INTERSPEECH 2009 BRIGHTON

Parallel Fast Likelihood Computation for LVCSR using Mixture Decomposition

Naveen Parihar[1], *Ralf Schlüter* [2], *David Rybach*[2], *Eric A. Hansen*[3]

[1]Dept. of Electrical and Computer Engineering, Mississippi State University, USA.
[2]Human Lang. and Pattern Recognition, Comp. Sc. Dept., RWTH Aachen University, Germany.
[3]Dept. of Computer Science and Engineering, Mississippi State University, USA.

np1@ece.msstate.edu, {schlueter,rybach}@cs.rwth-aachen.de, hansen@cse.msstate.edu

10.21437/Interspeech.2009-565

Abstract

This paper describes a simple and robust method for improving the runtime of likelihood computation on multi-core processors without degrading system accuracy. The method improves runtime by parallelizing likelihood computations on a multi-core processor. Mixtures are decomposed among the cores and each core computes the likelihood of the mixture allocated to it. We study two approaches to mixture decomposition – Chunk based and Decision-tree based. When applied to RWTH TC-STAR EPPS English LVCSR system on an Intel Core2 Quad processor with varying pruning-beam width settings, the method resulted in a 54% to 70% improvement in the likelihood computation runtime, and a 18% to 59% improvement in the overall runtime.
Index Terms: speech recognition, fast likelihood computations, fast gaussian calculations, parallel processing

1. Introduction

Gaussian-mixture likelihood computations take from 30% to 90% of total computation time in state-of-the-art LVCSR systems. A number of methods have been proposed to increase the speed of likelihood computations. These methods can be broadly divided into two categories. The first category consists of methods that achieve speedup by sacrificing a slight amount of recognition accuracy. Methods such as Gaussian components selection based on nearest neighbor search [1], and tree-based search [2, 3, 4, 5, 6], reduce the number of likelihood computations by evaluating only a subset of components in a mixture. These methods are sensitive to tuning parameters for the recognition system and poor parameter tuning degrades accuracy.

The second category consists of methods that do not result in any degradation in system accuracy. Methods such as partial distance elimination (PDE) [7], and maximum probability increase estimation (MPIE) [8], fall in this category. There is also a sub-category of methods that make use of efficient programming techniques. Examples in this subcategory include the use of SIMD instructions in modern processors to parallelize likelihood computations [9], and likelihood batch strategy [10].

This paper presents a method for improving the runtime of likelihood computations by harnessing the computational power of multi-core processors. Mixtures are statically decomposed among the cores. At runtime, each core computes the likelihood of the portion of the active mixtures that belong to it. Thus, the likelihood computation runs in parallel on multiple cores resulting in a decrease in elapsed runtime. In this paper, we compare and analyze two approaches to mixture decomposition – Chunk based and Decision-tree based.

2. Fast Likelihood Computation

We begin with a brief review of the likelihood computation module [9]. It is an on-demand module that computes the likelihood of the mixtures corresponding to the active search-space.

2.1. Likelihood

The likelihood $p(x_t|m)$ of a mixture index m is expressed as the weighted sum of likelihoods, $p(x_t|l,m)$, of its density components, for an acoustic vector $x \in IR^D$:

$$p(x_t|m) = \sum_{l=1}^{l(m)} p(l|m) \cdot p(x_t|l,m), \quad (1)$$

where $p(l|m)$ is the weight of the l^{th} density component $p(x_t|l,m)$.

Assuming a Gaussian distribution with a diagonal pooled covariance matrix that scales both the mean and the observation vectors as $\mu'_{m,l}$ and x'_t , the mixture negative log-likelihood can be computed by minimizing over the squared l_2 norm,

$$-\log(p(x_t|m)) = \min_l\{\frac{1}{2}||x'_{t,d} - \mu'_{m,l,d}||^2 + c_{m,l}\}, \quad (2)$$

where $c_{m,l}$ is the constant composed of component weight and normalization term, and d is the feature vector dimensionality.

