Fast Speaker Diarization Using a Specialization Framework for Gaussian Mixture Model Training

Ekaterina Gonina

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-128
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-128.html

December 12, 2011

Fast Speaker Diarization Using a Specialization Framework for Gaussian Mixture Model Training

UNIVERSITY OF CALIFORNIA, BERKELEY

MASTER'S THESIS

Author: Ekaterina Gonina
Advisor: Professor Kurt Keutzer

Kurt Keutzer _____

Armando Fox _____

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 thesis we present a Python-based GMM training specialization framework that abstracts low-level GPU code and instead automatically selects the best parallel implementation of the training algorithm based on the diarization problem size and the processor features and maps the computation onto the parallel platform. Our specialization framework can automatically map the GMM training algorithm onto any CUDA-programmable NVIDIA GPU as well as multi-core CPUs. We then present a full speaker diarization system captured in about 50 lines of Python that uses our specialization framework and achieves 37-166$\times$ faster than real-time performance without significant loss in accuracy. We also investigate a trade-off between diarization accuracy and performance and show that for a 3% gain in Diarization Error Rate (DER), the application performance decreases by 5$\times$. Using our framework allows the scientist to focus on developing the application algorithm while achieving significant additional performance improvements by automatically utilizing parallel hardware.

1 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 2), 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 2).

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 thesis, 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* - a programmer tool that the programmer imports as a Python library and creates Python objects using the framework's class types. When calling specialized meth-

ods on those objects, the framework automatically generates an optimized parallel implementation of the GMM 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 37×-166× *faster than real-time*[1] 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.

This thesis is structured as follows: Section 2 discusses related work. Section 3 describes Gaussian Mixture Models (GMMs) and the Expectation Maximization (EM) algorithm for training GMMs. Section 4 then describes the speaker diarization system. Section 5 describes the parallelization of GMM training on the GPU and different parallel implementation variants. Section 6 discusses the specialization framework implementation and the process for automatically executing the training function on the parallel hardware. Section 7 then shows the resulting speaker diarization implemented in Python using the specialization framework. Section 8 shows performance results and Section 9 concludes.

2 Related Work

2.1 Parallelizing Speech Processing

There is much prior work on accelerating speech recognition applications. In the 1993 Ravishankar et al. [7] implemented parallel speech recognition engines on shared memory multiprocessors (SMP) by statically partitioning data and tasks among threads. Their implementation achieved up to 3.85× speedup using 5 threads. In 1999 Phillips et al. [8] parallelized a speech recognition engine on an SMP and and obtained factors of 3-6 speedup on 4-12 processors respectively. Then in 2006, Ishikawa et al. [9] implemented a Large Vocabulary Continuous Speech Recognition (LVCSR) system on cellphone-oriented platforms achieving real-time speech processing performance. You et al. [10] achieved about 2× faster than real-time performance on a 20,000 word speech recognition task by parallelizing the algorithm on multi-core processors using OpenMP [11]. All of these

[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.

efforts focused on parallelizing the acoustic observation probability computation phase of the inference engine since it took up majority of the execution time, leaving the network traversal phase sequential. Only recent work (including our own) parallelized the entire engine on parallel processors.

Most recently, Graphics Processing Units (GPUs) emerged as programmable highly parallel processors and presented an interesting new target for speech recognition applications. Chong et al. [12] first parallelized the entire recognition engine on a GPU obtaining $9\times$ speedup compared to a sequential version of the algorithm. GPUs have also been used to accelerate acoustic model computations in speech recognition and speaker diarization applications by [13] and [14]. Dixon et al. ([13]) offloaded the observation probability computation in a speech recognition engine to the GPU, while Kumar et al. ([14]) used the GPU to train Gaussian Mixture Models to obtain $164\times$ speedup over a sequential CPU implementation. EhKan et al. [15] implemented a Gaussian Mixture Model-based speaker identification system on FPGAs, achieving $90\times$ speedup over sequential software version.

2.2 SEJITS

Our methodology for developing the specialization framework for accelerating audio content analysis applications using a high-level scripting language is based on the the Selective Embedded Just-In-Time Specialization (SEJITS) mechanism originally described in [16]. In this proposed work we use a specific implementation of SEJITS called Asp [17]. The general concept of providing portable, high performance abstractions was pioneered by autotuning libraries such as FFTW [18], Spiral [19], and OSKI [20]. These libraries specialize individual function calls, but are not embedded into a host language, and their auto-tuning machinery does not generalize to other use cases. In contrast, Asp allows the domain programmer to stay in the host language (Python) and variant selection can occur at runtime using a general set of just-in-time code generation techniques applicable to a wide variety of computational kernels.

