Fast Speaker Diarization Using a High-Level Scripting Language

Ekaterina Gonina [#1], Gerald Friedland [*2], Henry Cook [#3], Kurt Keutzer [#4]

[#] *University of California, Berkeley*
[1] egonina@eecs.berkeley.edu
[3] hcook@eecs.berkeley.edu
[4] keutzer@eecs.berkeley.edu

[*] *International Computer Science Institute*
[2] fractor@icsi.berkeley.edu

***Abstract*—Most current speaker diarization systems use agglomerative clustering of Gaussian Mixture Models (GMMs) to determine "who spoke when" in an audio recording. While state-of-the-art in accuracy, this method is computationally costly, mostly due to the GMM training, and thus limits the performance of current approaches to be roughly real-time. Increased sizes of current datasets require processing of hundreds of hours of data and thus make more efficient processing methods highly desirable. With the emergence of highly parallel multicore and manycore processors, such as graphics processing units (GPUs), one can re-implement GMM training to achieve faster than real-time performance by taking advantage of parallelism in the training computation. However, developing and maintaining the complex low-level GPU code is difficult and requires a deep understanding of the hardware architecture of the parallel processor. Furthermore, such low-level implementations are not readily reusable in other applications and not portable to other platforms, limiting programmer productivity. In this paper we present a speaker diarization system captured in under 50 lines of Python that achieves 50-250× faster than real-time performance by using a specialization framework to automatically map and execute computationally intensive GMM training on an NVIDIA GPU, without significant loss in accuracy.**

I. Introduction

Speaker diarization is the process of segmenting an audio signal into speaker-homogeneous regions, addressing the question "who spoke when?" without any prior knowledge of the number of speakers, specific speaker models, text, language, or amount of speech present in the recording. One popular diarization method is the Bayesian Information Criterion (BIC) [1] with Gaussian Mixture Models (GMMs) trained with frame-based cepstral features [2],[3]. This method combines the speech segmentation and segment clustering tasks into a single stage using agglomerative hierarchical clustering, a process by which many simple candidate models are iteratively merged into more complex, accurate models. Figure 1 shows the general organization of such a diarization system.

While recent publications have achieved sub-realtime performance with specialized algorithmic optimizations (see Section III), performance still limits many applications that rely on processing large amounts of data and/or are not compatible with a certain algorithmic optimization. This limitation is especially relevant in applications where a speaker diarization system is used online, as part of large-scale event detection or as front-end for an automatic speech recognition system on a large corpus.

Fundamental physical constraints on processor design have led us into an era where programmers can no longer expect sequential implementations of their applications to obtain automatic performance improvements on new generations of processors [4]. However, these hardware constraints have led to the recent widespread commoditization of a variety of *parallel* platforms. Servers and laptops have multi-core CPUs, and many also contain high-end graphics processing units (GPUs) with tens of cores each capable of executing operations on large data vectors. Modern GPUs can be programmed with languages such as CUDA [5] and OpenCL [6], which are intended to be used to accelerate both general-purpose and scientific applications. Many speech processing algorithms have already seen significant speedups on parallel processors (see Section III).

While the performance benefits of creating a parallel implementation of an algorithm are appealing, developers must overcome two challenges to productivity: increased code complexity and a need for detailed knowledge of the underlying hardware architecture. In this paper, we present a new speaker diarization system that combines several algorithmic optimizations while automatically utilizing available parallel hardware. Our system exemplifies a tiered approach to parallel programming that provides the performance gains of parallel processing without incurring productivity burdens. Specifically, we use a *specialization framework* to automatically generate an optimized parallel implementation of the Gaussian Mixture Model training algorithm based on training problem parameters and the details of the available hardware parallelism.

Our speaker diarization system, based on agglomerative hierarchical clustering of GMMs using the BIC, is captured in about 50 lines of Python. However, using the specialization framework it achieves 50×-250× *faster than real-time*[1]

[1]For readability, we inversed the xRT numbers throughout the paper: 10× faster than real-time denotes that processing ten minutes of audio requires 1 minute.