2.2. Quantization and SIMD

The scaled floating point mean and observation vectors are quantized to unsigned 8-bit integer values in order to exploit parallel SIMD capabilities (SSE2) of modern processors to compute the squared norm,

$$x''_{t,d} = \alpha \cdot x'_{t,d} + \beta, \quad (3)$$

$$\mu''_{t,l,d} = \alpha \cdot \mu'_{t,l,d} + \beta, \quad (4)$$

where α is a single scaling value and β is a single bias for all the vector components. Their values are appropriately chosen to minimize the number of overflows.

The negative log-likelihood of the mixture is then given by:

$$-\log(p(x_t|m)) = \frac{1}{\alpha^2} \cdot \min_l\{\frac{1}{2}||x''_{t,d} - \mu''_{m,l,d}||^2 + c_{m,l}\}. \quad (5)$$

2.3. Caching Likelihood Every Frame

Caching of mixture likelihood scores during processing of each frame is possible for the following reasons: (a) The use of phonetic decision trees allows sharing a set of unique mixtures between HMM states. (b) Lexical prefix tree search keeps copies of trees for different contexts, which means that multiple hypotheses of the same HMM states (of the same tree arcs/words) can be expected to be active at a time. (c) Within a single tree copy, different words, and therefore different arcs share the same triphones/HMM states.

Whenever a mixture is evaluated during processing of a frame, the mixture's likelihood score is cached. In the same time frame, if any other active HMM state tied to the same mixture needs to compute its likelihood, the likelihood is retrieved from cache. The speed-up from caching the likelihood scores depends on the degree to which the groups of HMM states share the same mixtures, and the average number of lexical prefix tree copies. This process is depicted in Figure 1. If N_m is the number of unique mixtures, the likelihood cache is a vector of dimension $(1 * N_m)$.

2.4. Likelihood Pre-computation

In addition to the optimization described above, we added another optimization that involves likelihood pre-computation corresponding to the next eight frames. This optimization is similar to the likelihood batch strategy described in [10]. In the approach described in section 2.3, the likelihood scores of all active mixtures in a given frame are computed in that specific frame. During the likelihood computation at each frame, the mean vectors corresponding to the active mixtures are read from memory. These read operations incur significant cache misses and the main motivation is to reduce these cache misses, thereby increasing the speed of likelihood computations. Speech signals can be considered quasi-stationary and speech features in neighboring frames are likely to have similar distributions. Hence, the mixtures that are active in the current frame are also likely to be active in the neighboring future frames. Whenever mean vectors corresponding to a mixture are read from memory, the likelihood scores corresponding to the current frame and the next eight future frames are computed and cached in memory.

3. Mixture Decomposition

As described in the previous section, SIMD based parallelization divides the most basic operation of likelihood computation, the distance computation (l_2 norm), given a density. Hence, the SIMD based parallelization operates at a fine level of granularity. However, to distribute likelihood computations among multiple cores, we choose to parallelize at a coarser level of granularity. In particular, we parallelize the likelihood computations at mixture level by dividing the mixtures among multiple cores. The scalability of this approach is limited by the number of unique mixtures in the system.

Chunk based Approach: One way to distribute likelihood computations among multiple cores is to take a list of overall mixtures and simply divide it in chunks among the cores.

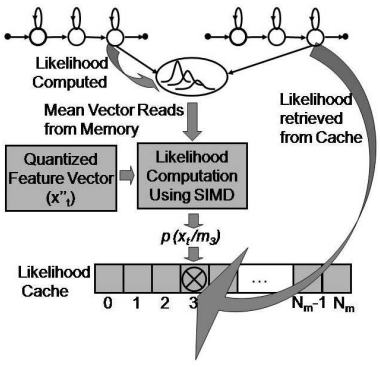

Figure 1: *Likelihood Computation Module.*

Hence, each core is assigned a set of unique mixtures. At runtime, each core computes the likelihood of the active mixtures allocated to it.