SEJITS inherits standard advantages of JIT compilation, such as the ability to tailor generated code for particular argument values and sizes. While conventional JIT compilers such as HotSpot [21] also make runtime decisions about what to specialize, SEJITS does not require any additional mechanisms for dealing with non-specialized code, which is just executed in Python. Even specialized functions can fall back on a Python-only implementation if the specializer is targeted for different hardware than is available at runtime. Other SEJITS-like frameworks include Copperhead by Catanzaro et al. [22], a data-parallel compiler for Python and Theano [23], a CPU/GPU math-compiler for Python. Finally, the Delite framework [24] is a Domain Specific Language (DSL) creation framework and runtime,

developed by Chafi et al.

3 Gaussian Mixture Models and the EM Algorithm

Gaussian Mixture Models (GMMs) are one of the most widely-used parametric probabilistic models for clustering data. Gaussian Mixture Models are defined as a weighted sum of i Gaussian component densities. Each of the ith component is represented by a D-dimensional Gaussian probability density function with means μ_i and a $D \times D$ covariance matrix Σ_i:

$$g(x \mid \mu_i, \Sigma_i) = \frac{1}{(2\pi)^{\frac{D}{2}} \mid \Sigma_i \mid^{\frac{1}{2}}} exp\{-\frac{1}{2}(x-\mu_i)^T \Sigma_i^{-1}(x-\mu_i)\} \tag{1}$$

The probability of observing data x for a Gaussian mixture then is:

$$\begin{aligned} p(x_j \mid \mu_i, \Sigma_i) &= \sum_i \pi_i g(x_j \mid \mu_i, \Sigma_i) \\ &= \sum_i \pi_i \frac{1}{(2\pi)^{\frac{D}{2}} \mid \Sigma_i \mid^{\frac{1}{2}}} exp\{-\frac{1}{2}(x_j-\mu_i)^T \Sigma_i^{-1}(x_j-\mu_i)\} \end{aligned} \tag{2}$$

Given N observed data points (x_js) each a D-dimensional feature vector, we need to learn the parameters μ_i, Σ_i for each component and the weight parameters π_i for combining them into the overall mixture model. A common way to learn these parameters is to use the Expectation-Maximization (EM) algorithm [25].

Given an initial estimate of the parameters, the EM algorithm iterates between two phases: the E-step and the M-step. The E-step computes the expectation of the log-likelihood of the events (i.e. observations) given parameter estimates,

$$p_{i,j} = \frac{\pi_j^k g(x_j \mid \theta)}{\sum_i \pi_i g(x_j \mid \theta)} \tag{3}$$

Where $p_{i,j}$ is the probability of event j belonging to component i and k is the iteration number.

Then the M-step in turn computes the parameter estimates that maximize the expected log-likelihood of the observation data.

$$\pi_i^{k+1} = \frac{\sum_j p_{i,j}}{N} \tag{4}$$

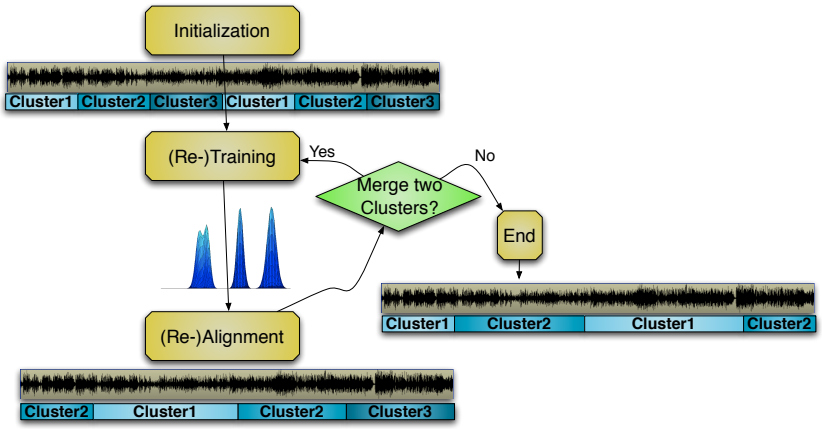

Figure 1: *Illustration of the segmentation and clustering algorithm used for speaker diarization.*

$$\mu_i^{k+1} = \frac{\sum_j x_j p_{i,j}}{\sum_j p_{i,j}} \tag{5}$$

$$\Sigma_i^{k+1} = \frac{\sum_j (p_{i,j}(x_j - \mu_i^{k+1})(x_j - \mu_i^{k+1})^T}{\sum_j p_{i,j}} \tag{6}$$

These two steps repeat until a convergence criterion is reached.

