Using Parallel Architectures in Speech Recognition

Patrick Cardinal¹,², Pierre Dumouchel¹,², Gilles Boulianne¹
¹Centre de Recherche Informatique de Montréal (CRIM), Montréal, Canada
²École de Technologie Supérieure, Montréal, Canada
{patrick.cardinal,pierre.dumouchel,gilles.boulianne}@crim.ca

Abstract

The speed of modern processors has remained constant over the last few years and thus, to be scalable, applications must be parallelized. In addition to the main CPU, almost every computer is equipped with a Graphics Processing Unit (GPU) which is in essence a specialized parallel processor. This paper explores how performances of speech recognition systems can be enhanced by using GPU for the acoustic computations and multi-core CPUs for the Viterbi search in a large vocabulary application. The multi-core implementation of our speech recognition system runs 1.3 times faster than the single-threaded CPU implementation. Addition of the GPU for dedicated acoustic computations increases the speed by a factor of 2.8, leading to a word accuracy improvement of 16.6% absolute at real-time, compared to the the single-threaded CPU implementation.

Index Terms: speech recognition, parallel, GPU, multi-core

1. Introduction

Large vocabulary automatic speech-recognition is a computationally intensive task. Most speech recognizers run under a sequential implementation that cannot take advantage of modern processors with multi-core technology. In order to exploit this power, a parallel speech recognition system must be implemented.

Other architectures specialized in parallel computations can be used as a coprocessor to outperform the speeds offered by a modern CPU alone. In fact, almost every modern-day computer contains such a device. Indeed, modern graphic cards incorporate 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 years, the GPU has evolved into a flexible processor.

A noteworthy technological advance was achieved in 2007, when NVidia and ATI introduced the unified architecture. This greatly enhanced the flexibility and usability of the GPU, to the extent that it is becoming a mainstream alternative for general purpose calculations.

This paper explores how parallel architectures can be used to improve speech recognition system performance. In this work, the GPU is used to compute acoustic probabilities while a multi-core CPU is dedicated to the Viterbi search. The paper is organized as follows. In section 2, we first make a wrap-up of GPU applications and parallel implementations of speech recognition systems. In the third section, we describe how acoustic computations have been implemented in the GPU and how the Viterbi search has been parallelized. In the fourth section, our experimental results are presented and we conclude by discussing what comes next in our parallel implementation of a speech recognition system.

2. Related Work

A parallel implementation of a speech recognition system is presented by Phillips et al. [1]. Their system builds the transducer on the fly during the decoding process. They have obtained a performance of 0.8x real-time on a 16 CPU computer for the North American Business News (NAB) database. The same experiment ran at a speed of 3.9x real-time on a single CPU.

In their experimentation of using external hardware for improving a speech recognition system, Nedveschi et al. [2] 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 gave results similar to software implementation. Lin et al. [3, 4] 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 computer vision, the GPU has been used by Fung [5] to accelerate signal processing algorithms such as blurring, low-pass filtering and downsampling. They obtain a speed-up by a factor of 3.5 over a CPU. In image processing, Erra [6] implemented fractal image compression algorithms on a NVidia GeForce FX 6800 and obtained a speed-up of 280x over the equivalent CPU-based algorithm. In computational geometry, GPU has been successfully used in distance fields, collision detection, transparency computation and particle tracing. These, and many other applications implemented on GPUs are discussed in [7].

In signal processing, the FFT has been efficiently implemented in a GPU considering that the FFT is clearly memory bound. An openGL implementation, described in [8], yields a gain of 1.9x. More recently, a RapidMind [9] implementation showed a gain of 2.7x over a highly optimized CPU implementation running on the fastest CPU at the time [10].

The use of GPU has been explored for feature extraction by Bremer et al. [11]. The implementation showed a speed-up of 7.1x over the software implementation. Chong et al. [12] implemented a 50000 word speech recognizer system in a GPU and they obtain a speed-up of 9x over a CPU implementation. However, their implementation does not use Weighted Finite-State Transducers [13] that make up current state-of-the-art systems. Moreover, in many real-world applications, GPUs do not have enough memory to contain all the required data. For this reason, we prefer to establish a cooperation between the CPU and the GPU.
3. Implementation

3.1. CPU Implementation

The parallel Viterbi search is very similar to the sequential one. The following describes the changes applied to our sequential version.

![Parallelized Viterbi search](image)

**Figure 1: Parallelized Viterbi search**

The set of active states is divided into \( n \) subsets, where \( n \) is the number of threads dedicated to the state expansion process. As shown in Figure 1, some transitions lead to states belonging to another thread. Updating these states can lead to data incoherence. One solution is the use of a protection mechanism such as mutexes, which are very time consuming. Another approach is to duplicate destination state informations and merge them once all states have been expanded. Although this solution implies an overhead, it is much faster than the use of mutexes and, for this reason, is the approach upon which our implementation is based.