Decision-tree based Approach: Another approach to distributing the likelihood computations among cores exploits the fact that during the processing of each frame during search, HMM states (and therefore, corresponding mixtures) that produce high likelihood of the input speech data are likely to be evaluated. We distribute the acoustically similar mixtures among multiple cores. However, the key to good load balancing between the cores for likelihood computation using this approach is the identification of sets of mixtures that are acoustically similar. We use a phonetic decision tree to accomplish this goal.

The construction of a phonetic decision tree is both a knowledge and data-driven process. Each leaf node of a binary phonetic decision tree represents a tied mixture. Any leaf node has the greatest acoustic similarity to the leaf node with the same parent node. Let's assume a complete binary tree of height h. The $(h + 1)$ levels of this tree vary from level 0 at the root (top) node and level h at the leaf nodes (bottom). A non-leaf node at level $(h - 1)$ represents a set of two child leaf nodes that are acoustically similar and can be distributed among 2 cores. There are 2^h such acoustically similar sets, each with 2 leaf nodes. Similarly, if we need to divide the leaf nodes among 4 cores, we can identify non-leaf nodes at level $(h - 2)$. Each of these non-leaf will correspond to a set of 4 child leaf nodes that can be distributed among 4 cores. The binary tree has 2^{h-1} such sets.

The phonetic decision tree is not a complete binary tree, and one way of successfully dividing the leaf nodes in a way that maximizes the division of acoustically similar mixtures among the cores is to first create a list of leaf nodes in a strict left-to-right order. This can be done by traversing the binary tree in a depth-first fashion. Clearly, the list of leaf nodes can then be divided among the cores by interleaving. This approach can be applied to LVCSR systems with a single decision tree or multiple decision trees [11]. For multiple decision trees, this approach has to be applied to each of the decision trees.

4. Experimentation

4.1. Recognition System

All the experiments were performed using the RWTH 2007 TC-STAR EPPS English LVCSR baseline system 1 [12]. This system uses an acoustic front end consisting of 16 melfrequency cepstral coefficient (MFCC) features (including the zeroth coefficient) derived from 20 filterbanks. Cepstral mean normalization was applied and a voicedness feature was appended to the 16 features. The MFCCs and voicedness features from nine consecutive frames were concatenated and a linear discriminative analysis was used to reduce the dimensions to 45 components. The system realizes a one-pass four-gram lexical prefix tree based Viterbi decoder, using ML estimated acoustic models with vocal tract length normalization. Phonetic decision tree based state tying resulted in a single tree with 4501 generalized triphone states (mixtures). These mixtures consist of a total of 880,244 density components or an average of 195.56 density components per mixture. We use the term *mixture* to refer to a mixture of Gaussian distributions and the term *density component* to refer to an individual Gaussian of the mixture. Ten segments totaling 121.7 seconds were selected from the 2007 evaluation set as a test set to facilitate rapid experimentation. All the timing measurements are averaged over five runs.

Experiments were run on an Intel Core 2 Quad (Model number Q6600) running Fedora Core 6 Linux operating system. Each of the four cores runs at a clock speed of 2.40 GHz. Each core has its own 32 KB L1 data cache and 32 KB L1 instruction cache. Each pair of cores shares one of the two available 4 MB L2 caches. The total size of L2 cache is 8 MB. The size of main memory is 2 GB. The compiler used is GNU gcc version 4.1.1. The parallel version is implemented using the P-thread library.

4.2. Feasibility Analysis

The three main steps of the likelihood computation module are:

1. Looping over all the active HMM states,
2. Cache-Management, described in sections 2.3, 2.4, and
3. Distance-Computation using SIMD.

Figure 2 presents the distribution of the total elapsed runtime among these three steps, and other (Search Mang.+LM) computations. Standard deviation in runtime was observed between $0.25 - 2.00$ for various beam-pruning widths. The WER corresponding to all the width settings is at 8.8% with one exception. When the width was set to 200, the WER increased to 10.0%. We draw several conclusions. First, when the width is reduced, all the three steps of likelihood computations module together take larger percentage of total runtime compared to rest of the computations involving Search Management and LM Computations. With a wider width, a greater number of HMM states are active but many of the active HMM states share same underlying mixtures. Hence, once all the unique mixtures are evaluated, the likelihood is retrieved from the likelihood cache which is faster. Second, Distance-Computation dominates among the three likelihood computation steps at short beam-pruning widths but Cache-Management starts to dominate at wider beam-pruning widths. Third, the time taken by the Looping step increases when the width is increased because the number of active HMM states increases.