GMM training using the EM algorithm is the central computation in the speaker diarization application. We first describe the diarization algorithm in the next section and then go into detail of parallelizing the GMM EM training algorithm in Section 5.

4 Speaker Diarization

Our diarization system uses GMMs to model different speakers in a meeting. It consists of a pre-processing phase and a segmentation and clustering phase.

4.1 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 [26] and described in [27]. This methodology is an HMM/GMM system trained on broadcast news data that generalizes well to the meeting diarization context.

4.2 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 [28] 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 [26]):

1. Initialization: Train a set of GMMs, one per initial segment, using the EM algorithm.

2. Re-segmentation: Re-segment the audio track using majority vote over the GMMs' likelihoods for $2.5\,s$ duration [29].

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 re-training 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 $1\times$ real-time [30]. Previous analysis of the diarization engine [31] showed two main computational bottlenecks: the training of the GMMs, and choosing the GMM pair to merge during agglomeration.

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. [30]. 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 specialization framework for automatically training GMMs on the GPU, described in the following section.

5 GMM training on the GPU

Training of the Gaussian Mixture Models is performed on the GPU using a specialization framework that automatically executes efficient parallel code on the GPU. The framework adopts the expectation-maximization algorithm described in [32] written in CUDA, a low-level extension to the C programming language [5]. We also have a Cilk+ [33] implementation that targets multi-core CPUs, however in this work we mostly focus on GPU performance.

We use the platform-neutral OpenCL[6] terminology to describe the parallel algorithm, which is implemented in NVIDIA's CUDA language [34]. There are two levels of parallelism: workgroups (CUDA thread blocks) are parallelized across cores on the chip , and a workgroup's work items (CUDA threads) are executed on a single core, potentially utilizing that core's SIMD vector unit. 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 workgroup is assigned a Gaussian component and a subset of the feature frames to compute the log likelihood for. Each work item 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 workgroup is assigned a Gaussian component (total of

M thread blocks) and each work item 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 workgroup is assigned one dimension in one Gaussian component's mean vector ($M \times D$ thread blocks) and each work item 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.

5.1 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 [32] 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, as show in in Section 3:

$$\Sigma_i^{(k+1)} = \frac{\sum_{j=1}^{N}(p_{i,j}(x_j - \mu_i^{(k+1)})(x_j - \mu_i^{(k+1)})^T)}{\sum_{j=1}^{N} p_{i,j}} \tag{7}$$

where $p_{i,j}$ is the probability of frame j belonging to component i and x_j 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 ($D \times D$), 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.

5.2 Code Version Selection

The three possible degrees of freedom in data parallelism suggest different strategies for parallelizing the algorithm on manylane hardware. We found the optimal strategy depends on the problem parameters (N, D, M) as well as certain hardware parameters (e.g. number of cores, SIMD vector width, local memory size). Figure 2 shows the pseudocode for the computation, and we describe them below.

Code Variant 1 (V1)—baseline: The EM on CUDA implementation from Pangborn [32]. Launches $M \times D \times \frac{D}{2}$ workgroups - one for *one* cell for *one* cluster's matrix (shown by the first two for loops in Figure 2(V1)). work items correspond to the loop over events (N). The mean vector is stored in

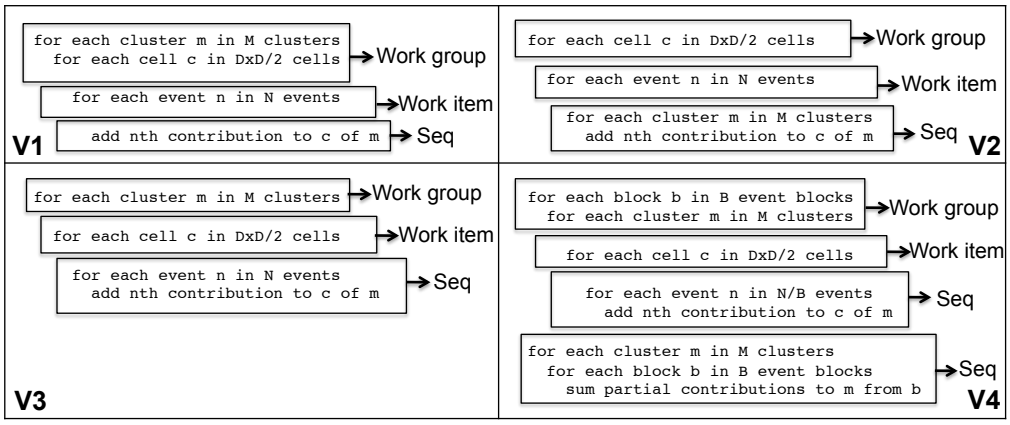

Figure 2: Four code variants for computing the covariance matrix during M step. The computation loops are reordered and assigned to work groups and items as shown above. The "Seq" part of the computation is done sequentially by each work item.

local memory, however only two values are used (corresponding to the row and column of the cell the group is computing).

