Massively Parallel Hidden Markov Models for Wireless Applications

Shawn R. Hymel

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science
In
Electrical Engineering

December 5, 2011
Blacksburg, VA

Keywords: Hidden Markov Models, Parallel Processing, Signal Recognition, GPU, GPGPU, CUDA

Massively Parallel Hidden Markov Models for Wireless Applications

Shawn R. Hymel

ABSTRACT

Cognitive radio is a growing field in communications which allows a radio to automatically configure its transmission or reception properties in order to reduce interference, provide better quality of service, or allow for more users in a given spectrum. Such processes require several complex features that are currently being utilized in cognitive radio. Two such features, spectrum sensing and identification, have been implemented in numerous ways, however, they generally suffer from high computational complexity. Additionally, Hidden Markov Models (HMMs) are a widely used mathematical modeling tool used in various fields of engineering and sciences. In electrical and computer engineering, it is used in several areas, including speech recognition, handwriting recognition, artificial intelligence, queuing theory, and are used to model fading in communication channels.

The research presented in this thesis proposes a new approach to spectrum identification using a parallel implementation of Hidden Markov Models. Algorithms involving HMMs are usually implemented in the traditional serial manner, which have prohibitively long runtimes. In this work, we study their use in parallel implementations and compare our approach to traditional serial implementations. Timing and power measurements are taken and used to show that the parallel implementation can achieve well over 100× speedup in certain situations. To demonstrate the utility of this new parallel algorithm using graphics processing units (GPUs), a new method for signal identification is proposed for both serial and parallel implementations using HMMs. The method achieved high recognition at -10 dB E_b/N_0. HMMs can benefit from parallel implementation in certain circumstances, specifically, in models that have many states or when multiple models are used in conjunction.

Acknowledgements

I want to express my utmost gratitude to my family and friends who have supported me throughout my endeavor to continue my education. I must thank my parents, Maureen and Bryan, my sister Margaux, and my brother William, all of whom have been a continuing source of support and inspiration in my life, both personally and academically.

I am especially appreciative to my advisor, Dr. Jeffrey Reed, who has been instrumental in offering guidance, support and direction in my research. Additionally, I could not have gotten far without Dr. Ihsan Akbar's invaluable help, often over the phone while taking time out of his busy schedule. I also want to thank Dr. Steven Ellingson, who is part of the Mobile and Portable Radio Research Group (MPRG) at Wireless@VT, for providing support and offering inspiration to me.

I wish to thank all the faculty, staff, and fellow researchers in the MPRG. They fostered a sense of family and encouraged hard work for the sake of research in a growing field. I felt like part of a family in the lab and I am eternally grateful for all their support.

I would also like to convey my sincerest appreciation to Alina for proofreading this lengthy document.

Finally, many thanks to my friends at Solely Swing for giving me a second home and for providing a creative outlet for stress relief while at Virginia Tech.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Hidden Markov Models (HMMs) [1] are a statistical modeling tool used to describe a series of events that go through varying states. Each state affects the probability of the outcome for each specific event. These models have seen a number of uses in many fields, including biology for protein structure [2], channel modeling in communications [3], and pattern recognition [4]. In the realm of pattern recognition, HMMs have seen wide success in facial recognition [5,6] and automatic speech recognition (ASR) [4]. Additionally, much research has been accomplished using HMMs for other purposes, including handwriting analysis, recognition in various languages [7,8], signal detection and classification [9], and specific emitter identification [10]. The algorithms using HMMs, however, require a high computational complexity. In recent years, research has shown an improvement on utilizing new hardware and programming techniques to reduce this complexity.

One such approach utilizes parallel processing in order to achieve massive a massive speedup in many algorithms. Central Processing Units (CPUs) in computers have trended toward multiple cores recently, in order to support multitasking and parallel processing among tasks. Graphics Processing Units (GPUs) have been in existence since the 1980s and have focused on parallel processing in order to enhance the on-screen graphics experience for users. This includes video games, video playback, and 2D/3D video rendering. Recently, GPU designers have provided ways for users to program graphics cards in order to perform tasks outside of video processing, in what has become known as "General-Purpose Computing on Graphics Processing Units" (GPGPU) [11]. GPUs use the "Single Instruction Multiple Data" (SIMD) computational method in order to accomplish their tasks. SIMD is implemented by loading many small processing units with different sets of data and instructing them to execute a single instruction at the same time. This means that multiple threads cannot be run simultaneously; however, it provides a method to parallelize calculations in order to achieve a speedup in various algorithms. While the GPU architecture was specifically built for graphics and video, there are many algorithms that can benefit from parallelization, including Hidden Markov Models [12, 13].

1.1 Thesis Objectives

This thesis denotes the following objectives: detail the various HMM algorithms; discuss the architecture, advantages, and tradeoffs of GPUs; propose methods to improve existing implementations of HMM algorithms; outline the tests and results used to verify such improvements; introduce a new method utilizing GPUs and HMMs to identify modulation schemes within a collected signal.

First, HMMs are discussed in detail in order to provide a solid background on probabilities and their use in HMMs. Three major HMM algorithms are presented, along with methods to evaluate, train, and discover the hidden states. These algorithms see use in the various applications presented, including the proposed architecture to identify modulation schemes.

Second, the history of GPGPU is given along with the various aspects of processing on the GPU. NVIDIA's Compute Unified Device Architecture (CUDA) is used throughout the thesis as the primary means to program a graphics card. As a result, the specific software and hardware features of CUDA are described along with methods to perform general parallelization of computations. The specific algorithms described in the first section are then broken down and parallelized for implementation in GPGPU and specifically CUDA. Tradeoffs and problem areas are discussed. These parallel algorithms are implemented in CUDA. Time and power measurements are taken in order to compare the C and CUDA implementations and discussed for specific sets of hardware.

Finally, a novel approach to signal classification using the Spectral Correlation Function (SCF) is proposed. The design is implemented in C and CUDA. Results for the classification are discussed, as well as timing details comparing the C and CUDA implementations. The thesis culminates with several conclusions about HMM parallelization and recommendations for potential future research.

The third and fourth sections (chapters four and five) have been combined into a paper accepted in the Wireless Innovation Conference and Product Exposition (SDR'11 - WInnComm). Future papers are projected based on the novel approach for signal classification and parallel HMM algorithms.

1.2 Summary

Chapter 1 describes the basic concepts behind Hidden Markov Models and gives several applications of HMMs. A brief overview of general computing on graphics cards is given along with several important definitions and terms.

Chapter 2

Basic Theory

This chapter provides the necessary background on Markov processes, Hidden Markov Models, the associated algorithms, modulation, and the Spectral Correlation Function. HMMs are built up from basic probabilities and the Markov process. The basic formats for discrete-density HMMs are provided along with a brief discussion on continuous-density HMMs. The algorithms for the three canonical HMM problems are discussed along with their use in pattern recognition. Finally, a brief overview of signal modulation is given and how the SCF can be applied to such signals. Several examples for each of the concepts are provided for clarification.

2.1 State Transitions

We can mathematically model many phenomena as a series of states. For our purposes, weather will be used as the example. While the exact type of weather at any given time is made up of a number of variables (temperature, humidity, air pressure, exact amount of rain/snow, wind, etc.), we can generalize these variables into states, such as "sunny," "windy," "rainy," "snowy," etc. If we constrain this example even further, we can reduce the number of states. Let us assume that we live in a particular biome that only experiences rain or sun throughout the year with absolutely no chance of snow (e.g., southern United States). Additionally, we will assume that we are not concerned with "windy" being a state. Therefore, we have reduced the model to two states: "sunny" and "rainy."

While the weather obviously changes in a continuous manner, we can model the changes as discrete patterns. Continuing the example, let us assume that each day Andronicus wakes up at exactly 9:00 am and notes the weather outside of him window. he chronicles the state of the weather as either "sunny" or "rainy," because he lives in the same southern US town. As a result, him journal might look something like the following:

Rainy
Sunny
Rainy
Rainy
Rainy
Sunny
Sunny
...

This can also be depicted as a state diagram with transitions, as shown in Figure 2.1.

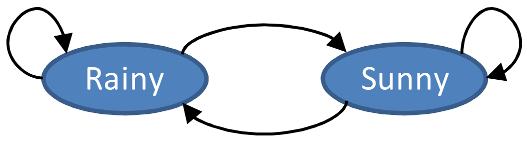


Figure 2.1: 2-State Transition Diagram

Note that this model can be extended to accommodate any number of states and transitions, as shown in Figure 2.2. In this example, we use a generic numbering system for the states, 1 to *N*. These states can be used to describe any number of phenomena, such as weather, number of lily pads in a pond that a frog can choose to jump to [14], state of the market, etc. Additionally, note that some states may be isolated from other states (islands), some states can transition back into themselves, and not all states require transitions to all other states.

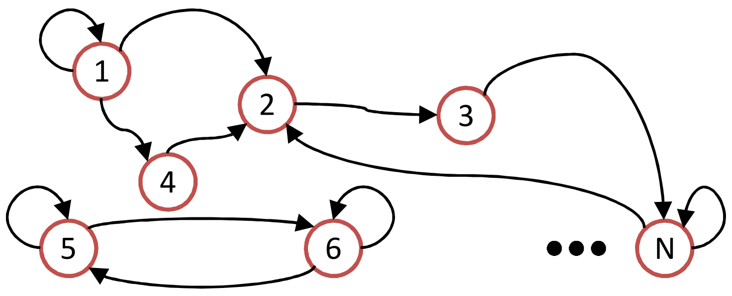


Figure 2.2: ***N*-State Transition Diagram**

Now that we have assigned numbers to each of the states, we can derive the *state-transition* process. Let s_n represent the state of the system at any given time *n*. Note that this is a *discrete-time* system as described earlier, based on samples taken at each instant *n*. This gives us the ability to record the entire sequence of states taken at each *n* as

$$s_0, s_1, s_2, \ldots, s_{n-1}, s_n, s_{n+1}, \ldots$$

Therefore, with the assumption that the transitions are *nondeterministic*, we can model the behavior of the transitions as a set of probabilities, given by

$$\Pr(s_{n+1} \mid s_n, s_{n-1}, \ldots s_0) \tag{2.1}$$

which states that the probability of transitioning to a given state s_{n+1} at time n+1 is dependent on all other states prior to that transition, starting with the first state s_0 up through the state at time *n*.

2.2 Markov Process

A process like the system described in section 2.1 is considered to be a *Markov process* if it exhibits the *Markov property*. Specifically, the discrete-time Markov process is known as a *Markov chain*. The Markov property is an assumption that states that the probability of moving from one state to the next (e.g. *n* to *n*+1) is dependent only on the previous state in the system (e.g., s_n) [15]. This is stated formally by the equation

$$\Pr(s_{n+1}|s_n, s_{n-1}, \ldots s_0) = Pr(s_{n+1} \mid s_n) \tag{2.2}$$

Equation 2.2 specifies that the probability of transitioning from state s_n to state s_{n+1} is independent of any of the other prior transitions. While few real physical systems exhibit this property, it is a useful assumption in mathematically modeling various phenomena that demonstrate the ability to move through states.

Additionally, we assume other aspects of the Markov process for the purposes in this thesis. For example, there are a limited, pre-determined, discrete, and countable number of states in the modeled system. As a result, the probability of transitioning from any one state to another must be *equal or greater than* 0 and the probabilities of transitioning from one state to all of the other states must sum to 1:

$$\sum_{j=0}^{N} \Pr(s_{n+1} = j \mid s_n) = 1 \tag{2.3}$$

as per the *axioms of probability* [15].

2.3 Transition Probabilities

Given a Markov chain, we can begin to define the properties of the model. The most important aspect of the chain is the definition of the state transition probabilities. This is given by

$$a_{ij} = \Pr(s_{n+1} = j \mid s_n = i) \quad 1 \le i,\ j \le N,\ n = 0, 1, 2, \ldots \tag{2.4}$$

where a_{ij} gives the probability of transitioning from state i to state j from time n to n+1.

In continuing with our weather example, we can monitor the weather over the course of weeks, months, years, etc. and determine the probability of the transitions from one type of weather to the next (assuming, of course, that the weather exhibits the Markov property). Examples of such transitions are shown in Figure 2.3.

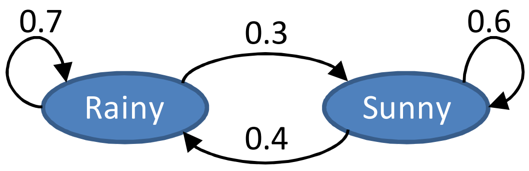


Figure 2.3: Weather Transition Probabilities

Assuming that rainy is state 1 and sunny is state 2, we can define the transition probabilities as follows: $a_{11} = 0.7$, $a_{12} = 0.3$, $a_{21} = 0.4$, $a_{22} = 0.6$. For convenience, we can express these probabilities in a matrix:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}$$

We will call this matrix **A**. Expressed in the general form, we get:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1j} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2j} & \cdots & a_{2N} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{ij} & \cdots & a_{iN} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{Nj} & \cdots & a_{NN} \end{bmatrix} \tag{2.5}$$

Since the elements of **A** must be in the range of [0, 1] and the rows sum to 1 as per the axioms of probability, **A** is considered to be a *stochastic matrix* [15]. Figure 2.4 portrays the general probabilities from **A** on each of the states from Figure 2.2.

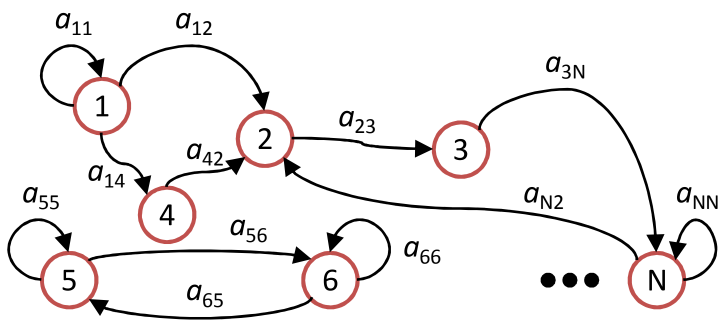


Figure 2.4: Example of General State Transition Probabilities

2.4 Symbol Output Probabilities

Let us assume that we have an urn filled with colored balls, as per the example given by [16]. Each ball can be 1 of 3 colors: red (R), blue (B), and yellow (Y).

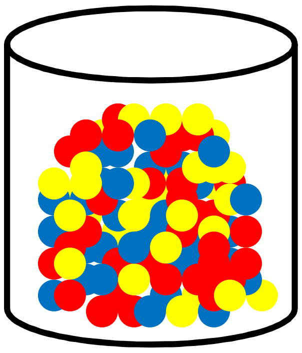


Figure 2.5: Urn Filled With Different Colored Balls

We pick a ball at random, note its color, and then return the ball to the urn. The proportion of each type of ball in the urn determines the probability of randomly choosing a particular color. The probabilities of observing each color on each draw is given by: Pr(R), Pr(B), and Pr(Y).

We can then consider a more complex case: multiple urns. We will use the example of three urns. As before, each of the urns is filled with a number of colored balls. However, the proportion of each of the colored balls is different for each urn. As a result, the probability of drawing a particular color from each urn differs.

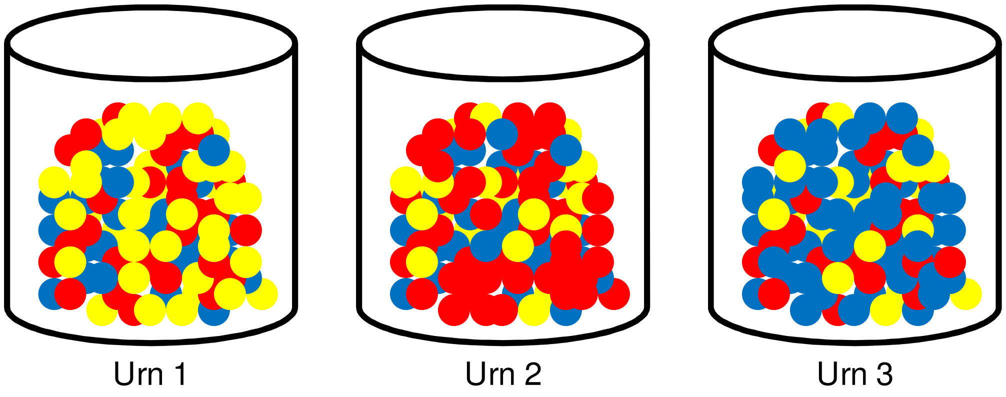


Figure 2.6: Three Urns Filled With Different Colored Balls

The probability of choosing a particular color is dependent on the urn from which it is drawn. This is expressed as

$$\Pr(color \mid urn)$$

For example, the probability of choosing a blue ball from urn 2 is expressed as

$$\Pr(color \ = "blue" \mid urn = 2)$$

For simplicity, we can assign this probability to a variable b. We will use multiple b values, where the subscript determines the observable (color) given the state (urn). For instance, we use the following parameters: yellow = 1, red = 2, blue = 3 and each of the urns (1, 2, and 3) correspond to a similarly numbered state. Therefore, we express the probability of drawing a blue ball (3) from urn 2 as

$$b_{32} = \Pr(color \ = "blue" \mid urn = 2)$$

As an example, let's assign some probabilities. We will assume that urn 1 favors yellow, urn 2 favors red, and urn 3 favors blue. We perform some statistical experiments (numerous random draws) and determine the following, expressed in matrix form:

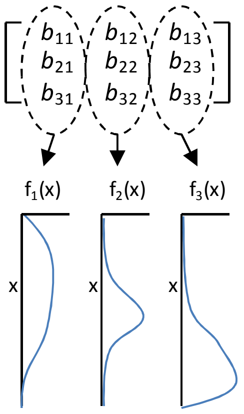
$$\begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} 0.8 & 0.2 & 0.2 \\ 0.1 & 0.7 & 0.2 \\ 0.1 & 0.1 & 0.6 \end{bmatrix}$$

We will call this matrix **B**. Expressed in the general form using *N* states (urns) and *M* symbols (colors), we have:

$$\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1j} & \cdots & b_{1N} \\ b_{21} & b_{22} & \cdots & b_{2j} & \cdots & b_{2N} \\ \vdots & \vdots & & \vdots & & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{kj} & \cdots & b_{kN} \\ \vdots & \vdots & & \vdots & & \vdots \\ b_{M1} & b_{M2} & \cdots & b_{Mj} & \cdots & b_{MN} \end{bmatrix} \tag{2.6}$$

Notice that each *row* in **B** expresses the probabilities of drawing one particular color *k* from all of the urns, and each *column* gives the probabilities of drawing all the colors from one particular urn. As a result, each column must sum to 1. Therefore, this is a transposed *stochastic matrix*.

Most importantly, each column, taken individually, gives the *probability mass function* (pmf) of observing each of the symbols (colors) from a particular state (urn). It is possible to use continuous densities, and thus the *probability density function* (pdf), in place of a discrete-value **B** matrix, but these are left for discussion outside of this thesis and more information can be found in [17]. However, it is sometimes useful to visualize the **B** matrix as a collection of pdf's or pmf's, and which pdf/pmf to choose is based on the current state [5]. Figure 2.7 shows this visualization with the assumption that the individual *b* values can be approximated by continuous pdf's.



b_{11} b_{12} b_{13}
b_{21} b_{22} b_{23}
b_{31} b_{32} b_{33}
$f_1(x)$
$f_2(x)$
$f_3(x)$
x
x
x

Figure 2.7: Visualizing B as a Collection of PDFs

2.5 Hidden Markov Models

We have almost completely defined a Hidden Markov Model. However, we still need to introduce the concepts of time, initial state probabilities, and hidden states.

Using the urn example, let us pick an urn at random and draw a ball from it, noting the color. To elaborate, the probabilities of choosing the next urn is dependent on the current urn chosen. This provides the notion of *states*, and the probability of choosing a particular urn is given by **A**. Then, from that urn, we randomly draw a ball, note its color, and replace the ball. The probability of choosing a color is given by **B**. We then choose a new urn, and the process is repeated. An example output of this is shown in Figure 2.8.

Time (n):	...	3	4	5	6	7	8	9	...
Urn:	...	2	3	3	1	3	2	2	...
Color:	...	🔴	🔵	🟡	🟡	🔵	🔴	🔵	...

Figure 2.8: Example Sequence of Balls Drawn

While this model works well for any time n, how does one determine the initial state when $n = 0$? **A** requires the previous state to be known in order to determine the probability of the next state. Therefore, we need to introduce the concept of *initial state probabilities*. This is given by the matrix $\boldsymbol{\pi}$. Each element of $\boldsymbol{\pi}$ gives the probability of the model starting in a particular state at time $n = 0$. So,

$$\boldsymbol{\pi} = [\pi_1 \quad \pi_2 \quad \cdots \quad \pi_i \quad \cdots \quad \pi_N] \tag{2.7}$$

depicts the probabilities of the model entering into one of N states upon initialization. For our urn example, we can let

$$\boldsymbol{\pi} = [0.5 \quad 0.4 \quad 0.1]$$

which states that the probability of choosing urn 1 (Pr("choose urn 1" at $n = 0$)) is 0.5, choosing urn 2 is 0.4, and choosing urn 3 is 0.1.

Moving to the previous example of weather, let us assume that Andronicus has a friend, Bacchus, who lives on the other side of the planet. We will also assume that Bacchus has no connection to the outside world except for a telephone. Since Bacchus is a good friend and wants to know about Andronicus's daily activities, he asks him what he will be doing that day. Andronicus only has three responses: "Walk," "Shop," or "Clean." This is similar to observing a particular color from the urns. As a result, we say that the activities are the possible *symbols*. Once again, we can construct an output probability matrix to model Andronicus's behavior ("Walk" = 1, "Shop" = 2, "Clean" = 3):

$$\mathbf{B} = \begin{bmatrix} 0.1 & 0.6 \\ 0.4 & 0.3 \\ 0.5 & 0.1 \end{bmatrix}$$

Note the differences between the activities example and the urn example. While both have three possible symbols, the urn example has 3 states whereas the weather example has only 2.

Additionally, we can construct an initial probability matrix, which contains the probabilities of the weather for the very first time Bacchus asks Andronicus about him activities:

$$\boldsymbol{\pi} = \begin{bmatrix} 0.6 & 0.4 \end{bmatrix}$$

This model can be visualized by Figure 2.9.

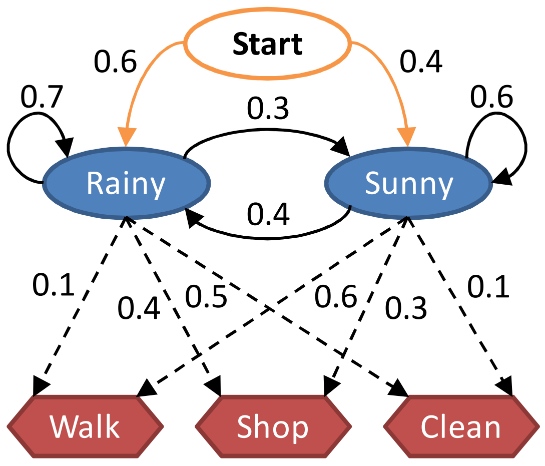


Figure 2.9: HMM Visualization

Since Bacchus cannot observe the weather where Andronicus lives nor does he ask him about the weather, he must make some assumptions or very educated guesses. Using the model given by **A**, **B**, and $\boldsymbol{\pi}$, Bacchus can determine the most likely state of the weather. He knows the likelihood of the weather moving from one state to the next, he knows the likelihood of Andronicus performing a particular activity given the state of the weather, and he knows what the weather is likely going to be like the first time he calls. However, he simply cannot observe the weather. Thus, the state of the weather is *hidden*.

This is similar to the urn example if another person randomly chose the urn and the ball. If the person recording the colors has no idea which urn was chosen, the resulting log might look like the sequence in Figure 2.10.

Time (n):	...	3	4	5	6	7	8	9	...
Color:	...	🔴	🔵	🟡	🟡	🔵	🔴	🔵	...

Figure 2.10: Examples of Balls Drawn from Hidden Urns

The pressing question is then: "Which urn was chosen?" Given the parameters **A**, **B**, and $\boldsymbol{\pi}$, there are ways to determine this, in addition to other questions. These parameters become a powerful tool in statistical modeling, as they are the discrete-time and discrete-density parameters for a *Hidden Markov Model*. The Hidden Markov Model (HMM) is often expressed as the triplet

$$\lambda = (\mathbf{A},\ \mathbf{B},\ \boldsymbol{\pi}) \tag{2.8}$$

to denote the complete set of parameters in compact form.

While creating statistical models of weather patterns and colored balls can be quite entertaining, several important questions arise when dealing with HMMs. The three *canonical problems* [1] are:

1. Given an observation sequence $O = O_1O_2...O_T$ and a model λ, determine the likelihood that the model will generate O: $\Pr(O|\lambda)$.
2. Given an observation sequence O and a model λ, determine the most likely state sequence $Q = q_1q_2...q_T$.
3. Given an observation sequence O, find the parameters of λ that maximizes the probability $\Pr(O|\lambda)$.

Problem 1 is known as model *evaluation*. In this case, we are trying to determine how well a given model outputs an observed sequence. We can also say that we are trying to determine how well a model "fits" an observed sequence. This is extremely useful in cases where we have one observed sequence and several models to choose from, which allows us to pick the model that fits the data best. Choosing the best model for a given data set forms the basis for *pattern recognition* with HMMs. Problem 1 can be solved with the *Forward Algorithm* or the *Backward Algorithm*. This thesis will focus on the *Forward Algorithm* for model evaluation.

Problem 2 can be solved with the *Viterbi Algorithm*. This algorithm is useful in pattern recognition when the sequence of states is as important as or more important than simple model matching. For example, parts of speech may be modeled as individual states, and the sequence of those states could make up a word, phrase, etc. Therefore, we can use the Viterbi Algorithm to determine the best possible word/phrase heard from various parts of speech [18].

Problem 3 is where we attempt to *train* the model using a set of observations. The particular sequence is used to iteratively adjust the model parameters to maximize the probability of the model matching the sequence. Training can be accomplished by several approaches; one of the more popular ones is the *Baum-Welch Algorithm*. For example, spoken words may be modeled as HMMs, in which case the HMM is adjusted to better fit a particular word.

2.6 Forward Algorithm

The solution to problem 1 is accomplished by finding the probability that the model will produce the observed sequence. For this solution we need an observation sequence that *could have* come from the model, meaning that the symbols produced come from the same set of symbols contained in the model's parameters (i.e. **B** matrix). Additionally, we need an HMM to compare the sequence to. Mathematically, this is expressed by $\Pr(O|\lambda)$.

There are two algorithms that can accomplish the evaluation problem. That is the *Forward Algorithm* and *Backward Algorithm*. Both examine the probability of transitioning to various states depending on the observed symbol, and both have the same computational complexity: $O(TN^2)$, where T is the number of observations in the sequence and N is the number of states in the HMM. We will focus expressly on the

Forward Algorithm. Note that the Backward Algorithm is given later as the *backward variables* which are part of the Baum-Welch Algorithm.

The Forward Algorithm is solved by iteratively computing the probabilities of the observations for all T. We introduce a new variable, α, which denotes the probability of the model producing the observed symbol at time t given a particular state j along with the probability of traversing through all states up to state j. Therefore, the sum of all probabilities producing o at time t, given each of the states, gives the probability of the model producing the observation sequence up through t. Figure 2.11 shows an example of these partial probabilities using 3 states.

Each probability is given by

$$\alpha_t(\mathrm{j}) = \Pr(o_t \mid s = j) \times \Pr(\text{all paths to state } j \text{ at } t) \tag{2.9}$$

which calculates the value of α for each state at a particular time t. It is often useful to think of α as a $N \times T$ matrix with each column giving the partial probabilities at time t.

Because the forward algorithm is recursive, we must have an initialization and termination procedure. The initialization is given by calculating the probability of starting in a particular state (given by $\boldsymbol{\pi}$) and producing the first observable, given by $\pi_i b_i(O_1)$. In order to terminate the algorithm, we sum the final values of α across all states at $t = T$. This gives us the probability of traversing through all the states and producing each of the observables in the sequence, thus evaluating the probability that the model produced the observation sequence.

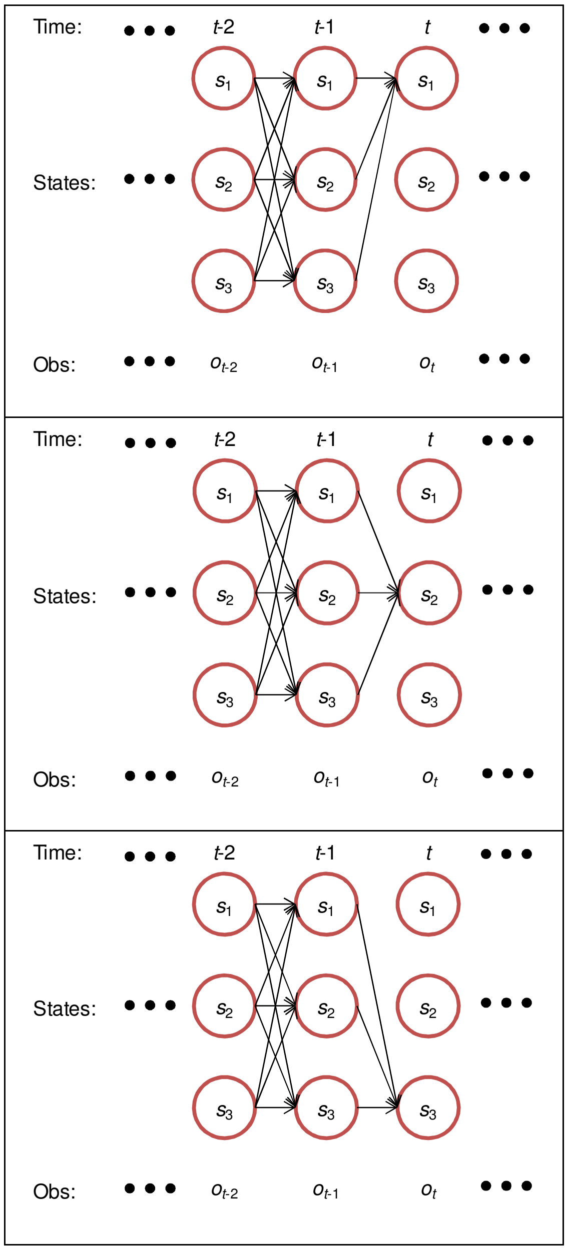


Figure 2.11: Partial Probabilities through the States

Formally, the Forward Algorithm is given by the following steps [1]:

1) Initialization:

$$\alpha_1(i) = \pi_i b_i(O_1), \ \ i = 1, 2, ... N \tag{2.10}$$

2) Induction:

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i) a_{ij} \right] b_j(O_{t+1}), \ 1 \le t \le T-1, \ 1 \le j \le N \tag{2.11}$$

3) Termination:

$$\Pr[O \mid \lambda] = \sum_{i=1}^{N} \alpha_T(i) \tag{2.12}$$

It is important to note that as t increases, the α variables become smaller. As a result, many computer systems risk numerical underflow, which will compromise the algorithm (i.e., all results thereafter are 0). In order to combat the problem of underflow, [19] discusses the concept of using a scaling factor. When implementing the Forward Algorithm, the α values at each instance t are summed.

$$C_t = \sum_{i=1}^{N} \alpha_t(i) \tag{2.13}$$

This produces a 1×T matrix with all of the scaling values for α. Each set of α's at time t are scaled by their appropriate C_t value:

$$\bar{\alpha}_t(j) = \frac{\alpha_t(j)}{C_t} \tag{2.14}$$

The evaluation criteria $\Pr(O| \lambda)$ can now be calculated by

$$\Pr(O|\lambda) = \prod_{t=1}^{T} C_t \tag{2.15}$$

However, this also has the possibility of experiencing underflow. We often take the *log-likelihood* of the evaluation, which is given by $\log_{10}\Pr(O| \lambda)$. Therefore, we can use (2.14) to express the log-likelihood as follows:

$$\log_{10}\Pr(O|\lambda) = \sum_{t=1}^{T} \log_{10} C_t \tag{2.16}$$

The log-likelihood will be used for specific results of model evaluation, assuming the model has been implemented in a program where underflow is likely to be an issue.