In our parallel implementation, each core executes the Loop step over all active HMM states. A mixture corresponding to an active HMM state is evaluated only if it belongs to the core. Hence, the time complexity of the Loop step in the parallel implementation remains the same. However, both Distance-Comp. and Cache-Mang. get distributed between the cores and run concurrently. In order to achieve good speedup, the time taken by each of these two parallel steps during the processing of each frame should ideally be perfectly balanced between the cores. We can estimate the time taken by each of these parallel steps based on one or more statistics about the search space. The number of density components evaluated is a search space statistic that influences the time taken by the Distance-Comp. step. Because we distribute the mixtures equally between the cores and the number of density components in each mixture vary, there is a possibility of an uneven distribution of density components among the cores. Similarly, the time taken by Cache-Mang. is dependent on the number of cache lookups.

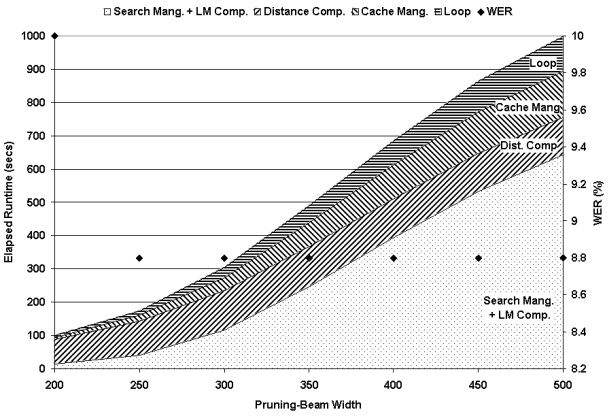

Figure 2: Distribution of overall elapsed runtime in seconds.

In general, we can predict the speedup for each parallel step, p, based on its search space statistic, s, as follows:

$$\hat{S_p} = \frac{serial\_runtime}{parallel\_elapsed\_runtime} = \frac{N_c \cdot \beta_s}{\beta_s + \alpha_s}, \quad (6)$$

where β_s is the perfectly balanced work due to search space statistic s, α_s is the load imbalance in search space statistic s, and N_c is the total number of cores.

If N_f is the total number of frames, the perfectly balanced work due to search space statistic s is given by:

$$\beta_s = \sum_{f=1}^{N_f} \beta_{s,f}, \quad (7)$$

$$\beta_{s,f} = \frac{\sum_{c=1}^{N_c} n_{s,f,c}}{N_c}, \quad (8)$$

where $n_{s,f,c}$ is the value of search space statistic s in core c during the processing of frame f.

The load imbalance in search space statistic s is:

$$\alpha_s = \sum_{f=1}^{N_f} \max_{c=1}^{N_c} \{n_{s,f,c} - \beta_{s,f}\}, \quad (9)$$

For a perfectly balanced load, $n_{s,f,c} = \beta_{s,f}, for\ all\ c, f$. Hence, $\alpha_s = 0$ and $\hat{S_p} = N_c$. Because load imbalance in any parallel step only influences the time spent in that specific step, we can apply these speedups to the time taken by the Distance-Comp. and Cache-Mang. steps in Figure 2, and add them to get a prediction of the elapsed runtime. The search space statistics for both these parallel steps are described earlier in this section. Adding the time taken by the serial steps (Search Mang.+LM Comp.+Loop) to the predicted runtime taken by parallel steps, provides the prediction of the elapsed runtime as shown below:

$$\hat{t} = \sum_{p=1}^{N_p} \{ts_p \cdot \hat{S_p}\} + \sum_{r=1}^{N_r} ts_r. \quad (10)$$

where N_p is the number of steps in serial search that can be parallelized, N_r is the number of serial steps, and ts_x is the time taken by search step x.