Code Variant 2 (V2): Modifies V1 by assigning each workgroup to compute *one* cell for *all* clusters. Work groups correspond to the loop over $D \times \frac{D}{2}$ cells in the matrix. work items correspond to events as in V1.

Code Variant 3 (V3): Makes better use of per-core memory by assigning each work group to compute the *entire* covariance matrix for *one* cluster (M). Each work item in the workgroup is responsible for one cell in the covariance matrix ($D \times \frac{D}{2}$ items). Each work item loops through all events sequentially.

Code Variant 4 (V4-BXX): Improves upon V3 by making it more resilient to small M by adding blocking across the N dimension. Launches $M \times B$ workgroups, where B is a blocking factor, i.e. the number of desired event blocks. Each workgroup computes the contribution to its entire covariance matrix for its block of events ($\frac{N}{B}$), followed by a *sum()* reduction over the partial matrices across all event blocks (Figure 2(V4) shows the additional blocking and reduction loops). In this work we use two values of B, 32 and 128.

We test all the variants on NVIDIA GTX285 and GTX480 GPUs using a regular sampling of problem sizes to gain an understanding of the tradeoffs amongst the

variants. The GTX285 has more cores, but the GTX480 has longer SIMD vectors and better atomic primitives. Figure 3 shows some example results. Overall, for the space of problem sizes we examined ($1 \leq D \leq 36, 1 \leq M \leq 128, 10,000 \leq N \leq 150,000$), the best-performing code variant for a given problem instance gave a 32% average performance improvement in covariance matrix calculation time compared to always running the baseline code variant V1. This performance gap increases further with larger problem sizes, e.g. for ($D = 36$, $M = 128$, $N = 500,000$) the difference grows to 75.6%. Figure 3 plots a slice through the 3D space of possible input parameters, allowing the average runtimes of different implementations of the covariance computation to be compared.

V1, V3 and V4 with different B parameters are mutually competitive and show trade-offs in performance when run on various problem sizes and on two GPU architectures. V2 shows consistently poor performance compared to the other code variants. The tradeoff points are different on the two GPUs. While there are general trends leading to separable areas where one variant dominates the others (e.g. V1 is best with small D values), we had difficulty formulating a hierarchy of rules to predetermine the optimal variant because each hardware feature affects each variant's performance differently. This finding suggests that variant selection cannot necessarily be reduced to a compact set of simple rules, even for a specific problem in a restricted domain of problem sizes.

6 Specialization Framework Implementation

We have shown that the EM algorithm can be implemented in multiple ways, and that the best implementation is dependent on both features of the target hardware and the problem size. Our goal is now to encapsulate these variations and dependencies such that the domain application developer does not have to reason about them, and equally importantly, so that the code variant selection (*how* to do the computation) can be kept separate from the actual application (*what* to do), allowing the application to be performance-portable [35].

6.1 Generalizing and Encapsulating the GMM Training Algorithm Using SEJITS

We chose Selective Embedded Just-in-Time Specialization (SEJITS) [16] as the mechanism to accomplish this separation of concerns. In the SEJITS approach, the domain programmer expresses her application entirely in Python using libraries of domain-appropriate abstractions, in this case objects representing GMMs and functions that can be invoked on them, e.g. training via the EM algorithm. How-