2.7 Viterbi Algorithm

Named for Andrew Viterbi [20, 21], the Viterbi Algorithm attempts to find the most likely state solution given a model and an observation sequence. The algorithm is based on *dynamic programming*, which attempts to break a large, complex problem into smaller sub-problems. Dynamic programming has seen wide use in biology, economics, pattern matching, etc.

One application of dynamic programming is the *shortest path problem* [22]. Given the diagram in Figure 2.12 (depicted as a *trellis*), we wish to find the most cost effective route from A to K with the cost of individual hops given on the path arrows. This models a number of things, such as hops to routers in the Internet, distance to connecting cities for airplanes, and so on.

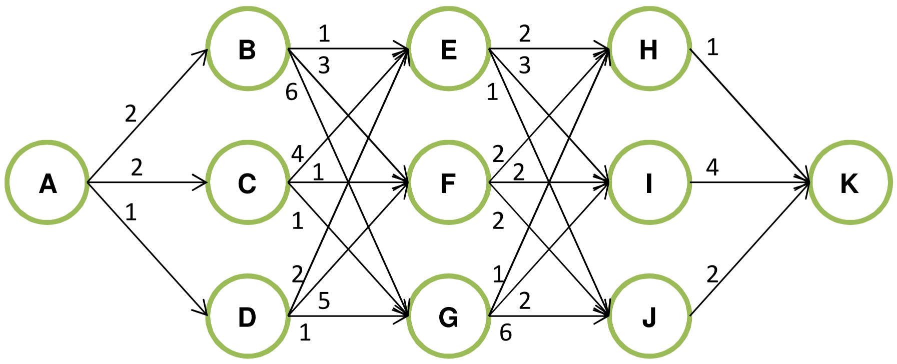


Figure 2.12: Cost of Path Hops Example

The obvious solution is to calculate the cost of all possible routes and then choose the shortest. While this works for a very small number of hops, the number of calculations can quickly grow out of hand. As a result, we need to break up the problem into stages. A stage is a method to separate the different nodes in the diagram. Stage 1 contains A, stage 2 contains B, C, and D, stage 3 contains E, F, and G, and so on. Notice that these are conveniently broken up into the columns of the trellis. We then calculate the cost of all hops from stage 1 to stage 2. The total is then used to calculate the total running cost from stage 2 to stage 3. This step is repeated until we reach the final stage.

With the total cost computed at each stage, we can work our way backwards looking for the shortest route and thereby finding the least costly path.

We now want to apply the shortest path problem to HMMs. Instead of nodes at physical location, let us use states in a given HMM, and instead of node-to-node hop costs, we will use state transition probabilities given by **A**. Figure 2.13 depicts this new model.

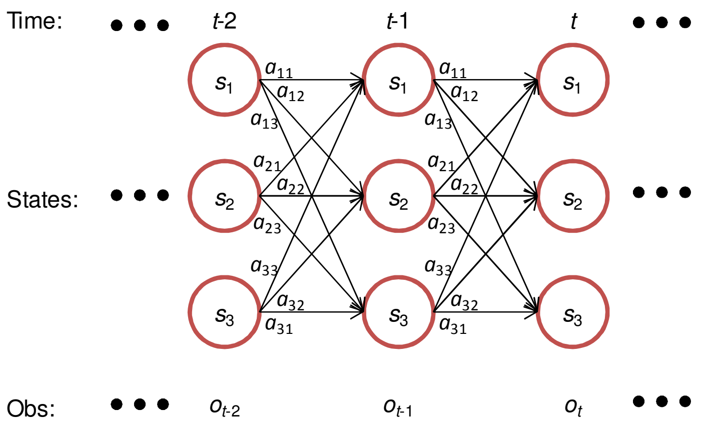


Figure 2.13: State Transition Trellis

We can use the same algorithm found in the shortest path procedure to discover the most likely state sequence. The difference between the path example and HMM states is that we are looking for the maximum cost, which corresponds to the maximum probability instead of the minimum cost. The formal Viterbi algorithm is given by:

1) Initialization:

$$\delta_1(i) = \pi_i b_i(O_1),\ \ i = 1, 2, \ldots N \tag{2.17}$$

2) Recursion:

$$\delta_{t+1}(i) = \max_j \left[\delta_t(j) a_{ij}\right] b_j(O_{t+1}),\ \ 1 \le i \le N,\ \ 1 \le t \le T-1 \tag{2.18}$$

$$\psi_t(i) = \arg\max_j \left[\delta_t(j) a_{ij}\right],\ 1 \le i \le N,\ 1 \le t \le T-1 \tag{2.19}$$

3) Termination:

$$q_T^* = \arg\max_i \left(\delta_T(i)\right) \qquad (2.20$$

4) Path backtracking:

$$q_t^* = \psi_{t+1}\left(q_{t+1}^*\right),\ t = T-1, T-2, \ldots 1 \tag{2.21}$$

The calculations performed in the Viterbi Algorithm are similar to those found in the Forward Algorithm and therefore have a computational complexity of $O(TN^2)$. Unlike the Forward Algorithm, however, we must save the most likely path and then backtrack through those likelihoods to uncover the exact state transitions.

The Viterbi Algorithm suffers the same numerical underflow problems as the Forward Algorithm. As a result, we must scale the running total probability values, δ, at every stage. This is accomplished by:

$$\overline{\delta_t}(i) = \delta_t(i) / \textstyle\sum_{i=1}^{N} \delta_t(i) \tag{2.22}$$

Fortunately, we can discard the δ values once the most likely path is found, as they are not used in further calculations in the algorithm. Only the relative values (i.e., the maximum) of δ are needed, not the exact values. This scaling value can be found throughout the implementations of this algorithm.

2.8 The Baum-Welch Algorithm

The most difficult problem involving HMMs is the process of adjusting the model parameters to match a given observation sequence. However, this requires the model to be physically possible of generating the sequence; the symbol output probabilities must be able to generate all of the symbols seen in the sequence.

There is no known method to analytically solve for **A**, **B**, and **π**. As a result, we must start with a model λ and adjust its parameters until $\Pr(O|\lambda)$ is maximized using a training sequence as shown in Figure 2.14. There are several ways to accomplish this, but we will focus on an application of the *Expectation Maximization* (EM) technique [23] known as the *Baum-Welch Algorithm* (BWA) [16, 25]. This involves calculating several new variables, namely ξ and γ and using them to adjust the model parameters **A**, **B**, and **π** in order to maximize $\Pr(O|\lambda)$.

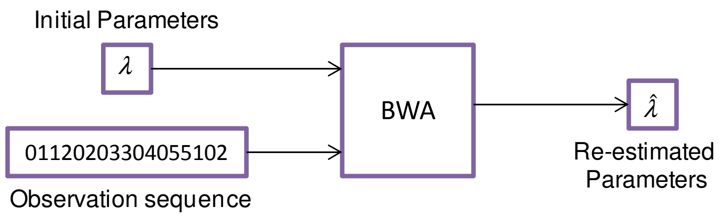


Figure 2.14: Training a Model with the BWA

ξ can be thought of as a 3-dimensional matrix where each 2D slice contains the probability of being in state s_i at time t and state s_j at time t + 1, as depicted by Figure 2.15 and Equation 2.23.

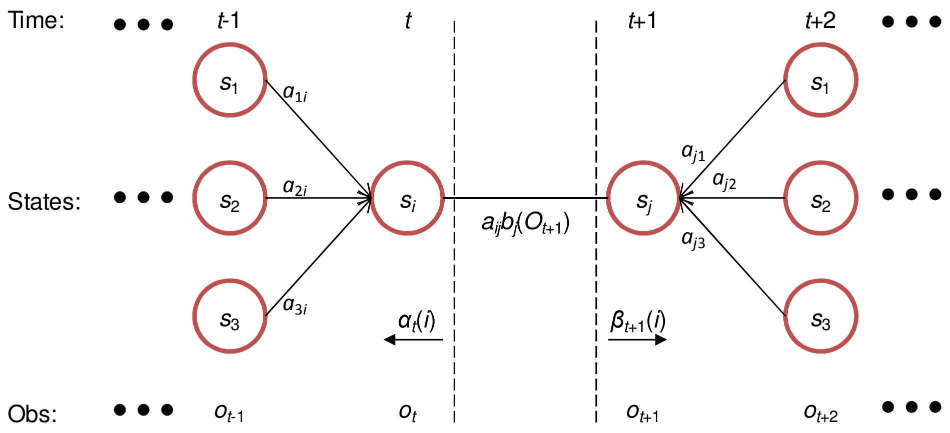


Figure 2.15: Example of Transition from s_i to s_j

This transition is given by the equation

$$\xi_t(i,j) = \Pr\,(q_t = s_i, q_{t+1} = s_j | O, \lambda) \tag{2.23}$$

The Forward Algorithm can be used to obtain the probability of being in state s_i at time *t,* and the Backward Algorithm may be used to obtain the probability of being in state s_j at time $t + 1$. Here, we introduce the concept of β, which describes the probability of being in a state and producing the given observation, given the probability of all state transitions since time *T*. It should be noted that the Backward Algorithm contains the exact same steps as the Forward Algorithm, the difference being that it works backward from *T*. While the Backward Algorithm can be used for the same calculation of $\Pr(O|\lambda)$, it requires that the entire sequence be known. As such, it cannot be used for near real-time pattern recognition as well as the Forward Algorithm. The formal definition of the BWA shows the calculation of β variables, using the Backward Algorithm.

Using α and β, we can calculate ξ:

$$\xi_t(i,j) = \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{\Pr\,(O|\lambda)} \tag{2.24}$$

We must define γ as the probability of being in state s_i at t given the model and observation sequence. Therefore, we can conclude that

$$\gamma_t(i) = \sum_{j=1}^{N} \xi_t(i,j) \tag{2.25}$$

As a result, we can sum ξ and γ across all t to obtain the expected number of transitions from s_i and the expected number of transitions from s_i to s_j.

$$\sum_{t=1}^{T-1} \gamma_t(i) = \text{expected number of transitions from } s_i \tag{2.26}$$

$$\sum_{t=1}^{T-1} \xi_i(i,j) = \text{expected number of transitions from } s_i \text{ to } s_j \tag{2.27}$$

A set of re-estimation parameters can be derived:

$$\hat{\pi}_i = \text{expected number of times in state } s_i \text{ at time } t = 1 \tag{2.28}$$

$$\hat{a}_{ij} = \frac{\text{expected number of transitions from state } s_i \text{to state } s_j}{\text{expected number of transitions from state } s_i} \tag{2.29}$$

$$\hat{b}_j(k) = \frac{\text{expected number of times in state } s_j \text{ and observing symbol } v_k}{\text{expected number of times in state } s_j} \tag{2.30}$$

These parameters are then used to create a new model, $\hat{\lambda}$, then fed back into the BWA to further estimate new parameters for a better model of observation sequence. This iterative process can be done for set number of times or until Pr(O| λ) reaches a maximum. However, realizing an absolute maximum for all possible sets of **A**, **B**, and $\boldsymbol{\pi}$ is intractable. As a result, we must settle for a local maximum or continue to adjust the initial set of parameters until a re-estimated model is deemed sufficient.

The Baum-Welch Algorithm as per [19] is given below:

Step 0: Start with an initial model λ

Step 1: Compute the Forward variables (α)

Initialization:

$$\alpha_1(i) = \pi_i b_i(O_1), \ i = 1, 2, \ldots N \tag{2.31}$$

Induction:

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i) a_{ij}\right] b_j(O_{t+1}), \ 1 \le t \le T-1, \ 1 \le j \le N \tag{2.32}$$

Termination:

$$\Pr[O \mid \lambda] = \sum_{i=1}^{N} \alpha_T(i) \tag{2.33}$$

Step 2: Compute the Backward variables (β)

Initialization:

$$\beta_T(i) = 1, \ i = 1, 2, \ldots N \tag{2.34}$$

Induction:

$$\beta_t(i) = \sum_{j=1}^{N} \beta_{t+1}(j) b_j(O_{t+1}) a_{ij}, \ 1 \le t \le T-1, \ 1 \le j \le N \tag{2.35}$$

Step 3: Compute γ

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\Pr[O \mid \lambda]}, \;\; i = 1, 2, \ldots N \tag{2.36}$$

Step 4: Compute ξ

$$\xi_t(i,j) = \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{\Pr[O \mid \lambda]} \tag{2.37}$$

Step 5: Re-estimate the HMM parameters

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)} \tag{2.38}$$

$$\hat{b}_j(e_k) = \frac{\sum_{t=1 \mid O_t = e_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)} \tag{2.39}$$

$$\hat{\pi}_i = \alpha_1(i)\beta_1(i) \tag{2.40}$$

Step 6: Using the re-estimated parameters, go back to step 1 and repeat for a specified number of iterations or until the desired point of convergence is reached.

Both the Forward and Backward Algorithms in the BWA share the same complexity of $O(TN^2)$. The γ variables require $O(TN)$ operations, and ξ variables require $O(TN^2)$. Re-estimating $\boldsymbol{\pi}$ requires $O(N)$, **A** requires $O(N^2)$ (assuming a running total for ξ), and **B** requires $O(TMN)$ for the implementation given in this thesis. Therefore, the computational complexity of the BWA is either $O(TN^2)$ or $O(TMN)$, whichever is larger. For the sake of simplicity, we will use $O(TN^2)$ for the BWA.

It should be noted that the scaling values calculated for α are also used to scale β to prevent underflow. Additionally, β is initialized by

$$\bar{\beta}_T = \frac{1}{c_T} \tag{2.41}$$

and subsequent β values are calculated as

$$\bar{\beta}_t(i) = \frac{\beta_t(i)}{c_t} \tag{2.42}$$

Finally, the log-likelihood is computed as a byproduct Forward Algorithm and used to compare to a threshold. This determines whether the algorithm should continue to run. For timing tests, the algorithm is often told to run for a set number of times (e.g., once).

2.9 Specifying Model Parameters

2.9.1 Number of Symbols

When we set out to model a set of observation sequences using an HMM, we start with only observations. The easiest question to answer often is the number of symbols to use. With *Continuous Hidden Markov Models* (CHMMs), the number of possible output symbols is infinite. As a result, we use mean and covariance matrices to specify output probabilities rather than **B**. For Discrete-density HMMs, the **B** matrix is used, and we must determine the number of possible symbols.

Specifying symbols is generally accomplished by manually examining the observation streams. For example, binary data means that there are two possible symbols. We can often divide analog data into discrete values in order to model the data with an HMM. This can be accomplished using a variety of methods, such as examining minimum/maximum levels, power, etc.

2.9.2 Number of States

Specifying the number of states can be more problematic than choosing the number of symbols [26]. When choosing the number of states, we often must intuitively consider the number and types of states that a system might pass through. For example, if we are modeling a communications system that experiences Rayleigh fading [3], then we generally want 2 or 3 states; system is either in a fade or not. On the other hand, automatic speech recognition (ASR) systems will use hundreds or more states, because each state represents a spoken utterance [27]. Choosing the correct number of states is crucial to appropriately modeling the system. Simply increasing the states in the model (of the model) will not give best results from training. Therefore, it is advisable to experiment with the number of states to find the optimal number. For detail reading on how to tackle this problem, refer to [28, 29, 30].

2.9.3 HMM Topology

HMMs can be realized in a number of different topologies as exemplified by Figure 2.16 [4]. The easiest to understand is the *ergodic* model, which allows the system to move to any state, from any state.

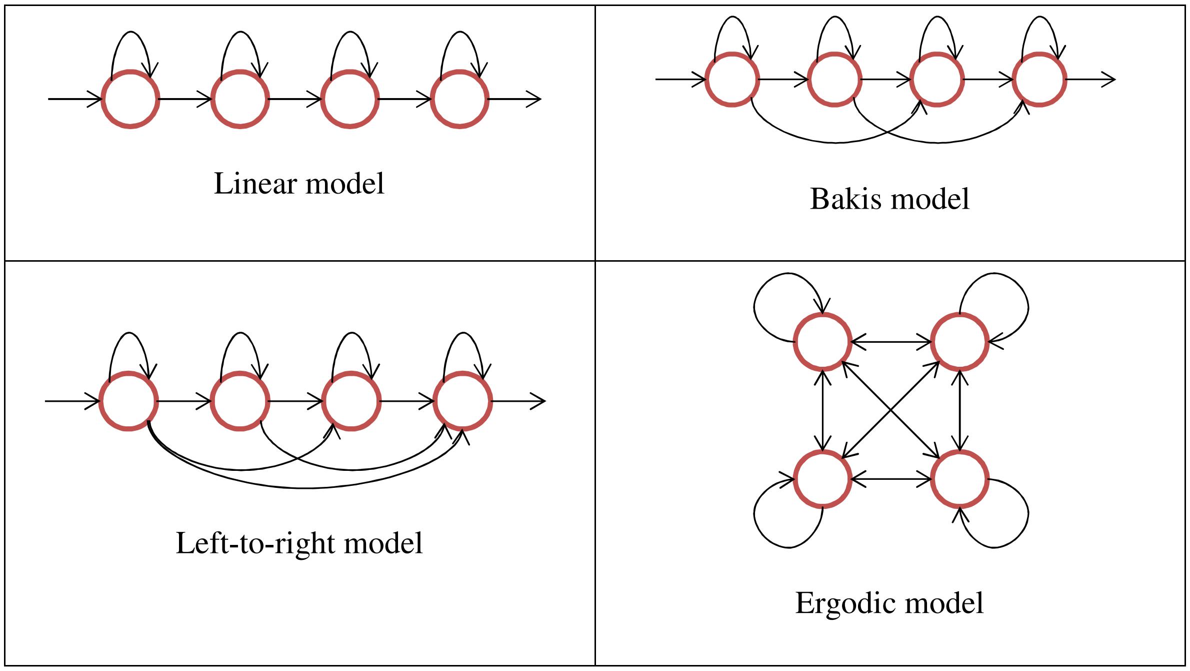


Figure 2.16: Examples of HMM Topologies

Another possible topology is the *linear model*. This model allows transitions from a state to either itself or one other possible state. Similarly, the *Bakis model* allows for skipping states, and the general *left-to-right model* allows movement from any one particular state to any other state, so long as the movement is not backwards on the specified state chain.

The four topologies detailed are generally considered to be the most popular. Linear, Bakis, and Left-to-Right models see use in speech and printed/written character recognition as it allows linear scanning over time or space. Ergodic models are used in a variety of applications, including channel modeling and face recognition.

Choosing the right topology is just as important as choosing the correct number of symbols or states. Setting up the topology can be as simple as configuring the initial **A** matrix to allow only for transitions to and from certain states. After adjusting the parameters in the HMM through the BWA, the topology will not be altered.

2.9.4 Initial Parameters

Specifying the initial parameters for BWA is often the trickiest step. Because the BWA can guarantee only a local maxima, the initial parameters determine the direction that the BWA will re-estimate the parameters before $\Pr(O| \lambda)$ converges. For pattern recognition, the wrong initial parameters can mean that one HMM may model all possible sequences better than any other model, thus having all test subjects be identified as a single case.

For some systems, starting with a set of neutral parameters will allow the BWA to ideally move the parameters in different directions when fed varying training sequences. For example, we could start a 3-state, 2-symbol model with the following:

$$\boldsymbol{A} = \begin{bmatrix} 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3 \end{bmatrix}$$

$$\boldsymbol{B} = \begin{bmatrix} 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & 0.5 \end{bmatrix}$$

$$\boldsymbol{\pi} = \begin{bmatrix} 0.3 & 0.3 & 0.3 \end{bmatrix}$$

While this offers a simple method to choose initial parameters, the model suffers from the possibility of having different models end up with the same final parameters.

More complex algorithms exist, such as the *Segmental K-Means Algorithm,* which assist with dividing the data into various segments and choosing initial parameters [31].

Similar to choosing the number of states, we can also intuitively guess the initial parameters based on actual observations. While this is less precise, it often works well for starting the model prior to BWA optimization. Choosing random starting parameters, so long as they follow the topology desired and create stochastic matrices, is the final step. This can even be further optimized by choosing a large number of random starting parameters, training each with the same sequence, and choosing the model with the best $\Pr(O|\lambda)$.

2.10 Pattern Recognition with HMM

HMMs lend themselves to pattern recognition, as they can be used to find the most likely sequence of events. This is accomplished by using the BWA to train a number of models and the Forward or Backward Algorithm to compute $\Pr(O|\lambda)$.

For any pattern recognition system, a database must be populated with various models as trained from collected sequences of known sources. A label is then attached to the model that explains the category to which the model belongs (be it a particular face, word, signal, etc.). These models are trained using, potentially, the BWA as shown in Figure 2.17.

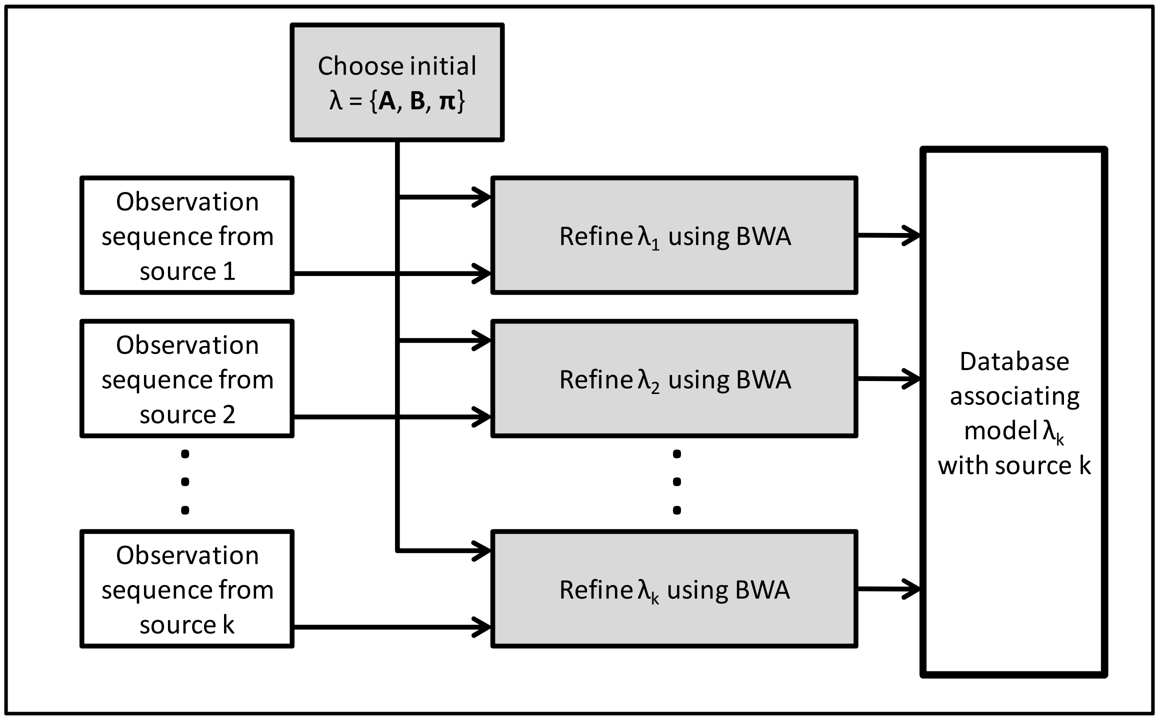

Choose initial
λ = {A, B, π}
Observation sequence from source 1
Observation sequence from source 2
Observation sequence from source k
Refine λ1 using BWA
Refine λ2 using BWA
Refine λk using BWA
Database associating model λk with source k

Figure 2.17: Multiple Model Training Using BWA

Figure 2.18 depicts a general pattern recognition engine using HMMs and the Forward Algorithm. A new, unknown, observation sequence is collected, and the Forward Algorithm (FO) is run using each of the models from the database. The model that produces the highest $\Pr(O|\ \lambda)$ is chosen as the most likely model to match the collected sequence.

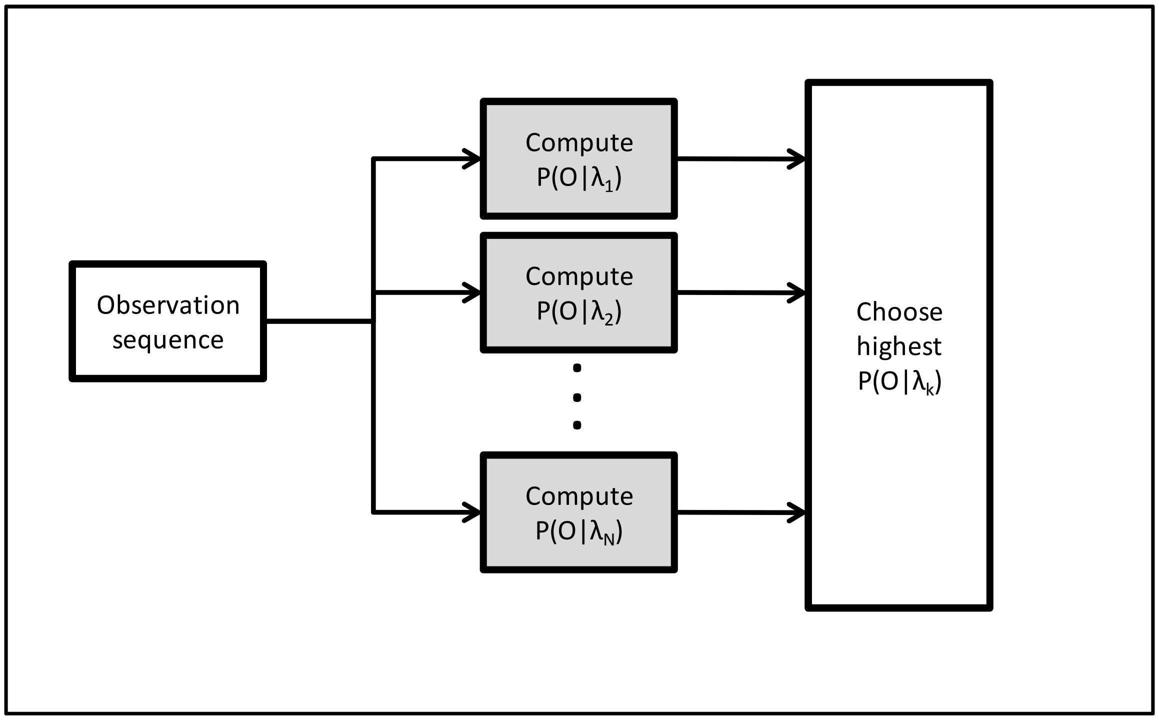


Figure 2.18: Pattern Recognition Engine with HMMs

While other systems are possible for pattern recognition (e.g. using the Viterbi Algorithm to identify a word), we will focus on the system given in Figure 2.18 for parallelization purposes.

2.11 Modulation

Modulation is a technique in telecommunications whereby a high-frequency, periodic waveform (*carrier*) is varied using the lower-frequency, often non-periodic signal (*modulating signal*) that contains the information to be transmitted. Modulation allows for different signals to be transmitted at the same time on different bandpass carrier frequencies (*channels*). Modulation also allows for transmission in environments that could produce detrimental effects at lower frequencies (e.g., *baseband*), such as those with natural or manmade *noise*. We will examine 3 different types of modulation in an attempt to build a pattern recognition engine to automatically identify each of the different types of modulation.

2.11.1 Baseband

Baseband transmission involves sending information through a medium without any modulation. While this is often avoided in wireless transmission schemes for the reasons given above, it is widely used in wired transmission, as it is simple to implement. For this implementation, we use a simple unipolar level scheme of 0 for logic 0 and 1 for a logic 1 and a non-return to zero (NRZ) line code. Figure 2.19 shows this implementation.

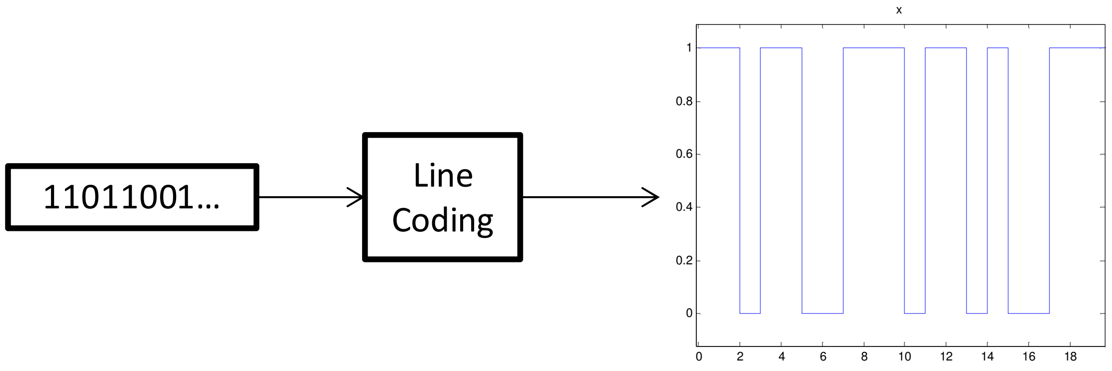


Figure 2.19: Unipolar NRZ at Baseband

2.11.2 Phase Shift Keying

Phase Shift Keying (PSK) encodes data by changing the phase of the carrier [32]. Binary phase shift keying (BPSK) uses a 180° phase shift in the carrier to denote a logic 0 versus a logic 1. As shown by Figure 2.20, this is accomplished by first coding the data as ± 1 and multiplying the data by the carrier. This, in effect, produces the desired 180° shift.