The predicted elapsed runtime ($\hat{t}$) for both Chunk based and Decision-tree based approaches is shown in Table 1. Similarly, Table 2 presents $\hat{t}$ due to all the three likelihood computation steps. Clearly, the Decision-tree based approach provides better load balancing when the pruning-beam width is small.

4.3. Results and Analysis

A comparison of the overall elapsed runtime for 2-core and 4-cores is presented in Table 1. Similarly, Table 2 shows the elapsed runtime due to likelihood computations. We make the following observations:

- As predicted in the previous section, the improvement in overall runtime using the Decision-tree based approach is better than the Chunk based approach at small beam widths. As the width is increased, the Chunk based approach starts to outperform the Decision-tree based approach. On 2-cores, the best overall runtime improvement of the two approaches varies approx. between

beam	1-core	2-core						4-core					
		Chunk based			Decision-tree based			Chunk based			Decision-tree based		
		$\hat{t}$	t		$\hat{t}$	t		$\hat{t}$	t		$\hat{t}$	t	
200	99.2	67.7	67.5	(31.9%)	60.7	60.4	(39.1%)	50.8	49.2	(50.3%)	41.2	41.7	(57.9%)
250	173.8	125.2	121.9	(29.8%)	116.1	113.8	(34.5%)	97.0	93.8	(46.0%)	86.9	86.5	(50.2%)
300	303.5	239.1	228.8	(24.6%)	227.4	221.1	(27.1%)	201.3	193.0	(36.3%)	189.0	187.2	(38.3%)
350	490.7	413.2	392.3	(20.0%)	398.1	386.3	(21.2%)	366.7	351.4	(28.3%)	351.3	349.5	(28.7%)
400	684.4	596.6	569.4	(16.7%)	578.6	564.3	(17.5%)	543.5	524.5	(23.3%)	525.0	529.3	(22.6%)
450	863.6	767.4	731.6	(15.2%)	747.0	727.6	(15.7%)	708.8	685.3	(20.6%)	688.1	695.0	(19.5%)
500	998.3	896.9	860.9	(13.7%)	875.4	860.3	(13.8%)	835.3	813.9	(18.4%)	814.5	825.9	(17.2%)

Table 1: Overall elapsed runtime in seconds. Improvement over serial system (1-core) in percentage is shown within parenthesis.

beam	1-core	2-core						4-core					
		Chunk based			Decision-tree based			Chunk based			Decision-tree based		
		$\hat{t}$	t		$\hat{t}$	t		$\hat{t}$	t		$\hat{t}$	t	
200	85.2	53.8	51.4	(39.7%)	46.8	45.3	(46.8%)	36.1	31.9	(62.5%)	27.3	25.8	(69.6%)
250	132.4	83.8	76.5	(42.2%)	74.7	70.4	(46.8%)	55.6	46.4	(64.9%)	45.5	41.6	(68.5%)
300	186.7	122.3	106.1	(43.1%)	110.5	102.0	(45.3%)	84.4	66.4	(64.4%)	72.1	65.1	(65.1%)
350	243.0	165.5	139.9	(42.2%)	150.5	137.4	(43.4%)	119.0	93.6	(61.4%)	103.7	96.4	(60.3%)
400	290.4	202.7	171.3	(41.0%)	184.6	170.2	(41.3%)	149.5	121.1	(58.2%)	131.1	127.7	(56.0%)
450	331.6	235.4	197.2	(40.5%)	215.0	197.1	(40.5%)	176.8	145.0	(56.2%)	156.1	155.4	(53.1%)
500	355.8	254.5	215.9	(39.3%)	233.0	218.8	(38.5%)	192.8	163.3	(54.1%)	172.0	176.0	(50.5%)

Table 2: Elapsed runtime in seconds taken by likelihood computation. Improvement over serial system in % is shown within parenthesis.

