ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

THESIS PRESENTED TO
ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
Ph.D.

BY
Patrick CARDINAL

SPEECH RECOGNITION ON MULTI-CORE PROCESSORS AND GPUS

MONTREAL, JULY 3, 2013

BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS:

Mr. Pierre Dumouchel, Thesis Director
Génie logiciel et technologie de l'information, ETS

Mr. Tony Wong, Committee President
Génie de la production automatisée, ETS

Mr. Douglas O'Shaughnessy, External Examiner
Centre Énergie Matériaux Télécommunication, INRS

Mr. Mohammed Cheriet, Examiner
Génie de la production automatisée, ETS

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND PUBLIC

ON JUNE 6, 2013

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

ACKNOWLEDGEMENTS

I would like to thank my supervisor Pierre Dumouchel for his guidance and encouragement throughout this work. This has been greatly appreciated.

I also truly thank Gilles Boulianne, the director of the speech recognition group at CRIM, for constructive discussions about this work and other topics. CRIM has provided me with a wonderful research environment.

I also truly thank Mohammed Cheriet, Douglas O'Shaughnessy and Tony Wong for agreeing to evaluate the work of this thesis.

A special thanks to Michel Comeau for his help in the correction of this thesis and for comments.

I also wish to thank all members of the speech recognition group at CRIM and more particularly to Vishwa Gupta for having included me in the copy detection project.

Special thanks to my friends Najim Dehak, Marc Boulé, Francis Cardinal and my girlfriend Renata Podbielski for their encouragement and comments throughout this work.

SPEECH RECOGNITION ON MULTI-CORE PROCESSORS AND GPUS

Patrick CARDINAL

ABSTRACT

The speed of processors has remained stable over the past few years. The trend may even be towards slower speeds in order to satisfy the ever increasing demands of energy efficiency. This tendency is already apparent in the area of mobile devices. In order to take full advantage of the processing power offered by modern and future processors, applications must integrate parallelism and speech recognition is no exception.

The classic decoding algorithm of Viterbi, a dynamic programming approach for searching in the recognition network, does not make full use of this power. The main reason being that the algorithm searches through a knowledge graph containing millions of nodes and transitions. In practice, a thorough search through such an enormous network is unfeasible. As a result, the graph is pruned so as to retain the most promising hypotheses only. The pruning process is however connected with a misuse of the memory architecture of Intel-based computers. To overcome this problem, another search algorithm is proposed: the A* search. This type of search makes use of a heuristic that provides an approximation of the distance for reaching the final node. A good heuristic results in a negligible number of nodes having to be explored, allowing to transfer the computational load of the network search towards the computation of the heuristic, so designed to make optimal use of modern processor architectures. The heuristic represents a much smaller knowledge graph for speech recognition. Because of its small size, the graph can be exhaustively explored thus eliminating the problems relating to memory architecture mismanagement.

Acoustic model computations represent an important component of speech recognition. For this task, a 3.6x speed increase was achieved on a quad core processor with respect to the single core version. On GPU, the acceleration is 24.8x with respect to the sequential version. In regards to the recognition network search, the A* algorithm is shown to explore 28 times less nodes than the sequential version of the original algorithm. In addition, the heuristic computation is 4.1 and 10.1 times faster on a quad core and GPU than the sequential version respectively. Overall, the new parallelized version offers a 4% absolute increase in real-time recognition accuracy compared to the classic version.

Keywords: Speech recognition, parallel computing, multi-core processor, GPU, A* search

LA RECONNAISSANCE DE LA PAROLE SUR LES PROCESSEURS MULTI-COEURS ET GPUS

Patrick CARDINAL

RÉSUMÉ

Depuis plusieurs années, la vitesse des processeurs demeure stable. La tendance semble maintenant être à la diminution de la vitesse afin de réduire la consommation d'énergie. Cette tendance est déjà visible dans le monde des appareils mobiles. Pour profiter de toute la puissance de calcul des processeurs modernes et à venir, les applications se doivent d'intégrer le parallélisme et la reconnaissance de la parole ne fait pas exception.

Malheureusement, l'algorithme de décodage (Viterbi), qui utilise la programmation dynamique pour la recherche dans le graphe de reconnaissance, n'arrive pas à utiliser pleinement toute cette puissance. La raison principale est que ce graphe de reconnaissance contient plusieurs millions de noeuds et de transitions, il est donc impensable l'explorer exhaustivement et doit être élagué afin d'explorer seulement les hypothèses les plus prometteuses. Cet élagage fait en sorte que l'architecture de la mémoire utilisée dans les ordinateurs de type Intel n'est pas utilisée de manière efficace. Pour contourner le problème, un autre type d'algorithme de recherche est envisagée: la recherche A*. Ce type de recherche utilise une heuristique qui donne une approximation de la distance à parcourir pour atteindre le noeud final. La proposition d'une bonne heuristique fait en sorte que le nombre de noeuds explorés devient négligeable, ce qui a pour effet de transférer le temps de calcul de la recherche dans le graphe au calcul de l'heuristique, qui peut être conçu afin de profiter au maximum de l'architecture des processeurs actuels. Pour la reconnaissance de la parole, un graphe de reconnaissance beaucoup plus petit est utilisé comme heuristique pouvant ainsi être explorer exhaustivement, ce qui permet d'éliminer les problèmes de mauvaise utilisation de l'architecture mémoire.

Un aspect important pour la reconnaissance de la parole est le calcul acoustique. Pour cette tâche, une accélération par un facteur de 3,6 a été observée sur un processeur à 4 coeurs. Sur GPU, l'accélération est de 24,8x par rapport à l'algorithme de Viterbi. En ce qui concerne la recherche dans le graphe de reconnaissance, les résultats ont montré que le nombre de noeuds explorés par l'algorithme A* est 28 fois inférieur comparé à sa version séquentielle à l'algorithme originale. De plus, le calcul de l'heuristique est respectivement 4,1 et 10,1 fois plus rapide sur un processeur à 4 coeurs et sur GPU par rapport à la version séquentielle. Finalement, si on compare la version originale et la nouvelle version parallélisée du point de vue du taux de reconnaissance au temps réel, la version parallèle a un taux de reconnaissance supérieure de 4% absolu par rapport à la version classique.

Mots-clés: reconnaissance de la parole, processeur multi-coeurs, GPU, recherche A*

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Page

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS

ASR	Automatic Speech Recognition
ASIC	Application Specific Integrated Circuit
CPU	Central Processor Unit
CUDA	Compute Unified Device Architecture
DARPA	Defense Advanced Research Projects Agency
DBN	Deep Belief Network
ETS	École de Technologie Supérieure
FPGA	Field-Programmable Gate Array
FST	Finite State Transducer
GMM	Gaussian Mixture Model
GPU	Graphic Processor Unit
HMM	Hidden Markov Model
LPC	Linear Predictive Coding
MFCC	Mel Frequency Cepstral Coefficients
NMOS	N-type Metal-Oxide-Semiconductor
NN	Neural Network
PCI	Peripheral Component Interconnect
PDF	Probability Density Function
PMOS	P-type Metal-Oxide-Semiconductor
QPI	QuickPath Interconnect

RAM	Random Access Memory
SIMD	Single Instruction, Multiple Data
SSE	Streaming SIMD Extensions
SVM	Support Vector Machine
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuits
WER	Word Error Rate
WFST	Weighted Finite State Transducer

INTRODUCTION

Speech recognition is the process that allows a machine to identify words and phrases of spoken languages. This is a complex task that researchers have been working on for more than five decades (Juang and Rabiner (2004)).

One of the first speech recognition systems was built by Davis *et al.* of Bell Laboratories (Davis *et al.* (1952)). This system was dedicated to the recognition of isolated spoken digits from a single speaker. The circuit built for this task had to be adapted for each speaker.

In 1956, Olsen and Belar from RCA Laboratories developed a system that was able to recognize 10 syllables from a single speaker (Olsen and Belar (1956)). A few years later, Forgie and Forgie of MIT Lincoln Lab built a similar system that was speaker-independent (Forgie and Forgie (1959)).

In the 1960's, Sakai and Doshita (Sakai and Doshita (1962)) built a phoneme recognizer which had the particularity of using a segmenter allowing the analysis and recognition of different portions of the spoken phrases. This is considered as the first continuous speech recognition system (Juang and Rabiner (2004)). Continuous speech recognition involves the recognition of the fundamental units of natural speech (word, digit, phone, ...) from a single recording.

The use of statistical information has been introduced by Fry and Denes from University College in England. They built a recognizer that was able to recognize 4 vowels and 9 consonants (Fry and Denes (1959)). They used a statistical model to constrain the phoneme sequences to those allowable in English. This approach allowed to improve the recognition accuracy.

In 1972, Atal and Hanauer proposed a way of representing the speech waveform in terms of time-varying parameters related to the transfer function of the vocal tract (Atal and Hanauer (1971)). Linear Predictive Coding (LPC) represents the spectral envelope with a small number of parameters based on a predictive model. This allowed the use of pattern recognition techniques in speech recognition, an example being the work of Rabiner *et al.* (Rabiner *et al.* (1979)).

Another way of extracting parameters from the speech signal has been introduced by Merlmelstein and Davis (Mermelstein (1976),Davis and Mermelstein (1980)). The Mel-Frequency Cepstral Coefficients (MFCCs) have been developed to approximate the human auditory system's response. Both LPC and MFCCs are still used in state-of-the-art speech recognition systems.

In the mid-1970s, several researchers began to use Hidden Markov Models in speech recognition (Jelinek *et al.* (1975),Baker (1975)). The HMM models the intrinsic variability of the speech signal as well as the structure of spoken language in a consistent statistical modeling framework. This approach has been a major step forward from the simple pattern recognition and acoustic-phonetic methods used in earlier speech recognition systems (Juang and Rabiner (2004)).

The increase of computational resources combined with technological advances have led to continuous improvements in speech recognition systems. The Defense Advanced Research Projects Agency (DARPA) has organized evaluations of speech recognition systems with progessive degrees of difficulties. The tasks and systems' improvements over the years are depicted in Figure 0.1.

Evaluations were first performed on read speech. The first evaluation was the DARPA resource management task with a 1000 word vocabulary. In 1989, Lee *et al.* achieved an accuracy up to 96% on this task (Lee *et al.* (1989)). At the beginning of the 1990's, the Wall Street Journal database was introduced. This consisted of recorded dictations of Wall Street Journal articles by various speakers. In a first study, evaluations were performed on a 5K word vocabulary. The second version of the database consisted of a 20K word vocabulary. In 1994, the accuracy on this task was 89.2% (Gauvain *et al.* (1994)).

Afterwards, a more difficult task was proposed : that of recognizing spontaneous speech. The corresponding database was made up of recordings from telephone conversations. The accuracy result of 62.6% reported by Zeppenfeld *et al.* reflects the difficulty of this task (Zeppenfeld

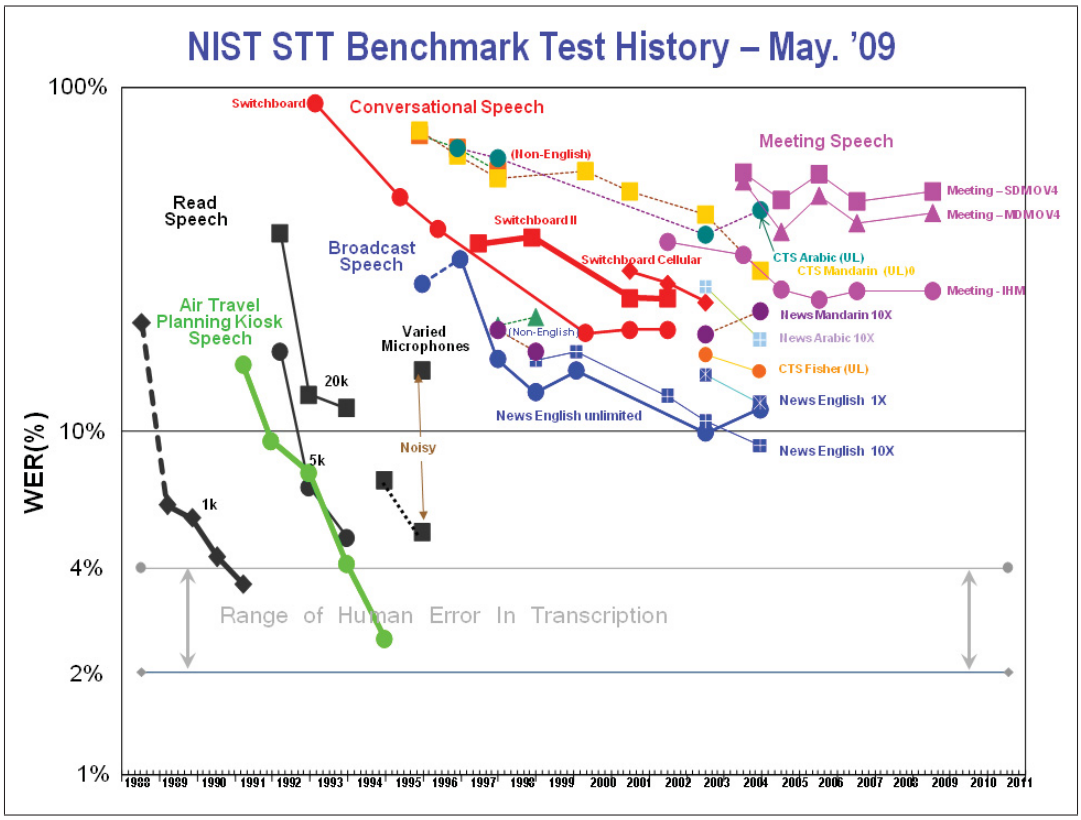

Figure 0.1 NIST STT benchmark test history
Source: http://www.itl.nist.gov/iad/mig/publications/ASRhistory/.

et al. (1997)). The following task was aimed at recognizing meeting transcriptions, made up of multi-speaker audio recordings that at times include several speakers talking simultaneously.

A great deal of work continues to be pursued worldwide for developing technologies aimed at improving the accuracy of speech recognition systems. The task aimed by this work is the large vocabulary speech recognition of spontaneous speech with several speakers in a possibly noisy environment. Several very interesting real-world applications could be developped if speech recognition systems were efficient in these conditions. For example, automatic closed-captioning of live tv shows could take advantage of advances in this task. Currently, re-speakers are needed to ensure a good accuracy. A more robust system could be used to produce closed-captions automaticaly without any human intervation. A more robust system usually requires more complex models, which need more computational power.

This work specifically explores how current processors can be used to improve speech recognition systems. Indeed, a few years ago, a new processor model meant faster applications since processor speeds increased at the same rate as the integration capacity, which followed faithfully Moore's law as shown in figure 0.2. This "law" states that the number of transistors in integrated circuits doubles approximately every 2 years, which leads to an increasing number of computation cores in processors. While this rule still applies, the speed of processors has stagnated in recent years. Since the current trend is to reduce energy consumption, processors could become even slower in the years to come.

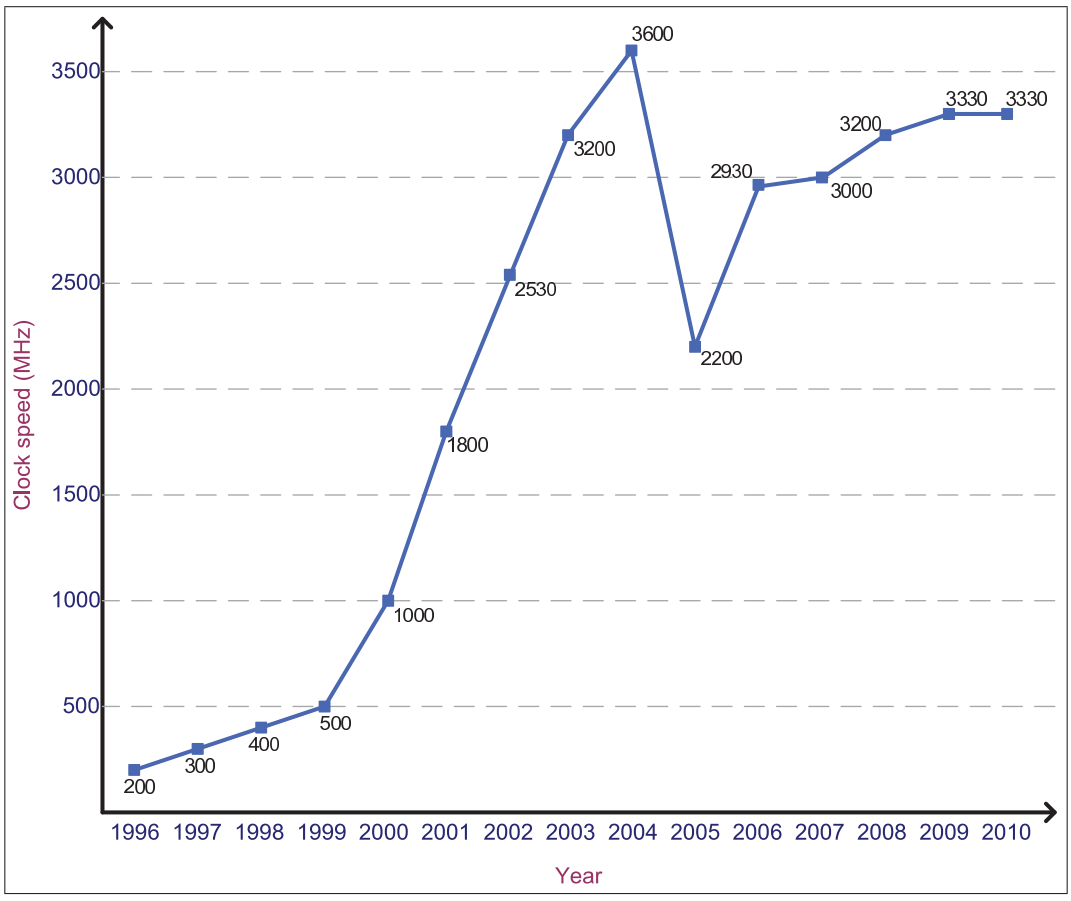

Figure 0.2 Average processor speed over recent years
Source: Mah and Castle (2010).

In addition to the main processor, almost every modern-day computer contains a graphic card that incorporates a specialized processor called Graphics Processing Unit (GPU). A GPU is

mainly a Single Instruction, Multiple Data (SIMD) parallel processor that is computationally powerful, while being quite affordable.

Over the last years, GPUs have evolved into flexible processors. A noteworthy technological advance was achieved in 2007, when NVidia and ATI introduced a unified architecture that eliminated the graphical pipeline. This greatly enhanced the flexibility and usability of the GPU, to the extent that it is becoming a mainstream alternative for general purpose calculations.

Taking advantage of the processing power offered by modern processors implementing multi-core technology and/or GPU necessarily involves the parallelization of sequential algorithms. Most speech recognizers run under a sequential implementation that cannot take advantage of this technology.

The speech recognition task

There are two main time consuming tasks involved in automatic speech recognition. The first one is the computation of acoustic likelihoods, which takes up 30% to 70% of the total time, depending on the application. When Gaussian Mixture Models (GMMs), in combination with Hidden Markov Models (HMMs) are used, as is the case in state-of-the-art speech recognition systems, this computation involves mostly arithmetic operations that incorporate the dot product. Under these circumstances, the computation can be efficiently implemented on SIMD parallel architectures. For example, SSE (Streaming SIMD Extensions) registers are available on every Intel architecture. Another example of a SIMD architecture is a GPU, which is available in almost all computers.

The second major task is the recognition network search that consumes most of the remaining time. For several real-life applications, the size of the recognition network grows rapidly when a large vocabulary is involved. The basic unit in speech recognition can be the word, syllable or the phone[1], which are modeled by an automaton. Figure 0.3 shows a simplified phone model. At each time frame (typically 10 ms), a transition is used to pass from one state to another.

[1] A phone is an acoustical realization of a phoneme

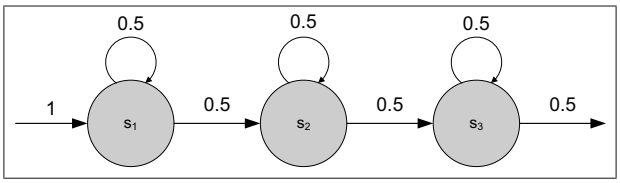

Figure 0.3 Simplified phone model.

The self-loops are introduced to model the fact that the phone duration may vary from one person to the next. This language characteristic increases the complexity of the search since it is possible to be in any given state at any given time.

Figure 0.3 shows the model for one phone. Continuous speech involves the combination of phones to form words. Accordingly, phone models are combined to create words. Figure 0.4 shows how the word "le" is modeled by the concatenation of phones *l* and *oe*.

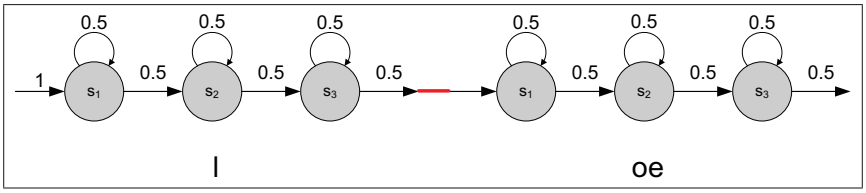

Figure 0.4 Simplified model for the word "le".

However, the speech recognition system does not know that the uttered word is "le". Since there are several words, or parts of words with similar phone sequences, the search algorithm has to take all of these into account. Figure 0.5 shows an example of a network of two words.

It is easy to see that the recognition network becomes very large as the number of words increases. The classic way of implementing the optimal path search in the graph is the Viterbi beam[2] search algorithm. The main advantage of this algorithm is its efficiency since it explores a fraction of the entire search graph. This makes it difficult to efficiently parallelize the Viterbi algorithm on multi-core computers since only 1% of the states are active at each time frame and are scattered in memory. This, in conjunction with the small amount of computation needed by each state, leads to a misuse of the memory architecture of Intel-based computers. It is

[2]At each time frame t, only most promising states are explored by considering states that have a smaller cost than Δ_t, the best cost at time t plus the beam value. This is referred to as the pruning process.

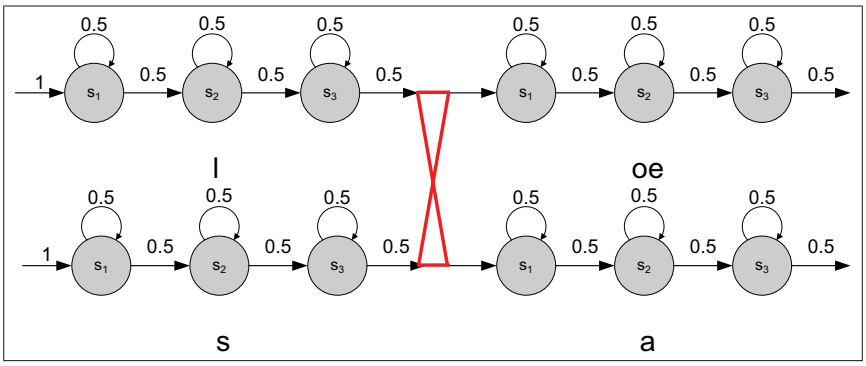

Figure 0.5 Simplified network for 2 words.

a well established fact that searching through a sparse graph on a parallel architecture of the Intel processor type represents a major challenge, as pointed out in Lumsdaine *et al.* (2007). Harish *et al.* have reported quite good results searching the shortest path in dense graphs on a GPU but have reached a similar conclusion in regards to the difficulty of searching sparse graphs (Harish and Narayanan (2007)).

Related Work

Successful attempts to parallelize a speech recognition system have already been made by using dedicated hardwares (FPGA or ASIC). These platforms are programmed using specific languages such as VHDL. A first example is the computation of Gaussian mixture models that have been dedicated to specialized hardware by Shi *et al.*. The GMMs used in the context of classification performed very well compared to its software counterpart. Their software implementation showed a performance of 0.067 classifications per second, while the FPGA accomplished 5.4 per second, a speed-up of 90 (Shi *et al.* (2006)).

Several researchers have implemented a complete speech recognition system in hardware. The aim was firstly to improve performance; later reducing the energy consumption became an important issue. In their experimentations of using external hardware for improving a speech recognition system, Nedevschi *et al.* (2005) implemented a 30-word system for recognizing numbers in a FPGA or ASIC. Their implementation was shown to be very efficient in terms of

energy consumption and performed with comparable accuracy to the software implementation. Lin *et al.* (2006, 2007) implemented a 1000-word speech recognition system in a FPGA that was 7x faster than their software implementation (SPHINX) and resulted in a real-time speech recognizer. In more recent work, they have implemented a 5000-word speech recognizer in a multi-FPGA. Their implementation was 10 times faster than real-time, notwithstanding that the multi-FPGA was running at a clock rate approximately 30 times slower than CPUs of conventional computers (Lin and Rutenbar (2009)). In 2012, Johnston *et al.* built a finite state transducer-based speech recognizer in hardware. With a 60K-word vocabulary, their system ran 127 times faster than real-time with 92.3% accuracy (Johnston and Rutenbar (2012)). The power consumption was only 500 mW. Another system, presented by He *et al.*, reduced the consumption to 144 mW for a word accuracy of 91.3% in a 60K-word vocabulary speech recognizer (He. *et al.* (2012)).

Several approaches towards the parallelization of speech recognition systems on modern-day computers have been proposed. One of the first undertakings in the field was presented in 1999. Phillips and Roggers (1999) described a parallel implementation of a speech recognition system running on a 16-CPU computer. On the North American Business News (NAB) database, they cut down the processing time from 3.9 to 0.8 times real-time[3]. This represents a speed-up factor of 4.9.

Parihar *et al.* (2010) parallelized the search component of a lexical-tree based speech recognizer. In this work, lexical-tree copies are dynamically distributed among the cores to ensure a good load balancing. This results in a speed-up of 2.09 over a serialized version on a Core i7 quad (4 cores) processor. The speed-up was limited by the memory architecture.

A parallel implementation in a cellphone using a 3-core processor is presented in Ishikawa *et al.* (2006). The process was divided into three independant steps with each core being dedicated to each of these. They reported a speed-up factor of 2.6 but their approach is not scalable since the steps involved are not easily parallelizable.

[3]Real-time is defined as the ratio of the overall processing time with the duration of the utterance

The first investigations on the use of GPUs for accelerating speech recognition systems through dedicated acoustic likelihood computations were reported in (Dixon *et al.* (2007)). A more detailed implementation was published the following year by (Cardinal *et al.* (2008)). In this work, the likelihood of every distribution was computed, with all Gaussians, at every frame. Several optimizations were later proposed. One of them was to compute likelihoods for several frames for each distribution (Dixon *et al.* (2009a,b); Cardinal *et al.* (2009)). This approach reduces the number of memory transfers from GPU global memory to processor local memory. This allowed a speed-up of approximatively 40%. A similar approach is described by Vaněk *et al.* (2011).

Usually, only a small amount of Gaussians in a distribution influences the total likelihoods (Knill *et al.* (1996)). In a CPU implementation, it is common practice to compute the likelihood of a distribution by taking into account a correspondingly relevant selection of Gaussians only. Kveton *et al.* proposed an hierchical approach for implementing this optimization in a GPU (Kveton and Novak (2010)). The Gaussians are first grouped into a small number of clusters. Each cluster is then represented by a single Gaussian. At run-time, the likelihood of each cluster are computed and the N best ones are selected. Only Gaussians of the active clusters are then evaluated. They reported a speed-up of 2x over the usual approach.

Another approach aimed at reducing the computational load, which is commonly used in CPU implementations, is to take into account distributions related to active states only. Due to the particular memory architecture of GPU which requires coalesced memory accesses to be efficient, this approach increases the memory bandwith overhead. Gupta *et al.*, in (Gupta and Owens (2009); K.Gupta and Owens (2011)), proposed a multilayer optimization approach to reduce the memory bandwith used by acoustic computations. Their experiments show that states remain active[4] for 11-14 frames. To take advantage of this temporal locality, they processed frames in chunks. In their approach, when a state is activated, the likelihood of its distribution is computed for every following frame in the chunk, regardless if the state has been deactivated or not. This approach allowed to reduce the bandwith use by 80% at the expense of a 20% overhead on the computational load with no loss of accuracy. They proposed

[4] Active states are those that have survived pruning process.

another Gaussian selection optimization, which consists in computing only mixtures that have been selected at the beginning of the chunk, regardless of new activated states. They reported an 82% saving of the bandwith but at the cost of a 10% decrease in accuracy.

The preceding approaches all assume diagonal covariance matrices. Recently, Vaněk *et al.* have proposed a full covariance GMM implementation and compared performances on six different GPUs (Vaněk *et al.* (2012)).

Some work aimed at implementing a complete large vocabulary speech recognition system in a GPU has also been carried out. Several papers from Chong *et al.* and You *et al.* reported a speed-up from 10.5 to 13.75 times compared to their sequential CPU implementation (Chong *et al.* (2009, 2010)). Most of this speed-up is however achieved in the computation of the acoustic likelihoods for which they reported a speed-up factor of 17.7x (3.6x on a multi-core CPU) compared to only 3.7x (2.7x on a multi-core CPU) for the search phase (You *et al.* (2009)). These results illustrate the difficulty of parallelizing the search in a sparse graph. An improvement of 21.9% has been achieved by Kim *et al.* by efficiently packing data with the result of reducing the synchronization overhead (Kim *et al.* (2011)).

In a recent work, Kim *et al.* developed a multi-user speech recognition system in which the GPU was used to improve the throughput and latency of the engine (Kim and Sung (2012)). In another work, Kim *et al.* presented another approach that consists in computing a first decoding pass with smaller models in the GPU followed by a lattice rescoring pass computed on the CPU using bigger models (Kim *et al.* (2012)). The spirit of this work is, in a way, similar to the approach proposed in this work.

All of these works use the Viterbi algorithm for which the memory architecture of usual processors is not adapted. To circumvent this problem, the classical algorithm has been abandoned and replaced by the A* search. The A* search is not a new approach in speech recognition; it has previously been applied to speech recognition by (Paul (1991) and Kenny *et al.* (1992)). This algorithm divides the search operation into two steps. The first step is the computation of a heuristic that yields an estimate of the cost for reaching the final state from any given

state in the graph. The second step is a best-first search guided by the heuristic. The search is still hard to parallelize since active states are still scattered throughout memory. However, it will be shown that the search itself takes up only 7% of the total computation time given a suitable heuristic. Acoustic likelihoods and heuristic cost computations then dominate the total computation time. Fortunately, both of these operations are easier to parallelize.

Thesis Outline

In this thesis, a parallel version of the CRIM speech recognition engine is presented. This new version takes advantage of parallel architectures such as multi-core processors and GPUs.

In Chapter 1, an overview of the theoretical concepts upon which this thesis is based is explored. Firstly, a description of major components of state-of-art speech recognition systems is presented. This is followed by an introduction to weighted finite state transducers that are used to build and manage the recognition network. Finally, a survey of the multi-core processor and GPU architectures is presented.

Chapter 2 presents how acoustic likelihoods can be efficiently computed on SIMD parallel architectures. In most state-of-the-art systems, the acoustic features are modeled by GMMs; one for each phone in a specified context. The main task is to compute the probability that the observation vector has been produced by a given GMM. Since a medium-sized speech recognition system contains approximatively 600 000 Gaussians, this is a computationally intensive task. The key to efficiently implementing this computation within a SIMD architecture is to reduce the acoustic likelihood computation to a dot product. This chapter presents how the computation of acoustic likelihoods can be implemented in GPUs.

The following chapter, Chapter 3, deals with searching the recognition network. This task is basically geared towards searching for the best path in relation to both the language model a priori probabilities encoded in the network and the acoustic likelihoods computed on the fly. This is usually implemented by the Viterbi algorithm. However, the nature of the recognition network and the constraint imposed on the search make it very difficult to parallelize.

This chapter presents how the A* algorithm can be used to replace the search-related computational load by the computation of a heuristic that can be efficiently computed on parallel architectures. In the past, the A* algorithm has been abandoned on account of the difficulty in finding a suitable heuristic. This chapter describes how a smaller recognition network can easily and efficiently be used as a heuristic. Owing to the generic nature of its representation, its integration does not require any modification to the speech recognition engine code. This represents a major advantage since any such heuristic can then be readily incorporated in the speech recognition system.

The results presented in Chapter 4 show that the A* algorithm offers the same accuracy as the classical Viterbi algorithm while performing much more efficiently on parallel architectures. The results will show that when both systems are configured to run in real-time, the parallelized version of the A* search is 5% more accurate than the Viterbi search.

In Chapter 5, parallel architectures are used on a different application, namely copy detection. The copy detection algorithm, that applies common features used in speech recognition, is presented. The CPU version developed at CRIM, although highly accurate, was adversely slow for use in real-life situations. The GPU implementation of the algorithm led to a speed-up of 200x over the CPU version. This improvement allowed the algorithm to be used in an international evaluation in which CRIM obtained very good results in terms of both accuracy and processing speed.

Finally, the thesis is concluded by reviewing the work that has been accomplished and offers suggestions aimed at improving the speech recognition engine and its implementation on specialized hardware.

CHAPTER 1

BACKGROUND

This chapter provides an introduction to the various concepts used throughout this thesis. The first section describes the main components of all speech recognition engines. These concepts will be extensively used in Chapters 2 and 3. The signal processing component is not parallelized since the corresponding processing time is negligible with respect to overall tasks. Many of the underlying details of the signal processing are nevertheless given since it represents the major component of the copy detection algorithm presented in Chapter 5.

The second part of this chapter focuses on the Finite State Transducer (FST) framework that is used to build and represent the recognition network. The concepts presented in chapter 3 make use of the operators that are presented there.

Finally, the last section provides an introduction to the parallel architectures of computer systems. These concepts provide the basis for designing speech recognition systems that take full advantage of the power offered by modern day computers.

1.1 Speech Recognition

The main task of a speech recognition system is to maximize the probability that a sequence of words $\mathbf{w} = w_1, w_2, w_3, ..., w_N$ has been generated by the sequence of observations $\mathbf{o} = o_1, o_2, ..., o_T$:

$$\arg\max_{\mathbf{w}} p(\mathbf{w}|\mathbf{o}) = \arg\max_{\mathbf{w}} \frac{p(\mathbf{o}|\mathbf{w}) \cdot p(\mathbf{w})}{p(\mathbf{o})} \quad (1.1)$$

$$= \arg\max_{\mathbf{w}} p(\mathbf{o}|\mathbf{w}) \cdot p(\mathbf{w}) \quad (1.2)$$

where $p(\mathbf{w})$ is the probability of the word sequence and $p(\mathbf{o})$ the probability of the observation sequence. $p(\mathbf{o})$ can be ignored since it is the same value for all sequences of observations. This

probability is then not considered. Figure 1.1 shows an overview of the principal components used for computing this equation.

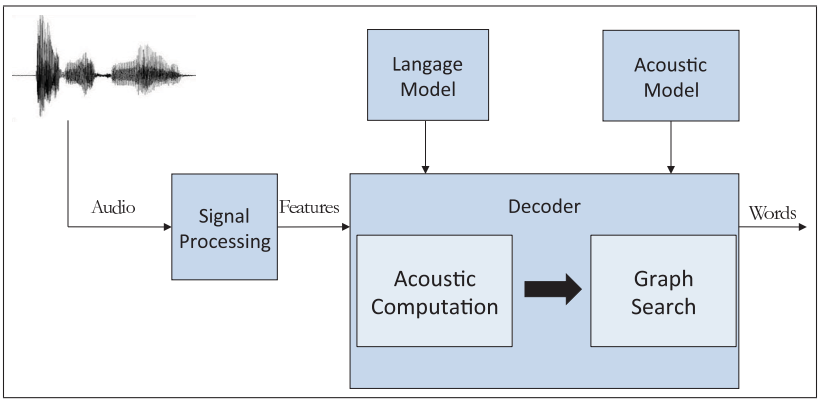

Figure 1.1 Overview of a speech recognition system.

Firstly, a sequence of observations is extracted by analyzing the input audio. An observation vector is produced every 10 ms. Then, the process explores a recognition network that contains information about the language ($p(\mathbf{w})$) and acoustic ($p(\mathbf{o}|\mathbf{w})$) models to find the sequence of words that maximize $p(\mathbf{w}|\mathbf{o})$.

This section details how the acoustic and language models are built and used in order to determine the sequence of words uttered in continuous audio speech.

1.1.1 Feature Extraction

A question that one could ask is: what are the observations in a speech recognition system? The observations are features extracted from the speech waveform. This section describes how this task is achieved.

