to the next step until it converged. In the second approach, we reduced the data dimension by using Principal Component Analysis. As discussed in Chapter 1, PCA can find the directions that represent the most variation of the original data. We used the first 10 components that could represent 98% variation of the data set. By reducing the dimension to 10, there was much less chance that full covariance matrix could become singular. The last method was to use a diagonal matrix instead of the full covariance matrix. Using a diagonal matrix can enabled log calculations of the determination and prevented numerical zeros. By using the diagonal covariance matrix, we made the assumption that there was no correlation between each dimension in the 160 dimensional space. Making this assumption can cause a loss of information in the original data set. However the first two methods also lost information by recondition and reducing the vector dimensions. We tested all these methods and the best method was the diagonal matrix as shown in the next section.

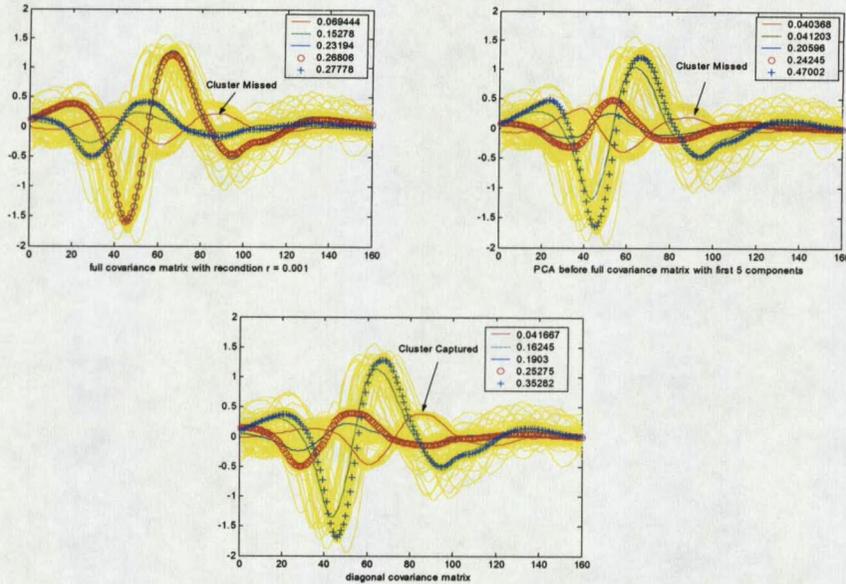


Figure 3-5 Results of three different approach. The diagonal covariance approach was the only method that captured the small cluster.

As we mentioned in the previous section, it was hard to validate the cluster result since we didn't actually know the typical spike shapes that came from individual neurons. However, by looking at the mean vectors of each cluster, we had a good idea about the ability of each approach to capture the clusters among the data. Figure 3-5 shows all the cluster means of the three methods. All the spikes are in yellow and the mean vector of each cluster is shown in a different color. While all of the three approaches caught the main clusters, the diagonal matrix approach did the best in catching the smallest cluster (red line). The other two methods failed to catch this smallest cluster. Notice that in the recondtion approach, the prior probabilities for the small clusters tend to be big. This is because adding a number to the covariance prevents the cluster to be small. Also notice that the biggest cluster in the

reconditioned and PCA method is different from the diagonal covariance matrix method. This is due to the inability of these two methods to distinguish small variance in the low-amplitude waveforms. They actually include more waveforms in this cluster where some spike shapes are noticeably different. Using the diagonal covariance matrix approach, the small variance was captured and a small cluster was generated to represent this specific spike shape. For our data set, the diagonal covariance approach lost the least amount of information in the original spike waveforms. We chose this approach for all hardware implementations.

#### Implementation Result in Matlab

Using the initialization condition mentioned in section 3.3.1, the training process for 720 160-dimensional inputs took 8 iterations to converge. The mean waveform and priors of each cluster are show in Figure 3-6. Figure 3-7 shows the cluster results of the 720 training data using the GMM.

As you can see in Figure 3-7, all the main clusters have been recognized. Cluster 1 and Cluster 2 are the clusters with the most data. Cluster 3 and Cluster 4 can be regarded as the noisy versions of Cluster 1 and Cluster 2. Their mean vectors are very close to Cluster 1 and Cluster 2. But their waveforms are noisier, thus the covariance of these two clusters are bigger. Cluster 5 has its own specific spike shape. It has the least number of spikes among the five clusters so it has the smallest prior probability. Overall, this 5-component Gaussian Mixture Model can pretty well model the whole data set from visual inspection.

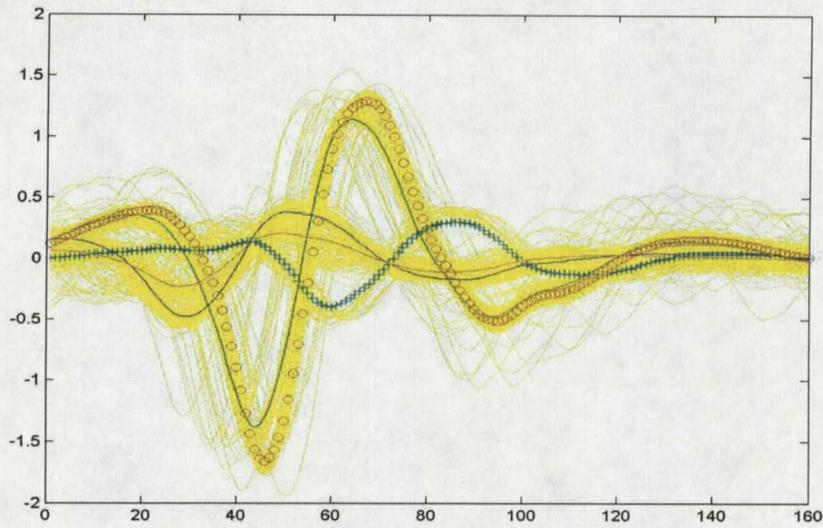


Figure 3-6 Clustering result from Matlab.

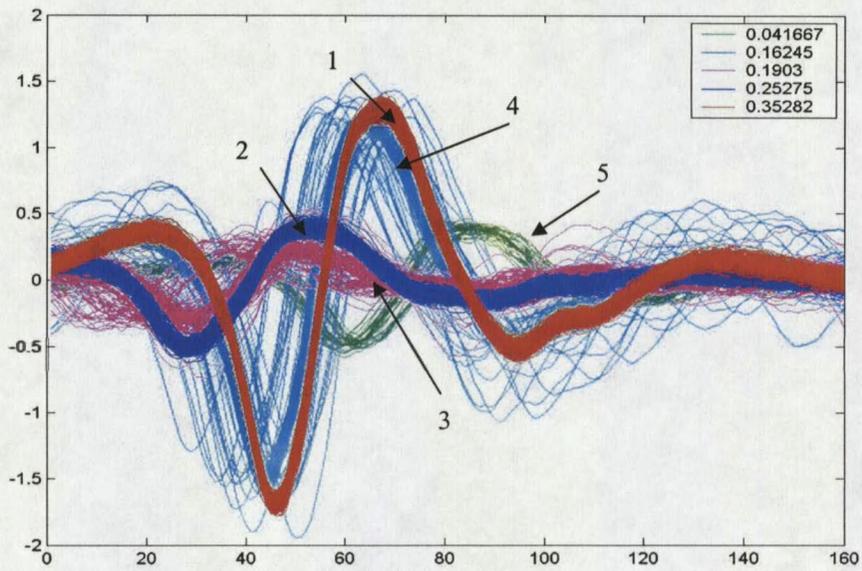


Figure 3-7 Clustering result for the training data. Five clusters are labeled using different color.

### Matlab Performance

Since we want to implement a high-speed spike sorting system where the models are trained as fast as possible, we have to know how fast this algorithm can run on a PC. As described in Chapter 2, the whole spike sorting process can be divided into two phases: the model training phase and the spike classification phase. We tested the algorithm using the newest version (Version 6.5 Release 13) of Matlab on different PCs. The test results are shown in Table 3-1.

Table 3-1 Performance of the spike sorting system on several PCs

	Execution Time				Average	Best
	Pentium III 1.2G <sup>1</sup>	AMD 1.37G <sup>1</sup>	AMD 1.2G Duo <sup>2</sup>	Pentium IV 3.3G <sup>3</sup>		
Training (s)	8.33	3.42	1.45	1.31	3.22	1.31
Classification (720 data points) (s)	0.42	0.14	0.05	0.04	0.16	0.04
Classification (per data point) (ms)	0.58	0.19	0.06	0.05	0.21	0.05

<sup>1</sup> The Pentium III 1.2G and AMD 1.37G systems both have 512 MB DDRAM.

<sup>2</sup> The AMD 1.2G Duo system has 4 GB DDRAM.

<sup>3</sup> The Pentium IV 3.3G system has 1GB DDRAM.

The performance of making classifications on a PC is pretty good. With the existing models, a spike waveform can be classified within less than 0.2 ms. This training process can meet the real time requirement since the typical neural spike period is around 2-3 ms. However, the training process ran much slower than the classification process. The best performance we got was 1.31 seconds while the average speed was around 3 seconds. This is quite slow comparing to the average neural spike period. The slow speed is due to the intensive computation needed for the

training of the Gaussian Mixture Model based on the EM algorithm. We needed the training process to be as fast as possible so that minimum time is spent in the training phase. As a result, PCs are not as fast as we would like in order to do this.

In the rest of the thesis, we will focus on how to speed up the training process of the EM algorithm. In next chapter, the parallel DSP implementation will be introduced. Later an FPGA implementation will also be presented.

## CHAPTER 4

## PARALLEL DSP IMPLEMENTATION

Introduction

To speed up the GMM based spike sorting system, the first approach we used was to implement the EM algorithm on a parallel DSP system with 4 DSPs. Digital Signal Processors have become more powerful with the increase in speed and size of the on-chip memory. Furthermore, some modern DSPs are optimized for multiprocessing. The Analog ADSP-21160M DSP we used for our project has two types of integrated multiprocessing support, which are the link ports and a cluster bus. We used Bittware's Hammerhead PCI board to test our implementation. The board contains four Analog ADSP-21160 floating-point DSPs. We wrote a parallel version of the EM algorithm to take advantage of the multiprocessing capability of the board.

In the next section, the structure and components on the Hammerhead board will be described. The software we use to develop the system will also be introduced. In the rest of Chapter 4, we focus on how to parallelize the EM algorithm and how to effectively implement it on a parallel DSP system. In the end of this Chapter, the performance of the EM implementation on this DSP system will be presented.

HardwareSHARC ADSP-21160M DSP Processor

On the Hammerhead PCI board, there are four high performance 32 bit floating point DSP processors, namely the Analog SHARC ADSP-21160s. This DSP features

4 Mbits on-chip dual-ported SRAM and the Super Harvard Architecture with the Single Instruction Multiple Data (SIMD) computational engine. Running at 80MHz, ADSP-21160 can perform 480 million math operations (peak) per second.

The ADSP 21160 has two independent, parallel computation units. Each unit consists of an ALU, multiplier, and shifter. In SIMD mode, the parallel ALU and multiplier operations occur in both processing elements. The ADSP 21160 has two independent memories - one for data and the other for program instructions. Two independent address generators (DAG) and a program sequencer supply address for memory access. ADSP 21160 can access both data memory and program memory while fetching an instruction. In addition, it has a zero overhead loop facility with a single cycle setup and exit. The on-chip DMA controller allows zero-overhead data transfers without processor intervention.

The ADSP 21160 offers powerful features to implement multiprocessing DSP systems. The external bus supports a unified address space that allows direct interprocessor accesses of each of the ADSP 21160's internal memory. Six link ports provide a second method of multiprocessing. Based on the link ports, a large multiprocessor system can be constructed in a 2D or 3D fashion.

#### Bittware Hammerhead PCI board and VisualDSP++

The structure of Bittware Hammerhead PCI board is shown in Figure 4-1. This board has four Analog Device's ADSP-21160 processors, up to 512 MB of SDRAM, 2 MB of Flash Memory, and two PMC mezzanine sites. Bittware's FIN ASIC is used to interface the ADSP-21160 DSPs to the 64-bit, 66 MHz PCI bus, the Flash memory, and a peripheral bus. The Hammerhead-PCI's four DSPs can communicate with the

host PC through the PCI bus. Four of the link ports are connected to other DSPs, which allow the DSPs to communicate through the link ports. Each processor is also connected to a common 50MHz, 64-bit ADSP-21160 cluster bus, which gives it access to the other three processors and to up to 512 MB of SDRAM.

In our application, we used all four ADSP-21160 processors on the board as well as the 256 MB SDRAM. We used the cluster bus to communicate between four DSPs and SDRAM. The cluster bus approach is relatively easy to implement. It was also faster than using the link ports.

We used VisualDSP++ 3.0 to develop the program for the Hammerhead PCI board. Most of the code was written in C. Small sections, such as the DMA transfer routine were written in assembly language. All the C functions we used were optimized for the ADSP-21160's SIMD mode.

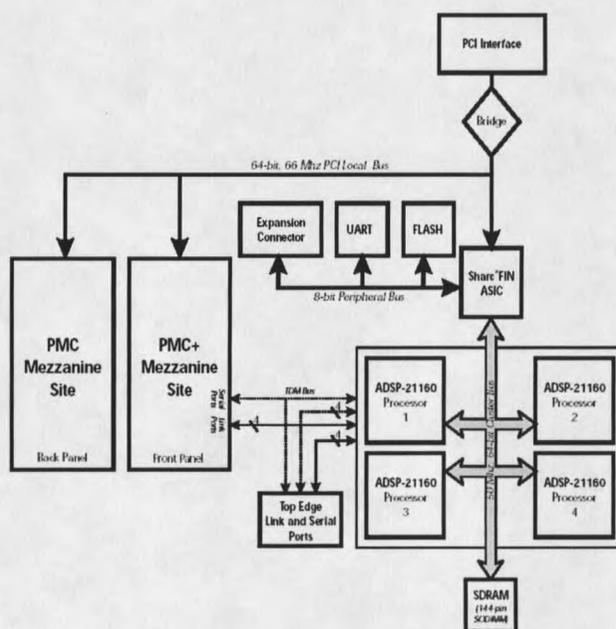


Figure 4-1 Block diagram of Hammerhead PCI system

### Analysis of the EM Algorithm

To parallelize the EM algorithm, we needed to find out where the computational intensive parts of the EM algorithm were. These parts should be mapped into multiple DSPs so that each DSP can share part of the computational load. Thus, by handling these parts simultaneously, we can speed up the whole system and enhance the performance.

We used the *profile* command in Matlab to analyze the EM algorithm. The *profile* function records execution time for each line in the code as well as for each function call in a Matlab (.m) file. By comparing the execution time, we can know which part of the algorithm Matlab has spent the most time on. The most computational intensive parts then should be parallelized on multiple DSPs.

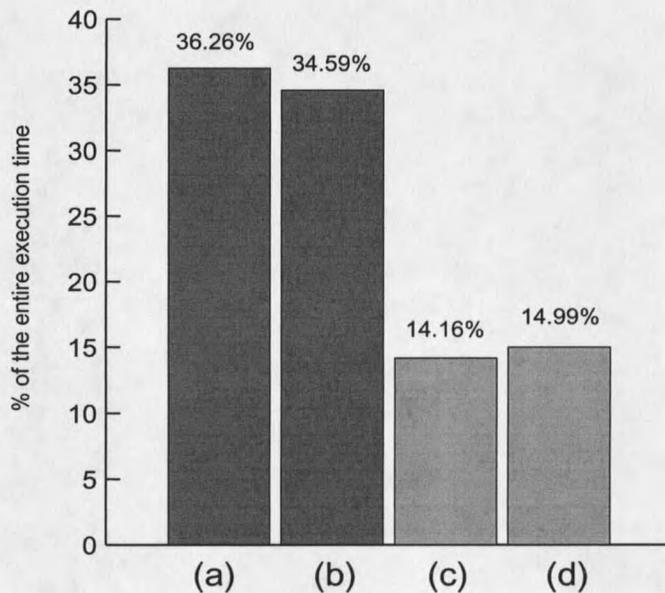


Figure 4-2 The diagram shows most computational intensive parts in the EM algorithm and their execution percentage in the whole process. (a) Calculation of  $p(x_i | o_i)$ . (b) Update of means and covariance matrices. (c) Calculation of  $p(o_i | x_i)$  and  $\zeta(x_i | O)$ . (d) Rest of the algorithm.

Figure 4-2 shows the most computational part in the EM algorithm. The calculation of  $p(x_i | o_i)$  and the update of means and covariance matrices take about 70% of all the execution time. These two parts should be implemented on multiple DSPs. The calculation of  $p(o_i | x_i)$  and  $\zeta(x_i | O)$  and rest the algorithm is relatively less computationally intensive. A close look at parts (a) and (b) shows that most of the computational operations are multiply and multiply-accumulation type of instructions. Thus the floating point MAC inside the ADSP-21160 will be able to speed up the operations.

#### Parallel Transformation of the EM Algorithm

To fully use the computational power of the four DSPs on the Bittware board, the EM algorithm was transformed to a parallel version. The most straightforward way is to map the computation of each cluster onto each single DSP. However, there were several drawbacks of this approach.

First, there are five clusters in our model while there are only 4 DSPs on the Hammerhead board. Hence we have to put the computation of two clusters into one DSP creating an uneven computational load. This will reduce the speed of the whole system since three DSPs have to wait while the other DSP deals with the extra computation for the second cluster. Secondly, the update of parameters is not independent among all the clusters. Before updating the parameters of each cluster, the probability of each data in the current model has to be calculated. In Eq. 4-1 the probability of each data in each cluster has to be added together across all the clusters. So at this point all the middle results have to be sent to one DSP. Last but not the least,

the limited on-chip memory requires the DSP to read data from external memory. If you look closely at the algorithm, for the calculation of  $p(x_i | o_i)$ , as well as the update of the means and covariance, the whole data set is needed. In other words, during these calculations, the DSPs have to get the value of the whole data and put them into ALU for computing. However, with the limitation of the internal memory size, it is impossible to store all the data set inside a single DSP. The only option is to read the data set from external RAM, i.e. the on-board SDRAM. These operations will require extra overhead and will slow down the process. They can also disable the SIMD capability of DSP. Since all the DSPs and SDRAM share the same 50MHz cluster bus, multiple simultaneous uses of the bus will cause extra delays and even bus contention.