14%-39%. Similar improvements on 4-cores varies approx. between 18%-58%. At larger widths, the better spatial cache locality in the Chunk based approach results in a lower number of Non-prefetched Retired Load L2 Cache Misses than the Decision-tree based approach. This is the primary reason behind more improvement provided by the Chunk based approach at larger widths.

- The greatest improvement in likelihood computations runtime on 2-cores varies between 39%-47%. On 4-cores, the improvement is between 54%-70%.
- For the most part, the actual elapsed runtime measurements are better than the predicted elapsed runtime measurements. The improvement is attributed to a smaller amount of the elapsed time taken by the loop step in the parallel module as compared to the serial version.

5. Conclusions

The Chunk based and Decision-tree based approaches for multicore parallelization of likelihood computations are robust because they do not degrade system accuracy. On 2-cores, when the pruning-beam width was varied, the elapsed runtime improved from 39% to 47% over the baseline likelihood computation module which already incorporates a number of fast likelihood computation algorithms such as parallel SIMD instructions, likelihood caching, and likelihood pre-computation. On 4-cores, the improvement lies between 54% and 70%. The overall runtime improved by 14% to 39% on 2-cores and 18% to 59% on 4-cores. This method is simple enough to be easily implemented in any speech recognition system in order to achieve similar runtime improvements.

6. Acknowledgements

This work was supported in part by NSF grant IIS-0812558.

7. References

[1] E. Bocchieri, "Vector Quantization for the Efficient Computation of Continuous Density Likelihoods," in *Proceedings of the ICASSP*, Minneapolis, MN, USA, April 1993, pp. 692–895.

[2] F. Seide, "Fast Likelihood Computation for Continuous-mixture Densities using a Tree-based Nearest Neighbor Search," in *Proc. of the EUROSPEECH*, Madrid, Spain, Sept. 1995, pp. 1079–1082.

[3] A. Chan *et al.*, "On Improvements to CI-based GMM selection," in *Proc. of the EUROSPEECH*, Madrid, Sept. 1995, pp. 565–568.

[4] J. Fritsch *et al.*, "Speeding Up the Score Computation of HMM Speech Recognizers with the Bucket Voronoi Intersection Algorithm," in *Proceedings of the EUROSPEECH*, Madrid, Spain, September 1995, pp. 1091–1094.

[5] ——, "The Bucket Box Intersection (BBI) Algorithm for fast approximative evaluation of Diagonal Mixture Gaussians," in *Proc. of the IEEE ICASSP*, Atlanta, USA, May 1996, pp. 837–840.

[6] S. Ortmanns *et al.*, "Fast Likelihood Computation Methods for Continuous Mixture Densities in Large Vocabulary Speech Recognition," in *Proceedings of the EUROSPEECH*, Rhodes, Greece, USA, September 1997, pp. 139–142.

[7] L. Fissore *et al.*, "Analysis and Improvements of the Partial Distance Search Algorithm," in *Proceedings of the IEEE ICASSP*, Minneapolis, MN, USA, April 1993, pp. 315–318.

[8] N. Morales *et al.*, "Fast Gaussian Likelihood Comp. by Maximum Probability Increase Estimation for Continuous Speech Recognition," in *Proc. of the ICASSP*, LasVegas, USA, April 2008.

[9] S. Kanthak, K. Schütz, and H. Ney, "Using SIMD Instructions for Fast Likelihood Calculation in LVCSR," in *Proceedings of the IEEE ICASSP*, Istanbul, Turkey, June 2000, pp. 1531–1534.

[10] M. Saraclar *et al.*, "Towards automatic closed captioning: low latency real-time broadcast news transcription," in *Proceedings of the ICSLP*, Denver, USA, September 2002.

[11] K. Beulen *et al.*, "State Tying for Context Dependent Phoneme Models," in *Proceedings of the EUROSPEECH*, Rhodes, Greece, September 1997, pp. 1179–1182.

[12] J. Lööf *et al.*, "The RWTH 2007 TC-STAR Evaluation System for European English and Spanish," in *Proceedings of the INTERSPEECH*, Antwerp, Belgium, August 2007, pp. 2145–2148.