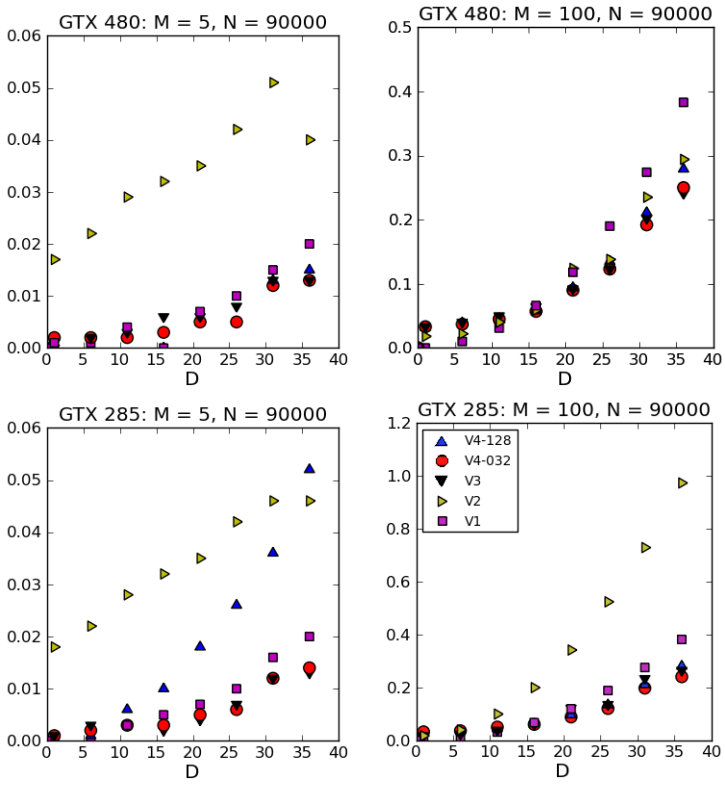

Figure 3: Average runtimes in seconds of covariance computation variants on GTX480 (left) and GTX285 (right) across D, for $N = 90000$ and $M = 5$ or $M = 100$. Different markers signify different variants' runtimes. The best variant depends on both the problem size and the underlying machine. V1 and V2 have a large amount of work item parallelism while V2 has limited work-group parallelism if D is small. V3 can be limited in both work-group and work item parallelism if M and D are small. V1 underutilizes the local memory and requires many streaming reads, whereas V2 and V3 utilize local memory more efficiently and V2 requires a factor of M fewer event data streaming reads than V1. V4 mitigates V3's limited work-group parallelism by blocking over N, but requires a global reduction across B work-groups.

ever, when these speciaized functions are called, they do not execute the computations directly; instead, the Asp framework interposes itself and generates CUDA source code, which is then JIT-compiled, cached, dynamically linked, and executed via the GPU's foreign-function interface, with the results eventually returned to Python. From the Python programmer's view, this experience is like calling a Python library, except that performance is potentially several orders of magnitude faster than calling a pure Python library implementation of the same computation.

6.2 Specialization Mechanism Using Asp

Asp [36] is a particular implementation of the SEJITS approach for Python[2]. Asp contains facilities to automate the process of determining the best variant to use, emit source code corresponding to that variant, compile and call the optimized code, and pass the results of the computation back to the Python interpreter. A *specializer* is a piece of Python code that uses these facilities in combination with code templates expressing the different code variants hand-coded by a human expert, in our case a CUDA programmer who serves a similar role to a library developer. Our specializer implements the EM algorithm described previously, but Asp's facilities are general and could be used for very different specializers; indeed, it has previously been used to specialize stencil/structured-grid codes on multicore CPUs [16, 36].

Once Asp has been imported into a Python program, it transparently and selectively redirects calls of certain functions (in our case, GMM functions in the *scikit* [37] library API) to the appropriate specializer. From the domain expert's point of view, a one-line change to any of her Python programs that use the existing GMM library suffices to get the performance benefits we reported in the previous section.

Our GMM specializer actually emits two kinds of lower-level source code: C++ host code that implements the outer (convergence) loop of EM training, and CUDA GPU code that implements the different parallel kernels used during each EM substep. Code variant selection occurs for one particular kernel of the training process — covariance matrix creation in the M-step. We hand-coded templates for the variants described previously; Asp transforms these templates into syntactically correct source code via the templating library Mako [38], and compiles, caches and links them using the Python extension CodePy [39]. The templates contain placeholders that are filled in by our specializer, such as the number of work items launched, number of event blocks and limits on the number of main loop iterations of the EM algorithm.

[2]Recursive acronym: Asp is SEJITS for Python

In general, the specializer has two jobs: variant selection and intra-variant code optimizations.

6.2.1 Variant selection.

Part of our specializer is the variant selection logic. As explained earlier, the best variant to use depends on properties of the input problem data (the size of M, N, and D). We select a variant by telling Asp to examine the values of the parameters passed to the specialized function, and to treat functions with different parameter values as different functions. The current Asp implementation tries a different variant every time a particular function/problem size is specialized, until all variants have been tested. Thereafter, the specializer remembers what the best runtime was for that particular function/problem size, and always reuses the associated variant's cached binary. This variant selection method is naive; future work will attempt to use performance models or machine learning to make decisions about what variant to use without exhaustively searching all options. However, the important observation for the present work is that the mechanism and policy for variant selection are well-encapsulated and can be replaced without touching the original application source code or the code variant templates themselves.

6.2.2 Intra-variant performance optimizations.

In modern systems, GPUs and CPUs have separate memories, and the programmer is resposible for copying any necessary data and results back and forth between them. Since a common use case is to use the same dataset to train many GMMs (each with a different number of mixtures M), flushing and recopying that data (hundreds of megabytes) would be wasteful. Therefore, our specializer tracks whether the data stored on the GPU by one GMM module is being reused by a second, and only lazily flushes the data and replaces it. This is another example of an optimization that the application programmer need not concern herself with. A similar optimization is the use of the PyUBLAS Python extension [40], which allows numerical data structures to be shared by reference between C++ and Python, eliminating unnecessary value copies.