In speech recognition, the speech waveform is transformed into a sequence of vectors called MFCCs for "Mel Frequency Cepstral Coefficients". Figure 1.2 shows the scheme of the mechanism involved in the transformation of the speech signal into features.

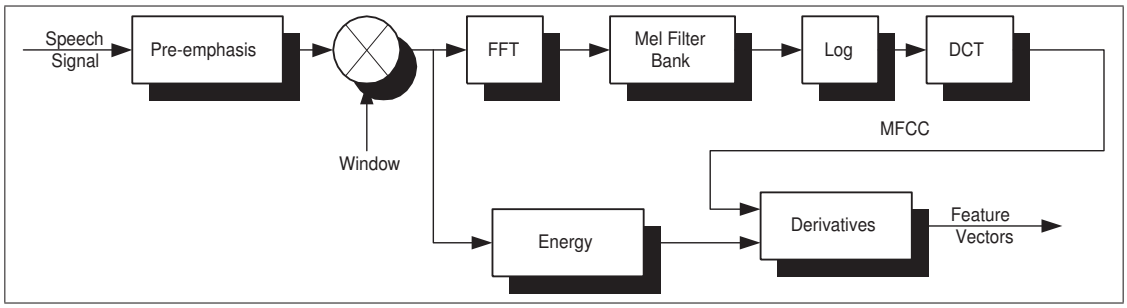

Figure 1.2 MFCC Processing

1.1.1.1 Pre-Emphasis

Phones are divided into voiced and unvoiced sounds. Voiced phones are dominant in speech and exhibit a 6 db/octave decrease in energy. Voiced phones are also characterized by three to four main resonances that are formed by the configuration of the vocal tract. These resonances are called formants.

The high frequency formants have a smaller energy because of the vocal tract characteristics (Huang *et al.* (2001)). Since these formants also contain important information about the uttered phones, the signal may have to be readjusted by a pre-processing function.

The problem is resolved by applying a first order difference filter used to boost formants of the appropriate spectral range. This process is called the pre-emphasis and is defined in the frequency domain by the following transfer function:

$$H(z) = 1 - k * z^{-1} \tag{1.3}$$

where k is the pre-emphasis coefficient (usually 0.95) which should be in the range $0 < k < 1$. For computational efficiency, the filter is usually applied in the time domain. The time domain of equation 1.3 is obtained by applying the inverse Z-transform:

$$s'_n = s_n - ks_{n-1} \tag{1.4}$$

where s_n is the n^{th} sample. The filter can be applied either to the complete signal or to the specific processed window.

1.1.1.2 Windowing

In order to process the non-stationary speech waveform, it is segmented into a sequence of short-term frames whose individual signals can be considered as quasi-stationary. This means that the statistical properties of each component signal are roughly constant over the chosen frame duration. The short-term signal of each frame is processed independently to reduce its acoustic characteristics to a single vector of features (observation), as illustrated in Figure 1.2. The segmentation process, referred to as windowing, is depicted in Figure 1.3. Frames are periodically extracted at a given time interval that is generally less than the frame duration. When that is the case, two consecutive frames have overlapping areas (as illustrated in Figure 1.3). In this figure, $x_1, x_2, x_3, ...$ denote the observation vectors produced at each frame.

In this work, frames of 25 ms duration are extracted at 10 ms intervals, which are typical values used in speech processing. The overlap between adjacent frames is thus 15 ms. The frame rate is 100 frames/s and the speech audio is accordingly converted to a representation of 100 feature vectors per second.

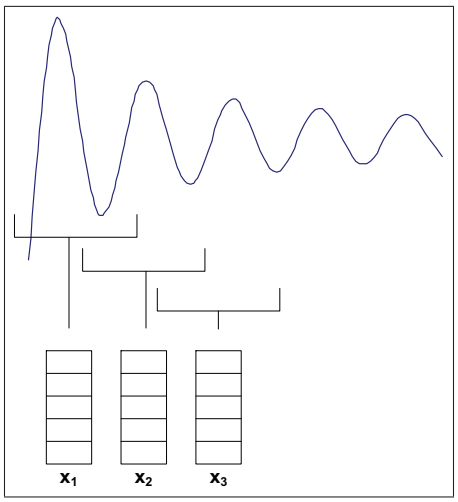

Figure 1.3 Windowing process

There exist many possible window shapes. The simplest one is the rectangular window defined by:

$$w(n) = \begin{cases} 1 & 0 \leq n \leq N-1 \\ 0 & \text{otherwise} \end{cases} \tag{1.5}$$

where N is the window length or frame size in samples. This shape is the simplest since no calculation is involved in the windowing process; the function is constant over the window range. On the other hand, this window shape may lead to a distortion on the estimated spectrum since a discontinuity can be created in the input signal. One way to reduce this effect is to use another shape: the Hamming window whose shape is defined by the cosine function as shown by the following equation.

$$w(n) = \begin{cases} 0.54 - 0.46\cos(\frac{2\pi n}{N-1}) & 0 \leq n \leq N-1 \\ 0 & \text{otherwise} \end{cases} \tag{1.6}$$

This shape is the most commonly used in speech recognition since it represents a good compromise between the time and spectrum resolution.

1.1.1.3 Fourier Transform

Since the analysis is performed in the spectrum domain, a FFT is applied on the input window. To efficiently exploit this transformation, the number of sampling points defining the frame size is equal to a power of two. If the chosen frame size happens to not meet this condition, the number of sampling points is increased to the nearest power of two with zero padding. Note that only half of the output produced by the FFT is used in the spectrum analysis since the second half is symmetric to the first.

1.1.1.4 Filterbank Analysis

Psycho-acoustic experiments have shown that the frequency resolution of the human ear is frequency-dependent. Indeed, the human ear has a greater resolution at low frequency compared to high frequency. Filter banks, called mel-scale filters, have been designed to exploit this fact and to reduce the number of features to be dealt with. Filters have a triangular shape and are spaced along the frequency range following the mel-scale, which is defined by:

$$mel(f) = 2595 \log_{10}(1 + \frac{f}{700}) \tag{1.7}$$

To implement this filterbank, each magnitude coefficient belonging to a filter is scaled according the corresponding filter gain and the results accumulated as shown by the following equation.

$$m_k = \log \sum_{i=\omega_k}^{\omega_{k+1}} w_i c_i \tag{1.8}$$

where m_k is the k^{th} filterbank amplitude bounded by ω_k and ω_{k+1} , w_i is the weight of the i^{th} magnitude coefficient c_i. Each filter produces a amplitude as shown by Figure 1.4.

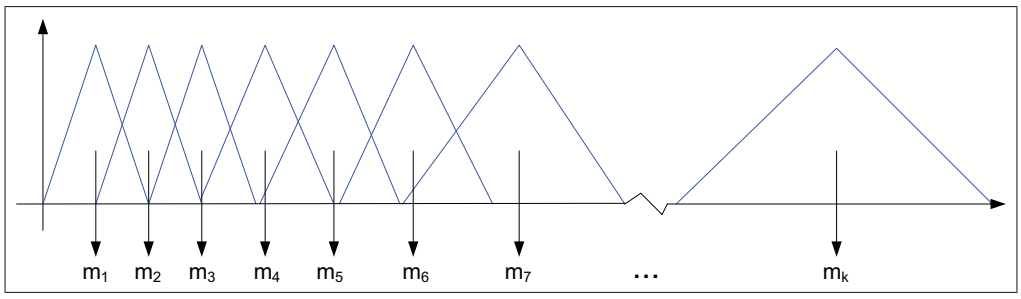

Figure 1.4 Mel-scale filter bank

The MFCCs are then calculated from the log filterbank amplitude by applying a DCT (Discrete Cosine Transform) as described by the following equation (Young and al. (1999)):

$$c_i = \sqrt{\frac{2}{N}} \sum_{k=1}^{N} m_k \cos(\frac{\pi i}{N}(k - 0.5)) \tag{1.9}$$

where N is the number of filterbank channels, c_i is the i^{th} cepstral coefficiant and m_m is the k^{th} log filterbank amplitude. The DCT is used to decorrelate the energy parameters and to reduce the number of parameters to deal with.

The energy of the frame can also be added to the features vector. The energy is computed by applying the formula

$$E = \log \sum_{n=1}^{N} s_n^2 \tag{1.10}$$

to the samples $\{s_0, s_1, ..., s_n\}$ in the time-domain (Young and al. (1999)).

1.1.1.5 Delta and Acceleration Coefficients

The performance of a speech recognition system can be improved by taking into account the dynamic evolution of the speech signal. The addition of time derivatives to the static coefficients is used to capture such information about the input signal. The delta coefficients are computed using the regression formula as described in Young and al. (1999):

$$d_t = \frac{\sum_{\theta=1}^{\Theta} \theta(c_{t+\theta} - c_{t-\theta})}{2 \sum_{\theta=1}^{\Theta} \theta^2} \tag{1.11}$$

where d_t is a delta coefficient at time t and Θ is an interval of static coefficients around the center (usually 2). The same formula is applied to the delta coefficients to obtain the acceleration coefficients.

1.1.2 Language Model

Recall that the speech recognition problem consists in finding the sequence of words that maximizes $p(\mathbf{w}|\mathbf{o})$. Equation 1.2 stated that this probability can be computed as

$$p(\mathbf{w}|\mathbf{o}) = \arg\max_{\mathbf{w}} p(\mathbf{o}|\mathbf{w})p(\mathbf{w}) \tag{1.12}$$

where $\mathbf{w} = (w_1, w_2, ..., w_N)$ is a sequence of words. This section describes the language model which allows the computation of $p(\mathbf{w})$, the a priori probability of the word sequence $\mathbf{w}$. The most common framework for modelling the language model is the N-gram.

The N-gram language model is a statistical model which reflects the probability that a given sequence of words occurs in an utterance. More formally, this probability is defined as

$$\begin{aligned} p(\mathbf{w}) &= p(w_1, w_2, ..., w_n) && (1.13)\\ &= p(w_1)p(w_2|w_1)p(w_3|w_1, w_2)\cdots p(w_n|w_1, w_2, ..., w_{n-1}) && (1.14)\\ &\approx \prod_{i=1}^{n} p(w_i|w_1, w_2, ..., w_{i-1}) && (1.15) \end{aligned}$$

where $p(w_i|w_1, w_2, ..., w_{i-1})$ is the probability of the word w_i given that the previous words of the sentence were $w_1, w_2, ..., w_{i-1}$, which is called the history of word w_i. The number of possible histories grows exponentially with the number of words in the text. For example, for a vocabulary of 25000 words and sentences of 25 words, the number of possible histories is $64000^{25} \geq 10^{120}$. To put things into perspective, the number of atoms in the observable universe is estimated to be 10^{80}. In addition, this probability is hard to estimate accurately since most of the histories may have been encountered but once or not at all, even in a large training set. These problems are circumvented by assuming that a word depends only on the n previous words. This approximation is called a $N - gram$ where $N - 1$ is the size of the history. The simplest version of this model is the *1-gram* often called the *unigram*.

The unigram language model does not consider any history. Thus, the probability of a word is simply its frequency in the training set:

$$p(w) = \frac{C(w)}{\sum_{w \in W} C(w_i)} \tag{1.16}$$

where $C(w_i)$ is the number of times that the word w_i appears in the training set. A more powerful model is the *bigram* which calculates the probability $p(w_i|w_{i-1})$ of a word given the preceding one. The estimation of this probability is obtained in a similar way as that of the unigram:

$$p(w_i|w_{i-1}) = \frac{C(w_{i-1}w_i)}{\sum_W C(w_{i-1}w)} \tag{1.17}$$

$$= \frac{C(w_{i-1}w_i)}{C(w_{i-1})} \tag{1.18}$$

where $C(w_{i-1}w_i)$ is the number of times that the word sequence $w_{i-1}w_i$ appears in the training set. Thus, $p(w_i|w_{i-1})$ is the relative frequency of the sequence $w_{i-1}w_i$ over the frequency of the word history. The most commonly used model is the *trigram* which is an even more realistic representation since most words are strongly dependent on the two previous ones and the quantity of data needed to train it is still reasonable. The estimation of the trigram probabilities are obtained in the same way as those of the bigram:

$$p(w_i|w_{i-1}, w_{i-2}) = \frac{C(w_{i-2}w_{i-1}w_i)}{C(w_{i-2}w_{i-1})} \tag{1.19}$$

Suppose that one wants the probability of the sentence *I like French food* and suppose that the word sequence "like French" scarcely appears in the training data. The sentence to evaluate should receive a good probability since it is a well structured sentence. However, this sentence has a near zero probability given the few occurences of the "like French" bigram in the training

data. This is an important weakness of N-gram models; they are sensitive to the sparseness of the data and unfortunately, natural languages are very sparse. A solution to this problem is the smoothing of probabilities.

1.1.2.1 Smoothing of N-Gram Models

The basic idea of the smoothing technique is to remove a small part of the probability mass of high probabilities and then to redistribute this probability mass to the low or zero probabilities. The simplest smoothing technique is the *add-one smoothing*, which considers that all N-grams have been seen one more time than they actually have (Huang *et al.* (2001)). The estimation of the probability becomes:

$$p(w_i|w_{i-n+1}) = \frac{1 + C(w_{i-n+1}, ..., w_i)}{\sum_W 1 + C(w_{i-n+1}, ..., w)} \tag{1.20}$$

$$= \frac{1 + C(w_{i-n+1}, ..., w_i)}{V + C(w_{i-n+1}, ..., w_{i-1})} \tag{1.21}$$

$$\tag{1.22}$$

where V is the size of the vocabulary. The problem with this technique is that too much probability mass is moved to the zero probabilities. This technique has a number of other weaknesses as mentioned in Jurafsky and Martin (2000). A better technique is the Witten-Bell discounting.

1.1.2.2 Witten-Bell Discounting

Witten-Bell discounting is only slightly more complicated than the add-one technique but leads to better results. The idea is to consider the unseen N-gram as one that has not happened yet (Chen and Goodman (1999)). Thus, the probability of the unseen N-gram can be modeled by

the probability of seeing an N-gram for the first time. Some part of the probability mass must be used for the zero probability N-gram. For a bigram, this probability mass is given by:

$$\sum_{i:c(w_x,w_i)=0} p^*(w_i|w_x) = \frac{T(w_x)}{N(w_x)+T(w_x)} \tag{1.23}$$

where $T(w_x)$ is the number of bigram types for which the word history is w_x and $N(w_x)$ is the number of times these bigrams appear in the training set. Let $Z(w_x)$ be the number of unseen bigrams with the history word w_x :

$$Z(w_x) = \sum_{i:c(w_x,w_i)=0} 1 \tag{1.24}$$

The probability mass must be distributed among all unseen bigrams and removed from the N-gram with non-zero probabilities:

$$p^*(w_i|w_{i-n+1}) = \begin{cases} \frac{T(w_{i-1})}{Z(w_{i-1})(N(w_{i-1})+T(w_{i-1}))} & \text{if} \quad C(w_{i-1},w_i)=0 \\ \frac{C(w_{i-1},w_i)}{C(w_{i-1})+T(w_{i-1})} & \text{if} \quad C(w_{i-1},w_i)>0 \end{cases} \tag{1.25}$$

The model can be extended to trigram and higher-ordered models.

1.1.2.3 Good-Turing Discounting

The idea of this estimation is to group N-grams according to their frequency in the training data set (Huang *et al.* (2001)). The Good-Turing discounting states that an N-gram occurring c times should occur c^*. The new count value is estimated by taking into account the number of bigrams occurring the same number of times and the number of those occurring one more time:

$$c* = (c+1)\frac{N_{c+1}}{N_c} \tag{1.26}$$

where N_c is the number of N-grams occurring c times. The probability of the N-grams is given by:

$$p(w_i|w_{i-n+1}, ..., w_{i-1}) = \frac{c^*}{N} \tag{1.27}$$

where $N = \sum_{c=0}^{\infty} N_c \cdot c$ is the original number of counts in the distribution (Huang *et al.* (2001)).

The smoothing techniques described so far redistribute a part of the probability mass to the unseen N-grams. An extension of these techniques consists in combining probabilities of N-gram models with those of the lower order ones. The most popular approach for combining is the backoff N-gram model.

1.1.2.4 Backoff N-Gram Model

The problem with the N-gram model is that some of the histories have never been seen in the training set, even with a small history. The backoff model is used to circumvent this problem.

The basic idea of the backoff model is the use of the probability of a lower-levelled N-gram when the one at the higher level has a zero probability. For example, if the trigram $p(w_i|w_{i-2}, w_{i-1})$ is not available, it is possible to use the bigram probability $p(w_i|w_{i-1})$ to which a backoff penalty has been added (Huang *et al.* (2001)). Thus, the trigram probability is defined as:

$$p(w_i|w_{i-2}, w_{i-1}) = \begin{cases} p(w_i|w_{i-2}, w_{i-1}) & \text{if} \quad C(w_{i-2}, w_{i-1}, w_i) > 0 \\ \alpha(w_{i-2}, w_{i-1})p(w_i|w_{i-1}) & \text{if} \quad C(w_{i-2}, w_{i-1}, w_i) = 0 \text{ and} \\ & \quad\ C(w_{i-1}, w_i) > 0 \\ \alpha(w_{i-1})p(w_i) & \text{otherwise} \end{cases} \tag{1.28}$$

where $\alpha(x)$ can be thought as a penalty for using a lower-levelled model and is set in such a way to ensure that the probability mass of all lower models sum up to the probability mass discounted in the higher model. The value of $\alpha(x)$ is defined as (Jurafsky and Martin (2000)):

$$\alpha(w_{i-n-1}, ..., w_{i-1}) = \frac{1 - \sum_{w_i:C(w_{i-n+1},...,w_i)>0} p^*(w_i|w_{i-n+1}, ..., w_{i-1})}{1 - \sum_{w_i:C(w_{i-n+1},...,w_i)>0} p^*(w_i|w_{i-n+2}, ..., w_{i-1})} \tag{1.29}$$

where $p^*(x)$ is the probability obtained with the discounted counts. This model has been introduced by Katz (1987).

1.1.2.5 Evaluation of Language Models

The most common metric used for the evaluation of a language model is the perplexity. Given a sequence of words $\mathbf{w} = w_1, w_2, ..., w_N$, the perplexity of a langage model on this word sequence is:

$$PP(\mathbf{w}) = p(\mathbf{w})^{-\frac{1}{N}} \tag{1.30}$$

The perplexity can be interpreted as the geometric mean of the branching factor of the text of a given language model (Huang *et al.* (2001)). A higher perplexity means that, on average, the possible number of words following a previous word increases. In this sense, a higher perplexity indicates a harder task.

1.1.3 Acoustic Model

The second part of the probability to be computed, as stated by equation 1.2, is $p(\mathbf{o}|\mathbf{w})$. This is the probability that a sequence of observations $\mathbf{o} = o_1, o_2, ..., o_T$ corresponds to the sequence of words $\mathbf{w} = (w_1, w_2, ..., w_N)$. The most common method for evaluating this probability is the Hidden Markov Model (HMM).

1.1.3.1 Hidden Markov Model

Since each state of a Markov chain corresponds to an observable event, this model is too restrictive to be applied to complex problems such as speech recognition. Thus, the Markov model must be extended to include the case where an observation is a probabilistic function of the state. The resulting model is a doubly stochastic process with an underlying stochastic process describing the evolution of the states that are not observable (this is why the model is called hidden) except through another set of stochastic processes producing the sequence of observations (Rabiner (1989)). Figure 1.5 shows a 3-state HMM.

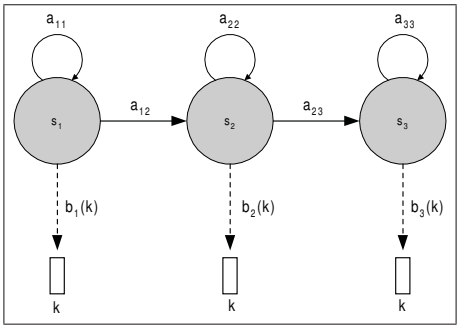

Figure 1.5 A HMM with 3 states

Thus, a HMM has two sets of probabilities. The first set is the transition set which represents the probabilities of going from state i to state j. The second set is made up of the probability density functions (pdf) defining the probability of emitting the output symbol k, represented by a rectangle in the figure, while being in a state i.

In this figure, output symbols are generated by states but they could also be generated by transitions.

Definition 1.1. *More formally, a hidden Markov model is a 4-tuple* $\lambda = (S,A,B,\pi)$ *where*

- $S = \{s_1, s_2, ..., s_N\}$ *is a set of* N *states,*
- $A = \{a_{ij}\}$ *where* $a_{ij} = p[q_{t+1} = s_j | q_t = s_i]$ *(transition probabilities),*
- $B = \{b_i(o_t)\}$ *where* $b_i(o_t) = P[o$ *at time* $t | q_t = s_i]$ *(observation probabilities),*
- $\pi\{\pi_i\}$ *where* $\pi_i = p[q_1 = s_i]$ *(initial probablities).*

Given this definition of hidden Markov models, there are three problems of interest:

The evaluation problem

Given a sequence of observations $\mathbf{o} = \{o_1, o_2, \cdots, o_T\}$ and a model λ modeling a symbol, how to compute $p(\mathbf{o}|\lambda)$.

The decoding problem

Given a sequence of observations $\mathbf{o} = \{o_1, o_2, \cdots, o_T\}$ and a model λ modeling a symbol, what is the state sequence in λ that most likely produced the observation sequence.

The training problem

Given a model λ and a set of observations $\mathbf{o}$ for a specified symbol, how parameters of λ should be adjusted to maximize $p(\mathbf{o}|\lambda)$.

These three fundamental problems must be solved to use hidden Markov models in real-world applications such as speech recognition.

1.1.3.2 Evaluation Problem

The evaluation problem is to calculate $p(\mathbf{o}|\lambda)$, the probability of the observation sequence $\mathbf{o} = o_1, o_2, o_3, \cdots, o_T$ given the model λ. This probability can be calculated by enumerating every state sequence of length T, which is the number of observations, and then summing their probabilities.

Let $\mathbf{q} = q_1, q_2, q_3, \cdots, q_T$ be a sequence of T states. The probability that the observation sequence $\mathbf{o}$ was generated by the state sequence $\mathbf{q}$ of model λ is:

$$p(\mathbf{o}|\mathbf{q}, \lambda) = b_{q_1}(o_1) \cdot b_{q_2}(o_2) \cdots b_{q_T}(o_T) \tag{1.31}$$

where the independence of observations is assumed. The probability of such a state sequence is written as:

$$p(\mathbf{q}|\lambda) = \pi_{q_1} \cdot a_{q_1 q_2} \cdot a_{q_2 q_3} \cdots a_{q_{T-1} q_T} \tag{1.32}$$

Thus, the probability that $\mathbf{o}$ and $\mathbf{q}$ occur simultaneously, which is called the joint probability of $\mathbf{o}$ and $\mathbf{q}$, is the product of (1.31) and (1.32):

$$p(\mathbf{q}, \mathbf{o}|\lambda) = p(\mathbf{o}|\mathbf{q}, \lambda) p(\mathbf{q}|\lambda) \tag{1.33}$$

In order to obtain $p(\mathbf{o} \mid \lambda)$, the probabilites of all possible state sequences of length T are summed. Therefore:

$$p(\mathbf{o}|\lambda) = \sum_{all\ \mathbf{q}} \pi_{q_1} \prod_{t=1}^{T} b_{q_t}(o_t) a_{q_t q_{t+1}} \tag{1.34}$$

This probability can be directly calculated using the HMM parameters A and B. However, the computation is clearly exponential since all possible sequences of length T have to be explicitly enumerated.

Fortunately, a more efficient procedure exists. This procedure is called the *forward pass* or the *forward algorithm*. Let the forward probability $\alpha_j(t)$ for a model λ with N states be defined as

$$\alpha_j(t) = p(o_1, o_2, \cdots, o_t, q_t = s_j|\lambda) \tag{1.35}$$

which is the joint probability, given the model λ, of observing the first t observations and being in state j at time t . This probability can be efficiently computed using the following recursion:

$$\alpha_j(t) = \begin{cases} 1 & : \quad j = 1 \text{ and } t = 1 \\ a_{1j} \cdot b_j(o_1) & : \quad j \neq 1 \text{ and } t = 1 \\ \left[\sum_{i=1}^{N} \alpha_i(t-1) a_{ij}\right] b_j(o_t) & : \quad t > 1 \end{cases} \tag{1.36}$$

The probability of the observation sequence O given the model λ is then obtained by summing forward probabilities of every state:

$$p(\mathbf{o}|\lambda) = \alpha_N(T) = \sum_{i=1}^{N} \alpha_i(T). \tag{1.37}$$

This procedure is clearly more efficient than the previous equation since sequences of states are not explicitly enumerated. Moreover, the idea of the forward algorithm is used to solve the decoding problem.

In a similar manner, it is possible to consider a *backward algorithm*. Let $\beta_j(t)$ be the probability of generating the observation sequence $o_{t+1}, o_{t+2}, \cdots, o_T$ at time t given the state j and the model λ. More formally, $\beta_j(t)$ is defined as:

$$\beta_j(t) = P(o_{t+1}, o_{t+2}, \cdots, o_T \mid q_t = s_j, \lambda) \tag{1.38}$$

This probability can be computed using the following recursion:

$$\beta_i(t) = \begin{cases} a_{iN} & : \quad t = T, 1 < i < N \\ \sum_{j=1}^{N} a_{ij} \cdot b_j(o_{t+1}) \cdot \beta_j(t+1) & : \quad t > 1 \end{cases} \tag{1.39}$$

This backward procedure has the same efficiency as the forward procedure described earlier and it leads to exactly the same result. Thus, both of them can be used to compute $P(O|\lambda)$.

1.1.3.3 Decoding Problem

The forward algorithm computes the probability that a sequence of observations was produced by a given model but does not give any information about the state sequence that produced these observations.

Recall that, by definition, the state sequence is hidden. However, finding the state sequence can be useful in several applications such as speech recognition. The best thing that can be done is to produce the state sequence that has the highest probability (maximum likelihood) of being taken while generating the observation sequence.

The procedure used to find the state sequence is called the Viterbi algorithm. This algorithm is essentially the same as the forward procedure except that the summation is replaced by the maximum function. Let $\phi_j(t)$ be the maximum likelihood of observing $o_1, o_2, \cdots, o_t$ and being in state j. The partial likelihood can be computed using the following recursion:

$$\phi_j(t) = \begin{cases} 1 & : \quad j = 1 \text{ and } t = 1 \\ a_{1j} \cdot b_j(o_1) & : \quad j \neq 1 \text{ and } t = 1 \\ \max\limits_i(\phi_i(t-1) \cdot a_{ij}) b_j(o_t) & : \quad t > 1 \end{cases} \tag{1.40}$$

The maximum likelihood is then given by:

$$\phi_N(T) = \max_i(\phi_i(T) \cdot a_{iN}) \tag{1.41}$$

In practice, log probabilities are used since the multiplication of probabilities leads to very small numbers, which results in an underflow. Another advantage is the improvement of the calculation speed since multiplications involved in the calculations are replaced by additions.

The recursion given in Eq. 1.40 is the base of the Viterbi algorithm. Figure 1.6 shows that the algorithm can be visualized as a best path algorithm applied to a matrix for which the horizontal dimension represents the time and the vertical one represents the states of the HMM.

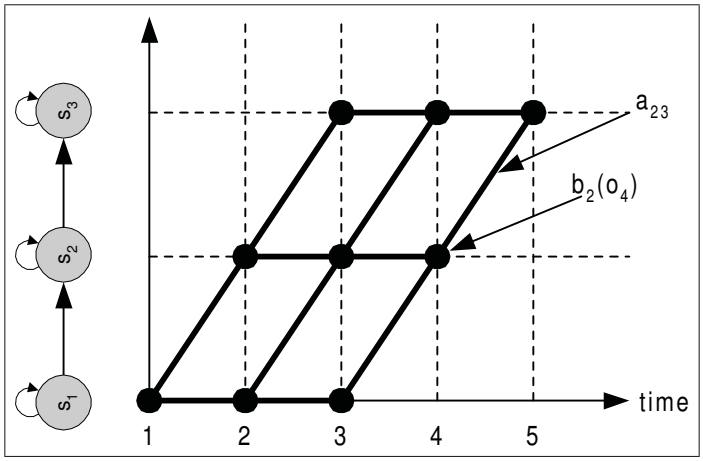

Figure 1.6 The Viterbi algorithm

In this figure, dots denote the probability (or the log probability) of generating an observation o at time t and transitions correspond to transition probabilities in the HMM.

At the end of the algorithm, a backtracking along the best path is performed to obtain the sequence of states with the best likelihood.

1.1.3.4 Learning Problem

The learning problem, which is the most difficult of the three problems, can be stated as: given a model λ and a set of observation sequences $o_1, o_2, ..., o_T$, how can the HMM parameters be optimized to maximize the probability that the observation sequences were generated by λ. Unfortunately, there is no known analytical method to solve this problem. Thus, an iterative method or a gradient descent technique must be used (Kai-Fu (1989)) such as the *Baum-Welch* algorithm, also called the *forward-backward* procedure or the *EM algorithm for HMM*. The Baum-Welch algorithm is presented here such as described in Bilmes (1997) and Rabiner (1989).

1.1.3.5 Preliminary Defintions

First, let $\xi_{ij}(t)$ be the probability of being in state s_i at time t and in state s_j at time $t+1$ given the observation sequence O and the model λ:

$$\begin{aligned} \xi_{ij}(t) &= p(q_t = s_i, q_{t+1} = s_j \mid \mathbf{o}, \lambda) && (1.42) \\ &= \frac{p(q_t = s_i, q_{t+1} = s_j, \mathbf{o} \mid \lambda)}{p(\mathbf{o}|\lambda)} && (1.43) \end{aligned}$$

This probability can be calculated using the forward and backward probabilities defined before:

$$\xi_{ij}(t) = \frac{\alpha_i(t) \cdot a_{ij} \cdot b_j(o_{t+1}) \cdot \beta_j(t+1)}{\sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i(t) \cdot a_{ij} \cdot b_j(o_{t+1}) \cdot \beta_j(t+1)} \tag{1.44}$$

From $\xi_{ij}(t)$, the expected number of transitions from state i to j for the observation sequence O can be derived :

$$\sum_{t=1}^{T-1} \xi_{ij}(t) \tag{1.45}$$

Similary, let $\gamma_i(t)$ be the probability of being in state i at time t given the observation sequence:

$$\begin{aligned} \gamma_i(t) &= p(q_t = s_i \mid \mathbf{o}, \lambda) && (1.46) \\ &= \frac{p(q_t = s_i, \mathbf{o} \mid \lambda)}{p(\mathbf{o}|\lambda)} && (1.47) \end{aligned}$$

This probability can also be calculated using the forward and backward probabilities:

$$\gamma_i(t) = \frac{\alpha_i(t) \cdot \beta_j(t)}{\sum_{i=1}^{N} \alpha_i(t) \cdot \beta_j(t)} \tag{1.48}$$

The expected number of times in state i is obtained by the summation of $\gamma_i(t)$ over time:

$$\sum_{t=1}^{T} \gamma_i(t) \tag{1.49}$$

1.1.3.6 Baum-Welch Algorithm

The EM algorithm can now be applied for resolving the training problem. Let the sequence of observations $\mathbf{o} = (o_1, o_2, ..., o_T)$ be the observable variables and the sequence of states $\mathbf{q} = (q_1, q_2, ..., q_T)$ be the hidden variables. The problem is to find the parameters λ that maximize the complete-data likelihood of the density function $p(\mathbf{o}, \mathbf{q}|\lambda)$. Thus, the Q function is:

$$Q(\lambda, \hat{\lambda}) = \sum_{q \in \mathbf{q}} \log p(\mathbf{o}, q|\lambda) p(\mathbf{o}, \mathbf{q}|\hat{\lambda}) \tag{1.50}$$

where λ is the current estimates of the model parameters and Q is the set of all possible state sequences of length T. Given a state sequence q, the likelihood of the observation sequence and the state sequence is

$$p(\mathbf{o}, \mathbf{q}|\lambda) = \pi_{q_1} b_{q_1}(o_1) \prod_{t=2}^{T} a_{q_{t-1}, q_t} b_{q_t}(o_t) \tag{1.51}$$

Thus, the Q function becomes:

$$
\begin{aligned}
Q(\lambda, \hat{\lambda}) &= \sum_{q \in Q} [\log \pi_{q_1} + \sum_{t=2}^{T} \log a_{q_{t-1}, q_t} + \sum_{t=1}^{T} \log b_{q_t}(o_t)] \cdot p(\mathbf{o}, \mathbf{q}|\hat{\lambda}) && (1.52) \\
&= \sum_{q \in Q} \log \pi_{q_1} p(O, q|\hat{\lambda}) + \sum_{q \in Q} \left(\sum_{t=2}^{T} \log a_{q_{t-1}, q_t} \right) p(\mathbf{o}, \mathbf{q}|\hat{\lambda}) \\
&\quad + \sum_{q \in Q} \left(\sum_{t=1}^{T} \log b_{q_t}(o_t) \right) p(\mathbf{o}, \mathbf{q}|\hat{\lambda}) && (1.53) \\
&= Q_{\pi_i}(\lambda, \hat{\lambda}) + Q_{a_{ij}}(\lambda, \hat{\lambda}) + Q_{b_j}(\lambda, \hat{\lambda}) && (1.54)
\end{aligned}
$$

Since the parameters to be optimized are now split into three independent terms, each one can be optimized individually.

Optimization of Q_{π_i}

Recall that π_i is the probability of state i to be the initial state. Thus, the auxiliary function can be rewritten to reflect that by only considering the first state of the sequence:

$$
\sum_{q \in Q} \log \pi_{q_1} p(O, q|\hat{\lambda}) = \sum_{i=1}^{N} \log \pi_i p(O, q_0 = i|\hat{\lambda}) \tag{1.55}
$$

The optimal value of π_i is obtained by adding the Lagrange multiplier ρ with the constraint $\sum_{i=1}^{N} \pi_i = 1$ and setting the derivative to zero:

$$
\begin{aligned}
\frac{\delta \sum_{i=1}^{N} \log \pi_i p(\mathbf{o}, q_1 = i|\hat{\lambda}) + \rho(1 - \sum_{i=1}^{N} \pi_i)}{\delta \pi_i} &= 0 && (1.56) \\
\frac{1}{\pi_i} p(\mathbf{o}, q_1 = i|\hat{\lambda}) - \rho &= 0 && (1.57)
\end{aligned}
$$

$$\frac{\delta \sum_{i=1}^{N} \log \pi_i p(\mathbf{o}, q_1 = i|\hat{\lambda}) + \rho(1 - \sum_{i=1}^{N} \pi_i)}{\delta \rho} = 0 \quad (1.58)$$

$$1 - \sum_{i=1}^{N} \pi_i = 0 \quad (1.59)$$

By solving this system, the resulting expression for π_i is:

$$\pi_i = \frac{p(\mathbf{o}, q_1 = i|\hat{\lambda})}{\sum_{i=1}^{N} p(\mathbf{o}, q_1 = i|\hat{\lambda})} \quad (1.60)$$

This expression computes the ratio of being in state i over being in any state at time $t = 0$. Thus, this quantity is the probability of being in state i at time $t = 0$ and can be expressed by:

$$\pi_i = \gamma_i(0) \quad (1.61)$$

Optimization of $Q_{a_{ij}}$

The auxiliary function $Q_{a_{ij}}$ can be rewritten :

$$\sum_{q \in Q} \left(\sum_{t=2}^{T} \log a_{q_{t-1}, q_t} \right) p(\mathbf{o}, q|\hat{\lambda}) = \sum_{i=1}^{N} \sum_{j=1}^{N} \left(\sum_{t=2}^{T} \log a_{ij} \right) p(\mathbf{o}, q_{t-1} = i, q_t = j|\hat{\lambda}) \quad (1.62)$$

since passing through all state sequences means passing through all transitions from state i to state j weighted by the probability of being in state i at time $t - 1$ and in state j at time t.