Due to these drawbacks, we decided to choose another approach. We divide the whole data set into small blocks and each processor handles one block of data. This way, we eliminated the overhead of transferring data from external memory. With all the data inside the DSPs, we fully used the SIMD mode of the DSPs to accelerate the process. This also allowed us to divide the computation evenly for each DSP regardless of the cluster number. By being independent on the cluster number, this approach will allow scaling to more DSPs. We still need to transfer data between DSPs due to the dependence of updating the parameters. But the transfer is limited to middle results of the parameters which is much smaller in size comparing to the whole data set. This transmission can be monitored by one master DSP. DMA mode was used to accelerate the parameter transfers.

We divided the whole data set into four blocks. Each block contained 180 data points. These blocks were fitted into a single DSP's internal memory. One of the four DSPs was considered to be the master while other three were slaves. The master did the same computation tasks as the slaves except for two additional tasks: cross-data calculation and the management of DMA transfers. Whenever cross-data calculation was needed, all the middle results were sent to the master from the slaves through the cluster bus. When the master finished the calculation, the results were sent back to the slaves. If the DMA mode was used, the master was the only DSP that had the control of all the DSP's DMA controllers. Synchronization between the DSPs was handled via semaphores. Semaphores were also used to prevent DMA bus contention. Table 4-1 shows all the semaphores that were used in the program. The master semaphore was a global semaphore that all slave processors read to know the status of the Master processor. Each slave processor had its own local semaphore which told the Master DSP their status.

Table 4-1 List of semaphores and their functions.

	Function of each semaphore	
	SET (1)	CLEAR (0)
Master Semaphore	Master Processor is running	Master task finished
Local Semaphore 1-3	Slave Processors is running	Slave tasks finished
DMA Semaphore	DMA bus available	DMA bus occupied

The system diagram is shown in Figure 4-3. The algorithm was divided into 6 phases. The phases in shaded blocks were implemented on multiple DSPs. Notice that the master processor acts also like a slave while doing the parallel computation. Most of the computational intensive parts, which have been discussed in section 4-2, were

implemented on multiple DSPs. The parts implemented solely on the master processor were just basically additions. So, during the whole training process of the EM algorithm, most of the time, all four processors were running at full speed.

Another significant amount of time was spent on transferring data between DSPs. The cluster bus ran at 50MHz and was shared by the all four DSPs as well as the on-board SDRAM. We first let the DSPs resolve the bus contention automatically. The on-chip arbitration logic of ADSP- 21160 processors determined which DSP was the current bus master based on the Bus Request (BR) pins of the DSPs. Multiple DSP implementations can be done in this mode. However the approach was about 700 times slower than the other approaches because of the bus contention issue.

We then set a semaphore to exclude the use of the cluster bus. By reading this semaphore, only one DSP could access the bus at any time. This gave a much better performance than the first approach.

Finally, we used the DMA transfer mode to transfer parameters between the DSPs. The master processor was in charge of the DMA controllers of all the processors. Whenever a data transfer was needed, the master first made sure that no DSP was using the bus. The master then wrote into the registers of both the transceiver and receiver's DMA controller. After the DMA transfer is initialized, no other DSP could access the cluster bus until this DMA transfer finished. This sped up the process and gave the best performance. The performance comparison will be shown in the next section.

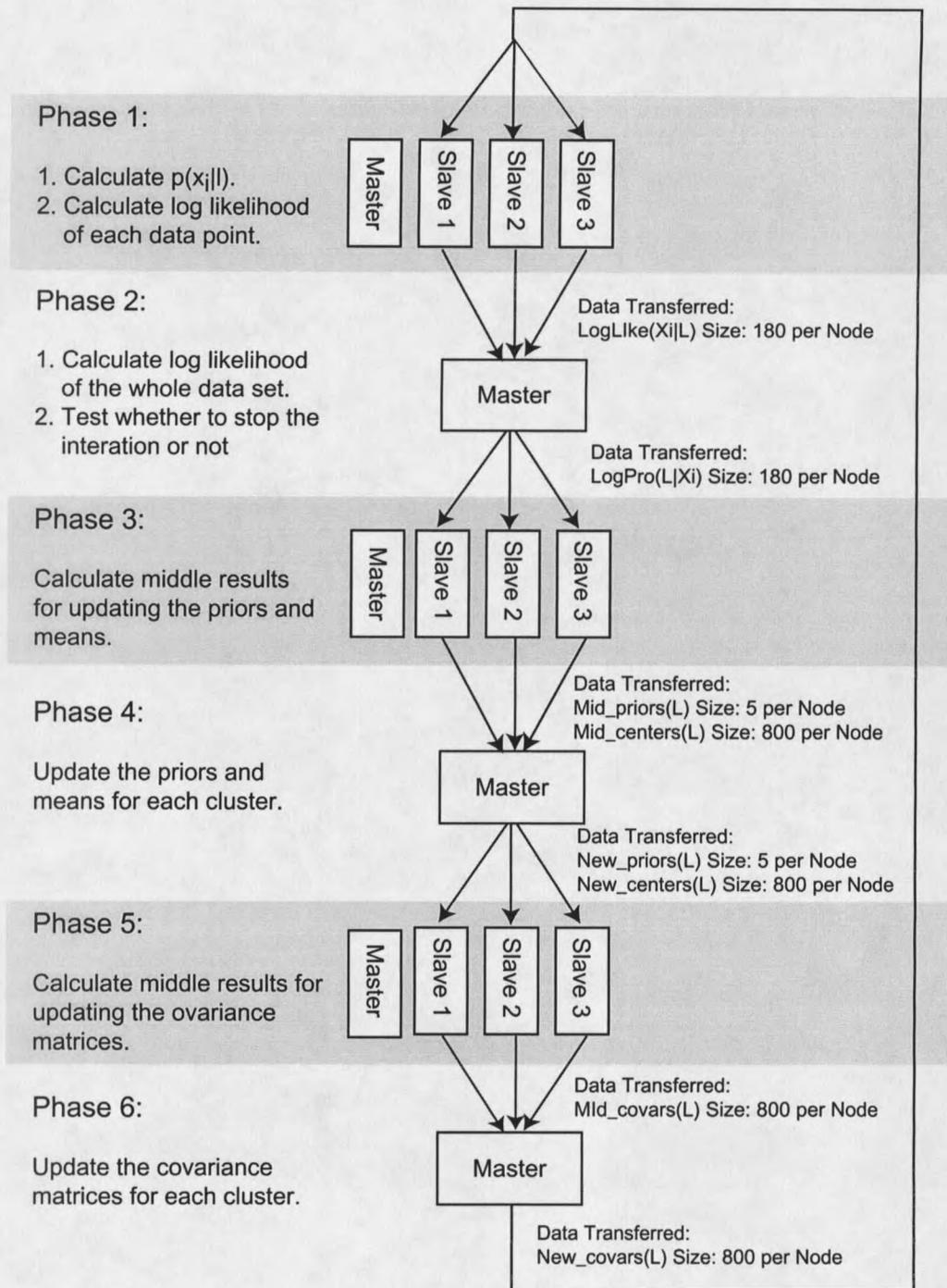


Figure 4-3 Diagram of multiple DSP implementation of EM algorithm

Performance on the Parallel DSP Platform

Table 4-1 gives the execution time for 8 iterations of the EM algorithm until it converged. Performance on a single DSP and multiple DSPs are both shown. Also different data transfer methods are compared. The results show that the computational speed of the 4-DSP system was almost 4 times better than single DSP. Also, the DMA transfer sped up the whole system and gave the best performance. Notice that the bus contention auto resolution method in the multiple DSP system significantly slowed down the performance of the whole system. Using semaphores was necessary to accelerate the data transferring process.

Table 4-2 Execution time for 8 iterations of EM algorithm on single and 4 DSP system

Data Transfer Methods	Single DSP (sec)	Four DSP (sec)
Normal Read	14.5	13.5
DMA transfer	15.9	3.4
Bus Contention	N/A	> 7000

Table 4-3 shows the performance of the same EM application between the DSP and Personal Computer (PC) configurations. All performances on the PC were from Matlab implementations. The performance of the parallel DSP implementation was at the same level as the average PC configuration but not as good as the high end PCs.

Table 4-3 Performance of the EM algorithm on DSP system and PCs

	Single DSP	Four DSP	PC Average <sup>1</sup>	PC Best <sup>2</sup>
Execution Times	13.5	3.4	3.22	1.31

<sup>1</sup> Average from following four systems: (1) AMD 1.37GHz, 512 MB memory; (2) Pentium III 1.0 GHz, 512 MB memory, (3) AMD 1.2GHz Duo Processor, 4 GB memory; (4) Pentium IV 3.2 GHz, 1 GB memory

<sup>2</sup> Best performances from Pentium IV 3.2 GHz system with 1GB memory.

<sup>3</sup> VisualDSP++ 3.0 was used in the DSP implementation.

There are two reasons for why the parallel DSP method was similar if not slower than the PCs. First, the core speed of DSP is much less than the high end PC CPUs. The core speed of ADSP 21160 is running at 80MHz while the core clock of Pentium IV is running at 3.3 GHz. The high-end PC is about 40 times fast than the DSP. In spite of the optimal structure of DSP core for signal processing operations, the overall performance of floating point operations on a high end PC is much better than on individual DSPs. Another reason is the bus speed. Since the lack of on chip memory of DSPs, the data has to be transferred between DSPs through cluster bus. The speed of cluster bus is at 50MHz which is much slower comparing to the front bus on the PCs which is up to 133MHz. Although using the DMA transfer mode, the cluster bus is still the bottle neck of speed in the multiple DSP system.

Performance on the DSP system could probably be enhanced by more manual tuning on the assembly language level. Using DSPs with high core speed or clustering more DSPs can also improve the performance. However, the improvement will be very limited and too much time and effort will be required. So, to gain better performance for the real time spike sorting system, we turned to another solution, namely Field Programmable Gate Arrays (FPGA).

In the next Chapter, we will implement our EM algorithm on a Xilinx Virtex II FPGA system. To implement the EM algorithm on FPGAs, a fixed point version of the algorithm was first developed. Also, specialized math operations have to be transformed to facilitate the implementation of the algorithm on a FPGA. These problems will be addressed in the next Chapter and the implementation results will be presented.

## CHAPTER 5

## FPGA IMPLEMENTATION

Introduction

Field Programmable Gate Arrays (FPGAs) are reconfigurable hardware that can be changed in electronic structure either statically or dynamically. Because the implementation is on the actual hardware, FPGAs can obtain a significant performance improvement over the software implementations based on DSP chips or PCs [30].

One drawback of a FPGA implementation is that, for complex algorithms like the EM algorithm, it usually takes a long time to design and optimize the algorithms for the FPGA structures. We used a recently developed tool, namely AccelFPGA [34], to shorten the design time. AccelFPGA can compile Matlab code directly into VHDL code. By using AccelFPGA, we could focus on optimizing the algorithm in Matlab without dealing with hardware details inside the FPGA. The design cycle of the FPGA implementation using this method can be reduced from months to weeks. The FPGA implementation through AccelFPGA is not as optimized as directly VHDL coding. However, for a prototype design, AccelFPGA can provide a time-efficient FPGA solution with reasonably good performance. We will show that our implementation on a FPGA gives much better performance than on either the PC or the parallel DSP platforms.

The structure of Chapter 5 is as follows: In the next section, the internal structure of Xilinx 3000 FPGA will be introduced. Then we will introduce the AccelFPGA

compiler for FPGAs. In section 5.4, we address some practical problems of FPGA implementations. A fixed point version of the EM algorithm is presented and compared to the floating point version. The transformation of mathematics operations and parallelism of the algorithm for FPGA implementation are also discussed. In the last section, the performance of the EM algorithm on the FPGA platform is presented and compared to the parallel DSP platform and PCs.

### The FPGA and Its Structure

#### Field Programmable Gate Arrays

A Field Programmable Gate Array is an ASIC where logic functions can be changed to suit a user's needs. The circuit elements within these devices can be configured repeatedly because the current configuration is stored in SRAM cells [31]. The FPGAs can be configured to operate in a nearly unlimited number of user-controlled ways by downloading various configuration bit streams.

The FPGA devices are usually organized into generic logic blocks that can perform a limited number of logic functions. The specific function being performed is determined by the configuration bits preloaded into the hardware. These logic blocks are connected by programmable routing, which is also determined by the configuration bits.

The particular structures of the logic blocks and routing are different inside the FPGAs from different manufacturers. The FPGA we use for our application is the Xilinx Virtex II series FPGAs. The structure of this FPGA is described in the following section.

### Structure of Xilinx's Virtex II FPGA

The Xilinx Virtex II family is a FPGA developed for high-speed high-density logic designs with low power consumption. As shown in Figure 5-1, the programmable device is comprised of input/out blocks (IOBs) and internal configurable logic blocks (CLBs). In addition to the internal configurable logic blocks there are three major elements:

- 1) Block SelectRam memory provide 18 Kbit of dual-port RAMs
- 2) Embedded 18 x 18-bit dedicated multipliers
- 3) Digital Clock Manager (DCM) blocks

The functionality of these elements will be introduced later. All of these elements are interconnected by routing resources. The general routing matrix is an array of routing switches. Each programmable element is tied to a switch matrix, allowing multiple connections to the general routing matrix. All programmable elements, including the routing resources, are controlled by values stored in static memory cells. These values are loaded in memory cells during configuration and can be reloaded to change the functions of the programmable elements.

Configurable Logic Blocks (CLB) are the basic functional units of Xilinx FPGAs. These blocks are organized in an array and are used to build combinational and synchronous logic designs. Each CLB element comprises 4 similar slices. The four slices are split in two columns with two independent carry logic chains and one common shift chain. Figure 5-2 shows the details of a slice inside one CLB. Each slice includes two 4-input function generators, carry logic, arithmetic logic gates, wide function multiplexers and two storage elements. Each 4-input function generator

can be programmed as a 4-input LUT, 16 bits of distributed SelectRAM memory, or a 16-bit variable-tap shift register. Each CLB is wired to the switch matrix to access the general routing matrix to be interconnected to other CLBs.

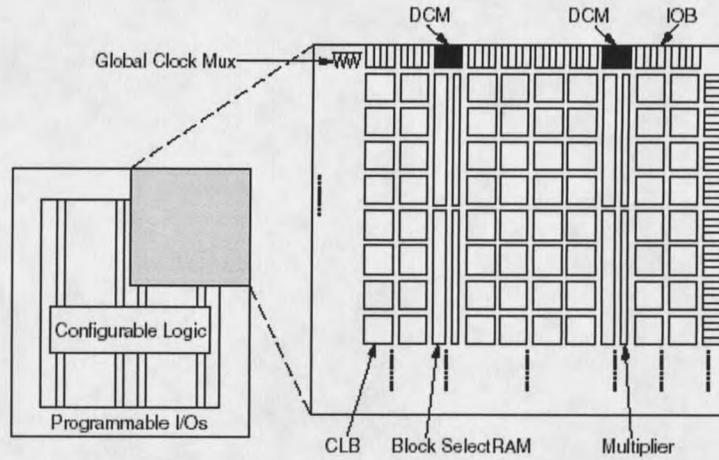


Figure 5-1 Virtex II architecture overview [31].

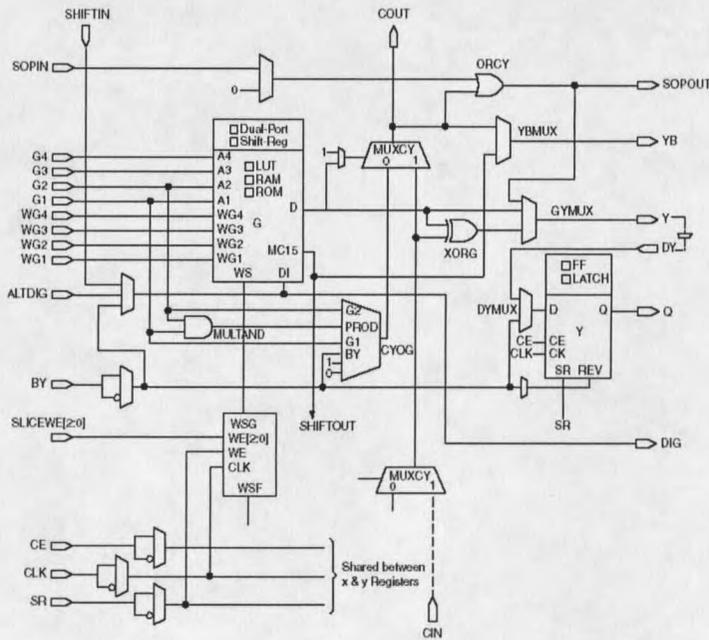


Figure 5-2 Slice Structure of Virtex II FPGA [31].

The Virtex II devices incorporate large amounts of 18 Kbit block memories, called Block SelectRAM or BlockRAM. Each Virtex II blockRAM is an 18 Kbit true dual-port RAM with two independently controlled synchronous ports that access a common storage area. The blockRAM supports various configurations, including single- and dual-port RAM and various data/address aspect ratios. Supported memory configurations are shown in Table 5-1.