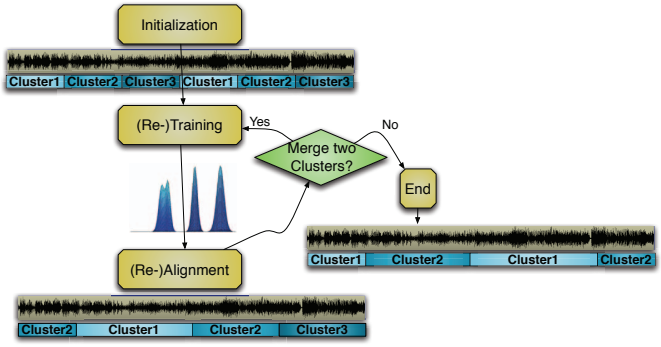

Fig. 1. *Illustration of the segmentation and clustering algorithm used for speaker diarization.*

performance by utilizing a parallel NVIDIA GPU processor, without significant loss in the diarization accuracy.

Furthermore, any other algorithm that uses GMM training (e.g. acoustic modeling for automatic speech recognition or speech activity detection) can use the same framework to automatically obtain significant performance improvements without significant engineering effort. With this paper we would like to encourage the research community to do so.

This paper is structured as follows: Section II describes the speaker diarization system. Section III discusses related work. Section IV describes the computational bottlenecks of the diarization system, algorithmic optimizations and parallelization of GMM training on the GPU. Section V discusses the specialization framework and the process for automatically mapping the training function to the parallel hardware. Section VI shows performance results and Section VII concludes.

II. Speaker Diarization

Our diarization system consists of a pre-processing phase and a segmentation and clustering phase.

A. Pre-Processing

The diarization is based on 19-dimensional, gaussianized, Mel-Frequency Cepstral Coefficients (MFCCs). We use a frame period of 10 ms with an analysis window of 30 ms in the feature extraction, as well as the speech/non-speech segmentation used in [7] and described in [8]. This methodology is an HMM/GMM system trained on broadcast news data that generalizes well to the meeting diarization context.

B. Speaker Diarization

In the segmentation and clustering stage of speaker diarization, an initial segmentation is generated by uniformly partitioning the audio track into K segments of the same length. K is chosen to be much larger than the assumed number of speakers in the audio track. For meeting recordings of about 30 minute length, previous work [9] experimentally determined $K = 16$ to be a good value.

The procedure for diarization is shown in Figure 1 and takes the following steps (more details can be found in [7]):

1) Initialization: Train a set of GMMs, one per initial segment, using the expectation-maximization (EM) algorithm [10].
2) Re-segmentation: Re-segment the audio track using majority vote over the GMMs' likelihoods for $2.5\,s$ duration [11].
3) Re-training: Retrain the GMMs on the new segmentation.
4) Agglomeration: Select the most similar GMMs and merge them. At each iteration, the algorithm checks all possible pairs of GMMs, looking to obtain an improvement in BIC scores by merging the pair and retraining it on the pair's combined audio segments. The GMM clusters of the pair with the largest improvement in BIC scores are permanently merged. The algorithm then repeats from the re-segmentation step until there are no remaining pairs whose merging would lead to an improved BIC score.

The result of the algorithm consists of a segmentation of the audio track with n segment subsets and with one GMM for each subset, where n is assumed to be the number of speakers.

This system was proven to be highly effective in the past, but the computational burden was such that the processing took about real-time [12]. In the following sections we discuss how software frameworks such as ours can provide performance gains to mitigate such overheads, while still allowing innovation at the application level.

III. Related Work

There is much prior work on accelerating speaker diarization [13]. For example Huang et al. [12] describes strategies for selecting most likely clusters to merge during agglomerative clustering to enable fast matching. Most recently, Anguera et al. [14] presented an algorithmic trick to reduce speaker diarization processing time to almost $10\times$ faster than real-time at the cost of a slight decrease in accuracy.