Again, the Lagrange multiplilers with the constraint $\sum_{j=1}^{N} a_{ij} = 1$ are applied (Nilson (2005)):

$$\frac{\delta}{\delta a_{ij}} \sum_{i=1}^{N}\sum_{j=1}^{N}\left(\sum_{t=2}^{T}\log a_{ij}\right) p(\mathbf{o}, q_{t-1}=i, q_t=j|\hat{\lambda}) + \sum_{i=1}^{N}\rho_i\left(1-\sum_{j=1}^{N} a_{ij}\right) = 0 \quad (1.63)$$

$$\sum_{t=2}^{T}\frac{1}{a_{ij}} p(\mathbf{o}, q_{t-1}=i, q_t=j|\hat{\lambda}) - \rho_i = 0 \quad (1.64)$$

$$\frac{\delta}{\delta \rho_i} \sum_{i=1}^{N}\sum_{j=1}^{N}\left(\sum_{t=2}^{T}\log a_{ij}\right) p(\mathbf{o}, q_{t-1}=i, q_t=j|\hat{\lambda}) + \sum_{i=1}^{N}\rho_i\left(1-\sum_{j=1}^{N} a_{ij}\right) = 0 \quad (1.65)$$

$$1-\sum_{i=1}^{N} a_{ij} = 0 \quad (1.66)$$

By solving this system, the resulting expression for a_{ij} is:

$$a_{ij} = \frac{\sum_{t=2}^{T} p(\mathbf{o}, q_{t-1}=i, q_t=j|\hat{\lambda})}{p(\mathbf{o}, q_{t-1}=i|\hat{\lambda})} \quad (1.67)$$

which is the expected number of transitions from state i to state j relative to the expected number of transitions leaving the state i. This quantity can be expressed by

$$a_{ij} = \frac{\sum_{t=2}^{T}\xi_{ij}(t)}{\sum_{t=2}^{T}\gamma_i(t)} \quad (1.68)$$

Optimization of Q_{b_j}

Computing the emission probability for all state sequences is equivalent to computing it for each state i and weighting it by the probability of being in state i at each time t. Thus, the auxiliary function can be expressed by:

$$\sum_{q\in Q}\left(\sum_{t=1}^{T}\log b_{q_t}(o_t)\right) p(\mathbf{o}, q|\hat{\lambda}) = \sum_{i=1}^{N}\left(\sum_{t=1}^{T}\log b_i(o_t)\right) p(\mathbf{o}, q_t=i|\hat{\lambda}) \quad (1.69)$$

The optimization of the function depends on the nature of $b_i(o_t)$. The most popular model used in speech recognition is the mixture of Gaussian probability density functions (PDF). Thus, the auxiliary function becomes:

$$\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\log p(\mathbf{o},\mathbf{q},\mathbf{z}|\lambda)\right)p(O,\mathbf{q},\mathbf{z}|\hat{\lambda}) \tag{1.70}$$

where $\mathbf{q}$ and $\mathbf{z}$ are the hidden variables. More precisely, $\mathbf{q}$ is the sequence of states and $\mathbf{z}$ indicates which mixture component has produced the observation at each time.

Equation 1.70 is almost identical to the one in the Gaussian mixture example in the EM algorithm example. The only difference is the addition of the state sequence. Fortunately, the function can be optimized in exactly the same way. The resulting expressions are:

$$\hat{\alpha}_{im} = \frac{\sum_{t=1}^{T}\gamma_{im}(t)}{\sum_{t=1}^{T}\sum_{m=1}^{M}\gamma_{im}(t)} \tag{1.71}$$

$$\hat{\mu}_{im} = \frac{\sum_{t=1}^{T}\gamma_{im}(t)\mathbf{o_t}}{\sum_{t=1}^{T}\gamma_{im}(t)} \tag{1.72}$$

$$\hat{\Sigma}_{im} = \frac{\sum_{t=1}^{T}\gamma_{im}(t)(\mathbf{o_t}-\mu_{\mathbf{im}})(\mathbf{o_t}-\mu_{\mathbf{im}})^{\mathbf{T}}}{\sum_{t=1}^{T}\gamma_{im}(t)} \tag{1.73}$$

where T denotes the vector transpose, $\gamma_{im}(t)$ is the probability of being in state i at time t and the probability that the mixture component m has produced the observation $\mathbf{o_t}$ and is formally defined as:

$$\gamma_{im}(t)=\frac{\alpha_i(t)\beta_i(t)}{\sum_{j=1}^{N}\alpha_j(t)\beta_j(t)}\cdot\frac{\alpha_{im}\mathcal{N}(\mathbf{o_t}|\mu_{\mathbf{im}},\Sigma_{im})}{\sum_{l=1}^{M}\alpha_{il}\mathcal{N}(\mathbf{o_t}|\mu_{\mathbf{il}},\Sigma_{il})} \tag{1.74}$$

This value is a generalization of $\gamma_i(t)$ defined earlier. The Baum-Welch algorithm consists in iteratively using these equations to re-estimate the HMM paramters as shown by Algorithm 1. This algorithm is an implementation of the EM (expectation-maximization) algorithm.

Algorithm 1: Baum-Welch Algorithm

input : Untrained HMM network
output: Trained HMM network

Guess an inital value for $\bar{a}_{ij}$ and $\bar{b}_j$, $1 \leq i, j \leq N$
while *some convergence criteria is not met* **do**
  $\pi_i = \gamma_i(0)$
  $a_{ij} = \frac{\sum_{t=2}^{T} \xi_{ij}(t)}{\sum_{t=2}^{T} \gamma_i(t)}$
  $\hat{\alpha}_{im} = \frac{\sum_{t=1}^{T} \gamma_{im}(t)}{\sum_{t=1}^{T} \sum_{m=1}^{M} \gamma_{im}(t)}$
  $\hat{\mu}_{im} = \frac{\sum_{t=1}^{T} \gamma_{im}(t) \mathbf{o_t}}{\sum_{t=1}^{T} \gamma_{im}(t)}$
  $\hat{\Sigma}_{im} = \frac{\sum_{t=1}^{T} \gamma_{im}(t) (\mathbf{o_t} - \mu_{\mathbf{im}})(\mathbf{o_t} - \mu_{\mathbf{im}})^{\mathbf{T}}}{\sum_{t=1}^{T} \gamma_{im}(t)}$

The procedure can easily be extended to the case where multiple observation sequences are available. The only addition to update the formula is a summation over the observation sequences.

1.1.4 Evaluation

A common metric for evaluating the performance of speech recognition is the Word Error Rate (WER). The recognized utterances are compared to the reference word sequence using a dynamic programming algorithm that will handle sequences of different length. From this alignment, the WER is defined as the number of errors in the alignment:

$$WER = \frac{\#_{ins} + \#_{subs} + \#_{del}}{N} \cdot 100\% \tag{1.75}$$

where $\#_{ins}$ is the number of words that have been inserted by the recognition engine, $\#_{subs}$ is the number of words that have been replaced by an incorrect word, $\#_{del}$ is the number of

words that have been omitted by the recognition engine and N is the number of words in the reference. Another metric regularly used is the word accuracy, which is defined as:

$$W_{Acc} = 100\% - WER \tag{1.76}$$

Performance evaluations of speech recognition systems presented in this thesis use this metric, which indicates the number of words that have been correctly recognized.

1.2 Weighted Finite State Transducers

Finite-state automata have been extensively studied over the years. Originally, automata theory had been proposed to model brain functions (Hopcroft *et al.* (2000)). This model is very useful for many other purposes and is now used in many important software such as compilers, speech recognition systems and bioinformatics.

The use of finite-state machines is motivated by their computational efficiency. The time efficiency is achieved by using deterministic automata. In such machines, the generation of the output depends only on the length of the input sequence. From this point of view, sequential machines are considered optimal. The space efficiency is achieved with the classical minimization algorithm (Aho *et al.* (1974)). This algorithm ensures that the size of the automaton is minimal according to the language described. The efficiency of such automata has been proven in applications such as compiler design (Aho *et al.* (1986)).

Several operations can be done on finite-state transducers. Some of them are borrowed from graph theory such as the shortest-path algorithm and depth-first search-based algorithms. Other operations are based on the more classic operations of automata theory. These operations have been generalized for weighted string-to-string transducers by Mohri (Mohri (1997)). For example, the composition of transducers is a generalization of the intersection of automata.

Automata are widely used in traditional speech recognition since they represent efficient models for expressing language phenomena such as lexical rules (Becchetti and Ricotti (1999);

O'Shaughnessy (2000); Mohri (1997)). The recent generalization of transducers to the weighted case by Mohri allowed the use of them to build a speech recognition system. From the transducer point of view, $p(\mathbf{o}|\mathbf{w})$ is the transduction from an observation to a word sequence. The recognition network is made up of several stages relating different levels of representation. For example, in a four layer recognition network, the first stage reprensents the HMM, the second one imposes constraints on triphone sequences, the third one describes how words are phonetically represented and the last one imposes constraints on word sequences (language model). The layers are combined together and optimized to create a single network that will be searched by the decoder.

The main advantage of this system over the traditional one is that all speech knowledge is expressed using the same transducer representation, allowing to make changes in the network without modifying the decoder (Kanthak *et al.* (2002)).

This section presents an overview of the WFST framework.

1.2.1 Automata

Automata are a way to describe a set of strings and thus, represent a language. A language is called a *regular language* if and only if it can be represented by a finite automaton. Figure 1.7 depicts a simple automaton.

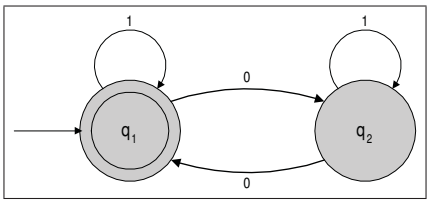

Figure 1.7 Finite automaton with two states

This automaton has two states labelled q_1 and q_2; the *initial state* is characterized by an arrow pointed to it from nowhere; the *final state*, also called *accepting state*, is represented by a double circle; the labelled arrows connecting two states are called *transitions*. In this example, q_1 is both the initial and the final state.

An automaton processes an input string such as 1010 by following transitions from an initial state, depending on the symbols in the input string. Each symbol of the input string is consumed by the automaton from left to right. The output of the automaton is either to accept or to reject the input string. The string is accepted if after having processed all symbols of the input string, the automaton is in an accepting state. If not, the string is rejected by the automaton.

Thus, in the example of Figure 1.7, the state sequence for the input string 1010 will be q_1, q_1, q_2, q_2, q_1. Since the last state q_1 is a final state, the string is accepted by this automaton.

Another interpretation of an automaton is to view it as a generator, rather than a consumer, of symbols. Starting from the initial state and following transitions produces a sequence of symbols, thus a string. The string is valid if the last state visited is a final state.

In the specific example of Figure 1.7, the automaton accepts all strings that have an even number of 0's. Thus, the language is the set:

$$L(\mathcal{A}_1) = \{w \mid w \text{ is the empty string } \epsilon \text{ or has an even number of } 0's\}$$

Definition 1.2. *More formally, a finite automaton $\mathcal{A}$ is a 5-tuple (Q, i, F, Σ, E), where:*

- *Q is a set of states,*
- *$i \in Q$ is the initial state,*
- *$F \subseteq Q$ is the set of final states,*
- *Σ is the alphabet of $\mathcal{A}$,*
- *$E \subseteq Q \times \Sigma \times Q$ is the set of transitions.*

Instead of a set of transitions, it is common to have a transition function mapping a state q and a symbol a to a destination state. More formally, this function is defined as $\delta : Q \times \Sigma \longrightarrow Q$. This function can be extended to $Q \times \Sigma^*$ using the following recurrence relation given by Mohri (1997):

$$\delta^*(q, wa) = \delta(\delta(q, w), a) \quad \forall q \in Q, \forall w \in \Sigma^*, \forall a \in \Sigma \tag{1.77}$$

Thus, a string w is accepted by $\mathcal{A}$ if and only if $\delta^*(i, w) \in F$.

1.2.2 Weighted Automata

Weighted automata, also called weighted acceptors, output a weight depending on the input string and not simply a reject/accept value. The weight carried by transitions along the symbols are $\oplus$-additioned according to a given weight semiring such as the tropical semiring or the log semiring. The choice of the semiring should reflect the intended interpretation of the weights. Figure 1.8 shows a weighted acceptor.

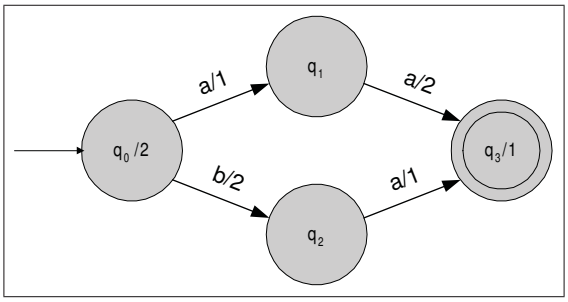

Figure 1.8 Example of a string-to-weight transducer

The weight associated with a string takes into account the output weights of transition but also a weight associated with the initial state and another weight associated with the final state.

Definition 1.3. *More formally, a weighted acceptor $\mathcal{A}$ is a 7-tuple $(Q, i, F, \Sigma, E, \lambda, \rho)$, where:*

- *Q is the set of states,*
- *$i \in Q$ is the initial state,*
- *$F \subseteq Q$ is the set of final states,*
- *Σ is the alphabet of the automaton,*
- *$E \subseteq Q \times \Sigma \times K \times Q$ is the set of transitions,*
- *$\lambda : i \longrightarrow K$ is the initial weight function,*
- *$\rho : F \longrightarrow K$ is the final weight function.*

The set of transitions can be replaced by a transition function, as is the case for non-weighted automata, and by an output function mapping a state q and a symbol a to a weight semiring. More formally, the output function is defined as $\sigma : Q \times \Sigma \longrightarrow K$. As is the case for the transition function, the function can be extended to $Q \times \Sigma^*$ using the following recurrence equation given in Mohri (1997):

$$\sigma^*(q, wa) = \sigma(q, w) \cdot \sigma^*(\delta(q, w), a) \quad \forall q \in Q, \forall w \in \Sigma^*, \forall a \in \Sigma \tag{1.78}$$

Thus, if the string w is accepted by $\mathcal{A}$, its output will be $\sigma(i, w)$.

1.2.3 Epsilon Transitions

An epsilon or null transition is one that does not consume any input symbol. In the graph representation, the epsilon is denoted by the Greek symbol ϵ. Figure 1.9 shows an example of an automaton with ϵ-transitions.

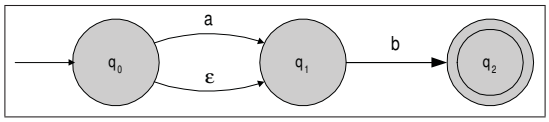

Figure 1.9 Automaton with ϵ-transitions

The language accepted by this automaton is $\{ab, b\}$. Since no input symbols are consumed when an ϵ-transition is taken, the language accepted by the automaton is not influenced by it. The use of epsilons is proposed to simplify the creation of automata.

1.2.4 Determinism

A finite-state automaton is called deterministic (DFA) if and only if for any input string w, the sequence of states is unique. Figure 1.10a shows a non-deterministic finite-state automaton (NDFA) since there are two transitions with the symbol a going out of state q_0. Figure 1.10b shows a deterministic automaton accepting the same language as the automaton of Figure 1.10a.

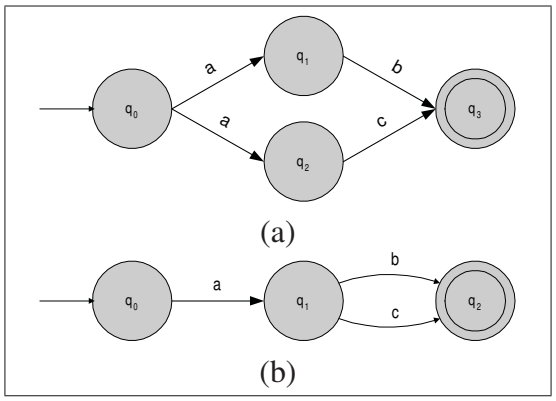

Figure 1.10 Non-deterministic and deterministic automata

Definition 1.4. *More formally, an automaton* $(Q, i, F, \Sigma, \delta)$ *is deterministic if:*

$$|\delta^*(q, w)| \leq 1 \qquad \forall q \in Q, \forall w \in \Sigma^*$$

Every language that can be described by a NDFA can also be described by a DFA (Hopcroft *et al.* (2000); Sipser (1997)). This property helps with the design of automata since it is often easier to construct a new automaton as NDFA and then to transform it to a DFA. Since DFAs are computationally more efficient, this operation is very useful.

1.2.5 Finite-State Transducers

Transduction is the process that maps an input string w_i over the alphabet Σ_i to an output string w_o over the alphabet Σ_o.

Definition 1.5. *A transduction is a mapping function defined as* $\mathcal{T} : \Sigma_i^* \longrightarrow \Sigma_o^*$ *where* Σ_i^* *is the set of input strings and* Σ_o^* *is the set of output strings.*

Definition 1.6. *A weighted transduction is a mapping function defined as* $\mathcal{T} : \Sigma_i^* \longrightarrow \Sigma_o^* \times K$ *where* Σ_i^* *is the set of input strings,* Σ_o^* *is the set of output strings and K is a weight semiring.*

Transducers are a type of automaton whose transitions carry an output symbol in addition to the input symbol. Thus, the output of a transducer is a string over a given alphabet and not just a weight or a reject/accept value as with automata.

1.2.6 String-To-String Transducers

A string-to-string transducer represents the function $T : \Sigma_i^* \longrightarrow \Sigma_o^*$ where Σ_i^* and Σ_o^* are the sets of input and output strings. Figure 1.11 shows an example of a string-to-string transducer. In this example, the string aa is mapped to the string cd while the string ba is mapped to the string ec. All other strings are rejected by the transducer.

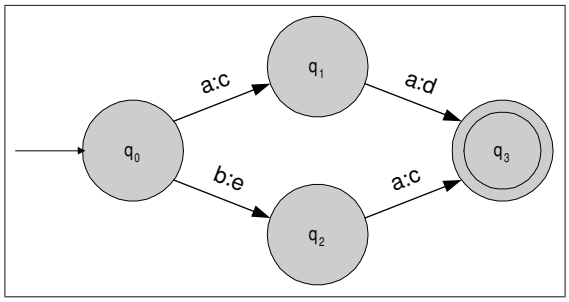

Figure 1.11 Example of a string-to-string transducer

Definition 1.7. *More formally, a string-to-string transducer $\mathcal{T}$ is a 6-tuple $(Q, i, F, \Sigma_i, \Sigma_o, E)$, where:*

- *Q is the set of states,*
- *$i \in Q$ is the initial state,*
- *$F \subseteq Q$ is the set of final states,*
- *Σ_i is the input alphabet of the automaton,*
- *Σ_o is the output alphabet of the automaton,*
- *$E \subseteq Q \times \Sigma_i \times \Sigma_o \times K \times Q$ is the set of transitions.*

As is the case for acceptors, the set of transitions can be replaced by a transition function and an ouput function. The transition function is the same as for acceptors while the ouput function becomes $\sigma : Q \times \Sigma_i \longrightarrow \Sigma_o$. Both functions can be extended using the recurrence relations expressed in equations 1.77 and 1.78.

1.2.7 Weighted String-To-String Transducers

The weighted string-to-string transducer is the most general finite-state automaton discussed in this work. It maps a pair consisting of an output string and a weight.

More formally, the mapping function of a weighted string-to-string transducer is $T : \Sigma_i^* \longrightarrow \Sigma_o^* \times K$ where Σ_i^* and Σ_o^* are the sets of input and output strings respectively and K is a weight semiring. Figure 1.12 shows a weighted string-to-string transducer.

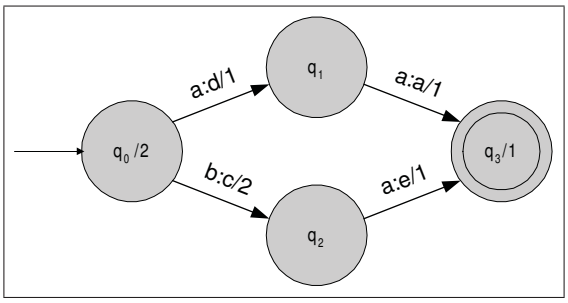

Figure 1.12 Example of a weighted string-to-string transducer

As in the case for weighted acceptors, a weighted string-to-string transducer also provides an initial and a final weight.

Definition 1.8. *A weighted string-to-string transducer $\mathcal{T}$ is a 8-tuple $(Q, i, F, \Sigma_i, \Sigma_o, E, \lambda, \rho)$, where:*

- *Q is a set of states,*
- *$i \in Q$ is the initial state,*
- *$F \subseteq Q$ is the set of final states,*
- *Σ_i is the input alphabet of the automaton,*
- *Σ_o is the output alphabet of the automaton,*
- *$E \subset Q \times \Sigma_i \times \Sigma_o \times K \times Q$ is the set of transitions,*
- *$\lambda : i \longrightarrow K$ is the initial weight function,*
- *$\rho : F \longrightarrow K$ is the final weight function.*

As in the case for string-to-string transducers, the set of transitions can be replaced by a transition function and an output function. The transition function is identical to that of the string-to-string transducer and the output function becomes $\sigma : Q \times \Sigma_i \longrightarrow \Sigma_o \times K$. Both functions can be extended using the recurrence relations expressed in equations 1.77 and 1.78.

1.2.8 Epsilon Symbols in String-To-String Transducers

As is the case for automata, epsilon symbols are allowed in string-to-string transducers both for input and output symbols. An input string and its corresponding output string do not necessarily have the same length. Thus, epsilons are used to fill the “blanks".

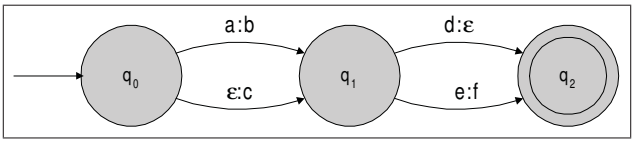

Figure 1.13 Example of a transducer using epsilons.

Figure 1.13 shows a transducer using epsilons to map strings of different length. In a transducer, ϵ-transitions are represented by a transition with input and output epsilons.

1.2.9 Sequential Transducers

A transducer is called sequential if it is deterministic from the point of view of its input. Figure 1.14a shows a non-sequential transducer since there are two transitions with the symbol a outgoing from state q_0. Figure 1.14b shows a sequential transducer.

The empty string, namely ϵ, is not allowed as an input symbol in a sequential transducer. Sequential transducers are computationally efficient since the time requirements depend only on the size of the input string and not on the size of the transducer. This efficiency comes from the fact that for a given input string, the output string is written by following the only corresponding path.

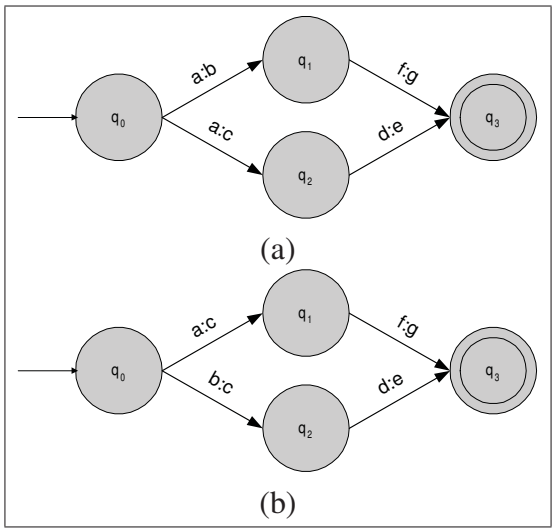

Figure 1.14 A non-sequential and a sequential transducer

1.2.10 Operations on Transducers

As is the case for automata, many operations are available for working with transducers. This section will briefly describe the more important of them.

1.2.10.1 Reverse

This operation consists in reversing all transitions of the given transducer. The operation also transforms final states into an initial state and the initial state into a final state. The reverse operation is denoted by $T_{res} = T_{in}^{r}$. Figure 1.15b shows the reverse of transducer of figure 1.15a. Note that applying the reversal operation twice on a transducer T produces a new transducer equivalent to T in which there is only one final state, *i.e.*, $|F| = 1$.

1.2.10.2 Composition

Composition is a generalization of the intersection operation for automata. This operation is very useful since it allows the construction of complex transducers from simpler ones. Figure 1.16 shows a cascade of two transducers.

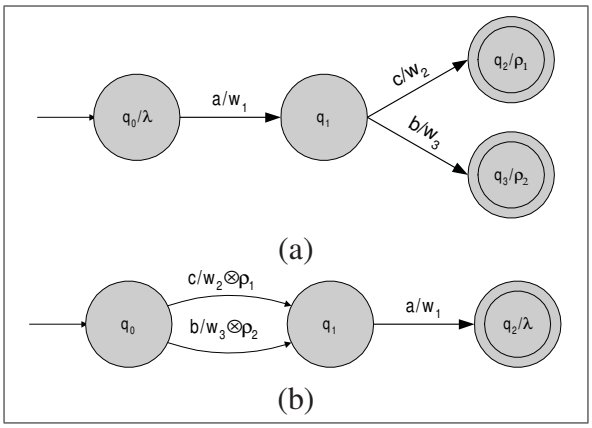

Figure 1.15 Example of transducer reversal

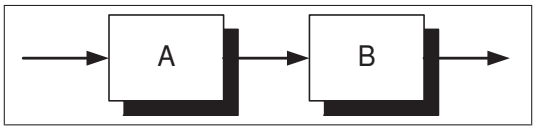

Figure 1.16 A cascade of two transducers

The transducer A maps Σ_i^* to Δ^*. Thus, the set Δ^* becomes the input of transducer B that maps Δ^* to Σ_o^*. Therefore, the general behaviour of the cascade is: $A \circ B = \Sigma_i^* \longrightarrow \Sigma_o^*$. The composition creates the transducer equivalent to this cascade.

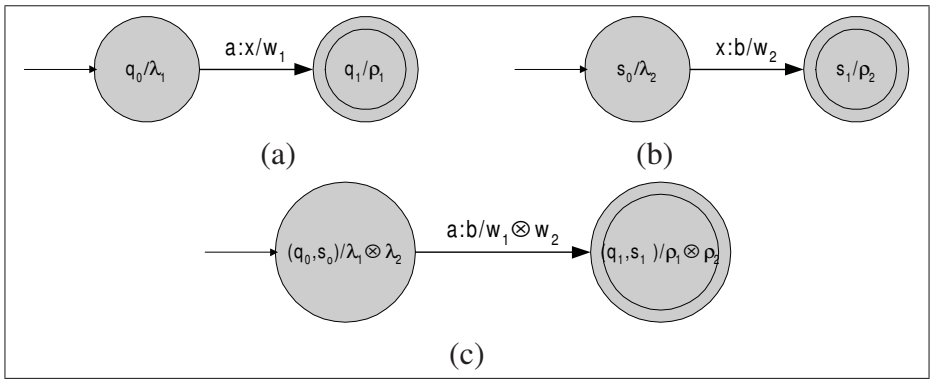

Figure 1.17 Example of transducer composition

Given a transducer A in which there is a path mapping sequence x to sequence y and a transducer B in which there is a path mapping sequence y to sequence z, the composition $A \circ B$ has a path mapping x to z. The weight of this path is the $\otimes$-product of the weights of the cor-

responding path in A and B (Mohri *et al.* (1996)). Figure 1.17 shows two simple transducers and the result of their composition.

The composition is a key operation in transducer-based applications since it is used to construct complex transducers representing complex functions. For example, in the case of speech recognition, the composition is used to construct the knowledge network needed by the recognition system. This network is constructed by the composition of different levels of representation (acoustic, lexical and semantical) for which transducers are associated. The construction of this network will be described in detail later.

1.2.10.3 Determinization

Deterministic automata and sequential transducers have already been defined. Any non-deterministic automaton has an equivalent deterministic one. Determinization is the process which takes a non-deterministic automata as input and produces a deterministic one as output. Figure 1.18b shows a deterministic automaton constructed from the automaton of figure 1.18a.

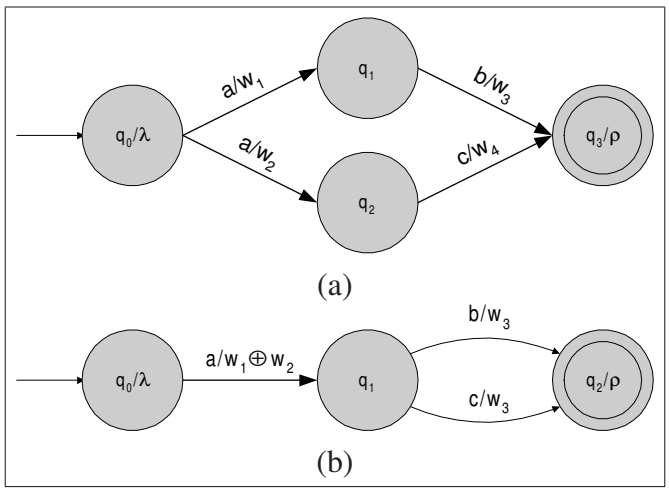

Figure 1.18 Example of transducer determinization

Deterministic automata are computationally more efficient but in practice, the number of states involved is often greater than the equivalent non-deterministic counterpart. In the worst case, the smallest deterministic automaton can have 2^n states while the smallest non-deterministic automaton describing the same language has n states.

The same operation can be applied to non-sequential transducers to obtain sequential ones. Unfortunately, this process does not terminate for all transducers. This point will be discussed in the next chapter.

1.2.10.4 Other Operations

The major FST operations have been presented but there exist some other useful manipulations that can be done on a FST, and are briefly described here:

Minimization

Return an equivalent transducer with the minimal number of states.

Inversion

Invert the transducer by swapping the input and output symbols on transitions.

Arithmetic

Apply some arithmetic operation (addition or multiplication) on weights of weighted FSM.

Projection

Convert a transducer to an acceptor by keeping either only the input or the output symbol.

Best paths

Find the k paths of lowest weight from the initial state to a final state in a weighted FSM.

Topological sort

This operation numbers states such that for any transition from a state numbered i to a state numbered j, the condition $i \leq j$ is respected.

1.3 Parallel Architectures

This section presents a brief introduction to multi-core processors and GPUs. Both of these components represent parallel architectures that are omnipresent in modern day computers. The algorithms described in this thesis have been designed by taking into consideration the specific characteristics of these architectures.

1.3.1 Multicore Processors

Figure 1.19 shows an overview of the interconnections between an Intel Core i7 processor and its components. In terms of parallel processing capabilities, the main improvement of this architecture with respect to the previous Core 2 architecture is that the memory is directly connected to the processor via QuickPath Interconnect (QPI) links, affording a transfer rate up to 25.6 GB/s. However, all memory banks are connected through the same link. As a result, only one core at a time can interact with the memory. In Core2 technology the memory was interfaced through the north bridge. In addition to a slower transfer rate (10.6 GB/s), the same link was used for accessing other connection ports such as the PCI express port or hard disk.

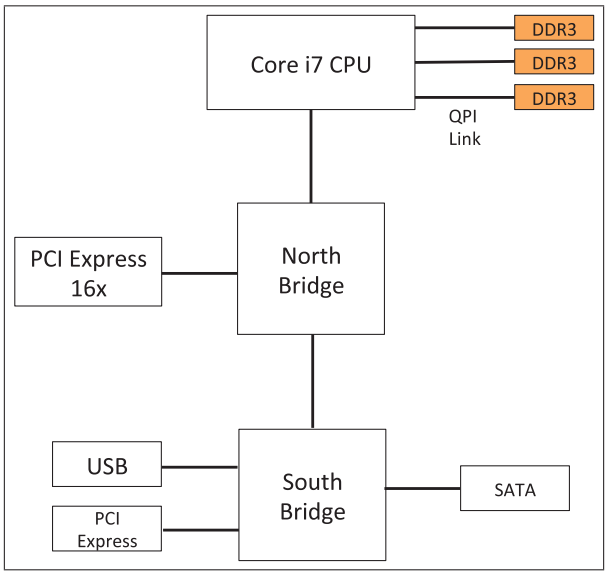

Figure 1.19 Overview of the Core i7 architecture.

In order to take full advantage of the benefits offered by parallel processing, it is essential to have a clear understanding of the various kinds of memory architectures that are accessed during program execution. The efficiency of parallel implementations of memory bounded algorithms depends on optimal memory management. Indeed, communication with memory can become an important bottleneck when several cores access memory simultaneously.

There are mainly two types of memory: dynamic and static. The structural simplicity of dynamic memory, depicted in Figure 1.20(a), allows it to reach very high densities. For this reason, it is used for computer CPU main memory. This memory is referred to as dynamic since the capacitor has the inconvenience of leaking charge over time. As a result, a special circuitry is needed to refresh it periodically.

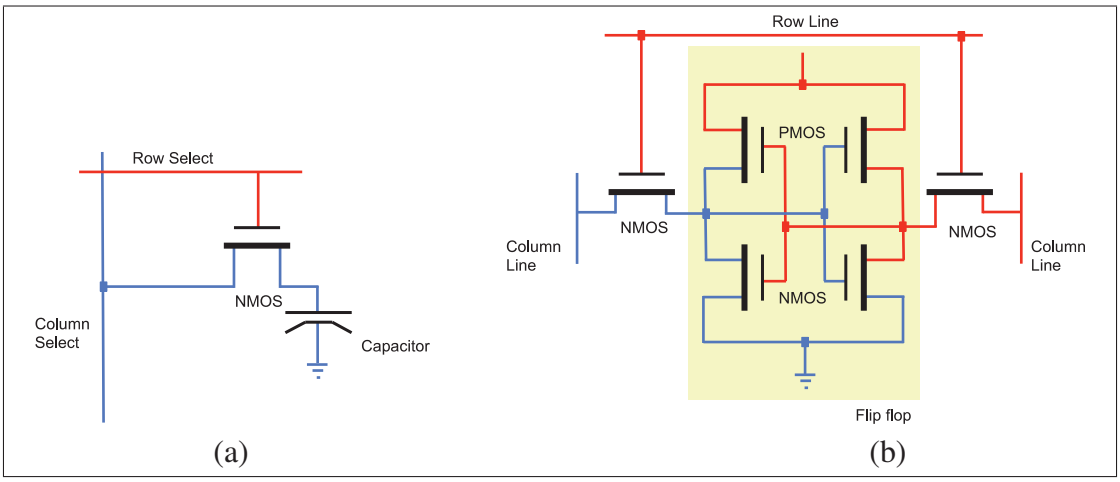

Figure 1.20 The two main types of memory: (a) dynamic memory and (b) static memory

The disadvantage of dynamic memory is that it suffers from high latency (the time that the proper segment of memory is located, read and sent to the processor), a performance-inhibiting factor. For this reason, another type of memory is used in complement with dynamic memory, namely static memory. This memory, depicted in Figure 1.20(a), is architecturally much more complex and therefore much more expensive. In contrast with dynamic memory, its integration density is much smaller but offers the advantage of having less latency. Static memory is

typically used for cache implementations in the CPU. The cache has the property of reducing the average memory latency by storing the most frequently used main memory data.

Several levels of cache memory may be present, as illustrated in Figure 1.21. The cache configuration depends on the CPU architecture. On Intel Core i7, there are 3 levels of cache, one of them (the level 3 cache) is shared among all cores of the processors. The smallest level one is the fastest.

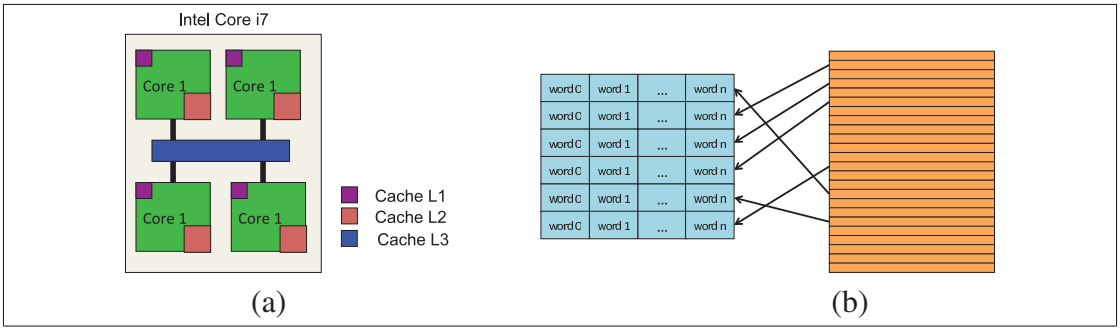

Figure 1.21 Overview of the Core i7 cache memory architecture. (a) Different levels of cache in Core i7 CPU. (b) Relation between the main memory and the cache.

In relation to this work, due consideration must be given to the manner that cache is utilized. Memory is typically divided into lines of 16KB, 32KB, 64KB or 128KB. The cache is similarly organized (the size of the lines must be identical). When the processor needs to access in-memory data, the cache memory is searched. If the data is not found in the cache, the line containing the required data is then transferred from main memory to the cache. Since all words of the lines are transferred together, a degree of latency is hidden since there is no need to locate and read other words of the line. This architecture is used because it is assumed that the line of data may need to be accessed again soon. When that is the case, this approach considerably reduces the average data latency. It is incumbent upon the programmer to structure the data in such a way as to take advantage of the benefits afforded by this architecture. This is particulary true in parallel programming since more than one core may want to acccess the memory simultaneously.