Table 5-1 Supported configuration of the block SelectRAM

16K x 1 bit	2K x 9 bits
8K x 2 bits	1K x 18 bits
4K x 4 bits	512 x 36 bits

The Virtex II devices also incorporate a number of embedded multiplier blocks. These multiplier blocks are 18-bit by 18-bit 2's complement signed multipliers. Some of the interconnection to the Switch Matrix is shared between the BlockRAM and the embedded multiplier. This sharing of the interconnection is optimized for an 18-bit-wide BlockRAM feeding the multiplier and allows a fast implementation of a multiplier-accumulator (MAC) function which is commonly used in DSP applications.

The Virtex II family is comprised of 11 members, ranging from 40K to 8M system gates. The FPGA we used for the thesis was the XC2V3000, which features 3M system gates. Table 5-2 shows the specifications of the XC2V3000 FPGA.

Table 5-2 The specifications of the Virtex II XC2V3000 FPGA

Device	System Gates	CLB Slices	Multiplier Blocks	SelectRAM Blocks	DCMs	Max I/O pads
XC2V3000	3M	14336	96	96	8	720

### AccelFPGA

FPGA programming can be done using a Hardware Descriptive Language (HDL) such as VHDL or Verilog. These HDLs are low level languages which have a very fine control of the actual hardware implementation. However, for the complex applications, the HDL coding process will be tedious and error-prone and requires costly debugging iterations. Furthermore, a lot of low-level work is repetitive and can be automated. To solve this problem, many researchers have focused on how to raise the level of abstraction to a general purpose programming language such as C/C++ [32] or Java [33]. Recently, a compiler has been developed that can take Matlab code and generate optimized hardware for a FPGA. The AccelFPGA Compiler V1.6 was used for this thesis.

Figure 5-3 shows the basic steps in the process of converting an algorithm in MATLAB into an FPGA implementation using AccelFPGA. The first step is to convert the floating-point model of the algorithm into a fixed-point model that meets the algorithm's signal fidelity criterion. Next, the user adds compiler directives for AccelFPGA. These directives can specify the usage of FPGA's embedded hardware resources such as the BlockRAM and the embedded multiplier blocks. The third step is the creation of the RTL model in VHDL or Verilog and the associated testbenches which are automatically created by the AccelFPGA compiler. The fourth step is to synthesize the VHDL or Verilog models using a logic synthesis tool. The gate-level netlist is then simulated to ensure functional correctness against the system specification. Finally the gate-level netlist is placed and routed by the place-and-route appropriate for the FPGAs used in the implementation.

One advantage of AccelFPGA is that bit-true simulations can be done in the Matlab environment. Testbenches are automatically generated along with the simulations. You can use these testbenches later in the hardware simulation and compare the results with the Matlab simulation. As a result, it is very easy to know whether your hardware implementation is correct or not.

Since Matlab is a high level language, the compiled implementation may not give as good performance as direct HDL coding. However, by using proper directives, you can specify the parallelism of your algorithm and maximize the usage of the on-chip hardware resources. Hence, you can get a reasonably good performance of your implementation.

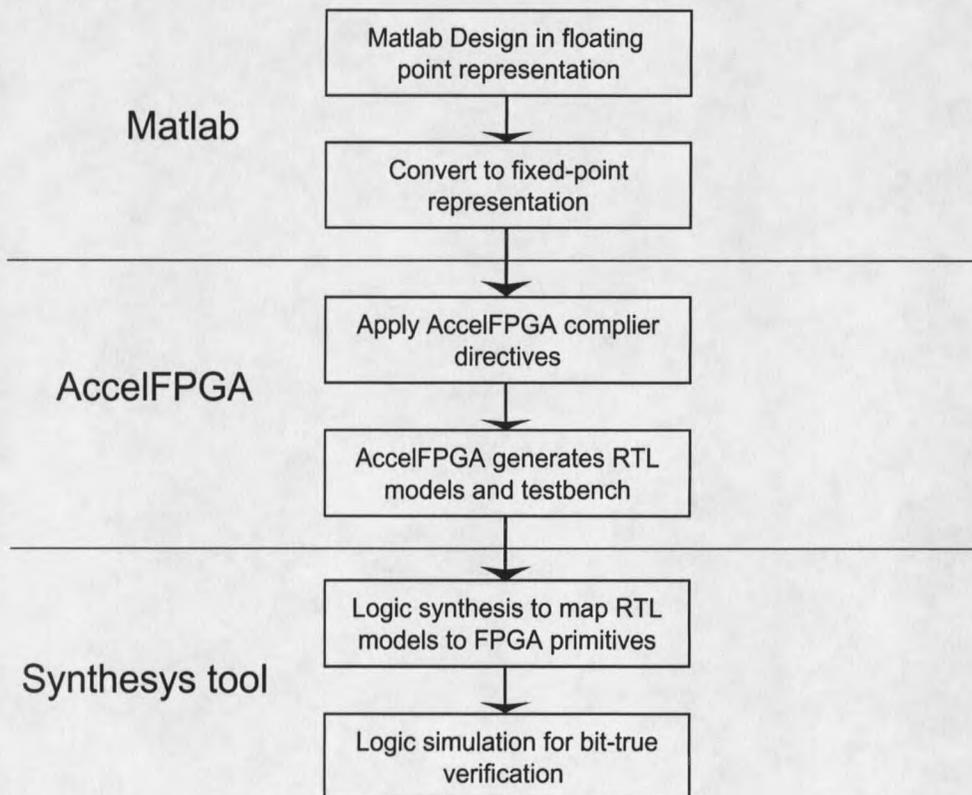


Figure 5-3 Top-down design process using AccelFPGA. The versions of all the tools used in our FPGA implementation can be found in Table 5-5.

### The Fixed Point Representation of the EM Algorithm

To use AccelFPGA for the hardware implementation, the first step is to convert the algorithm from a floating point into a fixed point representation. The main difference between these two representations is the limited precisions. For the EM implementation, the problem becomes whether the algorithm will still converge to the right result as in the floating point domain. We used the *quantizer* function in Matlab to change the algorithm into a fixed point format and tested the whole algorithm to make sure it gave the correct results.

The most important part of the fixed point implementation is to determine the word length and binary point of each variable in the algorithm. AccelFPGA can support up to 50-bit wide words. Although long word length can provide better precisions, more bits will be needed to store and process each variable. Thus, more resources on the FPGA will be required and the system performance will be reduced. To maximum the system performance, we should use as short a word length as possible while still maintaining enough precision for the algorithm. We chose to use 18 bits for normal variables and 30 bits for accumulation results. Variables with 18 bit word length can be fitted into the BlockRAM blocks and are easy to feed into the embedded multiplier in the FPGA. Since the data dimension was high for our algorithm, it was likely to generate a large number after many accumulations. Using 30 bit word length prevented overflow problems for the accumulation results.

After deciding on the word length, the next step was to choose the binary point for each variable. This step actually determined the precision we got from the FPGA implementation. Since some parts of the algorithm have been transformed into the log

domain, the dynamic ranges of each variable are quite different. So with the same word length, each variable will need its own binary point to meet the precision requirement. This would be quite hard to implement in a FPGA if an HDL coding approach was used. Whenever a math operation occurs between variables with different binary points, a shift is needed to align these variables. A lot of work has to be done in the low level HDL programming to make sure every math operation has the binary point aligned correctly. But in AccelFPGA, the compiler will automatically align the binary point. This allows the binary point of each variable to be easily changed and the algorithm tested to make sure the fixed point implementation is working correctly. The newest version of AccelFPGA (V2.0) can automatically find the suitable word length and binary point for your algorithm. However, at the time the thesis was done, this feature was not available and all the binary points were tuned manually in AccelFPGA V1.6.

#### Using Lookup Tables

The structure of the FPGAs is optimized for high-speed fixed-point addition, subtraction and multiplication. However, a number of other math operations such as division or exponentiation have to be customized.

Using Taylor series is an option which can transform the complicated operation into simple additions and multiplications. However, for the EM algorithm, the dynamic ranges of these operations are quite large so that it will require a very high order Taylor series to obtain good approximations. Another way is to use a look up table (LUT). Knowing the dynamic range of these math operations, we can use a one-to-one look up table to approximate them. By dividing the dynamic range input into

substantially smaller intervals, this approach will gain enough precision for the algorithm to converge. The BlockRAM on the Virtex II FPGA can be used to store the content of the look up tables and their address bus can be used as the look up table inputs. The look up table operations can be done within one system clock cycle based on the BlockRAMs.

Nonlinear mapping and output interpolation can be used to improve the LUT performance. Here, we just used a single linear LUT approach, in which we divided the input data space evenly into small spaces and mapped every point in the input space to the output space. Two factors, input precision and the LUT size have to be determined for each LUT depending on the input dynamic range. These two factors have the following relationship.

$$LUT \text{ size} = \frac{\text{Dynamic Range}}{\text{Precision}} \quad \text{Eq. 5-1}$$

Ideally, we want the dynamic range to be as large as possible while the precision as small as possible, which leads to an infinitely large LUT size. However, in most of the real world applications, the input dynamic range is limited to a certain range. The precision requirement may vary for different applications. For the EM algorithm, all we needed was to estimate the parameters for the GMM and be able to classify the neural spikes. As long as the EM algorithm converges to the right solution, the precision of each variable is not extremely important. Hence, a reasonable size of the LUT can be used for each math operation in the EM algorithm and can be implemented on the Virtex II FPGA platform.

We built the LUTs using the BlockRAM in the Xilinx Virtex II FPGA. As described in section 5.2.2, the BlockRAM are 18 Kbit block memories. Each 18 Kbit block memory can be configured into various address/data aspect ratios as shown in Table 5-1. We chose the 1K 18bit configuration for our LUT implementation. The 18bit output bit width gave enough precision for the LUT outputs. Furthermore, the output width matched the embedded multipliers' input width which maximized the usage of the embedded multiplier resources.

After choosing the LUT size, which was 1024 (1K) in our case, we found the input precision by tracking the input dynamic range. Once the input dynamic range was determined, the precision was found by dividing the dynamic range using the LUT size. Then, we ran the training process of the EM algorithm to see with if a precision caused the EM algorithm to converge or not.

If problems occurred and the algorithm didn't converge, we had to improve the input precision by increasing the LUT size. There are two ways to do this. One approach was to increase the LUT size by shortening the output bit width. With a shorter word width, the BlockRAM can hold more than 1024 elements in a single bank. But the decreased output precision may not meet the output precision requirement. Another way was to use more than one bank of block SelectRAM to form one single LUT. Theoretically, you can use as many BlockRAMs as is available to build a LUT. However, additional resources and control circuit will be needed to linearly address all these memory blocks. These additional circuits will use more area of the FPGA and decrease the system performance. In our application, the 1024 LUT

size worked properly with all the operations so neither of the above methods were used.

An example of how the look up table approach actually worked in the FPGAs is shown below. We implemented the exponential operation in Eq. 5-2 using look up table approach.

$$p(x_i | o_i) = \exp(\log\_p(x_i | o_i)) \quad \text{Eq. 5-2}$$

First, we tracked the input/output dynamic range during the whole training process. Figure 5-4 shows the histogram of the input and output data. While the input dynamic range is quite large (0 to  $-10^4$ ), the output dynamic range was limited to [0 1]. This means quite a large span of input space are mapped to a small range of output space. This can be more clearly shown in Figure 5-5. The output of the exponential function decreases with the decrease of the input. While approaching zero, the output barely changes even though the input keep decreasing. Obviously, the input precision of large negative values will not affect the output since the output is almost zero. Knowing this, we can decrease the input dynamic range by setting a threshold so that any input smaller than a particular value will lead the output to be zero.

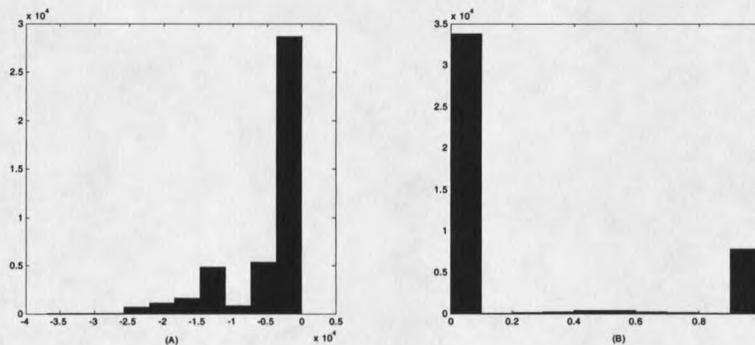


Figure 5-4 Histogram of the input and output data range for the LUT table. (a) The input. (b) The output.

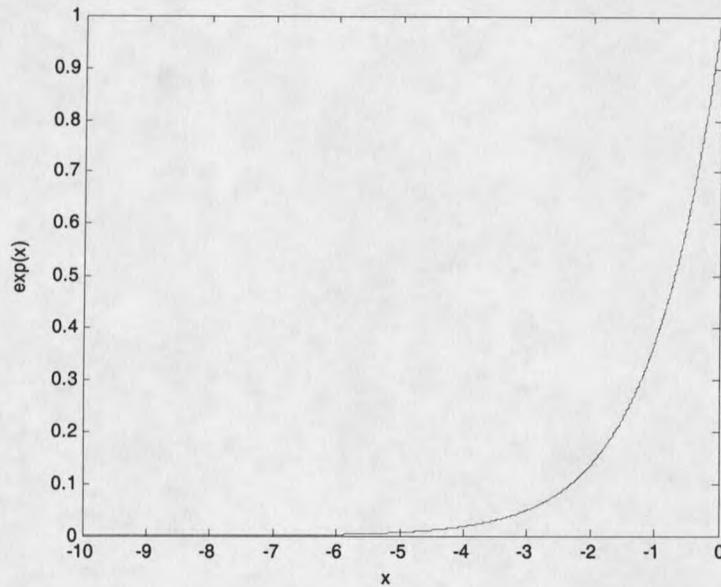


Figure 5-5 Input and output curve of the exponential operation.

We chose the threshold value to be -4 so that the dynamic range of the input was  $[-4\ 0]$ . Choosing the LUT size to be 1024 means the input precision can be determined by  $(0 - (-4))/1024 = 2^{-8}$ . We tested this precision by making sure the EM algorithm converged. Table 5-3 shows the difference between the floating point output and LUT outputs. The error was about 1% for each variable which had a small affect on the convergence of the EM algorithm.

Table 5-3 The error between the original floating point implementation and the LUT simulations.

The fixed-point LUT output	
Error per data	0.0126

Figure 5-6 shows how this LUT was implemented in the FPGA hardware. The input variables, which start from bit 4, represented the least precision  $2^{-8}$  and are used

as the input to the LUT. The bits after bit 4 are neglected since these bits are not required for the input. The content in the LUT is determined by Eq 5-3.

$$LUT(k) = \exp(-k \times 2^{-8}) \quad \text{Eq. 5-3}$$

in which  $k$  represented the decimal address of the block memory. Notice that the inputs of the LUT were positive which are the inverse of the real inputs. This was done in the previous calculation without additional overhead. Another thing to notice is the last element in the LUT is 0 instead of the result from Eq 5-3. The zero accounts for all the outputs with the inputs less than -4.

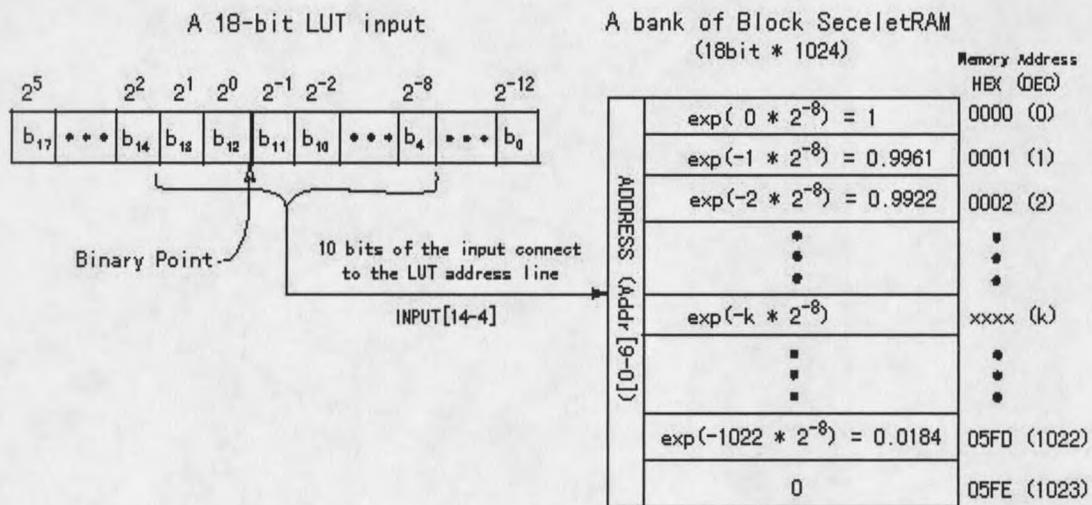


Figure 5-6 Diagram of implementing the LUT in block memory

All the other LUTs were determined and tested in Matlab in the similar way. A list of the LUTs is found in Appendix D.

### Parallelism Needed for FPGA Implementation

To fully use the computational power of the FPGA, we needed to parallelize the EM algorithm. In Matlab, programs are executed sequentially according to the program sequence. AccelChip offers directives which can tell the compiler when to parallelize the algorithm and when to use the embedded FPGA resources.

The following are some directives that have been used extensively in the Matlab program.

Table 5-4 List of directives can be used in AccelFPGA.

Directives	Purpose
REGISTER	Apply hardware registers for specific variables to add pipeline stages.
MEM_MAP	Map the array variable to a specific block memory on the FPGA
TILE	Specify an array of computations to happen concurrently.
USE	Specify the hardware resources such as embedded multiplier for the particular operations

The code in List 5-1 shows how to actually using these directives in the Matlab code. The operation in the *for* loop is parallelized using TILE directive. Block memories are used on each array variable and an embedded multiplier is used for each multiplication. By using these directives, the multiplication operations applying to all 5 clusters can be executed in one clock cycle.