Chong et al. discussed the benefits that parallelization of computationally intensive components can offer diarization applications [15]. Many speech and speaker recognition applications have also seen performance benefits from parallelization [16], [17], [18], [19]. While these efforts demonstrate significant speedups (e.g. Chong et al. achieve $10.5\times$ faster than real-time performance with a parallel speech recognition decoder [17]), their parallel implementations are written in complex low-level code, are not readily reusable in other applications, and the applications as a whole are not portable to other platforms. In contrast, our approach allows for code reuse by abstracting the GMM training computation into a specialization framework which can be used by any application on multiple parallel platforms, all transparent to the high-level application writer.

Our methodology for developing the specialization framework for accelerating GMM training using a high-level scripting language is based on the the Selective Embedded Just-In-Time Specialization (SEJITS) mechanism originally described in [20]. We use a specific implementation of SEJITS called

Asp [21][2]. Our framework extends the original GPU GMM training code described in [22]. Other parallel GMM implementations on the GPU include [23] and [24]. Dixon et al. ([23]) offloads the observation probability computation in a speech recognition engine to the GPU, while Kumar et al. ([24]) uses the GPU to train GMMs to obtain $164\times$ speedup over a sequential CPU implementation (no $\times$RT factor was given). EhKan et al. [25] implemented a GMM-based speaker identification system on FPGAs, achieving $90\times$ speedup over sequential software version.

IV. Speaker Diarization Implementation

Previous analysis of the diarization engine [26] showed two main computational bottlenecks: the training of the GMMs, and choosing the GMM pair to merge during agglomeration.

A. Algorithmic Optimizations

To mitigate the overhead of choosing the GMM pair to merge, we algorithmically reduced the re-segmentation bottleneck by using an unscented-transform based approximation of KL-divergence introduced by Huang et al. [12]. The approximation gives top k candidate pairs of GMMs to merge at a much lower computational cost without affecting accuracy.

Secondly, we reduced the bottleneck of training the Gaussian Mixture Models by using a parallel implementation of the training on the GPU, described in the following section.

B. GMM Training on a GPU

Training of the Gaussian Mixture Models is performed on the GPU by adopting the expectation-maximization algorithm described in [22] written in CUDA, a low-level extension to the C programming language [5]. We also have a Cilk+ [27] implementation that targets multi-core CPUs, however in this paper we focus on GPU performance.

A CUDA application is organized into sequential *host* code written in C running on the CPU and many parallel *device* kernels running on the GPU [5]. The kernel executes a set of scalar sequential programs across a set of parallel threads. The programmer can organize these threads into thread blocks, which are mapped onto the processor cores at runtime. Each core has a small software-managed fast local memory. To run an application on the GPU, data structures must be explicitly transferred from host to device. Task scheduling and load balancing are handled by the device driver automatically.

The parallel GMM training code consists of a set of kernel functions for the expectation and maximization steps of the EM algorithm. There are four parameters for this computation: K - the number of GMMs, M - the number of Gaussian components in each GMM, D - the dimensionality of the Gaussian components and N - the number of feature frames in the meeting. The algorithm is outlined as follows:

1) Copy the input data from the CPU to the GPU
2) Initialize the GMMs and copy model data to the GPU
3) Launch expectation kernels; aggregate log-likelihood values from each GMM
4) Launch maximization kernels; aggregate parameters from each GMM
5) Repeat steps 3 and 4 some number of times (we use 3)
6) Copy model parameter values to the CPU

In the expectation step, each thread block is assigned a Gaussian component and a subset of the feature frames to compute the log likelihood for. Each thread computes the log likelihood of one frame. The log likelihoods are then added across components for each frame.