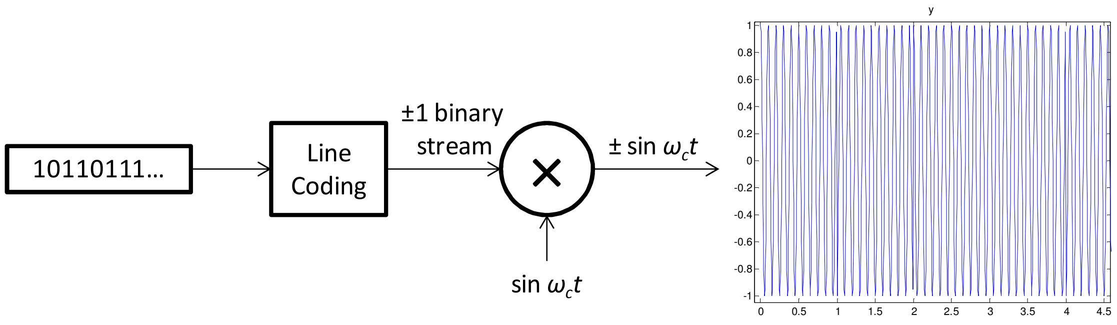


Figure 2.20: BPSK Modulator

We can also use more than 2 levels (e.g., ±1) on the stream. This allows us to encode 2, 3, 4, etc. bits per *symbol*. These symbols are then used to modulate the carrier, resulting in higher data rates (*bit rate*) for the same symbol-per-second (*baud rate*). For 2 bits per symbol, we can modulate a sine wave and a 90° offset (e.g., cosine) sine wave with each bit. These separately modulated signals are then summed together to produce *quadrature phase shift keying* (QPSK). The receiver is able to *demodulate* the signal and decipher the individual bits from the sine and cosine waves due to their orthogonality. Figure 2.21 depicts a QPSK transmitter.

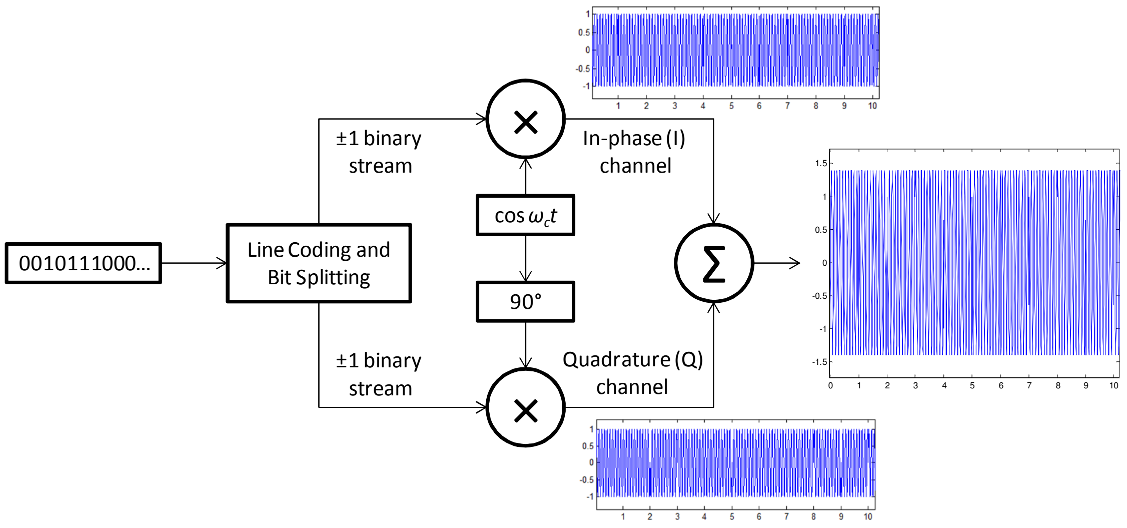


Figure 2.21: QPSK Modulator

2.11.3 Frequency Shift Keying

Frequency shift keying (FSK) uses a combination of two or more carrier frequencies to convey data. This is accomplished by modulating one frequency with one bit and another frequency with a different bit. The two different frequencies alternate use, which is dependent on the specific bit being transmitted. FSK with 1 bit per frequency has a fairly simple implementation, but utilizes 2 separate channels. Figure 2.22 depicts a 2-level FSK modulator.

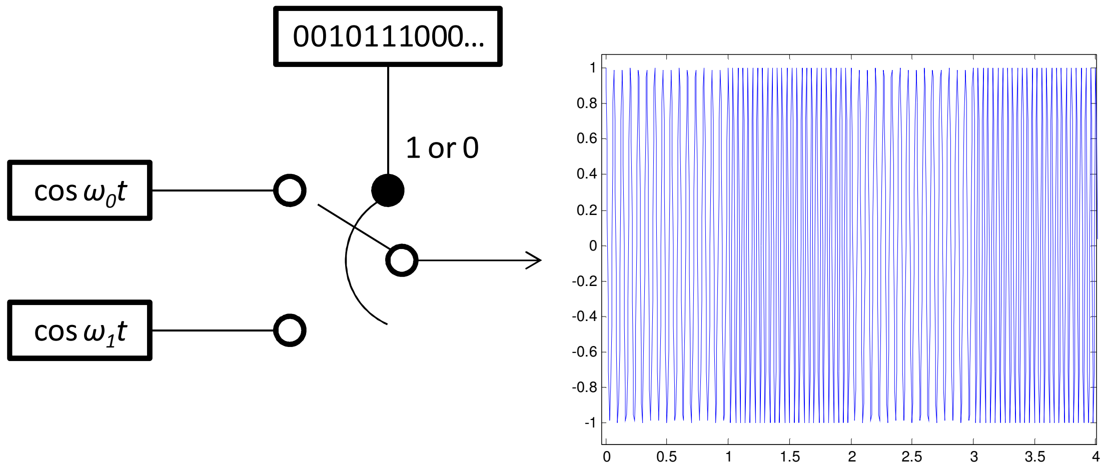


Figure 2.22: 2FSK Modulator

Much like *M*-ary PSK, we can encode multiple bits with varying frequencies. This allows us to put more bits in a single symbol, effectively increasing the data rate for the same baud rate. Figure 2.23 shows this type of modulation for 2 bits per symbol.

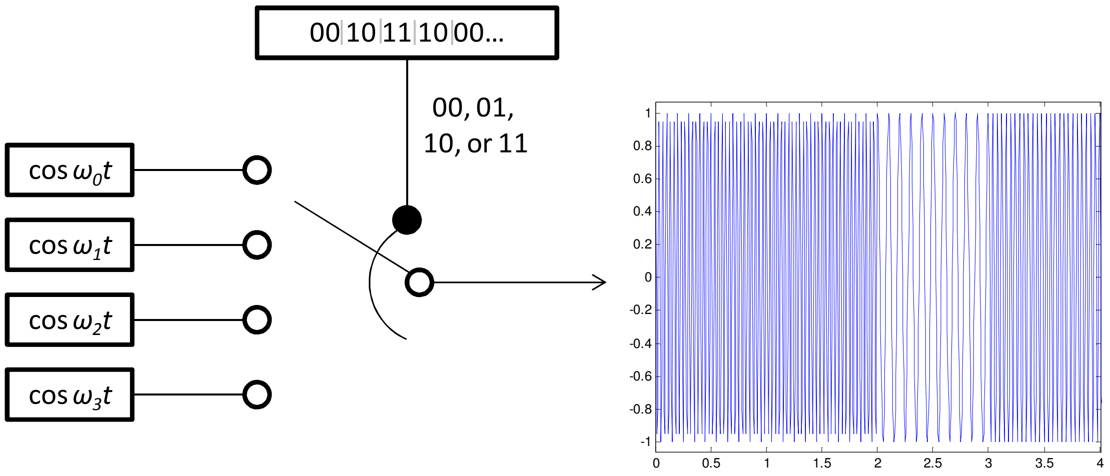


Figure 2.23: 4FSK Modulator

2.12 Spectral Correlation Function

The *spectral correlation function* (SCF) allows us to decompose a time-series of non-periodic data into its repeating, spectral components [33]. The specifics of the SCF lie outside the realm of this thesis. However, we will briefly cover the SCF in order to establish the extracted features from various modulated signals for the purposes of pattern recognition.

The use of SCF in communications is a growing area of research for pattern recognition in noisy environments [34]. The SCF allows for cyclostationary features to be extracted from a signal. In effect, we are able to view repeating patters in a signal while removing noise components. This permits us to identify signals even with noise and degradation.

Modulated signals often contain various repeating patterns, such as sine wave carriers, spreading, hopping, etc. [35]. The data is a *wide-sense stationary* process, while the modulated signals are considered *cyclostationary* [15]. The data often consists of random, yet, periodic elements that can be transformed into the frequency domain. Similar to autocorrelation, we can introduce the spectral correlation function:

$$S_x^\alpha(f) = \lim_{T\to\infty} \lim_{\Delta t\to\infty} \frac{1}{\Delta t}\int_{-\Delta t/2}^{\Delta t/2} \frac{1}{T} X_T(t, f+\alpha/2) X_T^*(t, f-\alpha/2)\, dt \quad (2.43)$$

where the Fourier transform is given by:

$$X_T(t,f) = \int_{t-T/2}^{t+T/2} x(u)e^{-j2\pi fu}du \tag{2.44}$$

Note that the SCF is a 2-dimensional transform, resulting in frequency (*f*) along one axis, the periodic frequencies (α) along another, and the complex spectral density on the third axis.

For feature extraction, we take the real part of the SCF of various signals and transform them into 2D images; each pixel contains a normalized value of the spectral density. Notice that "spikes" occur at areas where the periodic frequencies match the modulation frequency or frequencies.

Figure 2.24 shows the real portion of the SCF of a baseband signal with 1 bit per second, and additive, white, Gaussian noise (AWGN) included, to producing 0dB E_b/N_0. The data is generated using a uniform random set of 1s and 0s. The image is created as a cropped section of the SCF to show only the important peaks. It is stored as a grayscale image for training or recognition. The same window is used for all subsequent SCF calculations.

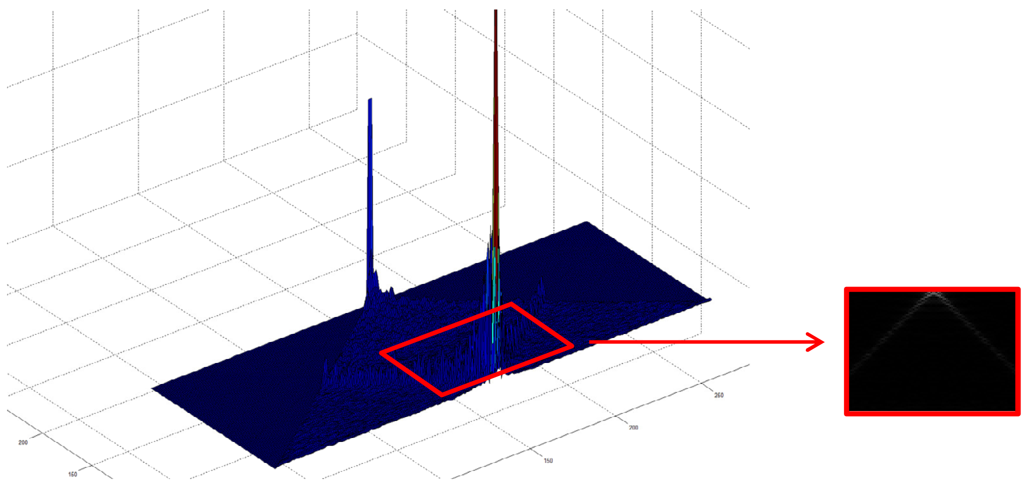


Figure 2.24: SFC of a Baseband Signal

Figure 2.25 depicts the SCF of a BPSK signal modulated at 10 Hz and a data rate of 1 bit per second. The same AGWN is used to produce 0dB E_b/N_0.

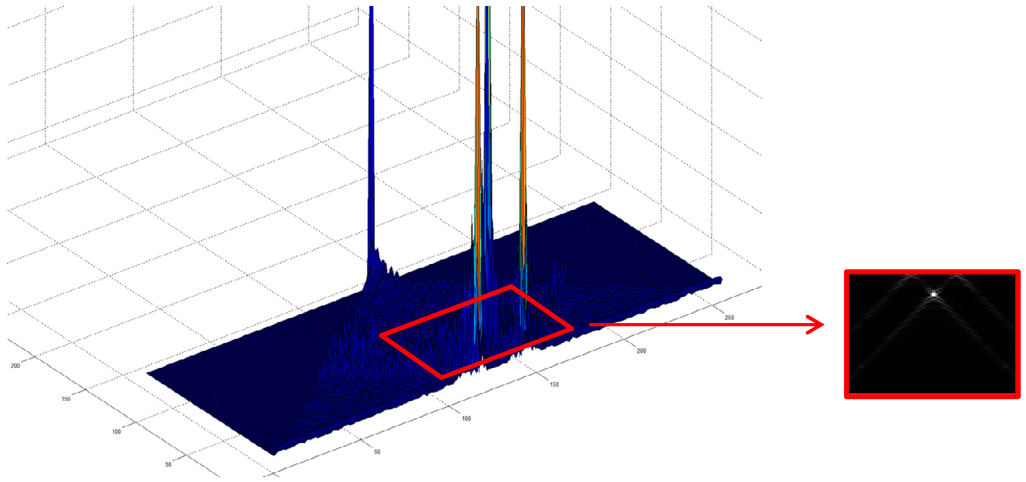


Figure 2.25: SCF of a BPSK Signal

Similarly, Figure 2.26 shows the SCF of a QPSK signal.

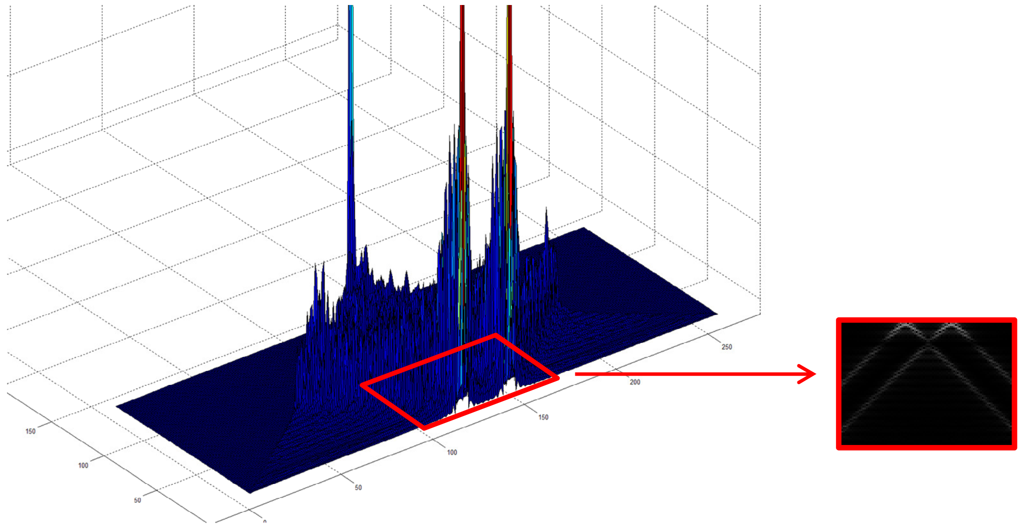


Figure 2.26: SCF of a QPSK Signal

We then create SCF images for 2FSK and 4FSK signals. For 2FSK (Figure 2.27), we use two different frequencies of 25 Hz and 15 Hz for 0 and 1, respectively. This is done at 1 bit per second with enough AWGN to produce 0dB E_b/N_0. 4FSK (Figure 2.28) utilizes the frequencies listed in Table 2.1 for each of the bit pairs given.

Table 2.1: 4FSK Frequency Mapping

Bits	**Frequency**
00	25 Hz
01	15 Hz
10	30 Hz
11	10 Hz

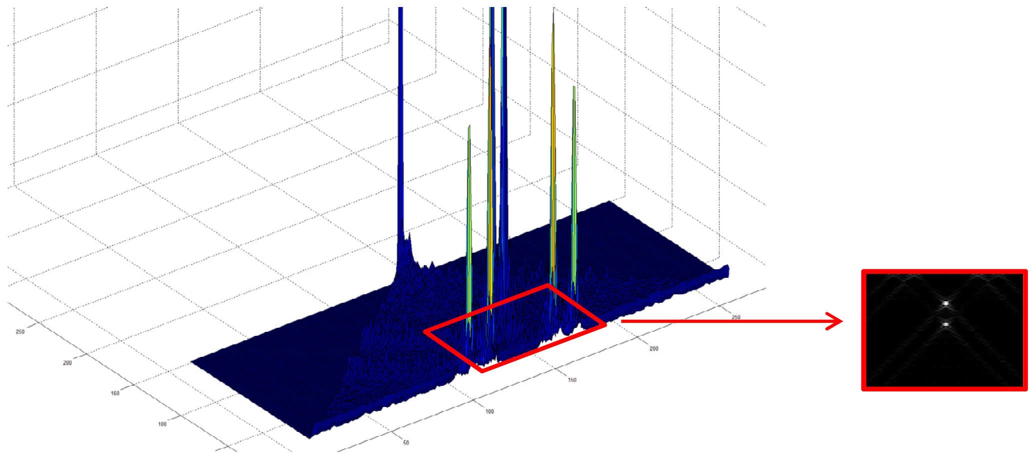


Figure 2.27: SCF of a 2FSK Signal

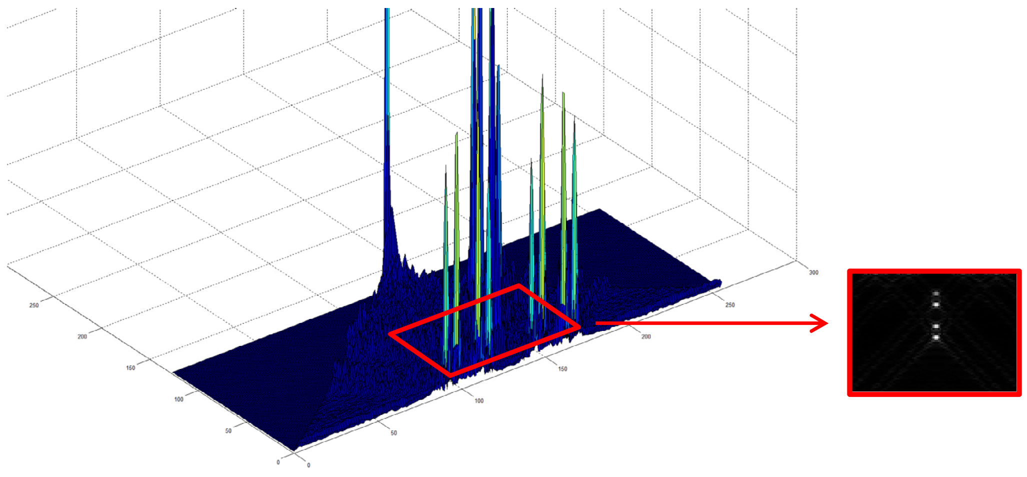


Figure 2.28: SCF of a 4FSK Signal

The images extracted from the SCF are used for both training and recognition purposes as described in section 5. For each training or test image, a different random set of bits are used. However, the other parameters (noise, carrier frequency, etc.) remain the same.

2.13 Summary

This chapter provided an in-depth look at discrete-time and discrete-valued Hidden Markov Models, building up from the notion of state transitions and culminating in a full-blown model. These models are used to describe patterns which exhibit transitions through different states. Three different algorithms were covered in an attempt to solve crucial problems involving HMMs. Finally, the notion of modulation and spectral correlation function were introduced for the purposes of pattern recognition in the wireless realm.

Chapter 3

Overview of GPGPU

With the advent of graphical user interfaces for personal computing in the 1990s, computer hardware companies began creating 2D image accelerators to assist with bitmap rendering [36]. Around the same time, 3D graphics began to become popular in defense, consumer electronics, and movie markets. These markets, as well as the introduction of 3D games such as Doom, Quake, and Duke Nukem, gave rise to the need for 3D graphics accelerators. This demand caused ATI, NVIDIA, and 3dfx Interactive to produce graphics cards for consumers and created an entire market for graphics hardware, known as the graphics processing unit (GPU).

In the early 2000s, GPUs utilized pixel shaders, which computed 3D effects on a per-pixel basis. Programmers discovered that through the use of graphics API such as OpenGL or DirectX, they could load the shaders with any data rather than just color information. For data-parallel tasks, GPUs could essentially be tricked into computing arbitrary data. This gave rise to informal General Purpose Computing on Graphics Processing Units (GPGPU).

Eventually, companies and organizations got wind of the GPGPU phenomena and began formalizing languages and software development kits (SDKs) along with modifications to GPU hardware to allow for general data computations. AMD/ATI released a language called "Stream," which is a modification of ANSI C. NVIDIA promoted its own language known as "Compute Unified Device Architecture" (CUDA) for its own graphics cards. Apple also developed a unified language, called "OpenCL", which is based on C99. “OpenCL” allows the creation of *kernels* for cross-platform GPUs. The OpenCL standard was eventually given to the Khronos Group to maintain, and they continue to use it under a royalty-free license. While other languages exist, Stream, CUDA, and OpenCL are three of the most popular languages for GPGPU.

This thesis focuses exclusively on CUDA. CUDA has the greatest support, and allowed for an easy implementation of the required algorithms. It is possible to port the CUDA program to Stream in order to run on ATI cards or OpenCL for cross-platform variability.

3.1 The Graphics Processing Unit

The *Central Processing Unit* (CPU) still dominates the market as the primary processing core in a computer. Up until recently, CPUs used a single core to perform calculations. A very basic CPU core is given in Figure 3.1.

It is important to note that the CPU contains a single math unit to perform calculations. Therefore, at most, it can perform one calculation per clock cycle. Over the past few years, companies have taken great strides toward increasing the clock speed and to allow for more calculations and instruction executions per second. These steps include heavy pipelining, code prediction, and larger data buses. Unfortunately, current technology, mainly transistor switching speeds and thermodynamic dissipation issues, limits the maximum clock rate. As a result, many manufacturers have been turning toward using multiple cores to allow for concurrent execution of programs, threads, etc. While this does not increase the speed of a program, it does allow for programs to run at the same time, or allows several tasks within a program to be executed simultaneously.

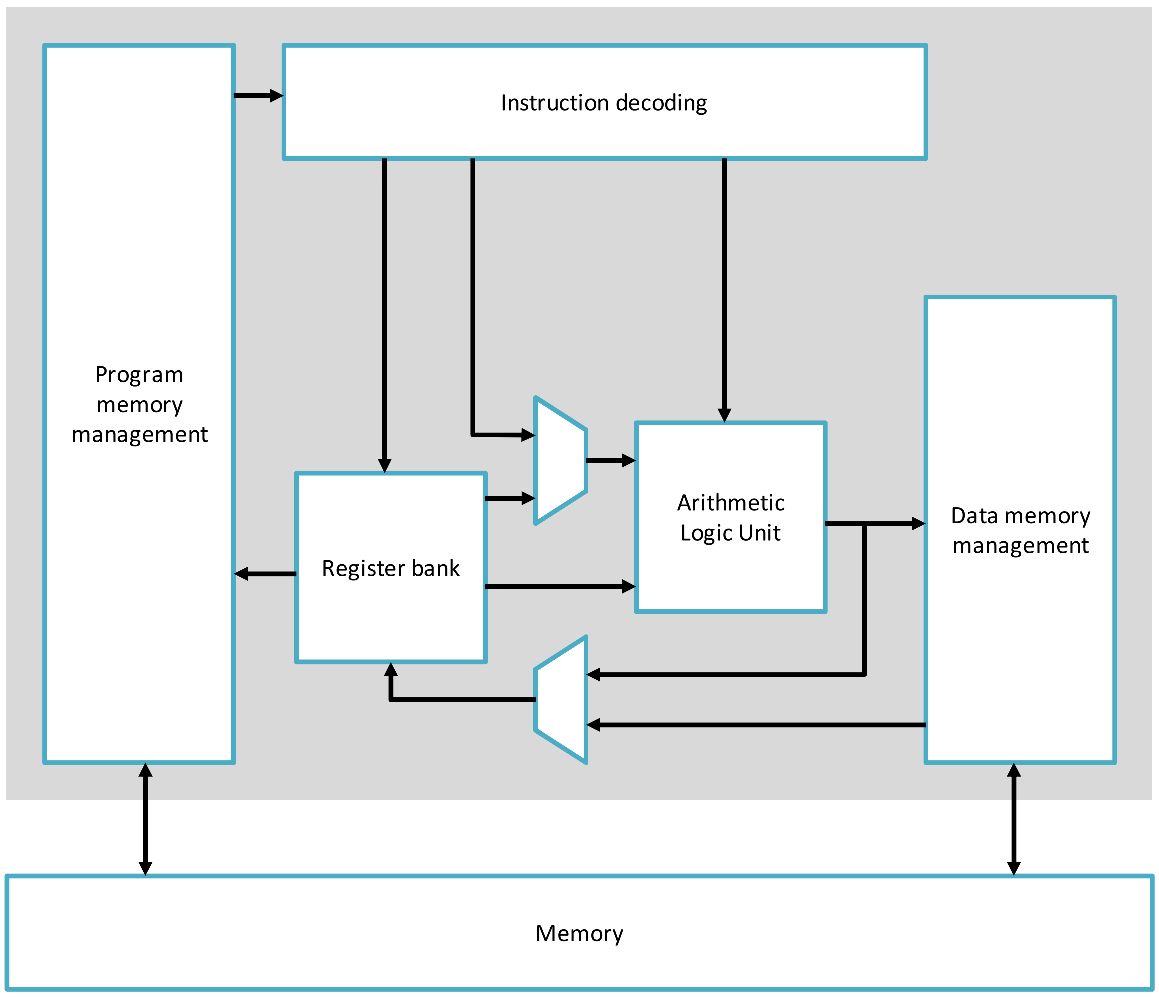


Figure 3.1: A Simple CPU Architecture

GPUs, on the other hand, focus on a *single instruction, multiple data* (SIMD) architecture. This involves loading up several Arithmetic Logic Units (ALUs) with different data and then performing one instruction on each set of data. These types of calculations are highly conducive to vector and matrix math, and are exemplified in bitmap and 3D graphics rendering. Figure 3.2 shows an example of a GPU architecture.

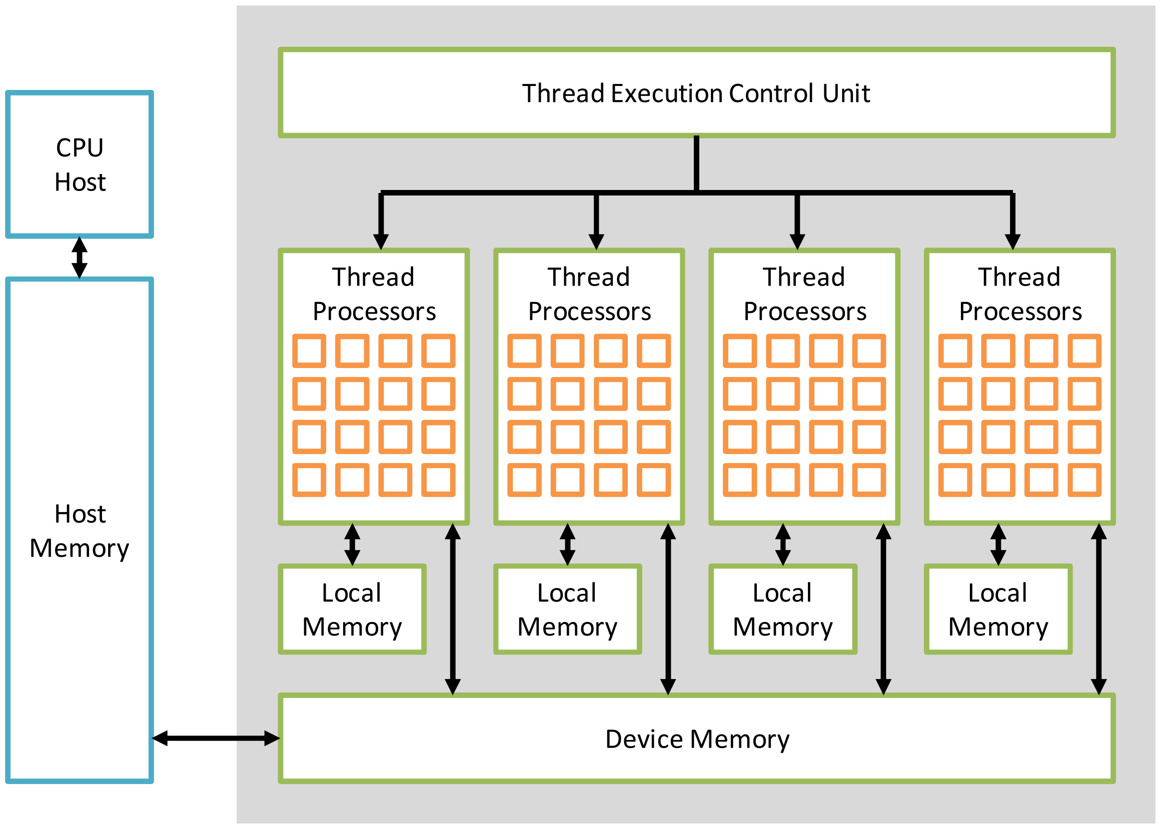


Figure 3.2: A Simple GPU Architecture

In order to use the GPU for general purpose computing, we must write a program that resides in the Host memory, which is often the Random Access Memory (RAM) of a computer. The specific calls to a GPU are loaded into the device memory for execution. Because the GPU is an accessory to the computer, exact control lies with the CPU. Therefore, the CPU must initiate the execution of GPU code. Additionally, any data to be loaded on to the GPU must come from the Host-side memory. As a result, GPUs can offer tremendous computing power, but often suffer data throughput problems as the memory transfer can be a bottleneck.

As shown in Figure 3.2, the individual processing units are known as *thread processors* and are often grouped together into *blocks*. Additionally, threads will generally have access to a small number of local registers, blocks will have access to a block-wide local memory, and all blocks in the device will have access to a larger, but slower, device memory. Initially, the device (GPU) data and program must be provided by the CPU. Consequently, the GPU is a massive co-processor for the CPU.

3.2 CUDA

3.2.1 Hardware Architecture

In 2006, with the release of its GeForce 3 architecture, NVIDIA modified their line of GPU hardware to allow for general purpose computing. This modification was accomplished by including a *unified shader pipeline*, which allowed for the individual processing units on a GPU to be used for both graphics calculations and general purpose computing. Each of the ALUs was constructed to specifically handle single-precision, floating point math and the thread processors were changed to allow for arbitrary reads and writes to memory. While NVIDIA was the first to produce consumer-level GPGPU-ready graphics cards, AMD/ATI quickly followed suit to jump into the GPGPU market [33].

3.2.2 Software Components

In addition to changing the hardware, NVIDIA also introduced CUDA. This software framework provided several additional functions and keywords to C for simple integration and a low learning curve. Furthermore, CUDA was provided with its own compiler, `nvcc`. The compiler is configured to handle both C/C++ Host code and GPU device code in the form of kernels or specific device calls from the Host.