## List 5-1 Example codes for using directives in Matlab

```

%!ACCEL TILE m_9
%!ACCEL MEM_MAP mid_centers(:,m_9) TO ram_s18_s18(m_9+20) AT 0
%!ACCEL MEM_MAP centers(:,m_9) TO ram_s18_s18(m_9+15) AT 0
%!ACCEL USE embedmults(m_9+25)
for m_9 = quantize(q_int,1:5)
    centers(d_3,m_9) = quantize(q_data,mid_centers(d_3,m_9) *
mid_priors_inv(m_9));
    mid_centers(d_3,m_9) = quantize(q_mid_centers,const_zero);
end

```

In the FPGA, all the computations among the five clusters were parallelized and ran at the same time. A lot of BlockRAMs were used to ensure that the multiple use of a single resource were possible. The embedded dedicated 18-bit 2's complement multipliers were used wherever multiplication is needed to give the highest system performance. Registers were added to pipeline the operations to obtain a high system clock rate.

The training data was streamed into the FPGA to implement the training process of the EM algorithm. The data has to be steamed in twice for each iteration step since both the E-step and M-step need the training data for the computation. The advantage of streaming the data is to make the algorithm independent of the data size since there is relatively little chip memory compared to PCs. The final results will were stored in the block memories inside FPGA (Figure 5-7).

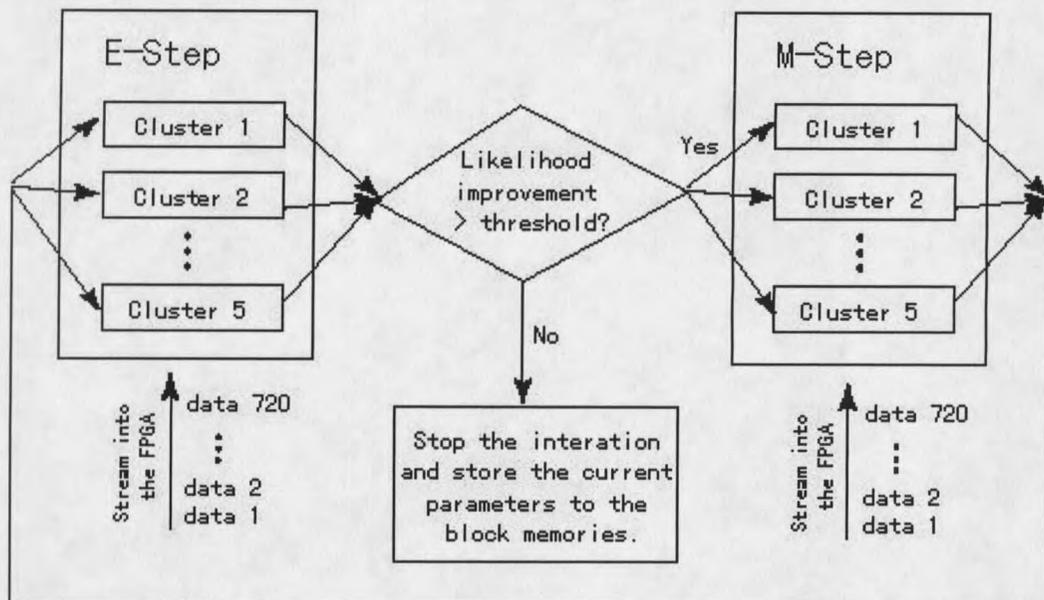


Figure 5-7 Diagram of the EM algorithm implementation in the FPGA

### Synthesis and Simulation

After generating the VHDL code from AccelFPGA, the next step was to simulate and synthesize the VHDL modules. The RTL VHDL code could be simulated in the simulation tools such as Modelsim using the testbenches generated by AccelFPGA. They could also be put into synthesis tools like ISE from Xilinx to generate the bit stream to be downloaded into the FPGAs. The toolchain used can be seen in Table 5-5.

Table 5-5 The toolchain used in the FPGA implementation.

Toolchain	Version
Matlab	V6.5 Release 13
AccelFPGA	V1.6
ISE	V5.2.02i
Modelsim	Starter 5.6e

First, a RTL level simulation was made to verify that the VHDL code was working correctly. Then, a post synthesis model was generated by the ISE synthesis tool from Xilinx. This post synthesis model is simulated again in the simulation tool Modelsim. All the simulation results were the same as the output of the fixed point Matlab implementation. This shown that the FPGA implementation of the EM algorithm was working correctly.

The system performance of the FPGA implementation can be seen in the synthesis report from the ISE tool. The highest clock rate of this EM system in Xilinx Virtex II 3000 platform is 62MHz (Table 5-7). The hardware resources used is shown in Table 5-5 and the actual floorplan inside the FPGA is shown in Figure 5-8.

Table 5-6 Device utilization summary of the Xilinx Virtex II FPGA.

FPGA resources	Number of used units	Number of available units	Percentage of the available resources
Slices	6378	14336	44%
Slice Flip Flops	4790	28672	16%
4 input LUTs	11473	28672	40%
bonded IOBs	293	720	40%
Embedded Multiplier	42	96	43%
BlockMems	58	96	60%
GCLK	1	16	6%

Table 5-7 The post-synthesis timing report.

System Parameters	Performance
Minimum Period	15.987 ns
Maximum Frequency	62.551 MHz
Maximum input arrival time	9.901 ns
Maximum output required time	7.333 ns

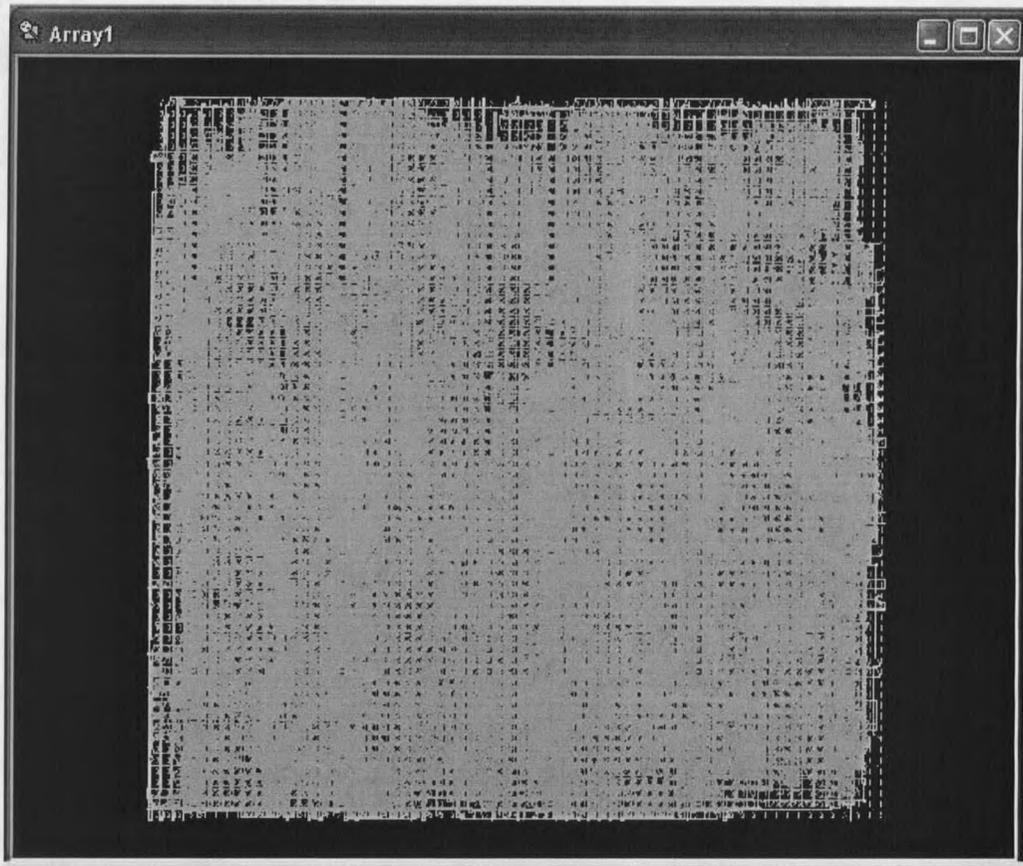


Figure 5-8 Floorplan of the EM algorithm implementation on a Virtex II FPGA

### FPGA Result Evaluation

Since we used fixed point data in the FPGA implementation, the precision of the output was not as good as the floating point implementation. The LUT application for the math operations furthermore decreased the precision of the output. We compared the final GMM parameters from the floating point implementation and the fixed point FPGA implementation.

Figure 5-9 (b) shows the plot of the result means of each cluster from the FPGA implementation. The difference is pretty small comparing to the floating point results

(Figure 5-9 (a)). Table 5-8 shows the average errors between the FPGA implementation and floating point implementation. We can see that the differences are small.

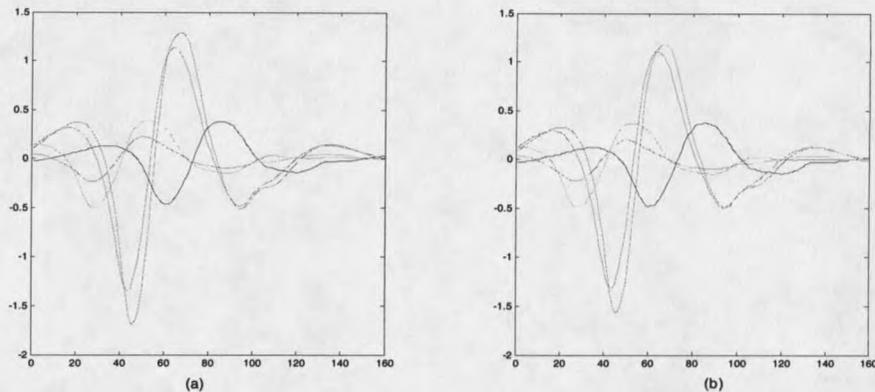


Figure 5-9 The mean vectors of the GMMs. (a) is the floating point version. (b) is the output of the FPGA implementation.

Table 5-8 The difference between the floating point output and FPGA output

GMM parameters	Error between floating point and FPGA output (per vector)
means	0.0170
covariance matrices	0.0019
priors	0.0147

We ran the same training data through the two models to see the classification results. 18 out of 720 data points are misclassified using the FPGA models when compared to the floating point implementation in Matlab. The classification rate is 97.25%. The misclassifications are shown in the confusion matrix in Table 5-9.

Table 5-9 Confusion matrix of fixed point spike sorting result for 720 neural spikes from 5 neurons. Overall correct rate is 97.25% comparing to the floating point result.

	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
Cluster 1	252	0	0	2	0
Cluster 2	0	182	0	0	0
Cluster 3	0	3	133	1	0
Cluster 4	12	0	0	105	0
Cluster 5	0	0	0	0	30

The confusion matrix shows that the most misclassifications happened between cluster 4 and cluster 1 (14 misclassifications). Figure 5-10 shows that cluster 1 and cluster 4 are heavily overlapped and probably represent the action potentials from the same neuron. The same situation exists for cluster 2 and 3 which have 3 misclassifications. Overall, a small number of misclassifications happened between very close clusters which would not greatly affect the final spike sorting results.

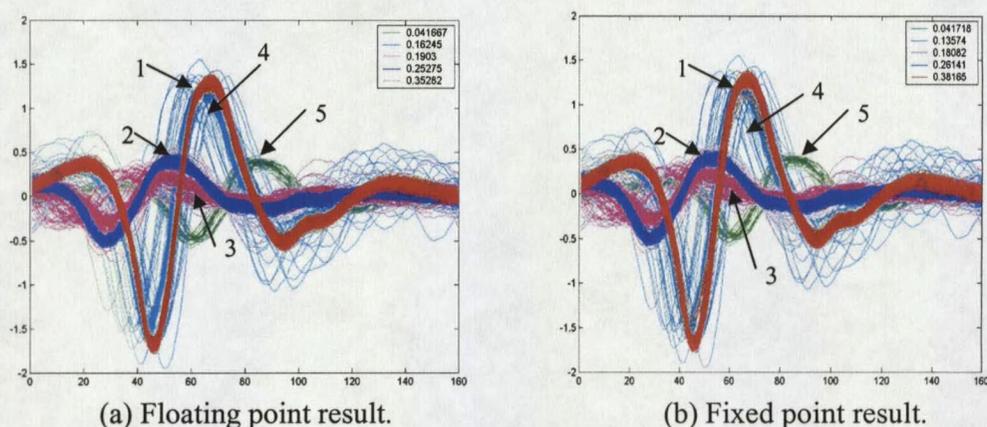


Figure 5-10 The clustering results of both floating point and fixed point implementations.

### FPGA Performance

In the post synthesis simulation, it took 60M clock cycles to finish one step of the EM algorithm in the FPGA. Running at 60MHz, it took  $60M \cdot 1 / 60M \cdot 8 = 0.08$  second to finish the training process. Table 5-10 shows that the FPGA implementation is 16.4 times faster than the fastest PC and 42.5 times faster than the parallel DSP implementation.

Table 5-10 Performance comparison between all the platforms.

	Single DSP	Four DSP	PC Average <sup>1</sup>	PC Best <sup>2</sup>	FPGA <sup>3</sup>
Execution Time (s)	13.5	3.4	3.22	1.31	0.08
Speedup Related to the best PC (times)	0.05	0.4	0.4	1	16.4

<sup>1</sup> Average from following four systems: (1) AMD 1.37GHz, 512 MB memory; (2) Pentium III 1.0 GHz, 512 MB memory, (3) AMD 1.2GHz Duo Processor, 4 GB memory; (4) Pentium IV 3.2 GHz, 1 GB memory

<sup>2</sup> Best performances from Pentium IV 3.2 GHz system with 1GB memory.

<sup>3</sup> AccelFPGA V1.6, ISE V5.2.02i and Modelsim Starter 5.6e were used in the FPGA implementation.

## CHAPTER 6

## CONCLUSION AND FUTURE WORK

Conclusion

This thesis presents the implementation results of a Gaussian Mixture Model training process on three different platforms. These platforms include PCs, multiple DSPs, and an FPGA. The Expectation Maximization algorithm was used to estimate the parameters of the GMM. The fastest implementation was obtained by using the Xilinx Virtex II XC2V3000 FPGA. By implementing the training procedure on a FPGA, the time to train models to be used in spike sorting will be shortened which will benefit neuroscience research. This will allow more time to be spent on collecting data.

Several practical issues related to the EM algorithm implementation were discussed. To prevent underflow problems caused by multiplications, we converted part of the EM algorithm into the log domain. Without prior knowledge of how many neurons were generating action potentials, we estimated the model number of the GMM using the likelihood function. The most computational parts of the EM algorithm were found and parallelized in both the multiple DSP and FPGA implementations.

For the parallel DSP implementation, we used four ADSP-21160 32-bit floating point DSPs. The bus contention between four DSPs significantly decreased the speed of the parallel DSP system. To prevent this, we set up a semaphore to exclude the usage of the cluster bus and used the DMA mode to accelerate the data transfer

among DSPs. However, because of the low DSP clock speed, the parallel DSP implementation didn't outperform the PC implementation.

For the FPGA implementation, a fixed point version of the EM algorithm was developed and simulated in Matlab. To implement the exponential function, a block memory based Lookup Table was used to approximate this functions.

We used a Xilinx Virtex II XC2V3000 FPGA for our system target. The Virtex II XC2V3000 FPGA has 96 18-bit 2's complement multipliers and 96 block memories. By using these embedded hardware resources, we efficiently parallelized the EM algorithm in the FPGA. The FPGA implementation was 16.4 times faster than a Pentium IV 3.2 GHz PC and 42.6 times faster than the 4-DSP implementation.

The work of this thesis shows the significant promise of using FPGAs for high-speed computational-intensive DSP applications. By using FPGAs, a significant performance improvement was achieved over the PCs and DSP processors without sacrificing system flexibility. It appears that with intensive I/O and parallel computational ability, FPGAs will become more widely used in the DSP world.

#### Future work

One direction of the future work could be to improve precision of the fixed-point implementation using a larger word length and different binary point positions. However, a larger word length will decrease the system performance of the FPGA since more hardware resources will be needed. The tradeoff between the word length and system speed has to be evaluated in order to find the optimal trade-off that will depend on implementation constraints.

The FPGA implementation in the thesis was translated directly from Matlab code using the AccelFPGA compiler. At this stage, the implementation could be improved by directly hand coding the algorithm in VHDL and even manually routing inside the FPGA. However, this would take much more work.

Another way to speed up the current system is to find the critical path of the FPGA implementation and add more registers. By increasing the pipeline depth of the critical path, we can increase the clock rate of the whole system. This could be explored in AccelFPGA by rewriting the Matlab code to produce more intermediate results and adding more compiler directives.

The current implementation on the Virtex II XC2V3000 FPGA used about half of the embedded hardware resources. More parallelism of the EM algorithm could be exploited by using more hardware resources in the FPGA. For example, in the training process of the each cluster, the whole training data set could be divided into smaller blocks. The computation of all the data blocks could be parallelized on the FPGA. Also, the use of multiple FPGAs could enable the more complex implementations, requiring more hardware resources than what is contained on a single FPGA.

