PRODUCTIVE GMM TRAINING WITH SEJITS FOR SPEAKER DIARIZATION

Katya Gonina, Henry Cook, Shoiab Kamil, Gerald Friedland, Armando Fox, David Patterson

ParLab Retreat, June 2, 2011
Components of the Meeting Diarist

- Speaker Diarization
  - "who spoke when"
  - "what was said"
  - "what's relevant to this"
  - Audio Signal

- Speech Recognition
  - "what was said"
  - Relevant Web Scraping

- Speaker Attribution
  - "who said what"

- Indexing, Search, Retrieval
  - "what are the main points"
  - Summarization

- Question Answering
  - higher-level analysis
Speaker Diarization

Audio track:

Segmentation:

Clustering:

Estimate “who spoke when” with no prior knowledge of speakers, #of speakers, words, or language spoken.
Speaker Diarization: Core Algorithm

1. Initialization
2. Start with too many clusters (initialized randomly)
3. Purify clusters by comparing and merging similar clusters
4. Resegment and repeat until no more merging needed
5. (Re-)Training
6. Merge two Clusters?
7. Yes: (Re-)Training
8. No: (Re-)Alignment
9. End
Parallelization of Diarization

- Five versions (so far):
  - Initial code (2006): $0.333 \times$ realtime (i.e., 1h audio = 3h processing)
  - Serially optimized (2008): $1.5 \times$ realtime
  - Parlab retreat summer 2010: Multicore+GPU parallelization: $14.3 \times$ realtime
  - Parlab retreat winter 2011: GPU-only parallelization: $250 \times$ realtime (i.e., 1h audio = 14.4 sec processing)
    - $\Rightarrow$ Offline = online!
  - Parlab retreat summer 2011: SEJITized! [1]

```python
def AHC(self):
    # Get the events, divide them into an initial k clusters and train each GMM on a cluster
    per_cluster = self.N/self.init_num_clusters
    init_training = zip(self.gmm_list, np.vsplit(self.X, range(per_cluster, self.N, per_cluster)))
    for g, x in init_training:
        g.train(x)

    # Perform hierarchical agglomeration based on BIC scores
    best_BIC_score = 1.0
    while (best_BIC_score > 0 and len(self.gmm_list) > 1):
        num_clusters = len(self.gmm_list)

        # Resegment data based on likelihood scoring
        likelihoods = self.gmm_list[0].score(self.X)
        for g in self.gmm_list[1:]:
            likelihoods = np.column_stack((likelihoods, g.score(self.X)))
        most_likely = likelihoods.argmax(axis=1)

        # Across 2.5 secs of observations, vote on which cluster they should be associated with
        split_events = split_events_based_on_votes(most_likely, self.X)
        for g, data in split_events:
            g.train(data)

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

        for gmm1_idx in range(len(self.gmm_list)):
            for gmm2_idx in range(gmm1_idx+1, len(self.gmm_list)):
                g1, d1 = self.gmm_list[gmm1_idx]
                g2, d2 = self.gmm_list[gmm2_idx]
                score = 0.0
                new_gmm, score = compute_distance_BIC(g1, g2, np.concatenate((d1, d2)))
                if score > best_BIC_score:
                    best_merged_gmm = new_gmm
                    merged_tuple = (g1, g2)
                    best_BIC_score = score

        # Merge the winning candidate pair
        if best_BIC_score > 0.0:
            self.gmm_list.remove(merged_tuple[0])
            self.gmm_list.remove(merged_tuple[1])
            self.gmm_list.append(best_merged_gmm)
```

---

**Initialization**

**(Re-)Training**

**(Re-)Alignment**

**Merge two Clusters?**

Yes
def AHC(self):
    # Get initial list of GMMs and train each GMM on a cluster
    for m in range(self.N, self.per_cluster):
        g.train(x)

    # Perform hierarchical agglomeration based on BIC scores
    while (best_BIC_score > 0) and len(self.gmm_list) > 1:
        num_clusters = len(self.gmm_list)
        # Resegment data based on likelihood scoring
        likelihoods = self.gmm_list[0].score(self.X)
        for g in self.gmm_list[1:]:
            likelihoods = np.column_stack((likelihoods, g.score(self.X)))
        most_likely = likelihoods.argmax(axis=1)
        # Across 2.5 secs of observations, vote on which cluster they should be associated with
        for g in g:
            on_votes = np.zeros(self.N)
            on_votes[most_likely] += 1
            if g.train(x):
                # Score all pairs of GMMs using BIC
                best_merged_gmm = None
                best_BIC_score = 0.0
                merged_tuple = None
                for gmm1idx in range(len(self.iter_bic_list) - 1):
                    for gmm2idx in range(gmm1idx+1, len(self.iter_bic_list)):
                        g1, d1 = self.iter_bic_list[gmm1idx]
                        g2, d2 = self.iter_bic_list[gmm2idx]
                        score = 0.0
                        best_BIC = True
                        for g in g:
                            new_gmm, score = g.concatenate((d1, d2))
                            if score > best_BIC:
                                best_BIC = score
                                best_merged_gmm = new_gmm
                                merged_tuple = (g1, g2)
                                best_BIC_score = score
                # Merge the winning candidate pair
                if best_BIC_score > 0.0:
                    self.gmm_list.remove(merged_tuple[0])
                    self.gmm_list.remove(merged_tuple[1])
                    self.gmm_list.append(best_merged_gmm)
                    break