In the maximization step, each parameter in the Gaussian models (weight, means and covariance matrix) is computed in a separate kernel. For the weight computation kernel, each thread block is assigned a Gaussian component (total of M thread blocks) and each thread computes the contribution of each frame to the weight of the Gaussian (N threads) followed by a normalization step. For the mean computation kernel, each thread block is assigned one dimension in one Gaussian component's mean vector ($M \times D$ thread blocks) and each thread is assigned a feature frame (N threads). For the covariance matrix computation we implement different versions of the kernel, as described in the next section.

1) Covariance Matrix Computation: Covariance matrix computation is the most computationally intensive step in the EM algorithm (50–60% of the overall runtime of the algorithm). We extend the parallel implementation from [22] to more efficiently compute the matrix for various training problem sizes in our specialization framework.

The covariance matrix for a Gaussian mixture component is the sum of the outer products of the difference between the observation feature vectors and the component's mean vector computed in the current iteration:

$$\Sigma_i^{(k+1)} = \frac{\sum_{j=1}^{N}(p_{i,j}(x_j - \mu_i^{(k+1)})(x_j - \mu_i^{(k+1)})^T)}{\sum_{j=1}^{N} p_{i,j}} \quad (1)$$

where $p_{i,j}$ is the probability of frame j belonging to component i and x_i is the feature frame.

The covariance computation exhibits a large amount of parallelism due to the mutual independence of each component's covariance matrix (M), each cell in a matrix (D), and each feature vector's contribution to a cell in the matrix (N). These degrees of parallelism allow different versions of the covariance kernel to target different dimensions of algorithmic parallelism. For example, one version assigns each thread block a Gaussian component and each thread a feature frame, while another assigns each thread block one cell in a matrix of one component and each thread a subset of feature frames (for a full description of the code versions refer to [28]).

2) Code Version Selection: The efficiency of each version depends on values of M, D and N as well as the hardware characteristics of the GPU. For example, during diarization the number of components in each cluster changes as we merge more clusters together. If the number of components in a GMM is smaller than the number of cores on the GPU

[2]Recursive acronym: Asp is a SEJITS for Python. Asp is a general framework for developing specializers, it is available at http://aspsejits.pbworks.com. Others are encouraged to contribute specialized algorithms and code generators.

and we assign each thread block to compute a covariance matrix of one component, we will underutilize the cores on the GPU and thus will need to choose a different kernel version. On average we saw a 30% performance improvement in covariance matrix computation time [28] by using the optimal code version compared to always using the original implementation described in [22].

Our framework automatically selects the best version given M, D and N values and the platform specifications instead of re-implementing the code for every specific application and manually optimizing it, as described in the next section.

V. Automatic Mapping to the GPU

Our framework uses the Selected Embedded Just-in-Time Specialization (SEJITS) [20] mechanism to automatically select best parallel code version of the GMM training and execute it on the GPU from high-level Python code.

A. SEJITS

We use selective just-in-time specialization as the mechanism to accomplish the separation of concerns: the application Python programmer can focus on developing and innovating the application and the parallel Cilk+/CUDA programmer can focus on developing fast parallel code for the target hardware architecture. Our GMM training specialization framework is implemented on top of Asp [21].

When using the SEJITS-based frameworks, scientists express their applications entirely in Python using Python libraries and tools. They also import a special library containing our GMM objects. When the training method of the GMM objects is called, the framework selects the best parallel version of the training algorithm, transfers the needed data to the GPU and executes the fast CUDA code, all transparent to the Python application writer. From the Python programmer's view, this experience is like calling a pure Python library, except that performance is potentially several orders of magnitude faster.

B. Code Version Selection

To find the best parallel implementation of the GMM training algorithm (specifically the covariance matrix computation), our specialization framework keeps track of the best performing implementation in a database. When a GMM training method is called on a particular problem size on a specific platform, the framework gets the name of the best-performing version from the database and executes it on the parallel processor. When a new generation of the GPU is available, the framework will automatically choose the best variant by running a set of test executions and recording best performing variant in the database.

Other specialized functions related to GMM evaluation and manipulation are also provided such as model data allocation and log likelihood computation.