Note that the logic implementing the above optimizations is itself written in Python. It is not only kept separate from the low-level computation kernels and the high-level application code, but also easier to modify and experiment with since it can exploit the full facilities of the Python language.

7 Speaker Diarization in Python

Using Asp, the implementation of our speaker diarization system is captured in less than 50 lines of Python code and is shown in Figure 4 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 4 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 [41] (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 4 (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.

```
from em import *

def cluster(self, M, D, K, data):

 gmm_list = new_gmm_list(M,D,K)
 per_cluster = N/K
 init = uniform_init(gmm_list, data, per_cluster, N)
 for gmm, data in init:
    gmm.train(data)

 # Perform hierarchical agglomeration
 best_BIC_score = 1.0
 while (best_BIC_score > 0 and len(gmm_list) > 1):

   # Resegment data based on likelihood scoring
   L = gmm_list[0].score(data)
   for gmm in gmm_list[1:]:
     L = np.column_stack((L, gmm.score(data) ))
   most_likely = L.argmax()
   split_data = split_obs_data_L(most_likely, data)

   for gmm, data in split_data:
      gmm.train(data)

   # Score all pairs of GMMs using BIC
   best_merged_gmm = None
   best_BIC_score = 0.0
   m_pair = None

   #find most likely merge candidates using KL
   gmm_pairs = get_top_K_GMMs(gmm_list, 3)

   for pair in gmm_pairs:
       gmm1, d1 = pair[0] #get gmm1 and its data
       gmm2, d2 = pair[1] # get gmm2 and its data
       new_gmm, score =
            compute_BIC(gmm1, gmm2, concat((d1, d2)))
       if score > best_BIC_score:
         best_merged_gmm = new_gmm
         m_pair = (gmm1, gmm2)
         best_BIC_score = score

   # Merge the winning candidate pair
   if best_BIC_score > 0.0:
     merge_gmms(gmm_list, m_pair[0], m_pair[1])
```

Figure 4: *Speaker diarization in Python. Components that are executed on the GPU are highlighted in light-gray*

8 Results

8.1 Test Sets

In order to evaluate the accuracy and speed of the approach described above, we use two popular meeting evaluation subsets: the Augmented Multi-Party Interaction (AMI) corpus [42] and the National Institute of Standards and Technologies: Rich Transcription Spring 2007 Evaluation (NIST RT07) corpus [43].

The AMI corpus is a subset of 12 meetings (5.4 hours) that 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 NIST RT07 corpus consists of 21 meetings from the National Institute of Standards and Technologies Rich Transcription 2007 evaluation and development set. It consists of meetings of various lengths contains meetings with different number of speakers.

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 using the AMI corpus, we use the beamformed far-field and near-field array microphone signals. For the NIST RT07 corpus, we use the single distant microphone signal.

8.2 Diarizer Results

To make the scores compatible with the baseline system, we used the Shout speech activity detection as described in Section 4 and in [26]. We calculated the MFCC features as a preprocessing step. The feature computation and speech activity detection step is not part of the runtime computation.

After processing the files, we scored the output of the diarization using Diarization Error Rate, which is defined by NIST [43]. 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 1 and Table 2 show the performance results for the AMI and NIST RT07

Table 1: *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	33.56 %	100.31×	17.96 %	131.48×
IS1001a	22.67 %	49.62×	26.37 %	86.38×
IS1001b	49.29 %	69.13×	21.52 %	106.73×
IS1001c	48.16 %	64.70×	10.15 %	98.82×
IS1003b	22.51 %	86.69×	24.61 %	151.97×
IS1003d	66.34 %	52.72×	50.49 %	76.79×
IS1006b	45.13 %	66.59×	24.00 %	166.32×
IS1006d	60.53 %	54.05×	52.26 %	81.13×
IS1008a	3.51 %	56.43×	2.6 %	70.48×
IS1008b	4.15 %	95.08×	5.06 %	163.02×
IS1008c	32.61 %	74.88×	12.84 %	130.21×
IS1008d	26.61 %	70.07×	22.70 %	123.48×
Average	34.59 %	70.02×	22.55 %	115.57×

corpus in terms of accuracy and speed. 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 for the AMI corpus in Table 1 and "DER" column in Table 2 shows %DER for the NIST RT07 corpus. The results for both evaluation sets are comparable to state-of-the art DER for the AMI corpus [44] and the NIST RT07 corpus [45].