1.3.2 Graphic Processor Units

A Graphic Processor Unit (GPU) is a parallel processor specialized in graphical rendering. A GPU is a set of multiprocessors, each of which contains a certain amount of calculation units. Each calculation unit executes the same instruction concurrently on different data. To be efficient, programs running on GPUs must be designed to take advantage of this architecture. An example of misusing this architecture is the utilization of data-dependent conditional branches. Such an implementation prevents concurrent execution of different instructions: while some calculation units process instructions of one possible branch, the other units are stalled. Upon execution completion of this branch, the stalled units will then restart in order to process instructions in their branch while the others are suspended.

Figure 1.22 illustrates the effect of a conditional branch in the kernel. This example highlights the need for avoiding conditional branches, particularly those that depend on the data. When this is not possible, data should be arranged in such a way as to minimize the use of different branches in a warp (meaning that all calculation units use the same branch). Several conditional branches in GPU programs can thus significantly hinder performance.

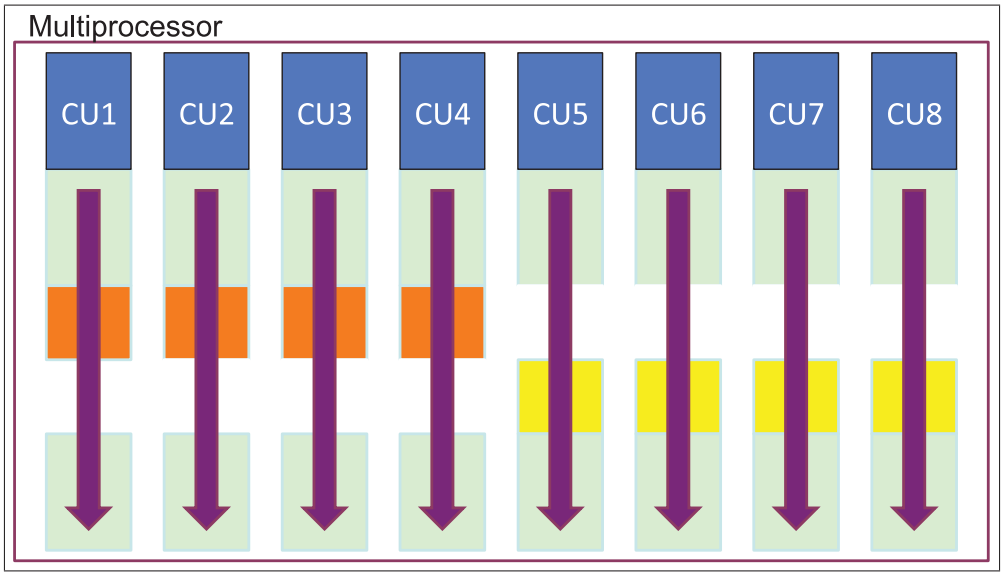

Figure 1.22 Effect of conditional branches on SIMD architectures.

In this work, we use a NVidia GPU which can be programmed with CUDA, a development platform for NVidia graphic cards (CUDA (2012)).

1.3.2.1 Introduction to CUDA

As described in NVidia (2007), the CUDA framework exposes the graphic card as a parallel coprocessor for the CPU. The development language is C with some extensions.

A program in the GPU is called a kernel and is made up of configurable amounts of blocks, each of which consists of a configurable amount of threads as shown in Figure 1.23. Data processing is handled with the aid of built-in variables that allow threads and blocks in a multiprocessor to access its dedicated data. Note that several kernels can be launched concurrently.

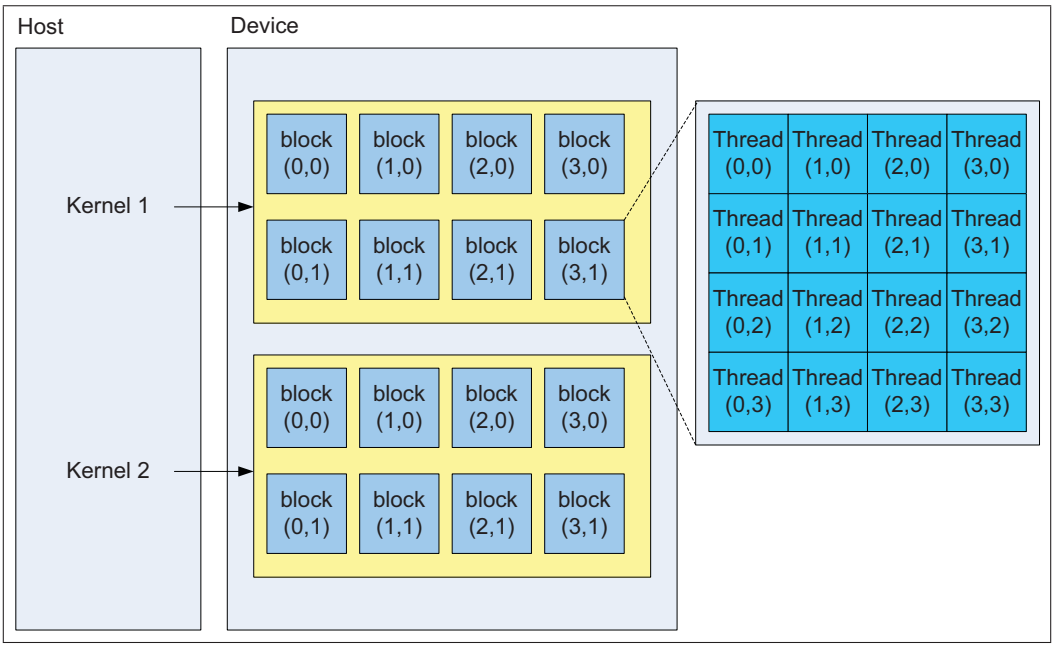

Figure 1.23 Overview of CUDA thread batching.
Source: NVidia (2007)

At execution time, each block is assigned to a multiprocessor. More than one block can be assigned to a given multiprocessor. Blocks are divided into groups of 32 threads called warps. In a given multiprocessor, the number of threads executed at the same time depends on the

model used. A time slicing-based scheduler switches between warps to maximize the use of available resources.

There are primarily two types of memory. The first is the global memory that is accessible by all multiprocessors. This memory is very slow (and not cached as in older architectures), so it is important to ensure that the read/write memory accesses by a warp are coalesced in order to improve performance. Recall that the SIMD architecture allows for all threads of a multiprocessor to access memory concurrently. Figure 1.24 illustrates schemes of non-coalesced and coalesced accesses. Suppose that a thread block consists of 4 threads. In Figure 1.24a, memory accesses among the threads (denoted by T_i) are not consecutive memory addresses. This implies that each thread has to issue a different memory request. In Figure 1.24b on the other hand, the memory accesses are consecutive during execution. Consequently, only one request to memory is needed.

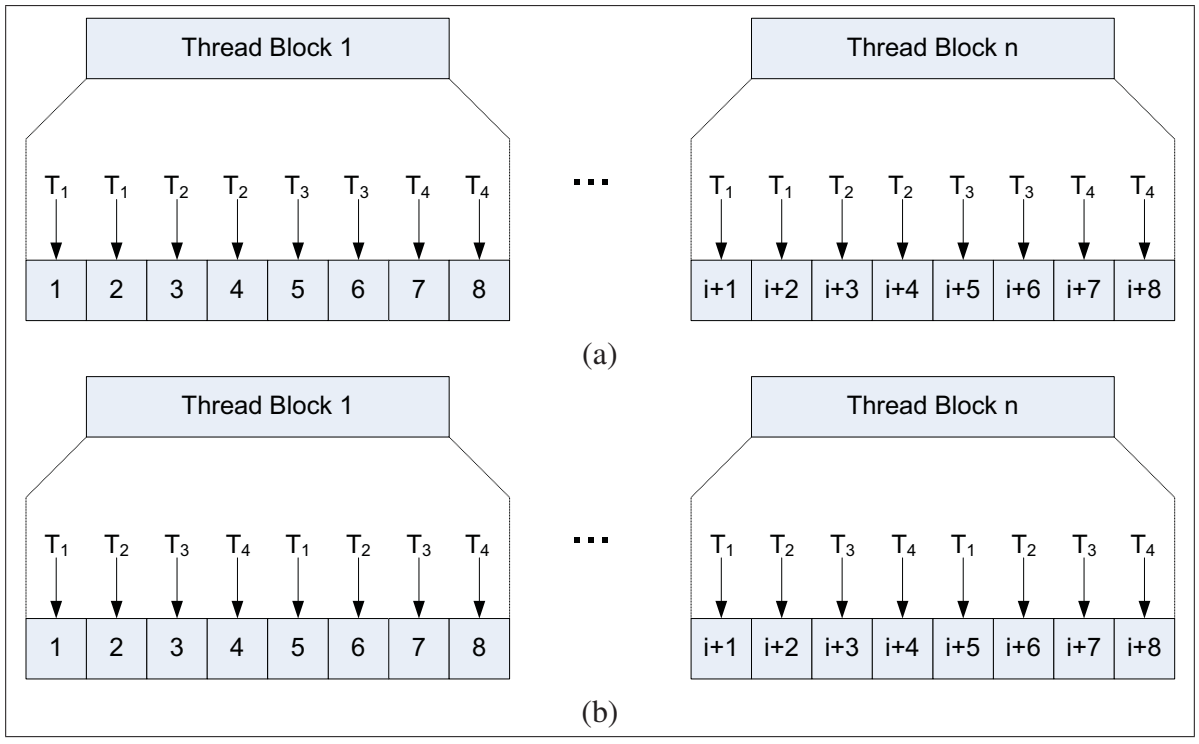

Figure 1.24 (a) Non-coalesced and (b) coalesced memory access.

The second type of memory is shared memory that is internal to multiprocessors and is shared within a block. This memory, which is a lot faster than global memory, can be viewed as user-managed cache. It is divided into banks in such a way that successive 32-bit words are in successive banks. To be efficient, conflicting accesses among threads must be avoided. Conflicts are resolved by serializing accesses. This incurs a performance drop that is proportional to the number of serialized accesses.

Another type of memory is also available: the texture memory. This represents a small part of the global memory that is cached. Texture memory can be efficient when data exhibits locality[1].

1.3.3 Performance Evaluation

The common metric for evaluating the performance of a parallel implementation is its speed-up over its sequential version. This speed-up is defined as the ratio of the CPU time of the sequential version of the application and the elapsed time of its parallel counterpart. The CPU time is the combined processing time of each core while the elapsed time is the time taken from the start of a procedure until the end as measured by an ordinary clock. When no overhead is induced by the parallelization process, the ratio of the CPU time and the elapsed time should be equal to the number of cores dedicated to the process.

1.4 Summary

This chapter introduced several aspects that will be used throughout this thesis. The chapter was divided into three parts. In the first part, an introduction to speech recognition has been given. That will allow the reader to understand how a speech recognition system works and will help to understand the complexity of the task.

[1]The principle of locality is a phenomenon describing the same value or related storage locations being frequently accessed. There are two basic types of reference locality. Temporal locality, refers to the reuse of specific data, and/or resources, within a relatively small time duration. Spatial locality, refers to the use of data elements within relatively close storage locations. Sequential locality, a special case of spatial locality, occurs when data elements are arranged and accessed linearly, such as, traversing the elements in a one-dimensional array (Wikipedia (2013b)).

The second part of the chapter described a framework based on weighted finite state transducers, which is a generalization of the theory of automata. In speech recognition, the use of transducers is motivated by their computational and space efficiency, making them ideal for representing the recognition network. In this thesis, the heuristic used by the A* algorithm will also be represented by a WFST. A major advantage of using this approach is that it is possible to make changes in the heuristic without modifying the decoder.

The last section introduced parallel architectures commonly found in every-day computers. Designing efficient parallel algorithms involves a thorough knowledge of these architectures. The designs of the algorithms and data structures presented throughout this thesis are motivated by the efficient use of these architectures in order to take full advantage of their computational power.

CHAPTER 2

ACOUSTIC LIKELIHOOD COMPUTATIONS

This section describes how to compute acoustic likelihoods in a parallel speech recognition system. Acoustic models are used to model the voices of speakers. There are typically three types of acoustic models:

a. Gender independent

b. Speaker independent

c. Speaker dependent

Gender independent speaker models can be used by everyone. However, these models are usually trained on language- and country-specific speakers. For example, models trained on Quebec French speakers are less compatible with speakers from France. Speaker independent models are usually trained on same-gender speakers. These models usually produce better results than gender independent models. Speaker dependent models usually give rise to still better results since they are trained on specific speakers. As shown in Figure 2.1, the computation of acoustic likelihoods is part of the decoder that is used to drive the search in the recognition network.

Acoustic features can be modeled by a variety of methods such as Support Vector Machines (SVM), Neural Network (NN) or Deep Belief Network (DBN). In state-of-the-art systems, they are usually modeled with a mixture of Gaussians (GMM). The smallest unit in speech processing is the phone. Usually, in-context phones are used. Each distribution models a triphone (or a 5-phone, 7-phone,...) with a certain amount of Gaussians. On average complex systems, there are typically 600 000 Gaussians. Since acoustic likelihoods are computed at every frame (each 10 ms), a huge amount of computation is involved.

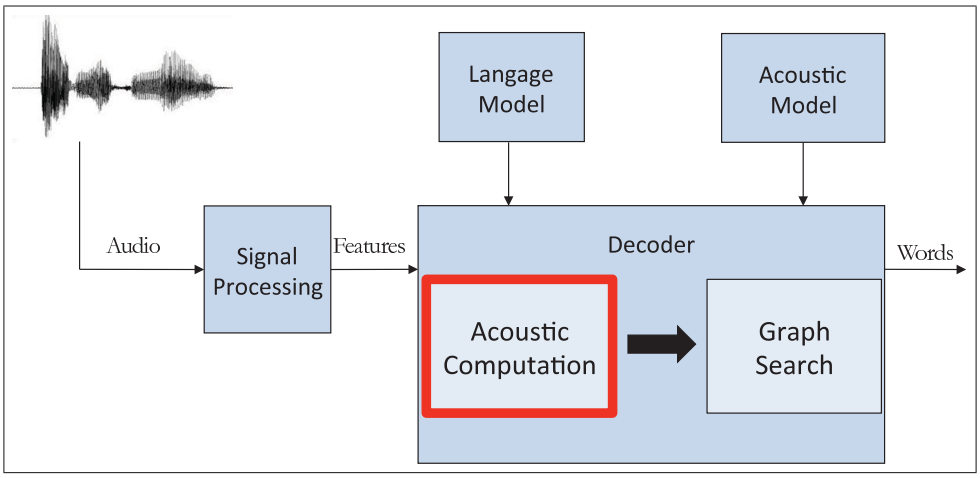

Figure 2.1 Acoustic likelihood computation in a speech recognition system.

As a result, the computation of acoustic likelihoods is a major part of speech recognition systems. Depending on the task, this step can account for between 30% and 80% of the total computation time. Consequently, optimizing this step can lead to significant improvements on the decoding speed of speech recognition systems.

This chapter describes how acoustic likelihoods can be efficiently computed on parallel architectures such as Intel multicore processors and GPUs.

2.1 Computation of Acoustic Likelihoods

Consider a set of T observation vectors (or feature vectors) and a set of Gaussian Mixture Models. The computation of acoustic likelihoods consists in computing the log-likelihood of each pair of observation vector and GMM. Considering that a typical medium-sized speech recognition system may contain on the order of 600 000 Gaussians, this is a highly intensive computational task.

However, during a Viterbi beam search, only acoustic likelihoods for Gaussians associated with states having survived the pruning process are needed. This considerably reduces the computational burden. Yet, even with this approach, the computation time of acoustic likelihoods remains an important part of the overall process.

In order to make this computation more efficient in the GPU architecture, the problem can be reduced to a dot product operation as follows. The GMM is defined as:

$$b(\mathbf{o}) = \sum_{c=1}^{C} \alpha_c \frac{1}{\sqrt{(2\pi)^d |\Sigma_c|}} e^{-\frac{1}{2}(\mathbf{o}-\mu_c)^T \Sigma_c^{-1} (\mathbf{o}-\mu_c)} \tag{2.1}$$

where $b(\mathbf{o})$ is the probability that the distribution generates the d-dimensional observation vector $\mathbf{o} = \{o_1, o_2, \cdots, o_d\}$, C is the number of Gaussians in the distribution, α_c is the weight of Gaussian c, μ_c and Σ_c are respectively the mean vector and the covariance matrix of Gaussian c. Recall that $\mathbf{x}^T$ denotes the transpose of the vector or matrix $\mathbf{x}$ and $|\mathbf{x}|$ denotes the determinant of the matrix $\mathbf{x}$.

The covariance matrix of a d-dimensional vector is a dxd matrix:

$$\begin{bmatrix} \sigma_{11} & \sigma_{12} & \dots & \sigma_{1d} \\ \sigma_{21} & \sigma_{22} & \dots & \sigma_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{d1} & \sigma_{d2} & \dots & \sigma_{dd} \end{bmatrix}$$

where σ_{ij} for $i \neq j$ measures how features i and j change together. The correlation between features is defined as the normalized covariance $\rho_{ij} = \frac{\sigma_{ij}}{\sigma_i \sigma_j}$. In the case of two uncorrelated features, $\sigma_{ij} = 0$. Note that σ_{ij} with $i = j$ is the variance of feature i. When features are not correlated with each other, the covariance matrix is reduced to a diagonal matrix.

For computational efficiency, the covariance matrix Σ_c is usually assumed to be diagonal. Thus, considering a diagonal covariance matrix, the log-likelihood of a single Gaussian component is:

$$\ln(b(\mathbf{o})) = \sum_{i=1}^{d} \left(\ln \alpha - \frac{1}{2} \ln((2\pi)\sigma_i^2) - \frac{1}{2} \frac{(o_i - \mu_i)^2}{\sigma_i^2} \right) \tag{2.2}$$

Note that subscripts indicating the mixture component have been omitted for clarity. By expanding $(o_i - \mu_i)^2$ and rearranging terms, we obtain:

$$\ln(b(\mathbf{o})) = \sum_{i=1}^{d} \left(\ln \alpha - \frac{1}{2} \ln((2\pi)\sigma_i^2) - \frac{\mu_i^2}{2\sigma_i^2} + \frac{o_i \mu_i}{\sigma_i^2} - \frac{o_i^2}{2\sigma_i^2} \right) \tag{2.3}$$

The first three terms are independent of the observations and can be considered a Gaussian-specific constant that can be readily pre-computed. Denoting this constant by h, it is:

$$h = \sum_{i=1}^{d} \left(\ln \alpha - \frac{1}{2} \ln(2\pi\sigma_i^2) - \frac{\mu_i^2}{2\sigma_i^2} \right) \tag{2.4}$$

The likelihood for a single Gaussian can thus be expressed as the summation of the constant and two dot products:

$$\ln(b(\vec{o})) = h + \sum_{i=1}^{d} \frac{o_i \mu_i}{\sigma_i^2} - \sum_{i=1}^{d} \frac{o_i^2}{2\sigma_i^2} \tag{2.5}$$

This is a dot product of augmented vectors from the observation vector and the Gaussian parameters:

$$\mathbf{obs} = (\bar{1}, o_1, o_2, \cdots, o_d, o_1^2, o_2^2, \cdots, o_d^2) \tag{2.6}$$

$$\mathbf{M} = (h, \frac{\mu_1}{\sigma_1^2}, \cdots, \frac{\mu_d}{\sigma_d^2}, -\frac{1}{2\sigma_1^2}, \cdots, -\frac{1}{2\sigma_d^2}) \tag{2.7}$$

where $\bar{1}$ is the identity element of multiplication. The log-likelihood of a Gaussian mixture distribution with C components is defined as :

$$\ln b(\vec{o}) = \bigoplus_{c=1}^{C} (\mathbf{obs} \cdot \mathbf{M}_c) \tag{2.8}$$

where $\bigoplus$ is the logarithmic addition and is defined as $\ln(e^x + e^y)$, which involves the computation of two exponentials. We can improve on the computation time of this term by approximating it with a term involving a single exponential, as shown in Algorithm 2.

Algorithm 2: Approximation of the logarithmic addition

input : a, b : two logaritmic values
output: The approximation of the logaritmic addition

```
1  if a = b then
2      return a + LN2
3  else
4      if a>b then
5          lga ← a;
6      else
7          lga ← b
8      diff ← −1 ∗ |b − a|
9      if diff > THRESHOLD then
10         lga ← lga + log(1.0 + e^diff)
11     return lga
```

The condition at lines 1-2 handles the case where both values are equal. In this case, the calculation is simplified to $LogAdd(a, b) = a + ln(2)$. Note that this is not true if both values are same-sign infinity. In the case where they are different, the logarithmic addition can be approximated by returning the biggest input value. If the difference between both values is less than a given threshold, the error is considered acceptable and no further calculation is made. Note that for an error of 0.000001, the threshold value is $-\ln(0.000001) = -13.82$. In the case where the error is unacceptable, the complete calculation must be performed. This case is handled by lines 8-10.

In the form presented here, the computation of acoustic probabilities is perfectly suitable for SIMD parallel architectures such as SSE registers[1] or GPUs since each distribution can be independently computed, and the results rest upon basic dot product operations.

[1] SSE is a SIMD instruction set extension to the Intel processor architecture allowing floating-point calculations to be performed in parallel

2.2 Computation of Acoustic Likelihoods on Multicore CPUs

On Intel processors, cross-products can be implemented on SSE registers. These registers can execute, in a SIMD fashion, 4 floating point operations concurrently. On multicore processors, each core has its own SSE registers allowing a straightforward multicore implementation.

Figure 2.2 shows two possible approaches for implementing the computation of acoustic likelihoods on multicore processors. The first method, shown in Figure 2.2(a), consists in distributing the computation of likelihoods of each frame among the cores. The second method consists in dedicating all computations of a given frame to a single core as shown in Figure 2.2(b). In this figure, d_i denotes distributions.

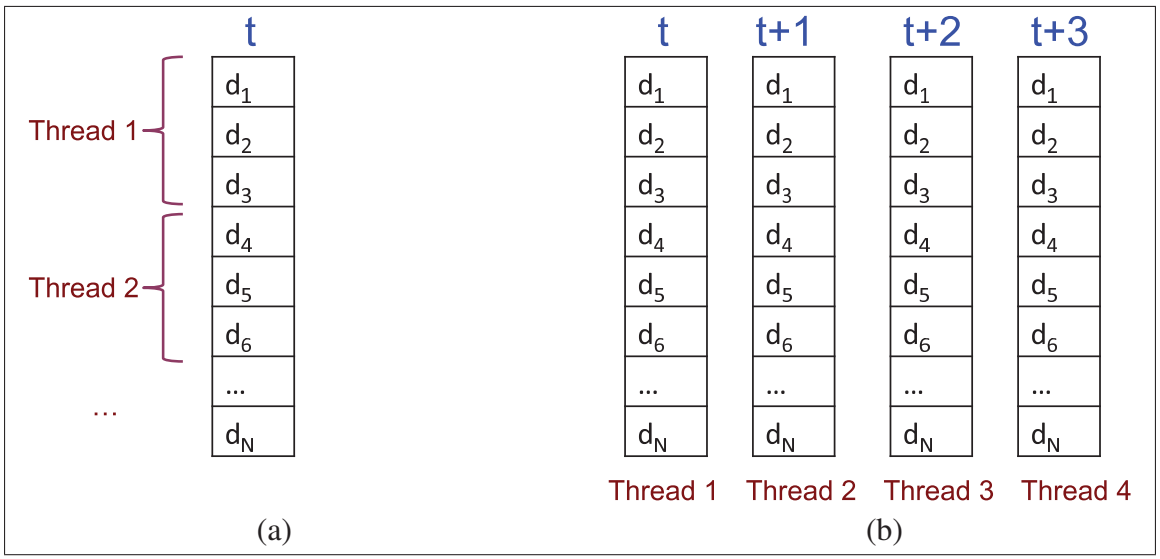

Figure 2.2 Different implementations on multicore processors. (a) All threads work on the same frame. (b) Each frame is dedicated to a thread.

Experiments have shown the second method to be slightly faster. This is because the second method is better suited for hiding the latency of the main memory since each thread is completely independent. With the first approach on the other hand, threads are simultaneously blocked at each frame since they have to wait for the transfer of the observation from main to cache memory. This acts as some kind of unwanted synchronisation point that slows down the entire process.

2.3 Computation of Acoustic Likelihoods on GPUs

2.3.1 Reduction Algorithm

The reduction algorithm is an important building block in parallel computing. This algorithm involves a reduction operator which takes two or more arguments and returns some combination of them. The addition and maximum operators are such operators. The reduction operator is iteratively applied until only one element remains. Figure 2.3 shows an example of reduction using the maximum operator.

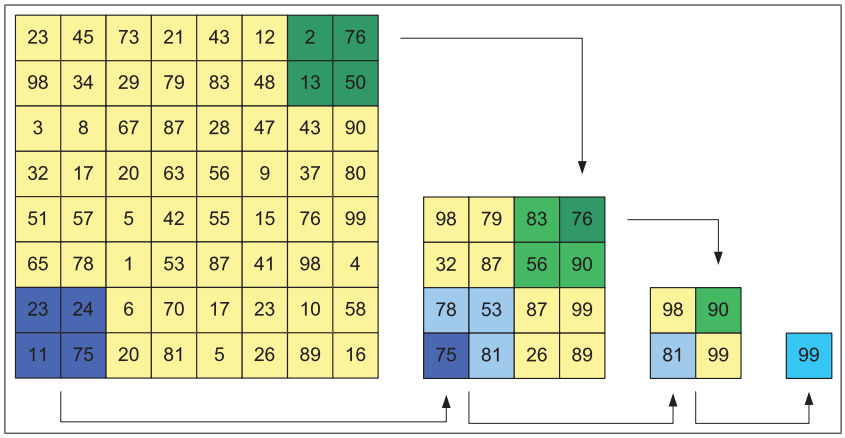

Figure 2.3 Reduction algorithm
Image is from Harris (2005)).

This operation can be implemented very efficiently on a GPU since all reductions are independent and can thus be executed in parallel. On a sequential CPU, this operation takes $O(n)$ where n is the number of elements in the set. On a parallel processor, the same operation takes $O(log_p n)$ where p is the number of processors.

2.3.2 Kernel for Acoustic Computation

As described above, the likelihood of a given mixture is the logarithmic addition of dot-products for each component of the mixture. This operation can be implemented as a reduction algorithm which uses the addition as reduction operator, except for the last C number of operations, for which the logarithmic addition is used to complete the reduction.

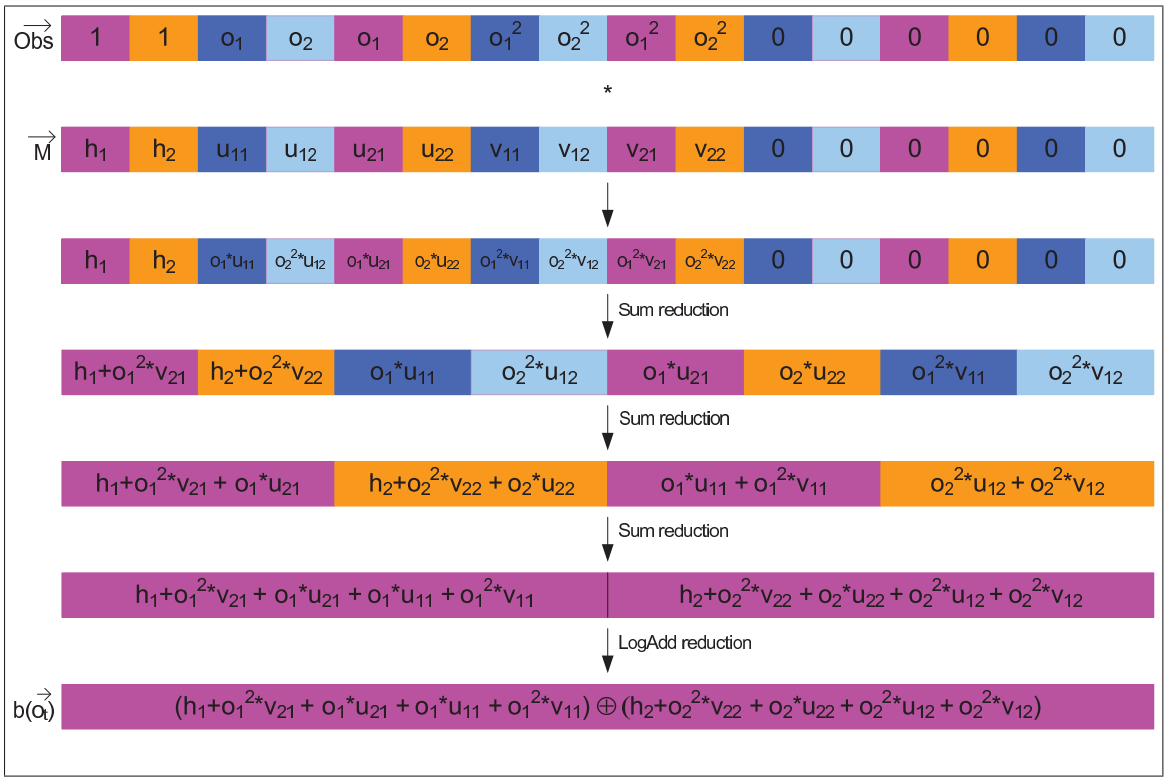

Figure 2.4 Reduction algorithm applied to the acoustic computation.

In the implementation used for this work, a block of 256 threads is dedicated to each mixture. Thus, the number of thread blocks is the number of GMMs in the model.

For efficiency, the observation vector $\vec{obs}$ is copied C times. As a result, it is the same length as a distribution vector. Since there is a direct correspondence between its elements and those of $\vec{M}$, index calculations are thus circumvented.

Moreover, to ensure efficiency of the reduction process and coalescing accesses to global memory, the model vector $\vec{M}$ is reorganized at the distribution level. It is organized in a way that the C first elements are the constants, followed by the $\mu_1\sigma_{11}^{-1}$ value of each component and so on. Figure 2.4 shows an example of the reduction algorithm applied in this context. In this figure, u_{xc} and v_{xc} denote the $\mu_x\sigma_{xx}^{-1}$ and $-\frac{1}{2}\sigma_{xx}$ values of component c. Note that the observation vector has also been reorganized in the same way to ensure consistency.

Algorithm 3 shows a possible implementation of kernel designed for 128-Gaussian mixture models. In the same way, the kernel for a 256-Gaussian model can be obtained by changing the addition at line 8 by a logarithmic addition.

Algorithm 3: Kernel for acoustic calculation

input : M : acoustic model, Obs : observation vector, $distSize$: size of a distribution
output: $Results$: contains the log likelihood of each distribution

```
1  tid ← threadIdx.x
2  __shared__ float aux[256]
3  baseIndex ← blockIdx.x * DistSize
4  for i ← 0; i < distSizeli ← i + 256 do
5      aux[tid] ← aux[tid] + M[baseIndex + tid + i] * Obs[tid + i]
6  syncThreads()
7  if tid < 128 then
8      aux[tid] ← aux[tid] + aux[tid + 128]
9  syncThreads()
10 if tid < 64 then
11     aux[tid] ← LogAdd(aux[x], aux[tid + 64])
12 syncThreads()
13 if tid < 32 then
14     aux[tid] ← LogAdd(aux[x], aux[tid + 32])
15     aux[tid] ← LogAdd(aux[x], aux[tid + 16])
16     aux[tid] ← LogAdd(aux[x], aux[tid + 8])
17     aux[tid] ← LogAdd(aux[x], aux[tid + 4])
18     aux[tid] ← LogAdd(aux[x], aux[tid + 2])
19     aux[tid] ← LogAdd(aux[x], aux[tid + 1])
20 if tid = 0 then
21     Results[blockIdx.x] ← aux[0]
```

The algorithm works as follows. The shared array declared at line 2 contains the results of the sucessive reduction. This array can be seen as a user-managed cache. The baseIndex variable contains the position of the distribution according to the block id. Recall that each block computes the likelihood of one distribution. The block at index 1 works on distribution 1, the block at index 2 on distribution 2, and so on.

The loop at lines 4-5 computes all multiplications and performs the first reductions to reduce the data size to 256 elements. The function $syncthreads()$ at line 6 ensures that all the intermediate computations are completed. The rest of the algorithm, lines 7-19, completes the reduction process with the exception that the last 32 reduction steps use the logarithmic addition as reduction operator.

This section could be implemented with a simple loop. However, as the reduction progresses, the number of required threads decreases. In a loop implementation, many threads will just pass through the loop without doing any operations. By using an unrolled implementation, these threads become available for other blocks running in the multiprocessor.

The last thing the algorithm does, at lines 20-21, is to save the reduction result in the results array at the right position according to the block index.

Note that a kernel for a 64-Gaussian mixture model can be obtained by changing the logarithmic addition at line 11 of Algorithm 3 by a normal addition. Indeed, a 64-Gaussian mixture model will have half fewer number of Gaussians which corresponds to one less iteration in the reduction process. Similarly, a kernel for the 256-Gaussian mixture model is obtained by changing the normal addition at line 8 of Algorithm 3 by a logarithmic addition.

Figure 2.4 shows how the reduction algorithm can be used to compute the likelihood of a distribution with a block of 4 threads (represented by different colors). In this simplified example, the observation is a 2-dimensional vector and the model is a 2-component Gaussian mixture model. Note that the LogAdd function implements the logarithmic addition. Our implementation is an approximation to avoid the computation of the two exponentials. The same algorithm is used in both the CPU and GPU implementations.

2.3.3 Consecutive Frame Computation

This algorithm can be improved by reducing the memory bandwidth used by the kernel. Indeed, the kernel downloads the model and the observation from the global memory to the GPU register at each frame. The amount of transferred data can be reduced by computing the likelihood

of several frames for each distribution. Figure 2.5 shows the speed-up obtained by computing several frames consecutively. This experiment has been conducted on a NVidia GeForce GTX295. This card contains two GPUs but only one has been used for this experiment.

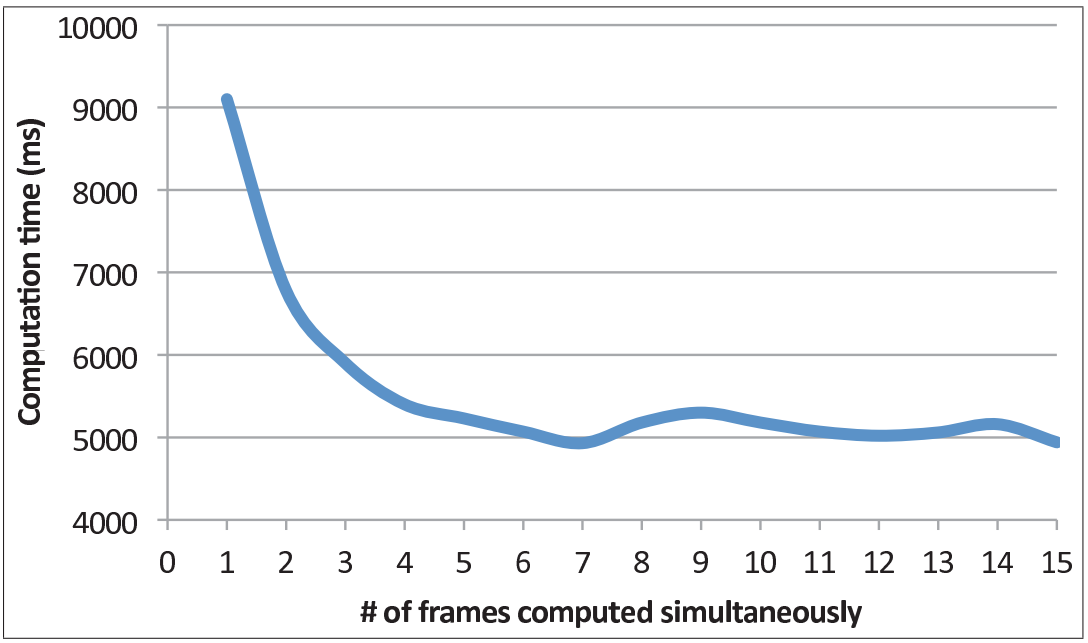

Figure 2.5 Speed-up when computing several frames consecutively.