C. Speaker Diarization in Python

Using Asp, the implementation of our diarization system is captured in less than 50 lines of Python code and is shown

```
1  from em import *
2
3  def cluster(self, M, D, K, data):
4
5    gmm_list = new_gmm_list(M,D,K)
6    per_cluster = N/K
7    init = uniform_init(gmm_list, data, per_cluster, N)
8    for gmm, data in init:
9      gmm.train(data)
10
11   # Perform hierarchical agglomeration
12   best_BIC_score = 1.0
13   while (best_BIC_score > 0 and len(gmm_list) > 1):
14
15     # Resegment data based on likelihood scoring
16     L = gmm_list[0].score(data)
17     for gmm in gmm_list[1:]:
18       L = np.column_stack((L, gmm.score(data) ))
19     most_likely = L.argmax()
20     split_data = split_obs_data_L(most_likely, data)
21
22     for gmm, data in split_data:
23       gmm.train(data)
24
25     # Score all pairs of GMMs using BIC
26     best_merged_gmm = None
27     best_BIC_score = 0.0
28     m_pair = None
29
30     #find most likely merge candidates using KL
31     gmm_pairs = get_top_K_GMMs(gmm_list, 3)
32
33     for pair in gmm_pairs:
34         gmm1, d1 = pair[0] #get gmm1 and its data
35         gmm2, d2 = pair[1] # get gmm2 and its data
36         new_gmm, score =
37             compute_BIC(gmm1, gmm2, concat((d1, d2)))
38         if score > best_BIC_score:
39           best_merged_gmm = new_gmm
40           m_pair = (gmm1, gmm2)
41           best_BIC_score = score
42
43     # Merge the winning candidate pair
44     if best_BIC_score > 0.0:
45       merge_gmms(gmm_list, m_pair[0], m_pair[1])
```

Fig. 2. *Speaker diarization in Python. Components that are executed on the GPU are highlighted in light-gray*

in Figure 2 with the components that are executed on the GPU highlighted in light-gray. The specialization framework itself is written in about 800 lines of Python and the C/CUDA code for GMM training is written in about 1500 lines. Both the specialization framework and the low-level GMM training code are written once and can be reused by all applications on any recent CUDA-programmable GPU.

Based on the algorithm description from Section II we now step through the Python code:

1) Initialization: First we import our specialization framework (line 1) and uniformly initialize a list of K GMMs (in our case 16 5-component GMMs) on line 5. After creating the list of GMMs, we perform initial training on equal subsets of feature vectors (lines 6-9). The training computation is executed on the GPU. Next, we implement the agglomerative clustering loop based on the Bayesian Information Criterion (BIC) score [29]

(line 12-13).

2) Re-segmentation: In each iteration of the agglomeration, we re-segment the feature vectors into subsets using majority vote segmentation (lines 16-18). We use the GPU to compute the log-likelihoods (`gmm.score()` method), which calls the E-step of the GMM training algorithm on the GPU.
3) Re-training: After re-segmentation we re-train the Gaussian Mixtures on the GPU on the corresponding subsets of frames (lines 22-23).
4) Agglomeration: After re-training, we decide which GMMs to merge by first computing the unscented-transform based KL-divergence of all GMMs (line 31). We then compute the BIC score of the top k pairs of GMMs (in our case $k = 3$) by re-training merged GMMs on the GPU as described in Section II (lines 33-37) and keeping track of the highest BIC score. Finally we merge two GMMs with the highest BIC score (lines 43-45) and repeat the iteration until no more GMMs can be merged.

All data structure allocation and transfers are handled by the framework transparent to the application writer. The following section describes accuracy and speedup results of the diarizer.