    if len(self.gmm_list) > 1:
        # (Re-)Training
        (Re-)Alignment
        Merge two Clusters?
        Yes
Gaussian Mixture Models & Training
GMM - probabilistic model for clustering (audio) data

- Assumes the distribution of observations follows a set (mixture) of multidimensional Gaussian distributions
- Each Gaussian in the mixture has a mean \((\mu)\) and a covariance \((\Sigma)\) parameters
- Gaussians in the mixture are weighted with weight

Example GMM in two dimensions
(Source: Dan Klein, UCB)
GMM Training using EM Algorithm

- Given a set of observations/events – find the maximum likelihood estimates of the set of Gaussian Mixture parameters \((\mu, \sigma, \pi)\) and classify observations

- Expectation Maximization (EM) Algorithm
  - E step
    - Compute probabilities of events given model parameters
  - M step
    - Compute model parameters given probabilities
      - weights, mean, **covariance matrix**
    - Iterate until convergence

- Covariance matrix – most computationally intensive step
Covariance Matrix Computation

- \( N \) – number of feature vectors, \( \sim 10K-100K \)
- \( D \) – feature vector dimension (19 for speaker diarization), \( \sim 10-100 \)
- \( M \) – number of Gaussian components, \( \sim 1-128 \)
- Matrix is symmetric – only compute the lower \( D \times D/2 \) cells

\[
\bar{R}_k = \frac{1}{N_k} \sum_{n=1}^{N} (y_n - \bar{\mu}_k)(y_n - \bar{\mu}_k)^t p_{x_n|y_n}(k|y_n, \theta^{(i)})
\]
Opportunities for parallelism (independent computations):

- Each component’s covariance matrix (M)
- Each cell in a covariance matrix (DxD/2)
- Each event’s contribution to a cell in a covariance matrix (N)

-> **Multiple code variants** to perform the same computation in different ways
- Two levels of parallelism:
  - Work-groups – parallelized across cores (CUDA threadBlock)
  - Work-groups’ work-items – executed on a single core, utilizing within-core parallelism (CUDA thread)
- Per-core local memory
Code Variants
Code Variants - Example

- Code variant 1:
  - 2D grid of work groups $M \times D \times D/2$
  - Each work group is responsible for computing one cell in the covariance matrix for one component
  - Work item parallelization over events ($N$)

for each component $m$ in $M$ comps
for each cell $c$ in $D \times D/2$ cells

for each event $n$ in $N$ events

add $n$th contribution to $c$ of $m$

V1

Work group

Work item

Seq.

V2

for each cell $c$ in $D \times D/2$ cells

for each event $n$ in $N$ events

for each component $m$ in $M$ comps

add $n$th contribution to $c$ of $m$

Seq.
Covariance Matrix Computation – Code Variants Summary

for each component \( m \) in \( M \) comps
for each cell \( c \) in \( D \times D/2 \) cells
for each event \( n \) in \( N \) events
add \( n \)th contribution to \( c \) of \( m \)

V1
Work group
Work item
Seq
S
Seq
V2
Work group
Work item
Seq
S
Seq
V3
Work group
Work item
Seq
S
Seq
V4
Work group
Work item
Seq
S
Seq

for each block \( b \) in \( B \) event blocks
for each component \( m \) in \( M \) comps
for each cell \( c \) in \( D \times D/2 \) cells
for each event \( n \) in \( N/B \) events
add \( n \)th contribution to \( c \) of \( m \)

for each component \( m \) in \( M \) comps
for each block \( b \) in \( B \) event blocks
sum partial contributions to \( m \) from \( b \)
GTX480

optimal code version names

D = 19
Results – Code Variant Performance

GTX285

optimal code version names

D = 19
Using best-performing code variant gave 32% average improvement in matrix computation time compared to always using original hand-coded variant (D: 1 to 36, M: 1 to 128, N: 10K to 150K).

Performance gap increases with larger problem sizes (75.6% for D=36, M=128, N=500,000).
Specialization
Specialization with ASP

- Given:
  - Problem Dimensions \((N, D, M)\)
  - Platform Parameters (targeting Nvidia GPUs)
    - Core count, shared memory size, SIMD width...