The results show that the processing speed becomes stable when 7 frames are computed consecutively, which leads to a speed-up factor of 1.85 over the initial version of the algorithm. This result may vary on different GPUs because it depends on memory latency, memory bandwidth, the number of multiprocessors and the number of calculation units in each multiprocessor. Note that the number of frames that can be computed together is limited by the size of the shared memory.

Algorithm 4 shows a new version of the algorithm that handles the computation of log-likelihoods of several frames consecutively. Observations are stored consecutively in an array, as are the results, so that log-likelihoods of a given frame are consecutive in the array.

This algorithm works in the same way as Algorithm 3. The first loop at lines 4-7 computes the multiplications and starts the reduction algorithm to reduce the array to 256 elements. The only

Algorithm 4: Kernel for acoustic calculation on several frames consecutively

input : M : acoustic model, Obs : observation vector, $distSize$: size of a distribution, $numOfFrames$: number of frames to compute
output: $Results$: contains the log likelihood of each distribution

```
1  tid ← threadIdx.x
2  __shared__ float aux[256*numOfFrames]
3  baseIndex ← blockIdx.x * DistSize
4  for i ← 0; i < distSize; i ← i + 256 do
5      accValue ← M[baseIndex + tid + i]
6      for frame ← 0; frame < numOfFrames; frame ← frame + 1 do
7          aux[tid + frame * 256] ←
           aux[tid + frame * 256] + accValue * Obs[tid + i + frame * distSize]
8  syncThreads()
9  if tid < 128 then
10     for frame ← 0; frame < numOfFrames; frame ← frame + 1 do
11         aux[tid+frame*256] ← aux[tid+frame*256]+aux[tid+128+frame*256]
12 syncThreads()
13 if tid < 64 then
14     for frame ← 0; frame < numOfFrames; frame ← frame + 1 do
15         aux[tid + frame * 256] ←
           LogAdd(aux[x + frame * 256], aux[tid + 64 + frame * 256])
16 syncThreads()
17 if tid < 32 then
18     for frame ← 0; frame < numOfFrames; frame ← frame + 1 do
19         aux[tid + frame * 256] ←
           LogAdd(aux[x + frame * 256], aux[tid + 32 + frame * 256])
20         aux[tid + frame * 256] ←
           LogAdd(aux[x + frame * 256], aux[tid + 16 + frame * 256])
21         aux[tid + frame * 256] ←
           LogAdd(aux[x + frame * 256], aux[tid + 8 + frame * 256])
22         aux[tid + frame * 256] ←
           LogAdd(aux[x + frame * 256], aux[tid + 4 + frame * 256])
23         aux[tid + frame * 256] ←
           LogAdd(aux[x + frame * 256], aux[tid + 2 + frame * 256])
24         aux[tid + frame * 256] ←
           LogAdd(aux[x + frame * 256], aux[tid + 1 + frame * 256])
25 if tid = 0 then
26     for frame ← 0; frame < numOfFrames; frame ← frame + 1 do
27         Results[blockIdx.x + frame * gridDim.x] ← aux[frame * 256]
```

difference there is that a given value of the model array is used for several frames since it is downloaded one time at line 5 and used for each frame in the loop at lines 6-7. The remaining reductions are performed in the same way but are applied to several frames instead of only one, as was the case in the previous alogithm.

2.4 Results

Experiments have been conducted on an Intel Core i7 quad core processor and with the NVidia Geforce GTX 295 GPU to determine the efficiency of the use of the parallel architecture for computing acoustic likelihoods. Experiments have been conducted on a 44 minute test set. The acoustic model consisted of 4600 128-Gaussian distributions with diagonal covariance matrices. It has been trained with 171 hours of speech coming from French television programs in Quebec. More details about the experimental setup can be found in Section 4.2. Table 2.1 shows the results of these experiments.

Table 2.1 Parallel computation speed-up.

Step	architecture	Computation time (sec) CPU	Computation time (sec) Elapsed	speed-up factor
Acoustic likelihoods	CPU 1-core	11354	11354	–
	CPU 4-cores	12640	3161	3.6x
	GPU	457	457	24.8x

The first section of Table 2.1 shows that the parallelization of acoustic likelihoods works very well on both multicore processors and GPU. The speed-up of 3.6x on a 4-core processor approaches the theoretical maximum, which is the number of cores available in the processor. The use of GPU also leads to an interesting increase in performances with a speed-up factor of 24.8 over a single CPU core using the SSE registers. Note that the classical Viterbi decoder could also benefit from parallelized acoustic computations, as shown in Cardinal *et al.* (2009). However, the gain in overall performance of the decoder would be less significant since only

a small subset of distributions (usually less than half) is actually computed in an on-demand scheme.

2.5 Summary

This chapter has presented how acoustic likelihoods can be efficiently computed in parallel on both multicore processors such as the Intel Core i7 quad and a GPU.

The acoustic features are usually implemented as a set of GMMs, one for each basic unit, which is generally a triphone or a 5-phone. The task is mainly to compute the probabilty that a given GMM has produced the given observation. This chapter described how this computation can be done as a dot product, which is suitable for parallel architectures such as SSE registers and GPUs.

On multicore processors, each frame is dedicated to a core, which uses its SSE registers to compute the log likelihoods for all distributions. This approach led to a speed-up factor of 3.6 on a quad core processor over the single core version.

The same algorithm has been implemented on GPU using the reduction algorithm to compute the log likelihood of a single distribution. A speed-up factor of 24.8 over the SSE implementation on a single core CPU has been achieved.

CHAPTER 3

SEARCHING THE RECOGNITION NETWORK

Searching through the recognition network can be very time consuming, accounting for 30% to 70% of the total recognition time. The more complex the task, the more important the time dedicated to the search will be. For example, in the case of a small vocabulary isolated word recognition application, the computation time will be dominated by the computation of the acoustic likelihoods. On the other hand, in the case of a broadcast news transcription application - a much more complex task - the computation time will be dominated by the search in the recognition network. Figure 3.1 illustrates the place of the search procedure step in a speech recognition system.

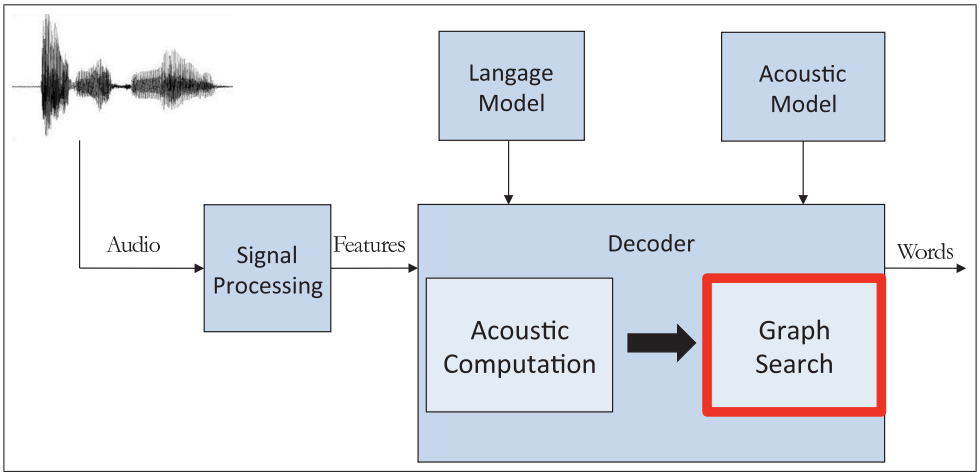

Figure 3.1 Graph search in a speech recognition system.

Since the processing times of practical applications of speech recognition are usually dominated by the network searching procedure, it is desirable to parallelize the computation of this step in order to accelerate the overall process. To that end, we can take advantage of parallel architectures such as multicore processors and GPUs. That being said, it is known that the parallel implementation of a search algorithm in a sparse graph is a difficult task, as discussed in Lumsdaine *et al.* (2007). It is particularly true in the context of speech recognition since:

a. the sparsity of the recognition network is high, with a branching factor of only approximately 3;

b. the size of the recognition network is too large to be exhaustively searched and must be pruned accordingly, which increases the sparsity of the graph;

c. the search is driven by an external factor, the acoustic likelihoods, which makes it impossible to arrange nodes and transitions in an efficient way.

This chapter describes how it is possible to efficiently implement the search in the recognition network. An introductory overview describing how the recognition network is built is first presented. Then, a parallelization algorithm of the classical Viterbi algorithm is presented and analyzed to point out its shortcomings with respect to efficient parallelization. The major part of this chapter describes how and why these difficulties can be overcome by using the A* algorithm instead of the Viterbi search. The chapter concludes with results detailing the parallelization efficiency that is achieved.

3.1 The Speech Recognition Network

Traditional speech recognition systems such as HTK are constructed using weighted automata. In speech recognition, the recognition network has many levels of representation. For example, possible sentences are represented by sequences of words that are themselves represented by sequences of phonemes. In the context of automata, these different representations are implemented using the substitution operation. For example, in the graph of words, a transition for a given word w is substituted by a subgraph representing its phonetic sequence. The major disadvantage of this approach is that a change in the network (for example, the addition of a new level of representation) implies that the program performing the search in the recognition network also has to be updated.

The composition operation allows FSTs to model many levels of representations in a normalized way. Therefore, the recognizer can work on different recognition networks (with different levels of representation) without updating the program itself.

This section presents how weighted transducers are used to construct a speech recognition system. The chapter begins with the description of each level of representation involved and how transducers implement them. Then, the method used to construct the knowledge network is discussed. Finally, the results obtained by experimentations are given.

Speech recognition is the process by which a computer identifies spoken words by analyzing the speech signal. To achieve this, it is assumed that the speech signal is a sequence of symbols composing a message. These symbols are called speech vectors or observations and are extracted from the speech signal at regular intervals of 10 ms. The aim of speech recognition is to map a sequence of vectors of observations to a sequence of symbols such as words, syllables or phonemes.

As discussed in Chapter 1, the main task of a speech recognition system is to compute:

$$\arg\max_{\mathbf{w}} p(\mathbf{w}|\mathbf{o}) = \arg\max_{\mathbf{w}} p(\mathbf{o}|\mathbf{w}) \cdot p(\mathbf{w}) \tag{3.1}$$

From the transducer's point of view, $p(\mathbf{o}|\mathbf{w})$ is a transduction between the message and observations. This transduction may involve several stages relating different levels of representation.

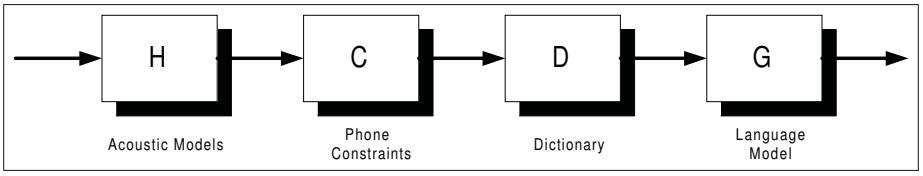

Figure 3.2 Transducers involved in speech recognition

Figure 3.2 shows the usual cascade of transducers used in speech recognition. Other intermediate transducers can be added to the chain. For example, transducers representing phonological rules should be added between transducers C and D. The meaning of each transducer will now be described.

3.1.1 Speech Recognition Transducers

3.1.1.1 Transducer H

Transducer H represents the constraints imposed by the HMMs used in speech recognition. HMMs can be used to model phones, syllables, words or any larger speech unit. Usually, context-dependent phones are used as the speech unit. A triphone is a phone modeled according to its neighbours. Triphones are denoted $a - b + c$ where b is the modeled phone, a and c are the neighbouring phones of b.

Transducer H maps a sequence of distributions to a sequence of triphone models (or of any other speech unit). Each triphone is typically modeled with a 3-state HMM. Transitions in a HMM carry a distribution index as an input symbol, the transition weight and no output symbol except for the transition leaving the HMM, which carries the triphone model associated with the HMM. Figure 3.3 shows the transducer H that is the union of all triphone models.

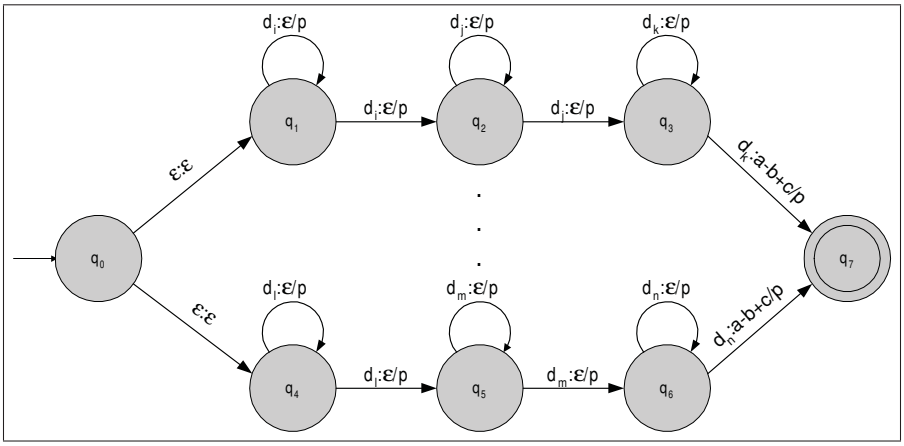

Figure 3.3 Observations to HMM transducer.

In this figure, p denotes transition probabilities involved in HMMs, $a-b+c$ is a triphone model and d_i is a distribution.

Note that the self-loop present on each state in the HMM can be omitted from the transducer and implemented implicitly in the decoder.

3.1.1.2 Transducer C

In practice, the number of triphones to model can be very high. Indeed, in English, there are 36 phonemes and thus the number of possible triphones is 36^3. In order to avoid modelling all triphones, only some of them are modeled with a HMM. Modelled triphones are called *physical* triphones and the others are referred to as *logical* triphones.

Logical triphones are mapped to physical ones according to a set of rules. This process is usually done using a decision tree. The first goal of transducer C is to implement this mapping. Figure 3.4 shows how this transducer is constructed.

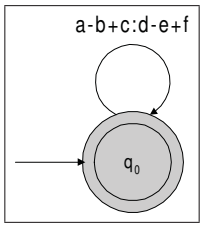

Figure 3.4 Transducer mapping physical triphones to logical ones.

The transducer has a self-loop transition for every triphone. The input symbol is a triphone, physical or logical, and the output symbol is the physical triphone associated with the input one. Thus, when the input triphone is a physical model, the output symbol is the same triphone.

The second goal of transducer C is to map a sequence of triphones to a sequence of phonemes. However, not all triphone sequences are allowed. A sequence of triphones A, B is allowed if the terminal pair of triphone A matches the pair at the beginning of triphone B. For example, the sequence $a-b+c, b-c+d, c-d+e$ is allowed while $a-b+c, c-d+e$ is not. Figure 3.5 shows how this restriction is implemented with a transducer.

Each state of the transducer implements a "memory" of the two previous phonemes in the sequence. Transitions leaving a state are those for which the two first phones composing the input triphone correspond to the state memory. All ingoing transitions of a state carry an input symbol such that the terminal pair coincides with the memory represented by this state.

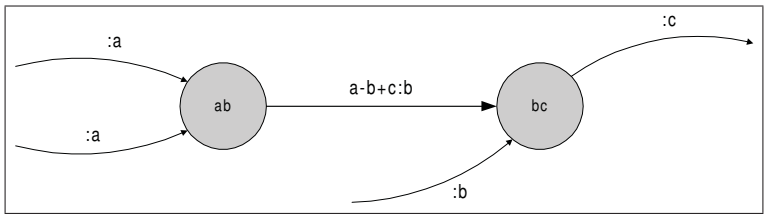

Figure 3.5 Transducer implementing triphone constraints.

3.1.1.3 Transducer D

In the context of speech recognition, the dictionary is a list of words with their phonetic transcriptions. Thus, the dictionary transducer implements the function $D : p^* \longrightarrow w$, which maps sequences of phonemes p to words w.

A string-to-string transducer is used to represent this relation. Figure 3.6 shows how this transducer is constructed.

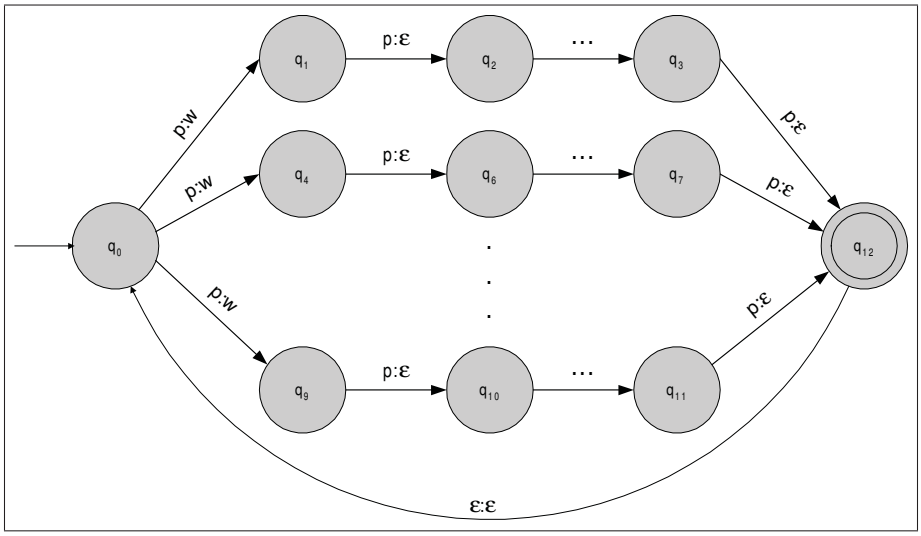

Figure 3.6 Dictionary transducer

In this figure, p is any phoneme and w is a word in the dictionary. The ϵ-transition leaving the final state to the initial state has been added to allow sequences of words. However, this loop transition induces an unbounded delay in the transducer when two words have the same pronunciation (homophones). This point will be discussed later.

3.1.1.4 Transducer G

Transducer G represents the language model. The language model gives a priori information about the probability of word sequences ($P(W)$). The transducer shown by Figure 3.7 implements a trigram model. In this model, the probability of a word given the two preceding words in the sequence is denoted $p(w_3|w_1w_2)$.

However, it is possible that a triple of words was not in the text used to train the language model. In this case, the probability of the word given the preceding word ($p(w_3|w_2)$) added to a penalty ψ_{w1w2} called the back-off penalty is used. Similarly, the unigram probability added to the back-off penalty is used when the bigram is also not available.

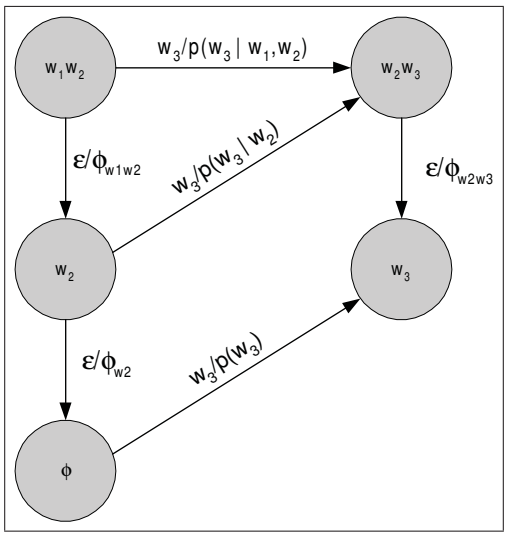

Figure 3.7 Language Model Transducer

In transducer G, each state encodes a ”memory" of two, one or no words. Transitions leaving a state q carry a word and the probability of this word given the words in the memory of q. In Figure 3.7, ϕ_{w1w2} denotes the back-off penalty for going to a unigram state (state with only 1 word memory).

Transducers can be used to describe other N-gram models such as bigram or 5-gram. They can also be used to describe other types of language models such as grammar-based syntactic structures.

3.1.1.5 Phonological Rules

In natural language, some phonological phenomena at the boundary of words such as the deletion or the insertion of phonemes happen frequently. These phenomena can be modeled with a transducer which can be inserted in the chain of transducers. An example of a phonological rule is that when the last phoneme of a word is t and the first phoneme of the following word is y, then t and y can be optionally replaced by the single phoneme ch. This rule applies to words "got you" which can be pronounced in two ways:

$$\text{g aa t} = y\ uw$$
$$\text{g aa} = ch\ uw$$

where the symbol $=$ denotes the word boundary. Figure 3.8 shows how this phonological rule can be implemented by a phoneme-to-phoneme transducer.

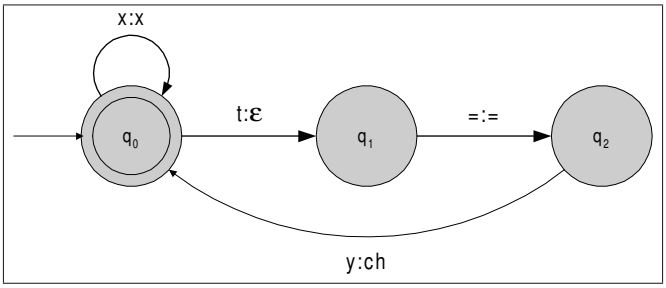

Figure 3.8 Transducer representing a phonological rule.

In this figure, the symbol x represents all phonemes in the language and the symbol $=$ is the word boundary. This transducer can be described as follows. All sequences of phonemes are accepted by the transducer thanks to the self-loop at the initial state. Moreover, the sequence $t = y$ is replaced by the phoneme $= ch$ since the transition leaving q_0 removes the phoneme t if it is followed by a word boundary and the phoneme y is replaced by ch if it follows the phoneme t and the word boundary. Thus, both sequences are accepted by the transducer that represents the phonological rule.

As noted before, phonological rules can easily be modeled in the recognition network by adding the transducers describing them in the chain of transducers between transducer C and transducer D. Taking into account phonological rules is crucial since a word can be pronounced differently in different contexts (surroundings). A classic example is the liaison[1] in French. WFST allows to optionally apply any phonological rules and select in which context a rule can be applied. It is also possible to add weigth to different rules with the effect of giving a priority to the most common.

3.1.2 Transducers Combination

The transducer HCDG is constructed using the composition operation. However, in the case of a large vocabulary system, the intermediate results grow very rapidly and there is not enough memory to perform the composition. The problem is solved by using the determinization operation since in the case of transducers used in speech recognition, the determinization considerably decreases the number of states and transitions by eliminating the number of redundant paths.

Therefore, the creation of HCDG proceeds in several steps. The transducer DG is obtained by the composition $D \bullet G$ and it has to be determinized. Recall that transducer D maps sequences of phonemes to words. The presence of homophones makes transducer DG non-determinizable since an unbounded delay is introduced. In particular, the presence of homophones comes from the fact that two or more different words have the same phoneme sequence. Figure 3.9 illustrates a disambiguated dictionary.

Auxiliary symbols, denoted $\#^i$ in the figure, are introduced to remove ambiguities. Henceforth, the transducer DG can be determinized and minimized. The next step is the composition $C \bullet DG$. However, the composition will fail since the auxiliary symbols added in D are unknown by C. Therefore, the markers have to be propagated along the cascade by adding to each state of transducer C a self-loop $(q, \#^i, \#^i, 0, q)$ for all i.

[1] Liaison is the pronunciation of a latent word-final consonant immediately before a following vowel sound (Wikipedia (2013a)).

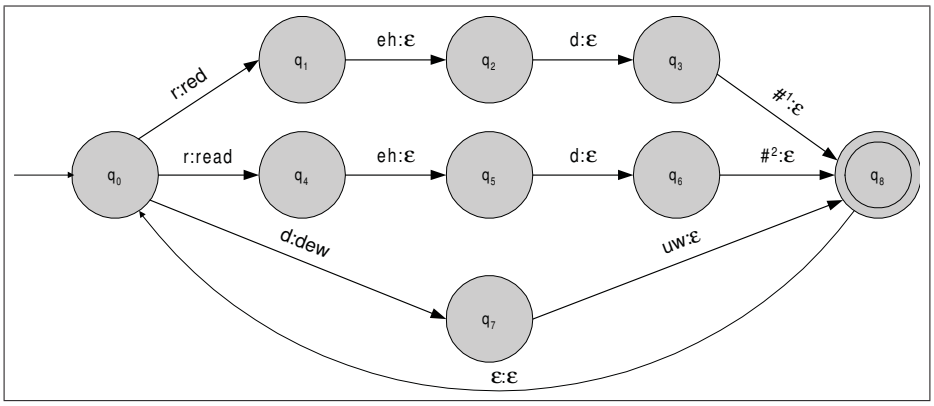

Figure 3.9 Disambiguated Dictionary Transducer

If the transducer C introduces new ambiguities, other auxiliary symbols have to be used. The same operations are repeated for all steps of the HCDG composition. Thus, HCDG is built according to the following computation:

$$HCDG = Min(Det(H \bullet Det(C \bullet Det(D \bullet G)))) \tag{3.2}$$

where Min denotes the minimization operation and Det is the determinization operation. Auxiliary symbols added during the construction of HCDG have to be removed at the end. The transducers shown in Figure 3.10 remove auxiliary symbols at the input and output by composing them with HCDG as follows: $L \bullet HCDG \bullet R$.

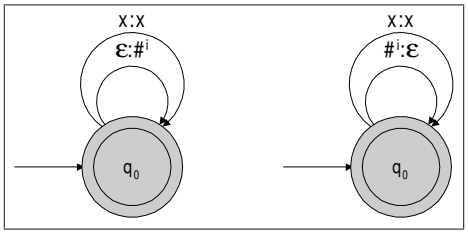

Figure 3.10 Transducers used to remove auxiliary symbols

In this figure, x denotes all non-auxiliary symbols.

3.2 Viterbi Algorithm

The Viterbi algorithm is a dynamic programming-based technique commonly used in speech recognition to explore the recognition network. Given the number of nodes and transitions in the graph, an exhaustive search is impossible in most applications. Consequently, at each time frame, states with a cost worse than a given beam value are marked inactive so paths passing through these states will not be searched further. This approach considerably reduces the computation time. The parallelization of the Viterbi algorithm is depicted in Figure 3.11.

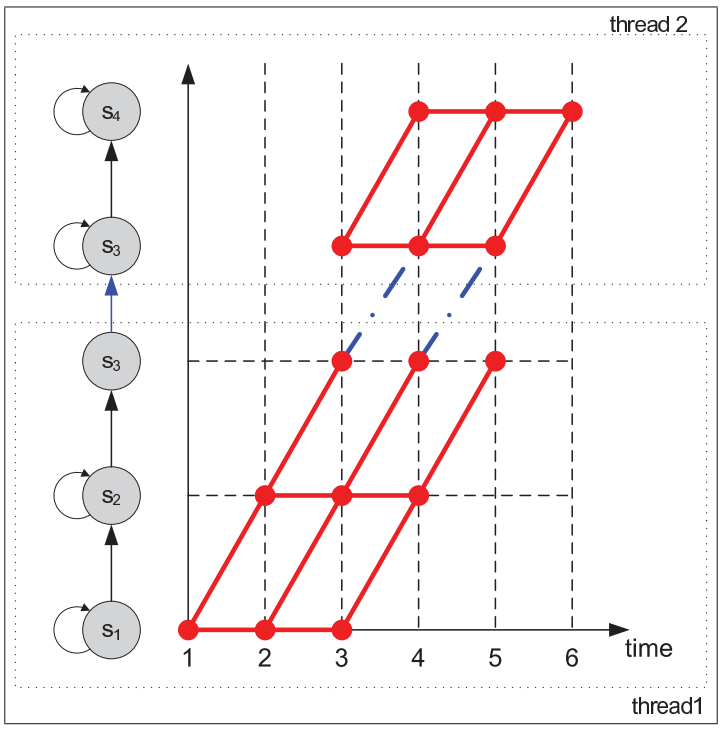

Figure 3.11 A parallel implementation of the Viterbi algorithm.

At each frame, the set of active states is divided into subsets and distributed among the available cores dedicated to the state expansion process as shown by Figure 3.11. Note that some states have transitions coming from states belonging to different cores. Updating these states simultaneously creates a race condition and can lead to data incoherency. We circumvented this problem by duplicating state information and merging them after all states have been expanded.

The overhead added by this approach is quite limited and it is much faster than protecting data with mutexes[2].

The technique allowed a speed-up factor of only 1.3 on a core2 quad processor over a single processor system. This result is mainly due to the sparsity of the active states in memory caused by beam pruning which leads to a misuse of the memory architecture. Recall that the Viterbi algorithm passes through active states s_i and expands them from time t to $t+1$ according to the following formula:

$$Min(ViterbiCost(s_j, t+1), ViterbiCost(s_i, t) + b_{ij}(o_t) + a_{ij}) \tag{3.3}$$

where $ViterbiCosts(s_j, t)$ is the best cost for reaching state s_j at time t from the initial state, $b_{ij}(o_t)$ is the acoustic probability for observation o_t computed with the distribution carried by the transition going from state i to j, and a_{ij} is the language model probability.

The mitigated speed-up can be traced back to active states that are not stored consecutively in memory since most of them have been pruned. Recall that when data has to be transferred from main to cache memory, a complete memory line has to be transferred. However, most of the transferred data is useless since they usually belong to states that have been pruned.

The parallel algorithm expands several states concurrently. However, the state expansion does not require a lot of calculations (three additions and one comparison). Consequently, cores perform several memory accesses concurrently and since the memory bus can be used by only one core at a time, cores are stalled several cycles for each state expansion. This considerably reduces the processing time and accounts for the modest speed-up of the parallel implementation. Since we cannot know which states will be used in advance, it is very difficult to overcome this problem. For the same reasons, GPUs will not be efficient in this situation.

[2]Synchronization mechanism that ensures that two threads do not enter into their critical section at the same time. A critical section can be, for example, a function that updates a variable in the shared memory space.

If, at the outset, the graph was small enough to allow for an exhaustive search, it would be much easier to implement a parallel version of it. This is the idea behind the use of the A* algorithm for which the heuristic is represented by a WFST.

3.3 A* Algorithm

The A* search algorithm (Russel and Norvig (1994)) can be seen as a combination of the Dijkstra algorithm (Cormen *et al.* (2001)), which always explores the state with the smallest distance already travelled, and the greedy best-first search (Russel and Norvig (1994)), which explores the most promising state, which is the closest one from the final state. Thus, the score of a state that takes into account both metrics is

$$Score(q,t) = g(q,t) + h(q',t+1) + cost(q,q',t) \tag{3.4}$$

where $g(q,t)$ is the score for reaching state q from an initial one at time t, h is the heuristic score that gives an estimate of the cost for reaching a final state from the adjacent state q' at time $t+1$ and $cost(q,q',t)$ is the cost for going to q' from q at time t. This algorithm always finds the shortest path in the graph if the heuristic is admissible. To be admissible, the heuristic has to satisfy the following condition

$$h(n,t) \leq h'(n,t), \forall n$$

where $h'(n,t)$ is the actual cost for reaching a final state from state n at time t. Thus, a heuristic is said to be admissible if, for every state, it underestimates the actual cost for reaching the final state. A pseudocode of the A* algorithm is shown in Algorithm 5. For simplicity, epsilon transition handling has not been illustrated in this algorithm.

The first input of the algorithm is the HCLG recognition network composed of HMMs (H), triphone context dependency (C), lexicon (L) and a trigram backoff language model (G). This network is represented by a WFST for which input symbols are distributions and output symbols

Algorithm 5: The A* algorithm

```
1  openList ← {((i, λ, 0), heuristic(i, 0))}
2  closedList ← ∅
3  while openList ≠ ∅ do
       // Extract state with lowest score
4      (q, t, g) ← openList.Extract()
5      if (q, t, g) ∈ closedList then
6          if g > closedList.get((q, t)) then
7              Go to next state;
8          end
9      end
10     closedList ← closedList ∪ (q, t)
11     if q ∈ F and t = numFrames then
           // Best path found
12         ExitSearch()
13     end
14     foreach (q, σ_i, σ_o, w, q') ∈ E[q] do
15         if (q', t + 1) ∉ closedList then
16             g' ← g + obsCost(σ_i, t) + w
17             h ← heuristic(q', t + 1)
18             entry ← (q', t + 1, g')
19             score ← g' + h
20             openList ← openList ∪ {(entry, score)}
21         end
22     end
23 end
```

are words (Mohri *et al.* (2000, 2002)). The second input is the heuristic function $h : q, t \rightarrow \mathbb{R}$, which gives the estimated cost for reaching a final state from state q at time t.

The algorithm works as follows. The algorithm is first initialized by adding initial states to a priority queue. This queue maintains alternate paths along the search. The most promising state q is extracted from the *open list* at line 5. The state is not expanded if it has already been explored and its current score is not better than the previous one found in the *closed list* (lines 6-10). Otherwise, scores of adjacent states of q are updated and added to the *open list* (lines 16-25). The algorithm explores states in the open list until a complete path has been found (lines 12-14).

3.3.1 Unigram Language Model Heuristic

The fundamental characteristic of the A* algorithm is the heuristic. Indeed, the better the heuristic, the faster the search will be. The chosen heuristic is a WFST built with same HMM topology and dictionary. The only difference is that a unigram model has been used instead of a trigram model. Figure 3.12 shows how both types of language models are implemented in a WFST-based recognition network. In this figure, w_i denotes a word in the dictionary, ϕ_{w_1,w_2} is the backoff penalty when the history w_1, w_2 does not exist, $p(w_i|w_1, w_2)$ is the probabilty of w_i given the history w_1, w_2 and the symbols $< s >$ and $< /s >$ denote the beginning and end of sentences.

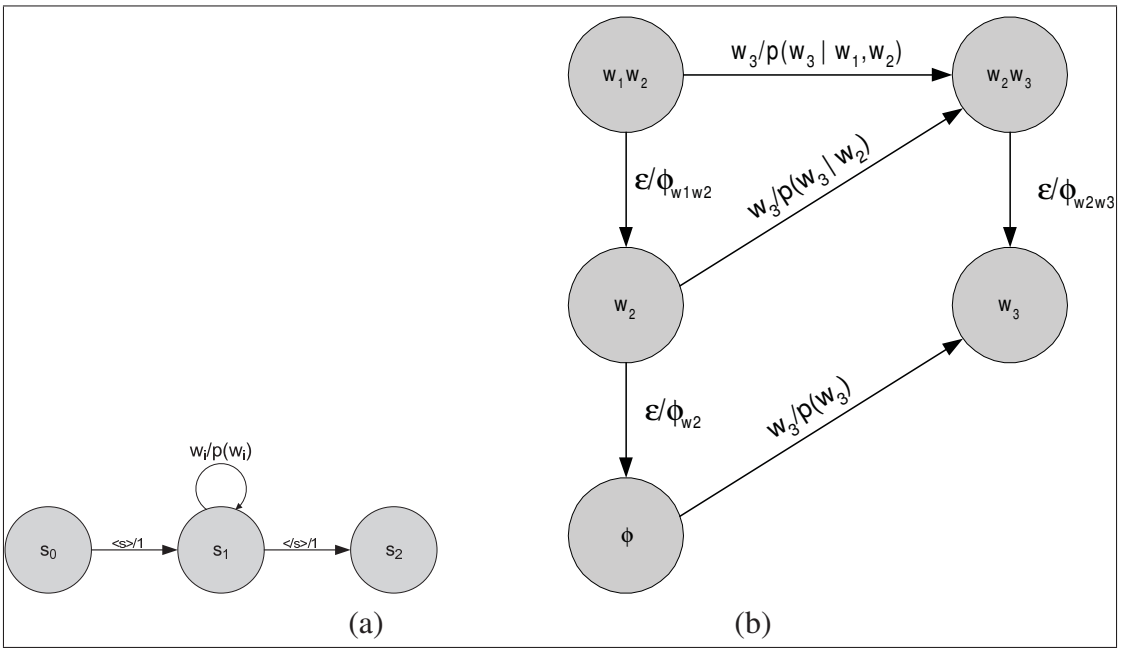

Figure 3.12 Representation of language model with WFST (a) Unigram language model (b) Trigram language model

The unigram model is straightforward. Each word of the vocabulary and its probability is represented by a self loop at state q_1. Thus, this model accepts every word sequence starting and ending with symbols $< s >$ and $< /s >$, respectively.

The trigram model shown in Figure 3.12(b) is much more complex. For simplicity, this figure shows how the word w_3 is modeled. This model is repeated for each word of the vocabulary,

which leads to a much more complex network as compared to the unigram case. When the complete trigram model is composed with the rest of the network, the number of nodes and transitions explodes. Table 3.1 shows the difference between the recognition and heuristic WFST.

Table 3.1 Comparison of trigram network WFST and heuristic WFST sizes.