VI. Results

A. Test Sets

In order to evaluate the accuracy and speed of the approach described above, we use a popular subset of 12 meetings (5.4 hours) from the Augmented Multi-Party Interaction (AMI) corpus [30]. The AMI corpus consists of audio-visual data captured from four to six participants in a natural meeting scenario. The participants volunteered their time freely and were assigned roles such as "project manager" or "marketing director" for the task of designing a new remote control device. The teams met over several sessions of varying lengths (15–35 minutes). The meetings were not scripted and different activities were carried out such as presenting at a projector screen, explaining concepts on a whiteboard or discussing while sitting around a table. The meeting recordings are therefore very close to real-world scenarios and participants interacted naturally, including talking over each other.

The 12-meeting subset contains the most comprehensively annotated meetings in the corpus and is therefore quite popular among researchers [31]. Since our work investigates an unsupervised approach that does not require any tuning, there is no need to split the data into test and training sets. For the experiments described here, the beamformed far-field and near-field array microphone signals were used. To make the scores compatible with the baseline system, the Shout speech activity detection was used as described in Section II and in [7]. MFCC features were calculated as a preprocessing step. The feature computation and speech activity detection step is not part of the runtime computation.

After processing the files, the output of the diarization was scored using Diarization Error Rate, which is defined by NIST [32]. DER is composed of three additive components: misses (speaker in reference, but not in hypothesis), false alarms (speaker in hypothesis, but not in reference), and speaker errors (mapped reference is not the same as hypothesized speaker). Table I columns "FF DER" and "NF DER" show the accuracy in terms of % DER for far-field array microphone (FF) and near-field array microphone (NF) setups. The results for both evaluation sets are comparable to state-of-the art DER for the AMI corpus [31].

TABLE I
Diarization Error Rate (DER) of the diarization system and the faster than real-time performance factor (×RT) for far-field (FF) and the near-field (NF) microphone array setup for the AMI corpus.

Meeting ID	FF DER	FF ×RT	NF DER	NF ×RT
IS1000a	40.99 %	71.19×	25.38 %	72.83×
IS1001a	27.38 %	80.88×	32.34 %	163.22×
IS1001b	41.28 %	70.02×	10.57 %	123.28×
IS1001c	46.83 %	59.71×	28.40 %	177.80×
IS1003b	41.54 %	80.85×	34.30 %	254.81×
IS1003d	66.89 %	64.33×	50.75 %	56.13×
IS1006b	29.88 %	74.03×	16.57 %	129.35×
IS1006d	63.68 %	54.87×	53.05 %	58.36×
IS1008a	2.19 %	64.29×	1.65 %	60.35×
IS1008b	4.99 %	81.46×	8.58 %	151.80×
IS1008c	32.43 %	67.20×	9.30 %	81.13×
IS1008d	27.84 %	83.42×	26.27 %	55.77×
Average	35.49 %	71.02×	24.76 %	115.40×

B. Performance Results

Table I columns "FF ×RT" and "NF ×RT" show corresponding performance for far-field (FF) and near-field (NF) microphone setup in terms of faster than real-time factor (×RT) using our specialization framework on NVIDIA GTX480 GPU using CUDA 3.2. The ×RT factor is computed by dividing the meeting time by the processing time. For example, a 100× real-time factor means we can process a ten minute meeting in six seconds. On average the far-field microphone array meetings take longer to process than near-field (71.02×RT and 115.40×RT respectively) with higher DER on average (35.49% and 24.76% respectively). The real-time performance varies by each meeting from about 50-250×RT, depending on the length of the audio as well as the number of clustering iterations computed before convergence. Overall, the system achieved about 2× speedup by using the KL-divergence fast-match strategy (shown in [12]) and $70 - 115\times$ speedup by using the GPU.

VII. Conclusion

This paper describes a fast speaker diarization algorithm that achieves 50-250× faster than real-time performance without significant loss in accuracy. Our diarization approach is captured in less than 50 lines of Python code and achieves fast performance by using a specialization framework for Gaussian Mixture Model (GMM) training on an NVIDIA graphics processing unit (GPU). When mapping a problem instance onto a parallel hardware platform, our framework automatically selects the best parallel implementation of the GMM training algorithm based on the diarization problem size and the processor features. This automation allows the scientist to focus on developing the application algorithm while achieving significant additional performance improvements. Expressing

the application in Python is pedagogically useful - it allows for a clear, high-level specification of the algorithm that is accessible and easy to understand. Our framework is available at http://diarization.icsi.berkeley.edu/diarization/ for the community to use to develop other GMM-based applications that automatically utilize parallel hardware.