**BIBLIOGRAPHY**

- [1] Hodgkin & A.L.Huxley (1952) A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117, 500-544.
- [2] F.Rieke, D.Warland, R.de Ruyter van Steveninck, & W.Bialek (1997) Spikes: Exploring the Neural Code. *MIT press, Cambridge, Massachusetts*.
- [3] J.G.Nicholls, A.R.Martin, & B.G.Wallace (1992) From Neuron to Brain. *Sinauer Associates, Inc. Sunderland, Massachusetts*.
- [4] M.Abeles (1991) Corticonics. *Cambridge University Press, Cambridge*.
- [5] Evarts EV (1968) A technique for recording activity of subcortical neurons in moving animals. *Electroencephalography Clin Neurophysiol.*
- [6] Georgopoulos AP, Taira M, & Lukashin (1993) Cognitive neurophysiology of the motor cortex. *Science*, 260.
- [7] Alexander G Dimitrov (2002) Spike sorting the other way. *Neurocomputing 2003*.
- [8] Carroll Michael (2002) Spike Sorting for Neurosurgical Localization. *Learning From Data, Winter 2002*.
- [9] Kyung Hwan Kim & Sung June Kim (2000) Neural Spike Sorting Under Nearly 0-dB Signal-to-Noise Ratio Using Nonlinear Energy Operator and Artificial Neural-Network Classifier. *IEEE Transactions on Biomedical Engineering*, 47.
- [10] Susumu Takahashi, Yuichiro Anzai, & Yoshio Sakurai (2002) Automatic Sorting for Multi-Neuronal Activity Recorded With Tetrodes in the Presence of Overlapping Spikes. *J Neurophysiol*, 89.
- [11] Abeles M & Goldstein MH (1977) Multispikes train analysis. *Proc IEEE*, 65, 762-733.
- [12] Kyung Hwan Kim & Sung June Kim (2003) Method for Unsupervised Classification of Multiunit Neural Signal Recording Under Low Signal-to-Noise Ratio. *IEEE Transactions on Biomedical Engineering*, 50.
- [13] McNaughton B.L., J.O'Keefe, & C.A.Barnes (1983) The stereotrode - a new technique for simultaneous isolation of several single units in central nervous-system from multiple unit records. *Journal of Neuroscience Methods*, 8, 391-397.
- [14] Recce M.L. & J.O'Keefe (1989) The tetrode: An improved technique for multi-unit extracellular recording. *Society for Neuroscience Abstracts*, 15, 1250.

- [15] KENNETH D.HARRIS, DARRELL A.HENZE, JOZSEF CSICSVARI, HAJIME HIRASE, & GYORGY BUZSAKI (2000) Accuracy of Tetrode Spike Separation as Determined by Simultaneous Intracellular and Extracellular Measurements. *J Neurophysiol*, 84.
- [16] Lewicki Micheal S. (1998) A review of methods for spike sorting: the detection and classification of neural action potentials. *Network: Computation in Neural Systems*.
- [17] M.H.Yang & N.Ahuja (1999) Gaussian Mixture Model for Human Skin Color and Its Application in Image and Video Databases. *Proc. of the SPIE*, 3635.
- [18] Douglas A.Reynolds (1995) Speaker identification and verification using Gaussian mixture speaker models. *Speech Communication*, 17.
- [19] Salganicoff M & M.Sarna (1988) Unsupervised waveform classification for multi-neuron recordings: a real-time software-based system. *J Neurosci Methods*, 25, 181-187.
- [20] Zouridakis,G. & D.C.Tam (2000) Identification of reliable spike templates in multi-unit extracellular recording using fuzzy clustering. *Comput Methods Programs Biomed*, 61, 91-98.
- [21] Ohberg F. & H.Johansson (1996) A neural network approach to real-time spike discrimination during simultaneous recording from several multi-unit nerve filaments. *J Neurosci Methods*, 64, 181-187.
- [22] Susumu Takahashi, Yuichiro Anzai, & Yoshio Sakurai (2003) Automatic Sorting for Multi-Neuronal Activity Recorded With Tetrodes in the Presence of Overlapping Spikes. *J Neurophysiol*, 89, 2245-2258.
- [23] Robin Hanson, John Stutz, & Peter Cheeseman (1996) Bayesian Classification Theory. *Technical Report FIA-90-12-7-01*.
- [24] Sahani,M. (1999) Latent Variable Models for Neural Data Analysis. *Ph. D Thesis*.
- [25] Richard A.Redner & Homer F.Walker (1984) Mixture Densities, Maximum Likelihood and EM Algorithm. *SIAM Review*, 26, 195-239.
- [26] Lei Xu & Michael I.Jordan (1995) On Convergence Properties of the EM Algorithm for Gaussian Mixtures. *C. B. C. L. Paper No. 111*.
- [27] Alexander G.Dimitrov, John P.Miller, Tomas Gedeon, Zane Aldworth, & Albert E.Parker (2000) Analysis of neural coding using quantization with an information-based distortion measure. *Network: Computation in Neural Systems*.

- [28] Paul M. Baggenstoss (2002) *Statistical Modeling Using Gaussian Mixtures and HMMs with MATLAB*.
- [29] Lewicki Micheal S. (1995) Bayesian Modeling and Classification of Neural Signals. *Neural Computation*.
- [30] Dominique Lavenier (2000) FPGA implementation of the K-means clustering algorithm for hyperspectral images. *Los Alamos National Laboratory LAUR 00-3079*.
- [31] (2002) *Virtex-II Platform FPGAs: Introduction and Overview*.
- [32] A. Ghosh, J. Kunkel, & S. Liao (1999) Hardware Synthesis from C/C++. *Proc. Design, Automation and Test in Europe Conference and Exhibition*.
- [33] R. Helaihel & K. Olukotun (1997) Java as a Specification Language for Hardware-Software Systems. *Proc. International Conference on Computer-Aided Design*, 690-697.
- [34] (2003) *AccelFPGA User's Manual V1.7*.

APPENDICES

APPENDIX A

SIMPLIFYING THE EXPRESSION OF  $Q(O, O^s)$



The procedure of how to estimate the parameters of the Gaussian Mixture Model using Expectation Maximization algorithm is introduced in Chapter 2. In this section, we will present the mathematic details about how to simplify the expression of the expected value of the log-likelihood based on the whole data set and current model parameters. We define it as  $Q(O, O^g)$ .

To find the equations to iteratively estimate parameters of a GMM, one critical step is to simplify the expression of  $Q(O, O^g)$ . From Eq. 2-9 we know that:

$$Q(O | O^g) = E[\log p(X, Y | O) | X, O^g]$$

Combine the above equation with Eq. 2-12, Eq. 2-13, Eq. 2-14 and Eq. 2-15, we can get the following equation:

$$\begin{aligned} Q(O, O^g) &= \sum_{Y \in Y} \log(\zeta(O | X, Y)) p(Y | X, O^g) \\ &= \sum_{Y \in Y} \sum_{i=1}^N \log(\alpha_{y_i} p_{y_i}(x_i | o_{y_i})) \prod_{j=1}^N p(y_j | x_j, O^g) \\ &= \sum_{y_1=1}^M \sum_{y_2=1}^M \dots \sum_{y_N=1}^M \sum_{i=1}^N \log(\alpha_{y_i} p_{y_i}(x_i | o_{y_i})) \prod_{j=1}^N p(y_j | x_j, O^g) \\ &= \sum_{y_1=1}^M \sum_{y_2=1}^M \dots \sum_{y_N=1}^M \sum_{i=1}^N \sum_{l=1}^M \delta_{l, y_i} \log(\alpha_l p_l(x_i | o_l)) \prod_{j=1}^N p(y_j | x_j, O^g) \\ &= \sum_{l=1}^M \sum_{i=1}^N \log(\alpha_l p_l(x_i | o_l)) \sum_{y_1=1}^M \sum_{y_2=1}^M \dots \sum_{y_N=1}^M \delta_{l, y_i} \prod_{j=1}^N p(y_j | x_j, O^g) \end{aligned}$$

The form of  $Q(O, O^g)$  can still be greatly simplified. We note that for  $l \in 1, \dots, M$

$$\begin{aligned} &\sum_{y_1=1}^M \sum_{y_2=1}^M \dots \sum_{y_N=1}^M \delta_{l, y_i} \prod_{j=1}^N p(y_j | x_j, O^g) \\ &= (\sum_{y_1=1}^M \dots \sum_{y_{i-1}=1}^M \sum_{y_{i+1}=1}^M \dots \sum_{y_N=1}^M \prod_{j=1, j \neq i}^N p(y_j | x_j, O^g)) p(l | x_i, O^g) \\ &= \prod_{j=1, j \neq i}^N (\sum_{y_j=1}^M p(y_j | x_j, O^g)) p(l | x_i, O^g) \\ &= p(l | x_i, O^g) \end{aligned}$$

since  $\sum_{y_j=1}^M p(y_j | x_j, O^g) = 1$ . Now we can write  $Q(O, O^g)$  as:

$$\begin{aligned}
 Q(O, O^g) &= \sum_{l=1}^M \sum_{i=1}^N \log(\alpha_l p_l(x_i | o_l)) p(l | x_i, O^g) \\
 &= \sum_{l=1}^M \sum_{i=1}^N \log(\alpha_l) p(l | x_i, O^g) + \sum_{l=1}^M \sum_{i=1}^N \log(p_l(x_i | o_l)) p(l | x_i, O^g)
 \end{aligned}$$

This is the same equation as Eq. 2-16. As shown in Chapter 2, from this equation we can use Lagrange multiplier and other methods to get the Eq. (2-20), (2-21) and (2-22), which together form the basis of the EM algorithm for iteratively estimating the parameters of a Gaussian Mixture Model.

APPENDIX B

FLOATING POINT MATLAB CODE FOR THE EM ALGORITHM

*diaghalfLogEmGmm.m:*

```
function [StoreLike ,LoopCount ,OldMgmm]= diaghalfLogEmGmm (ncluster, data,
OldMgmm);
```

```
    min_std = 0.0001;
    [ndatadim ndata] = size(data);
```

```
    % Initialize the covariance to be diagonal
    OldMgmm.covars = ones(ncluster,ndatadim);
```

```
    PrevLike = -100;
    StopCri = 0.0001;
    MaxLoop = 100;
    LoopCount = 0;
    StopOrNot = 100;
    StoreLike = zeros(1,MaxLoop);
```

```
    range_max = 0;
    range_min = 1000;
    range_per = 1000;
```

```
    track = zeros(3,1000);
```

```
    % The main EM Loop
    while (StopOrNot > StopCri & LoopCount < MaxLoop)
```

```
        % Get old a, u and E
        OldCov = OldMgmm.covars; % Old E's
        OldMean = OldMgmm.centers; % Old u's
        OldPrior = OldMgmm.priors; % Old a's
```

```
        % -----
        % ==Calculate the Probabilty of of each data point Xi in each cluster m = 1:L
        % ProXiL matrix stores the probablity of each data point Xi in each cluster m
        % Each column is a point of data and each row is a cluster
        % ProXiL is L * N matrix
        LogProXiL = zeros(ncluster,ndata);
        log_normal = log(2*pi)*(ndatadim/2);
        for m = 1: ncluster
            covTemp = OldCov(m,:).^-1;
            detTemp = 0.5 * sum(log(OldCov(m,:)));
            for i = 1: ndata
                poten = -1/2*(data(:,i)-OldMean(:,m)).*(data(:,i)-OldMean(:,m)).*covTemp';
                poten = sum(poten);
                LogProXiL(m,i) = poten - detTemp - log_normal;
```

```

end
end

% -----
% ==Calculate likelihood of each data point
% The likelihood of each data point, store in a row vector
% LikeHoodXi is 1 * n matrix
% !!! Likelihood can not be zero, or the priors will blow up
% Use LogLikelihood to avoid zeros

LogLikeHoodXi = Logx(LogProXiL,OldPrior);
% ==End of calculation of likelihood of each data point

% -----
% ==Calculate the probability of l given Xi ProLXi
% ProLXi is L * N matrix
LogProLXi = LogProXiL + log(repmat((OldPrior)',1,ndata)) -
LogLikeHoodXi(ones(ncluster,1),:);

LoopCount = LoopCount + 1;
% Log version of likelihood of the whole data set, to judge the convergence
LogLikeHoodX = sum(LogLikeHoodXi);

LogLikeHoodX = LogLikeHoodX/ndata;
StoreLike(LoopCount) = LogLikeHoodX;
StopOrNot = abs(PrevLike - LogLikeHoodX);
PrevLike = LogLikeHoodX;

if StopOrNot < StopCri
    break;
end
ProLXi = exp(LogProLXi);

NewPrior2 = ones(1,ndata)*ProLXi'; %% NO NEED TO DIVIDE NDATA
NewPrior = NewPrior2/ndata;
NewMean = data * ProLXi' ./ repmat(NewPrior2,ndatadim,1);

% ==Calculate the covariance
for i = 1:ncluster
    Smeandata = data - repmat(NewMean(:,i),1,ndata);
    Smeandata = Smeandata * Smeandata';
    OldMgmm.covars(i,:) = ProLXi(i,:) * Smeandata' / NewPrior2(i);
    % Prevent Covariance matrix go to singular
    if OldMgmm.covars(i,:) < eps

```

```
fprintf('**&*&\n');
OldMgmm.covars(i,:) = ones(1,ndatadim);
end
end
% ==End of covarance calculation

% -----
% ==Update the parameters
OldMgmm.centers = NewMean;
OldMgmm.priors = NewPrior;

end % for the EM whold loop
StoreLike = StoreLike(1:LoopCount);
track = track(:,1:LoopCount);

% Put the parameters in descent order according to the prior probablities
[OldMgmm.priors new_in] = sort(OldMgmm.priors);
OldMgmm.centers = OldMgmm.centers(:,new_in);
OldMgmm.covars = OldMgmm.covars(new_in,:);
```

APPENDIX C

'C' CODE FOR THE EM ALGORITHM ON THE PARALLEL DSP PLATFORM

For the Master DSP (*Master.c*):

```

/*-----
  INCLUDES
-----*/
#include <stdio.h>
#include <matrix.h>
#include <math.h>
#include <stdlib.h>
#include <vector.h>
#include <21160.h>
#include <def21160.h>
#include "speeddsp.h"
#include "diag_DMA_master_2.h"

#define DMA          1          // Use DMA channel for external memory data
transaction
#define GLOBAL      0x0750000  // Global Flag
#define DSP_JUMP    0x0100000  // Memory address jump between DSPs
#define LH_BLOCK_1  0x0250002  // Base address of Likelihood of 1st block
#define LC_FLAG_1   0x0250001  // Base address of local flag
#define MULTI_BASE  0x0700000
#define EXTERNAL_BASE 0x0800000

#define SLAVE_BASE_1  0x0200000
#define SLAVE_BASE_2  0x0300000
#define SLAVE_BASE_3  0x0400000

#define START        1          // Define start for global flag
#define STOP         0          // Define stop for global flag
#define nprocessors  4          // How many processors in the system

#define PR_BLOCK_1  0x0250004  // Base address for middle results of DSP 1
#define P_BLOCK_1   0x025064C  // Base address for transfer results to DSP 1
#define GLOBAL_P    0x075064C  // Global address for transfer results to all DSPs

#define PRIOR_LENGTH 3          // Length of priors array
#define CANDC_LENGTH 800       // Length of centers and covars array
#define P_BLOCK_LENGTH 1606    // Length of the Parameters Block

#define SEM_GLOBAL    ((int)MSGR0 + (int)MULTI_BASE)
#define SEM_LOCAL_1   ((int)MSGR1 + (int)SLAVE_BASE_1)
#define SEM_LOCAL_2   ((int)MSGR2 + (int)SLAVE_BASE_2)
#define SEM_LOCAL_3   ((int)MSGR3 + (int)SLAVE_BASE_3)
#define SEM_DMA_WRITE ((int)MSGR4 + (int)EXTERNAL_BASE)
#define SEM_DMA_READ  ((int)MSGR4 + (int)EXTERNAL_BASE)

/*-----
  GLOBAL DECLARATIONS
-----*/

```

```

-----*/
segment ("seg_global_flag") int global_flag = 0;
segment ("seg_local_flag") int local_flag = 0;
float priorsTemp[1][nclusters] = {0};
float candcTemp[nclusters][ndim] = {0};

segment ("seg_data") float data[ndata][ndim] = {
0//#include "rescal_720_data_directload.dat"
};
// Data to be transmitted to external memory

float blockdata[nblockdata][ndim] = {0};
segment ("seg_pmda") float blockdata_diff[nblockdata][ndim] = {0};
//float blockdata_diff[nblockdata][ndim] = {0};
float centers[nclusters][ndim] = {
0//#include "rescal_diag_centers_directload.dat"
};
float priors[1][nclusters] = {0};
float covars[nclusters][ndim] = {0};
float logproXiL_master[nclusters][nblockdata] = {0};
float logproLXi_master[nclusters][nblockdata] = {0};
float blocksums[ndim]= {0};
float loglhXi_master[1][nblockdata] = {0};

float logpriors_master[1][nclusters] = {0};
float logcenters_master[nclusters][ndim] = {0};
float logcovars_master[nclusters][ndim] = {0};

/*-----
Processor Identifications
-----*/

void main()
{

int i = 0, j = 0;
int i2,j2;
int loopCount = 0;
int loopCountAll = 0;
float testlog = 0;
float testlog2;

float cycle_count = 0;

int ntasks = 0;
int offset = 0;
int flagsum = 0;
int offsetTemp = 0;
int data_offset = 0;
int LoopCount = 0;

```

```

int *data_start = 0;
int *priors_start = 0;
int *candc_start = 0;
int *global;
int *blockdata_slave[3] = {0};

```

```

int start_count;
int stop_count;

```

```

float proTemp = 0;
float prepro = 0;
float logproXiLTemp;
float maxTemp;
float lhdata_master;

```

```

float lhdatanew;
float lhdataold;
float thres;

```

```

float logproLXiTemp[ndata] = {0};
float maxTempProLXi[nclusters] = {0};

```

```

offset = 0;
loopCount = 0;
prepro = 0;
testlog2 = logf(2.71);
testlog = nclusters;
lhdataold = -100;

```

```

global = (int *) GLOBAL;
priors_start = (int *) priorsTemp;
candc_start = (int *) candcTemp;