- Automatically select:
  - Optimal code variant
  - Optimal parameters (block size, number of blocks) for that parallelization strategy
SEJITS Framework: Overview

Python on Host

X = Read in data

gmm = GMM()

gmm.train(X)

CUDA sources

Template files

C sources

CUDA on GPU

kernel

kernel

kernel

kernel

C on Host

Train()

for()

launch

launch

launch


SEJITS Framework

- Python code that handles application
  - Manipulates problem data, determines learning targets
- C/CUDA code that runs quickly
  - Allocates GPU memory
  - Performs main EM iterative loop

- Specializer (ASP)
  - Selects appropriate code variant (from history)
  - Pulls in the template for the code variant, parameterizes it and compiles to binary
Speech Diarizer author (PLL) | Specializer author (ELL) | SEJITS team | 3rd party library

Application | Specializer | ASP core | CodePy PyCUDA

g.train() and input data | C/CUDA Train code variants | Python Code Variant Selection | Compiled module

g.train() call | ASP Module | Utilities |
new_gmm_list(M,D)

g.train(x)
Speaker Diarization in Python

Python

def AHC(self):
    # Get the events, divide them into an initial k clusters and train each GMM on a cluster
    per_cluster = self.N/self.init_num_clusters
    init_training = zip(self.gmm_list, np.vsplit(self.X, range(per_cluster, self.N, per_cluster)))
    for g, x in init_training:
        g.train(x)

    # Perform hierarchical agglomeration based on BIC scores
    best_BIC_score = 1.0
    while (best_BIC_score > 0 and len(self.gmm_list) > 1):
        num_clusters = len(self.gmm_list)

        # Regenerate data based on likelihood scoring
        likelihoods = self.gmm_list.score(self.X)
        for g in self.gmm_list:
            most_likely = np.argmax(np.vstack((likelihoods, g.score)) == 0)
            likelihoods[most_likely] = np.argmax(np.vstack((likelihoods, g.score)) == 0)

        # Across 2.5 secs of observations, vote on which
        split_events = split_events_based_on_votes(most_likely)

        for g, data in split_events:
            g.train(data)

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

        for g1idx in range(len(self.gmm_list[0])):
            for g2idx in range(g1idx + 1, len(self.gmm_list[0])):
                g1, d1 = self.gmm_list[g1idx]
                g2, d2 = self.gmm_list[g2idx]

                if score > best_BIC_score:
                    best_merged_gmm = new_gmm
                    merged_tuple = (g1, g2)
                    best_BIC_score = score

        # Merge the winning candidate pair
        if best_BIC_score > 0.0:
            self.gmm_list.remove(merged_tuple[0])
            self.gmm_list.remove(merged_tuple[1])
            self.gmm_list.append(best_merged_gmm)

C

15x LOC reduction
Results – Specializer Overhead

- Python AHC code is within $1.25x$ of pure C/CUDA implementation performance
- C/CUDA AHC (from winter retreat) – 250x realtime
- SEJITized AHC ~ 200x realtime
- Time lost in:
  - Outer loop and GMM creation in Python
  - Data copying overhead from CPU to GPU
  - GMM scoring in Python
We have implemented the Cilk backend for GMM training
ASP selects version based on available hardware
Current implementation ~100x realtime
5-10% C code reused
All specializer infrastructure reused

http://supertech.csail.mit.edu/cilk/
Specializes to two types of platforms (multi-core CPU, Nvidia GPU) to support **portability**
- Exact same application code

**Reuse** of infrastructure:
- Specializer creation and code variant selection mechanism reused

**Maintaining the code** for next generation of hardware
- Task of specializer writer, transparent to the application developer
Conclusion & Future Work

- SEJITized GMM training in Speaker Diarization component of Meeting Diarist
- Specialized covariance matrix computation with code variant selection to two platforms
- Currently a factor of 1.25x slower than pure C/CUDA implementation (200x faster than realtime)

Future work:
- Further specialize train kernel
- SEJITize other components
- Improve code variant selection mechanism
Thank you!

Questions?
Backup Slides
Results – Specializer Overhead in AHC

- Initial invocation – 81% overhead due to compiler invocations
- Future runs using automatically determined optimal code variant achieve 17% performance *improvement* over the original GPU implementation (V1)
## ASP vs Auto-tuning Libraries

<table>
<thead>
<tr>
<th></th>
<th>ATLAS</th>
<th>FFTW, Spiral, OSKI</th>
<th>ASP/GMM</th>
<th>ASP/Stencil</th>
<th>Delite/OptiML</th>
<th>Copperhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autuning of code</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Based on runtime</td>
<td></td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>information</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Based on higher</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>order func</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Using reusable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>framework</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Embedded in HLL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parallel processing is here

“This shift toward increasing parallelism is not a triumphant stride forward based on breakthroughs in novel software and architectures for parallelism; instead, this plunge into parallelism is actually a retreat from even greater challenges that thwart efficient silicon implementation of traditional uniprocessor architectures.”