Our future work includes implementing speech activity-detection using the same framework and integrating multi-stream approaches for diarization into the system.

VIII. Acknowledgements

Thank you to Shoaib Kamil and Armando Fox for providing the Asp infrastructure and insightful discussions during this work. Research supported by CISCO URP Grant 2010-07822 (3696) and DARPA (contract #FA8750-10-1-0191), as well as Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung.

References

[1] D. Reynolds and P. Torres-Carrasquillo, "Approaches and Applications of Audio Diarization," *Acoustics, Speech, and Signal Processing, 2005. Proceedings.(ICASSP'05). IEEE International Conference on*, vol. 5, pp. 953–956, March 2005.

[2] S. S. Chen and P. S. Gopalakrishnan, "Speaker, environment and channel change detection and clustering via the bayesian information criterion," in *Proceedings of the DARPA Broadcast News Transcription and Understanding Workshop*, Lansdowne, Virginia, USA, February 1998. [Online]. Available: http://www.nist.gov/speech/publications/darpa98/pdf/bn20.pdf

[3] X. Anguera, C. Wooters, B. Peskin, and M. Aguilo, "Robust speaker segmentation for meetings: The ICSI-SRI spring 2005 diarization system," in *Proceeding of the NIST MLMI Meeting Recognition Workshop, Edinburgh*. Springer, 2005.

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006.

[5] *NVIDIA CUDA Programming Guide*, NVIDIA Corporation, March 2010, version 3.2. [Online]. Available: http://www.nvidia.com/CUDA

[6] *OpenCL 1.1 Specification*, Khronos Group, September 2010, version 1.1. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[7] C. Wooters and M. Huijbregts, "The ICSI RT07s Speaker Diarization System," Baltimore, Maryland, 2007, pp. 509–519.

[8] M. A. H. Huijbregts, "Segmentation, diarization and speech transcription : surprise data unraveled," Ph.D. dissertation, Enschede, November 2008. [Online]. Available: http://doc.utwente.nl/60130/

[9] D. Imseng and G. Friedland, "Robust speaker diarization for short speech recordings," in *Proceedings of the IEEE workshop on Automatic Speech Recognition and Understanding*, 12 2009, pp. 432–437.

[10] C. M. Bishop, *Neural Networks for Pattern Recognition*. New York: Oxford Univ. Press, 1995.

[11] G. Friedland and O. Vinyals, "Live speaker identification in conversations," in *Proceeding of the 16th ACM international conference on Multimedia*, ser. MM '08. New York, NY, USA: ACM, 2008, pp. 1017–1018. [Online]. Available: http://doi.acm.org/10.1145/1459359.1459558

[12] Y. Huang, O. Vinyals, G. Friedl, C. Mller, N. Mirghafori, and C. Wooters, "A fast-match approach for robust, faster than real-time speaker diarization," in *in ASRU*, 2007.

[13] X. Anguera, S. Bozonnet, N. W. D. Evans, C. Fredouille, G. Friedland, and O. Vinyals, "Speaker diarization : A review of recent research," *Accepted for publication in "IEEE Transactions On Acoustics Speech and Language Processing" (TASLP), special issue on "New Frontiers in Rich Transcription", 2011*, 2011.

[14] X. Anguera and J.-F. Bonastre, "Fast Speaker Diarization based on binary keys," *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2011.