```

```

SetIOP(SEM_GLOBAL,0);

```

```

*global = STOP;

```

```

/*****

```

```

** initialize the covars and priors **
*****/

```

```

initGmm(priors,centers,covars);

```

```

//Start Timer
timer_set (0xFFFFFFFF, 0xFFFFFFFF);
start_count = timer_on (); /* enable timer */

```

```

for (i=0;i<100000;i++)
{;}

```

```

/*****
**      Get the start address of blockdata on slave DSPs  **
*****/
while
((GetIOP(SEM_LOCAL_1) | GetIOP(SEM_LOCAL_2) | GetIOP(SEM_LOCAL_3)) == 0)
{
    blockdata_slave[0] = (int *) (GetIOP(SEM_LOCAL_1));
    blockdata_slave[1] = (int *) (GetIOP(SEM_LOCAL_2));
    blockdata_slave[2] = (int *) (GetIOP(SEM_LOCAL_3));

    SetIOP(SEM_LOCAL_1,1);
    SetIOP(SEM_LOCAL_2,1);
    SetIOP(SEM_LOCAL_3,1);

/*****
**      Use DMA move data to internal and cal proXIL  **
*****/
    data_start = (int *) data ;
    offset = 0;

/*****
**      Read data from SRAM to internal memory  **
*****/
//-----Moving Data-----
if (DMA == 1)
{
    extmem_to_intmem
    (data_start,          // Source buffer.
     (int *)blockdata, // Destination buffer.
     Blocksize,         // Buffer size.
     10,                // DMA Channel
     TRUE );           // Wait for DMA to compl

    extmem_to_intmem
    ((int *) (data_start+Blocksize), // Source buffer
     (int *) (blockdata_slave[0]+SLAVE_BASE_1),
     Blocksize,                    // Buffer size.
     10,                            // DMA Channel
     TRUE );                        // Wait for DMA to compl

    extmem_to_intmem
    ((int *) (data_start+2*Blocksize), // Source buffer
     (int *) (blockdata_slave[1]+SLAVE_BASE_2),
     Blocksize,                    // Buffer size.
     10,                            // DMA Channel
     TRUE );                        // Wait for DMA to compl

    extmem_to_intmem
    ((int *) (data_start+3*Blocksize), // Source buffer

```

```

(int *)(blockdata_slave[2]+SLAVE_BASE_3),
Blocksize,          // Buffer size.
10,                // DMA Channel
TRUE );            // Wait for DMA to compl
}
else
{
    for (i2=0;i2<nblockdata;i2++)
    {
        for (j2=0;j2<ndim;j2++)
        {
            blockdata[i2][j2] = data[offset + i2][j2];
        }
    }
}
//-----Finish Moving data-----

while (loopCountAll < maxLoop)
{
//-----End and Wait and Start-----

SetIOP(SEM_GLOBAL,1);
while ((GetIOP(SEM_LOCAL_1) | GetIOP(SEM_LOCAL_2) | GetIOP(SEM_LOCAL_3))!=
0)
{;}
SetIOP(SEM_GLOBAL,0);
//-----

/*****
**   Clear all the log version matrix **
*****/
    for (i=0;i<nclusters;i++)
    {
        for (j=0;j<ndim;j++)
        {
            logcovars_master[i][j] = 0;    //Clear the logcovars matrix
            logcenters_master[i][j] = 0;   //Clear the logcenters matrix
        }
    }

/*****
**   Calculate the probability in each cluster **
*****/
    for (i=0;i<nblockdata;i++)
    {
        for (j=0;j<nclusters;j++)
        {

```

```

        logproXiL_master[j][i] = getproXi(centers[j],blockdata[i],covars[j]);
    }
}

/*****
**      Calculate the liklihood of each data in this model
**      *****/
for (i=0;i<nblockdata;i++)
{
    // Clear the store matrix
    loghXi_master[0][i] = 0;
    // Get the max number of each column
    for (j=0;j<nclusters;j++)
    {
        if (j==0)
        {
            maxTemp = logproXiL_master[0][i];
        }
        else
        {
            if (logproXiL_master[j][i] > maxTemp)
                maxTemp = logproXiL_master[j][i];
        }
    }
    // Normalize the pro to [0 1]
    // Also calculate the likelihood of each data
    for (j=0;j<nclusters;j++)
    {
        logproXiLTemp = logproXiL_master[j][i] - maxTemp;
        logproXiLTemp = expf(logproXiLTemp);
        logproXiLTemp = logproXiLTemp * priors [0][j];
        loghXi_master[0][i] = loghXi_master[0][i] + logproXiLTemp;
    }
    loghXi_master[0][i] = logf(loghXi_master[0][i]) + maxTemp;
}

lhdata_master = vsum_f_D(loghXi_master[0],nblockdata)/ndata;
// Likelihood of data in this block
lhdatanew = lhdata_master;

/*****
/*      Calculate the Likelihood of whole data set */
/*****
//-----End and Wait and Start-----

while ((GetIOP(SEM_LOCAL_1) & GetIOP(SEM_LOCAL_2)
        & GetIOP(SEM_LOCAL_3))!= 1)

```

```

        {;}

//-----
    for (i=0;i<nprocessors-1;i++)
        {
            lhdatanew = lhdatanew + *((float *) (LH_BLOCK_1 + i*DSP_JUMP));
        }

    thres = lhdatanew - lhdataold;
    thres = fabsf(thres);
    if (thres < setThres)
        break;
    lhdataold = lhdatanew;
    loopCountAll ++;

//-----End and Wait and Start-----
//-----End and Wait and Start-----

    SetIOP(SEM_GLOBAL,1);
    while ((GetIOP(SEM_LOCAL_1) | GetIOP(SEM_LOCAL_2) |
            GetIOP(SEM_LOCAL_3))!= 0)
        {;}
    SetIOP(SEM_GLOBAL,0);

//-----

/*****
**   Calculate probability of being each cluster   **
*****/
    for (i=0;i<nclusters;i++)
        {
            for (j=0;j<nblockdata;j++)
                {
                    logproLXi_master[i][j] = logproXiL_master[i][j] + logf(priors[0][i]) - loglhXi_master[0][j];
                    logproLXi_master[i][j] = expf(logproLXi_master[i][j]); // Use float instead of log **
                }
        }

/*****
**   Update the priors   **
*****/
    for (i=0;i<nclusters;i++)
        {
            logpriors_master[0][i] = vsum_sf_D(logproLXi_master[i],1,nblockdata);
            priors[0][i] = logpriors[0][i]/ndata;
        }

/*****

```

```

**      Updata the centers      **
*****/
for(i=0;i<nclusters;i++)
    {
        for(j=0;j<ndim;j++)
            blocksums[j] = 0;

        block_sum(blocksums,blockdata,logproLXi_master[i],offset);
        vecvadd(logcenters_master[i],blocksums,logcenters_master[i],ndim);
    }

/*****/
/*      Update Priors and centers      */
/*****/
//-----End and Wait and Start-----

        while ((GetIOP(SEM_LOCAL_1) & GetIOP(SEM_LOCAL_2)
                & GetIOP(SEM_LOCAL_3))!= 1)
            {;}

//-----
        for (LoopCount=0;LoopCount<nprocessors-1;LoopCount++)
            {
//-----Get mid results of priors and centers from slaves-----
//-----Moving Data-----
                if (DMA == 1)
                    {
                        extmem_to_intmem_s2m
                        ( (int *) (PR_BLOCK_1+LoopCount*DSP_JUMP), // Source buffer.
                          priors_start, // Destination buffer.
                          P_BLOCK_LENGTH-CANDC_LENGTH, // Buffer size.
                          10, // DMA Channel
                          TRUE ); // Wait for DMA to compl
                    }
                else
                    {
                        for (i2=0;i2<P_BLOCK_LENGTH-CANDC_LENGTH;i2++)
                            {
                                *(priors_start + i2) = *((int *)PR_BLOCK_1+(LoopCount*DSP_JUMP)+i2);
                            }
                    }
            }

//-----Finish Moving data-----

//-----Begin update the centers-----
for (i=0;i<nclusters;i++)
    {
        logpriors_master[0][i] = logpriors_master[0][i] + priorsTemp[0][i];
    }

```

```

vecvadd(logcenters_master[i],candcTemp[i],logcenters_master[i],ndim);
}

}

for (i=0;i<nclusters;i++)
{
priors[0][i] = logpriors_master[0][i]/ndata;
logpriors_master[0][i] = 1/logpriors_master[0][i];
vecsmult(logcenters_master[i],logpriors_master[0][i],centers[i],ndim);
}
//-----End update the centers-----
//-----Moving Data to slave dsps-----
if (DMA == 1)
{
intmem_to_extmem_m2s
((int *)priors, // Source buffer.
(int *)P_BLOCK_1, // Destination buffer.
P_BLOCK_LENGTH-CANDC_LENGTH, // Buffer size.
10, // DMA Channel
TRUE ); // Wait for DMA to compl

intmem_to_extmem_m2s
((int *)priors, // Source buffer.
(int *) (P_BLOCK_1+DSP_JUMP), // Destination buffer.
P_BLOCK_LENGTH-CANDC_LENGTH, // Buffer size.
10, // DMA Channel
TRUE ); // Wait for DMA to compl

intmem_to_extmem_m2s
((int *)priors, // Source buffer.
(int *) (P_BLOCK_1+2*DSP_JUMP), // Destination buffer.
P_BLOCK_LENGTH-CANDC_LENGTH, // Buffer size.
10, // DMA Channel
TRUE ); // Wait for DMA to compl
}
else
{
for (i2=0;i2<P_BLOCK_LENGTH-CANDC_LENGTH;i2++)
{
*((int *)GLOBAL_P + i2) = *((int *)priors + i2);
}
}
//-----Finish Moving data-----
//-----End and Wait and Start-----

SetIOP(SEM_GLOBAL,1);
while ((GetIOP(SEM_LOCAL_1) | GetIOP(SEM_LOCAL_2) | GetIOP(SEM_LOCAL_3))!=
0)

```

```

        {;}
SetIOP(SEM_GLOBAL,0);
//-----

/*****
**          data the covars          **
*****/

for(i=0;i<nclusters;i++)
    {
        for(j=0;j<nblockdata;j++)
            {
                vecvsub(blockdata[j],centers[i],blockdata_diff[j],ndim);
                vecvmlt(blockdata_diff[j],blockdata_diff[j],blockdata_diff[j],ndim);
            }

        for(j=0;j<ndim;j++)
            blocksums[j] = 0;

        block_sum(blocksums,blockdata_diff,logproLXi_master[i],offset);
        vecvadd(logcovars_master[i],blocksums,logcovars_master[i],ndim);

    }

/*****
**          Update the covars          */
*****/
//-----End and Wait and Start-----

while
(GetIOP(SEM_LOCAL_1) & GetIOP(SEM_LOCAL_2) & GetIOP(SEM_LOCAL_3))!= 1)
    {;}

//-----
        for (LoopCount=0;LoopCount<nprocessors-1;LoopCount++)
            {
//-----Get mid results of priors and centers from slaves-----
//-----Moving Data-----
if (DMA == 1)
    {
        extmem_to_intmem_s2m
        ((int          *) (PR_BLOCK_1+LoopCount*DSP_JUMP+P_BLOCK_LENGTH-
CANDC_LENGTH),
        candc_start,
        CANDC_LENGTH,          // Buffer size.
        10,                    // DMA Channel
        TRUE );                // Wait for DMA to compl
    }

```

```

else
{
for (i2=0;i2<CANDC_LENGTH;i2++)
{
*(candc_start + i)=$((int *) (PR_BLOCK_1+LoopCount*DSP_JUMP+P_BLOCK_LENGTH-
CANDC_LENGTH+i));
}
}
//-----Finish Moving data-----

//-----Begin update the covars-----
for (i=0;i<nclusters;i++)
{
vecvadd(logcovars_master[i],candcTemp[i],logcovars_master[i],ndim);
}
}
for (i=0;i<nclusters;i++)
{
vecsmult(logcovars_master[i],logpriors_master[0][i],covars[i],ndim);
}
//-----End update the covars-----
//-----Moving Data-----
if (DMA == 1)
{
intmem_to_extmem_m2s
(
(int *)covars, // Source buffer.
(int *) (P_BLOCK_1+P_BLOCK_LENGTH-CANDC_LENGTH),
CANDC_LENGTH, // Buffer size.
10, // DMA Channel
TRUE); // Wait for DMA to compl

intmem_to_extmem_m2s((int *)covars, // Source buffer.
(int *) (P_BLOCK_1+P_BLOCK_LENGTH-CANDC_LENGTH+DSP_JUMP),
CANDC_LENGTH, // Buffer size.
10, // DMA Channel
TRUE); // Wait for DMA to compl

intmem_to_extmem_m2s((int *)covars, // Source buffer.
(int *) (P_BLOCK_1+P_BLOCK_LENGTH-CANDC_LENGTH+2*DSP_JUMP),
CANDC_LENGTH, // Buffer size.
10, // DMA Channel
TRUE); // Wait for DMA to compl
}
else
{
for (i2=0;i2<CANDC_LENGTH;i2++)
{
*((int *) (GLOBAL_P+P_BLOCK_LENGTH-CANDC_LENGTH+i2))= *((int *)covars + i2);

```

```
    }  
  }  
  //-----Finish Moving data-----|  
  
}  
  
stop_count = timer_off();  
cycle_count = ((float)start_count - (float)stop_count)/80000000 ; //divide by 80MHz  
  
}
```

For the Slave DSPs (*Slave.c*):

```

/*-----
  INCLUDES
-----*/

#include <stdio.h>
#include <matrix.h>
#include <math.h>
#include <stdlib.h>
#include <vector.h>
#include <21160.h>
#include <def21160.h>
#include "speeddsp.h"
#include "diag_DMA_slave_2.h"

#define DMA 1 // Use DMA channel for external memory data transaction
#define processorID 3
#define START 1
#define STOP 0
#define MASTER_BASE 0x0100000
#define MULTI_BASE 0x0700000
#define EXTERNAL_BASE 0x0800000

#define SEM_GLOBAL MSGR0
#define SEM_LOCAL MSGR1
#define SEM_DMA_WRITE ((int)MSGR4 + (int)EXTERNAL_BASE)

/*-----
  GLOBAL DECLARATIONS
-----*/

segment ("seg_local_flag") int local_flag = 0;
segment ("seg_global_flag") int global_flag = 0;
segment ("seg_lh_block") float lhdata_block_1 = 0;

segment ("seg_prior_block") float logpriors_block_1[1][nclusters] = {0};
segment ("seg_centers_block") float logcenters_block_1[nclusters][ndim] = {0};
segment ("seg_covars_block") float logcovars_block_1[nclusters][ndim] = {0};

segment ("seg_p_block") float centers[nclusters][ndim] = {0};
segment ("seg_p_block") float covars[nclusters][ndim] = {0};
segment ("seg_p_block") float priors[1][nclusters] = {0};

segment ("seg_data") float data[ndata][ndim] = {
0//#include "rescal_720_data_directload.dat"
}; // Data to be transmitted to external memory

float blockdata[nblockdata][ndim] = {0};
segment ("seg_pmda") float blockdata_diff[nblockdata][ndim] = {0};

```

```
float logproXiL_block_1[nclusters][nblockdata] = {0};
float logproLXi_block_1[nclusters][nblockdata] = {0};
float blocksums[ndim]={0};
```

```
float cycle_count = 0;
```

```
float loglhXi_block_1[1][nblockdata] = {0};
```

```
/*-----
Processor Identifications
-----*/
```

```
void main()
```

```
{
```

```
int i = 0, j = 0;
int i2,j2;
int loopCount = 0;
int loopCountAll = 0;
float testlog = 0;
float testlog2;
int ntasks = 0;
int offset = 0;
int flagsum = 0;
int offsetTemp = 0;
int data_offset = 0;
int *data_start = 0;
```

```
float start_count;
float stop_count;
```

```
int proTemp = 0;
int prepro = 0;
```

```
float logproXiLTemp;
float maxTemp;
```

```
int *global;
```

```
float logproLXiTemp[ndata] = {0};
```

```
float maxTempProLXi[nclusters] = {0};
```

```
SetIOP(SEM_LOCAL,0);
offset = 0;
loopCount = 0;
prepro = 0;
testlog2 = logf(2.71);
```

```

testlog = nclusters;

/*****
**Get the address of blockdata and transfer to Master      **
*****/

SetIOP(SEM_LOCAL,(int)blockdata);

/*****
** initialize the covars and priors                        **
*****/
initGmm(priors,centers,covars);
local_flag = 1;

//Start Timer
timer_set (0xFFFFFFFF, 0xFFFFFFFF);
start_count = timer_on (); /* enable timer */

while (START)
{

//-----End and Wait and Start-----

        while (GetIOP(SEM_GLOBAL) == 0)
                {;}

        SetIOP(SEM_LOCAL,0);

//-----
/*****
** Clear all the log version matrix                      **
*****/
for (i=0;i<nclusters;i++)
{
        for (j=0;j<ndim;j++)
        {
                logcovars_block_1[i][j] = 0; //Clear the logcovars matrix
                logcenters_block_1[i][j] = 0; //Clear the logcenters matrix
        }
}

/*****
** Calculate the probablity in each cluster              **
*****/
for (i=0;i<nblockdata;i++)
{
        for (j=0;j<nclusters;j++)
        {
                logproXiL_block_1[j][i] = getproXi(centers[j],blockdata[i],covars[j]);