At the end of the expansion process, the best score is found and state pruning is applied in parallel. Then, surviving states of each subset are merged together to create the active state set for the next iteration. The merging step is performed sequentially. We should however point out that if the number of threads becomes important, it may prove more efficient to perform this step in parallel using the reduction algorithm [8].

Since a distribution likelihood can be used more than once, a cache is used to store precomputed values. In our parallel implementation, this cache is shared among all threads which means that if the acoustic probability for a given distribution has been computed, the result will be available by any other thread if needed. Consequently, in this scheme, the acoustic computations are also parallelized. Note that it is possible that two or more threads need to compute the same acoustic likelihood at the same time. In this case, the probability will be computed twice. Fortunately, this is a rare event (less than 0.1% in our experiments).

3.2. GPU Implementation

In our implementation, searching the recognition network is performed by the CPU while the computation of acoustic probabilities is delegated to the GPU.

The natural logarithm of the likelihood for a single Gaussian can be expressed as:

\[
\ln b_{je}(\vec{a}_t) = \ln \alpha_{jc} - \frac{1}{2} \ln((2\pi)^{d/2} |\Sigma_{jc}|) - \frac{1}{2} \vec{\mu}_{jc} \Sigma_{jc}^{-1} \vec{\mu}_{jc} + \vec{\mu}_{jc} \Sigma_{jc}^{-1} \vec{a}_t - \frac{1}{2} \vec{a}_t \Sigma_{jc}^{-1} \vec{a}_t
\]

The first three terms are independent of the observations and can be considered a Gaussian-specific constant that can readily be pre-computed. Denoting this term by \( h_{jc} \), it is:

\[
h_{jc} = \ln \alpha_{jc} - \frac{1}{2} \ln((2\pi)^{d/2} |\Sigma_{jc}|) - \frac{1}{2} \vec{\mu}_{jc} \Sigma_{jc}^{-1} \vec{\mu}_{jc}
\]

Excluding the observations, the last two terms can be denoted by:

\[
\begin{align*}
u_{jc} &= \vec{\mu}_{jc} \Sigma_{jc}^{-1} \\
v_{jc} &= Diag(-\frac{1}{2}\Sigma_{jc}^{-1})
\end{align*}
\]

where \( \Sigma_{jc}^{-1} \) is the diagonal covariance matrix. The likelihood for a single Gaussian can thus be expressed as:

\[
b_{jc}(\vec{a}_t) = (h_{jc} + u_{jc} \vec{a}_t + v_{jc} \vec{a}_t^2)
\]

This computation can be accomplished by a dot-product of the following two vectors, in which subscripts designating the distribution component have been omitted for clarity:

\[
\begin{align*}
o\vec{b} &= (1, o_1, o_2, \cdots, o_n, o_1^2, o_2^2, \cdots, o_n^2) \\
\vec{M} &= (h, \mu, \sigma_{11}^{-1}, \cdots, \mu, \sigma_{nn}^{-1}, -\frac{1}{2} \sigma_{11}^{-1}, \cdots, -\frac{1}{2} \sigma_{nn}^{-1})
\end{align*}
\]

where \( 1 \) is the identity element of multiplication. The likelihood of a distribution is defined as:

\[
\ln b_j(\vec{a}_t) = \bigoplus_{c=1}^{C_j} (o_{\vec{b}c} \cdot \vec{M}_{jc})
\]

where \( \bigoplus \) is the logarithmic addition and is defined as \( \ln(e^x + e^y) \). In this form, the computation of acoustic probabilities is perfectly suitable for a GPU since each distribution can be independently computed in parallel, and the results rest upon basic dot product operations.

3.2.1. CUDA development framework

We have implemented the acoustic computation module in CUDA, a development framework for NVidia graphic cards [14]. The CUDA framework shows 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 many of them can be concurrently launched. A kernel is made up of configurable amounts of blocks, each of which consists in a configurable amount of threads.

At execution time, each block is assigned to a multiprocessor. More than one block can be assigned to a given multiprocessor. Blocks are divided in groups of 32 threads called warps. In a given multiprocessor, 16 threads (half-warp) are executed at the same time. A time slicing-based scheduler switches between warps to maximize the use of available resources.
There are two kinds of memory. The first is the global memory which is accessible by all multiprocessors. Since this memory is not cached, it is important to ensure that the read/write memory accesses by a half-warp are coalesced in order to improve the performance. The texture memory is a small part of the global memory which is cached. The texture memory can be efficient when there is locality in data.