In order to understand the specifics of the CUDA framework, we must introduce several new concepts. Within a graphics *device*, there are one or more *grids*, which are responsible for executing a *kernel*. Grids can be considered the GPU analog to a single processor or CPU, as they encompass all the necessary features to ensure proper execution of GPU code. This code is known as a kernel, which is simply a set of instructions for the GPU to execute. Diving deeper, we notice that a grid contains several *blocks*. Each block is responsible for ensuring execution among several *threads*. Each thread is a single processor capable of executing a set of instructions in parallel with other threads. Much like threads in a program being run on an operating system, GPU threads are specifically tied to a single, linear program. Additionally, threads are allowed to execute concurrently. However, threads on a GPU are given their own processing unit, and can thus be guaranteed true concurrency at the block level. Figure 3.3 [37] depicts the multiple software architecture layers within a GPU.

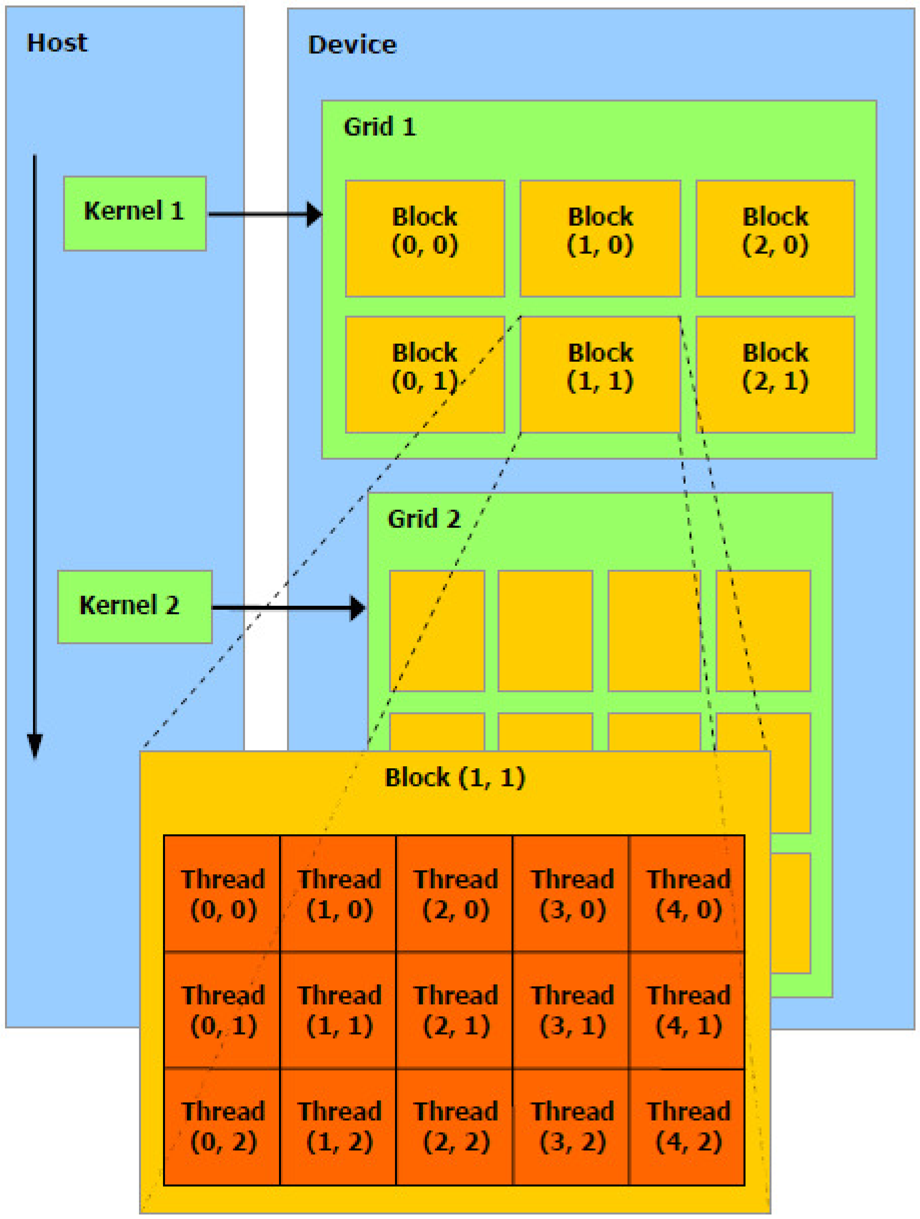


Figure 3.3: A CUDA Software Framework

Threads within a block can be guaranteed a level of concurrent execution. Those threads are able to communicate with each other through *shared memory* and the use of specialized calls, such as `__syncthreads()`. However, threads in different blocks have no real way of communicating, except through the use of *atomic locks* in global memory. As a result, a large amount of effort is required to provide inter-block communication and thread-safe data operations on global memory.

CUDA-enabled GPUs are designed to load one or more blocks into execution at a time. While threads in a block are guaranteed some level of concurrency, multiple blocks are not. If the hardware becomes saturated with too many blocks during execution, other blocks must wait their turn. The GPU hardware will then load the other blocks into memory for execution. This often creates the problem of losing some concurrent execution with very large datasets. However, as the hardware matures, GPU manufacturers not only increase the speed of chips but the number of concurrent executions as well.

3.2.2 Memory Management

The other major problem with GPU programming is memory management. Figure 3.4 [37] outlines the basics of GPU memory layout with respect to the software framework. Each thread has access to its own set of personal *registers*. These registers are analogous to the registers found in CPUs, which offer a small amount of capacity but extremely fast read and write operations. Each thread also has access to *local memory*, which is used to store variables unique to the thread. Within each block, a set of *shared memory* can be found, which offers slower read and write speeds, but allows for sharing of data between threads in a block. The slowest memory in a GPU is the *global memory*. It resides off-chip (similar to most RAM), and it requires long access times and can cause wait periods among threads trying to concurrently access the same data. However, it is by far the largest memory available and allows for complete sharing of data among all blocks and threads.

In addition to the global memory, GPUs also have specialized memory known as *constant memory* and *texture memory*. Only the Host can write to constant memory, and the Host offers extremely fast read-only access to threads. However, constant memory only offers benefits when all threads are reading from the same location. Like constant memory, texture memory is read-only from threads. Texture memory offers an additional memory path separate from constant memory and allows for quick reads from neighboring memory locations. For simplicity, we will not focus on constant or texture memory. However, it should be noted that many speed improvements can be made by correctly utilizing the GPU's specialized read-only memory.

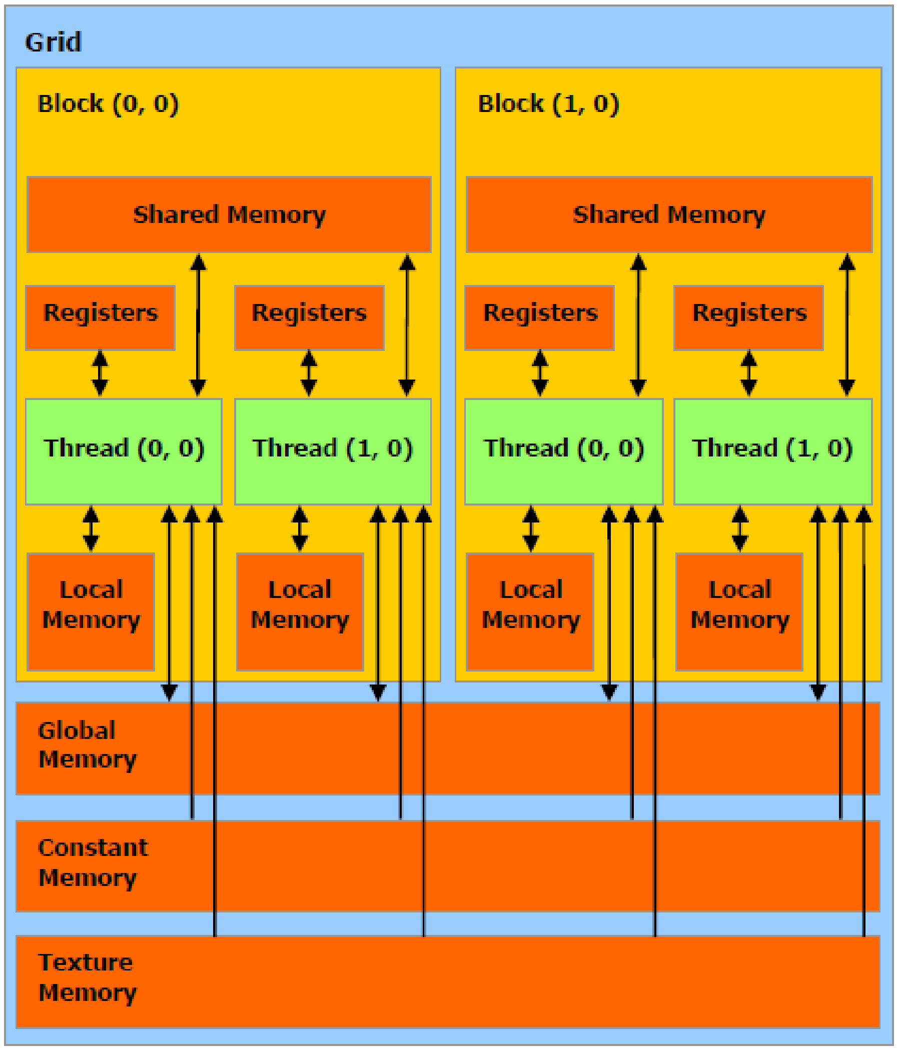


Figure 3.4: CUDA Memory Model

3.2.3 Program Flow

The basic code flow for a CUDA application involves 4 steps, once the compiled program has been loaded onto the GPU and CPU memory:

1. Copy the data to be processed from the Host's memory to the GPU's memory (often the global memory). This step is often a bottleneck for CUDA-enhanced applications.
2. The CPU must then instruct the GPU to execute the compiled program. Notice that the GPU is dependent on the Host for instructions.
3. The GPU executes the desired program. Each thread executes the same set of instructions in parallel. The results of the program are stored in the GPU's memory (e.g. global memory).
4. The results are copied back from the GPU's memory to the Host's memory. Similar to the first copy operation, this is often a bottleneck for applications.

Figure 3.5 depicts this basic code flow.

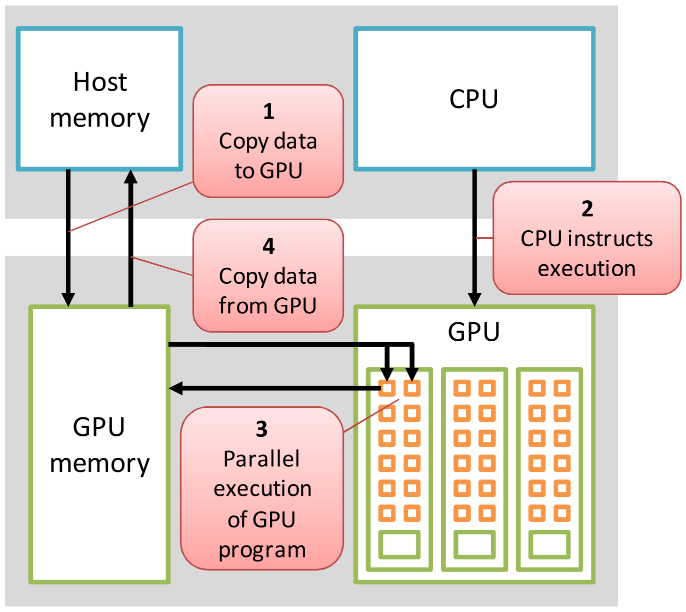


Figure 3.5: Basic CUDA Process Flow

3.3 Parallelization Techniques

3.3.1 Element-by-element Operations

GPUs excel at performing operations on data that remains separate in memory. This is exemplified through element-by-element matrix operations. For example, adding two arrays together, as shown in Figure 3.6.

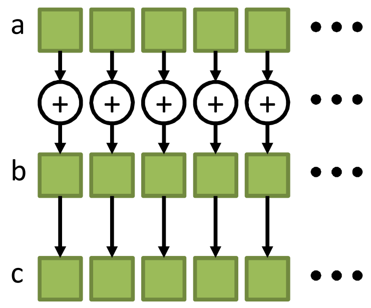


Figure 3.6: Summing Two Arrays

With serial execution, element-by-element matrix operations are accomplished by performing a for loop *N* times in which an element from each array is read, added, and stored into another array. The pseudo-code in Figure 3.7 depicts an example of element-by-element array addition.

```
for (i = 0; i < N; i++) {
     c[i] = a[i] + b[i];
}
```

Figure 3.7: Serial Implementation of Element-by-Element Addition

To implement this type of operation in CUDA, we need to write a kernel that handles element-by-element addition. Instead of each step in a ‘for loop’ indexing into the arrays, each thread in the GPU provides the index into the arrays, as shown in Figure 3.8. As can be seen, we use an ‘if statement’ instead of a ‘for loop.’ This forces any threads less than *N* to perform the array index read, execution, and write. As a result, a *O*(*N*) matrix operation becomes *O*(1). However, this does not take into account any of the memory reads or writes that need to be performed prior to this operation.

```
__global__ void add( int *a, int *b, int *c ){
     int i = threadIdx.x;
     if (i < N) {
          c[i] = a[i] + b[i];
     }
}
```

Figure 3.8: CUDA Implementation of Array Addition

3.3.2 Parallel Reductions

Unlike simple element-by-element math, some operations require data to be written back to the array before being read again, such as calculating the sum of an array. Array summation is shown in Figure 3.9.

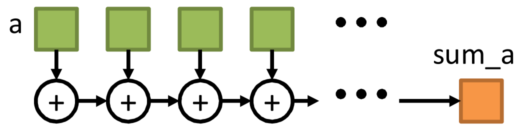


Figure 3.9: Array Summation

Much like the element-by-element serial implementation, this method is accomplished using a simple for loop, as depicted in Figure 3.10.

```
sum_a = 0;
for (i = 0; i < N; i++) {
     sum_a += a[i];
}
```

Figure 3.10: Serial Implementation of Array Summation

Because reading from, and writing to, an array element in the same clock cycle would cause major conflicts among threads, we must find a fast, yet thread-safe, solution to compute array-wide operations. A popular technique among SIMD-enabled programs is known as a *reduction*. Reduction involves summing only two elements at a time in each thread, and each thread uses two different elements in the array. On the next cycle, another set of pairs are summed, and so on until only one number remains in the first element of the array. Figure 3.11 shows this process.

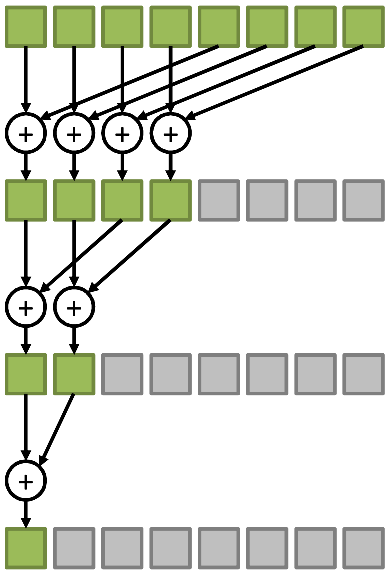


Figure 3.11: Example of Reduction

In CUDA, the reduction technique is accomplished at the block level, and steps in the reduction are synchronized using the `__syncthreads()` function. Notice that the step size between elements for the addition is *N*/2, which allows for a slightly increased performance over using *i*+1. This is due to memory access methods within the GPU [38]. Figure 3.12 gives an example CUDA implementation of a parallel reduction. Note that if the number of reduction operations in a single step exceeds the maximum allowable threads per block, then multiple kernel calls are necessary to permit global synchronization.

```
__global__ void sum( int *a, int sum_a) {
     int i = threadIdx.x;
     int stride;
     __syncthreads();
     for (stride = blockDim.x/2; stride > 0; stride /= 2) {
          if (i < stride) {
               a[i] += a[i + stride];
          }
          __syncthreads();
     }
     if (i == 0) {
          sum_a = a[0];
     }
}
```

Figure 3.10: CUDA Implementation of Array Summation

As is evident, the steps for a parallel reduction are considerably more complicated than the simple serial implementation. Also notice that the data in the original array is destroyed in order to obtain the summed value and it is advisable to copy the data to allocated shared memory prior to performing the calculation. However, a *O*(*N*) array-wide operation (e.g. summation, multiplication, etc.) is converted to a *O*(log *N*) operation.

3.3.3 Memory Constraints

As mentioned in 3.2.2, GPUs rely on several different types of memory to perform their calculations [39]. The biggest, most versatile type of memory is global memory. The CPU has the ability to copy memory directly from Host memory to the GPU's global memory. All threads in all blocks in a single grid have access to global memory. This allows for global variables, shared memory, some synchronization, etc. However, global memory is by far the slowest of all possible memory in the GPU.

Shared memory resides in the block level, meaning that one block does not have access to another block's shared memory. Threads within a block are able to communicate and share memory using this particular type of memory. Shared memory has a much faster (single cycle) access time.

Finally, local registers can be used to store simple variables. These have the same, single cycle access time as shared memory but do not require pre-allocation.

3.4 Summary

In this chapter, we looked at the basics of general purpose computing on GPUs and the specific implementation issues of NVIDIA's CUDA framework. We pointed out that GPUs are designed to handle data-intensive tasks, but can suffer from bottleneck issues due to memory transfers to and from the Host. Finally, we covered the basics of parallelization in preparation for full algorithm parallelization.

Chapter

4

Algorithm Parallelization

In this chapter we look at the various HMM algorithms and create a parallel implementation for the purposes of timing and other experiments. We look at each algorithm, break down its parts, and give a comparison of the computational complexity. The CUDA programs for each algorithm can be found in the appendices.

4.1 Previous Work

There has been much research accomplished in the realm of HMMs in order to achieve speed increases. This is due to the popularity of HMMs as a modeling tool and pattern classification. As a result, other researchers have experimented with CUDA implementations of HMMs.

Jun Li et al. look at the ability to perform the Forward-Backward Algorithm (a combination of calculating α and β variables) with several models and a single observation sequence [40]. This is akin to the pattern classification problem discussed in section 2.10. Using a CUDA implementation with 60 HMMs, 8 states, 8 symbols, and 200 observations, they found a speed increase of 3.5× over the serial implementation.

Chuan Liu, on the other hand, experimented with a single model and multiple observation sequences [41]. Using 512 states, 3 observations, and 512 sequences of 10 observations each, he achieved an 880× speedup of the Forward Algorithm and 180× speedup of the Baum-Welch Algorithm.

Zhang et al. also performed several tests with HMMs on the GPU. He implemented the Viterbi algorithm in CUDA and saw a speedup of about 3× over the serial implementation [42].

Each of these works has proven the success of utilizing GPU to perform HMM algorithms. However, we must look at all three algorithms, both individually as well as working together as part of a pattern recognition system.

4.2 Parallel Forward Algorithm

We first examine the Forward Algorithm (FO), which is given by equations (2.10) - (2.12). In the initialization step, we perform an element-by-element multiplication of $\pi_i b_i(O_1)$, where b_j is indexed by the first observation (assuming the observations have been set to the integers 0,1,2,...). The probability of occurrence for the observation for each state is multiplied by its corresponding initial probability π_i. In a serial operation, this is a for loop over all state, resulting in a $O(N)$. However, we can use CUDA to create an element-by-element operation that involves a $O(1)$ process. This step calculates the necessary $\boldsymbol{\alpha}_1$ values for all states, where $\boldsymbol{\alpha}_t$ is a vector of the $\alpha_t(i)$ values for each state i.

$$\boldsymbol{\alpha}_t = [\alpha_t(1), \alpha_t(2), \alpha_t(2), \dots] \tag{4.1}$$

For the induction step (2.11), we essentially perform a matrix-by-vector multiplication of $\boldsymbol{A}^T\boldsymbol{\alpha}_t$ followed by an element-by-element operation of that result multiplied by $\boldsymbol{b}(O_{t+1})$, where $\boldsymbol{b}(O_t)$ is a vector of the output probabilities of the observation symbol found at t for each state j.

$$\boldsymbol{b}(O_t) = [b_1(O_t), b_2(O_t), b_3(O_t), \dots] \tag{4.2}$$

In the serial implementation, the entire induction step is accomplished by a thrice-embedded 'for loop.' The outside loop runs over all T (for the recursion), and the inside two loops each run for N, in order to perform

$$\boldsymbol{\alpha}_{t+1} = (\boldsymbol{A}^T\boldsymbol{\alpha}_t).*\, \boldsymbol{b}(O_t) \tag{4.3}$$

where ".*" is the MATLAB-style notation for element-by-element multiplication. This gives a complexity of $O(TN^2)$ for the serial implementation.

For the CUDA implementation, we introduce the CUBLAS library for linear algebra. Its predecessor, Basic Linear Algebra Subprograms (BLAS), is a library for matrix math optimized for several different processors and languages. NVIDIA offers a CUDA-enabled version of BLAS, known as CUBLAS, that implements many of the BLAS functions in CUDA and is affectionately known as CUBLAS. Using CUBLAS, we can wrap many of the matrix operations into a single function call, and rely on the library to perform the necessary operations.

Therefore, we can use a CUBLAS call to calculate $\boldsymbol{A}^T\boldsymbol{\alpha}_t$. This result is then used in a CUDA kernel function to perform the element-by-element multiplication with $\boldsymbol{b}(O_t)$. Because the algorithm is recursive, and we must use the previous result from t to calculate $t + 1$, we cannot perform all calculations in parallel. Therefore, we need to have all CUDA calls encapsulated in a 'for loop' over all T. However, within each recursive step t, we can reduce the $\boldsymbol{A}^T\boldsymbol{\alpha}_t$ calculation from $O(N^2)$ to $O(\log N)$, and we can reduce the element-by-element operation from $O(N)$ to $O(1)$. If we include the for loop for all T, we end up with a total computational complexity of $O(T \log N)$ for the induction step.

It is important to note that we use the scaling value mentioned in section 2.6 to prevent underflow. This involves normalizing the $\boldsymbol{\alpha}_t$ vector at each step by summing all the values and dividing each element by that sum. This is accomplished after (4.3) has been calculated and introduces a $O(N)$ calculation for the serial implementation and a $O(\log N)$ calculation for CUDA. However, this does not change the final complexity of $O(TN^2)$ for serial and $O(T \log N)$ for parallel.

Finally, we must calculate the $\Pr(O|\ \lambda)$ by summing the final $\boldsymbol{\alpha}_T$ vector. For the serial implementation, we keep a log of all the scaling values and simply sum the $\log_{10}$ values over all T. In CUDA, we keep a running sum of the $\log_{10}$ of the scaling factors for each recursive step. This introduces a $O(T)$ calculation in the serial implementation and a $O(1)$ calculation in the CUDA version.

In conclusion, we have found that the $O(TN^2)$ for the serial implementation of the FO can be reduced to $O(T \log N)$ in CUDA. Some optimizations have been made through the use of library calls, but ideally both implementations could use further experimentation in memory accesses and math optimizations to reduce the execution time. The C implementation is assumed to be fairly straightforward; however, the CUDA implementation is given in Appendix A for the reader.

4.3 Parallel Viterbi Algorithm

The Viterbi Algorithm (VIT), given by equations (2.17) - (2.21), proves to be a bit more complicated than the FO for both serial and parallel implementations. It is also recursive and therefore cannot be parallelized in the induction step. However, the initialization is performed in the exact same manner as the FO by calculating

$$\boldsymbol{\delta}_1 = \boldsymbol{\pi}.*\, \boldsymbol{b}(O_1) \tag{4.4}$$

where $\boldsymbol{\delta}_t$ is the set of partial probabilities at time *t* that is the most probable path to state *i* given by

$$\boldsymbol{\delta}_t = [\delta_t(1), \delta_t(2), \delta_t(3), \dots] \tag{4.5}$$

Similar to the FO, this is accomplished by $O(N)$ operations serially and $O(1)$ operations in parallel.

For the recursive steps, we must find the maximum value and argument maximum of the state number *j* for (2.18) and (2.19). This is accomplished by first finding the inner product of $\delta_t(j)a_{ij}$ before finding the *max* and *argmax* of the result for *j*, for each *i*. Note that the results of the *argmax* operation become the $\Psi_t(i)$ elements. Serially, this is a two-step $O(N^2)$ process of calculating the inner product before performing a $O(N)$ search for the maximum value and argument. In the same $O(N^2)$ process, we can compute the next set of δ values by computing (2.18) using an element-by-element operation of multiplying the results of the *max* operation by the elements in the corresponding $\boldsymbol{b}(O_{t+1})$ vector. In total, the serial implementation of the recursive steps require $O(TN^2)$ operations.

As with the FO, we must perform some additional scaling of the δ variables to prevent numerical underflow. This scaling is accomplished by normalizing the $\boldsymbol{\delta}_t$ vector at each *t*, and introduces some extra steps in the implementation. However, it does not affect the overall computational complexity.

For CUDA, the element-by-element operations can be reduced to $O(1)$, as described earlier. However, the *max* and *argmax* functions must be accomplished using parallel reductions. The implementation for each is the same as described in 3.3.2, but instead of an addition for each element, a simple greater-than comparison is used. The larger value is kept in the first element until the largest value of the array remains in the first element of the array. During this calculation, the index to the largest element is also stored in a separate variable in order to solve the *argmax* problem. A custom subroutine was created for the *max* and *argmax* implementation, and multiple kernel calls were used to allow for more than one block in the reduction. However, this routine is not completely optimized, as memory accesses and multiple kernel calls add significant time to the algorithm. As a result, more experimentation is required to further optimize this part of the parallel Viterbi algorithm. This implementation allows the complexity to become $O(T \log N)$.

The serial implementation of the termination step, (2.20), requires a $O(N)$ for loop in order to find the index of the largest element in $\boldsymbol{\delta}_T$. The parallel implementation, on the other hand, utilizes a $O(\log N)$ CUBLAS call to find the index of the largest element.

The final step in the VIT, (2.21), requires backtracking through the most probable state sequences and storing each most likely state in an array, q_t^*. Both serial and parallel implementations are exactly the same, as the path is computed recursively via a for loop of $O(T)$ complexity.

In conclusion, the Viterbi algorithm can be reduced from $O(TN^2)$ serially to $O(T \log N)$ in parallel. The CUDA program for VIT is given in Appendix B.

4.4 Parallel Baum-Welch Algorithm

The Baum-Welch Algorithm is given by equations (2.31) - (2.40) and attempts to re-estimate the set of HMM parameters to better fit a given observation sequence. The forward variables (α) and the backward variables (β) are calculated in the exact same manner as the Forward Algorithm for both serial and parallel implementations.

The γ variables are stored as a running sum over all T, as given by equation (2.36). Serially, this is accomplished by a doubly-embedded for loop of $O(TN)$. As this is an element-by-element operation, this procedure can be accomplished using a $O(T)$ procedure in CUDA.

Similarly, the ξ variables are calculated as a running sum. However, at each step t we must compute a 2D matrix of $\xi_t(i, j)$. As a result, the serial implementation requires $O(TN^2)$ operations. The numerator of (2.37) can be found using an element-by-element operation of $O(1)$ complexity in CUDA. For the denominator at each instance t, $\Pr(O| \lambda)$ must be calculated up to that instance t. Therefore, we must calculate

$$\Pr_t(O|\lambda) = \sum_{i=1}^{N} \alpha_t(i)\beta_t(i) \tag{4.6}$$

which is a $O(\log N)$ reduction in CUDA. Specifically, this is calculated using the CUBLAS `cublasSdot()` function. As a result, the ξ variables require $O(T \log N)$ calculations in CUDA.

Because we have the ξ and γ variables summed across all T, we can use those values to compute the re-estimates of **A**. This involves element-by-element computations for each element of **A,** along with the required scaling operation to ensure that **A** is stochastic. With N states, this piece of the algorithm requires $O(N^2)$ operations for the serial implementation. For the parallel implementation, the initial estimates are computed in a $O(1)$ operation, but the summing for the scaling value requires a $O(\log N)$ operation.

Each element in **B** is calculated by summing the α and β values for each state and symbol, divided by $\Pr_t(O|\lambda)$ at t. These sums are then divided by the summed γ value at each state. Finally, the **B** elements are normalized to produce proper pmf's for each state as well as set any 0 values to a very small number. These steps are taken to prevent 0s from appearing in the re-estimation of **B,** if a symbol appears that originally had zero probability of appearing in that state according to the initial **B**. For the serial implementation, this re-estimation is accomplished with $O(TMN)$ operations. In parallel, this requires $O(T \log N) + O(\log M)$ operations, as shown in the CUDA implementation. Assuming $T \log N > \log M$ for simplicity, we conclude that re-estimating **B** requires $O(T \log N)$ operations.

Finally, re-estimating $\boldsymbol{\pi}$ involves normalizing the element-by-element multiplication of $\alpha_1(i)$ and $\beta_1(i)$ at time t = 1 for each new π_i. For the serial implementation this requires two $O(N)$ operations. For the parallel implementation, this requires a $O(\log N)$ operation, using the CUBLAS library, to calculate the scaling factor, and $O(1)$ to perform the element-by-element operation of multiplying $\alpha_1(i)$ and $\beta_1(i)$ and dividing that result by the scaling factor.

In essence, the complexity of the serial implementation of the BWA can be expressed as

$$\begin{cases} O(TMN), \; M > N \\ O(TN^2), \; else \end{cases} \tag{4.7}$$

Additionally, the parallel implementation can by expressed as $O(T \log N)$, assuming that $T\ log\ N > log\ M$, which should almost always be the case except for extremely large values of M and very small sequence lengths, T. The CUDA implementation of the BWA can be found in Appendix C.

4.5 Parallel Pattern Recognition Engine

For the purpose of this paper, we will assume that the HMMs have been trained as per section 2.10 and stored in a database. In this section, we propose a parallel implementation of the pattern recognition engine given in Figure 2.18 to allow for faster, near-real-time pattern matching. Instead of computing a single $\Pr(O| \lambda)$ which has been parallelized across all states, we want to compute all $\Pr(O| \lambda_k)$ for a single observation sequence.

In order to accomplish this type of parallelization, we must load all models into the GPU memory and perform the necessary calculations without having to loop through the models. This is accomplished by creating an additional dimension in each **A**, **B**, and $\boldsymbol{\pi}$ matrix. The extra dimension can then be indexed to determine the model k being examined.

Additionally, we want to run the entire FO in a single kernel call in order to approximate the use of the GPU as a co-processor in a pattern recognition situation. Ideally, the GPU would be loaded with the observed data and told to perform the necessary calculations while the CPU (or other processor) would be collecting the next set of data. This can be accomplished using multiple threads and a double buffer.

For these experiments, the FO is only run once on a single set of observations in order to better understand the execution time of serial versus parallel implementations. Obviously the serial implementation introduces another set of *K* computations, where *K* is the number of models in the database, which results in a total computational complexity of $O(TN^2K)$. However, if we are able to perform all the necessary computations in a single kernel call without having to loop over each model, we can see a complexity of $O(T \log N)$ for the parallelized Forward Algorithm and multiple models.