	# of states	# of arcs
Trigram network	9 148 722	34 499 962
Heuristic unigram network	181495	555 896

The heuristic is thus a WFST that accepts the same sequence of distributions but produces a sequence of words that is mostly unconstrained. Even if this kind of recognition network leads to a very low accuracy in large vocabulary speech recognition systems, it should be enough to give a good indication on the paths to explore on the real recognition network. Since the A* search uses an approximation of the cost for reaching a final state from a given initial state at a given time, the heuristic costs are computed by performing backward Viterbi decoding and since the unigram-based recognition network is small enough, it allows an exhaustive search.

Note that application of the Viterbi algorithm on the heuristic is simpler and faster than on the recognition network because no backpointers need to be kept to retrace the best state sequence. Moreover, since all states are explored at each frame, they reside in contiguous memory locations allowing optimal cache usage. To compute heuristic scores, acoustic likelihoods for all distributions are needed. An efficient way to compute them in parallel was the subject of Chapter 2. As far as the author knows, it is the first time that a unigram-based WFST is used for computing heuristic costs.

However, the direct use of the unigram probabilities leads to a non admissible heuristic. Recall that the trigram probability is defined as

$$p(w_i|w_{i-1}, w_{i-2}) = \frac{C(w_{i-2}w_{i-1}w_i)}{C(w_{i-2}w_{i-1})}, \tag{3.5}$$

which is the apparition frequency of word w_i in the context of the word history $w_{i-2}w_{i-1}$. In this equation, $C(x)$ denotes the number of times the word sequence x has appeared in the training database. On the other hand, the unigram probability is defined as

$$p(w_i) = \frac{C(w_i)}{\sum_{w \in W} C(w)}, \tag{3.6}$$

which is the apparition frequency of word w_i in the training database.

Recall that the perplexity measures the mean branching factor of a language model given a word sequence and the perplexity is defined as the reciprocal of the mean probability of a word sequence (equation 1.30). This effect can be visualized in Figure 3.12. The branching factor of the trigram model is smaller than the unigram one. Consequently, the unigram probability of a given word sequence is, on average, always lower (and thus, the corresponding cost higher) than the trigram probabilities. In the general case, N-gram language models have a higher perplexity than those of smaller values of N. This is a well established rule that, barring some rare exceptions, is borne out in practice. This means that the heuristic network always overestimates the word sequence probabilities of the recognition network. Since the A* search algorithm requires that the heuristic underestimates these probabilities, the unigram probabilities cannot be directly used.

To circumvent this problem, we built the unigram by assigning to each word the greatest value among the unigram, bigram and trigram probabilities. That means that the probability $p_h(w_i)$ assigned to each word w_i in the heuristic network is the largest one from the set of all probabilities for w_i in the trigram model:

$$p_h(w_i) = \max \begin{cases} p(w_i) & \\ p(w_i|w_{i-1}) & \forall w_{i-1} \\ p(w_i|w_{i-1}, w_{i-2}) & \forall (w_{i-1}, w_{i-2}) \end{cases} \tag{3.7}$$

where $p(w_i)$ is the unigram probability, $p(w_i|w_{i-1})$ is the bigram probability of w_i given the history w_{i-1} and $p(w_i|w_{i-1}, w_{i-2})$ is the trigram probability of w_i given the history (w_{i-1}, w_{i-2}). This approach ensures that there is no word sequence with a lower probability in the heuristic compared to the recognition FST. That makes the heuristic admissible.

3.3.2 Mapping Recognition FST States to Heuristic States

Recall that the A* search uses the heuristic cost given by the function $h(q_r, t)$, where q_r is a recognition FST state. In essence, this function performs a lookup in the Viterbi treillis computed on the heuristic. Thus, we need to know which state (q_h, t) in the heuristic is equivalent to (q_r, t). A mapping between states of the heuristic and those of the recognition FST must thus be discovered.

Both the heuristic and recognition FST map a sequence of distributions to a sequence of words. Since both FSTs are built with the same HMM, the same context dependency rules and the same list of words, they both translate the same sequence of distributions to a sequence of words. This characteristic can be used for building the mapping between states, considering that a sequence of states representing a sequence of distributions in the recognition FST should be equivalent to a sequence of states representing the same sequence of distributions in the heuristic FST. The word sequences produced by both FSTs can be ignored since they are useless for establishing the mapping.

The FST composition is used to establish this mapping. The inverted (input and output symbols swapped) heuristic FST is composed with the recognition FST so that output sequences of distribution symbols from the inverted heuristic FST match input sequences of the recognition FST. Figure 3.13 shows an example of simple FSTs composed together. In this figure, output symbols have been omitted, making this operation equivalent to automata intersection.

The description of the algorithm can help develop the intuition on the manner it can be used to compute the mapping between the recognition and heuristic FSTs. Firstly, the inital state (p_0, q_0) of $\mathcal{A}_3$ is created from initial states of both FSTs. From both initial states, there is an

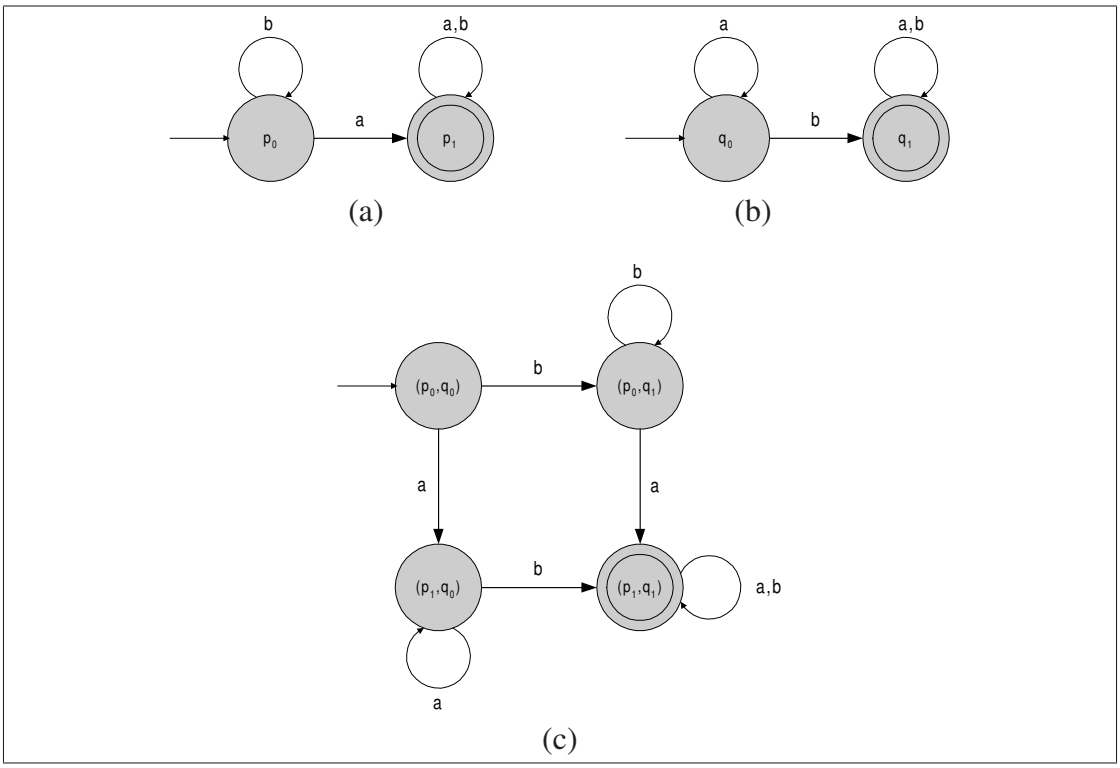

Figure 3.13 Simple example of automata intersection. (a) and (b) Input automata $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively. (c) $\mathcal{A}_3$, the intersection of $\mathcal{A}_1$ and $\mathcal{A}_2$

outgoing transition carrying the symbol 'a'. In $\mathcal{A}_1$, this transition goes to state p_1 while in $\mathcal{A}_2$, it is a self loop. The algorithm thus creates a new transition originating from (p_0, q_0) and going to (p_1, q_0) in $\mathcal{A}_3$, which represents the intersection of the path going from p_0 to p_1 in $\mathcal{A}_1$ and the one going from p_0 to p_0 in $\mathcal{A}_2$. With the same approach, the state (p_0, q_1) is created from a transition carrying the symbol 'b' going out the initital state on each automaton. In the same way, a transition from (p_0, q_1) to (p_1, q_1) is created because there is a transition carrying the symbol 'a' from p_0 to p_1 in $\mathcal{A}_1$ and from q_1 to q_1 in $\mathcal{A}_2$. The process is iteratively repeated for each state in $\mathcal{A}_3$ until all new states have been created.

More formally, a state in the composed FST is a pair $s_{hr} = (q_h, q_r)$ where q_h and q_r are, respectively, states of the heuristic and recognition FST. The existence of a state (q_h, q_r) implies that at least one path from i_h to q_h in the heuristic FST has the same distribution sequence as a path from i_r to q_r in the recognition FST. Since the composed FST is connected, there is also a path from q_h to a final state of the heuristic FST that has the same distribution sequence as

a path from q_r to a final state of the recognition FST. Consequently, both states are considered to be equivalent. Note that the FST resulting from the composition is not used, only the list of state pairs is useful. In addition, this mapping is computed offline.

A problem may arise since some recognition FST states are mapped to several heuristic states. This happens when a distribution sequence common to both FSTs corresponds to several paths in the heuristic FST, for example two different word sequences having the same phonetic transcription. This problem has been circumvented by adding word markers in the FSTs before composition to remove these ambiguities. After composition, the markers are replaced by epsilon labels.

This approach reduces a large part of ambiguous mappings, but not all of them. To deal with the remaining ambiguous mappings, we select the heuristic state with the smallest heuristic score. This approach guarantees that the heuristic remains admissible.

The drawback of this technique is that it decreases the discriminative power of the heuristic. If it were possible to know the heuristic state that best matches the FST state in the current context, we might have a better approximation of the remaining distance to a final state. This would allow to safely prune this state and thus, reduce the search time.

3.3.3 Block Processing

A major problem of the decoding procedure is the exponential growth of the number of states to explore when the number of frames increases. This is essentially the same problem that beam pruning solves in a classical Viterbi decoder.

In addition to the large number of states to explore, data structures required in the implementation of the A* search are significantly more complex than the simple arrays used in a Viterbi decoder. As described earlier, the A* search always explores the most promising state first. The most efficient way to extract smallest-cost nodes is to store them in a binary heap. Indeed, the insertion of a new key, the extraction of the smallest one and decreasing the value of a key already in a heap are $O(\log n)$ operations, where n is the number of elements in the heap.

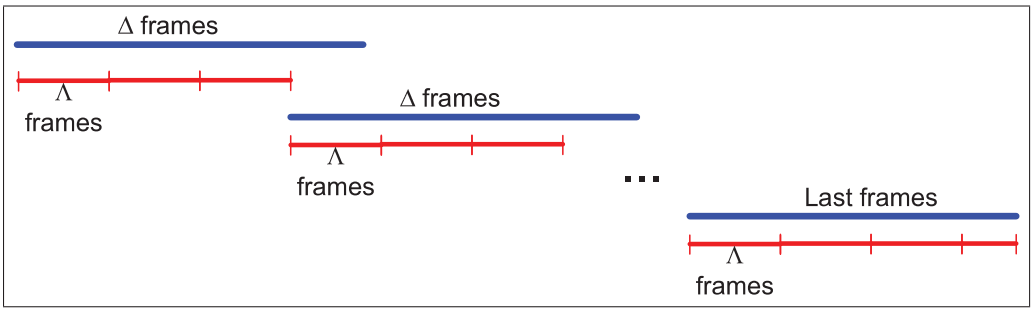

Figure 3.14 A* search by blocks of frames.

By contrast, searching a particular node is $O(n)$, making the verification of node existence an expensive operation. A hash table is thus used to keep track of nodes in the open list. In order to avoid the exploration of a given state several times, a closed list containing all nodes already explored is used. This closed list is also implemented with a hash table.

Both problems can be avoided by implementing a block approach. Firstly, the heuristic is computed for Δ frames. This set of frames is called the heuristic window. Then, the A* search is performed on a smaller window of $\Lambda < \Delta$ frames, called the search window. The extra frames $\Delta - \Lambda$ are the lookahead and they are not searched except for the last frames of the audio. The search in a window stops when a node at time Λ with a cost (path cost+heuristic cost) larger than the best cost plus the beam is extracted from the open list.

The window is then advanced by Λ frames. The search in the next window is initialized with the states that survived pruning at the last frame of the previous window. The process is applied until the end of the audio is reached. In order to save computation time, several consecutive searches can be performed with one heuristic computation as shown in Figure 3.14.

This approach is equivalent to beam pruning in the Viterbi algorithm. In order to limit the number of nodes in the open list, those with a cost outside of the beam are not included. Moreover, when the heap is full, nodes with larger costs are simply removed. This operation is efficiently performed ($O(1)$) in a heap by moving the last element index at the appropriate position.

In its simplest form, there is only one search window per heuristic window. Our experiments show that a minimum of 20 frames with a lookahead of 15 frames is necessary to obtain good results. The use of several search windows per heuristic window reduces the processing time. Indeed, each time the heuristic window is computed, the lookahead frames have to be recomputed. Thus, by applying several searches per heuristic window, this overhead is reduced.

The length of the heuristic window is mainly restricted by the available memory. Recall that the heuristic is the score for each state at a given time, a task that consumes a large amount of memory. If memory is not an issue, the heuristic can be computed on the complete utterance. Generally, the length of the heuristic window is a tradeoff between memory usage and computation overhead induced by the needs of a lookahead.

The length of the search windows is in turn used to control the exponential growth of states in the search space. A larger window will significantly increase both the search time and the required memory since the search space is pruned only at the end of a search window. A window that is too small will prune too many hypotheses leading to poor results.

3.3.4 Heuristic Decoding Parallelization

As mentioned before, computation of the heuristic uses the classic Viterbi algorithm on the reversed graph. The algorithm passes through all transitions, sorted with respect to their destination states and updates the cost of destination states. This cost is called the Viterbi cost.

Parallelization is obtained by dividing the set of transitions in subsets, with one for each thread. Each thread then performs the expansion of states with transitions belonging to its subset. If the number of subsets is equal to the number of cores in the processor, no synchronization method is needed. The source states are read only and thus, there is no need for a synchronization mechanism. However, it is possible for a destination state to be in two different subsets, which could lead to a race condition. However, transitions are sorted with respect to their destination state and are accessed in this order. Consequently, if a destination state belongs to two threads, one of them will be updated at the beginning of its thread life while the other one at the end of its thread life. Thus, they will not be accessed simultaneously. This is no longer true if there

are more subsets than the number of cores since we cannot know which subset of transitions will be expanded first.

3.3.5 Consecutive Block Computing

Cardinal *et al.* (2012b) report a speed-up by a factor of 3.1 on a 4-core processor in the parallelization of the heuristic computation using the window approach. It is a good improvement over the parallelized version of the Viterbi algorithm but further investigation showed that this approach does not efficiently use the memory architecture since only destination states are consecutive in memory. Recall that for each transition from state s_i to state s_j, the new score of s_j is :

$$Min(ViterbiCost(s_j), ViterbiCost(s_i) + b_{ij}(o_t) + a_{ij}) \tag{3.8}$$

where $b_{ij}(o_t)$ is the observation cost associated with transition a_{ij}.

The updating process requires four different memory accesses for each transition: the transition itself, the Viterbi cost of the source, the Viterbi cost at destination state and the acoustic likelihood associated with the transition. Since transitions are accessed in their memory order and they are sorted with respect to their destination state, these accesses are optimally using the cache system.

However, the source states are not consecutive in memory as depicted in Figure 3.15a. In this figure, arrays represent the Viterbi costs at time t and $t + 1$. For example, the Viterbi cost for state q_0 at time $t + 1$ is the lower of its actual cost and the expanded one, which is the sum of the Viterbi cost at state q_{85} at time t, the transition cost and the likelihood of the distribution associated with this transition. The acoustic likelihoods also lead to an inefficient use of the memory architecture since they are usually randomly accessed.

To ensure an efficient use of the cache memory, it is possible to take advantage of the fact that the heuristic computation in each window is completely independent. However, dedicating one heuristic window per thread leads to the same memory inefficiencies.

A more efficient solution is to compute the expansion from time t to $t+1$ in several heuristic blocks consecutively. With this approach, Viterbi cost arrays of different blocks have to be merged in such a way that scores of a state q in different blocks are consecutive in memory, as shown in Figure 3.15b. In this figure, $q_{x,y}$ denotes the state q_x in block y.

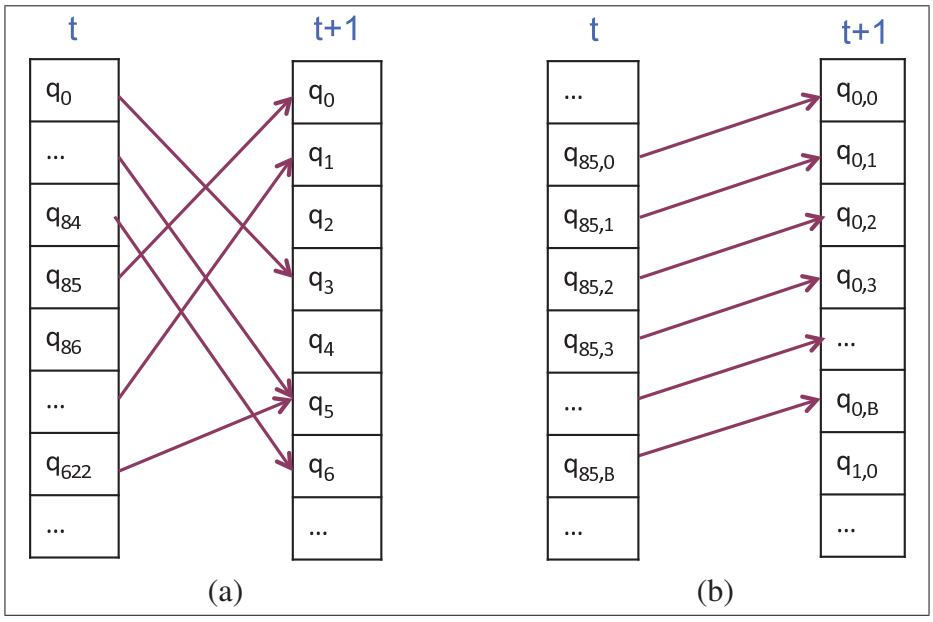

Figure 3.15 Memory accesses for (a) one heuristic window decoding and (b) several heuristic windows decoding.

The new version of the Viterbi algorithm works as follows. A transition $(q, \sigma_i, \sigma_o, w, q')$ is selected in the heuristic FST (in our implementation, transitions are sorted with respect to their destination state q'). The new score for state q' is then computed using equation 3.8. Then, the same transition is used to do the same computation in the next heuristic window. Thanks to the organization of Viterbi scores shown in Figure 3.15b, all needed data is already in the cache. Even in the sequential implementation, this approach leads to better performance. Note that the likelihoods computed before have to be stored in such a way as to be accessed sequentially in memory.

On Intel Core i7 architectures, the optimal number of blocks processed consecutively is 16. Indeed, Viterbi costs and observation costs use four bytes each. Since a cache line contains 128 bytes on Core I7 processors, 16 blocks will use a complete cache line for each cost information accessed by the algorithm.

There are two strategies for parallelizing this algorithm. A first strategy is to delegate one or more heuristic windows to each thread. All threads will work with the same transition on its assigned heuristic window. The second strategy delegates transitions to the thread and each one computes expansions in all heuristic windows. Our experiments have shown that the latter strategy offers better performance.

3.3.6 Computing Heuristic Costs on GPUs

This simplified version of the Viterbi algorithm used for computing heuristic costs can be efficiently implemented on GPUs. However, to take advantage of the GPU architecture, the parallelization approach differs. As discussed before, it is highly important to ensure that memory accesses by threads belonging to the same GPU block (thus, executed by the same multiprocessor) are coalesced. To achieve that, all threads in a given block work with the same transition and each thread performs the expansion of a state in a specific heuristic block as shown by Figure 3.16. With this approach, the optimal number of blocks processed consecutively is the warp size, which is 32 in current NVidia GPUs.

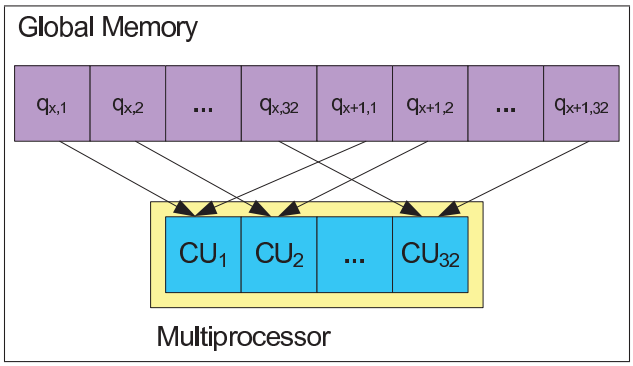

Figure 3.16 Parallelization of heuristic computation on GPUs

As shown in Figure 3.16, several transitions are assigned to each block of threads. Since it is impossible to know in advance on which multiprocessor any given block is executed nor in which order they are executed, it is important to ensure that no destination state is accessed by two threads simultaneously. It is however possible to use atomic functions for writing results; the best way to meet this condition is to assign all transitions associated with the same destination state to the same GPU block.

Another point to take into consideration is the amount of memory that is transferred from the host computer and the GPU. Recall that at each frame, the best Viterbi score for each state is stored in an array. Thus, for each frame, an array of $|Q|$ elements, where $|Q|$ is the number of states in the heuristic, have to be transferred from GPU global memory to CPU main memory. In the case of the unigram heuristic presented in this section, that means approximately 23MB (181485 states multiplied by 32 blocks, multiplied by 4 bytes per element) per frame. On a PCI express 2.0 bus, for which the maximum bandwith is 16 GB/s, that means approximately 1.4 ms per frame is needed, in the best case, to transfer results to the CPU main memory. It is an important point to take into consideration since that represents approximately 20% of the total time (computing the heuristic scores alone takes 5 ms per 32 frames on GTX295).

Fortunately, it is possible to transfer memory between the host and the GPU while kernels are being executed. Since Viterbi costs computed at a given time are not used later in the process, they can be transferred while the rest of the calculations are being performed. This allows one to hide the transfer time and leads to a significant improvement on the overall computation time.

Figure 3.17 shows the interactions between the CPU and GPU for computing heuristic costs in the GPU. In this figure, blue arrows denote data transfer between the host and the GPU, orange arrows denote the launch of kernels. Note that time t is relative to the beginning of the heuristic block. It is assumed there is enough memory for all frames and likelihoods. This is usually not the case but it is trivial to compute heuristic costs but on a subset of frames and then executing the kernel iteratively on each subset to obtain the final results.

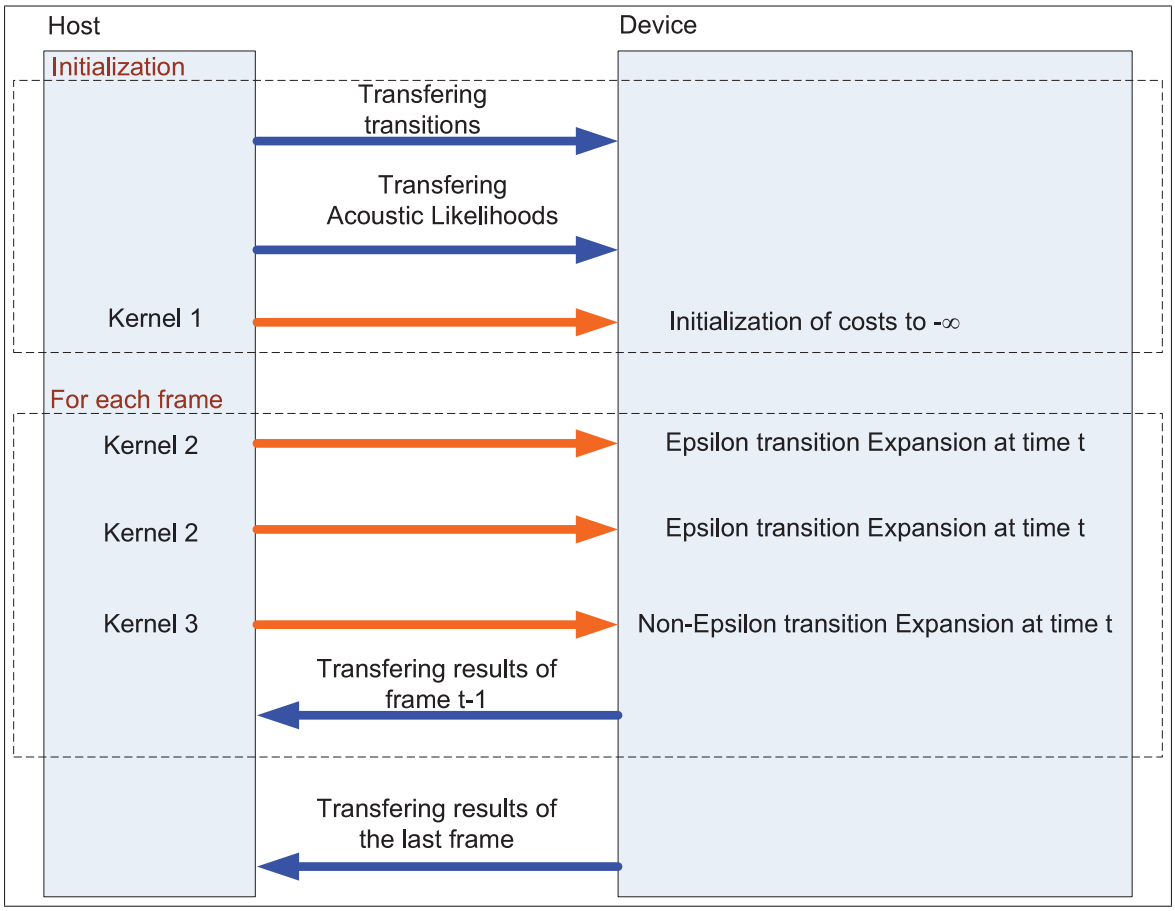

Figure 3.17 Diagram of operations involved in the heuristic costs computation in a GPU.

Note that during the initialization, both transitions and log-likelihoods are transferred in the GPU global memory. In the context of the speech recognition system, transitions can be transferred during the initialization of the system since they are the same for every audio. However, log-likelihoods have to be transferred every time a new decoding is needed.

Figure 3.17 also shows that the kernel 2 used to compute the expansion of epsilon transitions is called twice. This is to ensure that all expansions have been completed (in the case of consecutive epsilon transitions). The number of times the kernel must be called is constant for a given heuristic WFST. The number of times the kernel has to be executed should be verified every time a new heuristic WFST is used. However, the error in heuristic costs by missing iterations does not have a significant impact on the A* search. This number of iterations can be considered as a tuning parameter.

3.4 Real-Time Transcription

In some applications, such as closed-captioning of live television shows, it is desirable to output transcriptions in real-time instead of waiting until the end of the speech session. This feature can be implemented by windowing the Viterbi search. With this approach, a partial transcription can be produced for δ frames after a lookahead of λ frames have been explored. Consequently, the first partial transcription will be produced at time $\delta + \lambda$ and a new one will be produced every δ frames. The transcription at time $T - (\delta + \lambda)$ is produced by finding the state with the lowest cost at time T, then backtracking along the path that led to this state and outputing words between $T - (\delta + \lambda)$ and $T - \lambda$. Figure 3.18(a) shows an example of this process for which both δ and λ are set to 4 frames.

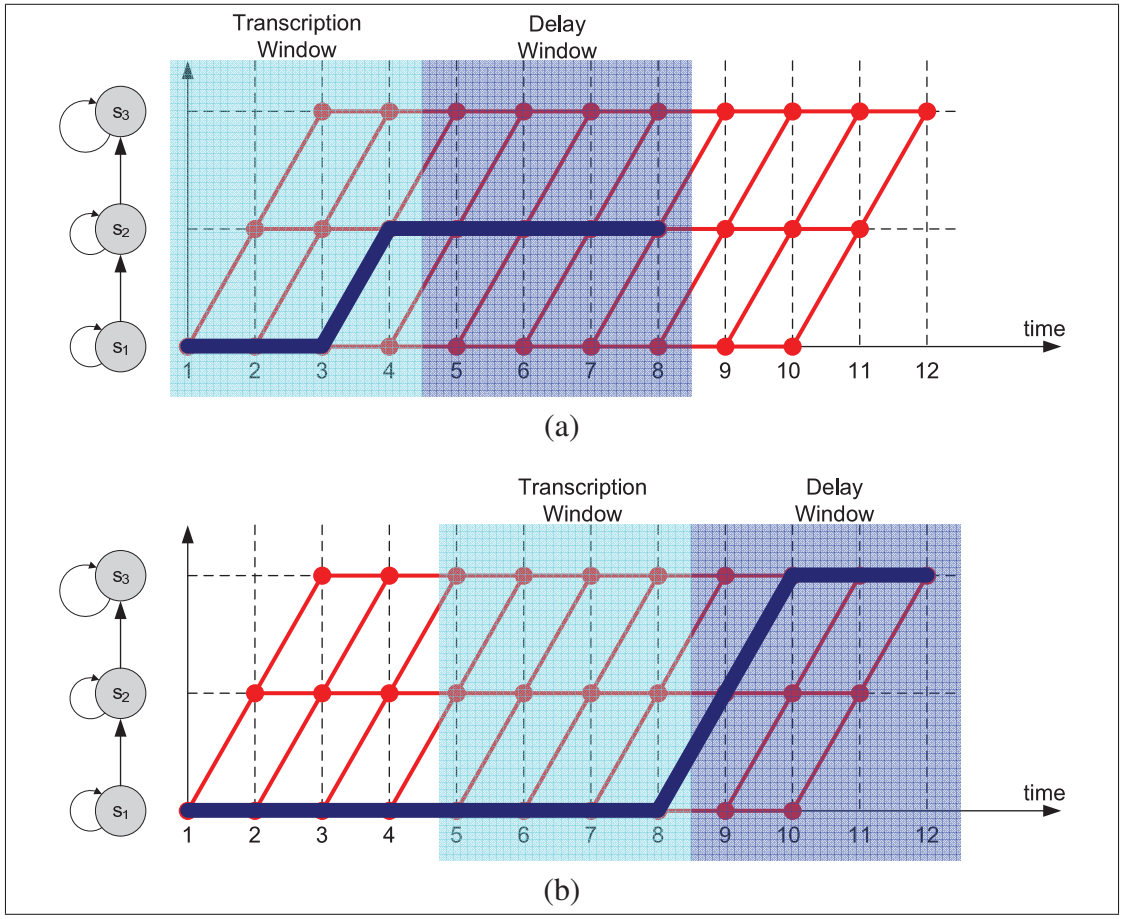

Figure 3.18 Example of a real-time transcription process.

This approach can however lead to a decrease of transcription accuracy. Indeed, the best path found from a given frame in the decoding can be completely different than the optimal one that will be found later in the search. But since the partial transcription based on the best path found at a previous frame has already been produced, the transcription cannot be modified. This problem is depicted by Figure 3.18(b) in which the state sequence for the first four frames is no longer the same after four new frames have been decoded. The effect of windowing on the accuracy is greatly reduced when the length of the window ($\delta + \lambda$) is long enough. Previous experiments on the closed-captioning system have shown that using 100 frames for both δ and λ reduced the accuracy by less than 0.5% absolute.

A major advantage of this approach is that it requires much less memory, since the memory required to maintain hypothesis information can be deleted when a partial transcription is produced. Thus, even when there is no need for real-time transcription production, this approach can still be used with large audio files that require large amounts of memory to be decoded.

3.4.1 A* Search Real-Time Transcription

Real-time transcription production is straightforward with the A* search described in section 3.3.3, since windowing is already implemented in the decoding process. A partial transcription can thus be readily produced after each search block.

However, it is not possible to produce real-time transcriptions when several heuristic blocks are computed simultaneously as described in section 3.3.5, except when a long delay is acceptable. Thus, for applications like real-time captioning, this approach cannot be used. Fortunately, most applications do not require real-time transcriptions but for those that do, the basic version of the search, which is nonetheless more proficient than the classical Viterbi decoder on parallel architectures, can be used.

3.5 Results

The experiments conducted in this section have been performed on the experimental setup discussed in Section 4.2. In this setup, acoustic models have been trained on 171 hours of

French television shows from Quebec. The language model has been trained with text from a local newspaper of approximately 93 million words. The test set is comprised of 44 minutes of similar audio to the training set.

3.5.1 Effect of the Lookahead on Accuracy and Computation Time

This experiment explores the effect of different lookahead values (between 10 and 200) on both the accuracy and computation time. Figure 3.19 shows that the accuracy initially increases with the duration of the lookahead and then falls rapidly. The A* search computation time shows a similar behavior, with a small drop at the beginning followed by an increase with the lookahead duration. The cause of this phenomenon is that the error, which is the difference between the heuristic approximation and the real cost, increases with the length of the lookahead. This reduces the discriminative power of the heuristic and, consequently, increases the complexity of the search.

The heuristic computation time increases linearly with the length of the lookahead. That is to be expected since the amount of work for each frame does not depend on the input audio and is thus constant for a given heuristic WFST.

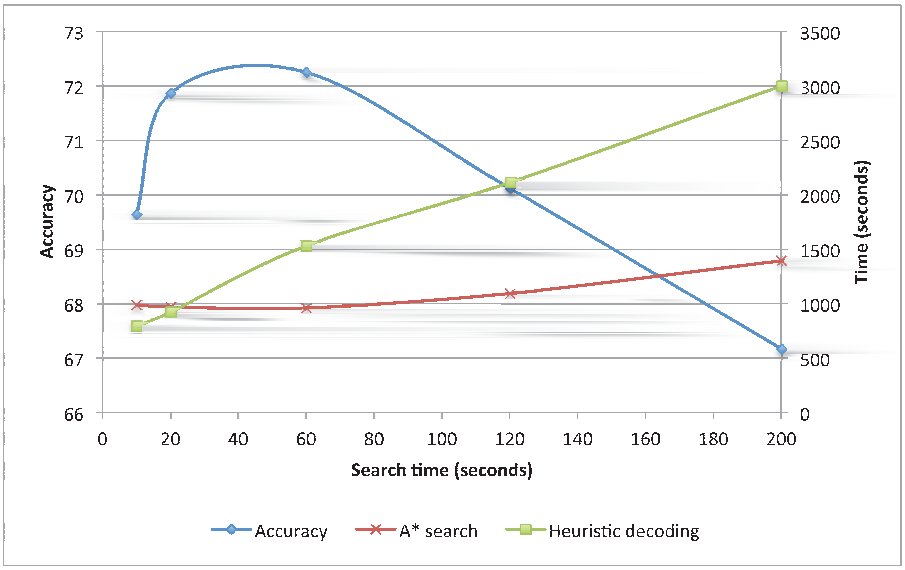

Figure 3.19 Effect of the lookahead on accuracy and computation time.

3.5.2 Parallelization of Heuristic Computation

As described earlier, the heuristic computation operates in 2 steps: computation of acoustic likelihoods and computation of heuristic costs. These steps account for more than 91% of the total search time. Table 3.2 shows how the computation time can be decreased by using multi-core and GPU architectures. Experiments have been conducted with 128 Gaussians acoustic models. CPU implementation of the heuristic costs operate on 16 heuristic windows compared to 32 for the GPU version.

Table 3.2 Parallel computation speed-up.

Step	architecture	Computation time (sec) CPU	Computation time (sec) Elapsed	speed-up factor
Heuristic costs	CPU 1-core	918	918	–
	CPU 4-cores	873	224	4.1x
	GPU	88	88	10.1x

On multi-core CPUs, our new approach leads to an efficient speed-up factor of 4.1 on a 4-core processor. This result is a significant improvement over our previous work in which we reported a speed-up factor of 3.1 (Cardinal *et al.* (2012b)). Another noteworthy point is that total CPU time is lower in the multi-core version, which results in a speed-up that is greater than the theoretical maximum of 4. That shows that our new approach efficiently uses the memory architecture of Intel processors.

Also note that epsilon transition expansions, which take up approximatively 8.5% of the Viterbi computation time, are not parallelized in the CPU version as they are in the case of the GPU.

The GPU version of the heuristic computation uses only half of the available cores (240). This is because the GTX 295 is in fact 2 GPUs of 240 cores in a single graphic card. Using both GPUs in this situation would result in extensive memory communication between them that would likely outweigh the benefits of using all available cores.