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

### 3.2.2. Kernel for acoustic calculation

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 [8] which uses the addition as reduction operator, except for the last Cj number of operations, for which the logarithmic addition is used to complete the reduction.

In our implementation, the computation of a mixture likelihood is computed by one-block of threads. Consequently, the number of launched blocks is the number of distributions in the acoustic model. Each block contains 256 threads.

For efficiency, the observation vector $o_{th}$ is copied $C_j$ times. As a result, it is the same length as a distribution vector. There is thus a direct correspondence between its elements and those of $\hat{M}$, thus avoiding index calculations.

Moreover, to ensure efficiency of the reduction process and coalescing access to the global memory, the model vector $\hat{M}$ is reorganized at the distribution level. It’s organized such that the $C_j$ first elements are the constants, followed by the $\mu_j \sigma_{j}^{-1}$ value of each component and so on. Figure 2 shows an example of the reduction algorithm applied in this context. In this figure, $\mu_{a,c}$ and $\sigma_{a,c}$ denote the $\mu_j \sigma_{j}^{-1}$ and $-\frac{1}{2} \sigma_{a,c}$ values of component $c$.

Note that the observation vector has also been reorganized in the same way to ensure consistency. In this example, the likelihood of a 2-dimensional observation is calculated by using a 2-Gaussian mixture model. Since the reduction algorithm works on power of 2 vectors, both the observation and model vector have to be padded with a neutral element which is 0 in this case.

### 4. Experimental Results

#### 4.1. Experimental Setup

Experiments have been conducted with a FST-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 local television programs in Quebec. The programs are a mix of weather, news, talk shows, etc. which have been transcribed manually at CRIM. 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 with diagonal covariance matrices. The test set is made up of 88 minutes of similar audio.

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). The vocabulary size is 59624 words.

The GPU implementation of acoustic calculations uses the SSE registers and runs on a Intel Core 2 quad at 2.3 GHz. Required probabilities are computed on-demand, which amount to approximately 40% of all probabilities.

The GPU used is the NVidia GeForce GTX 280 which contains 15 multiprocessors for a total of 240 stream processors and has 1024 MB of RAM. The implementation of GPU functions in our speech recognition system is straightforward. Computing likelihood on-demand is very costly in the GPU which works better on big sets of data. Consequently, it is more efficient to compute all acoustic likelihoods at each frame. The GPU computes acoustic likelihoods as fast as possible. Precomputed likelihoods are stored in a buffer in computer memory until used by the Viterbi decoder. When the buffer is full, the GPU simply waits for free space.

#### 4.2. Results

The baseline has been computed with our original decoder that uses only one core. We experimented with 32 (Figure 3a) and 128 (Figure 3b) Gaussian models.

We first experimented the multithreaded version of the decoder in which the Viterbi search and acoustic computations are parallelized. With the 32-Gaussian model, the results show that the speed increases by a factor of 1.2 over the single-threaded implementation. The relative improvement increases to 1.3 with the 128-Gaussian model. This speed-up leads to an improvement in word accuracy at real-time. Using acoustic models of 32 Gaussians, the multithread approach increases the word accuracy at real time up to 1.2% absolute. With the 128-Gaussian model, this improvement increases up to approximately 10% absolute.

The use of GPU for acoustic likelihood computations and a single CPU core for the Viterbi decoder leads to a more impressive improvement, particularly when acoustic models with a high number of Gaussians are used. In the case of 32 Gaussian models, the speed-up is 2.1 times faster which leads to a word accuracy improvement at real-time of 2.4% absolute. With the 128-Gaussian model, the speed-up is 2.8 times faster which leads to a word accuracy improvement of approximately 16.6% absolute at real-time. A higher number of Gaussians should lead to yet better real-time results, since even with
Graphics Processor Unit. The use of multi-core processors is up to 1.3 times faster leading to an improvement up to 10% absolute in the accuracy at real-time, while the use of a GPU is 2.8 times faster, improving the word accuracy up to 16.6% absolute at real-time.

We are now looking for a better implementation of the parallel Viterbi decoder which could improve the performances of the CPU-GPU implementation. This new implementation should also lead to better performances for the CPU-only implementation.

We also plan to implement the Viterbi search in the GPU, for a GPU-only implementation. We plan to distribute the underlying computational tasks among the CPU and GPU in several ways in order to find the combination that leads to optimal performance.

6. Acknowledgements

This project was funded in part by the Canadian Heritage New Media Research Networks Fund, which promotes innovation in new media or interactive digital content that pertain to the Canadian cultural sector.

7. References