The parallel approach has several drawbacks, however. As per the implementation given in Appendix D, each $\Pr(O| \lambda_k)$ calculation is performed in a separate CUDA block (i.e. there are *K* blocks). As a result, the number of states in each model is limited by the number of threads per block, which is hardware dependent. Recent NVIDIA graphics cards can support 512 threads per block. In order to allow some backward compatibility, we limit the maximum threads per block to 256. Due to the requirement for N^2 calculations in CUDA, this means that the maximum states allowed for this implementation is $\sqrt{\mathrm{MAX\_THREADS\_PER\_BLOCK}}$, which is set to 16.

Because we mimic a near-real-time type of implementation and make a single kernel call, we cannot use the optimized CUBLAS library, which requires separate kernel launches for each function. As a result, many of the memory access and specific implementation details may not be fully optimized. Once again, this would require extra experimentation to reduce the time of the parallel FO.

4.6 Summary

We looked at the specifics of parallel implementation for each of the three algorithms, as given in table 4.1. This involved breaking down specific equations and noting where operations could be performed in parallel. Specifically, any element-by-element operation could be accomplished in a single $O(1)$ cycle and reductions could be accomplished in $O(\log N)$ cycles. However, recursive steps could not be parallelized, as the results of one step are required in the next step. Finally, we proposed a parallel pattern recognition engine based on the Forward Algorithm. This involved performing all of the necessary steps in the FO in a single CUDA block, and each block evaluated a model from a different source.

Table 4.1: Computational Complexity Comparison Chart

Algorithm	**Serial**	**Parallel**
Forward	$O(TN^2)$	$O(T \log N)$
Viterbi	$O(TN^2)$	$O(T \log N)$
Baum-Welch	$O(TN^2)$ *or* $O(TMN)$	$O(T \log N)$
FO with K HMMs	$O(TN^2K)$	$O(T \log N)$

Chapter 5

Simulation: Architecture and Methodology

This chapter looks at the methodology used to test the various algorithms. We cover the specific hardware used for testing as well as the parameters used to test the time and power consumption of the various algorithms.

5.1 Test Hardware

The timing and power tests were completed on an Alienware M11x R1 laptop running Linux Ubuntu 10.04 LTS. The specifications for the laptop (Figure 5.1) are given in Table 5.1.

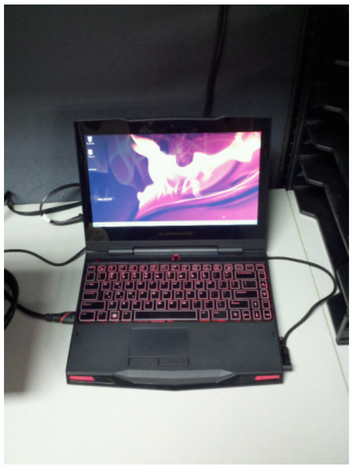


Figure 5.1: Alienware M11x

Table 5.1: Alienware M11x Specifications

Component	Specification
CPU	Intel Core 2 Duo U7300 @ 1.30 GHz
RAM	8GB DDR3 800MHz
GPU	NVIDIA GeForce GT 335M

Table 5.2 provides the specifics on the graphics processor in the M11x.

Table 5.2: NVIDIA GeForce GT 335M Specifications

Component	Specification
GPU Core Speed	450 MHz
GPU Shader Speed	1080 MHz
GPU Memory Speed	1066 MHz
GPU Memory	1024 MB GDDR3
CUDA Cores	72

In addition to the laptop, a custom-built desktop, shown in Figure 5.2, was utilized for timing experiments. Specifically, the desktop was used for the pattern recognition experiments. Table 5.3 gives the hardware for the desktop.

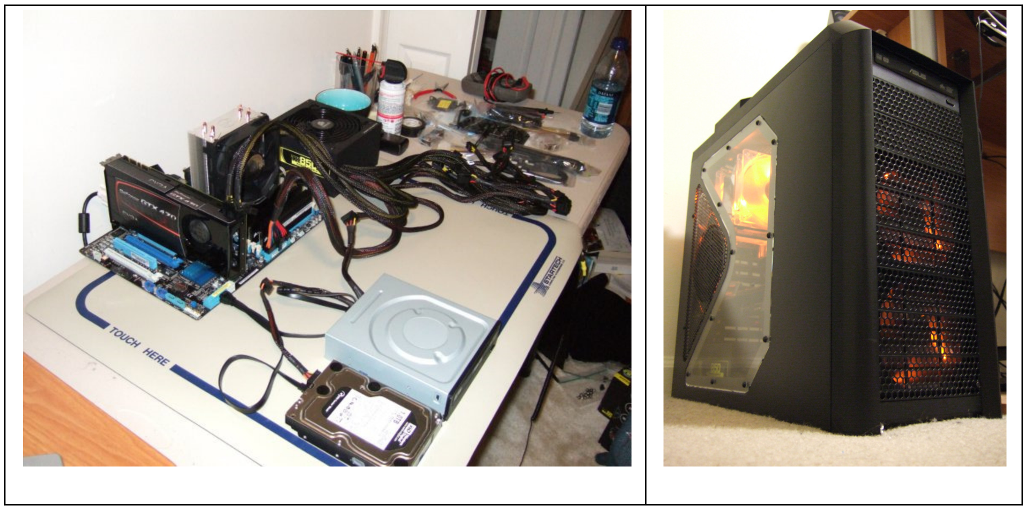


Figure 5.2: Custom Desktop Hardware

Table 5.3: Desktop Hardware Specifications

Component	Specification
Motherboard	ASUS P6X58D-E
CPU	Intel i7-930
RAM	6GB DDR3 1600MHz
GPU	NVIDIA GeForce GTX 470

Table 5.4 provides the specifics on the graphics card in the desktop. Note that the particular card used was factory overclocked by the manufacturer (EVGA).

Table 5.4: NVIDIA GeForce GTX 470 Specifications

Component	Specification
GPU Core Speed	625 MHz
GPU Shader Speed	1250 MHz
GPU Memory Speed	3402 MHz
GPU Memory	1280 MB GDDR5
CUDA Cores	448

5.2 Measurements

For the timing tests, each algorithm was run while varying one of the three main variables: number of states (N), number of symbols (M), and number of observations (T). While the number of states were varied, $M = 2$ and $T = 1000$. While varying the number of symbols, $N = 60$ and $T = 1000$. Finally, when differences in T was being observed, $N = 60$ and $M = 2$. These static numbers were chosen as the CPU and GPU were found to have similar timing results when the number of states was around 60, and the two observation symbols were chosen arbitrarily, as the number of symbols used were found to have little effect on system performance.

In each case, a throw-away set of HMM parameters were used. All of the elements in each matrix were set to the same value, creating a neutral HMM. While this type of HMM does little to describe a series of events (other than specifying that it is a uniform distribution over all symbol possibilities), the operations are still correctly run so that the program can be timed. The following gives an example of a 3-state, 2-symbol HMM used for timing:

$$\mathbf{A} = \begin{bmatrix} 0.33 & 0.33 & 0.33 \\ 0.33 & 0.33 & 0.33 \\ 0.33 & 0.33 & 0.33 \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & 0.5 \end{bmatrix}$$

$$\boldsymbol{\pi} = \begin{bmatrix} 0.33 & 0.33 & 0.33 \end{bmatrix}$$

The elements in **A** are each calculated as $1/N$, the elements in **B** are $1/M$, and the elements in $\boldsymbol{\pi}$ are $1/N$, which creates the desired stochastic, neutral matrices.

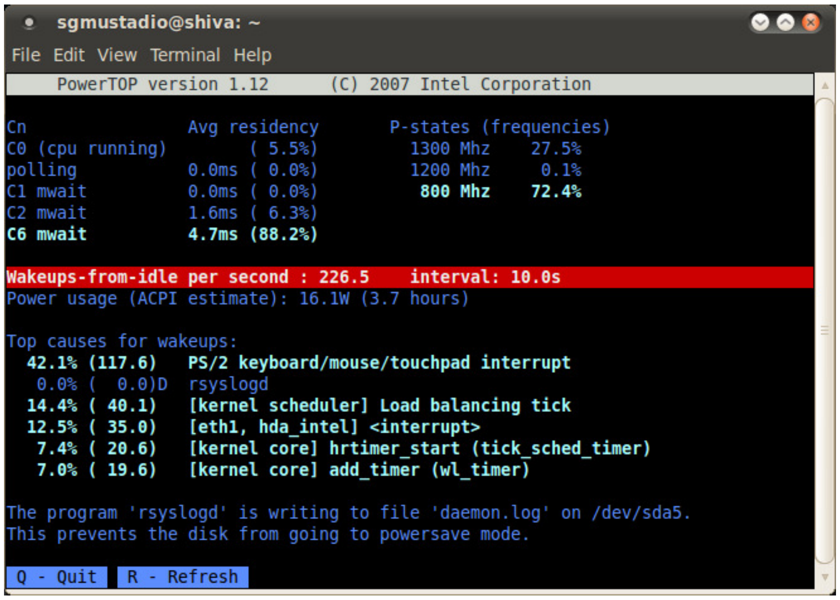
All observations are set to 0 in the sequence, in order to allow for a standard set of calculations among all tests. However, one of the set of tests varies the number of observations, T. As to be expected with complexities of $O(TN^2)$, the execution time should increase linearly as T increases linearly, for both serial and parallel implementations.

The time for each algorithm is calculated based on the start and finish of a single run of the algorithm. This assumes that all HMM parameters and observation sequences have been populated with the appropriate values. However, if memory needs to be allocated for intermediate steps, then those allocations are included in the timing. This is important to note because allocating memory, copying data to, and copying data from the GPU is included in the timing. Also, all timing information is based on the system clock and accessed via system calls.

The timing tests for the three algorithms were run on the M11x laptop.

For the pattern recognition engine, only the Forward Algorithm was used with 2 and 16 states. However, the number of models (k) was varied from 1 to 100. The timing tests for the pattern recognition were run on the desktop.

Average power consumption was calculated via the software PowerTOP version 1.12 [43]. This is a Linux command line tool that can be used to monitor the entire system power usage as shown in Figure 5.3. Each algorithm was run for 1 minute before recording the power measurement in PowerTOP (measured in Watts). This power was used to calculate the *total system energy* used to run each algorithm. PowerTOP does not distinguish between CPU usage, GPU usage, and other system components. All efforts were taken to ensure that the only difference between measurements was the type of algorithm run (CPU vs. GPU).



```
     PowerTOP version 1.12      (C) 2007 Intel Corporation

Cn                Avg residency       P-states (frequencies)
C0 (cpu running)        ( 5.5%)         1300 Mhz    27.5%
polling           0.0ms ( 0.0%)         1200 Mhz     0.1%
C1 mwait          0.0ms ( 0.0%)          800 Mhz    72.4%
C2 mwait          1.6ms ( 6.3%)
C6 mwait          4.7ms (88.2%)

Wakeups-from-idle per second : 226.5    interval: 10.0s
Power usage (ACPI estimate): 16.1W (3.7 hours)

Top causes for wakeups:
  42.1% (117.6)   PS/2 keyboard/mouse/touchpad interrupt
   0.0% (  0.0)D  rsyslogd
  14.4% ( 40.1)   [kernel scheduler] Load balancing tick
  12.5% ( 35.0)   [eth1, hda_intel] <interrupt>
   7.4% ( 20.6)   [kernel core] hrtimer_start (tick_sched_timer)
   7.0% ( 19.6)   [kernel core] add_timer (wl_timer)

The program 'rsyslogd' is writing to file 'daemon.log' on /dev/sda5.
This prevents the disk from going to powersave mode.
```

Figure 5.3: PowerTOP Software

5.3 Modulation Recognition

In order to prove the feasibility of signal classification through the use of the 2D SCF, MATLAB was used to generate random bit streams, modulate those streams, calculate the SCF, and train HMMs using the SCF images. Those HMMs were then used as models to identify other test images created from a different set of random bits.

Figure 5.4 shows the training method for each modulation scheme. 200 random bits were created for each modulation scheme. Each scheme used the parameters defined in section 2.11. Note that white Gaussian noise was added to produce a -10 dB E_b/N_0 for the first set of tests and -20 dB E_b/N_0 for the second set of tests.

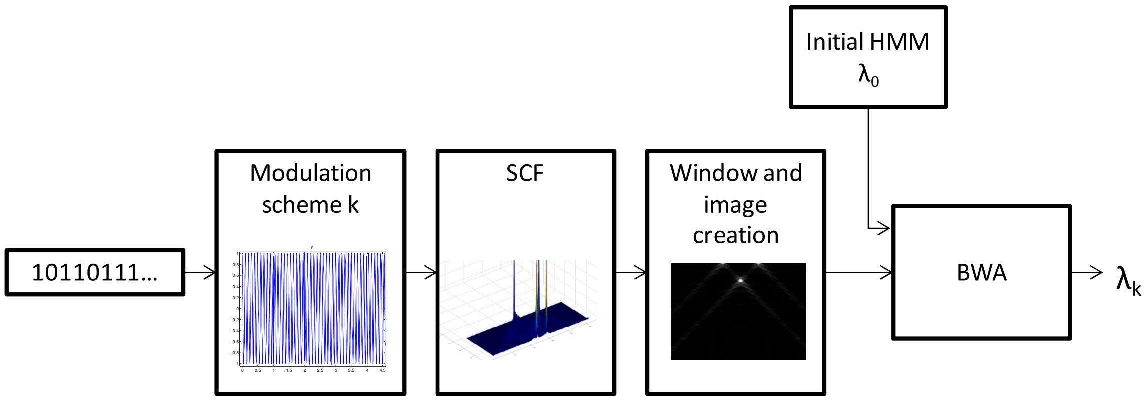


Figure 5.4: HMM Training for Signal Classification

Once the image of the SCF was created, the entire image was vectorized into a single column vector by scanning the grayscale pixel values in a left-to-right and top-to-bottom fashion (Figure 5.5).

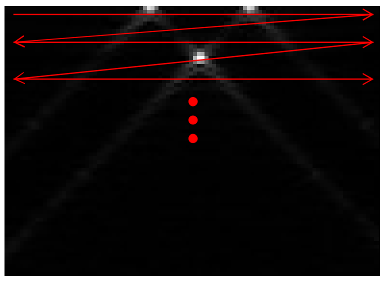


Figure 5.5: SCF Image Vectorization

This vectorized image became the observation sequence to train the HMM. Each 8-bit pixel value (0 through 255) was an observation.

The initial HMM parameters **A** and $\boldsymbol{\pi}$ are the same for all modulation schemes. Subjective intuition was used when determining the number of states, 2, for this application, as the SCF images appeared to be either at one of two extremes: low density or high density. **A** and $\boldsymbol{\pi}$ were arbitrarily chosen as

$$\mathbf{A} = \begin{bmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{bmatrix}$$

$$\boldsymbol{\pi} = \begin{bmatrix} 0.9 & 0.1 \end{bmatrix}$$

B, on the other hand, was carefully calculated as the pmf of pixel values in each state. For the "dark state" (low density), the histogram for all values less than 50 was generated. For the "spike state" (high density), the histogram for all values between 50 and 255 was created. These histograms were normalized to create the pmf of each state. This pmf was used as the associated **B** values in either the dark or spike state. This was repeated for each new training image, creating a new initial **B** matrix for each modulation scheme.

The initial HMM was then trained using the observation sequence generated from the SCF image. This process was repeated for each modulation scheme to produce a model for each scheme. These models were stored in a database for future reference (in this case, the database is simply a collection of text files, as only five modulation schemes were used).

After the models were trained, 100 test SCF images were created for each modulation scheme using a random bit stream of 200 bits each time. Each image was then utilized in the signal classification engine as shown in Figure 5.6 in an attempt to classify the modulation scheme as one of the 5 stored in the database. The five modulation scheme types are given in Table 5.5 and Table 5.6 along with the modulation parameters. As before, the image was vectorized and the observation sequence was used as the input to the Forward Algorithm. The Forward Algorithm was run for each modulation scheme in the database (e.g. 5 times for each of the 5 schemes), and the highest $\Pr(O|\,\lambda)$ was chosen as the most likely modulation scheme.

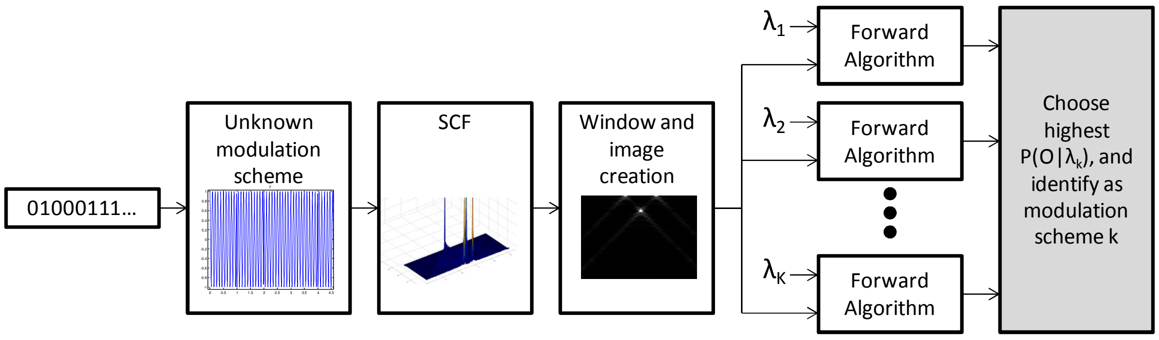

01000111...
Unknown modulation scheme
SCF
Window and image creation
λ_1
λ_2
λ_K
Forward Algorithm
Forward Algorithm
Forward Algorithm
Choose highest $P(O|\lambda_k)$, and identify as modulation scheme k

Figure 5.6: Signal Classification Block Diagram

Table 5.5: Modulation Types and Parameters for -10 dB E_b/N_0

Modulation	No. of Bits	Bits per Symbol	Baud	Carrier Frequency	Sampling Frequency	Scaling Factor	Eb/No
Baseband	200	1	1 Hz	-	100 Hz	1	-10 dB
2FSK	200	1	1 Hz	20 ± 5 Hz	100 Hz	1	-10 dB
4FSK	200	2	1 Hz	20 ± 7.5 ± 2.5 Hz	100 Hz	1	-10 dB
BPSK	200	1	1 Hz	20 Hz	100 Hz	1	-10 dB
QPSK	200	2	1 Hz	20 Hz	100 Hz	1	-10 dB

Table 5.6: Modulation Types and Parameters for -20 dB E_b/N_0

Modulation	No. of Bits	Bits per Symbol	Baud	Carrier Frequency	Sampling Frequency	Scaling Factor	Eb/No
Baseband	200	1	1 Hz	-	100 Hz	1	-20 dB
2FSK	200	1	1 Hz	20 ± 5 Hz	100 Hz	1	-20 dB
4FSK	200	2	1 Hz	20 ± 7.5 ± 2.5 Hz	100 Hz	1	-20 dB
BPSK	200	1	1 Hz	20 Hz	100 Hz	1	-20 dB
QPSK	200	2	1 Hz	20 Hz	100 Hz	1	-20 dB

This classifier was run in MATLAB to produce the confusion matrix found in Chapter 6.

The multiple-HMM Forward Algorithm used for pattern matching as described in section 4.4 was created for C and CUDA. Neutral HMMs and observation sequences similar to the other timing experiments were used for timing the pattern matching algorithm implementations.

5.4 Summary

In this chapter, we examined the specifications of the hardware used to test the various implementations of the algorithms. We also discussed the measurements necessary to form a quality comparison for timing and power. Finally, we described the architecture behind the model training and signal classification system using HMMs.

Chapter 6

Simulation Results

In this chapter, we present the findings of the timing, power, and signal classification experiments. The tests performed were discussed in Chapter 5, which involve timing the various HMM algorithms, measuring power requirements, and testing the effectiveness of the 2D SCF signal classifier.

6.1 Algorithm Timing

The implementation of the Forward Algorithm in C and CUDA was timed while varying the number of states, number of symbols, and the length of the sequence as per section 5.2. These results are presented in Figures 6.1, 6.2, and 6.3.

As can be seen, varying the number of states has the largest effect on the C versus CUDA implementation. It should be noted that up till about 60 states, the GPU is much slower than the CPU, due to the required start up time, memory transfers, and kernel call overhead. The GPU begins to exceed the CPU in timing after 60 states. As that number increases, the GPU is able to far outpace the CPU.

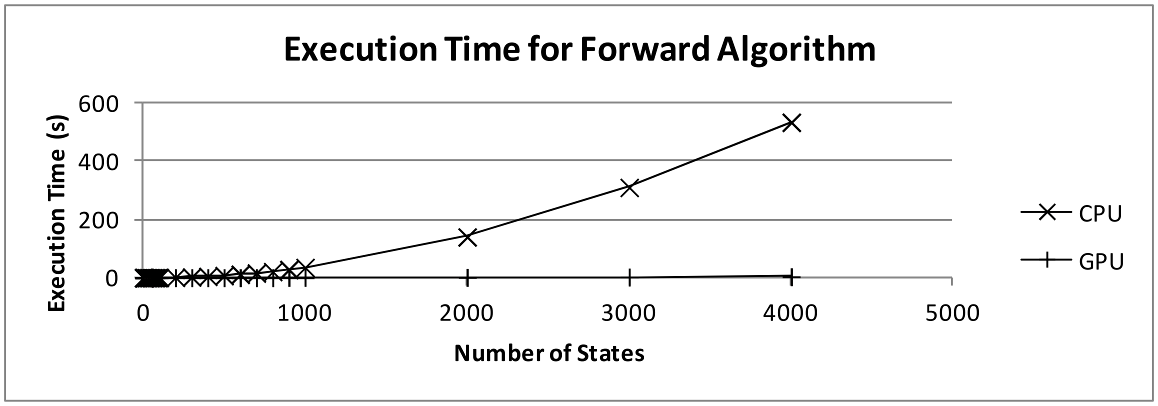


Figure 6.1: Varying States in the Forward Algorithm

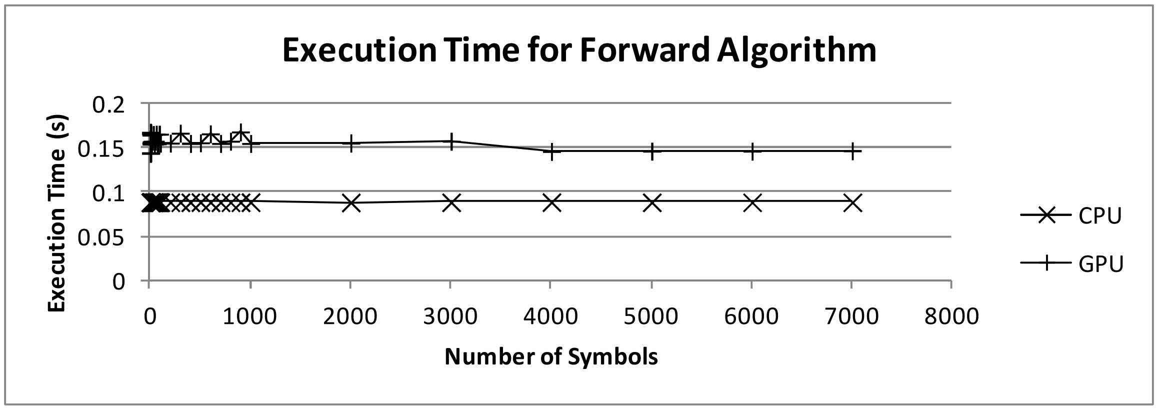


Figure 6.2: Varying Symbols in the Forward Algorithm

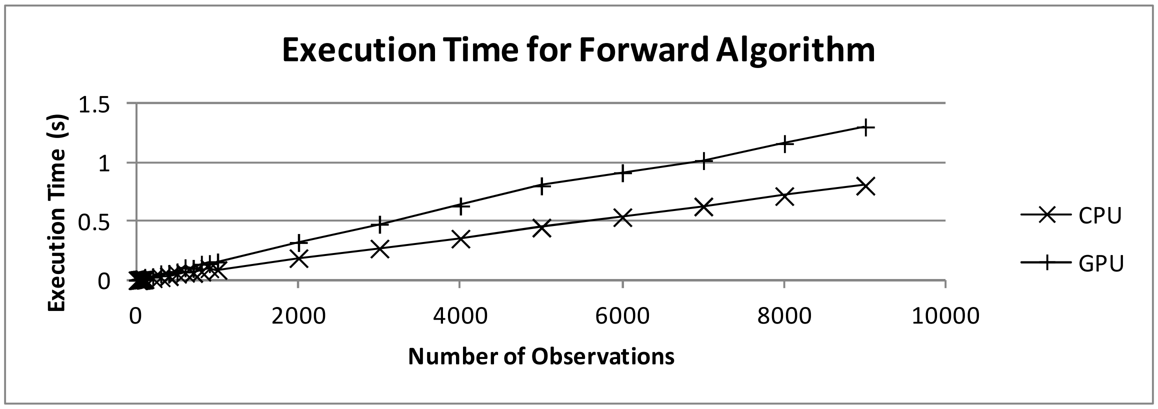


Figure 6.3: Varying Sequence Length in the Forward Algorithm

However, the GPU begins to experience diminishing returns after increasing beyond about 200 states, as seen in Figure 6.4. The diminishing returns can be attributed to the GPU reaching a saturation point with the number of concurrently executing blocks. As a result, some blocks must wait their turn, and some of the parallelization begins to become serial. This limit can be overcome with additional cores in the GPU or multiple GPUs working in parallel.

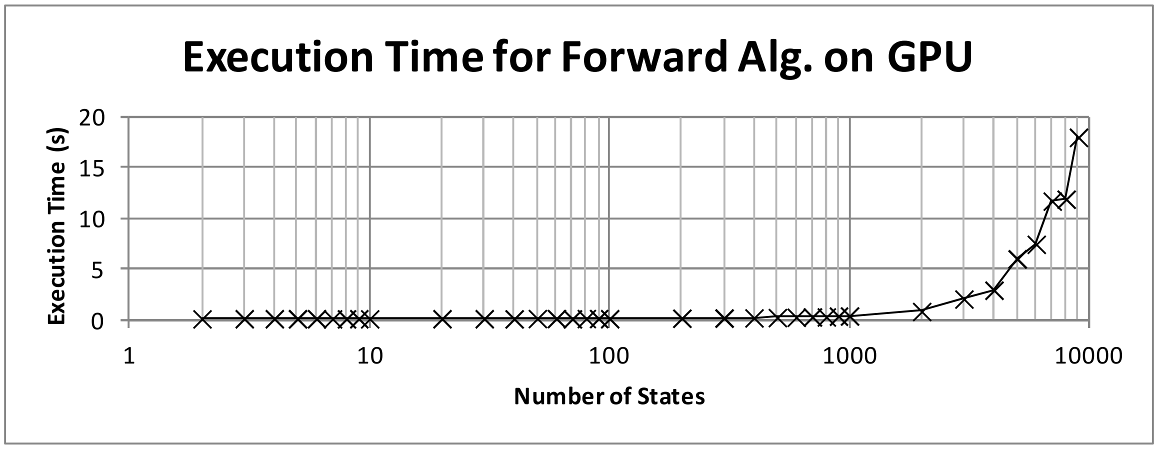


Figure 6.4: Varying States for the Forward Algorithm on the GPU

Varying the number of symbols seems to have little effect on the timing of the algorithm for both implementations. Increasing the number of observations causes the time to increase linearly in both cases, as expected. However, the increase is greater for CUDA due to the extra overhead of making GPU calls.

We then vary the number of states, symbols, and observations for the Viterbi algorithm, comparing the execution time of C versus CUDA. The results are presented in the graphs found in Figures 6.5, 6.6, and 6.7. Similar to the Forward Algorithm, the GPU performs better than the CPU after increasing the number of states past about 60. Also, varying the number of symbols did not affect performance, and increasing the observation length merely increased the execution time in a linear fashion.

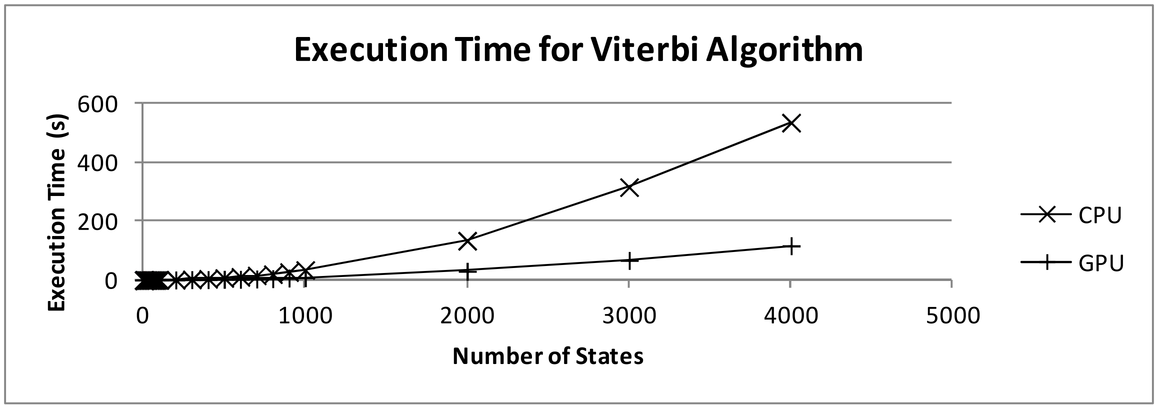


Figure 6.5: Varying States in the Viterbi Algorithm

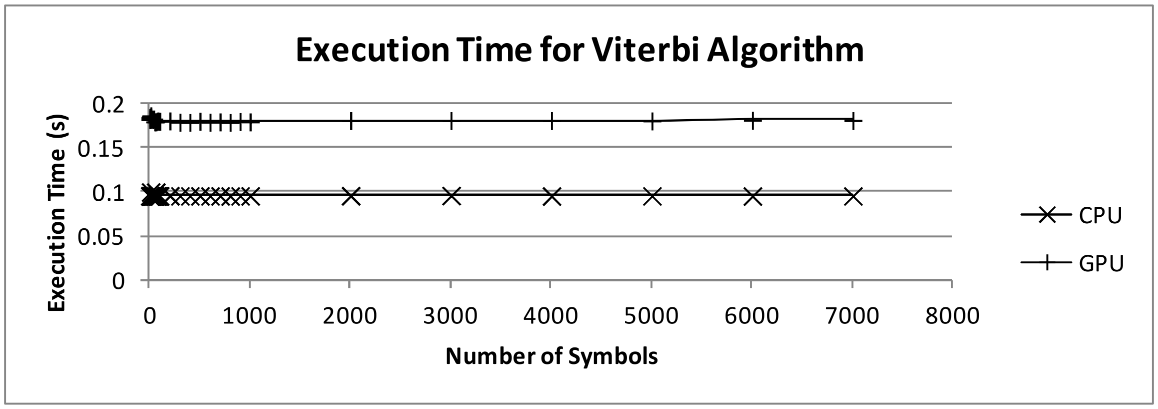


Figure 6.6: Varying Symbols in the Viterbi Algorithm

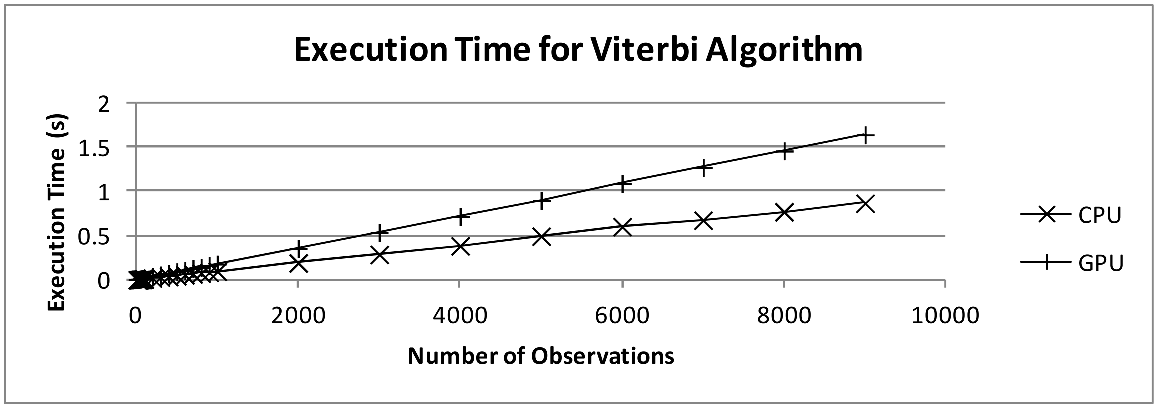


Figure 6.7: Varying Sequence Length in the Viterbi Algorithm

Comparing the speed gains of the Viterbi to the Forward algorithm, we notice that the benefits of the GPU are not as fruitful as hoped. Due to the custom implementation of the *max* and *argmax* functions, we can conclude that further optimization is required in order to see benefits closer to those experienced in the Forward Algorithm. As per Figure 6.8, we experience the same diminishing returns in the Viterbi when varying the number of states.