- The Berkeley View

Intel Processor Clock Speed
Writing Fast Code is Hard

Dense Matrix Multiply (V. Volkov)

Fraction of Arithmetic Peak

Dimension of Matrices

ACML (vendor-provided binary)

an optimized code
(unrolling, explicit vectorization, few levels of blocking)

naïve blocking
Finding Best Implementation is Hard

Naïve implementation

Best performing

900 MHz Itanium 2, Intel C v8: ref=275 Mflop/s

Implementation Based on structure of data

Figure from R. Vuduc
Scientists and domain experts prefer to use high-level languages such as Python or MATLAB.

However, to achieve sufficient performance, computationally-intensive parts of applications must eventually be rewritten in low-level languages.

In addition, parallel platform details and input parameters determine the best-performing parallel implementation.
Implementation Gap

Application developers make design tradeoffs with a limited knowledge of the hardware platform.

Expert parallel programmers have limited knowledge of application design tradeoffs.

End User

Application Developer

Parallel Programming Expert

HW Platform

Application

Platform

SW Infrastructure
Outline

- SEJITS approach
- Gaussian Mixture Model & Applications
- Covariance Matrix Computation & Code Variants
- Specialization
- Results
- Conclusion & Future Work
Key Idea: Generate, compile, and execute high performance parallel code at runtime using code transformation, introspection, variant selection and other features of high-level languages.

Invisibly to the user.
Selective Embedded JIT Specialization (SEJITS)

- Leverage patterns to bridge productivity and efficiency
- PLL (productivity-level language, eg Python) for applications
- "Specializers" generate ELL (efficiency-level language) code targeted to hardware
  - Code generation can happen at runtime
  - Specializers can incorporate autotuning
  - Think: pattern-specific embedded DSLs
- ELL performance with PLL effort
Selective Embedded JIT Specialization (SEJITS)

Productivity app

Interpreted code (.py)

\[ f() \]
\[ @g() \]
\[ @h() \]

SEJITS

Specializer

OS/HW info

HW Info

Compiled code (.c)

cc/ld

Cache

.so

ASP – A SEJITS for Python
Applications of Gaussian Mixture Models

- Applications
  - Can be used to cluster/classify any sequence of observations
  - Speech Recognition – speaker classification, acoustic modeling for speech recognition
  - Computer Vision – image segmentation, hand writing recognition
  - Biology – flow cytometry
  - Data mining – topic classification in web documents
  - Many more...
Application example – Agglomerative Hierarchical Clustering for Speaker Diarization

- Uses GMMs to represent distribution of audio features for speakers in a recorded meeting
- Iteratively trains GMMs using different number of components each time and measuring which number of components best fits the data
- Number of components in the best GMM corresponds to number of speakers in the meeting
Conclusions & Future Work

- ASP framework encapsulates code variant selection mechanisms and handcrafted templates to:
  - Allow domain expert to stay in the high-level language domain and focus on the application
  - Obtain high performance from expert-tuned code
- Example in Gaussian Mixture Model Applications
- Performance benefit of specialization outweighs the overhead of Python and the JIT process
- Expand to:
  - more platforms, applications, patterns
  - other code variant selection mechanisms
GTX480 – Varying D
Results – Version Comparison (Raw CUDA)

GTX285 vs. 480

GTX 480: M = 5, N = 90000

GTX 285: M = 5, N = 90000
SEJITS Framework: Current Implementation

- ASP framework
  - C and CUDA compiling with CodePy (using PyCuda)
  - PyUBLAS to eliminate copies between C and Python
  - Version selection based on previous timings

- Evaluation platforms:
  - GTX480 (Fermi)
    - 14 SM, 32 SIMD, 48K shared mem, 3GB DRAM
  - GTX 285
    - 30 SM, 8 SIMD, 16K shared mem, 1GB DRAM
  - CUDA SDK 3.2
  - NVCC 3.2
Covariance Matrix Computation – Code Variants

for each component \( m \) in \( M \) composites
for each cell \( c \) in \( D \times D/2 \) cells
for each event \( n \) in \( N \) events
\[ V_1 \] add \( n \)th contribution to \( c \) of \( m \)

\[ \rightarrow \text{Work group} \]
\[ \rightarrow \text{Work item} \]
\[ \rightarrow \text{Seq} \]

\[ \Rightarrow \text{Work group} \]
\[ \Rightarrow \text{Work item} \]
\[ \Rightarrow \text{Seq} \]

\[ V_2 \]