Finally, note that the timing includes the time needed for transferring data from computer to GPU global memory. The time required for transferring the acoustic models and heuristic FST is however not included. This activity occurs but once at the beginning of the process and its timing is negligible compared to the total execution time.

3.6 Summary

This chapter presented how searching through the recognition network can be efficiently implemented on parallel architectures such as multicore processors and GPUs. The A* algorithm has been used instead of the classical Viterbi algorithm for searching the graph. The algorithm uses a heuristic that provides an estimation of the cost for reaching a final state at the end of the utterance. This heuristic, another recognition network, is based on a unigram language model that can be efficiently decoded on parallel architectures.

The results show that decoding the heuristic on a Core i7 quad is 4.1 times faster than the single core version, which is better than the theorical speed-up. This is due to the fact that the memory architecture of the Intel processor is efficiently used.

On the GPU, the speed-up is by a factor of 10.1. The improvement over the CPU implementation is significant, but not as spectacular as in other applications such as the copy detection task presented in chapter 5. This is because the decoding algorithm is memory bounded, which imposes a restriction on GPU performance.

In the next chapter, global results are presented. They will confirm the assumption that the heuristic achieves a significant reduction in search time.

CHAPTER 4

RESULTS

In previous chapters, results have shown that the two major tasks of a speech recognition system can be efficiently parallelized. In chapter 2, the use of a 4-core processor was shown to provide a speed-up factor of 3.6 over its single core counterpart. The use of a GPU leads to a speed-up factor of 24.8.

Chapter 3 described the use of the A* search as an alternative over the classical Viterbi algorithm for searching the recognition network. This approach allowed to decrease the computational load of the search by introducing a heuristic that helps determine the hypotheses to explore in priority. Experiments have demonstrated that the computation of heuristic costs can be efficiently parallelized on both CPU and GPU architectures.

In this chapter, experiments with the entire speech recognition system are presented. The aims of these experiments are to:

- ensure that the accuracy of transcriptions produced by the A* decoder is on the same level as those produced by a classical Viterbi decoder;
- verify the hypothesis that the unigram heuristic allows to significantly decrease the number of explored nodes;
- ensure that the parallel version of the A* decoder is efficient with respect to both the transcription accuracy and audio decoding speed.

The chapter begins with a brief description of the speech recognition system used in the experiments. The experimental setup is then described. Finally, results of the experiments are presented and analyzed.

4.1 Putting It All Together

Figure 4.1 shows the diagram of the speech recognition system, which uses the GPU for computing acoustic likelihoods and heuristic scores.

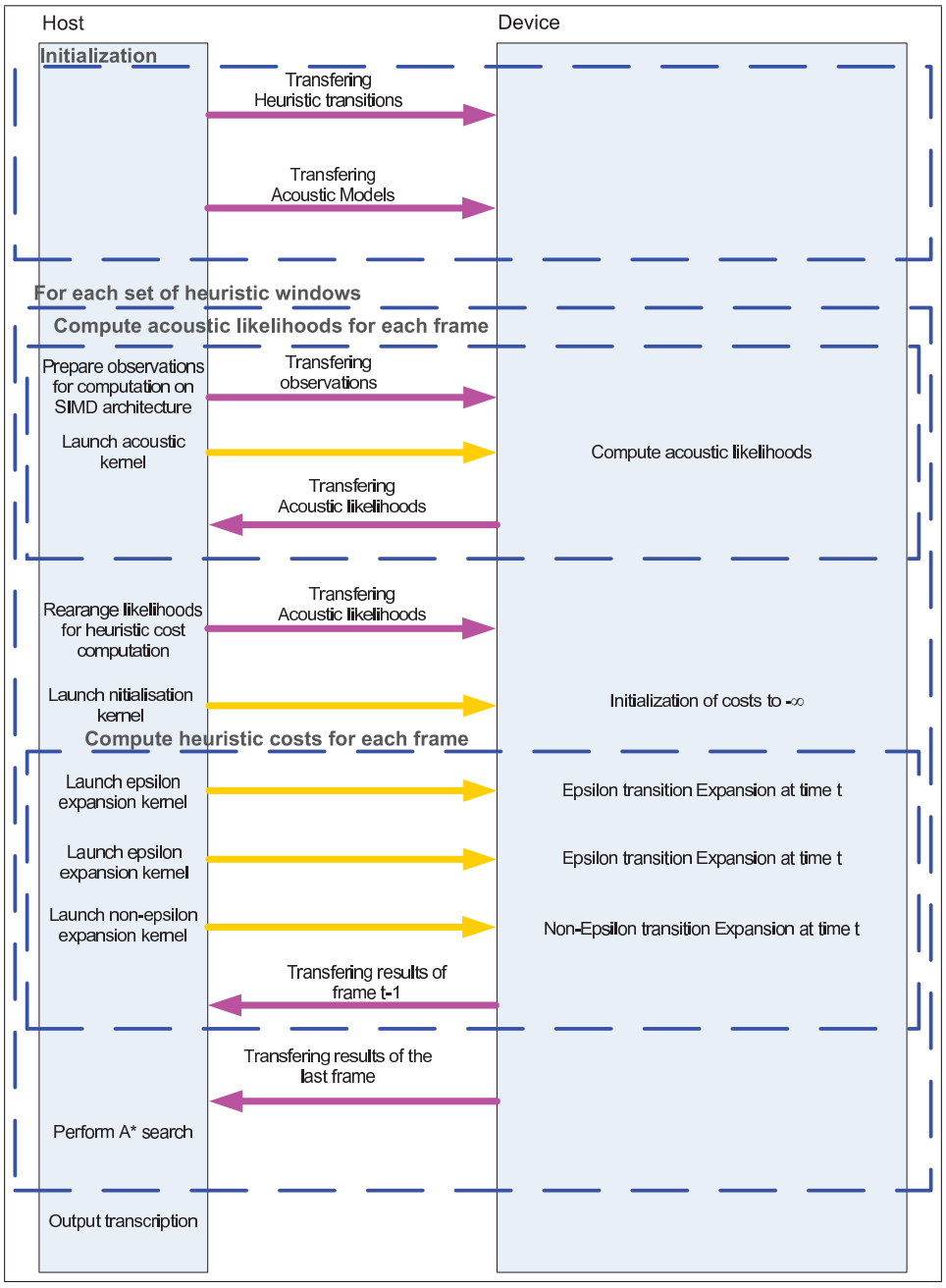

Figure 4.1 Diagram of the speech decoding process with a GPU.

In this figure, yellow arrows denote the launch of GPU kernels and fuchsia arrows denote data transfer between the host main memory and the GPU global memory.

During system initialization, the acoustic model and heuristic network are transferred into GPU global memory. These data are independent of the audio file to be decoded and can thus be transferred only once at the beginning of the process. They can also be reused for the decoding of several utterances.

Then, the acoustic likelihoods are computed for the heuristic window frames that are used to compute the heuristic costs. As discussed in chapter 2, the likelihoods of several frames are computed simultaneously. The likelihoods are then arranged to allow for their efficient access via the process that computes heuristic costs. The same disposition of likelihoods is used in both the CPU and GPU implementations.

The following step is the heuristic costs computation. This step requires a large amount of memory since the costs for each state in every frame of the set of heuristic windows have to be stored in memory. In the event that the GPU does not have sufficient memory to store all the frame data, it is straightforward to implement a procedure that computes costs of consecutive subsets.

The A* search is then applied to the frames for which heuristic costs have been computed. At this stage, the process could be optimized by concurrently computing the acoustic likelihoods and heuristic costs of the next set of heuristic window frames. This approach has the advantage of hiding the search time, thus reducing the computation time of the overall process. This procedure could also be applied to the CPU version but in that case, a core will need to be dedicated to the A* search. Consequently, the speed-up will be less appreciable in comparison to that afforded by the GPU.

Finally, the transcription can be produced once all the frames have been decoded. The system is then re-initialized in order to process the next utterance.

A similar diagram can be used to depict the multi-core implementation of the system with the distinction that all computations are performed on the CPU.

4.2 Experimental Setup

The baseline system of comparison is a WFST-based speech recognition system developed at CRIM and tuned for speaker-independent transcriptions of broadcast news.

The acoustic model has been trained with 171 hours coming from French television programs in Quebec. The programs are a mix of weather, news, talk shows, etc. that have been transcribed manually. The acoustic parameters consist of 12 MFCCs plus the energy component, corresponding first and second derivatives, for a total of 39 features. The model contains 4600 distributions of 128 Gaussians with diagonal covariance matrices.

The language model has been trained with text from a French local newspaper (La Presse, 93 million words) and the acoustic training set's textual transcripts (2.1 million words). Both the unigram and trigram language models use the same vocabulary of 59624 words.

The CPU used is an Intel Core i7 quad at 2.9 GHz with 8 GB of RAM. The operating system used for these experiments is Scientific Linux 6.3. Programs are compiled with g++ 4.4.6. Acoustic computations on the CPU use the SSE registers. In the Viterbi version, required acoustic likelihoods are computed on-demand. This optimization is not possible with the A* algorithm since all likelihoods are used for computing the heuristic.

The GPU used is the NVidia GeForce GTX295, which contains 2 GPUs of 240 cores and 896 MB of memory. Thus, a total of 480 cores are available. Version 3 of CUDA has been used.

For all experiments involving the A* algorithm, the heuristic window length Δ has been set to 80 frames. The A* search is performed on $\Lambda = 20$ frames with a lookahead of 20 frames. Thus, for each block of heuristic scores, 3 A* searches are performed.

The test set is made up of 44 minutes (2625 seconds) of audio files with a duration varying between 32 and 50 seconds.

4.3 Comparison with the Classical Viterbi Beam Search

The goal of the first experiment was to show that the A* search, used in conjunction with a unigram-based recognition network as heuristic, was able to reach the same performance in terms of word accuracy as the classical Viterbi search. The beams in both systems have been set to obtain approximately the same accuracy. The accuracy obtained in this experiment is approximately 99% of the maximum achievable limit with these models. Table 4.1 shows the results of this experiment.

Table 4.1 Viterbi vs A* performance.

Algorithm	Computation time (seconds)	# of explored nodes	Accuracy
Viterbi	10007	9 297 558 686	71.86 %
A* (1 Thread)	13328	319 417 949	71.93%
A* (4 Threads)	4627	319 417 949	71.93%

The results show that the A* search achieves the same accuracy as the Viterbi decoder by exploring approximately 29 times fewer nodes. The A* search itself accounts for only 7% of the total computation time, which confirms the discriminative power of the unigram-based heuristic.

Turning now our attention to the issue of execution speed, the results show that the sequential implementation of the A* search is approximately 33% slower than the Viterbi decoder. The main advantage of the Viterbi algorithm is its capacity of computing only the required acoustic likelihoods. Within this scenario, only 40% of all likelihoods are actually computed. Note that this scheme is not possible with the A* search since all likelihoods are needed to compute the heuristic costs.

In the case of the A* approach, the computation of acoustic likelihoods for 128 Gaussian component distributions accounts for 84% of the total computation time. In a previous experiment utilizing 32 Gaussian component distributions, the time dedicated to the acoustic likelihood computations was 64% of the total time. However, the use of 4-cores to compute acoustic likelihoods and the heuristic leads to a very interesting speed-up.

Another advantage of the Viterbi algorithm is its simplicity. As described earlier, its implementation uses two arrays, one for the current time t and the other one to store Viterbi costs at time $t + 1$. Consequently, states to explore are accessed and inserted in $O(1)$. However, the A* algorithm uses a priority queue, which allows to extract the most promising state in $O(logN)$, where N is the number of nodes waiting to be explored. The insertion of new nodes in the priority queue is also $O(logN)$. This is reflected in the timing results when compared to a Viterbi-based decoder. The real-time is defined as the duration of the test set (44 minutes).

4.4 Using a GPU and a Multi-Core Processor

The main experiment uses parallel architectures in the A* decoder. For this experiment, the heuristic was admissible. Figure 4.2 shows the results of this experiment. In this figure, the dashed line represents the real-time for this experimental setup.

The dissimilarity of the A* search curves with respect to that of the Viterbi search highlights the following point: the maximum achievable accuracy is reached much more quickly with the A* search. For example, the additional execution time required for increasing the accuracy from 69% to 72% is about 900 seconds using the A* search. With the Viterbi algorithm on the other hand, the same increase in accuracy comes at the cost of a delay of about 8500 seconds of processing time. This result underscores the efficiency of the unigram-based heuristic.

The use of a 4-core processor does not attain real-time performance. However, at the time where the maximum achievable accuracy of 72% is reached with the A* search on a 4-core processor, the Viterbi-based decoder scores an accuracy of 70%, a degradation of 2% absolute.

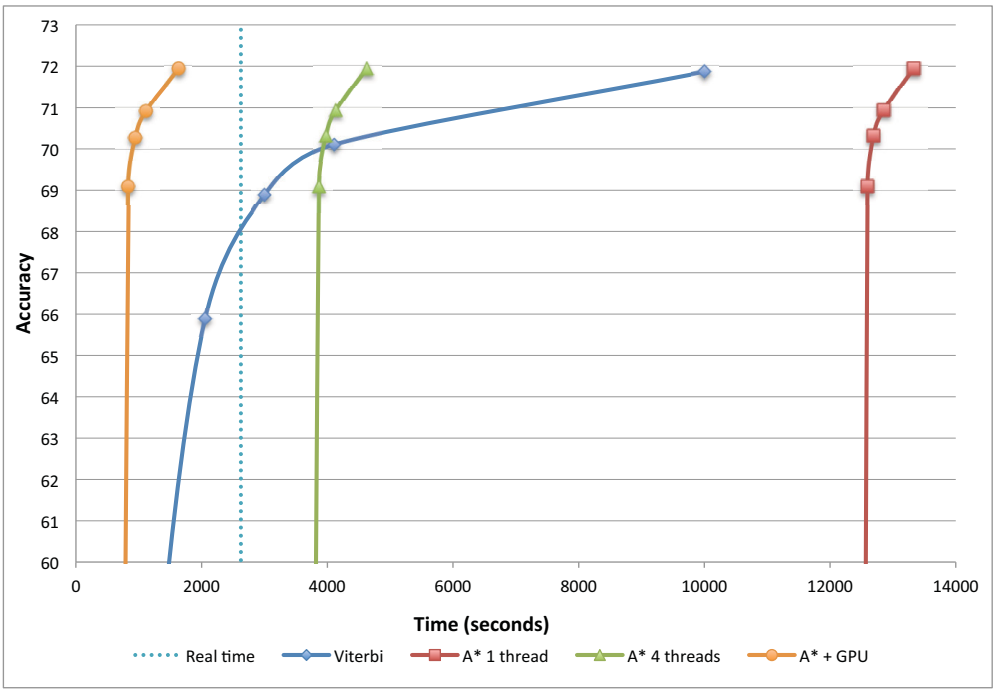

Figure 4.2 A* with GPU decoder accuracy vs execution time.

When a GPU is used for the computation of both the acoustic likelihoods and heuristic costs, the A* decoder now achieves 72% of word accuracy at 0.6 times the real-time. The processing time for reaching the maximum score with these models is 1633 seconds, which is 6.13 times faster than the classical Viterbi implementation. Moreover, at real-time, the accuracy is improved by roughly 4% absolute.

4.5 Using a Non-Admissible Heuristic

In many areas of applications, non-admissible heuristics may nonetheless be used for finding satisfactory solutions. This approach represents a trade-off between precision and speed: it has its merits when it is connected with an inappreciable sacrifice in accuracy that translates into a meaningful gain in processing speed. In this section we examine the feasibility of applying the A* algorithm with a non-admissible heuristic, according to the conditions laid out in Section 3.3.1. Specifically, the actual unigram probabilities are used in the construction of the heuristic network.

Table 4.2 Admissible vs non-admissible heuristic.

	Admissible heuristic		Non-Admissible Heuristic	
beam	# of explored nodes	Accuracy	# of explored nodes	Accuracy
60	66 933 972	69.1%	12 443 054	64.2%
80	160 183 317	70.9%	36 590 237	68.69%
100	319 642 164	71.9%	100 485 114	71.13%

Table 4.2 shows how the number of explored states is affected by the heuristic. As expected, the accuracy is lower with the non-admissible heuristic. However, the number of explored nodes is also significantly lower, up to 5.4 times, when the non-admissible heuristic is used. Figure 4.3 shows that it is possible to achieve the same accuracy when using the non-admissible heuristic by increasing the beam. At the maximum accuracy, the beam of the non-admissible heuristic system was at 120, compared to 100 for the admissible one.

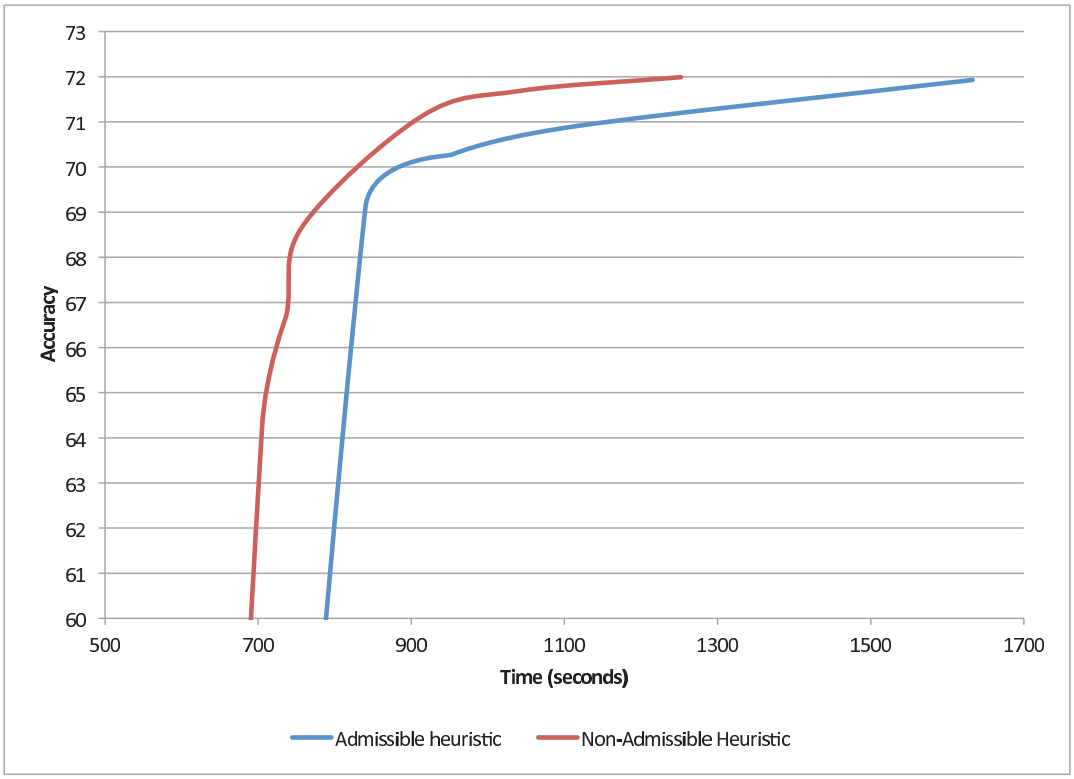

Figure 4.3 Using a non-admissible heuristic.

Figure 4.3 shows the system with the non-admissible heuristic to be approximately 25% faster when the beams are set to achieve the same accuracy. Note that, at the same decoding time, the use of the non-admissible heuristic leads to an improvement of the accuracy by approximately 14% absolute compared to the classic Viterbi decoder as showed in Figure 4.2. This analysis proves that the A* search maintains its overall proficiency notwithstanding the use of a non-admissible heuristic that, moreover, allows for faster processing times.

4.6 Summary

This chapter first discussed the integration of the two major components of a parallel speech recognition system capable of taking advantage of parallel processors.

Experiments have demonstrated the efficiency of the A* search algorithm that uses a unigram-based recognition network heuristic. Indeed, the number of nodes explored by the A* search is 29 times smaller than its Viterbi counterpart, with the same accuracy. This result demonstrates the quality of the heuristic and the hypothesis that the computation time dedicated to the search is sufficiently small for concluding that its parallelization would not lead to any significant improvement on the decoding speed.

Results have demonstrated that using a GPU leads to an accuracy improvement of 4% absolute over the classical Viterbi algorithm when both systems are configured to run in real-time. Moreover, the use of a non-admissible heuristic leads to a significant speed-up of about 25% over the admissible one. When compared to the classical Viterbi algorithm at the same processing time, the improvement is approximately 14% absolute.

CHAPTER 5

ANOTHER APPLICATION OF GPUS : COPY DETECTION

This chapter explores how multi-core processors and GPUs can be used for audio copy detection. There are many applications for which content-based copy detection proves to be useful. The most obvious application is the monitoring of peer-to-peer copying of music, movies or any other copyrighted audio recordings over the internet. The IFPI (International Federation of the Phonographic Industry) estimates that 3.6 billion downloads were purchased in 2011, an increase of 17% compared to 2010. This does not include copyright contents that have been illegally downloaded that represent a loss of billions of dollars in sales.

Another application is the monitoring of advertising campaigns over television and radio shows. Companies that advertise are interested in monitoring their advertisements to ensure they are broadcast as agreed with the broadcaster. They are also interested in monitoring their competitors' advertising for business intelligence. According to eMarketer's lastest report, worldwide advertisement spending sums to $470 billion in 2011.

The copy detection algorithm developed at CRIM by Vishwa Gupta (Héritier *et al.* (2009); Gupta *et al.* (2010a,b); Cardinal *et al.* (2010)) and implemented in a GPU as part of this thesis work performed very well in terms of detection accuracy and processing time at the TRECVID evaluation. This algorithm proved to be robust towards various audio recording transformations that could potentially mislead the copy detection process.

This chapter gives a brief overview of the CRIM's algorithms involved in its copy detection implementation. It also describes the nearest neighbor fingerprint computation that has been implemented in GPUs, making it fast enough to be used in real-world applications. Finally, descriptions and results for three applications are given that demonstrate the usefulness and efficiency of the process.

5.1 Detection Process

The task is to locate specific audio segments within a large amount of audio data. Algorithms presented in this chapter are based on fingerprint matching. A fingerprint is a condensed representation of large data. Just as the human fingerprint, the data fingerprint uniquely identifies a chunk of data. Two algorithms for computing the fingerprint of an audio sequence are presented and tested on various types of recordings and applications.

5.1.1 Fingerprint Matching

Copy detection is accomplished by computing a fingerprint for each frame of the reference audio. The fingerprint is also computed for each frame of the audio to be analyzed (queries). Basically, the search is done by moving the query audio (n frames) over the reference audio (m frames) and counting the number of fingerprint matches for every possible query and reference alignment, as illustrated in Figure 5.1. In this example, in which the query is aligned at frame 1, the match starts at frame 3 and ends at frame 7 with a score of 3, since there are 3 matching fingerprints.

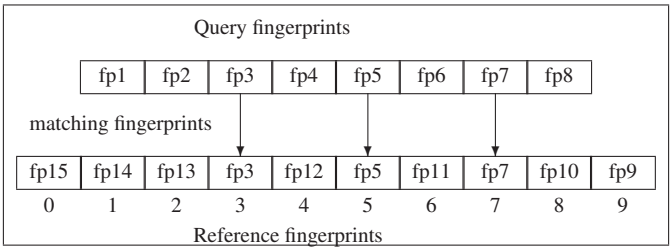

Figure 5.1 An example of matching a query audio to a reference.

From the $m - n$ alignments, only those with a count greater than a fixed threshold are considered. In our case, we used a threshold optimized for the copy detection task. The remaining alignments are then filtered according to the following rules:

Extension

Two alignments are considered synchronized if the positions of their starting frames differ by at most two frames. Figure 5.2 shows an example of synchronized alignments a_1 and a_2.

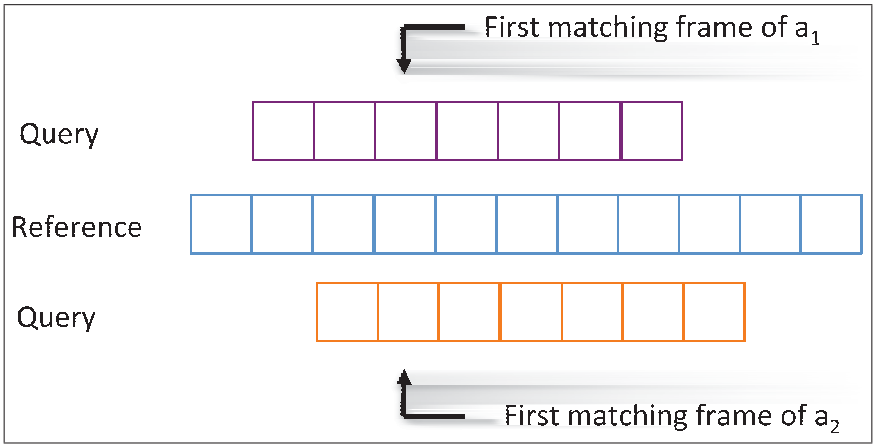

Figure 5.2 Example of synchronized alignments

More formally, two alignments a_1 and a_2 are synchronized if

$$|(refStart[a_1] - refStart[a_2]) - (queryStart[a_1] - queryStart[a_2])| \leq 2 \qquad (5.1)$$

where $refStart[a]$ and $queryStart[a]$ are respectively the first matching frame in the reference and the first matching frames in the query for the alignment a.

If two alignments are synchronized, the one with the lower count is eliminated and its count is added to the remaining one.

Overlap

Two alignments a_1 and a_2 overlap if one of the following conditions is met:

$$\begin{aligned} refStart[a_2] \leq refStart[a_1] \quad \text{and} \quad refEnd[a_2] \geq refStart[a_1] \\ refStart[a_2] \leq refEnd[a_1] \quad \text{and} \quad refEnd[a_2] \geq refEnd[a_1] \\ refStart[a_2] \geq refStart[a_1] \quad \text{and} \quad refEnd[a_2] \leq refEnd[a_1] \end{aligned}$$

where $refStart[a]$ and $refEnd[a]$ are respectively the first and last matching frame in the reference for the alignment a. When two alignments overlap, the one with the lower count is eliminated.

5.1.2 Copy Detector

The copy detector uses two types of fingerprints for accurately detecting queries in one or more references. Figure 5.3 shows a global view of the detection process.

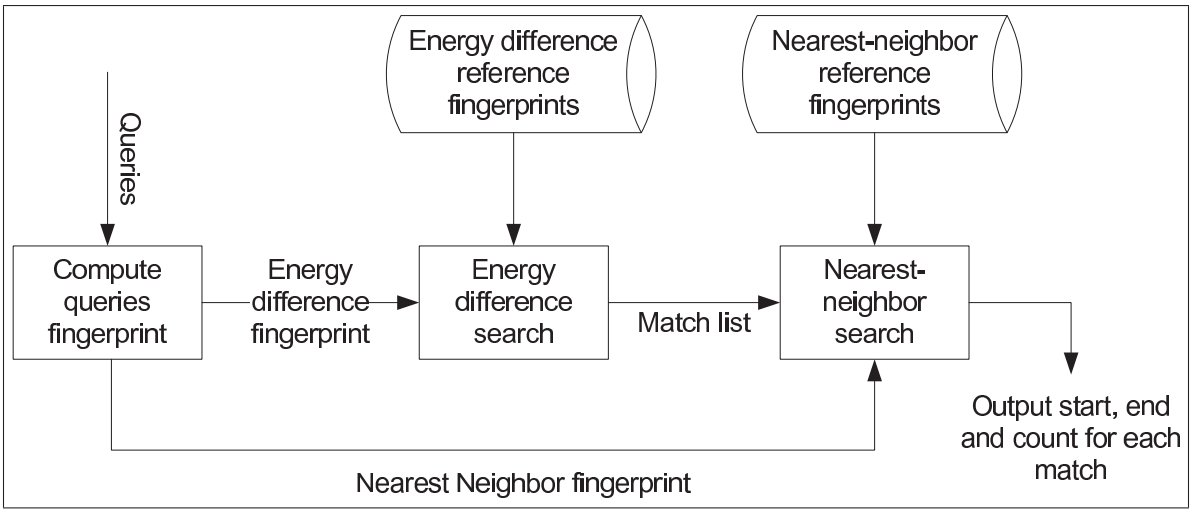

Figure 5.3 Copy Detection process

5.1.3 Energy-Difference Fingerprint

The first type of fingerprint is referred to as the energy-difference fingerprint. Basically, 15 bits/frame are extracted from the audio signal. In a first step, the audio signal is lowpass-filtered to 5.5 kHz; pre-emphasized with a coefficient of 0.97, and divided into 25 ms Hamming windows with 10 ms frame advance. The Fourier transform spectrum between 300 Hz and 5000 Hz is divided into 16 bands using mel-scale spaced triangular windows and the energy is computed in each band. The energy differences between the bands are used to compute the fingerprint. If $EB(n, m)$ represents the energy value of the n^{th} frame at the m^{th} band, then the m^{th} bit $F(n, m)$ of the 15-bit fingerprint is given by

$$F(n,m) = 1,\ if\ EB(n,m) - EB(n,m+1) > 0,$$
$$Otherwise,\ F(n,m) = 0.$$

In other words, the m^{th} bit is set to 1 if the energy over the m and $m + 1$ bank is growing. The search process is described in more detail in Héritier *et al.* (2009). This method is very fast and produces good results. Moreover, the search algorithm is very easy to parallelize on multicore/distributed systems since each query can be computed independently. Table 5.1 shows the processing time results of experiments conducted on an advertisement detection task.

Table 5.1 Processing times of energy difference fingerprint on a quad core CPU. The reference searches for 1379 advertisements over 51 hours of audio.

Number of threads	CPU time (min:sec)	Elapsed time (min:sec)
1	11:59	11:59
2	11:58	6:08
4	14:29	3:42

5.1.4 Nearest-Neighbor Fingerprint

The second type of fingerprint, the Nearest-Neighbor (NN) fingerprint, maps each frame of the reference to the closest frame of the query. For computing this measure of closeness, 12 cepstral coefficients and normalized energy are used. Their first and second derivatives can also be used, leading to a total of 26 and 39 features respectively. It is also possible to use any other number of features, but these are those that are typically used. The distance between a reference frame and a query frame is defined as $\sum_{i=1}^{n} |r_i - q_i|$ where $q_1, ..., q_n$ are the cepstral parameters for a query frame and $r_1, ..., r_n$ are the cepstral parameters for a reference frame. To each reference frame is associated its closest query frame. This process is depicted by Algorithm 6. Once each reference frame has been labeled with the closest query frame, matching proceeds as in Figure 5.1.

Algorithm 6: Nearest-Neighbor computation

Data: query frames, reference frames
Result: For each frame of the reference, the closest query frame

```
1  foreach f_ref ∈ reference do
2      min ← ∞
3      foreach f_query ∈ query do
4          d ← 0
5          for coeff ← 1 to n do
6              d ← d + | f_prg[coeff] - f_ad[coeff] |
7          end
8          if d < min then
9              results[f_ref] ← f_query
10             min ← d
11         end
12     end
13 end
```

Computing the closest query frame for each reference frame is computationally intensive. However, note that the search for the nearest query frame can be done independently for each reference frame. Consequently, an alternative processor, specialized in parallel computations, can be used to outperform the speed offered by modern CPUs.

Experiments have been performed with a database of 51 hours of reference audio and approximately 10 hours of query audio. This is the experimental setup for the query detector that will be described later in this chapter. Table 5.2 shows the performances of GPU over CPU for the NN fingerprint task.

Table 5.2 Processing times of nearest-neighbor fingerprint on GPU. The reference searches for 1379 advertisements over 51 hours of audio.

Platform	Execution time
CPU	464 hours
GPU	6.5 hours

The results show the GPU to be faster by a factor of 70 over its single-threaded CPU counterpart. Other experiments have revealed the GPU to be up to 200 times faster than a single-threaded Core i7 CPU when the number of frames in the query set is large compared to the number of frames in the reference.

5.1.5 Nearest-Neighbor Kernel

Figure 5.4 shows how the computation of the NN is calculated in the GPU. In this figure, t_{id} denotes the thread identifier for which the range is $[0..n[$ where n is the number of threads in the block. The value of $blockId$ denotes the block identifier in the grid. In this application, the number of blocks is the number of reference frames divided by 128. This value has been chosen to ensure that shared memory is used to its fullest potential and to ensure efficient data transfer from global to shared memory.

Firstly, the reference frames are divided into sets of 128 frames. Each set is associated with a multiprocessor running 128 threads. Thus, each thread computes the closest query frame for its associated reference frame.

Each thread in the multiprocessor downloads one query frame from global memory. Each thread can then compute the distance between its reference frame and all the 128 query frames

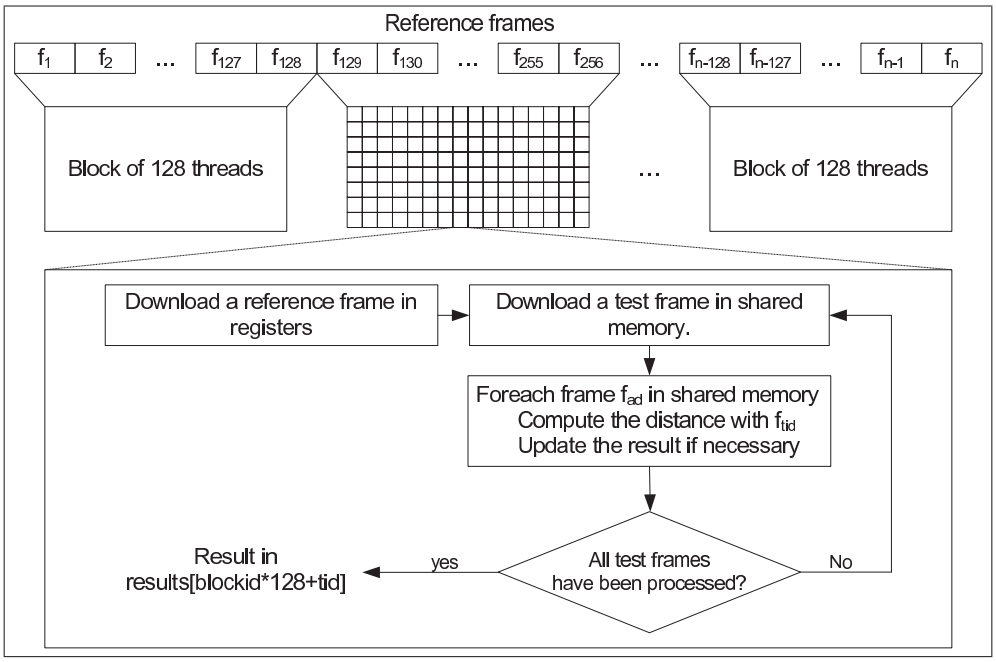

Figure 5.4 Nearest-Neighbor computation in the GPU

now in shared memory. This operation corresponds to lines $4 - 11$ of algorithm 6. When all threads have terminated, the next 128 query frames are downloaded and the process is repeated.

For increased performance, it is possible to process several references concurrently. The search algorithm is described in more detail in Héritier *et al.* (2009); Gupta *et al.* (2010a). In order to ensure storage of reference frames in GPU registers (which allows considerably much faster access than shared memory), a limit on the number of features must be imposed. With the GTX295, for which there are 16KB of registers, only the first 22 features can be used. This problem can however be circumvented by making several passes and combining the results in the CPU. The time required for this operation is negligible. On more recent GPUs that have 32KB of available registers, much more features can be used in a single pass.

5.1.6 Nearest-Neighbor Feature Search

Searching for audio segment matches in the query is trivial. We keep a count $c(i)$ for each frame i of the audio segment as a possible starting point for the query. Assume that for each audio segment frame i, $m(i)$ is the query frame that is closest to the audio segment frame i.

Then for each audio segment frame i, we increment the count $c(i - m(i))$. We also update the starting segment frame, and the last segment frame corresponding to frame $(i - m(i))$. The count $c(j)$ then corresponds to the number of matching frames between the audio segment and the query if the query started at frame j. Each frame j with a count $c(j)$ higher than a fixed threshold is in the list of found segments. More details about the search can be found in Héritier *et al.* (2009); Gupta *et al.* (2010a).

The NN fingerprint is more accurate than the energy-difference fingerprint. Comparing Tables 5.1 and 5.2, it can be seen that the NN fingerprint is much slower when a large set of data is considered. Another approach is to combine both fingerprints in a two-pass system.