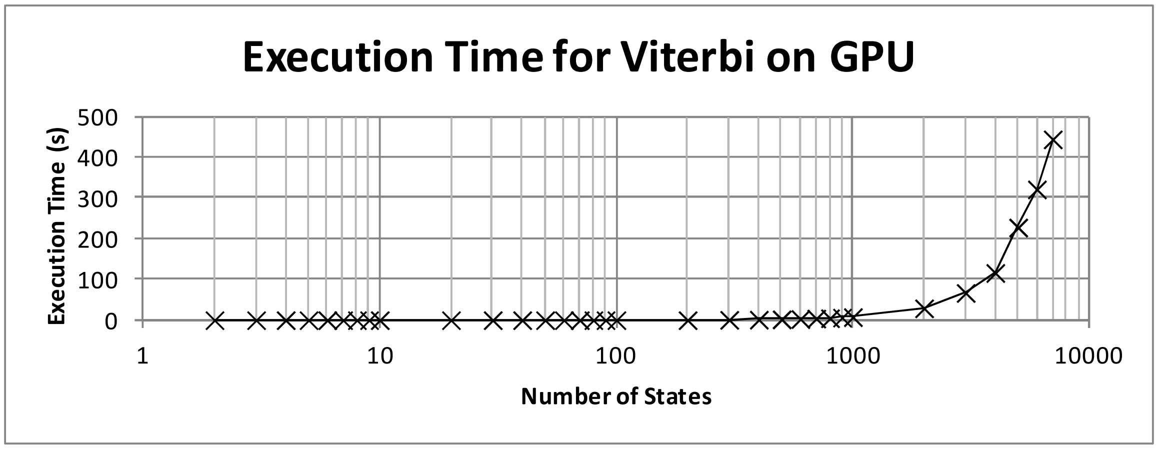


Figure 6.8: Varying States for the Viterbi Algorithm on the GPU

Finally, we compare the timing results of the Baum-Welch Algorithm. The results are presented in Figures 6.9, 6.10, and 6.11. We notice that the results are similar to those found in the Forward Algorithm. However, the execution time of the BWA is longer due to the fact that the BWA is a longer, more complicated algorithm than the FO.

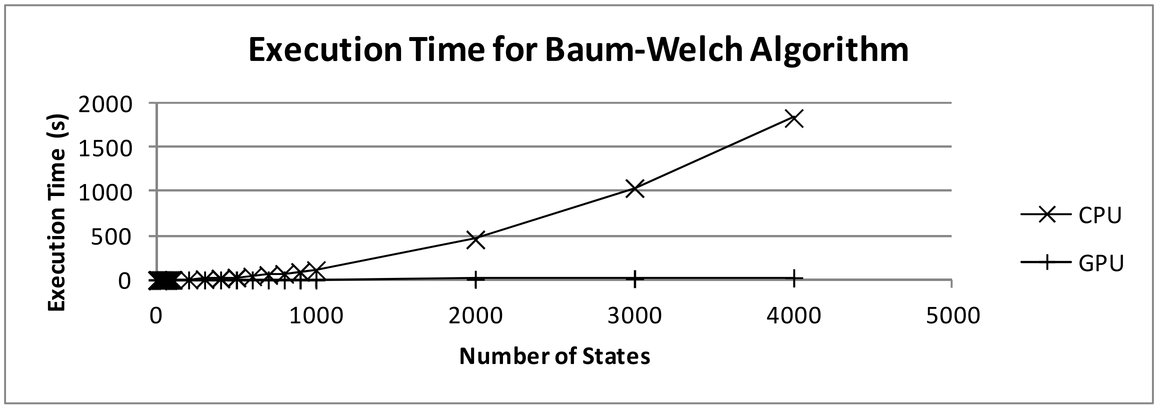


Figure 6.9: Varying States in the Baum-Welch Algorithm

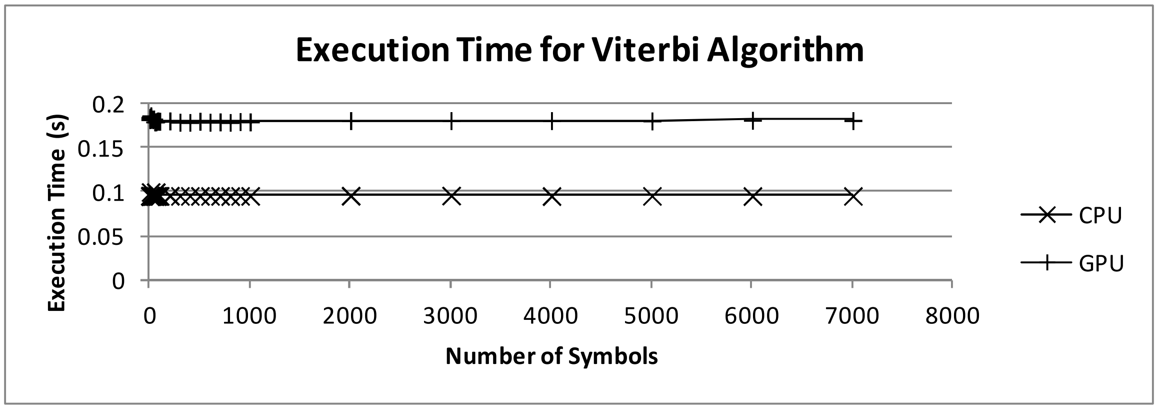


Figure 6.10: Varying Symbols in the Baum-Welch Algorithm

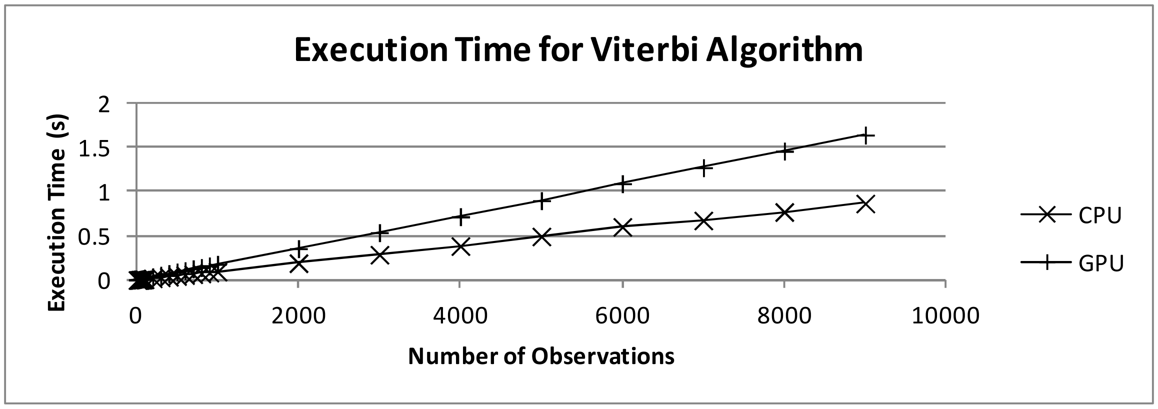


Figure 6.11: Varying Sequence Length in the Baum-Welch Algorithm

We also present the timing of just the GPU implementation of the BWA in Figure 6.12 to examine the diminishing returns that are experienced after about 200 states.

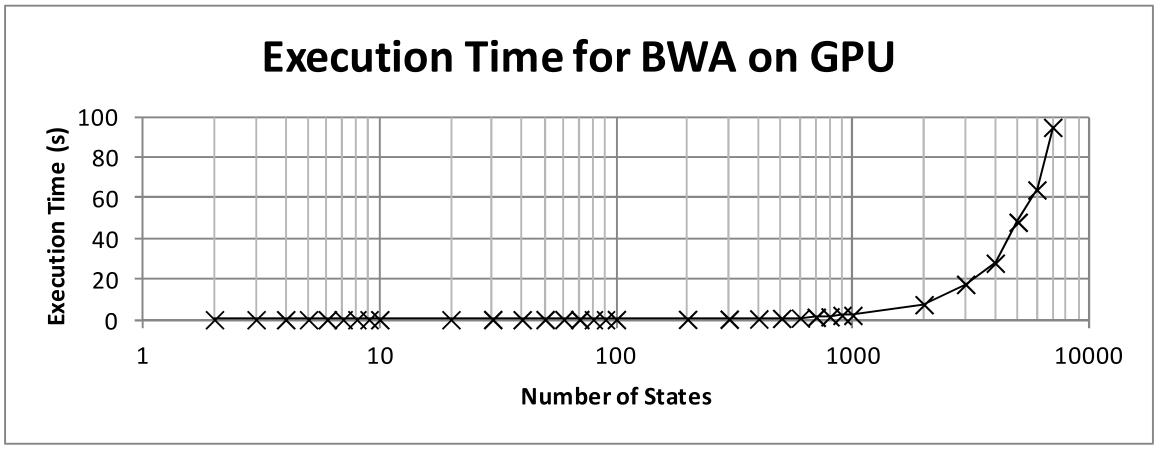


Figure 6.12: Varying States for the Baum-Welch Algorithm on the GPU

We can conclude that the GPU offers significant speed benefits over the CPU for cases where the number of states in the HMM is quite large (e.g. over 60). Table 6.1 presents the speed increase from the GPU to the CPU for 4, 40, 400, and 4000 states for each of the algorithms. We notice that with a low number of states, the GPU lags far behind the CPU, due to the required processing overhead and memory transfers. However, because the algorithms can be parallelized across states, the GPU quickly catches up and surpasses the CPU after about 60 states. At 4000 states, the GPU is about 180× faster than the CPU for the Forward Algorithm, 4× faster for the Viterbi Algorithm, and 65× faster for the Baum-Welch Algorithm for the laptop hardware described in Section 5.1.

Table 6.1: Speed Increase GPU vs. CPU for Each Algorithm

Number of States	CPU Runtime (s)	GPU Runtime (s)	Speed Increase
Forward Algorithm			
4	0.0010	0.1531	0.007x
40	0.0400	0.1393	0.287x
400	4.2816	0.2379	17.99x
4000	534.2028	2.9495	181.12 x
Viterbi Algorithm			
4	0.0033	0.1605	0.021x
40	0.0436	0.1801	0.242x
400	4.2684	1.6595	2.57x
4000	534.5543	116.2531	4.60 x
Baum-Welch Algorithm			
4	0.0021	0.4142	0.005x
40	0.1946	0.4299	0.453x
400	17.6719	0.7502	23.56x
4000	1834.672	28.1271	65.23 x

6.2 Power Measurements

In addition to execution times, the energy consumed for each algorithm was calculated by measuring the average power using the PowerTOP program as described in Section 5.2 and multiplied by the execution time. This resulted in the approximate energy consumed for each run of the algorithm, measured in kWh. Table 6.2 gives the average power utilized for each of the algorithms implemented in C and CUDA. As can be seen, the CUDA implementation consumed more energy than the version running on the CPU. Note that this measurement was taken on the laptop.

Table 6.2: Average Power Utilization

Algorithm	Power (W)	
	C	CUDA
Forward	18.5	26.5
Viterbi	18.5	29.1
BWA	18.3	26.1

Figures 6.13, 6.14, and 6.15 combine the average power utilization with the timing measurements for each of the algorithms to show how the differences in energy consumption between the CPU and the GPU.

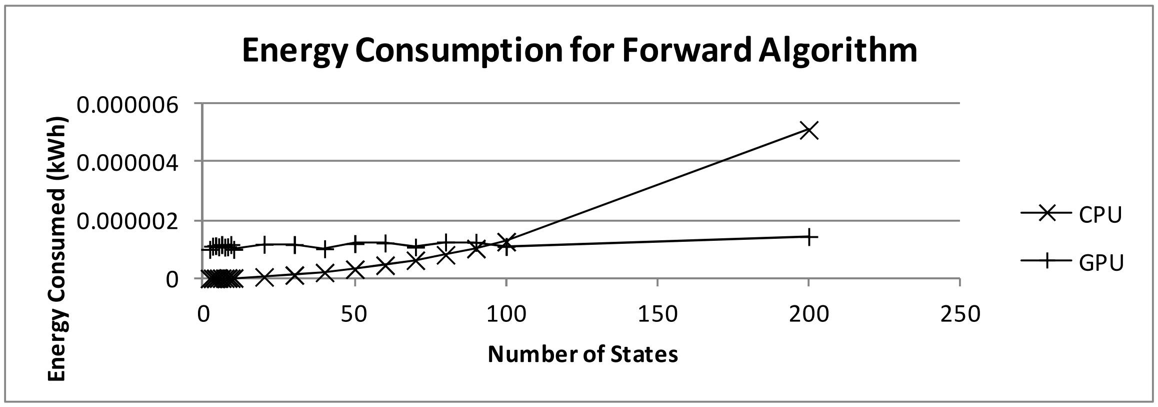


Figure 6.13: Energy Consumption for the Forward Algorithm

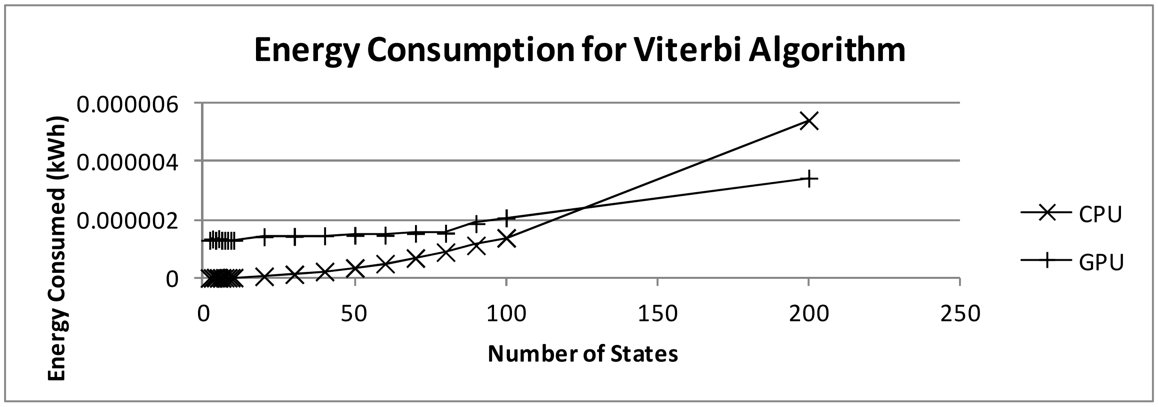


Figure 6.14: Energy Consumption for the Viterbi Algorithm

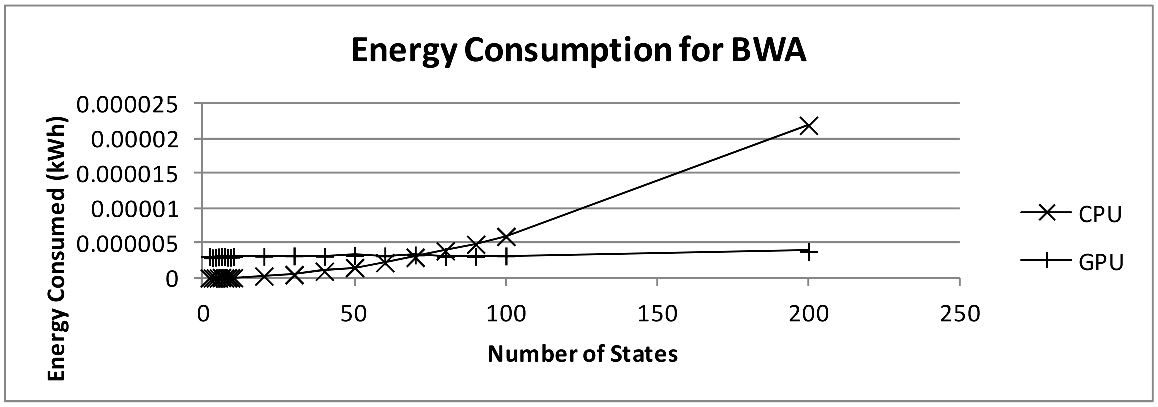


Figure 6.15: Energy Consumption for the Baum-Welch Algorithm

From the Figures, we can see that the GPU is not as energy efficient as the CPU for many applications. While the GPU can surpass the CPU in timing after increasing the states to past 60, we need about 100 states for the FO, 120 states for the Viterbi, and around 70 states for the BWA in order break even on energy consumption. However, after this breakeven point, the GPU can be more energy efficient than the CPU. This proves promising for the GPU to be able to handle HMM-related tasks in an energy efficient manner.

6.3 Signal Classification Results

As per the experiments defined in Section 5.3, we attempted to identify each of the signal modulation schemes. For each modulation type (Baseband, 2FSK, 4FSK, BPSK, and QPSK), we trained a model using 1 observation sequence generated from a random bit stream. Then, we created 100 new test sequences from another set of random bits for each modulation scheme. These sequences were identified using the pattern matching techniques previously defined. The results are shown in Tables 6.3 (-10 dB and Table 6.4 as a confusion matrix[1].

Table 6.3: SCF Matching Confusion Matrix for -10 dB E_b/N_0

Input Signal	*Identified Signal*				
	2FSK	**4FSK**	**Baseband**	**BPSK**	**QPSK**
2FSK	99	1	0	0	0
4FSK	5	95	0	0	0
Baseband	0	0	100	0	0
BPSK	0	0	0	99	1
QPSK	0	0	0	0	100

Table 6.4: SCF Matching Confusion Matrix for -20 dB E_b/N_0

Input Signal	*Identified Signal*				
	2FSK	**4FSK**	**Baseband**	**BPSK**	**QPSK**
2FSK	49	1	0	50	0
4FSK	0	100	0	0	0
Baseband	0	0	91	3	6
BPSK	12	0	2	64	22
QPSK	5	21	0	0	74

As can be seen, the system offered fairly accurate identification for each of the 5 different types of modulation schemes at -10 dB E_b/N_0. Once the noise reached -20 dB E_b/N_0, the classifier was much less accurate. While more experiments can be accomplished in the realm of signal classification by increasing/decreasing SNR, implementing filters, etc., the focus is on GPU implementation of a classifier.

[1] Note that a relatively small number of trials were performed for each modulation type, and an error rate greater than 0% was not observed but could exist.

6.4 Signal Classification System Timing

Using the same parameters for the signal classification system (N = 2, M = 256, T = 6790), we created a timing test using the Forward Algorithm with multiple models on the desktop system described in section 5.1. Figure 6.16 shows the results of this test. As can be seen, the GPU is able to outperform the CPU when more than 40 models are used, even with only 2 states. However, both sets of timing seem to increase linearly, which indicates additional overhead in the CUDA implementation. If 100 models are used, CUDA sees a 1.3× increase over C.

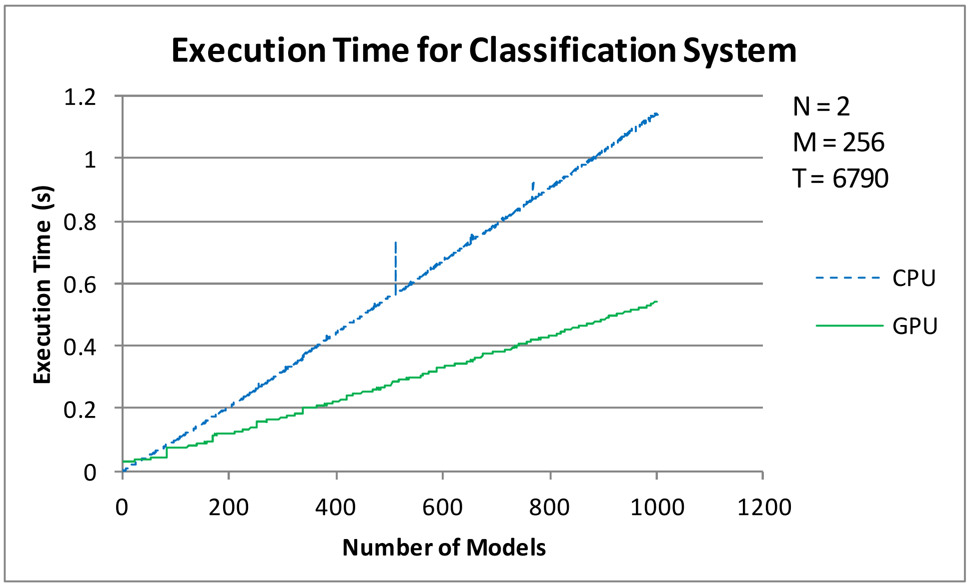


Figure 6.16: Varying Number of Models in FO with 2 States

While the classification system proposed only requires 2 states, other systems might require more. For the sake of completeness, we performed the same tests on a classification system requiring the maximum supported number of states, 16, as allows by the current implementation. These tests were conducted while varying the number of models from 2 to 100, as shown by Figure 6.17. As can be seen, the performance increased drastically for the GPU, as the GPU can parallelize the Forward Algorithm across the states as well as models. With 100 models and 16 states, the GPU experiences a 35× speed increase over the CPU.

We can conclude that for our 5-model, 2 state system, the CPU is a better choice for signal classification. However, as the number of models increases beyond 40, the GPU becomes a more enticing candidate for algorithm execution. Additionally, classification systems that require more than 2 states and require comparison to multiple models can benefit greatly from execution in the GPU.

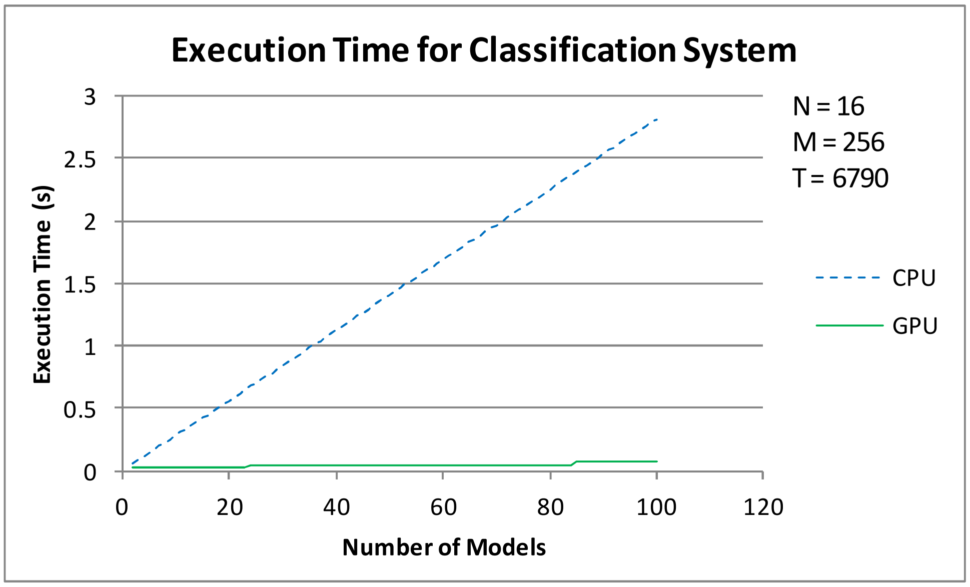


Figure 6.17: Varying Number of Models in FO with 16 States

6.5 Summary

In this chapter we looked at the results from testing the execution time, power consumption, and signal modulation classification. We found that using a 2D image of the real part of the SCF proved to be a near-ideal classifier of a small subset of modulation schemes with -10 dB E_b/N_0. Additionally, the GPU outperformed the CPU in timing when the number of HMM states was greater than 60. However, the GPU, for the given hardware, proved to be less power efficient than the CPU, except when the number of states grew to numbers greater than 100. Finally, the classification system presented was able to benefit from the GPU when more than 40 models were used in the Forward Algorithm.

Chapter

7

Conclusions and Future Work

We draw several conclusions about the results, discuss several potential applications for the research presented, and recommend areas of future research in this chapter. We focus on the GPU as a potential successor or potential co-processor for wireless communication systems.

7.1 Conclusions

From the results presented in Chapter 6, we can conclude that the GPU is better suited than the CPU to handling HMM algorithms, so long as the number of states is greater than 60. In fact, we see about **180× increase in the Forward Algorithm** from the CPU to the GPU with 4000 states, about a **4× faster for the Viterbi Algorithm**, and **65× for the Baum-Welch Algorithm**.

While GPUs have some of the most promising power efficiency (e.g. GFLOPS per Watt), much of that power is wasted if the processor is not saturated with blocks. Additionally, we do not see much power efficiency over the CPU used until around 100 states in the same HMM algorithm tests. If the algorithms cannot be parallelized, then the power efficiency gained by the GPU is lost, as seen in the results.

Using the 2D image of the real part of the Spectral Correlation Function proved successful in signal classification using HMMs, specifically, the Baum-Welch Algorithm for training and the Forward Algorithm for recognition. Very high identification accuracy was found when attempting to identify among Baseband, 2FSK, 4FSK, BPSK, and QPSK modulation schemes with -10 dB E_b/N_0.

Finally, the recognition system was implemented in CUDA, which was able to calculate the $\Pr(O|\,\lambda)$ for multiple models and a single observation sequence in parallel. For the specific classification system designed, this required 2 states and saw a speed increase of 1.3× from CPU to GPU with 100 models. However, if we increase the states to 16, the GPU is 35× faster than the CPU with 100 models.

We can conclude that the GPU is a viable co-processor if the CPU is capable of handing off tasks to the GPU. However, the GPU only truly shines when using HMMs if the number of states is large enough to take advantage of the SIMD-type of operations and allow for parallel execution of the required algorithm. As newer, smaller GPUs make their way into the market, the GPU could become a potential resource to allow for advanced *cognitive radio* (CR) and *software defined radio* (SDR) applications. These could include spectrum sensing, signal identification, and specific emitter identification.

7.2 Applications

HMMs see widespread use in many fields, the most popular including protein folding and speech analysis and recognition. The uses of HMMs have begun to creep its way into the realm of wireless communications. CR requires "smart" transmission and reception systems, which must be aware of its context, including sensing the spectrum for noise or other transmissions, sharing spectrum with other users, and adjusting transmission parameters to allow for better communication.

Kim et al. discuss the ability to utilize HMMs to detect and classify signals [9]. They rely on the SCF to produce a 1D plot of the most prominent cyclic frequencies. These cyclic frequencies become the feature vectors for an HMM.

Again, Kim et al. examine 6 WiFi cards from different manufacturers and are able to uniquely identify the card based on its wireless transmission properties [10]. This is accomplished using the SCF to extract the wireless properties and train an HMM for each WiFi card. The HMM pattern matching scheme worked to identify the cards.

Stamoulakatos and Sykas successfully use HMMs to model vehicle traffic flow patterns by collecting cell phone transmission properties [44]. This is accomplished by logging the received power level information of cell phones and constructing HMMs to model the volumetric flow of cars in a given area.

Motorola has also done research using cell phone data and HMMs to predict call dropping [45]. This is accomplished by comparing HMMs trained from good calls versus HMMs trained from bad calls.

Additionally, SIMD-type processors and GPUs have begun to see use in the SDR/CR realm as well. For example, Woh et al. [46] examine a new SIMD architecture, Ardbeg, and finds that the parallel executions allow for a 1.5-7× speed increase over its predecessor. While this study specifically refers to a specialized *digital signal processor* (DSP), GPUs can be found in many new smart phones, tablets, and laptops.

While GPUs are often less power efficient than DSPs or CPUs, they could prove useful as co-processors for pattern recognition. For example, Othman and Aboulnasr look at 2D HMMs and their application to facial recognition [47]. GPUs could be useful in handling 2D or 3D HMMs of extremely high complexity for similar types of pattern recognition and classification problems.

The Viterbi Algorithm is often used in *Viterbi Decoding* for *forward error correction* in transmitted bit streams. While the Viterbi Algorithm saw the least speed benefit of all the algorithms, potential benefits could be reaped from SIMD-type processing. Interest in specialized Viterbi decoders has been around for a few years [48].

With the advent of smaller GPUs for smart phones and tablets in order to provide 3D user experiences, cognitive radio has a potential co-processor available in many, already available consumer electronics. Once the manufacturers allow running arbitrary code on these tiny GPUs, the devices' CPUs and baseband processors can harness the GPU as a co-processor for CR function. This, however, obviously requires special CR code to be written for that device.