The "FF ×RT" and "NF ×RT" columns in Table 1 show corresponding performance in terms of speed using our specialization framework on NVIDIA GTX480 GPU for the AMI far-field and near-field setups and column "×RT" in Table 2 shows the corresponding performance in terms of speed for the NIST RT07 corpus. 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, for the AMI corpus, the far-field microphone array meetings take longer to process than near-field (70.02×RT and 115.57×RT respectively) with higher DER on average (34.59% and 22.55% respectively). For the NIST RT07 corpus, the average performance was similar to the AMI far-field microphone array meetings, 78.07×RT with average DER of 21.57%. In general, the real-time performance varies by each meeting from about 37-166×RT, depending on the length of the audio as well as the number of clustering iterations computed before

Table 2: *Diarization Error Rate (DER) of the diarization system and the faster than real-time performance factor (×RT) for the NIST RT07 corpus.*

Meeting ID	DER	×RT
AMI_20041210-1052	12.00 %	72.94×
AMI_20050204-1206	11.83 %	58.03×
CMU_20050228-1615	29.87 %	80.98×
CMU_20050301-1415	14.85 %	48.61×
CMU_20050912-0900	29.03 %	85.97×
CMU_20050914-0900	16.35 %	85.22×
EDI_20050216-1051	35.11 %	118.30×
EDI_20050218-0900	32.13 %	116.23×
ICSI_20000807-1000	18.72 %	77.25×
ICSI_20010208-1430	13.99 %	62.61×
LDC_20011116-1400	9.93 %	43.79×
LDC_20011116-1500	14.49 %	37.85×
NIST_20030623-1409	11.61 %	59.07×
NIST_20030925-1517	23.24 %	47.46×
NIST_20051024-0930	17.14 %	80.85×
NIST_20051102-1323	31.66 %	152.60×
TNO_20041103-1130	28.75 %	88.02×
VT_20050304-1300	35.70 %	83.02×
VT_20050318-1430	20.82 %	58.34×
VT_20050623-1400	22.58 %	97.32×
VT_20051027-1400	23.1 %	84.89×
Average	21.57 %	78.07×

Table 3: *Average Diarization Error Rate (DER) of the diarization system and the average faster than real-time performance factor (×RT) for the AMI and NIST RT07 corpus. Columns "%DER diff" and "Slowdown" show the gain in Diarization Error Rate the loss in diarization performance compared to using KL-divergence fast-match approximation respectively.*

	DER	×RT	%DER diff	×slower
AMI FF	33.61 %	21.09×	0.98 %	3.32×
AMI NF	19.61 %	22.68×	2.94 %	5.09 ×
NIST RT07	17.99 %	16.42×	3.58 %	4.75 ×

convergence.

To better understand the trade-off between speed and accuracy, we compared the performance of the diarization system to a version that does not use KL-divergence fast-match strategy but instead computes the BIC score for all pairs of GMMs per iteration. Table 3 shows the average DER and ×RT factor for the AMI far-field, AMI near-field and the NIST RT07 evaluation sets as well as the gain in the average %DER and the loss in the average speed. In general, to gain about 3% in DER the diarizer performance has to be 5 times slower.

8.3 Specializer Performance Results

Finally, we analyzed the raw performance of our CUDA and Cilk+ implementation of the GMM training algorithm within the specialization framework. Figure 5 shows that the GMM training performance of our specializer beats even the hand-coded CUDA [32] implementation by selecting the best-performing algorithmic variant at runtime based on training problem size [46]. The specializer can emit CUDA and Cilk+ code, making it performance-portable both within and across architecture families with no changes to the speaker diarization application.

Table 6 shows the speaker diarization performance of the original C/Pthreads implementation in terms of ×RT compared to performance of our Python implementation using the specialization framework on multicore CPUs using the Cilk+ and two different GPUs (NVIDIA GTX285 and NVIDIA GTX480) using CUDA. The table shows that the multicore Cilk+ implementation and both CUDA implementations outperform the basic parallel C/Pthreads version, but more importantly, the diarization application automatically sees this performance improvement by using our specialization framework, without needing to rewrite the application code.