5.1.7 Combining Both Fingerprints

As shown in Figure 5.3, the first step in the combined process is the detection of query samples using the energy-difference fingerprint search. For each query in the database, the search outputs a list of references that score as positive matches. The NN fingerprint is then applied to rescore each reference that was found. For efficiency, all references are processed concurrently by the GPU. The end result is the list of all query audio found within the entire set of references.

5.2 Applications of Copy Detection

This section describes three different applications of our copy detection algorithm.

5.2.1 Detection of Illegal Audio Copy

A very relevant application of our copy detection algorithm is the detection of illegally recorded copies of music or movies. The algorithm has been developed specifically for the NIST TRECVID evaluation task.

More specifically, the evaluated task involves searching for transformed audio queries of over 385 hours of test audio. The queries were transformed in seven different ways; three of these involved mixing unrelated speech to the original query, making it a much more difficult task

than advertisement detection. Table 5.3 details the types of audio transformations that were considered (from Gupta *et al.* (2010b)).

Evaluation results had already shown that the use of video streams for detecting specific segments is much less efficient both in terms of accuracy and processing time. Consequently, copy detection was applied to audio streams only.

Table 5.3 Query audio transformations used in TRECVID 2008/2009.

Transform	Description
T1	nothing
T2	mp3 compression
T3	mp3 compression and multiband companding
T4	bandwidth limit and single-band companding
T5	mix with speech
T6	mix with speech, then multiband compress
T7	bandpass filter, mix with speech, compress

The performance measure for this evaluation was the Normalized Detection Cost Rate (NDCR). This is a weighted linear combination of the missed detection probability and false alarm rate (measured per unit time). The missed detection probability is defined as:

$$P_{miss} = \frac{N_{Miss}}{N_{Targ}} \tag{5.2}$$

where N_{Miss} is the number of missed detections and N_{Targ} is the total number of events to be detected. Note that this ratio is dependent on a specified threshold value Θ. When the score returned by the system is higher than the threshold value, the query is detected as a copy of the reference. The false alarm rate, which is the number of times a query has been falsely marked as a copy, is defined as:

$$R_{FA} = \frac{N_{FA}}{Tref} \tag{5.3}$$

where N_{FA} is the number of false alarms and $Tref$ is the total duration, in hours, of reference audio segments. The combination of both metrics is defined as:

$$NDCR = P_{miss} + \frac{1}{200} R_{FA} \tag{5.4}$$

A NDCR value of 0 indicates perfect matching. Table 5.4 summarizes the results of experiments with 1407 (201 different audio recordings x 7 transformations) queries presented in Gupta *et al.* (2010b). In these experiments, one threshold has been used for all transformations. Setting a specific threshold for each transformation leads to better results but would not be representative of real-life applications for which transformations are not known in advance. In addition, the threshold has been chosen to discard false alarms (detecting a copy when it is not one).

Table 5.4 Minimal NDCR and computation time for the two fingerprints excluding false alarms.

Fingerprint	Transforms							Computation time
	1	2	3	4	5	6	7	
Energy Diff	.015	.037	.037	.022	.127	.135	.165	15 sec
NN	.007	0	.015	.015	.022	0	.03	360 sec
Energy Diff + NN rescoring	.007	0	.015	0.007	.037	0.03	.03	20 sec

The drawback of the energy difference algorithm is the number of false alarms it produces. The results confirm that the NN approach is more accurate than the energy difference approach but is much slower even when fingerprints are computed with a GPU. The most noteworthy point is that using the energy difference method as a pre-processing step for eliminating most of the segments has little impact on the combined approach results. Only T5 and T6 show correspondingly lower accuracies, but they are nevertheless much better than those provided by energy difference alone.

5.2.2 Advertisement Detection

Television advertising is widely used by companies to promote their products among the public. Worldwide, the TV and radio advertisement market was valued at over 214 billion dollars in 2008. In the US alone, TV and radio advertisements amounted to over 82 billion dollars in 2008. With all that money at stake, the advertiser is entitled to ascertain that its television advertising campaign is broadcast as requested and paid for.

Currently, monitoring of advertisement campaigns is offered as a service by many companies worldwide. Some companies use watermarking for automated monitoring of advertisements. In watermarking, they embed a unique code in the audio or the image before it is broadcast. This code can then be retrieved by their watermark monitoring equipment. Watermarking every commercial for subsequent monitoring by specialized equipment is however expensive. In addition, watermarking only allows companies to monitor their own advertisements and they cannot follow the campaigns of their competitors for business intelligence.

Another approach is the use of a content-based method that allows advertisement detection without the aforementioned constraints imposed by watermarking. Several works have been published dealing with content-based commercial detection. Most of these use repetition of sequences and/or video and audio features such as black frames or change in energy to detect advertisements in the broadcast stream Covell *et al.* (2006); Duygulu *et al.* (2004). These features do not however discriminate between specific commercials.

The copy detection algorithm has been tested on 51 hours of Canadian broadcast (French and English) divided in one hour segments. The advertisement database contains 1379 advertisements with an average length of 25.8 seconds. The results have been compared to a commercial product[1] which was the baseline. Table 5.5 shows the results.

[1]The product name cannot be divulged for confidentiality reasons.

Table 5.5 Performances of advertisement detection.

Fingerprint	Ads Detected	False Alarms	Subst.	Processing time
Baseline	329	11	3	180 s/h
Energy diff.	393	22	2	4.4 s/h
NN	401	20	0	458 s/h
Combined	393	7	2	9.5 s/h

The results show that our system outperforms the commercial one by finding at least 64 (18%) more advertisements with a comparable false alarm rate. Moreover, our system is very fast with a computation time of 0.3% of the real-time.

The energy-difference fingerprint is quite fast but produces more false alarms. Note that four of these false alarms were in fact the same advertisement with a different speech content but identical background music. The two errors were the same advertisement in a different language.

The NN fingerprint finds more advertisements but is the slowest. Some of the false alarms (8) were the same advertisement with different spoken texts (same musical background). Another one was the same speech content with different background music.

In the last experiment, the results produced by the energy difference fingerprint are rescored by the NN method in order to eliminate false alarms. This worked very well since 15 false alarms have been eliminated while four of the remaining seven are in fact an advertisement of the same product and thus are very similar (same background music, different speech content). The analyzing time is less than 10 seconds per hour of audio.

In another experiment, we have adjusted the thresholds (on counts and advertisement start) to eliminate false alarms and errors. Table 5.6 shows the results. The NN fingerprint provides a better discrimination of advertisements with a loss of only 19.5% compared to 50% with the energy difference fingerprint. Consequently, using the NN fingerprint can significantly improve the advertisement detector.

Fingerprint	Ads Detected	Difference
Energy diff.	196	-50.0%
NN	324	-19.2%
Combined	322	-18.0%

Table 5.6 No false alarms advertisement detection

When running experiments, it has been observed that the audio quality of advertisements in the database and audio stream is different. Indeed, the database advertisements were often of higher quality than the analyzed audio. Our results show that our fingerprints perform robustly towards differences in sound quality.

5.2.3 Film Edition

Movies are a sequence of reels that are edited by the filmmaker. When a new edition has to be produced - a blue-ray edition for example - edition data are used to reconstruct the movies from the reels. However, it happens that edition data are lost and the only available reference is a final cut of the movie. In this case, re-edition of the movie is a very time consuming task since the editor has to retrieve edition data from the reference movie.

The task is thus to automatically find which part of different reels have been used in the final version of the movie. There are typically three tracks to consider:

- audio and background sounds
- soundtrack
- special effects

The copy detection algorithm can thus be used to determine which cut has been used and its exact time in the reference. Since the algorithm is robust towards transformation, the detection precision remains high even if audio of lesser quality is used as reference.

The algorithm has been tested on a NFB (National Film Board) movie called Mario for which 11 cuts of 20 minutes were available. The reference was the VHS version of the movie. Fig-

ure 5.5 shows the results for each track. In some circumstances, only an already mixed version of the audio is available. This situation has also been experimented.

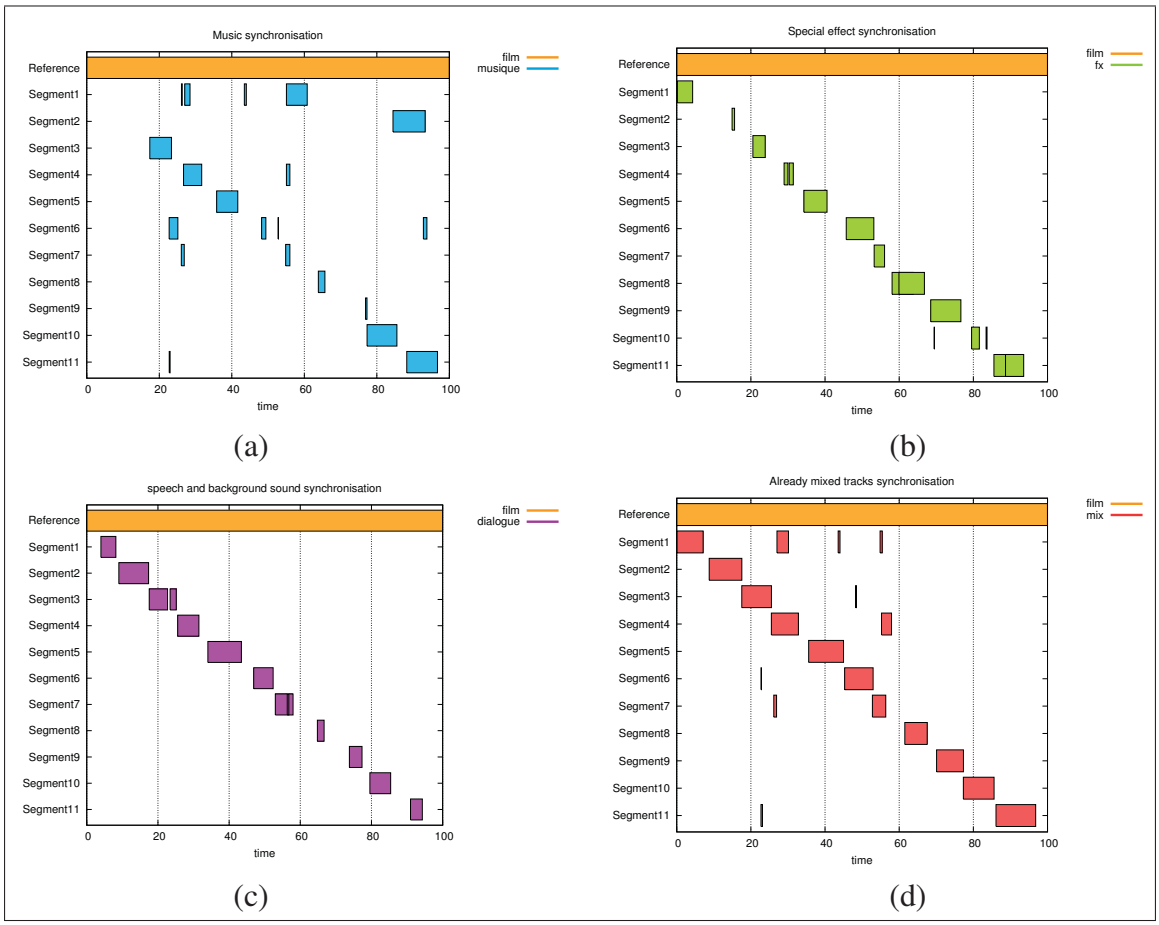

Figure 5.5 Results for using the copy detection algorithm for automatic movie edition. (a) Matching of the music recordings with the reference movie; (b) Matching of special effect recordings with the reference movie; (c) Matching of speech and background sound recordings with the reference movie; (d) Matching of mixed track recordings with the reference movie;

The segment alignment showed in Figure 5.5(a) seems to contain errors. However, the score of erroneous matching was very low compared to the good ones. Consequently, they could be easily removed by a correctly choosen threshold value. Another source of error is that a given song can appear several times in the movie. In this case, the algorithm selects the one with the

highest score. It is however possible to output several alignments in a semi-automatic way by allowing a human to make the final choice.

Note that there is no music during the first 15 minutes of the movie. This is why no matching has been found.

Overall, audio segments have been correctly positioned throughout the movie. In the case of overlapping segments, corrections have been applied by the operator via simple manual verification.

These results show that the algorithms can help save a lot of time by computing the alignments that can subsequently be corrected by a human. This is much faster than finding all alignments manually.

5.3 Summary

This chapter described the copy detection algorithm developed at CRIM. Two types of fingerprint methods are used to find copies of audio recordings. The energy difference search is fast but performs much less accurately than the MFCC-based nearest-neighbor approach. The major drawback of the NN approach is its processing time. Nevertheless, its GPU implementation has led to a convincing speed-up factor up to 200 times over its single threaded counterpart, making it a method of choice in real applications. Using it for rescoring the ouput produced by the energy difference-based copy detector showed a very small drop in accuracy compared to using the NN-based copy detection system alone. Experiments have shown that 93% of queries were correctly detected, even under the most arduous conditions.

Three real-life copy detection applications have been described:

- Detection of illegal copies of audio recordings;
- Advertisement detection;
- Automatic movie rebuilding from reels.

CONCLUSION

This chapter summarizes the work presented in this thesis. The main contributions are outlined in the first section. A list of relevant papers that I have authored or co-authored during the course of this research is also given. The second section discusses future work and describes how the work presented in this thesis can be used in the design of speech recognition specialized hardware.

Main Contributions

Chapter 2 has demonstrated how GPUs can be used to accelerate the computation of acoustic likelihoods, a major time-consuming task in many useful applications of automatic speech recognition. By dedicating this task to a GPU, a speed-up of 24.8 times over the sequential CPU implemention using SSE instructions has been obtained. This research topic has been published in the following papers:

Cardinal *et al.* (2008)	**Cardinal P.**, Dumouchel P., Boulianne G. and Comeau M., *GPU Accelerated Acoustics Likelihood Computations*, In Proceedings of 9th Annual Conference of the International Speech Communication Association (Interspeech), p.964-967, September 22-26, 2008
Cardinal *et al.* (2009)	**Cardinal P.**, Dumouchel P. and Boulianne G., *Using Parallel Architectures in Speech Recognition*, In Proceedings of 10th Annual Conference of the International Speech Communication Association (Interspeech), p.3039-3042, September 6-10, 2009

Searching the recognition network represents the other major time-consuming task of a speech recognition system. This search is driven by both the language model probabililites $P(W)$ that are encoded in the recognition network and the acoustic probabilities $P(O|W)$. The parallelization of the classical beam-pruned Viterbi search algorithm is a very difficult task. Indeed, the recognition network is very sparse and only a small fraction of states is explored during the search. These characteristics lead to a misuse of the memory architecture, as described in Chapter 3. The proposed solution aims at circumventing this problem through a substantial reduction of the search task. This was achieved through an appropriate implementation of the

A* algorithm that uses a heuristic to guide the search. The better the heuristic, the faster is the search.

The heuristic is a recognition network based on the same models as the original, except that a unigram language model is used instead of a trigram. As a result, it is a much smaller network that can be subjected to an exhaustive search by the classical Viterbi algorithm. This approach offers several benefits:

- Since the heuristic is represented by a general-purpose framework, namely the WFSTs in this case, it can be readily integrated to the speech recognition system without having to modify the source code of the decoder. It then becomes a straightforward matter, in the future, to replace the presently used unigram-based heuristic with its bigram-based representation when enough cores become available to decode it efficiently.
- The decoding process can be designed to be efficiently implemented on parallel architectures. This can be easily accomplished with a simplified version of the Viterbi algorithm, especially when the graph does not have to be pruned.
- Heuristic costs can be computed concurrently with the A* search.

Results show that the use of a unigram-based heuristic allows the A* search to explore 29 times fewer nodes than the classical Viterbi algorithm. This is precisely why this approach has been chosen: it allows the computational load to be shifted from the search to the computation of the heuristic, which in turn can be efficiently implemented on parallel architectures. This represents the main contribution of this thesis, *i.e.*, the feasibility of implementing a speech recognition system that uses the full computational power offered by parallel architectures.

Moreover, the GPU version of the A* search allowed an accuracy improvement of 4% absolute over the sequential implementation of the classical Viterbi algorithm when both systems are configured to run at real-time. When compared at the same speed, the accuracy improvement is approximately 10% absolute. Experiments show that using a non-admissible heuristic reduces the computation time by 25%. When compared to the classical Viterbi implementation at the same speed, the accuracy is 14% absolute higher.

The following papers have been written detailing this accomplishment:

Cardinal *et al.* (2009)	**Cardinal P.**, Dumouchel P. and Boulianne G., *Using Parallel Architectures in Speech Recognition*, In Proceedings of the 10th Annual Conference of the International Speech Communication Association (Interspeech), p.3039-3042, September 6-10, 2009
Cardinal *et al.* (2012b)	**Cardinal P.**, Dumouchel P. and Boulianne G., *Using A* for the Parallelization of Speech Recognition Systems*, In proceedings of The IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), p. 4433-4436 March 25-30, 2012
Cardinal *et al.* (2012a)	**Cardinal P.**, Dumouchel P. and Boulianne G., *The A* Speech Recognition System on Parallel Architectures*, International Conference on Information Science, Signal Processing and their Applications (ISSPA), p. 108-113 July 2-5, 2012

Chapter 5 introduced the copy detection task, another speech recognition-related application where the use of a GPU gave very impressive results. The copy detection algorithm developed at CRIM uses MFCCs to create a fingerprint of an audio recording. It produces a very high accuracy of 97% for detecting audio copies with respect to common transformations such as downsampling or added noise. This approach also proved to be very efficient towards much more complex transformations such as voice recordings over the original audio. Under these circumstances, an accuracy of 93% was achieved. However, its single-threaded implementation was markedly too slow to be used in practice. The parallel implementation of this task on GPUs has produced a speed-up factor up to 200 times over the single-threaded version, allowing it to be used in real-world applications. The excellent results produced by the CRIM's parallel implementation of this algorithm, both in terms of accuracy and processing speed, have been noticed during the NIST TRECVID evaluation.

The following papers have been dedicated to this topic.

Cardinal *et al.* (2010)	**Cardinal P.**, Gupta V. and Boulianne G., *Content-Based Advertisement Detection*, In Proceedings of the 11th Annual Conference of the International Communication Association (Interspeech), p.2214-2217, September 6-10, 2010
Gupta *et al.* (2010a)	Gupta V., Boulianne G. and **Cardinal P.** *Content-Based Audio Copy Detection Using Nearest-Neighbor Mapping*, In proceedings of The IEEE Internation Conference on Acoustics, Speech and Signal Processing (ICASSP), p. 261-264 March 14-19, 2010
Gupta *et al.* (2010b)	Gupta V., Boulianne G. and **Cardinal P.**, *CRIM'S Content-Based Audio Copy Detection System for TRECVID 2009*, 8th International Workshop on Content-Based Multimedia Indexing (CBMI), p. 1-6, June 23-25, 2010

Future Work

In this thesis, the heuristic is a recognition network comprised of a unigram language model. A bigram-based heuristic could also be used for this purpose, but it would be much too large to be exhaustively searched with the currently available processors. To overcome this difficulty, the bigram model can be pruned by removing bigram probabilities that do not degrade the language model perplexity of a test set more than a given threshold. This should offer the possibility of using a bigram-based heuristic network that can be reduced as much as needed. In the same way, a trigram model could also be used. Other types of networks could also be used as heuristics such as bigram or trigram of part-of-speech for example. With the arrival of more powerful processors, a less agressive pruning could be used, paving the way to an even better heuristic.

Currently, a great deal of research is concerned with low-energy implementations of speech recognition in hardware. The approach described in this work could lead to an efficient hardware implementation along these lines. Indeed, one possible way of reducing a chip's energy consumption is to limit external memory accesses. This is precisely the problem associated with common hardware implementations of the Viterbi algorithm, which the parallel imple-

mentation presented in this work solves. In addition, specialized hardware could use independent memory banks allowing for an efficient parallelization of the A* search. The search implementation could use a different heap for each frame of the search window. If an independent memory bank is available for each frame, states in each heap could be explored concurrently without interfering with memory transfer of other calculation units. The result would be a completely parallel speech recognition engine.

BIBLIOGRAPHY

Aho, A., J.E. Hopcroft, and J.D. Ullman, 1974. *The design and analysis of computer algorithms*. Addison Wesley.

Aho, A., R. Sethi, and J.D. Ullman, 1986. *Compilers Principle, Techniques and Tools*. Addison Wesley.

Atal, B. S. and S. L. Hanauer. 1971. "Speech Analysis and Synthesis by Linear Prediction of the Speech Wave". *Journal of Acoustical Society of America*, vol. 50, n. 2, p. 637-655.

Baker, J. 1975. "The DRAGON System–An Overview". *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 23, n. 1, p. 24-29.

Becchetti, C. and L.P. Ricotti, 1999. *Speech Recognition, Theory and C++ Implementation*. Wiley.

Bilmes, J. 1997. *A Gentle Tutorial on the EM Algorithm and its Application to Parameter Estimation for Gaussian Mixture and Hidden Markov Models*. Technical Report ICSI-TR-97-021. University of Berkeley.

Cardinal, P., P. Dumouchel, G. Boulianne, and M. Comeau. September 22-26 2008. "GPU Accelerated Acoustics Likelihood Computations". In *Proceedings of 9th Annual Conference of the International Speech Communication Association (Interspeech)*. p. 964-967.

Cardinal, P., P. Dumouchel, and G. Boulianne. September 6-10 2009. "Parallel Architectures in Speech Recognition". *Proceedings of 10th Annual Conference of the International Speech Communication Association (Interspeech)*, p. 3039-3042.

Cardinal, P., V. Gupta, and G. Boulianne. September 26-30 2010. "Content-Based Advertisement Detection". In *Proceedings of the 11th Annual Conference of the International Speech Communication Association (Interspeech)*. p. 2214-2217.

Cardinal, P., G. Boulianne, and P. Dumouchel. July 2-5 2012a. "The A* Speech Recognition Systems on Parallel Architectures". In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. p. 108-113.

Cardinal, P., P. Dumouchel, and G. Boulianne. March 25-30 2012b. "Using A* for the Parallelization of Speech Recognition Systems". In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. p. 4433-4436.

Chen, S. and J. Goodman. 1999. "Empirical Study of Smoothing Techniques for Language Modeling". *Computer Speech and Language*, vol. 13, n. 4, p. 359-393.

Chong, J., E. Gonina, Y. Yi, and K. Keutzer. September 6-10 2009. "A Fully Parallel WFST-based Large Vocabulary Continuous Speech Recognition on a Graphics Processing Unit". *Proceedings of 10th Annual Conference of the International Speech Communication Association (Interspeech)*, p. 1183-1186.

Chong, J., E. Gonina, K. You, and K. Keutzer. September 26-30 2010. "Exploring Recognition Network Representations for Efficient Speech Inference on Highly Parallel Platforms". *Proceedings of 11th Annual Conference of the International Speech Communication Association (Interspeech)*, p. 1489-1492.

Cormen, T., C.E. Leiserson, R.L. Rivest, and C. Stein, 2001. *Introduction to Algorithms, 2nd edition*. MIT Press, Cambridge, MA.

Covell, M., S. Baluja, and M. Fink. 2006. "Advertisement Detection and Replacement Using Acoustic and Visual Repetition". In *Proceedings of the 2006 IEEE 8th Workshop on Multimedia Signal Processing*. p. 461 - 466.

CUDA. 2012. "http://www.nvidia.com/object/cuda_home.html".

Davis, K., R. Biddulph, and S. Balashek. 1952. "Automatic Recognition of Spoken Digits". *Journal of Acoustical Society of America*, vol. 24, n. 6, p. 637-642.

Davis, S. and P. Mermelstein. 1980. "Comparison of Parametric Representations for Monosyllabic Word Recognition in Continuously Spoken Sentences". *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 28, n. 4, p. 357-366.

Dixon, P., D.A. Caseiron, T. Oonishi, and S. Furui. December 13-17 2007. "The Titech Large Vocabulary WFST Speech Recognition System". In *Proccedings of IEEE Workshop on Automatic Speech Recognition Understanding (ASRU)*. p. 443-448.

Dixon, P., T. Oonishi, and S. Furui. 2009a. "Harnessing Graphics Processors for the Fast Computation of Acoustic Likelihoods in Speech Recognition". *Computer Speech & Language*, vol. 23, n. 4, p. 510-526.

Dixon, P., T. Oonishi, and S. Furui. April 19-24 2009b. "Fast Acoustic Computations Using Graphics Processors". In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. p. 4321-4324.

Duygulu, P., M. Chen, and E. Hauptmann. 2004. "Comparison and Combination of Two Novel Commercial Detection Methods". In *Proceedings of the International Conference on Multimedia and Expo (ICME2004)*. p. 1267–1270.

Forgie, J. and C.D. Forgie. 1959. "Results Obtained from a Vowel Recognition Computer Program". *Journal of Acoustical Society of America*, vol. 31, n. 11, p. 1480-1489.

Fry, D. and P. Denes. 1959. "The Design and Operation of the Mechanical Speech Recognition at University College London". *Journal of the British Institution of Radio Engineers*, vol. 19, n. 4, p. 211-229.

Gauvain, J., L.H. Hamel, G. Adda, and M. Adda-Decker. April 19-22 1994. "The LIMSI Continuous Speech Dictation System: Evaluation on the ARPA Wall Street Journal Task". In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. p. 557-560.

Gupta, K. and J.D. Owens. December 13-17 2009. " Three-Layer Optimizations for Fast GMM Computations on GPU-Like Parallel Processors ". In *Proceedings of the IEEE Workshop on Automatic Speech Recognition Understanding ASRU 2009*. p. 146-151.

Gupta, V., G. Boulianne, and P. Cardinal. March 14-19 2010a. " Content-Based Audio Copy Detection Using Nearest-Neighbor Mapping ". In *Proceedings of International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. p. 261-264.

Gupta, V., G. Boulianne, and P. Cardinal. June 23-25 2010b. " CRIM'S Content-Based Audio Copy Detection System for TRECVID 2009 ". In *the 8th International Workshop on Content-Based Multimedia Indexing (CBMI)*. p. 1-6.

Harish, P. and P.J. Narayanan. December 19-22 2007. " Accelerating Large Graph Algorithms on the GPU using CUDA ". In *Proceedings of the 14th International Conference on High Performance Computing (HiPC)*. p. 197–208. Springer-Verlag.

Harris, M., 2005. *Mapping Computational Concepts to GPUs*, chapter 31, p. 493-508. Addison Wesley, éd. 1.

He., G., T. Sugahara, Y. Miyamoto, T. Fujinaga, H. Noguchi, S. Izumi, H. Kawaguchi, and M. Yoshimoto. 2012. " A 40 nm 144 mW VLSI Processor for Real-Time 60-kWord Continuous Speech Recognition ". *IEEE Transactions on Circuits and Systems*, vol. 59-I, n. 8, p. 1656-1666.

Héritier, M., V. Gupta, L. Gagnon, G. Boulianne, S. Foucher, and P. Cardinal. November 16-17 2009. " CRIM's Content-Based Copy Detection System for TRECVID ". In *Proceedings of NIST TREC Video Retrieval Evaluation Workshop (TRECVID)*.

Hopcroft, J., R. Motwani, and J.D. Ullman, 2000. *Introduction to Automata Theory, Languages & Computability Second Edition*. Addison-Wesley.

Huang, X., A. Acero, and H. Hon, 2001. *Spoken Language Processing: A Guide to Theory, Algorithm and System Development*. Prentice Hall.

Ishikawa, S., K. Yamabana, R. Isotani, and A. Okumura. May 14-19 2006. " Parallel LVCSR Algorithm for Cellphone-Oriented Multicore Processors ". In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. p. 177-180.

Jelinek, F., L. Bahl, and R. Mercer. 1975. " Design of a Linguistic Statistical Decoder for the Recognition of Continuous Speech ". *IEEE Transactions on Information Theory*, vol. 21, n. 3, p. 250-256.

Johnston, J. and R.A. Rutenbar. July 9-11 2012. " A High-Rate, Low-Power, ASIC Speech Decoder Using Finite State Transducers ". In *Proceedings of 2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. p. 77-85.

Juang, B. and L.R. Rabiner. 2004. *Automatic Speech Recognition - A brief History of the Technology Development*. Technical Report http:www.ece.ucsb.eduFacultyRabinerece259Reprints354_LALI-ASRHistory-final-10-8.pdf. University of California at Santa Barbara.

Jurafsky, D. and J. H. Martin, 2000. *Speech and Language Processing*. Prentice-Hall.

Kai-Fu, L., 1989. *Automatic Speech Recognition: The Development of the* SPHINX *System*. Kluwer Academic Publisher.

Kanthak, S., H. Ney, M. Riley, and M. Mohri. 2002. "A Comparison of Two LVR Search Optimization Techniques". In *Proceedings of the International Conference on Spoken Language Processing 2002 (ICSLP '02)*.

Katz, S. M. 1987. "Estimation of Probabilities from Sparse Data for the Language Model Component of a Speech Recognizer". *IEEE Transactions on Acoustics, Speech and Signal Processing*, p. 400-401.

Kenny, P., R. Hollan, G. Boulianne, H. Garudadri, M. Lennig, and D. O'Shaugnessy. February 19-22 1992. "An A* Algorithm for Very Large Locabulary Continuous Speech Recognition". In *Proceedings of the Workshop on Speech and Natural Language*. p. 333-338.

K.Gupta and J.D. Owens. December 18-21 2011. "Compute & Memory Optimizations for High-Quality Speech Recognition on Low-End GPU Processors". In *Proceedings of 18th International Conference on High Performance Computing (HiPC)*. p. 1-10.

Kim, J. and W. Sung. March 25-30 2012. "Multi-User Real-Time Speech Recognition with a GPU". In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. p. 1617-1620.

Kim, J., K. You, and W. Sung. May 22-27 2011. "H- and C-Level WFST-Based Large Vocabulary Continuous Speech Recognition on Graphics Processing Units". In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. p. 1733-1736.

Kim, J., J. Chong, and I.R. Lane. September 9-13 2012. "Efficient On-The-Fly Hypothesis Rescoring in a Hybrid GPU/CPU-based Large Vocabulary Continuous Speech Recognition Engine". In *Proceedings of 13th Annual Conference of the International Speech Communication Association (Interspeech)*.

Knill, K., M. Gales, and S. Young. October 3-6 1996. "Use if Gaussian Selection in Large Vocabulary Continuous Speech Recognition Using HMMs". In *Proceedings of the 4th International Conference on Spoken Language (ICSLP)*. p. 470-473.

Kveton, P. and M. Novak. September 26-30 2010. "Accelerating Hierachical Acoustic Likelihood Computation on Graphics Processors". In *Proceedings of 11th Annual Conference of the International Speech Communication Association (Interspeech)*. p. 350-353.

Lee, K., H.W. Hon, and M.Y. Hwang. 1989. "Recent Progress in the SPHINX Speech Recognition System". In *Proceedings of the workshop on Speech and Natural Language (HLT'89)*. p. 125-130.

Lin, E. and R.A. Rutenbar. 2009. "A Multi-FPGA 10x Real-Time High-Speed Search Engine for a 5000-Word Vocabulary Speech Recognizer". In *Proceedings of the 2009 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*. p. 83-92.

Lin, E., K. Yu, R. A. Rutenbar, and T. Chen. September 17-21 2006. "Moving Speech recognition from Software to Silicon: the In Silico Vox Project". In *Proceedings of 9th International Conference on Spoken Language Processing (Interspeech)*. p. 2346-2349.

Lin, E., K. Yu, R. A. Rutenbar, and T. Chen. 2007. "A 1000-Word Vocabulary, Speaker Independent, Continuous Live-Mode Speech Recognizer Implemented in a Single FPGA". In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*. p. 60-68.

Lumsdaine, A., D. Gregor, B. Hendrickson, and J. W. Berry. 2007. "Challenges in Parallel Graph Processing". *Parallel Processing Letters*, p. 5-20.

Mah, G. and A. Castle. 2010. "The History of a Dream: How the Ultimate PC Has Evolved in 15 Years". *http://www.maximumpc.com*, p. 1-3.

Mermelstein, P. 1976. Distance Measures for Speech Recognition: Psychological and Instrumental. Chen, C. H., editor, *Pattern Recognition and Artificial Intelligence*, p. 374-388. Academic Press, New York.

Mohri, M. June 1997. "Finite-State Transducers in Language and speech processing". *Computational Linguistics*, vol. 23, n. 2, p. 269-311.

Mohri, M., F. C. N. Pereira, and M. Riley. August 11-16 1996. "Weighted Automata in Text and Speech Processing". In *Proceedings of the 12th biennial European Conference on Artificial Intelligence (ECAI-96), Workshop on Extended finite state models of language*.

Mohri, M., F.C.N. Pereira, and M. Riley. 2000. "Weighted Finite-State Transducers in Speech Recognition". In *Proceedings of the ISCA Tutorial and Research Workshop, Automatic Speech Recognition: Challenges for the new Millenium (ASR2000)*.

Mohri, M., F.C.N. Pereira, and M. Riley. 2002. "Weighted Finite-State Transducers in Speech Recognition". *Computer and Speech Language*, vol. 16, n. 1, p. 69-88.

Nedevschi, S., R.K. Patra, and E.A. Brewer. 2005. "Hardware Speech Recognition for User Interfaces in Low Cost, Low Power Devices". In *Proceedings of the 42nd Annual Design Automation Conference*. p. 684-689.

Nilson, M. 2005. *First order hidden markov model theory and implementation issues*. Technical Report 2005:02. Blekinge Institute of Technology.

NVidia. 2007. *NVidia CUDA Compute Unified Device Architecture: Programming Guide*.

Olsen, H. and H. Belar. 1956. "Phonetic Typewriter". *Journal of Acoustical Society of America*, vol. 28, n. 6, p. 1072-1081.

O'Shaughnessy, D., 2000. *Speech Communications*. IEEE Press.

Parihar, N., R. Schluter, D. Rybach, and E. A. Hansen. September 26-30 2010. "Parallel Lexical-tree Based LVCSR on Multi-core Processors". *Proceedings of 10th Annual Conference of the International Speech Communication Association (Interspeech)*, p. 1485-1488.

Paul, D. May 14-17 1991. " Algorithms for an Optimal A* Search and Linearizing the Search in the Stack Decoder". In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. p. 693-696.

Phillips, S. and A. Roggers. August 1999. "Parallel Speech Recognition". *International Journal of Parallel Programming*, vol. 27, n. 4, p. 257-288.

Rabiner, L. 1989. " A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition". *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, vol. 77, p. 257-286.

Rabiner, L., A.E. Rosenberg, and J.G. Wilpon. 1979. "Speaker Independent Recognition of Isolated Words Using Clustering Techniques". In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. p. 336-349.

Russel, S. and P. Norvig, 1994. *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Sakai, J. and S. Doshita. 1962. "The phonetic Typewriter". In *In Proceedings of the IFIP Congress'62*. p. 445-450.

Shi, M., A. Bermak, S. Chandrasekaran, and A. Amira. December 10-13 2006. " An Efficient FPGA Implementation of Gaussian Mixture Models-Based Classifier Using Distributed Arithmetic". In *Proceedings of 13th IEEE International Conference on the Electronics, Circuits and Systems (ICECS).* p. 1276-1279.

Sipser, M., 1997. *Introduction to the theory of computation*. PWS Publishing Company.

Vaněk, J., J. Trmal, J.V. Psutka, and J. Psutka. August 28-31 2011. "Optimization of the Gaussian Mixture Model Evaluation on GPU". In *Proceedings of 12th Annual Conference of the International Speech Communication Association (Interspeech)*. p. 1737-1740.

Vaněk, J., J. Trmal, J. V. Psutka, and J. Psutka. December 12-15 2012. "Full Covariance Gaussian Mixture Models Evaluation on GPU". In *IEEE International Symposium on Signal Processing and Information Technology*.

Wikipedia. 2013a. "Liaison". http://en.wikipedia.org/wiki/Liaison_(French).

Wikipedia. 2013b. “ Locality of reference ”. http://en.wikipedia.org/wiki/Locality_of_reference.

You, K., J. Chong, Y. Yi, E. Gonina, C.J. Hughes, Y.K. Chen, W. Sung, and K. Keutzer. 2009. “ Parallel Scalability in Speech Recognition ”. *Signal Processing Magazine, IEEE*, vol. 26, n. 6, p. 124 -135.

Young, S. and al., 1999. *The HTK book*. Entropic.

Zeppenfeld, T., M. Finke, K. Ries, M. Westphal, and A. Waibel. April 21-24 1997. “ Recognition of Conversational Telephone Speech Using the Janus Speech Engine ”. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. p. 557-560.