7.3 Further Research

While the results presented prove promising for potential HMM applications in the wireless world, much research is required before implementation of such algorithms into commercial appliances. The following list offers several suggestions for potential research:

1. Focus on the specifics of optimizing one or more of the HMM algorithms. This can include adjusting memory accesses, using pre-existing libraries, or re-arranging program order in attempt to achieve a speed increase. While the GPU sees a speed advantage over 60 states, if this number were reduced, HMMs on the GPU could be quite viable in cognitive radio.
2. Construct a full pattern matching system, which includes signal (or other source) collection, feature extraction, and classification. This would involve running the GPU as a co-processer and use the CPU to feed data to the GPU in a separate thread.
3. Perform additional experiments on the signal classification scheme involving the 2D image of the SCF. This includes adding more noise, interfering signals, smoothing, filtering, etc.
4. Benchmark the performance of small, embedded GPUs, once the technology permits running arbitrary (GPGPU) code on them. Additionally, measure power consumption and compare to current DSPs, CPUs, etc.
5. Postulate a better SIMD architecture specifically for the HMM task at hand (number of states is equal to the number of processing units, for example).
6. Design a smart phone or tablet architecture that allows for direct communication between the baseband processor and GPU, which would allow for CR functions to be offloaded to the GPU.
7. Perform a comparison of a multi-threaded, multi-core CPU implementation of HMM algorithms versus the GPU implementation. If we assume that the algorithm requires no memory accesses and has perfect thread synchronization, we can expect adding a second core to half the execution time. However, the synchronization and memory operations hinder the multi-core performance. As a result, the scaling factor is not linear when we add multiple cores. We would

expect a performance increase of 1.5-2× each time we doubled the number of CPU cores with a program optimized for multiple cores.

8. Research other algorithms to supplement the Viterbi Algorithm and optimize them for SIMD architectures. This could include Dijkstra's algorithm or the Lazy Viterbi algorithm [49]. In particular, the Lazy Viterbi algorithm eschews the *max* and *argmax* functions and uses a *priority-queue* to store the trellis backtracking.

Bibliography

[1] L. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," *Proceedings of the IEEE*, vol. 77, no. 2, Feb. 1989.

[2] G. Mirceva and D. Davcev, "HMM based approach for classifying protein structures," *International Journal of Bio- Science and Bio- Technology*, vol. 1, no. 1, pp. 37-46, Dec. 2009.

[3] I. A. Akbar, "Markov Modeling of Third Generation Wireless Channels," M.S. thesis, Dept. Elect. Eng., Virginia Tech, Blacksburg, VA, 2003.

[4] G. A. Fink, *Markov Models for Pattern Recognition: From Theory to Applications*, Berlin, Germany: Springer, 2003.

[5] H. Othman and T. Aboulnasr, "A separable low complexity 2D HMM with application to face recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 25, no. 10, pp. 1229-1238, Oct. 2003.

[6] F. Silvestro, "Face Segmentation for Identification Using Hidden Markov Models," *Proc. British Maching Vision Conference '93*, BMVC Press, 1993.

[7] J. Hu, M. K. Brown, and W. Turin, "HMM Based On-line Handwriting Recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 18, no. 10, pp. 1039-1045, Oct. 1996.

[8] K. Maruyama and Y. Nakano, "Recognition Method for Cursive Word Written in Latin Characters," *Proc. of the Seventh Intl. Workshop on Frotiers in Handwriting Recognition,* pp. 133-142, Sept. 2000.

[9] K. Kim, I. Akbar, K. K. Bae, J. Um, C. M. Spooner, and J. H. Reed, "Cyclostationary Approaches to Signal Detection and Classification in Cognitive Radio," *2nd IEEE Intl. Symposium on New Frontiers in Dynamic Spectrum Access Networks, 2007,* pp. 212-215, April 2007

[10] K. Kim, I. Akbar, K. K. Bae, J. Um, C. M. Spooner, and J. H. Reed, "Specific Emitter Identification for Cognitive Radio with Application to IEEE 802.11," *IEEE Globecom*, 2008.

[11] *What is GPU Computing?* [Online] Available: http://www.nvidia.com/object/GPU_Computing.html

[12] J. Li, S. Chen,and Y. Li, "The Fast Evaluation of Hidden Markov Models on GPU," *IEEE International Conference on Intelligent Computing and Intelligent Systems*, Nov. 2009.

[13] C. Liu. (2006, May). cuHMM: a CUDA Implementation of Hidden Markov Model Training and Classification. [Online]. Available: http://liuchuan.org/pub/cuHMM.pdf

[14] R. A. Howard, *Dynamic Probabilistic Systems, Volume I: Markov Models*, John Wiley and Sons Inc., 1971.

[15] A. Leon-Garcia, *Probability, Statistics, and Random Processes for Electrical Engineering*, Upper Saddle River, NJ: Prentice Hall, 2008.

[16] L. R. Rabiner and B. H. Juang, "An Introduction to Hidden Markov Models," *IEEE ASSP Magazine*, Jan. 1986.

[17] F. S. Samaria, "Face Recognition Using Hidden Markov Models," Ph.D. dissertation, University of Cambridge, 1995.

[18] B. H. Juang and L. R. Rabiner, "Hidden Markov Models for Speech Recognition," *Technometrics*, vol. 33, no. 3, pp. 251-272, Aug. 1991.

[19] W. Tranter, et al., *Principles of Communication Systems Simulations with Wireless Applications*, Upper Saddle River, New Jersey: Prentice Hall, 2004, ch. 15, pp. 605-611.

[20] A. J. Viterbi, "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm," IEEE Transactions on Information Theory, vol. IT-13, pp. 260-269, April 1967.

[21] G. D. Forney, "The Viterbi Algorithm," *Proc. IEEE*, vol. 61, pp. 268-278, Mar. 1973.

[22] M. A.Trick. (1997). *A Tutorial on Dynamic Programming* [Online]. Available: http://mat.gsia.cmu.edu/classes/dynamic/dynamic.html

[23] A. P. Dempster, N. M. Laird, and D.B. Rubin, "Maximum Likelihood from Incomplete Data via the EM Algorithm," J. Roy. Stat. Soc., vol. 39, no. 1, pp. 1-38, 1977.

[24] L. Rabiner and B. H. Juang, *Fundamentals of Speech Recognition*, Englewood Cliffs, New Jersey: Prentice Hall, 1993.

[25] L. E. Baum, T. Petrie, G. Soules, and N. Weiss, "A Maximization Technique Occurring in the Statistical Analysis of Probabilistic Functions of Markov Chains," *Annals of Mathematical Statistics*, vol. 41, no. 1, pp. 164-171, Feb. 1970.

[26] D. L. Isaacson and R. W. Madsen, *Markov Chains Theory and Applications*, John Wiley & Sons, 1980.

[27] I. Zeljkovic and S. Narayanan, "Improved HMM phone and triphone models for real-time ASR telephony applications," *Proc. of the Fourth Intl. Conf. on Spoken Language*, vol. 2, pp. 1105-1108, Oct. 1996.

[28] P. Billingsley, *Statistical Inference for Markov Processes*, University of Chicago Press, Chicago, IL, Dec. 1961.

[29] I. A. Akbar and W. H. Tranter, "Order estimation of binary hidden Markov wireless channel models in Rayleigh fading," *Proc. of IEEE SoutheastCon*, pp. 202-207, April 2007.

[30] W. Zucchini and I. L. MacDonald, *Hidden Markov Models for Time Series: An Introduction Using R*, Boca Raton, Florida: Chapman and Hall/CRC, 2009.

[31] B. H. Juang and L. R. Rabiner, "The segmental k-means algorithm for estimating the parameters of hidden Markov models," *IEEE Trans. Accoust., Speech, Signal Processing*, vol. 38, no. 9, pp. 1639-1641, Sept. 1990.

[32] L. W. Couch, II, *Digital and Analog Communication Systems*, 6th ed., Upper Saddle River, New Jersey: Prentice Hall, 2001.

[33] W. A. Gardner, "The Spectral Correlation Theory of Cyclostationary Time-Series," *Signal Processing*, vol. 11, pp. 13-36.

[34] W. A. Gardner, "Signal Interception: A Unifying Theoretical Framework for Feature Detection," *IEEE Trans. on Communications*, vol. 36, no. 8., Aug. 1988.

[35] A. Tkachenko, A. D. Cabric, R. W. Brodersen, "Cyclostationary Feature Detector Experiments Using Reconfigurable BEE2," *2nd IEEE Intl. Sym. on New Frontiers in Dynamic Spectrum Access Networks*, pp.216-219, April 2007.

[36] J. Sanders and E. Kandrot, *CUDA By Example*, Boston, MA: Pearson Education, Inc., 2011.

[37] NVIDIA Corporation, "CUDA Programming Guide," ver. 1.1, 2007.

[38] M. Harris, *Optimizing Parallel Reduction in CUDA* [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf

[39] R. Farber (2009, Mar 18), *CUDA, Supercomputing for the Masses: Part 11* [Online]. Available: http://drdobbs.com/high-performance-computing/215900921

[40] J. Li, S. Chen,and Y. Li, "The Fast Evaluation of Hidden Markov Models on GPU," *IEEE International Conference on Intelligent Computing and Intelligent Systems*, Nov. 2009.

[41] C. Liu. (2006, May). cuHMM: a CUDA Implementation of Hidden Markov Model Training and Classification. [Online]. Available: http://liuchuan.org/pub/cuHMM.pdf

[42] D. Zhang, R. Zhao, L. Han, T. Wang, and J. Qu, "An Implementation of Viterbi Algorithm on GPU," *International Conference on Information Science and Engineering*, 2009.

[43] *PowerTOP* [Online]. Available: http://www.lesswatts.org/projects/powertop/

[44] T. S. Stamoulakatos and E. D. Sykas, "Signal Pattern Recognition, Hidden Markov Modeling and Traffic Flow Modeling Filters Applied in Existing Signaling of Cellular Networks for Vehicle Volume Estimation," *Proc. ISICT*, 2003, pp. 378-384.

[45] M. Mohammad, "Prediction of dropped calls in a cellular network," Motorola, Inc., Docket No. CE16868W, May 4, 2006.

[46] M. Woh et al, "From SODA to Scotch: The Evolution of a Wireless Baseband Processor," *International Symposium on Microarchitecture, 2008*. Nov. 2008.

[47] H. Othman and T. Aboulnasr, "A Separable Low Complexity 2D HMM with Application to Face Recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 10, Oct. 2003.

[48] H. Dawiv, G. Fettweis, and H. Meyr, "A CMOS IC for Gb/s Viterbi Decoding: System Design and VLSI Implementation," *IEEE Trans. on VLSI Systems*, vol. 4, no. 1, March 1996.

[49] J. Feldman, I. Abou-Faycl, and M. Frigo, "A fast maximum-likelihood decoder for convolutional codes," *Vehicular Technology Conference*, vol. 1, pp. 371 - 375, 2002.

Appendix A - Forward Algorithm in CUDA

```
/*
 * Copyright (c) 2011, Shawn Hymel <hymelsr@vt.edu>
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without
 * restriction, including without limitation the rights to use, copy,
 * modify, merge, publish, distribute, sublicense, and/or sell copies
 * of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
 * NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
 * BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
 * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 *
 */

/**
 * @file
 * @author  Shawn Hymel
 * @date    March 26, 2011
 * @brief   Implements the Forward Algorithm (FO) in CUBLAS
 */

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

#include <cuda.h>
#include <cutil.h>
#include "/usr/local/cuda/include/cublas.h"

#include "hmm.h"
extern "C" {                    /* required for functions to be visible in C */
#include "hmm_fo_cu.h"
}

#define EXIT_ERROR     1.0f    /**< return error */

/* Contains information about the graphics card's CUDA capabilities */
enum {
    MAX_THREADS_PER_BLOCK = 256
};

/*******************************************************************************
 * Kernels
 */

/* Initialize alpha variables */
__global__ void init_alpha_dev( float *b_d,
                                float *pi_d,
                                int nstates,
                                float *alpha_d,
                                float *ones_d,
                                int obs_t)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < nstates) {
        alpha_d[idx] = pi_d[idx] * b_d[(obs_t * nstates) + idx];
        ones_d[idx] = 1.0f;
    }
}

/* Calculate alpha variables */
__global__ void calc_alpha_dev( int nstates,
                                float *alpha_t_d,
                                float *b_d,
                                int obs_t)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < nstates) {
        alpha_t_d[idx] = alpha_t_d[idx] * b_d[(obs_t * nstates) + idx];
    }
}

/* Scale alpha values */
__global__ void scale_alpha_dev( int nstates, float *alpha_t_d, float scale)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < nstates) {
        alpha_t_d[idx] = alpha_t_d[idx] / scale;
    }
}


/******************************************************************************
 * Forward Algorithm
 */

/* Runs the forward algorithm on the supplied HMM and observation sequence */
float run_hmm_fo(Hmm *in_hmm, Obs *in_obs)
{
    /* Host-side variables */
    int size;
    float *scale;
    float *a = in_hmm->a;
    float *b = in_hmm->b;
    float *pi = in_hmm->pi;
    int nsymbols = in_hmm->nsymbols;
    int nstates = in_hmm->nstates;
    int *obs = in_obs->data;
    int length = in_obs->length;
    int threads_per_block;
    int nblocks;
    int t;
    float log_lik;
    cublasStatus cublas_status;

    /* Device-side variables */
    float *a_d;
    float *b_d;
    float *pi_d;
    float *obs_d;
    float *alpha_d;             /* All alpha variables */
    float *alpha_t_d;           /* nstate alpha values at time t */
    float *ones_d;

    /* Initialize CUBLAS */
    cublas_status = cublasInit();
    if (cublas_status != CUBLAS_STATUS_SUCCESS) {
        fprintf (stderr, "ERROR: CUBLAS Initialization failure\n");
        return EXIT_ERROR;
    }

    /* Allocate host memory */
    scale = (float *) malloc(sizeof(float) * length);
    if (scale == 0) {
        fprintf (stderr, "ERROR: Host memory allocation error (scale)\n");
        return EXIT_ERROR;
    }

    /* Allocate device memory */
    size = sizeof(float) * nstates * nstates;
    CUDA_SAFE_CALL( cudaMalloc((void**)&a_d, size) );
    CUDA_SAFE_CALL( cudaMemcpy(a_d, a, size, cudaMemcpyHostToDevice) );
    size = sizeof(float) * nstates * nsymbols;
    CUDA_SAFE_CALL( cudaMalloc((void**)&b_d, size) );
    CUDA_SAFE_CALL( cudaMemcpy(b_d, b, size, cudaMemcpyHostToDevice) );
    size = sizeof(float) * nstates;
    CUDA_SAFE_CALL( cudaMalloc((void**)&pi_d, size) );
    CUDA_SAFE_CALL( cudaMemcpy(pi_d, pi, size, cudaMemcpyHostToDevice) );
    size = sizeof(float) * length;
    CUDA_SAFE_CALL( cudaMalloc((void**)&obs_d, size) );
    CUDA_SAFE_CALL( cudaMemcpy(obs_d, obs, size, cudaMemcpyHostToDevice) );
    size = sizeof(float) * nstates * length;
    CUDA_SAFE_CALL( cudaMalloc((void**)&alpha_d, size) );
    size = sizeof(float) * nstates;
    CUDA_SAFE_CALL( cudaMalloc((void**)&alpha_t_d, size) );
    size = sizeof(float) * nstates;
    CUDA_SAFE_CALL( cudaMalloc((void**)&ones_d, size) );

    /* Initialize alpha variables */
    threads_per_block = MAX_THREADS_PER_BLOCK;
    nblocks = (nstates + threads_per_block - 1) / threads_per_block;
    init_alpha_dev<<<nblocks, threads_per_block>>>( b_d,
                                                    pi_d,
                                                    nstates,
                                                    alpha_d,
                                                    ones_d,
                                                    obs[0]);
    size = sizeof(float) * nstates;
    CUDA_SAFE_CALL( cudaMemcpy( alpha_t_d,
                                alpha_d,
                                size,
                                cudaMemcpyDeviceToDevice) );

    /* Sum alpha values to get scaling factor */
    cublasGetError();
    scale[0] = cublasSdot(nstates, alpha_t_d, 1, ones_d, 1);
    cublas_status = cublasGetError();
    if (cublas_status != CUBLAS_STATUS_SUCCESS) {
        fprintf (stderr, "ERROR: Kernel execution error\n");
        return EXIT_ERROR;
    }

    /* Scale alpha values */
    scale_alpha_dev<<<nblocks, threads_per_block>>>(    nstates,
                                                        alpha_t_d,
                                                        scale[0]);

    /* Copy temporary alpha values back to alpha matrix */
    CUDA_SAFE_CALL( cudaMemcpy( alpha_d,
                                alpha_t_d,
                                size,
                                cudaMemcpyDeviceToDevice) );

    /* Initialize log likelihood */
    log_lik = log10(scale[0]);

    /* Calculate the rest of the alpha variables */
    for (t = 1; t < length; t++) {

        /* Multiply transposed A matrix by alpha(t-1) */
        /* Note: the matrix is auto-transposed by cublas reading column major */
        cublasSgemv( 'N', nstates, nstates, 1.0f, a_d, nstates, alpha_t_d, 1, 0,
                                                                   alpha_t_d, 1);
        cublas_status = cublasGetError();
        if (cublas_status != CUBLAS_STATUS_SUCCESS) {
            fprintf (stderr, "ERROR: Kernel execution error\n");
            return EXIT_ERROR;
        }

        /* Calculate alpha(t) */
        calc_alpha_dev<<<nblocks, threads_per_block>>>( nstates,
                                                        alpha_t_d,
                                                        b_d,
                                                        obs[t]);

        /* Sum alpha values to get scaling factor */
        scale[t] = cublasSdot(nstates, alpha_t_d, 1, ones_d, 1);
        cublas_status = cublasGetError();
        if (cublas_status != CUBLAS_STATUS_SUCCESS) {
            fprintf (stderr, "ERROR: Kernel execution error\n");
            return EXIT_ERROR;
        }

        /* Scale alpha values */
        scale_alpha_dev<<<nblocks, threads_per_block>>>(    nstates,
                                                            alpha_t_d,
                                                            scale[t]);

        /* Copy temporary alpha values back to alpha matrix */
        CUDA_SAFE_CALL( cudaMemcpy( alpha_d + (t * nstates),
                                    alpha_t_d,
                                    size,
                                    cudaMemcpyDeviceToDevice) );

        /* Update log likelihood */
        log_lik += log10(scale[t]);
    }

    /* Free device memory */
    CUDA_SAFE_CALL( cudaFree(a_d) );
    CUDA_SAFE_CALL( cudaFree(b_d) );
    CUDA_SAFE_CALL( cudaFree(pi_d) );
    CUDA_SAFE_CALL( cudaFree(obs_d) );
    CUDA_SAFE_CALL( cudaFree(alpha_d) );

    return log_lik;
}
```

Appendix B - Viterbi Algorithm in CUDA

```
/*
 * Copyright (c) 2011, Shawn Hymel <hymelsr@vt.edu>
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without
 * restriction, including without limitation the rights to use, copy,
 * modify, merge, publish, distribute, sublicense, and/or sell copies
 * of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
 * NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
 * BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
 * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 *
 */

/**
 * @file
 * @author  Shawn Hymel
 * @date    June 1, 2011
 * @brief   Implements the Viterbi Algorithm in CUBLAS
 */

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

#include <cuda.h>
#include <cutil.h>
#include "/usr/local/cuda/include/cublas.h"

#include "hmm.h"
extern "C" {
#include "hmm_vit_cublas.h"
}

/* Contains information about the graphics card's CUDA capabilities */
enum {
    MAX_THREADS_PER_BLOCK = 256,
    BLOCK_DIM = 16
};

/*******************************************************************************
 * Kernels
 */

/* Initialize index block */
__global__ void init_idx( int *idx_d, int nstates )
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int idy = blockIdx.y * blockDim.y + threadIdx.y;

    if (idx < nstates && idy < nstates) {
        idx_d[(idy * nstates) + idx] = idx;
    }
}
```

```
/* Initialize delta */
__global__ void init_delta( float *delta_d,
                            float *pi_d,
                            float *b_d,
                            float *ones_n_d,
                            int nstates,
                            int obs_t)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < nstates) {
        delta_d[idx] = pi_d[idx] * b_d[(obs_t * nstates) + idx];
        ones_n_d[idx] = 1.0f;
    }
}

/* Scale delta values */
__global__ void scale_delta(    float *delta_d,
                                int nstates,
                                float sum_delta,
                                int t )
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < nstates) {
        delta_d[(t * nstates) + idx] = delta_d[(t * nstates) + idx] / sum_delta;
    }
}

/* Calculate the inner product for delta and psi */
__global__ void calc_inner_prod( float *inner_prod_d,
                                 float *delta_d,
                                 float *a_d,
                                 int nstates,
                                 int t)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int idy = blockIdx.y * blockDim.y + threadIdx.y;

    if (idx < nstates && idy < nstates) {
        inner_prod_d[(idy * nstates) + idx] = delta_d[((t-1) * nstates) + idx] *
                                        a_d[(idx * nstates) + idy];
    }
}

/* Perform reduction to find max in each row in an array */
__global__ void max_reduction(  float *idata_d,
                                int *idx_d,
                                int nstates)
{
    unsigned int stride;
    unsigned int tdx = threadIdx.x;
    unsigned int tdy = threadIdx.y;
    unsigned int bdx = blockIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int idy = blockIdx.y * blockDim.y + threadIdx.y;
    __shared__ float partial_max[BLOCK_DIM * BLOCK_DIM];
    __shared__ float partial_idx[BLOCK_DIM * BLOCK_DIM];

    /* Copy data to partial max */
    if (idx < nstates && idy < nstates) {
        partial_max[(tdy * BLOCK_DIM) + tdx] = idata_d[(idy * nstates) + idx];
        partial_idx[(tdy * BLOCK_DIM) + tdx] = idx_d[(idy * nstates) + idx];
        idata_d[(idy * nstates) + idx] = 0;
        idx_d[(idy * nstates) + idx] = -1.0f;
    } else {
        partial_max[(tdy * BLOCK_DIM) + tdx] = 0;
        partial_idx[(tdy * BLOCK_DIM) + tdx] = 0;
    }
    /* Calculate max and store in partial max */
    __syncthreads();
    for (stride = blockDim.x/2; stride > 0; stride /= 2) {
        if (tdx < stride && idy < nstates) {
            if (partial_max[(tdy * BLOCK_DIM) + (tdx + stride)] >
                            partial_max[(tdy * BLOCK_DIM) + tdx]) {
                partial_max[(tdy * BLOCK_DIM) + tdx] =
                            partial_max[(tdy * BLOCK_DIM) + (tdx + stride)];
                partial_idx[(tdy * BLOCK_DIM) + tdx] =
                            partial_idx[(tdy * BLOCK_DIM) + (tdx + stride)];
            }
        }
        __syncthreads();
    }

    /* Store answer */
    if (tdx == 0 && idy < nstates) {
        idata_d[(idy * nstates) + bdx] = partial_max[tdy * BLOCK_DIM];
        idx_d[(idy * nstates) + bdx] = partial_idx[tdy * BLOCK_DIM];
    }
}

/* Calculate delta */
__global__ void calc_delta_psi( float *delta_d,
                            int *psi_d,
                            float *inner_prod_d,
                            float *b_d,
                            int *idx_d,
                            int obs_t,
                            int nstates,
                            int t)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < nstates) {
        delta_d[(t * nstates) + idx] = inner_prod_d[idx * nstates] *
                                    b_d[(obs_t * nstates) + idx];
        psi_d[(t * nstates) + idx] = idx_d[idx * nstates];
    }
}

 /*******************************************************************************
 * Viterbi Function
 */

/* Runs the Viterbi Algorithm on the supplied HMM and observation sequence */
void run_hmm_vit(Hmm *hmm, Obs *in_obs, int *state_seq)
{

    /* Host-side variables */
    float *a = hmm->a;
    float *b = hmm->b;
    float *pi = hmm->pi;
    int nstates = hmm->nstates;
    int nsymbols = hmm->nsymbols;
    int *obs = in_obs->data;
    int *psi;
    int length = in_obs->length;
    int threads_per_block;
    int nblocks;
    int max_len;
    int needed_blocks;
    int size;
    float sum_delta;
    cublasStatus cublas_status;
    int t = 0;

    /* Device-side variables */
    float *a_d;
    float *b_d;
    float *pi_d;
    float *delta_d;
    int *psi_d;
    float *inner_prod_d;
    float *ones_n_d;
    int *idx_d;

    /* Initialize CUBLAS */
    cublas_status = cublasInit();
    if (cublas_status != CUBLAS_STATUS_SUCCESS) {
        fprintf (stderr, "ERROR: CUBLAS Initialization failure\n");
        return;
    }

    /* Allocate memory */
    psi = (int *)malloc(sizeof(int) * length * nstates);

    /* Allocate device memory */
    size = sizeof(float) * nstates * nstates;
    CUDA_SAFE_CALL( cudaMalloc((void**)&a_d, size) );
    CUDA_SAFE_CALL( cudaMemcpy(a_d, a, size, cudaMemcpyHostToDevice) );
    size = sizeof(float) * nstates * nsymbols;
    CUDA_SAFE_CALL( cudaMalloc((void**)&b_d, size) );
    CUDA_SAFE_CALL( cudaMemcpy(b_d, b, size, cudaMemcpyHostToDevice) );
    size = sizeof(float) * nstates;
    CUDA_SAFE_CALL( cudaMalloc((void**)&pi_d, size) );
    CUDA_SAFE_CALL( cudaMemcpy(pi_d, pi, size, cudaMemcpyHostToDevice) );
    size = sizeof(float) * nstates * length;
    CUDA_SAFE_CALL( cudaMalloc((void**)&delta_d, size) );
    size = sizeof(int) * nstates * length;
    CUDA_SAFE_CALL( cudaMalloc((void**)&psi_d, size) );
    size = sizeof(float) * nstates * nstates;
    CUDA_SAFE_CALL( cudaMalloc((void**)&inner_prod_d, size) );
    size = sizeof(float) * nstates;
    CUDA_SAFE_CALL( cudaMalloc((void**)&ones_n_d, size) );
    size = sizeof(float) * nstates * nstates;
    CUDA_SAFE_CALL( cudaMalloc((void**)&idx_d, size) );

    /* Initialization */
    threads_per_block = MAX_THREADS_PER_BLOCK;
    nblocks = (nstates + threads_per_block - 1) / threads_per_block;
    init_delta<<<nblocks, threads_per_block>>>( delta_d,
                                                pi_d,
                                                b_d,
                                                ones_n_d,
                                                nstates,
                                                obs[0]);
    sum_delta = cublasSdot(nstates, delta_d, 1, ones_n_d, 1);
    cublas_status = cublasGetError();
    if (cublas_status != CUBLAS_STATUS_SUCCESS) {
        fprintf (stderr, "ERROR: Kernel execution error\n");
        return;
    }
    scale_delta<<<nblocks, threads_per_block>>>( delta_d,
                                                nstates,
                                                sum_delta,
                                                0 );
    size = sizeof(float) * nstates;
    CUDA_SAFE_CALL( cudaMemset(psi_d, 0, size) );

    /* Recursive step */
    for (t = 1; t < length; t++) {

        dim3 threads(BLOCK_DIM, BLOCK_DIM);
        nblocks = (nstates + BLOCK_DIM - 1) / BLOCK_DIM;
        dim3 grid(nblocks, nblocks);

        /* Initialize index block */
        init_idx<<<grid, threads>>>(idx_d, nstates);
        /* Calculate inner product */
        calc_inner_prod<<<grid, threads>>>( inner_prod_d,
                                            delta_d,
                                            a_d,
                                            nstates,
                                            t);

        /* Calculate maximum values in each vector */
        size = sizeof(float) * nstates * nstates;
        needed_blocks = (nstates + BLOCK_DIM - 1) / BLOCK_DIM;
        max_len = nstates;
        while (max_len > 1) {

            max_reduction<<<grid, threads>>>(   inner_prod_d,
                                                idx_d,
                                                nstates);

            max_len = needed_blocks;
            needed_blocks = (max_len + BLOCK_DIM - 1) / BLOCK_DIM;
        }

        /* Calculate delta and psi */
        threads_per_block = MAX_THREADS_PER_BLOCK;
        nblocks = (nstates + threads_per_block - 1) / threads_per_block;
        calc_delta_psi<<<nblocks, threads_per_block>>>( delta_d,
                                                        psi_d,
                                                        inner_prod_d,
                                                        b_d,
                                                        idx_d,
                                                        obs[t],
                                                        nstates,
                                                        t);
        sum_delta = cublasSdot(nstates, delta_d + (t*nstates), 1, ones_n_d, 1);
        cublas_status = cublasGetError();
        if (cublas_status != CUBLAS_STATUS_SUCCESS) {
            fprintf (stderr, "ERROR: Kernel execution error\n");
            return;
        }
        scale_delta<<<nblocks, threads_per_block>>>(    delta_d,
                                                        nstates,
                                                        sum_delta,
                                                        t);

    }

    /* Termination */
    state_seq[length - 1] = cublasIsamax(   nstates,
                                            delta_d + ((length-1) * nstates),
                                            1);
    state_seq[length - 1] = state_seq[length - 1] - 1; /* Fix 1-based index */

    /* Find most probable path */
    size = sizeof(int) * nstates * length;
    CUDA_SAFE_CALL( cudaMemcpy(psi, psi_d, size, cudaMemcpyDeviceToHost) );
    for (t = length - 2; t >= 0; t--) {
        state_seq[t] = psi[((t+1) * nstates) + state_seq[t+1]];
    }

    /* Shutdown CUBLAS */
    if(cublasShutdown() != CUBLAS_STATUS_SUCCESS) {
        printf("ERROR: CUBLAS Shutdown failure\n");
        return;
    }

    /* Free memory */
    free(psi);
    CUDA_SAFE_CALL( cudaFree(a_d) );
    CUDA_SAFE_CALL( cudaFree(b_d) );
    CUDA_SAFE_CALL( cudaFree(pi_d) );
    CUDA_SAFE_CALL( cudaFree(delta_d) );
    CUDA_SAFE_CALL( cudaFree(psi_d) );
    CUDA_SAFE_CALL( cudaFree(inner_prod_d) );
    CUDA_SAFE_CALL( cudaFree(ones_n_d) );
    CUDA_SAFE_CALL( cudaFree(idx_d) );
}
```