[15] J. Chong, G. Friedland, A. Janin, N. Morgan, and C. Oei, "Opportunities and challenges of parallelizing speech recognition," in *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, ser. HotPar'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 2–2. [Online]. Available: http://portal.acm.org/citation.cfm?id=1863086.1863088

[16] K. You, J. Chong, Y. Yi, E. Gonina, C. Hughes, Y. Chen, W. Sung, and K. Keutzer, "Parallel scalability in speech recognition: Inference engine in large voc abulary continuous speech recognition," in *IEEE Signal Processing Magazine*, no. 6, November 2009, pp. 124–135.

[17] J. Chong, E. Gonina, Y. Yi, and K. Keutzer, "A fully data parallel WFST-based large vocabulary continuous speech recognition on a graphics processing unit," in *10th Annual Conference of the International Speech Communication Association (InterSpeech)*, September 2009.

[18] K. You, Y. Lee, and W. Sung, "OpenMP-based parallel implementation of a continous speech recognizer on a multi-core system," in *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, Taipei, Taiwan, 2009.

[19] J. Chong, Y. Yi, N. R. S. A. Faria, and K. Keutzer, "Data-parallel large vocabulary continuous speech recognition on graphics processors," in *Proc. Intl. Workshop on Emerging Applications and Manycore Architectures*, 2008.

[20] B. Catanzaro, S. Kamil, Y. Lee, K. Asanović, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox, "SEJITS: Getting productivity and performance with selective embedded JIT specialization," in *Workshop on Programming Models for Emerging Architectures (PMEA 2009)*, Raleigh, NC, October 2009.

[21] S. Kamil, D. Coetzee, and A. Fox, "Bringing parallel performance to python with domain-specific selective embedded just-in-time specialization," in *Python for Scientific Computing Conference (SciPy)*, 2011.

[22] A. D. Pangborn, "Scalable data clustering using gpus," Master's thesis, Rochester Institute of Technology, 2010.

[23] P. R. Dixon, T. Oonishi, and S. Furui, "Fast acoustic computations using graphics processors," in *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, Taipei, Taiwan, 2009.

[24] N. Kumar, S. Satoor, and I. Buck, "Fast parallel expectation maximization for gaussian mixture models on gpus using cuda," in *11th IEEE International Conference on High Performance Computing and Communications, 2009. HPCC'09*, 2009, pp. 103–109.

[25] P. EhKan, T. Allen, and S. F. Quigley, "Fpga implementation for gmm-based speaker identification," *Int. J. Reconfig. Comput.*, vol. 2011, pp. 3:1–3:8, January 2011. [Online]. Available: http://dx.doi.org/10.1155/2011/420369

[26] G. Friedland, J. Chong, and A. Janin, "Parallelizing speaker-attributed speech recognition for meeting browsing," in *Proceedings of the 2010 IEEE International Symposium on Multimedia*, ser. ISM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 121–128. [Online]. Available: http://dx.doi.org/10.1109/ISM.2010.26

[27] *Cilk 5.4.6 Reference Manual*, version 5.4.6. [Online]. Available: http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf

[28] H. Cook, E. Gonina, S. Kamil, G. Friedland, D. Patterson, and A. Fox, "Cuda-level performance with python-level productivity for gaussian mixture model applications," in *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, May 2011.

[29] D. Reynolds and P. Torres-Carrasquillo, "Approaches and applications of audio diarization," in *Acoustics, Speech, and Signal Processing, 2005. Proceedings. (ICASSP '05). IEEE International Conference on*, vol. 5, march 2005, pp. v/953 – v/956 Vol. 5.

[30] "Ami corpus." [Online]. Available: http://corpus.amiproject.org/

[31] G. Friedland, C. Yeo, and H. Hung, "Dialocalization: Acoustic speaker diarization and visual localization as joint optimization problem," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 6, pp. 27:1–27:18, November 2010. [Online]. Available: http://doi.acm.org/10.1145/1865106.1865111

[32] "National institute of standards and technologies: Rich transcription spring 2004 evaluation." [Online]. Available: http://www.itl.nist.gov/iad/mig/tests/rt/2004-spring/index.html