```

```

}
}

/*****
**      Calculate the likelihood of each data in this model      **
*****/
for (i=0;i<nblockdata;i++)
{
    // Clear the store matrix
    loghXi_block_1[0][i] = 0;
    // Get the max number of each column
    for (j=0;j<nclusters;j++)
    {
        if (j==0)
        {
            maxTemp = logproXiL_block_1[0][i];
        }
        else
        {
            if (logproXiL_block_1[j][i] > maxTemp)
                maxTemp = logproXiL_block_1[j][i];
        }
    }
    // Normalize the pro to [0 1]
    // Also calculate the likelihood of each data
    for (j=0;j<nclusters;j++)
    {
        logproXiLTemp = logproXiL_block_1[j][i] - maxTemp;
        logproXiLTemp = expf(logproXiLTemp);
        logproXiLTemp = logproXiLTemp * priors [0][j];
        loghXi_block_1[0][i] = loghXi_block_1[0][i] + logproXiLTemp;
    }
    loghXi_block_1[0][i] = logf(loghXi_block_1[0][i]) + maxTemp;
}

lhdata_block_1 = vsum_f_D(loghXi_block_1[0],nblockdata)/ndata;
// Likelihood of data in this block

/*****
**      Transfer likelihood of this block      **
**      If necessary continue the calculation  **
*****/
//-----End and Wait and Start-----
SetIOP(SEM_LOCAL,1);

while (GetIOP(SEM_GLOBAL) == STOP)
{;}

```

```

SetIOP(SEM_LOCAL,0);
//-----

/*****
** Calculate probability of being each cluster **
*****/
for (i=0;i<nclusters;i++)
{
for (j=0;j<nblockdata;j++)
{
logproLXi_block_1[i][j]
= logproXiL_block_1[i][j] + logf(priors[0][i]) - loghXi_block_1[0][j];
logproLXi_block_1[i][j] = expf(logproLXi_block_1[i][j]); // Use float instead of log **
}
}

/*****
** Update the priors **
*****/
for (i=0;i<nclusters;i++)
{
logpriors_block_1[0][i] = vsum_sf_D(logproLXi_block_1[i],1,nblockdata);
logpriors_block_1[0][i] = logpriors[0][i]/ndata;
}

/*****
** Update the centers **
*****/
for(i=0;i<nclusters;i++)
{
for(j=0;j<ndim;j++)
blocksums[j] = 0;

block_sum(blocksums,blockdata,logproLXi_block_1[i],offset);
vecvadd(logcenters_block_1[i],blocksums,logcenters_block_1[i],ndim);
}

/*****
*/ Transfer Priors and centers */
*/ Get New Priors and centers back */
*****/
//-----End and Wait and Start-----
SetIOP(SEM_LOCAL,1);

while (GetIOP(SEM_GLOBAL) == STOP)
{;}

SetIOP(SEM_LOCAL,0);

```

```

//-----
/*****
**          Update the covars          **
*****/

for(i=0;i<nclusters;i++)
{
for(j=0;j<nblockdata;j++)
{
vecvsub(blockdata[j],centers[i],blockdata_diff[j],ndim);
vecvmt(blockdata_diff[j],blockdata_diff[j],blockdata_diff[j],ndim);
}

for(j=0;j<ndim;j++)
blocksums[j] = 0;

block_sum(blocksums,blockdata_diff,logproLXi_block_1[i],offset);
vecvadd(logcovars_block_1[i],blocksums,logcovars_block_1[i],ndim);

}

SetIOP(SEM_LOCAL,1);

stop_count = timer_off();
cycle_count = ((float)start_count - (float)stop_count)/80000000 ; //divide by 80MHz

}

}

```

The library included in both C codes for master and slave processors (*lib.c*):

```

#include <stdio.h>
#include <matrix.h>
#include <math.h>
#include <stdlib.h>
#include <vector.h>

#include <def21160.h>
#include <21060.h>
#include <signal.h>

#include "speeddsp.h"
#include "diag_DMA_master_2.h"

// Global variables.

int dmac_ext_to_int = DMAC_MDI;
int dmac_int_to_ext = DMAC_MDO;

void initGmm (float priors[][nclusters], float centers[][ndim], float covars[][ndim])
{
    /*-----
    -- initialize the parameters
    -----*/
    int i = 0;
    int j = 0;
    float a = 0;

    /*-----
    -- Build a matrix for diagonal covars initialization
    -----*/
    for (i=0;i<nclusters;i++)
    {
        for (j=0;j<ndim;j++)
            covars[i][j] = 1;
    }

    /*-----
    Make sum(priors) = 1
    -----*/
    for (i=0;i<nclusters;i++)
        {priors[0][i] = 0;}

    a = nclusters;
    a = 1/a;
    vecsadd(priors[0],a,priors[0],nclusters);

    /* Just for testing */
    /* ----- */
}

```

```

float getproXi(float centers[ndim],float blockdata[ndim],float covars[ndim])
{
    int i = 0,j = 0 ;
    float *meanTemp;
    float *invcovTemp = {0};
    float *detcovTemp;
    float proXiTemp = {0};
    float divider = 0;
    float log_normal = 0;
    float proXi;

    // normalization constant
    log_normal = logf(2*3.1416)*(ndim/2);
    invcovTemp = (float *) calloc (ndim, sizeof (float));
    meanTemp = (float *) calloc (ndim, sizeof (float));
    detcovTemp = (float *) calloc (ndim, sizeof (float));

    for (j=0;j<ndim;j++)
    {
        invcovTemp[j] = 1/covars[j];
    }

    // log version of the diagonal covar
    vlog_f_DD(detcovTemp,covars,ndim);
    // det = prod(cov); logdet = sum(logcov)
    divider = vsum_sf_D(detcovTemp,1,ndim);
    divider = divider * 0.5;

    vecvsub(blockdata,centers,meanTemp,ndim);
    vecvmlt(meanTemp,meanTemp,meanTemp,ndim);
    vecvmlt(meanTemp,invcovTemp,meanTemp,ndim);
    proXiTemp = vsum_sf_D(meanTemp,1,ndim);
    proXiTemp = -0.5*proXiTemp;
    proXi = proXiTemp - divider - log_normal;

    free(invcovTemp);
    free(meanTemp);
    free(detcovTemp);

    return (proXi);
}

void block_sum(float blocksums[],float blockdata[][ndim],float logproLXi[], int offset)
{
    int i,j=0;

    for (i=0;i<ndim;i++)

```

```

    {
        for (j=0;j<nblockdata;j++)
        {
            blocksums[i] = blocksums[i] + blockdata[j][i] * logproLXi[j];
        }
    }
}

int extmem_to_intmem( int *extsrc, int *intdest, int num_to_copy, int dma_channel, int wait )
{
    int chan_offset;
    int DSP_offset;
    int New_intdest;
    int dma_status_bit;
    int testwait;

    // if internal memory is in MASTER, no offset and intdest need not change
    DSP_offset = 0;
    New_intdest = (int) intdest;

    // Check for valid DMA channel.
    if( (dma_channel < 10) || (dma_channel > 13) )
        return( FALSE );
    chan_offset = (dma_channel - 10) << 3;

    // if internal memory is in SLAVE, offset equals Upper portion of indest and indest
    // should be the lower portion
    if ((int)intdest > (int)0x0200000)
    {
        DSP_offset = (int) intdest & 0x0F00000;
        New_intdest = (int) intdest & 0x00FFFFFF;
    }

    // Prepare DMA from external memory to EPB0 (via DMAC10).
    SetIOP( (DSP_offset+DMAC10+(dma_channel-10)), DMAC_FLUSH ); // Flush
    SetIOP( (DSP_offset+II10+chan_offset ), (int)New_intdest );
    SetIOP( (DSP_offset+IM10+chan_offset ), 1 );
    SetIOP( (DSP_offset+C10+chan_offset ), num_to_copy );
    SetIOP( (DSP_offset+EI10+chan_offset ), (int)extsrc );
    SetIOP( (DSP_offset+EM10+chan_offset ), 1 );
    SetIOP( (DSP_offset+EC10+chan_offset ), num_to_copy );

```

```

SetIOP( (DSP_offset+DMAC10+(dma_channel-10)), DMAC_MDI ); // Enable,
Read from extmem.

```

```

// If we need to wait for completion.

```

```

if( wait )

```

```

{

```

```

    dma_status_bit = 1L << dma_channel;

```

```

    // Wait until dma completed.

```

```

    while( GetIOP((DSP_offset+DMASTAT)) & dma_status_bit )

```

```

    {;}

```

```

// Disable DMA.

```

```

SetIOP( (DSP_offset+DMAC10+(dma_channel-10)), DMAC_FLUSH );

```

```

}

```

```

return( TRUE );

```

```

}

```

```

int intmem_to_extmem_m2s( int *intsrc, int *extdest, int num_to_copy, int dma_channel, int
wait )

```

```

{

```

```

    int chan_offset;

```

```

    int DSP_offset;

```

```

    int New_intdest;

```

```

// Check for valid DMA channel.

```

```

if( (dma_channel < 10) || (dma_channel > 13) )

```

```

    return( FALSE );

```

```

chan_offset = (dma_channel - 10) << 3;

```

```

// Prepare DMA for slave DSP

```

```

if ((int)extdest > (int)0x0200000)

```

```

{

```

```

    DSP_offset = (int) extdest & 0x0F00000;

```

```

    New_intdest = (int) extdest & 0x00FFFFFF;

```

```

    SetIOP( (DSP_offset+DMAC10+(dma_channel-10)), DMAC_FLUSH ); //

```

```

Flush

```

```

    SetIOP( (DSP_offset+II10+chan_offset ), (int)New_intdest );

```

```

    SetIOP( (DSP_offset+IM10+chan_offset ), 1 );

```

```

    SetIOP( (DSP_offset+C10+chan_offset ), num_to_copy );

```

```

    SetIOP( (DSP_offset+DMAC10+(dma_channel-10)), DMAC_SLA_MDI ); //

```

```

Enable, Write to extmem.

```

```

}

```

```

// Prepare DMA from external memory to EPB0 (via DMAC10).

```

```

SetIOP( (DMAC10+(dma_channel-10)), DMAC_FLUSH ); // Flush

```

```

SetIOP( (II10+chan_offset ), (int)intsrc );

```

```

SetIOP( (IM10+chan_offset ), 1 );

```

```

SetIOP( (C10+chan_offset ), num_to_copy );
SetIOP( (EI10+chan_offset ), (int)(DSP_offset+EPB0+chan_offset));
SetIOP( (EM10+chan_offset ), 0 );
SetIOP( (EC10+chan_offset ), num_to_copy );
SetIOP( (DMAC10+(dma_channel-10)), DMAC_MST_MDO );
// If we need to wait for completion.
if( wait )
{
    int dma_status_bit = 1L << dma_channel;

    // Wait until dma completed.
    while( GetIOP( DMASTAT ) & dma_status_bit );
    while( GetIOP( DSP_offset+DMASTAT ) & dma_status_bit );

    // Disable DMA.
    SetIOP( (DMAC10+(dma_channel-10)), DMAC_FLUSH );
    SetIOP( (DSP_offset+DMAC10+(dma_channel-10)), DMAC_FLUSH );
}

return( 0 );
}

int extmem_to_intmem_s2m ( int *extsrc, int *intdest, int num_to_copy, int dma_channel, int
wait )
{
    int chan_offset;
    int New_extsrc;
    int DSP_offset;

    // Check for valid DMA channel.
    if( (dma_channel < 10) || (dma_channel > 13) )
        return( FALSE );
    chan_offset = (dma_channel - 10) << 3;

    // Prepare DMA for slave DSP
    if ((int)extsrc > (int)0x0200000)
    {
        DSP_offset = (int) extsrc & 0x0F00000;
        New_extsrc = (int) extsrc & 0x00FFFFFF;

        SetIOP( (DSP_offset+DMAC10+(dma_channel-10)), DMAC_FLUSH ); //
Flush
        SetIOP( (DSP_offset+II10+chan_offset ), (int)New_extsrc );
        SetIOP( (DSP_offset+IM10+chan_offset ), 1 );
        SetIOP( (DSP_offset+C10+chan_offset ), num_to_copy );
        SetIOP( (DSP_offset+DMAC10+(dma_channel-10)), DMAC_SLA_MDO ); //
Enable, Write to extmem.
    }

    // Prepare DMA from external memory to EPB0 (via DMAC10).

```

```

SetIOP( (DMAC10+(dma_channel-10)), DMAC_FLUSH ); // Flush
SetIOP( (II10+chan_offset ), (int)intdest );
SetIOP( (IM10+chan_offset ), 1 );
SetIOP( (C10+chan_offset ), num_to_copy );
SetIOP( (EI10+chan_offset ), (int)(DSP_offset+EPB0+chan_offset) );
SetIOP( (EM10+chan_offset ), 0 );
SetIOP( (EC10+chan_offset ), num_to_copy );
SetIOP( (DMAC10+(dma_channel-10)), DMAC_MST_MDI ); // Enable, Read
from extmem.

```

```

// If we need to wait for completion.
if( wait )
{
    int dma_status_bit = 1L << dma_channel;

    // Wait until dma completed.
    while( GetIOP( DMASTAT ) & dma_status_bit );
//      while( GetIOP( DSP_offset+DMASTAT ) & dma_status_bit );

    // Disable DMA.
    SetIOP( (DMAC10+(dma_channel-10)), DMAC_FLUSH );
    SetIOP( (DSP_offset+DMAC10+(dma_channel-10)), DMAC_FLUSH );
}

return( TRUE );
}

```

```

int SetSem(int sem)
{
    while( set_semaphore((int *)sem,1,1) != 0)
        {;}

    return (0);
}

```

```

int ClearSem(int sem)
{
    while( set_semaphore((int *)sem,0,1) != 0)
        {;}

    return (0);
}

```

```

//*****
// End of lib.c
//*****

```

APPENDIX D

FIXED-POINT MATLAB CODE USING ACCELFPGA

*AccelChip.m*

```

warning off;
% Directive to indicate the target for synthesis
%!ACCEL TARGET XC2V3000

%%package_quantizer_constant;
%%----->>>>All the Quantizers<<<<-----%%
% q_data:    [18 6] -- for data input
% q1:       [30 13] -- for all the accumulator and middle results
% q_lut_cov_inv:[18 3] -- for cov_inv look up table
% q_lut_cov_log:[18 13] -- for cov_log look up table
% q_lut_exp:  [18 13] -- for first exp look up table
% q_lut_log:  [18 13] -- for log look up table
% q_lut_prior:[18 13] -- for prior inv look up table
% q_int:     [18 0] -- for integer
% q_fra:     [18 17] -- for fractional number

% Define the width of the word to 30 and binary point is 13 -
% Just simplify the whole process using one fixed point length -
% May be able to devise algorithm to different precision.
wl = 30;
fp = 13;

% This word length is for the input data. 18 bit can be fit in the block
% memory.
wl_data = 18;
fp_data = 6;

% quantize for integer
wl_int = 18;
fp_int = 0;

% quantize for pure fractional number
wl_fra = 18;
fp_fra = 17;

% This word length for lut, binary point can be varied due to the
% precision`
% requirement.
% Most fixed points are the same, but in case I will change one later,
% just0
% leave it as different fp.
wl_lut = 18;
fp_lut_cov_inv = 3;
fp_lut_cov_log = 13;
fp_lut_exp = 13;
fp_lut_log = 13;
fp_lut_prior = 13;
fp_lut_inv = 13;

```

```

q_data = quantizer('floor',[wl_data, fp_data]);
q1 = quantizer('floor','wrap',[wl fp]);
q1_int = quantizer('floor','wrap',[wl 0]);
q_lut_cov_inv = quantizer('floor','wrap',[wl_lut fp_lut_cov_inv]);
q_lut_cov_log = quantizer('floor','wrap',[wl_lut fp_lut_cov_log]);
q_lut_exp = quantizer('floor','wrap',[wl_lut fp_lut_exp]);
q_lut_log = quantizer('floor','wrap',[wl_lut fp_lut_log]);
q_lut_prior = quantizer('floor','wrap',[wl_lut fp_lut_prior]);
q_int = quantizer('floor','wrap',[wl_int fp_int]);
q_fra = quantizer('floor','wrap',[wl_fra fp_fra]);
q_lut_inv = quantizer('floor','wrap',[wl_lut fp_lut_inv]);

q_mid_centers = quantizer('floor','wrap',[18 8]);
q_mid_priors = quantizer('floor','wrap',[18 8]);
q_small = quantizer('floor','wrap',[18 13]);
q_square_cov = quantizer('floor','wrap',[18 15]);
q_covars = q_square_cov;
q_priors = q_covars;
q_mid_covars = quantizer('floor','wrap',[18 12]);
q_loglike = quantizer('floor','wrap',[18 7]);

% fixed point for address line 10 bits
% Change to 30 bits to meet the compiler's requirement
q_addr = quantizer('ufixed','floor','wrap',[30 0]);

% These lookup table content should be written in txt file and load into

lut_exp_content = quantize(q_lut_exp,load('lut_exp_content.dat')); % 0 - 4 step: 2^-8
lut_log_content = quantize(q_lut_log,load('lut_log_content.dat')); % 0 - 0.5 step: 2^-11
lut_inv_content = quantize(q_lut_inv,load('lut_inv_content.dat')); % 0 - 512 step: 0.5
lut_covars_inv_content = quantize(q_lut_cov_inv,load('lut_covars_inv_content.dat')); % 0 -
0.25 step: 2^-12
lut_covars_log_content = quantize(q_lut_cov_log,load('lut_covars_log_content.dat')); % 0 -
0.25 step: 2^-12

lut_exp_ProLXi = quantize(q_lut_exp,load('lut_exp_ProLXi.dat'));
lut_exp_ProLXi = quantize(q_lut_exp,lut_exp_ProLXi);
lut_covars_inv_tile = quantize(q_lut_cov_inv,lut_covars_inv_tile);
lut_covars_log_tile = quantize(q_lut_cov_log,lut_covars_log_tile);

%----->>>>>Quantize the variables<<<<<<-----%
% Load the input data
indata = load('q_indata.dat');
indata = quantize(q_data,indata);
indata_1 = indata;
indata_2 = indata;
indata_3 = indata;
indata_4 = indata;
indata_5 = indata;