Appendix C - Baum-Welch Algorithm in CUDA

```
/*
 * Copyright (c) 2011, Shawn Hymel <hymelsr@vt.edu>
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without
 * restriction, including without limitation the rights to use, copy,
 * modify, merge, publish, distribute, sublicense, and/or sell copies
 * of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
 * NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
 * BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
 * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 *
 */

/**
 * @file
 * @author  Shawn Hymel
 * @date    May 28, 2011
 * @brief   Implements the Baum-Welch Algorithm (BWA) in CUDA
 */

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

#include <cuda.h>
#include <cutil.h>
#include "/usr/local/cuda/include/cublas.h"

#include "hmm.h"
extern "C" {
#include "hmm_bwa_cublas.h"
}

#define EXIT_ERROR  1.0f

/* Contains information about the graphics card's CUDA capabilities */
enum {
    MAX_THREADS_PER_BLOCK = 256,
    BLOCK_DIM = 16
};

/*******************************************************************************
 * Global variables
 */
cublasStatus cublas_status;         /* Holds status of CUBLAS call */
int nstates;                        /* The number of states in the HMM */
int nsymbols;                       /* The number of possible symbols */
int *obs;                           /* The observation sequence */
int length;                         /* The length of the observation sequence */
float *scale;                       /* Scaling factor as determined by alpha */
float *a_d;                         /* A matrix on GPU */
float *b_d;                         /* B matrix on GPU */
float *pi_d;                        /* Pi matrix on GPU */
float *alpha_d;                     /* Forward variables (alpha) on GPU */
float *beta_d;                      /* Backward variables (beta) on GPU */
float *gamma_sum_d;                 /* Sum of gamma variables on GPU */
float *xi_sum_d;                    /* Sum of xi variables on GPU */
float *c_d;                         /* Temporary array on GPU */
float *ones_n_d;                    /* Length <states> array of 1s on GPU */
float *ones_s_d;                    /* Length <symbols> array of 1s on GPU */

/********************************************************************************
 * Kernels
 */

/* Initialize ones vector */
__global__ void init_ones_dev(  float *ones_s_d,
                                int nsymbols)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < nsymbols) {
        ones_s_d[idx] = 1.0f;
    }
}

/* Initialize alpha variables */
__global__ void init_alpha_dev( float *b_d,
                                float *pi_d,
                                int nstates,
                                float *alpha_d,
                                float *ones_n_d,
                                int obs_t)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < nstates) {
        alpha_d[idx] = pi_d[idx] * b_d[(obs_t * nstates) + idx];
        ones_n_d[idx] = 1.0f;
    }
}

/* Calculate alpha variables */
__global__ void calc_alpha_dev( int nstates,
                                float *alpha_d,
                                float *b_d,
                                int obs_t)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < nstates) {
        alpha_d[idx] = alpha_d[idx] * b_d[(obs_t * nstates) + idx];
    }
}

/* Scale alpha values */
__global__ void scale_alpha_dev( int nstates, float *alpha_d, float scale)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < nstates) {
        alpha_d[idx] = alpha_d[idx] / scale;
    }
}

/* Initialize beta values */
__global__ void init_beta_dev(  int nstates,
                                float *beta_d,
                                float scale)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < nstates) {
        beta_d[idx] = 1.0f / scale;
    }
}

/* Calculate beta variables */
__global__ void calc_beta_dev(  float *beta_d,
                                float *b_d,
                                float scale_t,
                                int nstates,
                                int obs_t,
                                int t)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < nstates) {
        beta_d[(t * nstates) + idx] = beta_d[((t + 1) * nstates) + idx] *
                                      b_d[(obs_t * nstates) + idx] / scale_t;
    }
}

/* Sum next iteration of gamma variables */
__global__ void calc_gamma_dev( float *gamma_sum_d,
                                float *alpha_d,
                                float *beta_d,
                                int nstates,
                                int t)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < nstates) {
        gamma_sum_d[idx] += alpha_d[(t * nstates) + idx] *
                            beta_d[(t * nstates) + idx];
    }
}

/* Sum next iteration of xi variables */
__global__ void calc_xi_dev(    float *xi_sum_d,
                                float *a_d,
                                float *b_d,
                                float *alpha_d,
                                float *beta_d,
                                float sum_ab,
                                int nstates,
                                int obs_t,
                                int t)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int idy = blockIdx.y * blockDim.y + threadIdx.y;

    if (idx < nstates && idy < nstates) {
        xi_sum_d[(idy * nstates) + idx] += alpha_d[(t * nstates) + idy] *
                                          a_d[(idy * nstates) + idx] *
                                          b_d[(obs_t * nstates) + idx] *
                                          beta_d[((t+1) * nstates) + idx] /
                                          sum_ab;
    }
}

/* Re-estimate A matrix */
__global__ void est_a_dev(  float *a_d,
                            float *alpha_d,
                            float *beta_d,
                            float *xi_sum_d,
                            float *gamma_sum_d,
                            float sum_ab,
                            int nstates,
                            int length)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int idy = blockIdx.y * blockDim.y + threadIdx.y;
    if (idx < nstates && idy < nstates) {
        a_d[(idy * nstates) + idx] = xi_sum_d[(idy * nstates) + idx] /
                                    (gamma_sum_d[idy] -
                                    alpha_d[(length * nstates) + idy] *
                                    beta_d[(length * nstates) + idy] /
                                    sum_ab);
    }
}

/* Normalize A matrix */
__global__ void scale_a_dev(    float *a_d,
                                float *c_d,
                                int nstates)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int idy = blockIdx.y * blockDim.y + threadIdx.y;

    if (idx < nstates && idy < nstates) {
        a_d[(idy * nstates) + idx] = a_d[(idy * nstates) + idx] / c_d[idy];
    }
}

/* Accumulate B values */
__global__ void acc_b_dev(  float *b_d,
                            float *alpha_d,
                            float *beta_d,
                            float sum_ab,
                            int nstates,
                            int nsymbols,
                            int obs_t,
                            int t)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int idy = blockIdx.y * blockDim.y + threadIdx.y;

    if (idy < nsymbols && idx < nstates && obs_t == idy) {
        b_d[(idy * nstates) + idx] += alpha_d[(t * nstates) + idx] *
                                      beta_d[(t * nstates) + idx] / sum_ab;
    }
}

/* Re-estimate B values */
__global__ void est_b_dev(  float *b_d,
                            float *gamma_sum_d,
                            int nstates,
                            int nsymbols)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int idy = blockIdx.y * blockDim.y + threadIdx.y;

    if (idy < nsymbols && idx < nstates) {
        b_d[(idy * nstates) + idx] = b_d[(idy * nstates) + idx] /
                                      gamma_sum_d[idx];
    }
}

/* Normalize B matrix */
__global__ void scale_b_dev(    float *b_d,
                                float *c_d,
                                int nstates,
                                int nsymbols)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int idy = blockIdx.y * blockDim.y + threadIdx.y;

    if (idx < nstates && idy < nsymbols) {
        if (b_d[(idy * nstates) + idx] == 0) {
            b_d[(idy * nstates) + idx] = 1e-10;
        } else {
            b_d[(idy * nstates) + idx] = b_d[(idy * nstates) + idx] / c_d[idx];
        }
    }
}

/* Re-estimate Pi values */
__global__ void est_pi_dev( float *pi_d,
                            float *alpha_d,
                            float *beta_d,
                            float sum_ab,
                            int nstates)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < nstates) {
        pi_d[idx] = alpha_d[idx] * beta_d[idx] / sum_ab;
    }
}

/*******************************************************************************
 * BWA function
 */

/* Runs the Baum-Welch Algorithm on the supplied HMM and observation sequence */
float run_hmm_bwa(  Hmm *hmm,
                    Obs *in_obs,
                    int iterations,
                    float threshold)
{

    /* Host-side variables */
    float *a;
    float *b;
    float *pi;
    int threads_per_block;
    int nblocks;
    int size;
    float new_log_lik;
    float old_log_lik = 0;
    int iter;

    /* Initialize HMM values */
    a = hmm->a;
    b = hmm->b;
    pi = hmm->pi;
    nsymbols = hmm->nsymbols;
    nstates = hmm->nstates;
    obs = in_obs->data;
    length = in_obs->length;

    /* Initialize CUBLAS */
    cublas_status = cublasInit();
    if (cublas_status != CUBLAS_STATUS_SUCCESS) {
        fprintf (stderr, "ERROR: CUBLAS Initialization failure\n");
        return EXIT_ERROR;
    }

    /* Allocate host memory */
    scale = (float *) malloc(sizeof(float) * length);
    if (scale == 0) {
        fprintf (stderr, "ERROR: Host memory allocation error (scale)\n");
        return EXIT_ERROR;
    }

    /* Allocate device memory */
    size = sizeof(float) * nstates * nstates;
    CUDA_SAFE_CALL( cudaMalloc((void**)&a_d, size) );
    CUDA_SAFE_CALL( cudaMemcpy(a_d, a, size, cudaMemcpyHostToDevice) );
    size = sizeof(float) * nstates * nsymbols;
    CUDA_SAFE_CALL( cudaMalloc((void**)&b_d, size) );
    CUDA_SAFE_CALL( cudaMemcpy(b_d, b, size, cudaMemcpyHostToDevice) );
    size = sizeof(float) * nstates;
    CUDA_SAFE_CALL( cudaMalloc((void**)&pi_d, size) );
    CUDA_SAFE_CALL( cudaMemcpy(pi_d, pi, size, cudaMemcpyHostToDevice) );
    size = sizeof(float) * nstates * length;
    CUDA_SAFE_CALL( cudaMalloc((void**)&alpha_d, size) );
    size = sizeof(float) * nstates * length;
    CUDA_SAFE_CALL( cudaMalloc((void**)&beta_d, size) );
    size = sizeof(float) * nstates;
    CUDA_SAFE_CALL( cudaMalloc((void**)&gamma_sum_d, size) );
    size = sizeof(float) * nstates * nstates;
    CUDA_SAFE_CALL( cudaMalloc((void**)&xi_sum_d, size) );
    size = sizeof(float) * nstates;
    CUDA_SAFE_CALL( cudaMalloc((void**)&c_d, size) );
    size = sizeof(float) * nstates;
    CUDA_SAFE_CALL( cudaMalloc((void**)&ones_n_d, size) );
    size = sizeof(float) * nsymbols;
    CUDA_SAFE_CALL( cudaMalloc((void**)&ones_s_d, size) );

    /* Initialize ones array */
    threads_per_block = MAX_THREADS_PER_BLOCK;
    nblocks = (nstates + threads_per_block - 1) / threads_per_block;
    init_ones_dev<<<nblocks, threads_per_block>>>(ones_s_d, nsymbols);

    /* Run BWA for either max iterations or until threshold is reached */
    for (iter = 0; iter < iterations; iter++) {

        new_log_lik = calc_alpha();
        if (new_log_lik == EXIT_ERROR) {
            return EXIT_ERROR;
        }

        if (calc_beta() == EXIT_ERROR) {
            return EXIT_ERROR;
        }

        calc_gamma_sum();

        if (calc_xi_sum() == EXIT_ERROR) {
            return EXIT_ERROR;
        }

        if (estimate_a() == EXIT_ERROR) {
            return EXIT_ERROR;
        }

        if (estimate_b() == EXIT_ERROR) {
            return EXIT_ERROR;
        }

        if (estimate_pi() == EXIT_ERROR) {
            return EXIT_ERROR;
        }

        /* check log_lik vs. threshold */
        if (threshold > 0 && iter > 0) {
            if (fabs(pow(10,new_log_lik) - pow(10,old_log_lik)) < threshold) {
                break;
            }
        }

        old_log_lik = new_log_lik;

    }

    /* Copy device variables back to host */
    size = sizeof(float) * nstates * nstates;
    CUDA_SAFE_CALL( cudaMemcpy(a, a_d, size, cudaMemcpyDeviceToHost) );
    size = sizeof(float) * nstates * nsymbols;
    CUDA_SAFE_CALL( cudaMemcpy(b, b_d, size, cudaMemcpyDeviceToHost) );
    size = sizeof(float) * nstates;
    CUDA_SAFE_CALL( cudaMemcpy(pi, pi_d, size, cudaMemcpyDeviceToHost) );

    /* Shutdown CUBLAS */
    if(cublasShutdown() != CUBLAS_STATUS_SUCCESS) {
        printf("ERROR: CUBLAS Shutdown failure\n");
        return EXIT_ERROR;
    }

    /* Free memory */
    free(scale);
    CUDA_SAFE_CALL( cudaFree(a_d) );
    CUDA_SAFE_CALL( cudaFree(b_d) );
    CUDA_SAFE_CALL( cudaFree(pi_d) );
    CUDA_SAFE_CALL( cudaFree(alpha_d) );
    CUDA_SAFE_CALL( cudaFree(beta_d) );
    CUDA_SAFE_CALL( cudaFree(gamma_sum_d) );
    CUDA_SAFE_CALL( cudaFree(xi_sum_d) );
    CUDA_SAFE_CALL( cudaFree(c_d) );
    CUDA_SAFE_CALL( cudaFree(ones_n_d) );
    CUDA_SAFE_CALL( cudaFree(ones_s_d) );

    return new_log_lik;
}

/*******************************************************************************
 * Supporting functions
 */

/* Calculates the forward variables (alpha) for an HMM and obs. sequence */
float calc_alpha()
{

    int threads_per_block;
    int nblocks;
    int offset_cur;
    int offset_prev;
    float log_lik;
    int t;

    /* Initialize alpha variables */
    threads_per_block = MAX_THREADS_PER_BLOCK;
    nblocks = (nstates + threads_per_block - 1) / threads_per_block;
    init_alpha_dev<<<nblocks, threads_per_block>>>( b_d,
                                                     pi_d,
                                                     nstates,
                                                     alpha_d,
                                                     ones_n_d,
                                                     obs[0]);

    /* Sum alpha values to get scaling factor */
    cublasGetError();
    scale[0] = cublasSdot(nstates, alpha_d, 1, ones_n_d, 1);
    cublas_status = cublasGetError();
    if (cublas_status != CUBLAS_STATUS_SUCCESS) {
        fprintf (stderr, "ERROR: Kernel execution error\n");
        return EXIT_ERROR;
    }

    /* Scale alpha values */
    scale_alpha_dev<<<nblocks, threads_per_block>>>(    nstates,
                                                         alpha_d,
                                                         scale[0]);

    /* Initilialize log likelihood */
    log_lik = log10(scale[0]);

    /* Calculate the rest of the alpha variables */
    for (t = 1; t < length; t++) {
        /* Calculate offsets */
        offset_prev = (t - 1) * nstates;
        offset_cur = t * nstates;

        /* Multiply transposed A matrix by alpha(t-1) */
        /* Note: the matrix is auto-transposed by cublas reading column-major */
        cublasSgemv( 'N', nstates, nstates, 1.0f, a_d, nstates,
                    alpha_d + offset_prev, 1, 0, alpha_d + offset_cur, 1 );
        cublas_status = cublasGetError();
        if (cublas_status != CUBLAS_STATUS_SUCCESS) {
            fprintf (stderr, "ERROR: Kernel execution error\n");
            return EXIT_ERROR;
        }

        /* Calculate alpha(t) */
        calc_alpha_dev<<<nblocks, threads_per_block>>>( nstates,
                                                        alpha_d + offset_cur,
                                                        b_d,
                                                        obs[t]);

        /* Sum alpha values to get scaling factor */
        scale[t] = cublasSdot(nstates, alpha_d + offset_cur, 1, ones_n_d, 1);
        cublas_status = cublasGetError();
        if (cublas_status != CUBLAS_STATUS_SUCCESS) {
            fprintf (stderr, "ERROR: Kernel execution error\n");
            return EXIT_ERROR;
        }

        /* Scale alpha values */
        scale_alpha_dev<<<nblocks, threads_per_block>>>(nstates,
                                                        alpha_d + offset_cur,
                                                        scale[t]);

        /* Update log likelihood */
        log_lik += log10(scale[t]);
    }

    return log_lik;
}

/* Calculates the backward variables (beta) */
int calc_beta()
{

    int threads_per_block;
    int nblocks;
    int t;

    /* Initialize beta variables */
    threads_per_block = MAX_THREADS_PER_BLOCK;
    nblocks = (nstates + threads_per_block - 1) / threads_per_block;
    init_beta_dev<<<nblocks, threads_per_block>>>( nstates, beta_d +
                                                    ((length - 1) * nstates),
                                                    scale[length - 1]);

    /* Calculate the rest of the beta variables */
    for (t = length - 2; t >= 0; t--) {

        /* Calculate first step of beta: B.*beta/scale */
        calc_beta_dev<<<nblocks, threads_per_block>>>(  beta_d,
                                                        b_d,
                                                        scale[t],
                                                        nstates,
                                                        obs[t+1],
                                                        t);

        /* Multiply transposed A matrix by beta(t) */
        cublasSgemv( 'T', nstates, nstates, 1.0f, a_d, nstates,
                    beta_d + (t * nstates), 1, 0, beta_d + (t * nstates), 1 );
        cublas_status = cublasGetError();
        if (cublas_status != CUBLAS_STATUS_SUCCESS) {
            fprintf (stderr, "ERROR: Kernel execution error\n");
            return EXIT_ERROR;
        }

    }

    return 0;
}

/* Calculates the gamma sum */
void calc_gamma_sum()
{
    int threads_per_block;
    int nblocks;
    int size;
    int t;

    threads_per_block = MAX_THREADS_PER_BLOCK;
    nblocks = (nstates + threads_per_block - 1) / threads_per_block;

    size = sizeof(float) * nstates;
    CUDA_SAFE_CALL( cudaMemset(gamma_sum_d, 0, size) );

    /* Find sum of gamma variables */
    for (t = 0; t < length; t++) {
        calc_gamma_dev<<<nblocks, threads_per_block>>>( gamma_sum_d,
                                                        alpha_d,
                                                        beta_d,
                                                        nstates,
                                                        t);
    }

}

/* Calculates the sum of xi variables */
int calc_xi_sum()
{
    float sum_ab;
    int nblocks;
    int size;
    int t;

    size = sizeof(float) * nstates * nstates;
    CUDA_SAFE_CALL( cudaMemset(xi_sum_d, 0, size) );

    /* Calculate running xi sum */
    dim3 threads(BLOCK_DIM, BLOCK_DIM);
    nblocks = (nstates + BLOCK_DIM - 1) / BLOCK_DIM;
    dim3 grid(nblocks, nblocks);

    /* Find the sum of xi variables */
    for (t = 0; t < length - 1; t++) {

        /* Calculate denominator */
        sum_ab = cublasSdot(nstates, alpha_d + (t * nstates), 1,
                            beta_d + (t * nstates), 1);
        cublas_status = cublasGetError();
        if (cublas_status != CUBLAS_STATUS_SUCCESS) {
            fprintf (stderr, "ERROR: Kernel execution error\n");
            return EXIT_ERROR;
        }

        /* Calculate xi sum */
        calc_xi_dev<<<grid, threads>>>( xi_sum_d,
                                        a_d,
                                        b_d,
                                        alpha_d,
                                        beta_d,
                                        sum_ab,
                                          nstates,
                                          obs[t+1],
                                          t);
    }

    return 0;
}

/* Re-estimates the state transition probabilities (A) */
int estimate_a()
{
    float sum_ab;
    int nblocks;

    /* Calculate running xi sum */
    dim3 threads(BLOCK_DIM, BLOCK_DIM);
    nblocks = (nstates + BLOCK_DIM - 1) / BLOCK_DIM;
    dim3 grid(nblocks, nblocks);

    /* Calculate denominator */
    sum_ab = cublasSdot(nstates, alpha_d + ((length - 1) * nstates), 1,
                                beta_d + ((length - 1) * nstates), 1);
    cublas_status = cublasGetError();
    if (cublas_status != CUBLAS_STATUS_SUCCESS) {
        fprintf (stderr, "ERROR: Kernel execution error\n");
        return EXIT_ERROR;
    }

    /* Calculate new value of A */
    est_a_dev<<<grid, threads>>>(   a_d,
                                    alpha_d,
                                    beta_d,
                                    xi_sum_d,
                                    gamma_sum_d,
                                    sum_ab,
                                    nstates,
                                    length);

    /* Sum rows of A to get scaling values */
    cublasSgemv( 'T', nstates, nstates, 1.0f, a_d, nstates,
                ones_n_d, 1, 0, c_d, 1 );
    cublas_status = cublasGetError();
    if (cublas_status != CUBLAS_STATUS_SUCCESS) {
        fprintf (stderr, "ERROR: Kernel execution error\n");
        return EXIT_ERROR;
    }

    /* Normalize A matrix */
    scale_a_dev<<<grid, threads>>>( a_d,
                                    c_d,
                                    nstates);

    return 0;
}

/* Re-estimates the output symbol probabilities (B) */
int estimate_b()
{

    float sum_ab;
    int size;
    int t;

    size = sizeof(float) * nstates * nsymbols;
    CUDA_SAFE_CALL( cudaMemset(b_d, 0, size) );

    /* Calculate number of threads and blocks needed */
    dim3 threads(BLOCK_DIM, BLOCK_DIM);
    dim3 grid(  (nstates + threads.x - 1) / threads.x,
                (nsymbols + threads.y - 1) / threads.y);
    for (t = 0; t < length; t++) {

        /* Calculate denominator */
        sum_ab = cublasSdot(nstates, alpha_d + (t * nstates), 1,
                                     beta_d + (t * nstates), 1);
        cublas_status = cublasGetError();
        if (cublas_status != CUBLAS_STATUS_SUCCESS) {
            fprintf (stderr, "ERROR: Kernel execution error\n");
            return EXIT_ERROR;
        }

        /* Accumulate B values */
        acc_b_dev<<<grid, threads>>>(   b_d,
                                        alpha_d,
                                        beta_d,
                                        sum_ab,
                                        nstates,
                                        nsymbols,
                                        obs[t],
                                        t);

    }

    /* Re-estimate B values */
    est_b_dev<<<grid, threads>>>(b_d, gamma_sum_d, nstates, nsymbols);

    /* Sum rows of B to get scaling values */
    cublasSgemv( 'N', nstates, nsymbols, 1.0f, b_d, nstates,
                ones_s_d, 1, 0, c_d, 1 );
    cublas_status = cublasGetError();
    if (cublas_status != CUBLAS_STATUS_SUCCESS) {
        fprintf (stderr, "ERROR: Kernel execution error\n");
        return EXIT_ERROR;
    }

    /* Normalize B matrix */
    scale_b_dev<<<grid, threads>>>( b_d,
                                    c_d,
                                    nstates,
                                    nsymbols);

    return 0;
}

/* Re-estimates the initial state probabilities (Pi) */
int estimate_pi()
{

    float sum_ab;
    int threads_per_block;
    int nblocks;

    /* Calculate denominator */
    sum_ab = cublasSdot(nstates, alpha_d, 1, beta_d, 1);

    /* Estimate Pi values */
    threads_per_block = MAX_THREADS_PER_BLOCK;
    nblocks = (nstates + threads_per_block - 1) / threads_per_block;
    est_pi_dev<<<nblocks, threads_per_block>>>( pi_d,
                                                alpha_d,
                                                beta_d,
                                                sum_ab,
                                                nstates);

    return 0;
}
```

Appendix D - Parallel Forward Algorithms in CUDA

```
/*
 * Copyright (c) 2011, Shawn Hymel <hymelsr@vt.edu>
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without
 * restriction, including without limitation the rights to use, copy,
 * modify, merge, publish, distribute, sublicense, and/or sell copies
 * of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
 * NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
 * BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
 * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 *
 */

/**
 * @file
 * @author  Shawn Hymel
 * @date    October 24, 2011
 * @brief   Implements the Forward Algorithm (FO) in CUDA
 */

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

#include <cuda.h>
#include <cutil.h>

#include "hmm.h"
extern "C" {                        /* required for functions to be visible in C */
#include "hmms_fo_cuda.h"
}

/* Contains information about the graphics card's CUDA capabilities */
enum {
    MAX_STATES = 16                 /* Static depending on gfx card capabilities*/
};

/*******************************************************************************/
/* CUDA - Run Forward Algorithm on multiple models */
__global__ void calc_alpha_dev( float *a_d,
                                float *b_d,
                                float *pi_d,
                                int nstates,
                                int nsymbols,
                                int length,
                                int nmodels,
                                int *obs_d,
                                float *alpha_d,
                                float *scale_d,
                                float *log_liks_d)
{

    unsigned int tdx = threadIdx.x;
    unsigned int tdy = threadIdx.y;
    unsigned int tid = (tdy * blockDim.x) + tdx;
    unsigned int bid = blockIdx.x;
    unsigned int stride;
    int obs_t;
    int t;
    __shared__ float partial_sum[MAX_STATES*MAX_STATES];

    /* Initialize alpha */
    if ((tid < nstates) && (bid < nmodels)) {
        obs_t = obs_d[0];
        alpha_d[(bid*nstates*length) + tid] = pi_d[(bid*nstates) + tid] *
                        b_d[(bid*nstates*nsymbols) + (obs_t*nstates) + tid];
    }

    /* Calculate scaling factor and initialize log likelihoods */
    if ((tid < nstates) && (bid < nmodels)) {
        partial_sum[tid] = alpha_d[(bid*nstates*length) + tid];
    } else {
        partial_sum[tid] = 0;
    }
    __syncthreads();
    for (stride = blockDim.x/2; stride > 0; stride /= 2) {
        if ((tid < stride) && (bid < nmodels)) {
            partial_sum[tid] += partial_sum[tid + stride];
        }
        __syncthreads();
    }
    if ((tid == 0) && (bid < nmodels)) {
        scale_d[bid] = partial_sum[0];
        log_liks_d[bid] = log10(scale_d[bid]);
    }

    /* Scale initial alpha */
    if ((tid < nstates) && (bid < nmodels)) {
        alpha_d[(bid*nstates*length) + tid] =
                            alpha_d[(bid*nstates*length) + tid] / scale_d[bid];
    }

    /* Induction step */
    for (t = 1; t < length; t++) {

        obs_t = obs_d[t];

        /* Calculate intermediate step in matrix multiplication */
        if ((tdx < nstates) && (tdy < nstates) && (bid < nmodels)) {
            partial_sum[tid] =
                    alpha_d[(bid*nstates*length) + ((t-1)*nstates) + tdy] *
                    a_d[(bid*nstates*nstates) + (tdy*nstates) + tdx] *
                    b_d[(bid*nstates*nsymbols) + (obs_t*nstates) + tdx];
        } else {
            partial_sum[tid] = 0;
        }

        /* Sum up the columns to complete matrix multiplication */
        __syncthreads();
        for (stride = blockDim.y/2; stride > 0; stride /= 2) {
            if ((tdy < stride) && (bid < nmodels)) {
                partial_sum[tid] = partial_sum[tid] +
                                   partial_sum[tid + (stride*blockDim.x)];
            }
            __syncthreads();
        }

        /* Copy to alpha */
        if ((tid < nstates) && (bid < nmodels)) {
            alpha_d[(bid*nstates*length) + (t*nstates) + tid] =
                                                    partial_sum[tid];
        }

        /* Calculate scaling factor */
        if ((tid < nstates) && (bid < nmodels)) {
            partial_sum[tid] = alpha_d[(bid*nstates*length) +
                                                        (t*nstates) + tid];
        } else {
            partial_sum[tid] = 0;
        }
        __syncthreads();
        for (stride = blockDim.x/2; stride > 0; stride /= 2) {
            if ((tid < stride) && (bid < nmodels)) {
                partial_sum[tid] += partial_sum[tid + stride];
            }
            __syncthreads();
        }
        if ((tid == 0) && (bid < nmodels)) {
            scale_d[(t*nmodels) + bid] = partial_sum[0];
            log_liks_d[bid] += log10(scale_d[(t*nmodels) + bid]);
        }

        /* Scale alpha values */
        if ((tid < nstates) && (bid < nmodels)) {
            alpha_d[(bid*nstates*length) + (t*nstates) + tid] =
                    alpha_d[(bid*nstates*length) + (t*nstates) + tid] /
                    scale_d[(t*nmodels) + bid];
        }
    }
}


/******************************************************************************/
/* Runs the forward algorithm on the supplied HMM and observation sequence */
void run_hmms_fo(Hmm **in_hmm, Obs *in_obs, int nstates, int nsymbols,
                                                int nmodels, float log_liks[])
{

    /* Host-side variables */
    int size;
    int h;
    float *alpha;
    int *obs = in_obs->data;
    int length = in_obs->length;

    /* Device-side variables */
    float *a_d;
    float *b_d;
    float *pi_d;
    int *obs_d;
    float *alpha_d;
    float *scale_d;
    float *log_liks_d;

    alpha = (float *) malloc(sizeof(float) * nstates * length * nmodels);

    /* Check for max number of states */
    if (nstates > MAX_STATES) {
        printf("Maximum number of states exceeded.\n");
        return;
    }

    /* Allocate memory and copy needed parameters to device */
    size = sizeof(float) * nstates * nstates * nmodels;
    CUDA_SAFE_CALL( cudaMalloc((void**)&a_d, size) );
    size = sizeof(float) * nstates * nstates;
    for (h = 0; h < nmodels; h++) {
        CUDA_SAFE_CALL( cudaMemcpy(     a_d + h*nstates*nstates,
                                        in_hmm[h]->a,
                                        size,
                                        cudaMemcpyHostToDevice) );
    }
    size = sizeof(float) * nstates * nsymbols * nmodels;
    CUDA_SAFE_CALL( cudaMalloc((void**)&b_d, size) );
    size = sizeof(float) * nstates * nsymbols;
    for (h = 0; h < nmodels; h++) {
        CUDA_SAFE_CALL( cudaMemcpy(     b_d + h*nstates*nsymbols,
                                    in_hmm[h]->b,
                                    size,
                                    cudaMemcpyHostToDevice) );
    }
    size = sizeof(float) * nstates * nmodels;
    CUDA_SAFE_CALL( cudaMalloc((void**)&pi_d, size) );
    size = sizeof(float) * nstates;
    for (h = 0; h < nmodels; h++) {
        CUDA_SAFE_CALL( cudaMemcpy(     pi_d + h*nstates,
                                    in_hmm[h]->pi,
                                    size,
                                    cudaMemcpyHostToDevice) );
    }
    size = sizeof(int) * length;
    CUDA_SAFE_CALL( cudaMalloc((void**)&obs_d, size) );
    CUDA_SAFE_CALL( cudaMemcpy(obs_d, obs, size, cudaMemcpyHostToDevice) );
    size = sizeof(float) * nstates * length * nmodels;
    CUDA_SAFE_CALL( cudaMalloc((void**)&alpha_d, size) );
    size = sizeof(float) * length * nmodels;
    CUDA_SAFE_CALL( cudaMalloc((void**)&scale_d, size) );
    size = sizeof(float) * nmodels;
    CUDA_SAFE_CALL( cudaMalloc((void**)&log_liks_d, size) );


    /* Run Forward Algorithm in CUDA */
    dim3 blocks( nmodels );
    dim3 threadsPerBlock(MAX_STATES, MAX_STATES);
    calc_alpha_dev<<<blocks, threadsPerBlock>>>(    a_d,
                                                    b_d,
                                                    pi_d,
                                                    nstates,
                                                    nsymbols,
                                                    length,
                                                    nmodels,
                                                    obs_d,
                                                    alpha_d,
                                                    scale_d,
                                                    log_liks_d);

    /* Copy results from device to host */
    size = sizeof(float) * nmodels;
    CUDA_SAFE_CALL( cudaMemcpy(log_liks, log_liks_d, size,
                                        cudaMemcpyDeviceToHost) );

    /* Free device memory */
    CUDA_SAFE_CALL( cudaFree(a_d) );
    CUDA_SAFE_CALL( cudaFree(b_d) );
    CUDA_SAFE_CALL( cudaFree(pi_d) );
    CUDA_SAFE_CALL( cudaFree(obs_d) );
    CUDA_SAFE_CALL( cudaFree(alpha_d) );
    CUDA_SAFE_CALL( cudaFree(scale_d) );
    CUDA_SAFE_CALL( cudaFree(log_liks_d) );

    /* Free allocated variables */
    free(alpha);

}
```