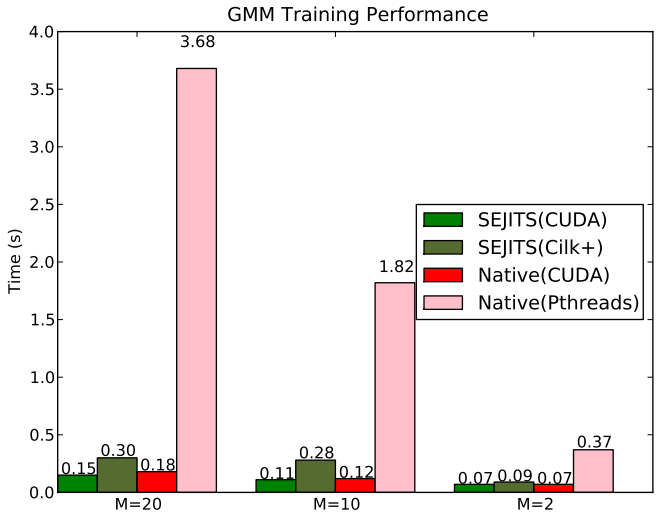

Figure 5: *GMM training performance given number of mixture-model components M, which varies as speaker diarization algorithm converges, using the CUDA backend and a native CUDA version (both on NVIDIA GTX480), and the Cilk+ backend and a C++/pthreads version (both on dual-socket Intel X5680 Westmere3.33GHz).*

Mic Array	**Orig. C++/pthreads**	**Py+Cilk+**	**Py+CUDA**
	Westmere	Westmere	GTX285/GTX480
Near field	$20\times$	$56\times$	$101\times$ / $115\times$
Far field	$11\times$	$32\times$	$68\times$ / $70\times$

Figure 6: *Diarizer application performance as a multiple of real time; "*$100\times$*" means that 1 second of audio can be processed in* $1/100$ *second. The Python application using CUDA and Cilk+ outperforms the native C++/pthreads implementation by a factor of 3-6.*

9 Conclusion

This thesis describes a fast speaker diarization system and the specialization framework for Gaussian Mixture Model (GMM) training that it uses to achieve 37-166× 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 the specialization framework on an NVIDIA graphics processing unit (GPU) and multicore CPUs. 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 as well as implementing other audio content analysis applications using the framework.

10 Acknowledgments

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] M. Ravishankar, "Parallel implementation of fast beam search for speaker-independent continuous speech recognition," 1993.

[8] S. Phillips and A. Rogers, "Parallel speech recognition," *Intl. Journal of Parallel Programming*, vol. 27, no. 4, pp. 257–288, 1999.

[9] S. Ishikawa, K. Yamabana, R. Isotani, and A. Okumura, "Parallel LVCSR algorithm for cellphone-oriented multicore processors," in *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, Toulouse, France, 2006.

[10] 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.

[11] *OpenMP Application Programming Interface*, May 2008, version 3.0. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[12] 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.

[13] 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.

[14] 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.

[15] 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

[16] 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.

[17] 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.

[18] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[19] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005.

[20] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," *Journal of Physics: Conference Series*, vol. 16, no. 1, pp. 521+, 2005. [Online]. Available: http://dx.doi.org/10.1088/1742-6596/16/1/071

[21] M. Paleczny, C. Vick, and C. Click, "The java hotspot(tm) server compiler," in *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine*

Research and Technology Symposium. Berkeley, CA, USA: USENIX Association, 2001, pp. 1–1.

[22] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: compiling an embedded data parallel language," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 47–56. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941562

[23] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: a CPU and GPU math expression compiler," in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, Jun. 2010, oral.

[24] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun, "A domain-specific approach to heterogeneous parallelism," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 35–46. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941561

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

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

[27] 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/

[28] 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.

[29] 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

[30] 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.

[31] 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

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

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

[34] *NVIDIA CUDA Programming Guide*, NVIDIA Corporation, March 2010, version 3.1. [Online]. Available: http://www.nvidia.com/CUDA

[35] B. Alpern, L. Carter, and J. Ferrante, "Space-limited procedures: a methodology for portable high-performance," in *Proceedings of the conference on Programming Models for Massively Parallel Computers*, ser. PMMP '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 10–. [Online]. Available: http://portal.acm.org/citation.cfm?id=525697.826742

[36] "ASP: A SEJITS implementation for python," http://aspsejits.pbworks.com/.

[37] "scikits.learn: machine learning in python," http://scikit-learn.sourceforge.net/index.html.

[38] "Mako templates for python," http://www.makotemplates.org/.

[39] "CodePy: a C/C++ metaprogramming toolkit for python," http://mathema.tician.de/software/codepy.

[40] "PyUBLAS: a seamless glue layer between numpy and boost.ublas," http://mathema.tician.de/software/pyublas.

[41] 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.

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

[43] "National institute of standards and technologies: Rich transcription spring 2007 evaluation." [Online]. Available: http://www.nist.gov/itl/

[44] 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

[45] G. Friedland, O. Vinyals, Y. Huang, and C. Muller, "Prosodic and other long-term features for speaker diarization," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 17, no. 5, pp. 985 –993, july 2009.

[46] 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.