```

```

break_flag = 0;
% Load the initial value of the model
centers = quantize(q_data,zeros(160,5));
centers = quantize(q_data,load('centers.dat'));
centers_1 = quantize(q_data,load('centers_1.dat'));
centers_2 = quantize(q_data,load('centers_2.dat'));
centers_3 = quantize(q_data,load('centers_3.dat'));
centers_4 = quantize(q_data,load('centers_4.dat'));
centers_5 = quantize(q_data,load('centers_5.dat'));
centers(:,1) = centers_1;
centers(:,2) = centers_2;
centers(:,3) = centers_3;
centers(:,4) = centers_4;
centers(:,5) = centers_5;
% Define constant
log_const = quantize(q1,147.0302); % log_const = log(2*pi)*(ndim/2);
ndim = quantize(q_int,160);
ndata = quantize(q_int,5);
ncluster = quantize(q_int,5);
stop_thre = quantize(q_int,6.25);
MaxLoop = quantize(q_int,27);
LoopCount = quantize(q_int,1);
loop_flag = quantize(q_int,1);
id = quantize(q_int,1);

centers_test = zeros(1,5);
covars_test = zeros(1,5);
upper_const = quantize(q_fra,0.2498);

covars = quantize(q_covars,zeros(ndim,ncluster)+1);
priors = quantize(q_priors,(zeros(1,ncluster)+1) * 0.2);
ndata_inv = quantize(q_fra,0.0014);

const_zero = quantize(q_int,0);

% The middle result we need

indatabuf = zeros(1,ndim);
LogProXiL = quantize(q1,zeros(1,ncluster));
LogLikeHoodXi = quantize(q_loglike,0);
LogLikeHoodX = quantize(q1,0);

ProLXi = quantize(q_lut_exp,zeros(ndata,ncluster));

mid_priors = quantize(q_mid_priors,zeros(1,ncluster));
mid_centers = quantize(q_mid_centers,zeros(ndim,ncluster));
mid_covars = quantize(q_mid_covars,zeros(ndim,ncluster));
mid_priors_inv = quantize(q_lut_inv,zeros(1,ncluster));

```

```

newLikeHood = quantize(q1,0);

covars_inv = quantize(q_lut_cov_inv,zeros(ndim,ncluster)+1); % initialize the cov_inv
covars_log = quantize(q_lut_cov_log,zeros(ndim,ncluster));
covars_det = quantize(q1,zeros(1,ncluster));
priors_log = quantize(q_lut_log,(zeros(1,ncluster)+1) * -1.6094);

result_covars = quantize(q_covars,zeros(ndim,ncluster));
result_centers = quantize(q_small,zeros(ndim,ncluster));
result_priors = quantize(q_priors,zeros(1,ncluster));

BUFSIZE = 160; % one at a time

% Revise: Use frame input buffer
% 160 point at once. To see how it works.

NUMSAMPS = 160 * 720; % 18860 input data point

%-----
%%
%% Main EM LOOP
%%
while 1
% data steam in
%!ACCEL STREAM n
for n = 1:ndim:NUMSAMPS-ndim+1

% section targeted for hardware starts here;
%!ACCEL BEGIN_HARDWARE indata_1, indata_2, indata_3, indata_4, indata_5
%!ACCEL SHAPE indatabuf_1(ndim)
%!ACCEL SHAPE indatabuf_2(ndim)
%!ACCEL SHAPE indatabuf_3(ndim)
%!ACCEL SHAPE indatabuf_4(ndim)
%!ACCEL SHAPE indatabuf_5(ndim)
indatabuf_1 = quantize(q_data, indata_1(n:n+ndim-1));
indatabuf_2 = quantize(q_data, indata_2(n:n+ndim-1));
indatabuf_3 = quantize(q_data, indata_3(n:n+ndim-1));
indatabuf_4 = quantize(q_data, indata_4(n:n+ndim-1));
indatabuf_5 = quantize(q_data, indata_5(n:n+ndim-1));

%!ACCEL SHAPE indatabuf(ndim,ncluster)

for d_0 = quantize(q_int,1:ndim);
indatabuf(d_0,1) = indatabuf_1(d_0);
indatabuf(d_0,2) = indatabuf_2(d_0);
indatabuf(d_0,3) = indatabuf_3(d_0);

```

```

indatabuf(d_0,4) = indatabuf_4(d_0);
indatabuf(d_0,5) = indatabuf_5(d_0);
end

```

```

%!ACCEL SYNTHESIS OFF

```

```

indatabuf(:,1) = indatabuf_1';
indatabuf(:,2) = indatabuf_2';
indatabuf(:,3) = indatabuf_3';
indatabuf(:,4) = indatabuf_4';
indatabuf(:,5) = indatabuf_5';

```

```

%!ACCEL SYNTHESIS ON

```

```

%%-----Memory map for look-up table

```

```

%!ACCEL MEM_MAP lut_exp_content to ram_s18_s18(56) at 0

```

```

%!ACCEL MEM_MAP lut_log_content to ram_s18_s18(58) at 0

```

```

%!ACCEL MEM_MAP lut_inv_content to ram_s18_s18(57) at 0

```

```

%!ACCEL MEM_MAP indatabuf_1 to ram_s18_s18(1) at 0

```

```

%!ACCEL MEM_MAP indatabuf_2 to ram_s18_s18(2) at 0

```

```

%!ACCEL MEM_MAP indatabuf_3 to ram_s18_s18(3) at 0

```

```

%!ACCEL MEM_MAP indatabuf_4 to ram_s18_s18(4) at 0

```

```

%!ACCEL MEM_MAP indatabuf_5 to ram_s18_s18(5) at 0

```

```

%%-----This part tricks accelchip to initialize the blockmem

```

```

%!ACCEL MEM_MAP centers_1 to ram_s18_s18(16) at 0

```

```

%!ACCEL MEM_MAP centers_2 to ram_s18_s18(17) at 0

```

```

%!ACCEL MEM_MAP centers_3 to ram_s18_s18(18) at 0

```

```

%!ACCEL MEM_MAP centers_4 to ram_s18_s18(19) at 0

```

```

%!ACCEL MEM_MAP centers_5 to ram_s18_s18(20) at 0

```

```

if loop_flag == 1

```

```

%-----Function to calculate the log version probability in each cluster

```

```

%----->>>>LogProXiL =

```

```

%----->>>>fun_logproxil(indatabuf,covars_inv,covars_det,centers);<<<<<<

```

```

%!ACCEL SHAPE acc_1(1)

```

```

%!ACCEL SHAPE acc_2(1)

```

```

%!ACCEL SHAPE acc_3(1)

```

```

%!ACCEL SHAPE acc_4(1)

```

```

%!ACCEL SHAPE acc_5(1)

```

```

%!ACCEL SHAPE LogProXiL(ncluster)

```

```

%!ACCEL SHAPE acc(ncluster)

```

```

%!ACCEL SHAPE sub_1(ncluster)

```

```

%!ACCEL SHAPE sub_2(ncluster)

```

```

%!ACCEL SHAPE mult(ncluster)

```

```

%!ACCEL SHAPE sqa(ncluster)

```

```

%!ACCEL SHAPE sub_temp(ncluster)

```

```

acc(1) = quantize(q1,0);

```



```

%!ACCEL SHAPE maxa(1)
%!ACCEL SHAPE a(5)
maxa = quantize(q1,max(LogProXiL));

```

```

%!ACCEL REGISTER a
% NOTE: This loop will not after synthesis in ISE.
a(1) = quantize(q1,-LogProXiL(1) + maxa);
a(2) = quantize(q1,-LogProXiL(2) + maxa);
a(3) = quantize(q1,-LogProXiL(3) + maxa);
a(4) = quantize(q1,-LogProXiL(4) + maxa);
a(5) = quantize(q1,-LogProXiL(5) + maxa);

```

```

for m_2 = quantize(q_int,1:ncluster)
  if a(m_2) > 4
    a(m_2) = quantize(q1,3.998);
  end
end

```

```

for m_3 = quantize(q_int,1:ncluster)
  %!ACCEL SHAPE a_addr(1)
  a_addr = accel_bitshl(a(m_3),8,q1,q_addr); % Use shift left to get the address,
  %!ACCEL SHAPE lut_a_content(1024)
  %!ACCEL SHAPE a_lut_out(5)
  a_lut_out(m_3) = quantize(q_lut_exp,lut_exp_content(a_addr+1));
end
%!ACCEL SHAPE c(1)
  c = quantize(q1,0);
  %!ACCEL REGISTER c_temp
  %!ACCEL REGISTER c
  %!ACCEL USE embedmults(10)
%,embedmults(11),embedmults(12),embedmults(13),embedmults(14)

```

```

for m_4 = quantize(q_int,1:ncluster)
  c_temp = quantize(q1, priors(m_4) * a_lut_out(m_4));
  c = quantize(q1,c + c_temp);
end

```

```

if c > 0.5
  c = quantize(q1,0.4995);
end
%!ACCEL REGISTER c_addr
%!ACCEL REGISTER c_lut_out
%!ACCEL SHAPE c_addr(1)
c_addr = accel_bitshl(c,11,q1,q_addr);
%!ACCEL SHAPE lut_log_content(1024)
%!ACCEL SHAPE c_lut_out(1)
c_lut_out = quantize(q_lut_log,lut_log_content(c_addr+1));

```

```

%!ACCEL REGISTER LogLikeHoodXi
%!ACCEL SHAPE LogLikeHood(1)
%!ACCEL SHAPE LogLikeHoodX(1)
LogLikeHoodXi = quantize(q_loglike,c_lut_out + maxa);

%!ACCEL REGISTER LogLikeHoodX_temp
%!ACCEL SHAPE LogLikeHoodX_temp(1)
%!ACCEL USE embedmults(15)
LogLikeHoodX_temp = quantize(q1,LogLikeHoodX + LogLikeHoodXi * ndata_inv);
LogLikeHoodX = quantize(q1,LogLikeHoodX_temp);

%!ACCEL SHAPE LogProLXi(5)

LogProLXi(1) = quantize(q1,-LogProXiL(1) - priors_log(1) + LogLikeHoodXi);
LogProLXi(2) = quantize(q1,-LogProXiL(2) - priors_log(2) + LogLikeHoodXi);
LogProLXi(3) = quantize(q1,-LogProXiL(3) - priors_log(3) + LogLikeHoodXi);
LogProLXi(4) = quantize(q1,-LogProXiL(4) - priors_log(4) + LogLikeHoodXi);
LogProLXi(5) = quantize(q1,-LogProXiL(5) - priors_log(5) + LogLikeHoodXi);

for m_5_2 = quantize(q_int,1:5)
    if LogProLXi(m_5_2) > 4
        LogProLXi(m_5_2) = quantize(q1,3.998);
    end
end
for m_5_3 = quantize(q_int,1:5)
    if LogProLXi(m_5_3) < 0
        LogProLXi(m_5_3) = quantize(q1,0);
    end
end

for m_6_1 = quantize(q_int,1:ncluster)
    %!ACCEL SHAPE LogProLXi_addr(5)
    LogProLXi_addr(m_6_1) = accel_bitshl(LogProLXi(m_6_1),8,q1,q_addr);
end
    %!ACCEL SHAPE lut_exp_ProLXi(1024,ncluster)
    %!ACCEL TILE m_6
    %!ACCEL MEM_MAP ProLXi(:,m_6) TO ram_s18_s18(m_6+5) AT 0
    %!ACCEL MEM_MAP lut_exp_ProLXi(:,m_6) TO ram_s18_s18(m_6+10) AT 0
    for m_6 = quantize(q_int,1:ncluster)
        %!ACCEL SHAPE ProLXi(ndata,ncluster)
        ProLXi(id,m_6) = quantize(q_lut_exp,lut_exp_ProLXi(LogProLXi_addr(m_6) + 1,m_6));
    end

    %!ACCEL SHAPE mid_priors(5)
    %!ACCEL TILE m_7
    for m_7 = quantize(q_int,1:ncluster)
        mid_priors(m_7) = quantize(q_mid_priors,mid_priors(m_7) + ProLXi(id,m_7));
    end
    %!ACCEL SHAPE mid_centers(ndim,ncluster)
    %!ACCEL SHAPE mid_centers_temp_1(ncluster)

```

```

%!ACCEL SHAPE mid_centers_temp_2(ncluster)
%!ACCEL REGISTER mid_centers_temp_1
%!ACCEL REGISTER mid_centers_temp_2
for d_2 = quantize(q_int,1:ndim)

    %!ACCEL TILE m_i_1
        %!ACCEL USE embedmults(15+m_i_1)

    for m_i_1 = quantize(q_int,1:ncluster)
        mid_centers_temp_1(m_i_1)
            = quantize(q1,indatabuf(d_2,m_i_1) * ProLXi(id,m_i_1));
        mid_centers_temp_2(m_i_1)
            = quantize(q1,mid_centers_temp_1(m_i_1)+mid_centers(d_2,m_i_1));
        mid_centers(d_2,m_i_1)
            = quantize(q_mid_centers,mid_centers_temp_2(m_i_1));
    end

end

id = quantize(q_int,id+1);

if id == quantize(q_int,ndata+1);

    % Here is a 30 bit multiplier
    step_incr = quantize(q1,LogLikeHoodX - newLikeHood);
    % To calculate the increase of the likelihood
    newLikeHood = quantize(q1,LogLikeHoodX);

    %!ACCEL SYNTHESIS OFF
    newLikeHood_bin = num2bin(q1,newLikeHood);
    newLikeHood_num = bin2num(q1_int,newLikeHood_bin);
    fprintf('%d: LH is %g      Improve is %d      Num is %d      Bin is
%s\n',LoopCount,newLikeHood,step_incr,newLikeHood_num,num2str(newLikeHood_bin));
    %!ACCEL SYNTHESIS ON

    if step_incr < 0
        step_incr = -step_incr;
    end
    %%-----Function to get the inv of the mid_priors
    %%----->>>>>>mid_priors_inv =
    %%----->>>>>>fun_lut_priors_mid_inv(mid_priors);
    %!ACCEL SHAPE mid_priors_inv_addr(1)
    %!ACCEL SHAPE mid_priors_inv(5)
    %!ACCEL SHAPE lut_inv_content(1024)
    for m_8 = quantize(q_int,1:ncluster)
        mid_priors_inv_addr = accel_bitshl(mid_priors(m_8),1,q1,q_addr);
        mid_priors_inv(m_8)
            = quantize(q_lut_inv,lut_inv_content(mid_priors_inv_addr+1));
    % Change to 2 for simulation

```





```

    if covars(d_6,1) > upper_const
covars(d_6,1) = quantize(q_covars,upper_const); % 2^-2 - 2^-12
    end
    end

for d_9 = quantize(q_int,1:ndim)
    if covars(d_9,2) > upper_const
covars(d_9,2) = quantize(q_covars,upper_const); % 2^-2 - 2^-12
    end
end

for d_10 = quantize(q_int,1:ndim)
    if covars(d_10,3) > upper_const
covars(d_10,3) = quantize(q_covars,upper_const); % 2^-2 - 2^-12
    end
end

for d_11 = quantize(q_int,1:ndim)
    if covars(d_11,4) > upper_const
covars(d_11,4) = quantize(q_covars,upper_const); % 2^-2 - 2^-12
    end
end

for d_12 = quantize(q_int,1:ndim)
    if covars(d_12,5) > upper_const
covars(d_12,5) = quantize(q_covars,upper_const); % 2^-2 - 2^-12
    end
end

for d_6_1 = quantize(q_int,1:ndim)
    if covars(d_6_1,1) < const_zero
covars(d_6_1,1) = quantize(q_covars,const_zero); % 2^-2 - 2^-12
    end
end

for d_9_1 = quantize(q_int,1:ndim)
    if covars(d_9_1,2) < const_zero
covars(d_9_1,2) = quantize(q_covars,const_zero); % 2^-2 - 2^-12
    end
end

for d_10_1 = quantize(q_int,1:ndim)
    if covars(d_10_1,3) < const_zero
covars(d_10_1,3) = quantize(q_covars,const_zero); % 2^-2 - 2^-12
    end
end

for d_11_1 = quantize(q_int,1:ndim)

```



```
id_5 = quantize(q_data,centers_5(1));
%-----
outdatabuf_1 = quantize(q1,covars (1));
outdatabuf_2 = quantize(q1,covars (2));
outdatabuf_3 = quantize(q1,covars (3));
outdatabuf_4 = quantize(q1,covars (4));
outdatabuf_5 = quantize(q1,covars (5));
outdatabuf_6 = quantize(q1,priors(1));
outdatabuf_7 = quantize(q1,priors(2));
outdatabuf_8 = quantize(q1,priors(3));
outdatabuf_9 = quantize(q1,priors(4));
outdatabuf_10 = quantize(q1,priors(5));
outdatabuf_11 = quantize(q1,newLikeHood);

outdata_1(n) = quantize(q1,outdatabuf_1);
outdata_2(n) = quantize(q1,outdatabuf_2);
outdata_3(n) = quantize(q1,outdatabuf_3);
outdata_4(n) = quantize(q1,outdatabuf_4);
outdata_5(n) = quantize(q1,outdatabuf_5);
outdata_6(n) = quantize(q1,outdatabuf_6);
outdata_7(n) = quantize(q1,outdatabuf_7);
outdata_8(n) = quantize(q1,outdatabuf_8);
outdata_9(n) = quantize(q1,outdatabuf_9);
outdata_10(n) = quantize(q1,outdatabuf_10);
outdata_11(n) = quantize(q1,outdatabuf_11);
%!ACCEL END_HARDWARE
outdata_1,outdata_2,outdata_3,outdata_4,outdata_5,outdata_11
end
%end
```

MONTANA STATE UNIVERSITY - BOZEMAN



3 1762 10392217 3